Development of GPU-based Strategies for Finite Element Simulation of Elastoplastic Problems

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

Doctor of Philosophy

by

Utpal Kiran
Roll No. 176103028

to the

DEPARTMENT OF MECHANICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI

October 2023

CERTIFICATE

It is certified that the work contained in the thesis entitled "***Development of GPU-based Strategies for Finite Element Simulation of Elastoplastic Problems***", by "*Mr. Utpal Kiran*", has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

Dr. Deepak Sharma
Department of Mechanical Engineering
IIT Guwahati

Dr. Sachin Singh Gautam
Department of Mechanical Engineering
IIT Guwahati

October, 2023

Acknowledgements

The work presented in this thesis has been possible only because of the support of numerous people in various aspects of my PhD study at IIT Guwahati. I take this opportunity to gratefully acknowledge their contributions.

I am grateful to my supervisors, Dr. Deepak Sharma and Dr. Sachin Singh Gautam for their guidance and support to me during this course of work. Their valuable suggestions have really helped me in getting things done. I express my sincere gratitude to my doctoral committee members, Prof. Amaresh Dalal, Dr. Aryabartta Sahu and Dr. Nelson Muthu. Their invaluable feedback and encouragement have been beneficial in improving my research work.

I am also thankful to my labmates and friends at IIT Guwahati, Dr. Saipraneeth Gouravaraju, Dr. Vishal Agrawal, Dr. Subhajit Sanfui, Dr. Shashi Ratnakar, Dr. Raktim Biswas, Sumit, Deepjyoti, Abhimanyu and Pankaj. Their companionship has been memorable and delightful, both within and outside the lab.

I express my deepest gratitude to my family members for their love, support and encouragement towards my work.

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

Utpal Kiran
IIT Guwahati
October, 2023

Abstract

Elastoplasticity is a phenomenon in which materials deform elastically up to a certain load limit and plastically afterward. The elastic deformation is recoverable, but plastic deformation is permanent. During the plastic deformation, the material behavior is non-linear and depends on the loading history. Elastoplastic behavior is commonly observed in materials of practical interest like metals, concrete, soils, rocks, biological tissues, etc., that yield when subjected to loads high enough. The design and optimization of such materials depend strongly on the elastoplastic analysis for the prediction of displacement and stress. However, elastoplastic analysis is computationally expensive and often requires the use of parallel computers in real-world applications like metal forming and crashworthiness. This thesis presents a parallel computing framework for finite element analysis of elastoplastic problems using a massively parallel Graphics Processing Unit (GPU) processor. The proposed framework is developed within incremental-iterative elastoplastic theory based on von Mises criteria.

Considering assembly-based approach, GPU-based parallel algorithms are proposed for all expensive steps in elastoplastic analysis, namely the computation of elemental matrices and their assembly, the computation of stress using the well-known radial-return method and the computation of internal force vectors and their assembly. Since GPUs have limited memory, assembly is done directly into a sparse storage format that can be seamlessly integrated with a GPU-based linear solver. The proposed algorithms are optimized for efficient memory access and fine-grain parallelism and prefer computation over data storage and reuse. In the proposed framework, all the computations are performed on the GPU and expensive data transfers to the CPU are avoided to achieve the best performance.

Furthermore, GPU-accelerated matrix-free solvers are used to obtain even better wall-clock timings for elastoplastic simulation. Matrix-free solvers never assemble large sparse global tangent matrix and perform computations with small dense elemental ma-

trices, reducing the storage requirement and avoiding the use of sparse storage formats. For problems using unstructured mesh, a novel matrix-free strategy is developed that uses only the symmetric part of elemental tangent matrices to compute sparse matrix-vector product (SpMV) by following the element-by-element technique. The proposed strategy achieves better performance than those available in the literature and occupies less storage space. However, the matrix-free solvers, like assembly-based solvers, remain memory bound for unstructured mesh. Matrix-free solvers have a unique advantage over assembly-based approaches for problems that can be solved using voxel-based mesh. Unlike assembly-based solvers, matrix-free solvers are capable of computing SpMV using only one elemental tangent matrix due to the congruency of elements in the voxel-based mesh. In elastoplasticity, the primary challenge in the matrix-free computation of SpMV for voxel-based mesh is the presence of both elastic and plastic states, which introduce branching issues in parallel implementation. To this end, single kernel and improved split kernel strategies are proposed to handle branching issues in GPU implementation efficiently. For GPU implementation, node-based, degree-of-freedom-based and element-by-element matrix-free strategies have been used with proposed modifications.

The performance of the proposed strategies is demonstrated by solving a number of benchmark examples from elastoplasticity. Compared with single-core CPU implementation, speedups of several orders of magnitude are achieved. When compared with existing GPU-based strategies from the literature, the proposed strategies show significant performance gain.

Journal Publications

- Kiran, U., Sharma, D. & Gautam, S. S. (2024), 'Development of GPU-based matrix-free strategies for large-scale elastoplasticity analysis using conjugate gradient solver', *International Journal for Numerical Methods in Engineering* 125(7), e7421.

- Kiran, U., Sharma, D. & Gautam, S. S. (2024), 'An efficient framework for matrix-free SpMV computation on GPU for elastoplastic problems', *Mathematics and Computers in Simulation* 216, 318–346.

- Kiran, U., Sharma, D. & Gautam, S. S. (2023), 'A GPU-based framework for finite element analysis of elastoplastic problems', *Computing* 105, 1673–1696.

- Kiran, U., Gautam, S. S. & Sharma, D. (2020), 'GPU-based matrix-free finite element solver exploiting symmetry of elemental matrices', *Computing* 102(9), 1941–1965.

Conference Publications

- Kiran, U., Gautam, S. S. & Sharma, D. (2022), 'Accelerating finite element assembly on a GPU', International Conference on Future Learning Aspects of Mechanical Engineering (FLAME - 2022), Amity University, Noida, 3–5th August.

- Kiran, U., Sanfui, S., Ratnakar, S. K., Gautam, S. S. & Sharma, D. (2019), 'Comparative analysis of GPU-based solver libraries for a sparse linear system of equations', In *Advances in Computational Methods in Manufacturing: Select Papers from ICCMM 2019*, Springer, Singapore, pp. 889-897.

Other Publication

- Ratnakar, S. K., Kiran, U., & Sharma, D. (2022), 'Acceleration of structural topology optimization using symmetric element-by-element strategy for unstructured meshes on GPU', *Engineering Computations* 39(10), 3354-3375.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Real-world problems are often nonlinear in nature. Nonlinearity makes a problem complex and generally necessitates advanced expertise to comprehend and analyze. Among various sources of nonlinearities, material nonlinearity deals with the nonlinear material response of bodies subjected to various types of loads. Depending on the constitutive model in use, the material nonlinearity can be of various types such as plasticity, creep, viscoplasticity, superelasticity, hyperelasticity, etc. Elastoplasticity (Simo & Hughes 1998, de Souza Neto et al. 2008) is a type of material nonlinearity in which materials first undergo elastic deformation up to a certain load limit and plastic deformation afterward. The elastic deformation is temporary and can be recovered but plastic deformation remains even after the load is removed. The phenomenon of elastoplasticity is commonly observed in materials of practical interest like metals, which are subjected to high loads and pressure during service, often leading to yielding. The elastoplastic behavior is also exhibited by materials that have extensive real-world applications, like soils (Hong et al. 2020), rocks (Maier & Hueckel 1979), concretes (Meschke et al. 1998), composites (Jiang et al. 2002), polymers (Reese & Wriggers 1997) and biological tissues (El Sayed et al. 2008), etc. Consequently, elastoplasticity has a wide range of application areas including crack development and propagation during impact (Gautam & Dixit 2010, Gautam et al. 2011, Yusa & Yoshimura 2014), ductile fracture (Gautam & Dixit 2010, 2012), crashworthiness (Jones 2011, Baroutaji et al. 2017, Cai et al. 2015) and geophysical simulations (Hong et al. 2020, Rayhani & El Naggar 2008), spread over a number of industrial sec-

tors like mechanical, aerospace, geotechnical engineering and biomechanics. Therefore, the study of elastoplasticity is extremely important and essential to accurately describe plastic deformations so that judicious decisions about the strength of engineering materials can be made. In addition, elastoplasticity is also important for industrial processes like metal forming where plastic deformation is intentionally introduced to manufacture engineering products of various shapes. In many cases, either large amount of time and capital is involved in conducting experiments or scope for experiments is limited as in the case of natural occurrences like earthquakes. Elastoplastic analysis plays a very important role in all such cases.

The development of computational methods for elastoplastic analysis remains a topic of great interest due to the need to describe the behavior of engineering materials in numerical contexts precisely. A significant quantum of work has been done in the past to develop efficient computational methods for numerical solution of elastoplastic problems (Simo & Hughes 1998, Dunne & Petrinic 2005, de Souza Neto et al. 2008). A typical numerical approach for simulation of elastoplastic problems is based on finite element method (FEM) (Simo & Hughes 1998, Dunne & Petrinic 2005, Kim 2015). FEM is the most popular numerical technique for finding approximate solutions to problems governed by partial differential equations. An effective finite element analysis of elastoplastic problems is crucial as it allows accurate estimation of load bearing capabilities of solids and structures under diverse loading scenarios. However, high computational cost in FEM is a well-known issue and often leads to high simulation timings. Further, there is a growing need to perform realistic simulations that demand large-scale three-dimensional (3D) models. The FEM computing time for large-scale models involving millions or billions of DOFs can be quite costly and may take a considerable amount of time. In such cases, the processing time of a large-scale 3D elastoplastic simulation may be too long to be useful. The large computational time in FEM can be reduced or controlled by employing a large number of compute resources in the form of parallel computing. There are several works in literature that demonstrate great advantage in FEM simulation with modern parallel computing systems, see for example Adams et al. (2004), Bhardwaj et al. (2002) and Yusa et al. (2019). However, with ever growing demand for high fidelity simulation that can be performed in real time, the development of efficient parallel algorithms for FEM simulation remains an active field of research.

In recent times, parallel computing has become a common practice in the field of scientific computing. This is largely due to saturating performance of single core computing. The exponential growth of clock frequency of CPUs has come to an abrupt end in the last decade, due to increasing cost of power delivery and dissipation, referred to as power wall (Asanovic et al. 2006). In addition, the CPU performance has also been affected by memory wall issue (Brodtkorb et al. 2013), preventing any significant progress in computing capability. However, the performance of modern CPUs continue to evolve in a different form. Now, instead of getting faster cores, CPUs are built with more number of cores. This requires rewriting and reinventing old algorithms to generate sufficient parallelism to get benefited from multiple cores. To this end, parallel computing has become an essential component in numerical simulations. Nowadays, modern computing hardware like Graphics Processing Units (GPUs) have become a popular tool for parallel computing across various disciplines. A GPU is a massively threaded processor architecture consisting of thousands of computing cores achieving computational throughput several folds higher than a conventional CPU. The popularity of GPU can be understood by the fact that eight of top ten supercomputers in June 2023 Top 500 list (https://www.top500.org) use GPUs. The primary reasons for wide acceptability of GPUs are massive parallelism, high memory bandwidth, high performance-to-cost ratio and continuous improvement in programmability. In addition, GPUs are answer to modern day challenges of increasing energy demands in parallel computing as they deliver more performance-per-watt than a CPU (Enos et al. 2010). Originally developed for gaming applications, modern advancements in GPU technology have led to its applicability in a wide range of sectors like graphics rendering, machine learning and artificial intelligence (AI), high performance computing (HPC), etc. In the field of scientific computation, GPUs have evolved to take a prominent place as a co-processor or accelerator to speed up compute intensive parts of scientific codes. The popular numerical methods like FEM (Georgescu et al. 2013, Kiran et al. 2018, Sanfui & Sharma 2020, 2021), computational fluid dynamics (CFD) (Aissa et al. 2017, Xie et al. 2020), molecular dynamics (MD) (Phillips et al. 2020), etc., have already seen great speedups due to GPU acceleration. However, GPU computing has a much broader impact on scientific computing than just being a hardware for parallel computing. Algorithms, once thought to be less appealing due to high computational requirements are now being pursued. The modern GPU has led to a revival in researchers' approach towards computing and unlocked new

possibilities. In this work, efforts are made to harness computational power of GPU for elastoplastic simulation. Readers are referred to Appendix A for more discussion on GPU architecture.

Elastoplasticity is a nonlinear phenomenon, and therefore requires an incremental-iterative procedure to obtain numerical solutions. In the standard procedure, the applied load is divided into multiple increments, and in every increment, linearized form of the governing equation is solved numerically by FEM. Each incremental solution is obtained by solving the linearized equation iteratively so that the true equilibrium configuration is established. Thus, we see that elastoplasticity involves repeated use of FEM to obtain a numerical solution. Since FEM is computationally expensive, its repeated use in elastoplasticity leads to prohibitively high simulation timings. FEM consists of three expensive steps, these are computation of elemental tangent matrices, assembly of global tangent matrix and solution of linear system of equations. Among these, the solution of linear system of equation is often the most time consuming step. As a result, it has attracted significant research for GPU acceleration (Li & Saad 2013, Filippone et al. 2017, Anzt et al. 2022). However, the repeated use of FEM in elastoplasticity demands acceleration of the complete pipeline. Therefore, apart from the solution of system of equations, the computation of elemental matrices and their assembly into a global tangent matrix constitute important stages in elastoplastic analysis. The deformation in the plastic region is characterized by nonlinear relationship between stress and strain, where determination of stress depends on history of deformation and stress-strain relation is often written in the rate form. The stress is determined by integration of constitutive relations using a suitable integration scheme. Since stress and strain are evaluated at each integration point, the associated computational cost can be prohibitively high. Therefore, in order to provide end-to-end acceleration of elastoplastic simulation on GPU, computation of stress along with all steps in FEM must be considered. Despite the existence of numerous works aimed at achieving parallel implementation of elastoplastic simulation on traditional parallel computing systems (Meyer & Michael 1997, Ding et al. 2008, Markopoulos et al. 2015, Khalevitsky, Burmasheva & Konovalov 2016, Yusa et al. 2018, Sefidgar et al. 2021), very few works can be found in literature that use GPU acceleration. A handful of works that demonstrate the usage of GPU in elastoplasticity can be found in (Khalevitsky, Burmasheva, Konovalov & Partin 2016, He et al. 2017, Prabhune & Suresh 2020).

These works show significant performance improvement in specific steps of elastoplastic analysis. However, none of the works propose acceleration of complete pipeline of elastoplastic analysis on GPU. In this work, we develop a framework to port all steps of elastoplastic analysis on GPU so that overall simulation time can be minimized.

Matrix-free iterative solvers with GPU acceleration provide additional opportunity to reduce simulation timings of elastoplastic problems. In matrix-free solvers, the computation of sparse matrix vector multiplication (SpMV) with the global tangent matrix is replaced by matrix-free computation of SpMV that works directly with the constituent elemental tangent matrices. This implies that the global tangent matrix is never assembled and the usage of performance detrimental sparse storage formats in an iterative solver is completely avoided. The elemental matrices are small, dense and provide better memory access than sparse formats. Typically, a matrix-free SpMV requires more number of arithmetic operations than assembly-based SpMV. This makes matrix-free strategies less suitable for single core computing. However, matrix-free solvers with GPU acceleration have been found to achieve excellent performance, as evident by recent works in fields like elasticity (Cai et al. 2013, Martínez-Frutos & Herrero-Pérez 2015, Pikle et al. 2018), fluid mechanics (Fehn et al. 2019) and topology optimization (Ratnakar et al. 2021, Sanfui & Sharma 2023). The matrix-free SpMV has a distinct advantage over the assembly-based SpMV for voxel-based mesh. For problems with simple geometry in 3D, a voxel-based mesh is a kind of structured mesh that consists of linear cubic elements having the same size and orientation, generating the elemental tangent matrices of the same value in FEM. This property allows matrix-free SpMV computation for linear elastic problems with only one elemental tangent matrix, dramatically reducing the memory requirement to a minimum. On the other hand, the assembly-based implementation still needs to construct the global tangent matrix and therefore can not take full advantage of the voxel-based mesh. In the literature, matrix-free solvers with GPU acceleration have been effectively used to reduce the execution timings of FEM-based engineering simulations. However, there is not much in the literature except (Prabhune & Suresh 2020) that discusses the application of matrix-free solvers to elastoplasticity. In this work, we develop matrix-free solvers for problems utilizing both unstructured and voxel-based meshes. Next, a brief survey of relevant literature is presented.

1.1 Literature review

At an early stage, researchers recognized the computational power of a GPU and decided to use it for their applications. They mainly used graphics APIs like OpenGL to make use of GPU resources. The early implementation of FEM can be found in Bolz et al. (2003), Wu & Heng (2004), Rodríguez-Navarro & Susín Sánchez (2006) and Göddeke et al. (2007). These implementations targeted very specific problems and used relatively simple expression for operator evaluation. However, these implementations successfully achieved descent speedup, drawing attention of more number of researchers. The increasing research interest to harness the computational resource of GPU was further fueled by the introduction of Compute Unified Device Architecture (CUDA) by NVIDIA in 2006. CUDA made the GPU programming much easier by providing C/C++ language-based parallel programming environment. Now, the non-graphics application could be easily ported to a GPU without making use of graphics API, see Appendix B for more details. In the following sections, we discuss the GPU implementation strategies for FEM and elastoplasticity.

1.1.1 Assembly-based FEM

The computation of elemental tangent matrix is done by numerical integration based on Gauss quadrature rule. The general procedure for numerical integration consists of a loop over integration points and double-loop over all the shape functions. The loop over the integration points and shape functions can be interchanged, implying different implementations and resource usage. When using outer-loop over integration points, least amount of computation is required, as geometrical parameters (Jacobian, calculation of shape function derivatives in physical coordinate etc.) can be calculated once for one integration point and used for all the shape functions. Here, memory requirement is high as intermediate values must be saved for further computation or summation. When using outer loop over shape functions, memory requirement is not an issue, but calculation for each integration point needs to be done redundantly for all the shape functions. In either of the cases, we see that numerical integration kernel in FEM requires significant amount of computational resources as well as memory access and storage space. The earliest work on GPU acceleration of numerical integration is found in Macioł et al.

(2010), where a single thread block is assigned to compute an elemental tangent matrix for electromagnetic application. It uses outer loop over integration points and inner loops over shape functions, achieving 3–19× speedup for higher order prismatic elements on NVIDIA GeForce 8800 GTX. The numerical integration for quadrilateral element with a curved geometry is implemented by Płaszewski et al. (2010) using OpenCL (Khronos Group 2020). In Dziekonski et al. (2012), numerical integration strategy for higher-order tetrahedral element is presented that uses outer loop over integration points. This strategy uses 81 thread blocks to perform computation corresponding to 81 Gauss points, achieving 30.02× speedup on NVIDIA Tesla C2075 GPU. The GPU implementation of numerical integration portable across different GPU architectures is found in Banaś et al. (2014), Banaś et al. (2016). Here, the authors propose several optimizations for effective GPU utilization using OpenCL for low order as well as higher order elements.

In assembly, entries of elemental tangent matrices are accumulated into a global tangent matrix on the basis of mesh connectivity. This is a memory bound operation as it involves large amount of data movement to-and-fro the global memory with assembly requiring little amount of arithmetic operations. The GPU hardware, which has large memory bandwidth and can launch great amount of parallel threads, aligns well with computational requirement of the assembly step. However, parallel assembly on GPU suffers from data read-write conflict referred to as the problem of race condition (NVIDIA Corporation 2022). It is a situation where two or more threads attempt to write values to the same location in memory, simultaneously. This kind of operation leads to undefined outcomes and must be avoided. In assembly, data race condition occurs due to the presence of common nodes among multiple elements. Coloring of finite element mesh (Farhat & Crivelli 1989) is a popular and robust method to handle race condition in FEM.

The initial investigation of GPU acceleration of finite element assembly was reported by Filipovic et al. (2009) for linear tetrahedral element, achieving 15× speedup on NVIDIA GeForce 280 GTX GPU. This work remained preliminary and targeted relatively simple problem. A detailed analysis of assembly strategies applicable to general class of problems was proposed by Cecka et al. (2011). This paper investigates many approaches using different types of GPU memory for storing elemental data and different kernel designs for assembly. It recommends using two different approaches for low

and high order elements. For low-order elements, elemental data is stored in the shared memory of GPU, while assembly is done by associating threads with non-zero entry of the global matrix. This approach is found better than coloring method. For higher-order elements, the best performing method uses single thread per element to calculate elemental data and store into the global memory of GPU. Assembly to system of equations is done by parallel reduction. In this study, optimization of numerical integration is not discussed and therefore elemental subroutine is treated as black box. Fu et al. (2014) presented parallelization of assembly by making efficient use of shared memory. The proposed strategy makes disjoint sets of nodes by mesh partition and assigns elements in such a way that each element belongs to one patch only. Assembly is done for all the elements belonging to a patch in the shared memory and final data is written in coalesced manner to the global assembled matrix in the global memory. In Zayer et al. (2017), mesh information is captured through sparse matrix representation and assembly of elemental matrices is converted into sparse matrix multiplication operation. This innovative approach achieves speedup of approximately 1.5× over the best strategy found in (Cecka et al. 2011).

There are other strategies that combine the computation of numerical integration and assembly into one computational step. Zhang & Shen (2013) proposed a coloring-based strategy using single GPU kernel that achieved speedup of 7× and 10× for quadrilateral and hexahedral (8-noded) elements, respectively on NVIDIA GeForce GT430. Another coloring-based strategy is found in (Ohshima et al. 2012), where two types of strategies are presented for 8-noded 3D elements. First uses single kernel for the computation, whereas the second strategy makes use of three kernels. The later implementation is found to be better, reducing the assembly time from 2.44 seconds to 0.65 seconds for 512,000 elements on NVIDIA Tesla C2050. Reguly & Giles (2013) presented a strategy which can scale to higher-order finite elements. Here, single thread calculates the entries of elemental matrix by using outer-loop over the shape functions and inner-loop over the Gauss points and stores them in the global memory. The assembly to the global memory is explored using different sparse storage formats. Kiran et al. (2018) presented a single kernel strategy for computation of elemental matrices and their assembly, allocating a warp to 8-noded hexahedral elements. In Sanfui & Sharma (2020), the assembly strategy including computation of elemental matrices for unstructured mesh is split into three

stages. Another recent work targeting assembly and computation of elemental matrix is found in Sanfui & Sharma (2021) where GPU implementation is divided into symbolic and numeric components.

1.1.2 Matrix-free FEM

The earliest work on matrix-free FEM solver is found in Hughes et al. (1983), where element-by-element solution scheme was used for a problem of nonlinear mechanics. Later, more investigations were performed in matrix-free implementation of FEM by using element-by-element (*EbE*) (Carey & Jiang 1986, Carey et al. 1988, Barragy & Carey 1988) and row-by-row (DOF-by-DOF) (van Rietbergen et al. 1996) strategies to establish its accuracy, convergence and suitability to parallel processing. These works demonstrated the advantages of matrix-free FEM over assembly-based strategy for small memory parallel machines to perform large-scale simulation. In recent times, as General Purpose computing on a GPU (GPGPU) gained popularity among scientific community, matrix-free solvers gained a fresh attention. An early GPU implementation of matrix-free FEM solver is found in Markall et al. (2010), Kiss, Badics, Gyimóthy & Pávó (2012) and Kiss, Gyimóthy, Badics & Pávó (2012), where *EbE* strategy is used. In Markall et al. (2010), the computation is split into multiple kernels, whereas Kiss, Badics, Gyimóthy & Pávó (2012), Kiss, Gyimóthy, Badics & Pávó (2012) perform computations by allocating single GPU thread to one finite element. In Cai et al. (2013), authors demonstrate node-by-node (*NbN*) thread allocation for GPU implementation of matrix-free FEM solver, achieving huge speedup. A comparison of matrix-free FEM with conventional assembly-based implementation is presented in Markall et al. (2013), Reguly & Giles (2013), which demonstrate superior performance by matrix-free FEM solver performing computation directly with local or elemental matrices. A DOF-by-DOF (*DbD*) thread assignment for matrix-free computation is found in Martínez-Frutos & Herrero-Pérez (2015), where proposed strategy loops through each element connected to a DOF to perform computation and gather the multiplication results. The *DbD* and *NbN* strategies provide fine grain parallelism compared to *EbE* strategy, but suffer from the problem of load unbalance. This was addressed in Martínez-Frutos et al. (2015) by two kernel strategy that keeps the advantage of *DbD* strategy intact and still achieves uniform work distribution. In

Pikle et al. (2018), GPU-based matrix-free FEM solver is used to solve an elasticity problem with unstructured mesh. The authors use DOF-based thread assignment for each element and propose several optimizations for improved occupancy and memory access. In another work by Kiran et al. (2020), *EbE* strategy for unstructured mesh problems is proposed which uses only symmetric part of elemental matrices for computation, reducing the storage requirement as well as global memory access. A detailed analysis of thread allocation strategy in *EbE* matrix-free solver is presented in Ratnakar et al. (2021), where allocation of eight threads per element is recommended for linear hexahedral element. In a recent work by Lopes et al. (2022), matrix-free FEM is used to solve up to 500 million DOFs numerical homogenization problem with preconditioned CG solver based on *EbE* and *NbN* strategies. Apart from the aforementioned works, there are numerous other research works that show significant computational performance by matrix-free solvers with GPUs in wide range of areas like elasticity (Martínez-Frutos et al. 2015, Pikle et al. 2018), topology optimization (Ratnakar et al. 2022, Sanfui & Sharma 2023), earthquake simulation (Yamaguchi et al. 2019), fluid mechanics (Knaus 2022), etc.

Few variations for GPU-based matrix-free FEM solver exist in terms of number of elemental tangent matrices that it uses. For certain applications, voxel-based mesh consisting of congruent elements is considered sufficient. In such case, only one elemental tangent matrix can be used to perform the matrix-free computation. The GPU implementation of matrix-free SpMV using single elemental tangent matrix provides the fastest kernel as dependency on input data is very small, see Cai et al. (2013) and Martínez-Frutos & Herrero-Pérez (2015). However, the matrix-free FEM solver needs to work with individual elemental tangent matrices for applications that require unstructured mesh. If tangent matrices are stored in GPU memory, the implementation that uses individual tangent matrices becomes memory bound and the performance is largely determined by the bandwidth of the GPU device, see Martínez-Frutos et al. (2015), Pikle et al. (2018) and Kiran et al. (2020). In some implementations, elemental matrices are not stored but computed on-the-fly when required. As shown in Reguly & Giles (2013), this strategy performs poorly due to high arithmetic overhead.

1.1.3 Elastoplasticity

The parallel implementation of elastoplasticity on conventional parallel machines have been studied by many researchers (Meyer & Michael 1997, Ding et al. 2008, Markopoulos et al. 2015, Khalevitsky, Burmasheva & Konovalov 2016, Yusa et al. 2018, Sefidgar et al. 2021). However, there are only a handful of works in the literature that discuss usage of GPU hardware for elastoplasticity. One of the early works to demonstrate acceleration of elastoplastic simulation with GPU is done by Irina et al. (2011). Here, the authors perform computation of elemental matrices and solution of system of equations on GPU for moderately sized mechanical problems. A comparative analysis of iterative solvers over a GPU cluster is presented by Khalevitsky, Burmasheva, Konovalov & Partin (2016), where performance achieved over six GPUs was found equivalent to hundred CPUs. He et al. (2017) proposed a GPU-based strategy for elastoplastic reanalysis, performing computation of elemental matrices on GPU, whereas the assembly of the global tangent matrix is done on the CPU. The computation of elemental matrix is divided into online and offline parts. The offline part remains constant over Newton iterations, whereas the online part is updated in each iteration on GPU by assigning single thread to each element. A significant amount of speedup with respect to the CPU is reported, even when an expensive CPU-GPU data transfer step is involved. Recently, GPU implementation of elastoplastic simulation for perfectly plastic material is presented by Prabhune & Suresh (2020) for additive manufacturing applications with matrix-free approach. The authors proposed *NbN* strategy for the matrix-free SpMV computation in deflated conjugate gradient (CG) solver. For efficient treatment of elastic and plastic elements, computation is split into two kernels. One kernel is assigned to do computation using the elastic part of an elemental tangent matrix for all the elements. Another kernel performs computation using the plastic part of elemental tangent matrices for only those elements that lie in the plastic zone, where plastic elements are identified using conditional statements. The authors report a significant speedup of up to 26× as compared with single CPU implementation of preconditioned CG solver. In another recent work by Wyser et al. (2021), significant performance gain is reported for GPU-based elastoplastic analysis using material point method. The authors demonstrate 200× speedup over single-core CPU for 3D slumping mechanics problem.

1.2 Motivation

Elastoplasticity is a nonlinear problem which requires an incremental-iterative solution procedure that involves repeated use of many computationally expensive steps. Along with FEM discretization, other important steps are computation of stress, internal force vectors, estimation of convergence parameters and variables update. Since FEM is computationally expensive, early works to use GPU in elastoplasticity revolves around it (Irina et al. 2011, Khalevitsky, Burmasheva, Konovalov & Partin 2016, He et al. 2017). However, these works target either computation of elemental matrix or solution of system of linear equations, but not all steps of elastoplasticity. In addition, these works are rudimentary as compared with the literature that discuss GPU acceleration of FEM in context of other application areas like elasticity. As discussed in Section 1.1, each step in FEM has been thoroughly discussed by numerous researchers, providing many recommendations for effective GPU implementation. However, the existing strategies are not directly applicable in elastoplasticity. The parallel implementation of elastoplastic analysis suffers from a well-known branching issue due to the presence of both elastic and plastic states. The computation of elemental tangent matrix becomes dependent on the state of a Gauss point and require different implementation for each state. The computation of stress in elastoplasticity requires integration of constitutive relation using a suitable scheme like radial-return method. However, no previous work exists that discusses GPU acceleration for the computation of stress. Since, an iterative procedure is followed for the solution, an efficient assembly procedure is also required. Moreover, to achieve maximum benefit from a GPU acceleration, end-to-end simulation must be performed on the GPU. Therefore, this thesis aims to develop a framework that explores all possibilities of minimizing execution timings by using GPU in every step of elastoplastic simulation.

Based on the review of the literature, it can be observed that a GPU-accelerated matrix-free FEM solver is effective for large-scale simulations and has great potential for reducing the elastoplastic simulation time. However, the application of matrix-free solver in elastoplasticity is less explored and therefore, a more comprehensive study is required. The *NbN* strategy has been shown to achieve great speedups with respect to a CPU-based CG solver (Prabhune & Suresh 2020). However, this work can be easily extended to use the *DbD* strategy that provides fine-grain parallelism. The *EbE* strategy is known

for balanced workload distribution and efficient memory access for unstructured mesh problems and can be used for elastoplasticity. In this work, both *DbD* and *EbE* strategies are used to implement matrix-free SpMV computation on GPU for elastoplasticity.

Due to congruency of elements in voxel-based mesh, only one elemental tangent matrix is used in matrix-free solvers (Cai et al. 2013, Martínez-Frutos & Herrero-Pérez 2015, Lopes et al. 2022). However, elastoplasticity contains both the elastic and plastic states. Unlike elastic zone where constitutive matrix remains fixed, each Gauss point in plastic zone has a unique plastic constitutive matrix that depends on the state of internal variables. The presence of a unique constitutive matrix leads to a unique elemental tangent matrix for elements in the plastic zone and must be taken into account for matrix-free computation. The elastic and plastic zones in the body evolve during the iterative solution process, and therefore matrix-free SpMV implementation has to work with different sets of elastic and plastic elements in each iterative step. This issue is handled by the use of two kernels (Prabhune & Suresh 2020), where identification of elements in the plastic zone is done by a conditional statement. As we know conditional statements introduce branching in parallel implementation, the current work aims to develop alternate strategy that avoids branching in matrix-free SpMV computation. The developed strategy works effectively with *NbN*, *DbD* and *EbE* strategies for GPU implementation, and uses only one elemental tangent matrix for the elastic zone and individual elemental matrices for elemental in the plastic zone. The development of an effective matrix-free strategy for SpMV computation is expected to enhance the performance of a CG linear solver, and subsequently reduce the overall simulation timings of elastoplasticity.

1.3 Objectives of the thesis

The objectives of the thesis can be summarized as follows:

- Development of a GPU-based parallel framework for elastoplastic analysis. It entails development of parallel strategies for all expensive steps in elastoplastic analysis, like computation of elemental matrices and their assembly, computation of internal force vectors and their assembly, and computation of stress using radial-return method.

- Development of matrix-free CG iterative solver for elastoplasticity applicable to unstructured meshes discretized using eight noded hexahedral elements.
- Development of matrix-free CG iterative solver for elastoplasticity applicable to voxel-based structured meshes.
- Performance evaluation of the proposed strategies over large-scale benchmark problems of elastoplasticity. The performance results are evaluated over problems involving associated flow rule and isotropic linear strain hardening with von Mises yield criteria.

1.4 Organization of the thesis

The rest of the thesis is organized as follows.

- **Chapter 2** provides the background for theory of elastoplasticity and its numerical solution using FEM. The numerical procedure for the estimation of stresses in finite element framework is also discussed. This is followed by description of sequential CPU implementation.
- **Chapter 3** discusses the proposed GPU framework for assembly-based elastoplastic solver. The performance of the proposed framework is discussed for both structured as well as unstructured meshes by applying it over various 3D benchmark examples.
- **Chapter 4** presents the matrix-free SpMV strategies for elastoplastic problems with unstructured mesh. A novel strategy is proposed that exploits the symmetry of elemental tangent matrices to reduce memory access and improve the performance.
- **Chapter 5** discusses two novel matrix-free SpMV strategies for elastoplastic problems, namely single kernel strategy and improved split kernel strategy for efficient usage of voxel-based structured mesh. Both the proposed strategies are implemented on GPU a using node-based and element-based parallelization strategies and performance is compared for various benchmark examples.

- **Chapter 6** concludes the thesis with a note on scope for future research directions.

Chapter 2

Theory of Elastoplasticity and Finite Element Method

In the present chapter, the background theory and numerical implementation aspects of elastoplasticity are provided (Simo & Hughes 1998, de Souza Neto et al. 2008). First, the governing equation is presented. Then, the background theory of plasticity is briefly discussed, followed by the derivation of the weak form using the principle of virtual work. Linearization of the weak form is discussed next, followed by the finite element formulation. A scheme to integrate the rate form of the constitutive relation is described in detail with the expression for the tangent modulus. The chapter concludes with a detailed discussion on the computer implementation aspect of the developed formulation.

2.1 Governing Equation

Consider a deformable body $\mathcal{B}$ as shown in Figure 2.1 with surface boundary Γ. The body is subjected to external body force per unit volume $\boldsymbol{b}$ in the bulk and surface traction $\bar{\boldsymbol{t}}$ on the boundary Γ_f. The motion of the body $\mathcal{B}$ is prescribed at the boundary Γ_u. The equation that governs the deformation of a continuous body for quasi-static

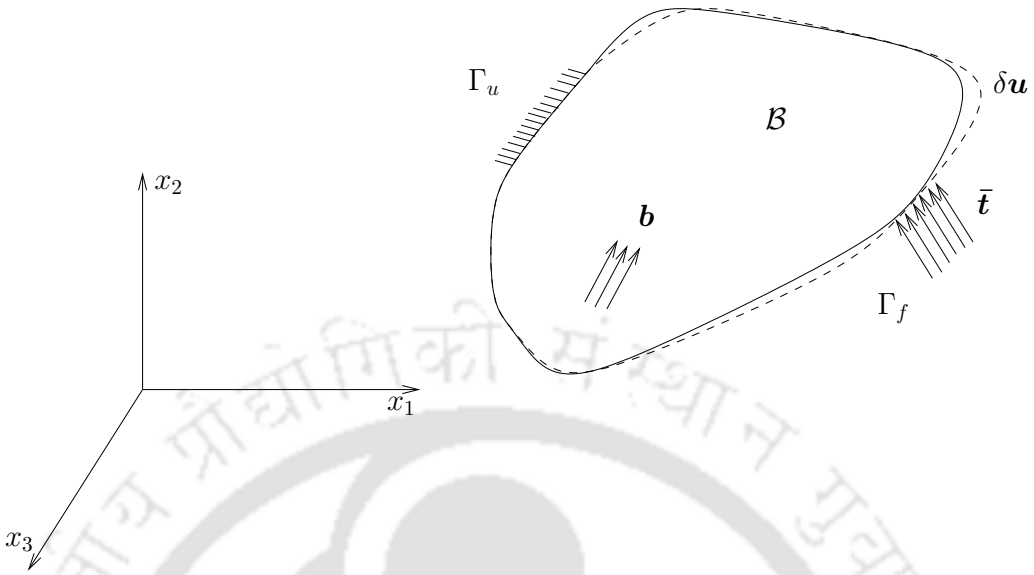


Figure 2.1: A deformable body under the action of external traction ($\bar{\boldsymbol{t}}$) and body forces ($\boldsymbol{b}$).

process along with boundary conditions is given by

$$\begin{aligned} \boldsymbol{\nabla} \cdot \boldsymbol{\sigma} + \boldsymbol{b} &= \boldsymbol{0}, \quad \forall \boldsymbol{x} \in \mathcal{B}, \\ \boldsymbol{u} &= \boldsymbol{u}_o, \quad \forall \boldsymbol{x} \in \Gamma_u, \\ \boldsymbol{t} &= \bar{\boldsymbol{t}}, \quad \forall \boldsymbol{x} \in \Gamma_f, \end{aligned} \tag{2.1}$$

where $\boldsymbol{\sigma}$ is the Cauchy stress, $\boldsymbol{u}_o$ and $\bar{\boldsymbol{t}}$ are the specified displacement and traction on the displacement boundary Γ_u and the traction boundary Γ_f, respectively. The governing equation given by (Eq. 2.1) is the local statement of the balance of linear momentum equation that establishes force equilibrium at every point inside the problem domain. The equilibrium solution must satisfy the above equation locally inside the domain as well as the boundary conditions.

2.2 Elastoplasticity

Under the assumption of small deformation and absence of large rotation, $\delta \boldsymbol{d} \approx \delta \boldsymbol{\varepsilon}$ (Tadmor et al. 2012). The small strain tensor can be additively decomposed into elastic

and plastic parts as (Simo & Hughes 1998, Kim 2015, de Souza Neto et al. 2008)

$$\boldsymbol{\varepsilon} = \boldsymbol{\varepsilon}^e + \boldsymbol{\varepsilon}^p. \tag{2.2}$$

The Cauchy stress tensor for an isotropic material can then be written using Eq. (2.2) as

$$\boldsymbol{\sigma} = \mathcal{C} : \boldsymbol{\varepsilon}^e = \mathcal{C} : (\boldsymbol{\varepsilon} - \boldsymbol{\varepsilon}^p) \tag{2.3}$$

where $\mathcal{C}$ is the fourth order elastic constitutive tensor that is given as

$$\mathcal{C}_{ijkl} = \lambda \delta_{ij}\delta_{kl} + \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}). \tag{2.4}$$

Here, λ and μ are the Lamé's constant and δ_{ij} is the Kronecker delta, which takes the value 1 if $i = j$ and 0 otherwise. It is well-known that a material yields when the magnitude of stress exceeds the yield stress. The identification of yield state requires a yield function which can differentiate between the elastic and the plastic states. In the present work, von Mises yield function is employed. The von Mises yield function f is given as

$$f(\boldsymbol{\sigma}, \kappa) = \sqrt{3J_2} - \kappa(\varepsilon^p_{\text{eq}}), \tag{2.5}$$

where J_2 is the second invariant of deviatoric stress $\boldsymbol{\sigma}'$ and κ is a hardening function that depends on the equivalent plastic strain $\varepsilon^p_{\text{eq}}$. The expression for J_2 and $\varepsilon^p_{\text{eq}}$ are given as

$$J_2 = \frac{1}{2}\boldsymbol{\sigma}' : \boldsymbol{\sigma}', \quad \varepsilon^p_{\text{eq}} = \sqrt{\left(\frac{2}{3}\boldsymbol{\varepsilon}^p : \boldsymbol{\varepsilon}^p\right)}. \tag{2.6}$$

The deviatoric part of the Cauchy stress tensor $\boldsymbol{\sigma}'$ is given by

$$\boldsymbol{\sigma}' = \boldsymbol{\sigma} - \frac{1}{3}\text{tr}(\boldsymbol{\sigma})\boldsymbol{I}. \tag{2.7}$$

Here, $\boldsymbol{I}$ is the second order identity tensor and $\text{tr}(\cdot)$ is the trace operator. The computation of stress in Eq. (2.3) requires the determination of unknown plastic strain tensor. In this work, the incremental plastic strain $d\boldsymbol{\varepsilon}^p$ is obtained using associated flow rule,

which is given by normality condition of plasticity as

$$d\boldsymbol{\varepsilon}^p = d\lambda \frac{df}{d\boldsymbol{\sigma}}, \tag{2.8}$$

where $d\lambda$ is the plastic multiplier. The direction of increment in plastic strain is determined by $\dfrac{df}{d\boldsymbol{\sigma}}$ and magnitude is controlled by the plastic multiplier.

The elastoplastic deformation is characterized by the variation in the yield stress of a material with the plastic flow. We consider isotropic strain hardening model with yield stress varying linearly with the equivalent plastic strain,

$$\kappa(\varepsilon^p_{\text{eq}}) = \sigma_y + H\varepsilon^p_{\text{eq}}, \tag{2.9}$$

where σ_y is the initial yield stress and H is the hardening modulus.

The elastoplastic model is governed by a *consistency condition* which states that loading stress must lie on the yield surface during plastic deformation. This constraint, along with loading and unloading condition, is implemented in the form of Karush-Kuhn-Tucker (KKT) condition, which must be satisfied for all the material states (Simo & Hughes 1998),

$$\lambda \geq 0 \quad f \leq 0, \quad \lambda f = 0. \tag{2.10}$$

2.3 Principle of Virtual Work

Generally, a weak form of the differential equation is required to establish the finite element equation. The principle of virtual work is used to obtain the weak form of Eq. (2.1) by taking $\delta\boldsymbol{u}$ as a virtual displacement field in the body (see Figure 2.1). The virtual displacement is taken arbitrarily, except that it is kinematically admissible, i.e., it satisfies the essential boundary condition (Eq. $(2.1)_2$).

The external virtual work expression can be written as

$$\delta\mathcal{W}_{\text{ext}} = \int_{\mathcal{B}} \boldsymbol{b} \cdot \delta\boldsymbol{u} dv + \int_{\Gamma_f} \boldsymbol{t} \cdot \delta\boldsymbol{u} ds. \tag{2.11}$$

The internal virtual work is obtained by integrating work done by the internal forces over the entire volume, given by

$$\delta \mathcal{W}_{\text{int}} = \int_{\mathcal{B}} \boldsymbol{\sigma} : \delta \boldsymbol{d} dv, \tag{2.12}$$

where $\delta \boldsymbol{d}$ is the virtual rate of deformation tensor due to the virtual displacement. However, for small strain plasticity problem with negligible inertia, the internal virtual work can be written as (Dunne & Petrinic 2005)

$$\delta \mathcal{W}_{\text{int}} = \int_{\mathcal{B}} \boldsymbol{\sigma} : \delta \boldsymbol{\varepsilon} dv, \tag{2.13}$$

where $\delta \boldsymbol{\varepsilon}$ is the virtual small strain tensor. The principle of virtual work states that the internal virtual work should be equal to the external virtual work for a body to be under equilibrium. This is written as

$$\delta \mathcal{W}_{\text{int}} = \delta \mathcal{W}_{\text{ext}}. \tag{2.14}$$

Using Eqs. (2.11) and (2.13), we can write

$$\int_{\mathcal{B}} \boldsymbol{\sigma} : \delta \boldsymbol{\varepsilon} dv = \int_{\mathcal{B}} \boldsymbol{b} \cdot \delta \boldsymbol{u} dv + \int_{\Gamma_f} \boldsymbol{t} \cdot \delta \boldsymbol{u} ds. \tag{2.15}$$

The principle of virtual work makes no assumption regarding the stress-strain relation and the type of deformation. Hence, it is equally applicable to linear and nonlinear problems in solid mechanics.

2.4 Linearization

If a body under external load is not in equilibrium, the principle of virtual work does not hold. A residual R is defined as the difference between the internal virtual work and external virtual work, i.e.,

$$R(\boldsymbol{u}, \delta \boldsymbol{u}) = \int_{\mathcal{B}} \boldsymbol{\sigma} : \delta \boldsymbol{\varepsilon} dv - \int_{\mathcal{B}} \boldsymbol{b} \cdot \delta \boldsymbol{u} dv - \int_{\Gamma} \boldsymbol{t} \cdot \delta \boldsymbol{u} ds. \tag{2.16}$$

The equilibrium solution would mean that the residual $R(\boldsymbol{u}, \delta\boldsymbol{u})$ becomes zero. The residual expression contains various kind of nonlinearities in form of the stress-strain and strain-displacement relation. Therefore, the problem of finding equilibrium solution can be formulated as a root finding problem of a nonlinear equation.

Generally, an iterative solution method like Newton-Raphson (NR) is adopted for the solution of nonlinear equation. The Newton-Raphson solution method starts with an initial guess $\boldsymbol{u}^0$ of the solution and finds an increment $\Delta\boldsymbol{u}$ such that the new approximate solution is close to the actual solution. For a k^{th} step of NR iteration we can write

$$\boldsymbol{u}^{k+1} = \boldsymbol{u}^k + \Delta\boldsymbol{u}, \quad k = 0, 1, 2, ... \tag{2.17}$$

where $\boldsymbol{u}^{k+1}$ is the new solution. This process is repeated until a sufficiently accurate solution is obtained. However, the computation of an increment requires linearization of the nonlinear equation.

Considering infinitesimal deformation, the residual expression (2.16) contains only stress-strain nonlinearity. The Cauchy stress is linearized using Taylor series expansion as

$$\boldsymbol{\sigma}(\boldsymbol{u} + \Delta\boldsymbol{u}, \delta\boldsymbol{u}) = \boldsymbol{\sigma}(\boldsymbol{u}) + \frac{d\boldsymbol{\sigma}}{d\boldsymbol{\varepsilon}} : \Delta\boldsymbol{\varepsilon}. \tag{2.18}$$

If the external load is considered independent of the displacement, the linearized residual expression is obtained as

$$\hat{R}(\boldsymbol{u} + \Delta\boldsymbol{u}, \delta\boldsymbol{u}) = \int_{\mathcal{B}} \boldsymbol{\sigma} : \delta\boldsymbol{\varepsilon} dv + \int_{\mathcal{B}} \delta\boldsymbol{\varepsilon} : \frac{d\boldsymbol{\sigma}}{d\boldsymbol{\varepsilon}} : \Delta\boldsymbol{\varepsilon} dv - \int_{\mathcal{B}} \boldsymbol{b} \cdot \delta\boldsymbol{u} dv - \int_{\Gamma} \boldsymbol{t} \cdot \delta\boldsymbol{u} ds. \tag{2.19}$$

Here, $\dfrac{d\boldsymbol{\sigma}}{d\boldsymbol{\varepsilon}}$ is known as the tangent modulus operator and is determined by elastoplastic theory which is discussed in the subsequent section (Dunne & Petrinic 2005).

2.5 Finite element formulation

In order to obtain an approximate numerical solution, the problem domain is discretized into a number of polygons or polyhedra, called as the element. The displacement at a

point $\boldsymbol{x}$ over a typical element domain $\mathcal{B}^e$ is approximated by Lagrange shape functions as (Reddy 2006)

$$\boldsymbol{u}(\boldsymbol{x}) \approx \sum_{i=1}^{n} \mathbf{u}_i^e N_i(\boldsymbol{x}), \tag{2.20}$$

where $\mathbf{u}_i^e$ is the nodal displacement vector of node i, n is the total number of nodes in an element, and N_i is the shape function associated with the node i. This can be rewritten in the following matrix form as

$$\boldsymbol{u} = \begin{Bmatrix} u_x \\ u_y \\ u_z \end{Bmatrix} \approx \mathbf{N}\mathbf{u}^e, \tag{2.21}$$

where u_x, u_y and u_z are the displacements in $x,\ y$ and z coordinates, and $\mathbf{N}$ is the shape function matrix given by

$$\mathbf{N} = \begin{bmatrix} N_1 & 0 & 0 & N_2 & 0 & 0 & \ldots & N_n & 0 & 0 \\ 0 & N_1 & 0 & 0 & N_2 & 0 & 0 & \ldots & N_n & 0 \\ 0 & 0 & N_1 & 0 & 0 & N_2 & 0 & 0 & \ldots & N_n \end{bmatrix}. \tag{2.22}$$

Here, N_i $(i = 1, \ldots, 8)$ are the shape functions associated with 8 nodes of linear hexahedral element which is considered in the present work. The elemental displacement vector $\mathbf{u}^e$ is given by

$$\mathbf{u}^e = \begin{Bmatrix} \mathrm{u}_x^1 & \mathrm{u}_y^1 & \mathrm{u}_z^1 & \mathrm{u}_x^2 & \mathrm{u}_y^2 & \mathrm{u}_z^2 & \ldots & \mathrm{u}_x^n & \mathrm{u}_y^n & \mathrm{u}_z^n \end{Bmatrix}^T, \tag{2.23}$$

where $\mathrm{u}_x^i, \mathrm{u}_y^i, \mathrm{u}_z^i (i = 1, \ldots, 8)$ are the x, y and z displacements of i^{th} local node. The small strain tensor is then obtained as

$$\boldsymbol{\varepsilon} = \frac{1}{2}\left[\nabla \boldsymbol{u} + (\nabla \boldsymbol{u})^T\right] \approx \sum_{i=1}^{n} \mathbf{B}_i \mathbf{u}_i, \tag{2.24}$$

where $\mathbf{B}_i$ is the strain-displacement matrix given by

$$\mathbf{B}_i = \begin{bmatrix} N_{i,x} & 0 & 0 \\ 0 & N_{i,y} & 0 \\ 0 & 0 & N_{i,z} \\ 0 & N_{i,z} & N_{i,y} \\ N_{i,z} & 0 & N_{i,x} \\ N_{i,y} & N_{i,x} & 0 \end{bmatrix}. \tag{2.25}$$

Here, $(\cdot)_{,x}, (\cdot)_{,y}$, and $(\cdot)_{,z}$ denote the derivatives with respect to x, y and z coordinates. The substitution of (2.21) into the linearized weak form (2.19) gives finite element equation for an element in matrix form as

$$\int_{\mathcal{B}^e} (\delta\mathbf{u})^T (\mathbf{B}^T \mathbf{D} \mathbf{B}) \mathbf{u}^e dv = \int_{\mathcal{B}^e} (\delta\mathbf{u})^T \mathbf{N}^T \boldsymbol{b} dv + \int_{\Gamma_f^e} (\delta\mathbf{u})^T \mathbf{N}^T \boldsymbol{t} ds - \int_{\mathcal{B}^e} (\delta\mathbf{u})^T \mathbf{B}^T \boldsymbol{\sigma} dv \tag{2.26}$$

The assembly of all elemental equations gives the finite element equilibrium equation in the global form as

$$^k\mathbf{K}\Delta\mathbf{u} = {}^k\mathbf{f}_{\text{ext}} - {}^k\mathbf{f}_{\text{int}}, \tag{2.27}$$

where $\mathbf{K} = \mathcal{A}(\mathbf{K}^e)$, $\mathbf{f}_{\text{ext}} = \mathcal{A}(\mathbf{f}^e_{\text{ext}})$, $\mathbf{f}_{\text{int}} = \mathcal{A}(\mathbf{f}^e_{\text{int}})$ and $\Delta\mathbf{u}$ is the increment in nodal displacement at iteration k. Here, $\mathcal{A}$ is the assembly operator. The expressions for the elemental quantities are given by

$$\mathbf{K}^e = \int_{\mathcal{B}^e} \mathbf{B}^T \mathbf{D} \mathbf{B} dv, \tag{2.28}$$

$$\mathbf{f}^e_{\text{ext}} = \int_{\mathcal{B}^e} \mathbf{N}^T \boldsymbol{b} dv + \int_{\Gamma_f^e} \mathbf{N}^T \boldsymbol{t} ds, \tag{2.29}$$

$$\mathbf{f}^e_{\text{int}} = \int_{\mathcal{B}^e} \mathbf{B}^T \boldsymbol{\sigma} dv. \tag{2.30}$$

The elastoplastic tangent modulus $\mathbf{D} = \dfrac{d\boldsymbol{\sigma}}{d\boldsymbol{\varepsilon}}$ is determined depending on the state of the Gauss point as

$$\mathbf{D} = \begin{cases} \mathbf{D}_e, & \text{if elastic,} \\ \mathbf{D}_p, & \text{if plastic.} \end{cases} \tag{2.31}$$

The state of stress is determined by unconditionally stable backward Euler method using elastic corrector and plastic predictor scheme (Dunne & Petrinic 2005).

2.6 Radial-return method

The constitutive relations for elastoplastic problems are given in the rate form and must be integrated over the time or load increments. The current work uses implicit backward Euler method for integration of the constitutive relation (Dunne & Petrinic 2005). The backward Euler method is quiet popular due to its simplicity and unconditional stability. In this method, determination of stress is done in two steps. The first step, also known as elastic predictor step, computes a trial stress by considering the strain increment as purely elastic. If the trial stress lies outside of the yield surface, it is brought back onto the yield surface in the second step, known as plastic corrector step. The elastic predictor-plastic corrector method of stress determination is also referred to as radial-return method.

Let the state at load step t_n is given as: ${}^{n}\boldsymbol{\sigma}$, $\Delta\boldsymbol{\varepsilon}$, ${}^{n}\boldsymbol{\varepsilon}^{e}$, ${}^{n}\boldsymbol{\varepsilon}^{p}$ ${}^{n}\varepsilon^{p}_{\text{eq}}$. The objective is to determine ${}^{n+1}\boldsymbol{\sigma}$, ${}^{n+1}\boldsymbol{\varepsilon}^{p}$ and ${}^{n+1}\varepsilon^{p}_{\text{eq}}$.

Elastic predictor

Assuming the strain increment to be entirely elastic, a trial stress is computed as

$$\boldsymbol{\sigma}^{\text{tr}} = \mathbf{D}_e({}^{n}\boldsymbol{\varepsilon}^{e} + \Delta\boldsymbol{\varepsilon}). \tag{2.32}$$

According to Eq. (2.5), the yield function becomes,

$$f(\boldsymbol{\sigma}^{\text{tr}}, {}^{n}\kappa) = \sqrt{3J_2} - \kappa({}^{n}\varepsilon^{p}_{\text{eq}}), \tag{2.33}$$

where,

$$J_2 = \frac{1}{2}\boldsymbol{\sigma}^{\mathrm{tr}'} : \boldsymbol{\sigma}^{\mathrm{tr}'}, \quad {}^{n}\kappa = \sigma_y + H\,{}^{n}\varepsilon^{p}_{\mathrm{eq}}. \tag{2.34}$$

If $f({}^{n+1}\boldsymbol{\sigma}^{\mathrm{tr}}, {}^{n}\kappa) \leq 0$, the state is considered elastic and quantities are updated as

$${}^{n+1}\boldsymbol{\sigma} = {}^{n+1}\boldsymbol{\sigma}^{\mathrm{tr}}, \; {}^{n+1}\varepsilon^{p}_{\mathrm{eq}} = {}^{n}\varepsilon^{p}_{\mathrm{eq}}, \; {}^{n+1}\boldsymbol{\varepsilon}^{p} = {}^{n}\boldsymbol{\varepsilon}^{p}. \tag{2.35}$$

if $f({}^{n+1}\boldsymbol{\sigma}^{\mathrm{tr}}, {}^{n}\kappa) > 0$, the state is considered plastic and the next step is followed.

Plastic corrector

Using Newton-Raphson iteration[1], following equations are solved to obtain the equivalent plastic strain,

$$\begin{aligned} {}^{i}\kappa &= {}^{n}\kappa + H\,{}^{i}(\Delta\varepsilon^{p}_{\mathrm{eq}}), \\ \delta\varepsilon^{p}_{\mathrm{eq}} &= \frac{\sqrt{3J_2} - 3G({}^{i}(\Delta\varepsilon^{p}_{\mathrm{eq}})) - {}^{i}\kappa}{3G + H}, \\ {}^{i+1}(\Delta\varepsilon^{p}_{\mathrm{eq}}) &= {}^{i}(\Delta\varepsilon^{p}_{\mathrm{eq}}) + \delta\varepsilon^{p}_{\mathrm{eq}}, \end{aligned} \tag{2.36}$$

where G is the shear modulus. Using $\Delta\varepsilon^{p}_{\mathrm{eq}}$, the plastic strain increment is calculated as

$$\Delta\boldsymbol{\varepsilon}^{p} = \frac{3}{2}\frac{\Delta\varepsilon^{p}_{\mathrm{eq}}}{\sqrt{3J_2}}\boldsymbol{\sigma}^{\mathrm{tr}'}. \tag{2.37}$$

The final values are taken as

$$\begin{aligned} {}^{n+1}\boldsymbol{\sigma} &= \boldsymbol{\sigma}^{\mathrm{tr}} - 2G\Delta\boldsymbol{\varepsilon}^{p}, \\ {}^{n+1}\boldsymbol{\varepsilon}^{p} &= {}^{n}\boldsymbol{\varepsilon}^{p} + \Delta\boldsymbol{\varepsilon}^{p}, \\ {}^{n+1}\varepsilon^{p}_{\mathrm{eq}} &= {}^{n}\varepsilon^{p}_{\mathrm{eq}} + \Delta\varepsilon^{p}_{\mathrm{eq}}. \end{aligned} \tag{2.38}$$

For material points in the plastic state, elastoplastic tangent modulus is used which is consistent with the integration scheme. For more details, refer to Dunne & Petrinic (2005).

[1]This Newton-Raphson iteration is different from the one mentioned in Section 2.4.

2.7 Computation of tangent modulus

The slope of stress-strain curve at any given stress or strain value is called as tangent modulus. In elastoplasticity, the tangent modulus is equal to the Young's modulus in the elastic zone. Beyond the yield point, the tangent modulus varies with strain and needs to be computed at the updated state. As shown in Eq. (2.31), the tangent modulus takes the form of elastic constitutive matrix for the elastic state and plastic constitutive matrix for the plastic state. In this work, the plastic constitutive matrix is derived from the implicit backward Euler method and therefore referred to as the consistent tangent modulus. The plastic constitutive matrix is given by (Dunne & Petrinic 2005),

$$\mathbf{D}_p = \frac{2GQ}{\sigma_{\text{eq}}^{\text{tr}}\sigma_{\text{eq}}^{\text{tr}}}\boldsymbol{\sigma}^{\text{tr}'} \otimes \boldsymbol{\sigma}^{\text{tr}'} + 2GR\,\mathbb{I} + \left(K - \frac{2}{3}GR\right)\boldsymbol{I} \otimes \boldsymbol{I}, \tag{2.39}$$

where $\sigma_{\text{eq}}^{\text{tr}}$ is the equivalent trial stress, G is the shear modulus, K is the bulk modulus and $\mathbb{I}$ is the fourth order identity tensor. The quantities Q and R are given as,

$$Q = \frac{3}{2}\left(\frac{1}{1+(3G/H)} - \frac{\sigma_{\text{eq}}}{\sigma_{\text{eq}}^{\text{tr}}}\right), \qquad R = \frac{\sigma_{\text{eq}}}{\sigma_{\text{eq}}^{\text{tr}}}. \tag{2.40}$$

2.8 Computer implementation

The finite element formulation of elastoplastic problems presented in previous sections is implemented in an incremental-iterative manner, since stress at any instant may not depend only on the instantaneous strain but also on the loading history. Algorithm 1 outlines the key steps in the numerical analysis of elastoplastic problems that are used in the current work for the development of CPU and GPU implementations. The external load is divided into a number of smaller increments and each incremental load is applied in successive steps. Line 2 of Algorithm 1 shows the outer most loop over the load steps. The external load vector $\mathbf{f}_{\text{ext}}$ is updated in line 3 and the computation of unbalanced force vector $\mathbf{f}_{\text{ub}}$ is done in line 4. The minimization of residual unbalance force vector is done by the Newton-Raphson method (line 5), which iteratively minimizes the residual till the tolerance becomes smaller than a given limiting value (see line 21). The Newton-Raphson iteration runs as long as convergence is not satisfied or user defined maximum

number of iterations (Max_NR_itr) is not reached.

Lines 6–13 of Algorithm 1 denote the computation of elemental stiffness matrices and their assembly into global tangent matrix for each iteration of the Newton-Raphson method. On CPU, the computation of elemental tangent matrices and their assembly is implemented in a conventional way by taking a loop over all the elements of the mesh. However, the tangent matrices are computed only for elements exhibiting plastic behavior. If an element is found undergoing plastic deformation, elemental tangent matrix is computed using the plastic constitutive matrix $\mathbf{D}_p$. Otherwise, the precomputed tangent matrix is read from array **K_local**. Separating the computation for elastic and plastic elements is important because only a small percentage of elements undergoes plastic deformation if the size of plastic zone is small. In case of problems having large plastic zone, this strategy can still help in the early stages of the solution procedure. The elemental matrices are assembled into a global tangent matrix using mesh connectivity information (see line 12). Line 15 represents the solution of linear system of equations obtained as a result of the discretization of governing equation. Using the global tangent matrix $\mathbf{K}$ as the coefficient matrix and unbalanced force vector $\mathbf{f}_{\text{ub}}$, the displacement increment $\Delta\mathbf{u}$ can be obtained using any suitable linear solver. The computed displacement is used to determine the stress state of a material point in the body using return mapping algorithm (line 17). Once the stress is determined, the computation of internal forces is done as shown in line 18. The unbalanced force is updated by subtracting the internal force vector from external force vector (line 19).

In terms of computational expense, the steps that determine performance of an elastoplastic analysis are: solution of linear system of equation, computation of elemental matrices and their assembly in a global matrix, computation of stress and internal force vectors. The computational details of all steps except solution of linear system of equation is provided in following sections. The discussion about solution of linear system is delayed to subsequent chapters.

2.8.1 Computation of elemental matrices

The amount of computation in elemental matrices depends upon the type and order of the element used in the mesh. The current work discusses the computational imple-

Algorithm 1 Computer implementation of elastoplastic analysis.

Input: Coordinates, Connectivity, Boundary conditions, Material parameters
Output: Displacements, Stress

Compute all local matrices $\mathbf{K}^e$ and store them in **K_local**.
for load steps = 1 to $\mathcal{L}$ **do** ▷ Loop over all load steps
  Update external force vector: $\mathbf{f}_{\text{ext}}$
  Compute unbalanced force: $\mathbf{f}_{\text{ub}} = \mathbf{f}_{\text{ext}} - \mathbf{f}_{\text{int}}$
  for *itr*=1 to *Max_NR_itr* **do** ▷ Newton-Raphson (NR) iteration
    for $e = 1$ to $\mathcal{E}$ **do** ▷ Loop over all elements
      if $estate(e) == true$ **then** ▷ If an element is in plastic state
        KE ← Compute local matrix $\mathbf{K}^e$ using $\mathbf{D}_p$
      else
        KE ← Read from **K_local**
      end if
      Assemble **KE** into global tangent matrix **K**
    end for
    Apply boundary conditions
    Solve: $\mathbf{K}\Delta\mathbf{u} = \mathbf{f}_{\text{ub}}$
    Update: $\Delta\mathbf{u}_{\text{inc}} += \Delta\mathbf{u}$ ▷ Incremental displacement
    Compute stress: Radial-return algorithm ▷ Refer to Algorithm 3
    Compute Internal force: $\mathbf{f}_{\text{int}}$
    Compute unbalanced force: $\mathbf{f}_{\text{ub}}$
    Compute current tolerance: *ctol*
    if *ctol* $< \xi_{NR}$ **then** ▷ Check for convergence
      Update variables
      Break the NR loop
    end if
  end for
end for

mentation for linear hexahedral element having 8 nodes with three DOFs per node. The integral form of the elemental tangent matrix (Eq. 2.28) is evaluated using a suitable numerical integration technique like the Gauss quadrature (Bathe 1996). Using the Gauss quadrature rule, the expression for elemental matrices can be obtained as

$$\mathbf{K}^e = \int_{\mathcal{B}^e} \mathbf{B}^T\mathbf{D}\mathbf{B}dv \approx \sum_{q=1}^{n_q} (\mathbf{B}^T\mathbf{D}\mathbf{B}\ \det\mathbf{J})\Big|_{(\xi_q,\zeta_q,\eta_q)} w_q \tag{2.41}$$

where $\mathbf{J}$ is the Jacobian for the transformation of geometry, (ξ_q, ζ_q, η_q) is the coordinate for the q^{th} Gauss point and w_q is its weight. The expression in parenthesis of Eq. (2.41) is evaluated for each Gauss point and summed over to get the elemental matrix. However,

in order to perform multiplication the operands $\mathbf{B}, \mathbf{D}$ and $\mathbf{J}$ must be constructed for each gauss points. The shape function derivatives and nodal coordinates are used to compute the Jacobian matrix. The inverse of the Jacobian matrix is used to compute shape function derivatives in the physical coordinate system, which constitutes the matrix $\mathbf{B}$. Once the matrix $\mathbf{B}$ is constructed, $\mathbf{K}^e$ can be evaluated by performing the matrix multiplication.

For 8-noded hexahedral element, the size of different components in numerical integration along with input and output data size is given in Table 2.1. Here, coordinates

Table 2.1: Variables size in numerical integration

Type	Variables	Size
Input Variables	Coordinates	3×8
	Shape function derivatives	3×8
Integration Variables	Jacobian	3×3
	Jacobian inverse	3×3
	Jacobian determinant	1
	Shape function derivative	3×8
	$\mathbf{B}$ matrix	6×24
	$\mathbf{D}$ matrix	6×6
Output Variables	$\mathbf{K}^e$	24×24

remain the same for all Gauss points whereas the shape function derivative in natural coordinates varies. The integration variables and output variables are unique to each Gauss point and must be computed independently. The values of output variable $\mathbf{K}^e$ from each Gauss point should be summed before assembling into the global tangent matrix. It can be observed that the computation of elemental tangent matrices requires considerable amount of computing resource and storage space.

In case of elastoplastic problem, additional computation is required for the generation of material tangent modulus. The plastic constitutive matrix depends on the local values of stress, which means that it should be computed at all Gauss points in the mesh.

Algorithm 2 shows the steps involved in the computation of elemental tangent matrix for fully-integrated 8-noded hexahedral element. Line 2 of Algorithm 2 shows the loop over optimum number of Gauss points given by quadrature rule, i.e, 8 in case of 8-noded hexahedral element. For each Gauss point, Jacobian, determinant and the inverse of Jacobian matrix is computed in sequence. Line 5 stands for the computation of matrix

B, which consists of shape function derivatives in physical coordinate system. The computation of elemental matrices and force vector is done by carrying-out the required multiplication in line 6.

Algorithm 2 Computation of elemental matrices.

Input: Coordinates, Shape function derivatives
Output: Elemental matrices $\mathbf{K}^e$

1: Initialize $\mathbf{K}^e$, $\mathbf{F}^e$ to zero.
2: **for** $q = 1$ to $\mathcal{Q}$ **do** ▷ Loop over Gauss points $\mathcal{Q}$
3: Compute $\mathbf{J}$
4: Compute determinant and inverse of $\mathbf{J}$
5: Compute $\mathbf{B}$
6: $\mathbf{K}^e += (\mathbf{B}^{\mathrm{T}}\mathbf{D}\mathbf{B})\mathrm{det}\mathbf{J}w_q$
7: **end for**

2.8.2 Assembly

In this step, all elemental tangent matrices are assembled into a global tangent matrix. The assembly of elemental tangent matrices is done on the basis of mesh connectivity. On CPU, the assembly procedure consists of a loop over all elements where non-zero values from elemental tangent matrices are accumulated into the global tangent matrix by mapping local DOF of an element to global DOF.

The global tangent matrix obtained in FEM is sparse in nature as elements are locally connected to each other in the discretized domain. The sparse matrix contains a mix of zero and non-zeros values. Since, zeroes are not useful for computations only non-zeros values are stored in the memory. For this purpose, different sparse storage formats like coordinate (COO), ELLPACK (ELL), compressed sparse row (CSR), etc. have been used in the past, implying different levels of memory utilization (Bell & Garland 2009). As the problem size increases, memory consumption in FEM becomes infeasibly large. Therefore, the usage of sparse formats is a must. However, working with a sparse storage format is not a trivial task as each format introduces a unique data structure. The assembly into global tangent matrix must take into account the data structure of underlying sparse storage format, and accordingly adopt an efficient strategy. In this work, CSR sparse format is used for the storage of the global tangent matrix. The assembly procedure for CSR format is discussed in the subsequent chapter.

2.8.3 Computation of stress

The constitutive relation in elastoplasticity is given in the form of rates and therefore it needs to be integrated over load increments to obtain the correct value. As discussed in Section 2.6, the stress is evaluated using the radial-return method. Algorithm 3 summarizes the key steps in the computation of stress using the radial-return method. Since, computation of stress depends on local values of internal variables like plastic strain, the radial-return method is followed for each Gauss point of each element. The computer implementation uses an outer loop over elements and an inner loop over all Gauss points in an element.

Algorithm 3 Radial-return algorithm.

Input: ${}^{n}\boldsymbol{\sigma}$, $\Delta\boldsymbol{\varepsilon}$, ${}^{n}\varepsilon^{p}_{eq}$
Output: ${}^{n+1}\boldsymbol{\sigma}$, ${}^{n+1}\varepsilon^{p}_{eq}$

1: Compute trial stress (elastic predictor): $\boldsymbol{\sigma}^{\mathrm{tr}}$ using Eq.(2.32)
2: Compute yield function for trial stress: f using Eq. (2.5)
3: **if** $f > 0$ **then**
4: Compute equivalent trial stress: $\sigma^{\mathrm{tr}}_{\mathrm{eq}} = \sqrt{\frac{3}{2}\left(\boldsymbol{\sigma}^{\mathrm{tr}'} : \boldsymbol{\sigma}^{\mathrm{tr}'}\right)}$
5: **while** $|\delta\varepsilon^{p}_{\mathrm{eq}}| > \mathrm{tol}$ **do** ▷ Newton iteration to compute $\Delta\varepsilon^{p}_{\mathrm{eq}}$
6: ${}^{i}\kappa = {}^{n}\kappa + H^{i}(\Delta\varepsilon^{p}_{\mathrm{eq}})$
7: $\delta\varepsilon^{p}_{\mathrm{eq}} = \dfrac{\sigma^{\mathrm{tr}}_{\mathrm{eq}} - 3G({}^{i}(\Delta\varepsilon^{p}_{\mathrm{eq}})) - {}^{i}\kappa}{3G + H}$
8: ${}^{i+1}(\Delta\varepsilon^{p}_{\mathrm{eq}}) = {}^{i}(\Delta\varepsilon^{p}_{\mathrm{eq}}) + \delta\varepsilon^{p}_{\mathrm{eq}}$
9: **end while**
10: $\Delta\boldsymbol{\varepsilon}^{p} = \dfrac{3}{2}\dfrac{\Delta\varepsilon^{p}_{\mathrm{eq}}(\boldsymbol{\sigma}^{\mathrm{tr}'})}{\sigma^{\mathrm{tr}}_{\mathrm{eq}}}$ ▷ $(\boldsymbol{\sigma}^{\mathrm{tr}'})$ is deviatoric part of $\boldsymbol{\sigma}^{\mathrm{tr}}$
11: ${}^{n+1}\boldsymbol{\sigma} = \boldsymbol{\sigma}^{\mathrm{tr}} - 2G\Delta\boldsymbol{\varepsilon}^{p}$
12: ${}^{n+1}\varepsilon^{p}_{\mathrm{eq}} = {}^{n}\varepsilon^{p}_{\mathrm{eq}} + \Delta\varepsilon^{p}_{\mathrm{eq}}$
13: **else**
14: ${}^{n+1}\boldsymbol{\sigma} = \boldsymbol{\sigma}^{\mathrm{tr}}$
15: **end if**

2.8.4 Computation of internal force vector

The computation of internal force is implemented in a similar way as the computation of elemental tangent matrices. The line 6 in Algorithm 2 is replaced by expression from Eq. (2.30) to compute internal force vector, rest of the steps remain identical. For each

element, computation is performed inside a loop over Gauss points.

2.8.5 Convergence criteria

The Newton iteration in Algorithm 1 continues until residual force vector becomes smaller than a certain given value. In this thesis, force-based convergence criteria is used to terminate the Newton iteration. The force-based convergence criteria is given as the L_2 norm of the ratio of unbalanced force vector $\mathbf{f}_{\text{ub}}$ and incremental force vector $\mathbf{f}_{\text{ext}}$. In mathematical form, it is expressed as

$$\frac{||\mathbf{f}_{\text{ub}}||_2}{||\mathbf{f}_{\text{ext}}||_2} \leq \xi_{NR}, \tag{2.42}$$

where $\mathbf{f}_{\text{ub}}$ and $\mathbf{f}_{\text{ext}}$ are taken for a given load step. Here, ξ_{NR} is the user specified convergence tolerance.

2.9 Closure

This chapter discusses the theory of plasticity for a deformable body under the action of external body force and surface traction. The plasticity theory is presented for an isotropic material model with von Mises yield criteria and associated flow rule. Further, finite element formulation is established for the numerical simulation of a three dimensional problem. The same has been used in chapter 3 for implementation on the GPU.

Chapter 3

GPU-acceleration of Assembly-based Elastoplasticity Solver

An efficient finite element analysis of elastoplastic problems is of great importance, as it enables one to estimate actual strength of solid and structures under various loading conditions. However, realistic simulation of various physical processes like metal forming or geophysical problems requires complex large-scale 3D models (Vi et al. 2018, Rietmann et al. 2017). Finite element computation for large-scale models, having millions of degrees of freedom (DOFs), can be very expensive and may lead to huge computational overhead. In addition, the nonlinear nature of elastoplastic problems requires FEM computations iteratively. In such scenario, the computational time of large-scale 3D elastoplastic simulation can be too large to use for practical purposes. Often, the large computational time associated with elastoplastic simulation is reduced by using parallel supercomputers (Bhardwaj et al. 2002, Adams et al. 2004, Balay et al. 2021, Trilinos Project Team 2020).

The computation in elastoplastic analysis consists of three expensive steps, viz., computation of the elemental matrices, assembly of the elemental matrices into a global tangent matrix and solution of linear system of equations. Often, the solution of system of

equations is the dominating step in terms of computational time, and has been studied extensively by many researchers. However, the computation of elemental matrices and their assembly too can become a bottleneck for elastoplastic analysis. The computation of elemental matrices for elements undergoing plastic deformation and their assembly is required in every iteration. As the size of plastic zone increases, the time associated with the computation of the elemental matrices also increases. Therefore, apart from the solution of system of linear equations, the computation of elemental matrices and their assembly into a global tangent matrix along with the computation of stress and strain constitute important stages in elastoplastic analysis. Since these operations are required iteratively, the wall-clock timing can be seriously affected if left unoptimized.

In this chapter[1], we implement all steps of an elastoplastic analysis on GPU and explore all possibilities to minimize the computation time. The incremental-iterative solution approach to elastoplastic analysis is implemented in the form of CPU-based loops, that launch compute kernels on GPU to perform all the computations. The expensive CPU-GPU data transfer is completely avoided inside the computational loops, as all the computations are performed on the GPU. Considering the limited memory space available in the GPU, the elemental matrices are directly assembled into compressed sparse row (CSR) storage format ready to be used by linear solvers. In addition, a parallel strategy for radial-return method is also proposed to compute the stresses on the GPU itself.

3.1 Proposed GPU framework

The primary goal of the current work is to reduce the wall-clock timings for simulation of elastoplastic problems on a GPU. Therefore, it becomes necessary to use GPU for all computationally expensive steps. Fig. 3.1 presents a flow-chart of the proposed solution strategy, indicating the major steps and corresponding hardware on which it is executed. The steps surrounded with single box are executed on CPU, whereas the steps with multiple boxes are executed on GPU. The user-defined functions in CUDA C/C++ (also known as kernels) are developed to implement the steps like computation of elemental matrices and their assembly, computation of stress and computation of

[1]This chapter has been published in **Computing** (Kiran et al. 2023).

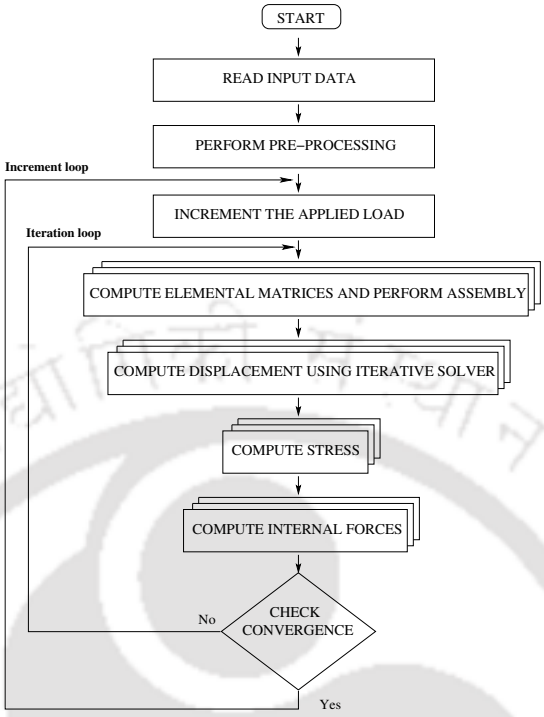


Figure 3.1: A flow-chart for the proposed GPU framework. The steps surrounded with single box are executed on CPU and those with multiple boxes are executed on GPU.

internal forces on the GPU. The computation of displacement can be done by using any suitable library that supports GPU acceleration. In this work, CUSP (Dalton et al. 2014) and Ginkgo (Anzt et al. 2022) libraries are used for the solution of linear system of equations. The algebraic operations like vector-vector addition or subtraction, scalar-vector multiplication, vector-dot product, etc., have been used for updating external force vector, computation of unbalanced force vector and evaluation of termination criteria. These operations are implemented on GPU using THRUST library (Bell & Hoberock 2012).

3.1.1 Computation of elemental matrices and their assembly

The computation of elemental matrix for each element can be performed independently of others, making this step an ideal candidate for parallelization, see Georgescu et al. (2013). Since a GPU has massively parallel processor architecture, a large number of parallel threads can be launched and associated with different elements to perform elemental computation simultaneously. The computed elemental matrices are stored and

later assembled into a global tangent matrix. However, considering a limited memory available with a GPU, the proposed strategy does not store the elemental matrices in memory, rather, assembles directly into a global tangent matrix.

Algorithm 4 shows the steps in computation of elemental matrices on GPU by a single thread. The computation of elemental matrix is implemented as a device function that uses local memory space for storage and computation (see line 1). The computation is done inside a loop over Gauss points as in the sequential implementation. If a Gauss point is in plastic state, as given by an array `pstate` in line 3 of Algorithm 4, the computation of tangent modulus is done, otherwise elastic constitutive matrix is used. For each Gauss point, Jacobian (`jacobian`), inverse Jacobian (`inv_jacobian`) and matrix $\mathbf{B}$ (`B_mat`) are computed, followed by the computation of $\mathbf{B}^{\mathrm{T}}\mathbf{D}\mathbf{B}$ in line 11. The computed elemental matrix is stored in the local memory of each thread.

Algorithm 4 GPU kernel for computation of elemental matrices

Input: `coordinate`, `dN`
Output: `K_e`

1: Allocate local memory for `jacobian`, `B_mat`, `D`, `K_e`
2: **for** $q = 1$ *to* $\mathcal{Q}$ **do** ▷ Loop over Gauss points $\mathcal{Q}$
3: **if** `pstate`$[q] ==$ *plastic* **then**
4: `D` ← calc_tangent_modulus()
5: **else**
6: `D` ← elasic_constitutive_mat()
7: **end if**
8: `jacobian` ← calc_jacobian(`coordinate`, `dN`)
9: `inv_jacobian` ←calc_inverse_jacobian(`jacobian`)
10: `B_mat` ← calc_matrix_B(`inv_jacobian`, `dN`)
11: `K_e`+ = calc_elemental_mat(`B_mat`,`D`)
12: **end for**

The assembly of elemental matrices is also a data parallel task, which is suitable for modern processors like the GPU. The data in a global tangent matrix is stored in a DOF-wise manner. Each DOF has a corresponding row in the global tangent matrix and receives contributions of elemental matrices from neighbouring elements. During parallel assembly, threads assigned to different elements may try to write values to the same memory location corresponding to a common DOF. This creates data race condition, where the outcome of any operation remains undefined. In this work, the issue of data race condition is handled by the use of mesh coloring (Cecka et al. 2011,

Kiran et al. 2018). The goal is to divide the mesh into distinct sets of elements, with no two elements from the same set sharing a node. A distinct color is assigned to each set. The computation for elements belonging to a color set can now be done without any conflict, as no common DOF exists. All colors are processed in sequence. In this thesis, coloring algorithm given in (Martínez-Frutos & Herrero-Pérez 2015) has been used. The race conditions can also be avoided by using atomic operations (NVIDIA Corporation 2022) provided by CUDA. Compared with coloring method, atomic operations are simple to implement and do not require multiple kernel launches. However, atomic operations can be expensive, and their performance is hardware-dependent. On recent versions of GPUs, overheads associated with atomic operations have significantly reduced, making atomic operations an attractive option. The strategies presented in this thesis can also be implemented by using atomic operations.

Since the global tangent matrix in FEM is sparse in nature, different sparse formats like coordinate (COO), ELLPACK (ELL), compressed sparse row (CSR), etc. are used for the storage. However, working with a sparse storage format is not a trivial task as each format introduces a unique data structure, which might not be efficient for other computational steps of FEM. Therefore, assembly into the global tangent matrix must take into account the data structure of underlying sparse storage format, and accordingly adopt an efficient strategy. A commonly used procedure for the assembly of the global tangent matrix consists of two steps. The first step assembles the elemental contribution in COO format where three arrays are used to store non-zero values, row indices and column indices of a matrix. However, the value array usually contains multiple entries for the same values of row and column indices. In second step, each of the arrays are sorted and entries having repeated values in row and column arrays are consolidated. The global stiffness matrix obtained in COO format can now be converted into any other sparse format, if required. This approach works well for a CPU and has been implemented by libraries like MATLAB (The MathWorks Inc. 2021) and Eigen (Guennebaud et al. 2010). However, this approach does not suit well to the GPU architecture. GPUs have limited memory and performance of a GPU-based code is highly dictated by the amount of memory access. The first step of assembly into COO format allocates more memory than required by the actual number of non-zeros. This puts pressure on GPU memory and limits the size of the problem. The second step of assembly into COO format involves

sorting and accumulation of repeated entries. These operations are memory intensive and require a lot of data movement, which is not favorable for the best performance on GPU. In the current work, assembly is directly performed into CSR formats by pre-computing indices of non-zero values.

3.1.1.1 Pre-computing indices into CSR matrix

The CSR sparse storage format uses three arrays to store a sparse matrix. These arrays are: value array to store non-zero values, column indices array to store column indices of non-zero values and row offsets array to store the location of beginning of each row. The row offsets array has size equal to size of the matrix incremented by one, the last entry contains the total number of non-zero values. Figure 3.2a shows an example mesh consisting of quadrilateral elements with a single degree of freedom per node. The full global tangent matrix corresponding to Fig. 3.2a is displayed in Fig. 3.2b, where $*$ denotes non-zero values, numbered row-wise (as stored in CSR format) as shown in Fig. 3.2c. The global tangent matrix has a total of 28 non-zero values, and therefore, the value and the column arrays are allotted space to keep 28 values. The row offsets array is assigned the size of seven values. Figure 3.2d shows the global tangent matrix in CSR format, where the values of column array are displayed for the first, second and last rows. The local to global mapping of elemental DOFs only provides rows and column indices for a particular non-zero value. While this information is sufficient to find the exact position in a full global tangent matrix, the locations into compressed global matrix cannot be found. The direct assembly of elemental matrices into CSR format requires prior knowledge of locations of non-zero values in the value array. For example, let's assume that the value corresponding to 6^{th} row and 5^{th} column is to be modified. Figure 3.2d shows the row offset value for 6^{th} row as 25, indicating that the values corresponding to 6^{th} row lies at locations starting from 25^{th} position in the column and value arrays. However, if one wants to modify the value in 5^{th} column of 6^{th} row, the location for 5^{th} column needs to be searched in the column array. As shown in the column array of Fig. 3.2d, the 5^{th} column comes at 3^{rd} position in the 6^{th} row. Therefore, one has to make changes at 27^{th} $(25 + 2)$ position in the value array to modify the required value. The exact location is obtained by adding relative position of desired column with respect to the first column in the corresponding row. Whenever the assembly is performed,

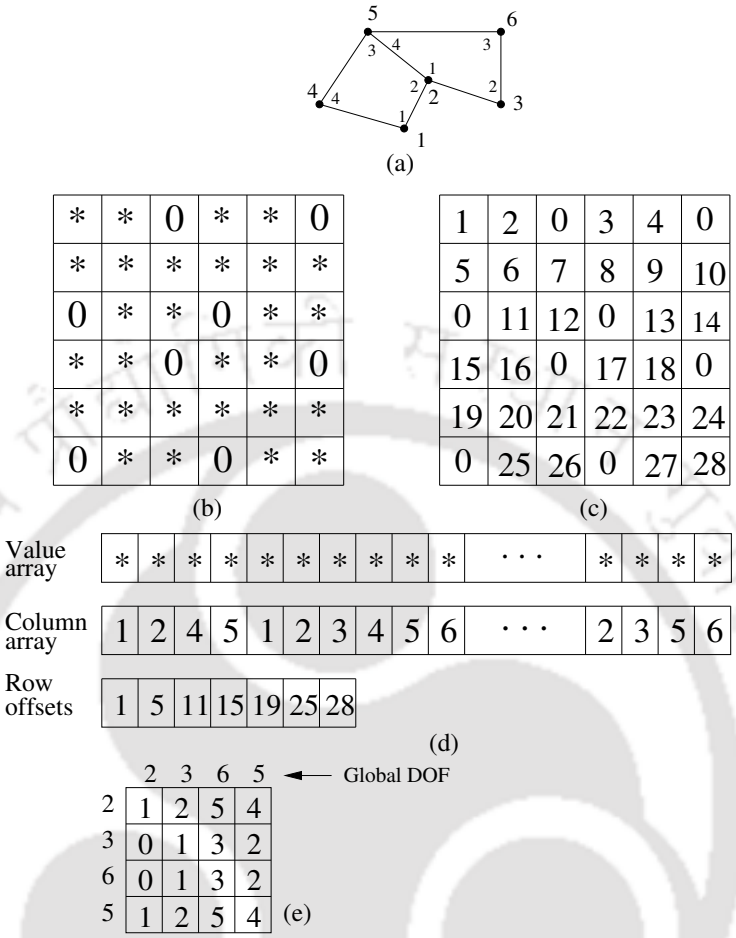


Figure 3.2: The storage of element connectivity for coalesced access.

the relative position of the column index of a non-zero value needs to be searched in the column array, which can be expensive for unstructured meshes. It is to be noted that the relative position of a column index remains fixed as long as the mesh is fixed. This implies that the expensive search operations into column array can be done prior to the assembly step and relative positions of column indices of nonzero values can be pre-computed and stored for later use. Figure 3.2e shows a 4×4 matrix that contains relative positions of column indices for each node of an element. The relative position of a column index depends on the immediate neighborhood of the node and elements with which it is connected. If a node is associate with multiple elements, the relative position is found for each element. If a node has multiple DOF associated with it, the same relative position can be used for all DOFs. The data shown in Fig. 3.2e is referred as CSR indices in the following discussions.

The CUDA kernel that performs assembly for one color is presented in Algorithm 5.

The primary inputs required for the assembly kernel are element connectivity (`connectivity`), coordinate (`coordinate`), derivative of shape functions in reference coordinate (`dN`), CSR indices (`csr_indices`) and row offsets array of CSR storage format (`offsets`). Assigning single thread to each element, the computation is performed by launching as many threads as the maximum number of elements (*max_element*) belonging to a color. Each thread is assigned the task of computation of an elemental matrix and its assembly into the global matrix. The elemental stiffness matrix is computed by following Algorithm 4 (see line 6). Lines 7–12 of Algorithm 5 denote assembly of elemental matrices into the global matrix, stored in CSR sparse storage format. The column and offsets arrays of CSR format depend on the mesh connectivity and remain fixed as long as the mesh does not change. In this thesis, the column and offsets arrays (`offsets`) are precomputed and stored to avoid repetitive assembly. As shown in line 9 of Algorithm 5, elemental matrices are assembled into the value array by computing indices based on element connectivity (`connect`), offset array (`offsets`), and precomputed indices into value array (`csr_indices`). The direct assembly into CSR storage format requires prior knowledge of locations of non-zeros into the value array. Here, `csr_indices` contains precomputed locations of non-zeros in element-wise manner. It is noted that accumulation of non-zero values in array `K_val` is conflict-free.

Since single thread per element strategy is used in computation, each thread is responsible for reading input data associated with one element. Keeping in mind that the most efficient data structure for a GPU is the one that allows coalesced memory access, the input data are reordered in such a way that consecutive threads access consecutive locations in the memory. The data access pattern for element connectivity is shown in Fig. 3.3. If data is not reordered, threads (denoted as T_0, T_1, etc.) make strided access to fetch data from memory. Here, stride size is eight due to eight entries of element connectivity per element (Fig. 3.3a). If corresponding entries of element connectivity for all elements are stored side-by-side (Cecka et al. 2011), the strided access can be totally prevented. As shown in Fig. 3.3b, first entry of element connectivity for all elements are accessed in a coalesced manner. Similar arrangement is made for the other entries. The coordinates and CSR indices are also reordered and accessed like the element connectivity.

Algorithm 5 GPU kernel for assembly of global tangent matrix.

Input: `coordinate`, `connectivity`, `csr_indices`, `offsets`, `dN`
Output: `K_val`

procedure CALC_TANGENT_MAT(`coordinate`, `connectivity`, `csr_indices`, `offsets`, `dN`)
  $threadId \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
  $elem\_no \leftarrow threadId$
  if $elem\_no < max\_element$ **then**
    `connect` ← get_connectivity()
    `K_e` ← calc_element_mat(`coordinate`, `dN`) ▷ Computation of elemental matrix, see Algorithm 4
    for $i = 1$ to 24 **do** ▷ Assembly into global matrix without conflict.
      for $j = 1$ to 24 **do**
        $id \leftarrow$ calc_indices(i,j, `csr_indices`, `offsets`, `connect`)
        `K_val`(id) += `K_e`(i, j)
      end for
    end for
  end if
end procedure

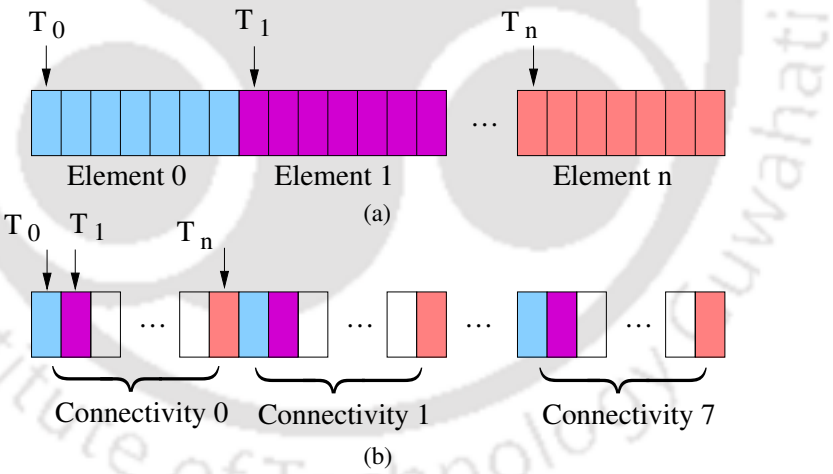


Figure 3.3: The storage of element connectivity for coalesced access.

3.1.2 Computation of stress

The computation of stress in elastoplastic analysis is performed by following the radial-return method (Section 2.6). The key steps in GPU implementation of radial-return method are presented in Algorithm 6. As can be seen, the computation of stress is divided into three parts or procedures: stress predictor, decompose stress and stress corrector. Each of the procedures is implemented as separate CUDA kernel and optimized to handle

Algorithm 6 GPU kernel for computation of stress using radial-return method

Input: `coordinate`, `dN`, `U`, `yield_value`, `D_e`
Output: `stress`, `pstate`

procedure STRESS_PREDICTOR(`coordinate`, `dN`, `U`, `D_e`)
  $threadId \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
  $elem\_no \leftarrow threadId$
  if $elem\_no < max\_element$ **then**
    for $q = 1$ *to* $\mathcal{Q}$ **do**
      `jacobian` ← calc_jacobian(`coordinate`, `dN`)
      `inv_jacobian` ←calc_inverse_jacobian(`jacobian`)
      `B_mat` ← calc_matrix_B(`inverse_jacobian`, `dN`)
      `strain` ← calc_strain(`B_mat`,`U`)
      `trial_stress` ← `D_e`*`strain`
    end for
  end if
end procedure
procedure DECOMPOSE_STRESS(`trial_stress`)
  $threadId \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
  $elem\_no \leftarrow threadId$
  if $elem\_no < max\_element$ **then**
    `dev_stress` ← calc_deviatoric_stress(`trial_stress`)
    `eq_stress` ← calc_equivalent_stress(`dev_stress`)
  end if
end procedure
procedure STRESS_CORRECTOR(`trial_stress`, `dev_stress`, `eq_stress`, `yield_value`)
  $threadId \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
  $elem\_no \leftarrow threadId$
  if $elem\_no < max\_element$ **then**
    for $q = 1$ *to* $\mathcal{Q}$ **do**
      `yield_fn` = `eq_stress`[q] - `yield_value`[q]
      if `yield_fn` == (+)ve **then**
        `pstate`[q] = *plastic*
        `plastic_strain` ← calc_plastic_strain(`dev_stress`,`eq_stress`,μ,H) ▷ Refer to Algorithm 3
        `stress` ← `trial_stress` - 2*μ*`plastic_strain`
      else
        `pstate`[q] = *elastic*
        `stress` ← `trial_stress`
      end if
    end for
  end if
end procedure

specific task. The element level parallelization strategy is adopted in each of the CUDA kernels (line 3), where single thread is assigned the task to compute stress for all Gauss points belonging to an element. The computations corresponding to each Gauss point is done inside a loop (see line 5). Lines 6–8 show the computation of Jacobian, inverse of Jacobian and matrix **B**, implemented in the same way as elemental matrices, using the local memory of GPU. The total strain (`strain`) and trial stress (`trial_stress`) are stored in global memory as they are needed later. Lines 18–19 stand for the computation of deviatoric part of trial stress and equivalent stress (von Mises stress). The deviatoric part of trial stress is used in computation of plastic strain along with the equivalent trial stress (see line 30). The yield function is evaluated for each Gauss point in line 27 using equivalent trial stress and yield value from previously converged load step. If yield function is found positive, the state of Gauss point is updated to plastic in the array `pstate`. For the Gauss points that undergo plastic deformation, increment in plastic strain is computed (line 30 of Algorithm 6) by using NR iteration as discussed in Algorithm 3. Consequently, the actual stress is determined by updating the trial stress. In case of negative yield function, the state of Gauss point remains elastic and trial stress is accepted as the final stress (see line 34 of Algorithm 6).

3.1.3 Computation of internal forces

The internal force vector is computed for each element and assembled into a global force vector. The computation of internal force vector for each element can be done independently of others, which prompt us to adopt element level parallelization strategy. However, parallel assembly of elemental force vectors into a global force vector suffers from data race condition, in the same way as assembly of elemental matrices (Section 3.1.1). The mesh coloring approach is used to assemble force vectors without conflict. Since mesh coloring is already used in assembly of elemental matrices, the same set of colored elements are used for the GPU implementation of internal forces. The GPU kernel for computation of internal force vector is presented in Algorithm 7. The parallel strategy assigns single thread to each element for computation of force vector (`F_e`) and its assembly into a global vector (`F`). Lines 6–8 show the computation of Jacobian, inverse of Jacobian and matrix **B**, implemented in the local memory along with internal force

vector. The assembly of the global force vector is done in line 11 by computing indices based on element connectivity. Since coloring approach is used for the computation, elemental force vectors are accumulated in a conflict-free manner.

Algorithm 7 GPU kernel for computation of internal forces

Input: `connectivity`, `coordinate`, `stress`, `dN`
Output: `F`

```
1:  procedure CALC_INTERNAL_FORCES(connectivity, coordinate, stress, dN)
2:      threadId ← blockIdx.x * blockDim.x + threadIdx.x
3:      elem_no ← threadId
4:      if elem_no < max_element then
5:          for q = 1 to Q do
6:              jacobian ← calc_jacobian(coordinate, dN)
7:              inv_jacobian ←calc_inverse_jacobian(jacobian)
8:              B_mat ← calc_matrix_B(inv_jacobian, dN)
9:              F_e += calc_int_force(stress, B_mat)
10:         end for
11:         for i = 1 to 24 do                    ▷ Assembly into global vector
12:             id ← calc_indices(i, connectivity)
13:             F[id] += F_e[i]
14:         end for
15:     end if
16: end procedure
```

The implementation of Newton iterations in the proposed GPU framework is presented in Algorithm 8. It can be seen that all computations are performed on GPU by launching a series of CUDA kernels inside NR loop running on the CPU. The sequence of CUDA kernel shows the control flow and dependency of each step on the result of previous step. The only data transfer between CPU and GPU is of computed tolerance value which is needed to check for the convergence.

3.2 Results and Discussion

The performance of the proposed framework is demonstrated by solving three benchmark examples. The solution of linear system of equations is done by diagonal preconditioned Conjugate Gradient (PCG) iterative solver from GPU-based CUSP (Dalton et al. 2014) and Ginkgo (Anzt et al. 2022) libraries. It is noted that the same library is used for

Algorithm 8 GPU-based implementation of Newton-Raphson (NR) iterations in the proposed framework.

Input: `coordinates`, `connectivity`, Problem parameters like material properties and boundary conditions

Output: `U, stress`

1: $Blocksize = 256$
2: $Nblock = \text{ceil}(nelem/256)$
3: **for** $Itr < Max\_itr$ **do** ▷ Newton-Raphson iterations
4: [`K_val,K_col`] ← CALC_TANGENT_MAT<<< $Nblock$, $Blocksize$ >>>(`coordinate`, `connectivity`, `dN`, `csr_indices`) ▷ `K` is stored in CSR format using `K_val, K_col` and `offsets` arrays.
5: Solve on GPU: $\mathtt{K}\Delta\mathtt{U}=\mathtt{F}_{\mathtt{ub}}$ ▷ Using CUSP or Ginkgo solver
6: thrust::transform(Δ`U`, Δ`U`+$ndof$, `U`, `U`, thrust::plus<double>()) ▷ Update `U` on GPU: `U`+=Δ`U`
7: `trial_stress` ← STRESS_PREDICTOR<<< $Nblock$, $Blocksize$ >>>(`coordinate`, `dN`, `U`)
8: [`dev_stress`, `eq_stress`] ← DECOMPOSE_STRESS<<< $Nblock$, $Blocksize$ >>>(`trial_stress`)
9: `stress` ← STRESS_CORRECTOR<<< $Nblock$, $Blocksize$ >>>(`trial_stress`, `dev_stress`, `eq_stress`)
10: $\mathtt{F}_{\mathtt{int}}$ ← CALC_INTERNAL_FORCES<<< $Nblock$, $Blocksize$ >>>(`coordinate`, `dN`, `stress`)
11: thrust::transform($\mathtt{F}_{\mathtt{ext}}$,$\mathtt{F}_{\mathtt{ext}}$+$ndof$, $\mathtt{F}_{\mathtt{int}}$, $\mathtt{F}_{\mathtt{ub}}$, thrust::minus<double>()) ▷ $\mathtt{F}_{\mathtt{ub}} = \mathtt{F}_{\mathtt{ext}} - \mathtt{F}_{\mathtt{int}}$
12: tol ← check_convergence() ▷ Executes on GPU using THRUST
13: **if** $tol < \xi_{NR}$ **then**
14: Update variables ▷ Executes on GPU using THRUST
15: Break the loop
16: **end if**
17: **end for**

both CPU and GPU implementations. The goal is to emphasize the effect of GPU acceleration in elastoplastic simulation for all steps, except the linear solver. The finite element meshes for all benchmark examples are created in commercial package ABAQUS (Systèmes 2017) by using 8-noded hexahedral elements. Depending upon the geometry of the examples, both structured and unstructured meshes are generated. However, both CPU and GPU implementations perform local computations for all elements in the mesh, and make no distinction between the structured and the unstructured mesh. The computational experiments seek answer to the following questions.

- How much is the speedup obtained by resorting to GPU for elastoplastic computation?
- How does the speedup vary with the mesh refinement and plasticity percentage?
- Are there bottlenecks in the GPU optimized code and what are the scopes for further

improvement?

The proposed strategies have been implemented with CUDA C/C++ and built using `nvcc` version 9.2 with arch=sm35 flag. All results are obtained by considering double precision floating point arithmetic. The kernel timings are reported as a average value for three runs. The hardware used for numerical experiments has the following specifications.

- CPU: Intel Xeon® E5-2650 processor clocked at 2.2 GHz with 128 GB RAM.
- GPU: NVIDIA Tesla K40c having 2880 CUDA cores and 12 GB of memory. The GPU is clocked at 745 MHz with memory bandwidth of 288 GB/sec.

3.2.1 Computational examples

3.2.1.1 A unit cube subjected to distributed load

In order to validate the GPU implementation, a standard test example with a known analytical solution as given in the literature (Markopoulos et al. 2015) is taken. It is a cube with unit dimension as shown in Fig. 3.4 with boundary conditions. The elastoplastic analysis is performed for von Mises yield criteria and isotropic linear hardening law with following material parameters: Young's modulus E= 200 GPa, Poisson's ratio $\nu = 0.3$, initial yield stress $\sigma_y = 450$ MPa, and hardening coefficient $H = 66$ GPa. Due to the boundary condition being considered (see Fig. 3.4), only uniaxial stress appears in the body and remains equal to the applied distributed load. A total of 600 MPa distributed load is applied in three successive steps. Thus, considering the yield stress of 450 MPa, plasticity appears in last step. A single element mesh is used for the analysis. Figure 3.5 shows the stress (σ_z) - strain (ϵ_z) plot for the GPU implementation and the analytical model given by Eq. (2.9). The GPU results are able to predict the same stress values as the analytical solution of the material model.

Now, the cube is discretized with different number of elements (see Table 3.1) to obtain the performance of GPU implementation for large-scale example. The material properties remains the same, but Dirichlet boundary condition is modified to constraint all DOFs on the plane $z = 0$ (see Fig. 3.6). A distributed load of 800 MPa is applied in one step. The convergence tolerance for Newton-Raphson algorithm is set to 10^{-6}, and

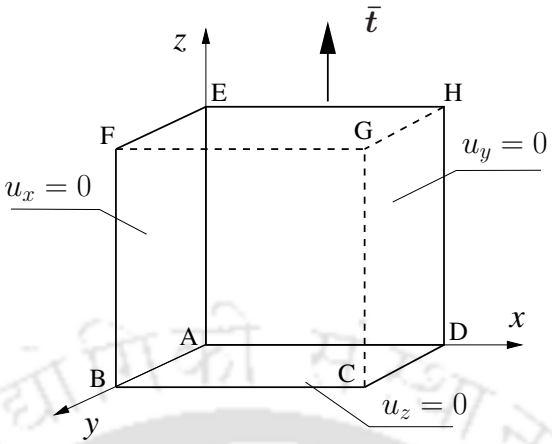


Figure 3.4: A unit cube with symmetric boundary conditions. A distributed load $\bar{t}$ is applied on the face EFGH.

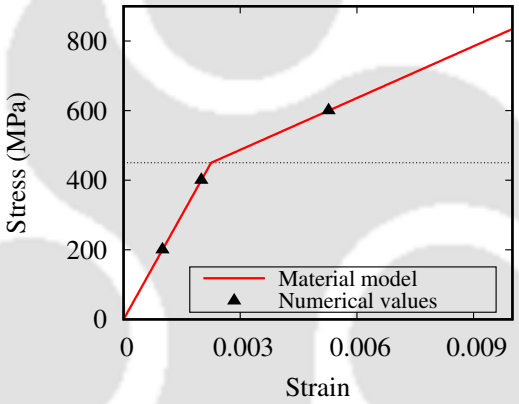


Figure 3.5: Stress (σ_z) - strain (ϵ_z) curve for a cube under uniaxial loading. The dashed horizontal line shows the initial yield stress $\sigma_y = 450$ MPa.

for PCG to 10^{-7}. Figure 3.7 shows a typical distribution of von Mises stress over the cube. As can be seen, von Mises stresses are concentrated near the vertices.

Table 3.1: Mesh for the cube problem.

Mesh	Elements	Nodes	Degrees of freedom
C1	110 592	117 649	352 947
C2	216 000	226 981	680 943
C3	4 38 976	456 533	1 369 599
C4	8 84 736	912 673	2 738 019
C5	1 520 875	1 560 896	4 682 688

The execution timings of the CPU and GPU implementations are presented in Tables 3.2 and 3.3, respectively. In both the tables, the first column shows the mesh and the

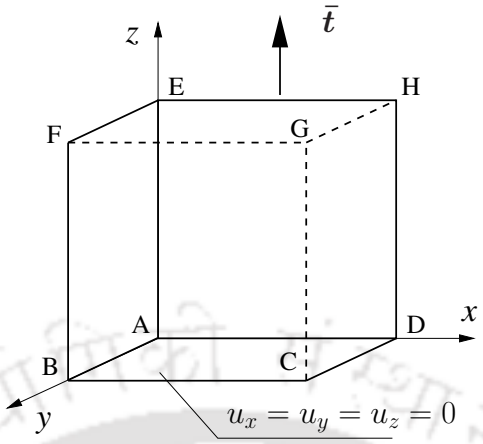


Figure 3.6: Cube example for large scale experiments.

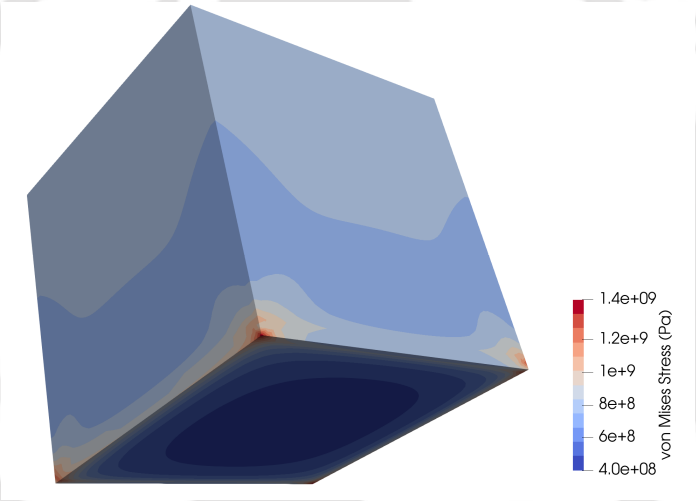


Figure 3.7: Distribution of von Mises stress over the cube.

second column indicates the number of NR iterations for convergence, while rest of the columns show timings for different steps in the elastoplastic analysis. Looking at the CPU timings, it can be observed that the computation of elemental matrices and assembly takes more time than the linear solver for all mesh sizes. It is noted that the linear solver is already running on the GPU. Further, the comparison of assembly timings and wall-clock timings in Table 3.2 reveals that the proportion of assembly timings remain in the range 61–72% of wall-clock timings for all mesh sizes. This leads to a conclusion that the acceleration of only linear solver step of FEM is not sufficient to achieve the best performance for elastoplastic solver on the GPU. Moreover, when the assembly step is performed on the GPU, new timings (Table 3.3) becomes lesser than the CPU timings of computation of stress and internal forces (see Table 3.2). The computation of stress

and internal forces now becomes a new bottleneck, indicating that GPU acceleration of all steps in elastoplastic solver is essential to achieve minimum wall-clock timings on the GPU. For assembly, the computation of CSR indices on CPU takes 15.67 sec for the finest mesh. On GPU, the timing gets reduced to 2.42 sec, indicating speedup of 6.48×. The storage of CSR indices requires additional 371.3 MB space in memory for the finest mesh.

Table 3.2: CPU timings (in sec) for cube under uniaxial loading, except for linear solver that is executed on GPU using CUSP library.

Mesh	NR	Assembly	Stress	Internal forces	Linear solver	Wall-clock
C1	4	42.83	4.53	4.25	7.17	61.58
C2	5	102.42	10.86	10.35	18.19	145.87
C3	5	209.86	22.25	20.99	48.41	307.97
C4	5	455.31	48.23	45.60	136.54	697.21
C5	5	778.59	83.01	78.32	320.48	1279

Table 3.3: GPU timings (in sec) for cube under uniaxial loading.

Mesh	NR	Assembly	Stress	Internal forces	Linear solver	Wall-clock
C1	4	0.71	0.08	0.06	6.08	10.23
C2	5	1.80	0.19	0.16	16.53	23.04
C3	5	3.47	0.38	0.32	46.27	57.05
C4	5	6.67	0.73	0.63	132.61	152.85
C5	5	11.17	1.29	1.09	313.74	345.25

The reduction in execution timings of assembly step demonstrates exceptional performance of the proposed strategy. The GPU-based strategy assigns a single thread to each element enabling balanced work distribution and providing abundant parallelism. Additionally, coalesced access to input data (Section 3.1.1) helps in reducing the latency of memory transactions and improving computational time. The computation of elemental matrices and assembly for the finest mesh consisting of 4.7 million DOFs is performed in 11.17 sec, which is just 3.24% of the wall-clock time. The GPU timings for the assembly step further show that performance-detrimental factors like extensive use of local memory and uncoalesced access for output data have minimum effect on performance. The computation of stress and internal force on the GPU also achieve remarkable reduction in the execution time. The GPU implementation of radial-return algorithm performs stress computation in just 1.29 sec for 4.7 million DOFs mesh. The computation and

assembly of internal force vectors on GPU are completed for 4.7 million DOFs in just 1.09 sec (see Table 3.3).

The speedups achieved by the GPU implementation over the CPU in assembly, computation of stress, computational of internal force and wall-clock timings are presented in Fig. 3.8. The GPU-based assembly strategy achieves speedups in the range 56.6×–69.7× for all the mesh sizes. The computation of stress and internal force achieves speedup in the range 56.4×–66.1× and 53.7×–62.5×, respectively (see Fig. 3.8b & 3.8c). The overall speedup using the proposed GPU strategy is shown in Fig. 3.8d by comparing the wall-clock timings of the CPU implementation. It is observed that the speedup decreases as the mesh size increases. This is due to the increasing proportion of the linear solver timings in the wall-clock timings of the CPU implementation. As shown in Table 3.2, we see the variation of solver time as 11.31%, 12.14%, 15.13%, 19.42%, and 24.87% of the total time with the mesh size. Because the linear solver remains the same in both the CPU and GPU implementations, the solver time do not contribute to the parallel speedup. Therefore, increasing proportion of the solver timings decreases the overall speedup with the mesh size.

3.2.1.2 L-bracket

The second example is an L-bracket, which is adopted from Prabhune & Suresh (2020). Figure 3.9 illustrates the geometry of the L-bracket in two dimensions (2D), where all dimensions are in meters. The corresponding 3D problem is solved by considering unit thickness and sliding boundary conditions. An external distributed force $\bar{\boldsymbol{t}}$ is applied on the top surface, and can be varied to get different level of plastic deformation in the body. The material properties are: Young's modulus E= 206900 Pa, Poisson's ratio ν = 0.29, initial yield stress σ_y = 450 Pa, and hardening coefficient H = 10000 Pa. To evaluate the performance of the GPU implementation, the L-bracket is discretized with varying number of elements to get different meshes (see Table 3.4). The external load is applied in the range 120–182 Pa to get different levels of plasticity. The execution timings are obtained for all meshes corresponding to each load. The distribution of von Mises stress is displayed in Fig. 3.10 along with a typical mesh.

The speedup obtained by the GPU implementation in assembly over CPU is pre-

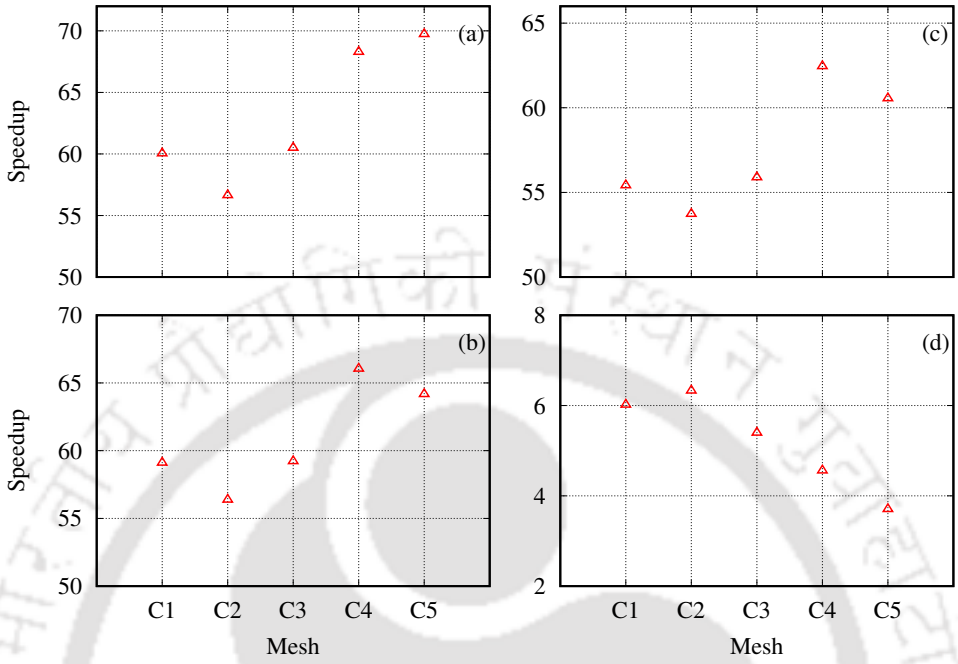


Figure 3.8: Speedup for cube problem with increasing mesh size in (a) Assembly, (b) Computation of stress, (c) Computation of internal stress, (d) Wall-clock time.

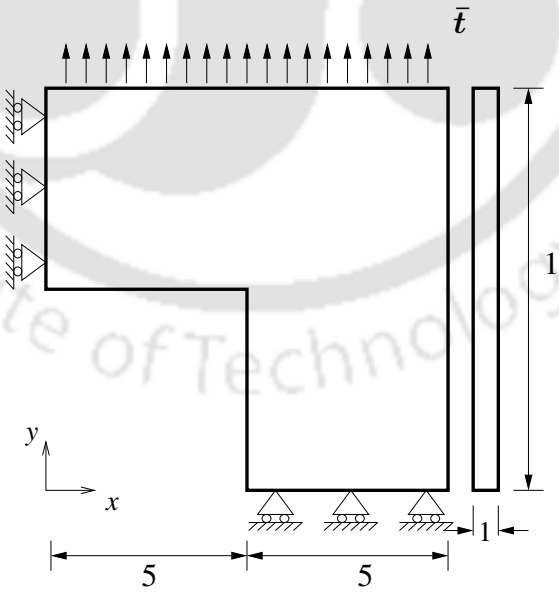
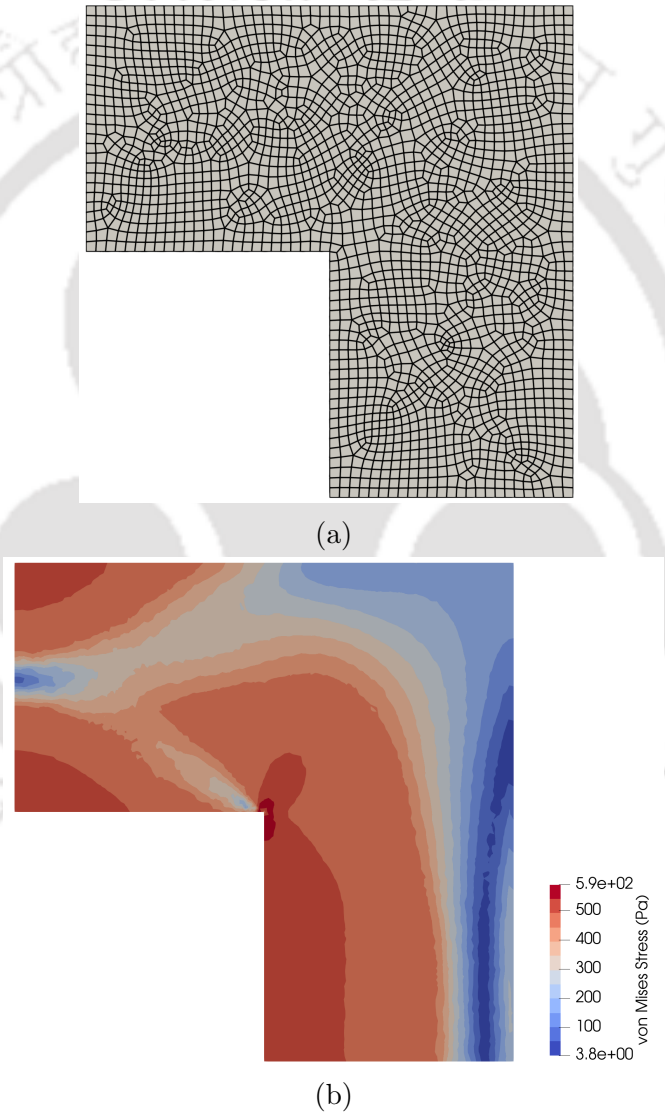


Figure 3.9: L-bracket benchmark. All dimensions are in meters (m).

sented in Fig. 3.11. The speedup is obtained for all the meshes (Table 3.4) and four load values (120 Pa, 144 Pa, 166 Pa, 182 Pa) indicating different levels of plasticity. As

Table 3.4: Mesh for the L-bracket benchmark.

Mesh	Elements	Nodes	Degrees of freedom
L1	99 740	112 002	336 006
L2	200 850	219 912	659 736
L3	400 672	431 307	1 293 921
L4	820 407	868 538	2 605 614
L5	1 580 982	1 655 640	4 966 920



(a)

(b)

Figure 3.10: L-bracket benchmark. A typical mesh and von Mises stress distribution (in Pa).

seen in Fig. 3.11a, the speedup is found to increase with increase in the plastic deformation. This is primarily due to the CPU strategy, which performs computation inside NR

loop only for elements undergoing plastic deformation. When the amount of plasticity is modest, the computation of elemental matrix is done only for a small number of elements exhibiting plastic behaviour, reducing the time spent in the assembly. At higher level of plasticity, as more and more number of elements reach plastic state, the assembly time increases due to the increased computing cost. The GPU-based assembly strategy is found effective in handling the increased computation associated with increasing plasticity and achieves speedups in the range 30×–35.1× for about 50% plasticity compared to 20.4×–24.2× speedups for about 5% plasticity. Figure 3.11b illustrates the speedup obtained in computation of stress using radial-return method. The GPU-based strategy achieves steady speedup in the range 47.3×–56.7× for all levels of plasticity. The speedup, however, increases slightly with the mesh refinement. Figure 3.11c presents the speedup achieved by GPU strategy in the computation of internal forces, reaching 63.9× for the finest mesh. The speedup increases with the mesh refinement but remains almost constant with respect to the plasticity percentage. The wall-clock timings of the GPU and the CPU implementations are compared by evaluating speedup and are presented in Fig. 3.11d. It is observed that speedup remains constant with the plasticity percentage and decreases with mesh refinement due to the reason explained in Section 3.2.1.1.

3.2.1.3 Plate with multiple holes

The next example is a flat plate with multiple circular holes, which is adopted from Yusa et al. (2018). Figure 3.12 describes the geometry of the plate with 16 holes and boundary conditions. The material parameters are taken as: Young's modulus $E = 200$ GPa, Poisson's ratio $\nu = 0.3$, initial yield stress $\sigma_y = 200$ MPa, and hardening coefficient $H = 20$ GPa. Table 3.5 presents the list of meshes used for the evaluation of performance characteristic of the GPU implementation. The external load $\bar{\boldsymbol{t}}$ is varied in the range 100–175 MPa to get the performance at different level of plasticity. Figure 3.13 depicts a typical mesh and distribution of von Mises stress over the plate. As expected the stresses are concentrated near each holes under the action of the applied tensile load.

Figure 3.14 shows the speedups obtained using the GPU implementation over the CPU in individual steps as well as the wall-clock timings. The speedup in assembly step

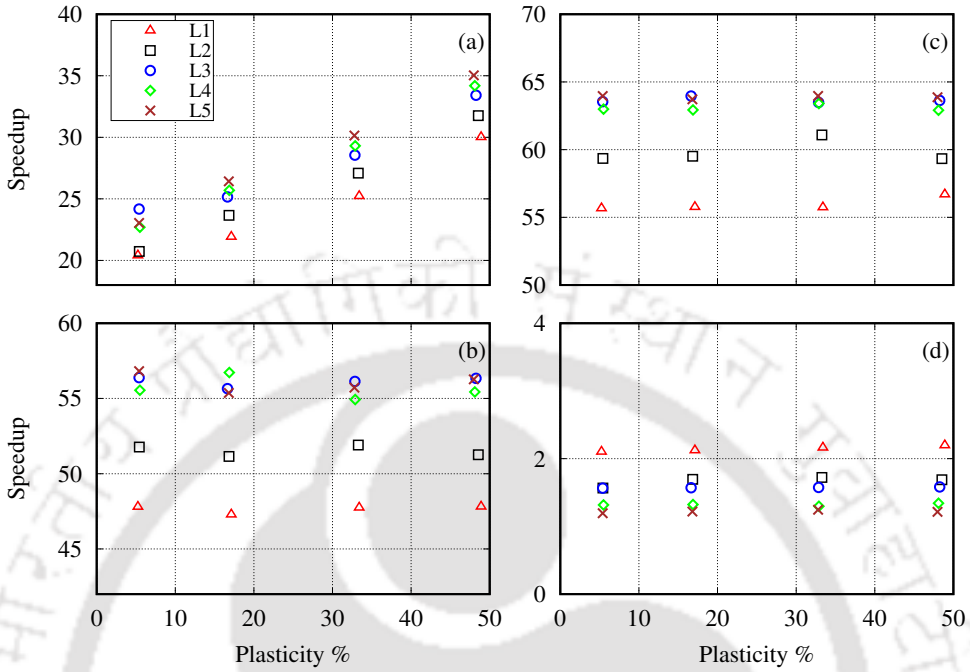


Figure 3.11: Speedup for L-bracket mesh with increasing plasticity percentage in (a) Assembly, (b) Computation of stress, (c) Computation of internal stress, (d) Wall-clock time.

Table 3.5: Mesh for the plate with multiple holes benchmark.

Mesh	Elements	Nodes	Degrees of freedom
P1	103 026	140 296	420 888
P2	250 428	318 015	954 045
P3	528 800	642 342	1 927 026
P4	724 645	878 778	2 636 334
P5	1 490 797	1 718 752	5 156 256

is shown in Fig. 3.14a, where speedup is found to be increasing with the plastic deformation. For low level of plastic deformation, the speedups are in the range of 21.7×–25.4×, whereas speedups of 23.4× to 28.7× are achieved for higher level of plastic deformation. This trend is similar to the one observed for the L-bracket and can be explained in the same way as in Section 3.2.1.2. Further, this establishes that the proposed GPU strategy performs well with increasing computational load due to plasticity. Figure 3.14b shows the speedup in computation of stress, indicating 47.2×–59.4× speedup for all mesh sizes. The speedup remains almost constant with the plastic deformation, whereas it increases

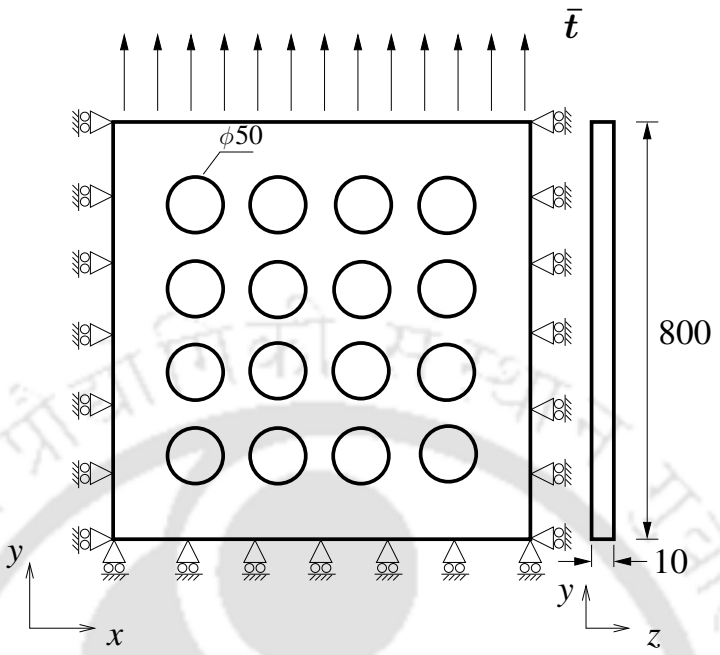
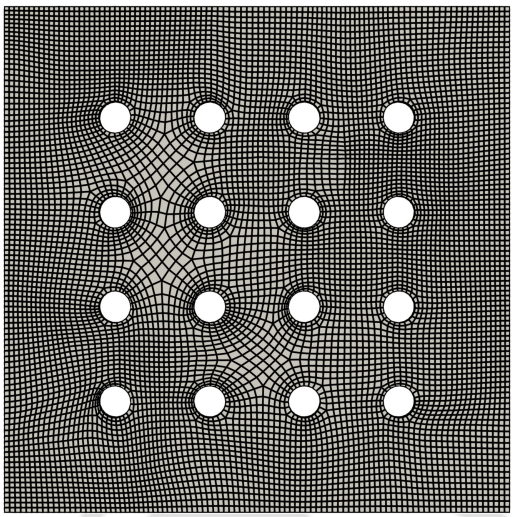


Figure 3.12: A square flat plate with multiple holes. All dimensions are in millimeters (mm).

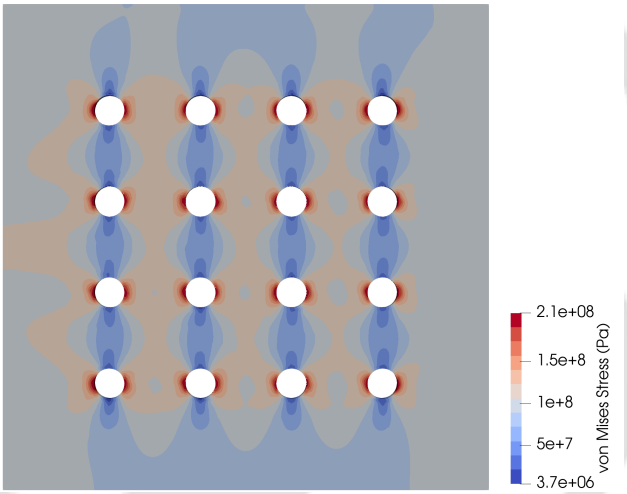
a bit with the mesh refinement. The speedup in range 55.4×–67.3× is obtained for the computation of internal forces, as shown in Fig. 3.14c. The GPU strategy achieves speedup consistent with increasing plastic deformation. The overall speedup achieved by the GPU strategy is plotted in Fig. 3.14d by comparing the wall-clock timings. The speedup is obtained in the range 1.2×–1.9× for all mesh sizes and remains consistent with the increasing plasticity percentage. The trend of decreasing speedup with increasing mesh size is similar to the cube and the L-bracket examples.

3.2.2 Performance limiter in GPU implementation

Figure 3.15 shows the break-up of execution timings for cube, L-bracket and plate with multiple holes examples. Results are displayed for L-bracket and plate with multiples holes examples at the loads that produce the highest amount of plasticity. As shown in Fig. 3.15, for the cube example the solver time constitutes 59.4 to 90.1% of the wall-clock time for all the meshes, whereas the share of assembly, computation of stress and internal forces together lies in the range 3.9 to 9.2% for all the mesh sizes. Similar trend is observed in case of L-bracket and plate with multiple holes examples, where



(a)



(b)

Figure 3.13: A typical mesh and von Mises stress distribution over plate with multiple holes.

linear solver consumes 89.8–98.9% and 91.9–98.7% of the wall-clock time, respectively. Therefore, it can be concluded that the linear solver is the most time consuming component in the GPU-based elastoplastic analysis. The rest of the steps achieve significant speedup on the GPU and contribute little to the overall execution timings unlike the CPU implementation. It can be argued that any effort to reduce the timing of linear solver will have significant impact on the wall-clock timings. Further, if the execution time of the linear solver is reduced, the proposed GPU strategy can achieve much higher speedup in wall-clock timings. In order to demonstrate the effect of linear solver timings on speedup, the CPU and the GPU implementations are executed with diagonal pre-

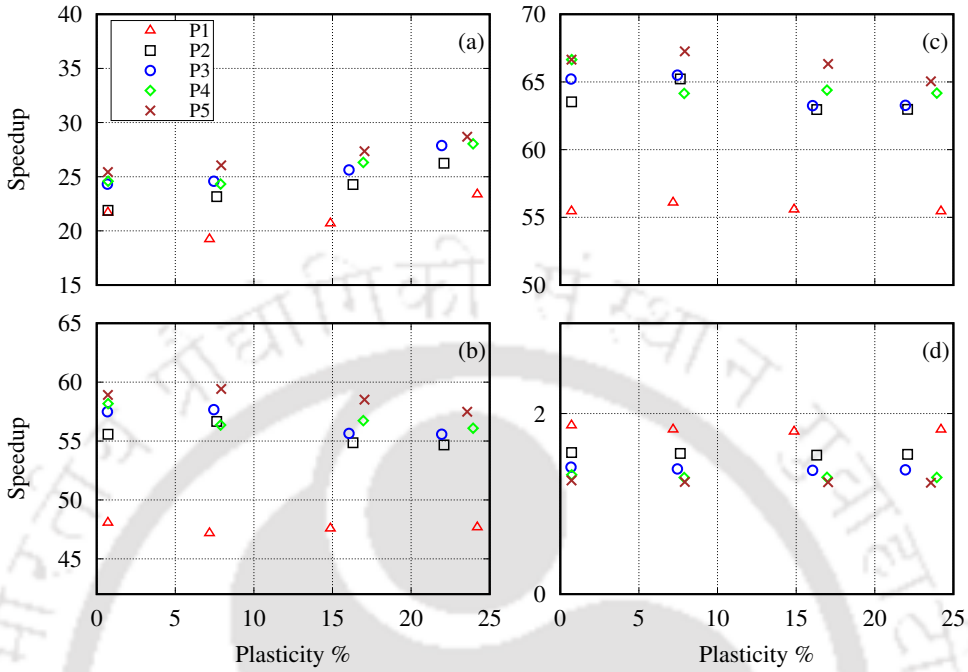


Figure 3.14: Speedup for the plate with multiple holes in (a) Assembly, (b) Computation of stress, (c) Computation of internal stress, (d) Wall-clock time.

conditioned CG solver from Ginkgo library. Ginkgo is a recently developed sparse linear algebra library designed specifically for GPUs. A comparison of the solver timings of Ginkgo and CUSP library is done for all the examples, and speedup achieved by Ginkgo is shown in Table 3.6. The CG solver from Ginkgo is found to be faster than that of the CUSP solver for all the examples and mesh sizes. For the finest mesh, speedups of 1.83×, 2.49× and 2.03× are achieved for cube, L-bracket and plate with multiple holes examples, respectively. Figure 3.16 shows the speedup achieved by the GPU implementation when using Ginkgo library and is compared with the previous speedups obtained using the CUSP library. The overall speedup for the cube example increases by a factor of 1.01×–1.35× for all the mesh sizes and 1.39× for the finest mesh. For L-bracket and plate with multiple holes examples an increment of 1.13×–1.34× and 1.03×–1.19× in the speedup is observed, respectively.

It is noted that the diagonal preconditioner used in this thesis is the simplest. To further improve the performance of the linear solver, more sophisticated preconditioners like LU must be used. Advanced preconditioners like multigrid are very useful in

Table 3.6: Speedup achieved by Ginkgo over CUSP solver on GPU.

Cube		L-bracket		Plate with holes	
Mesh	Speedup	Mesh	Speedup	Mesh	Speedup
C1	1.15×	L1	1.43×	P1	1.32×
C2	1.27×	L2	1.63×	P2	1.55×
C3	1.45×	L3	1.98×	P3	1.81×
C4	1.69×	L4	2.27×	P4	1.90×
C5	1.83×	L5	2.49×	P5	2.03×

accelerating linear system of equations from FEM and can be employed to speed up the linear solver step in elastoplastic analysis. However, the development and efficient implementation of multigrid preconditioners on GPU remains an active field of research.

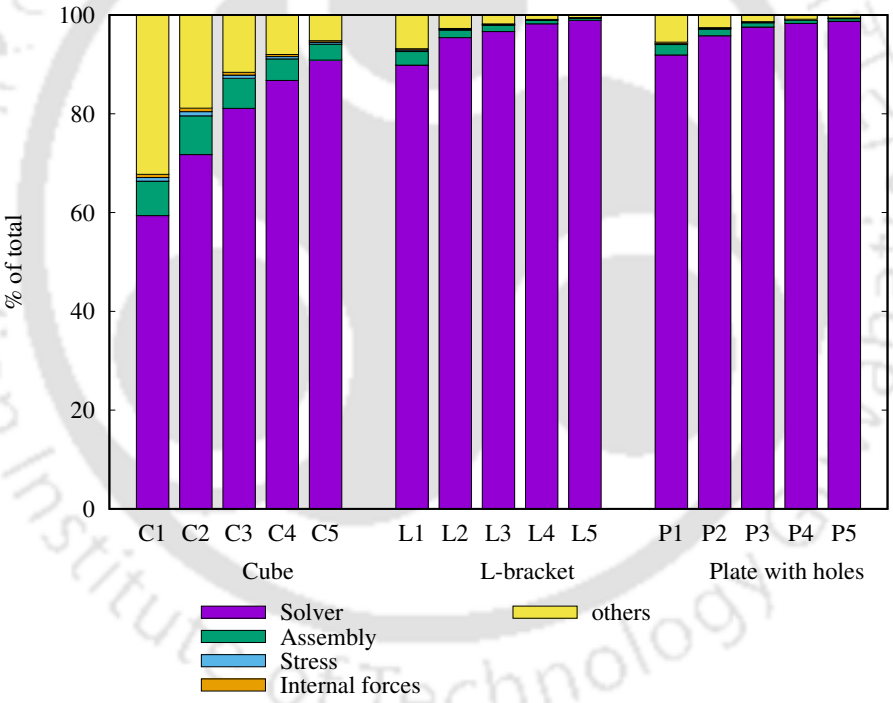


Figure 3.15: Execution time break-up of GPU solver.

3.3 Closure

In this chapter, a novel GPU-based framework for numerical analysis of elastoplastic problems was proposed. The framework was developed in CUDA C/C++ NVIDIA Corporation (2022) and performs all computations in finite element analysis of elastoplastic

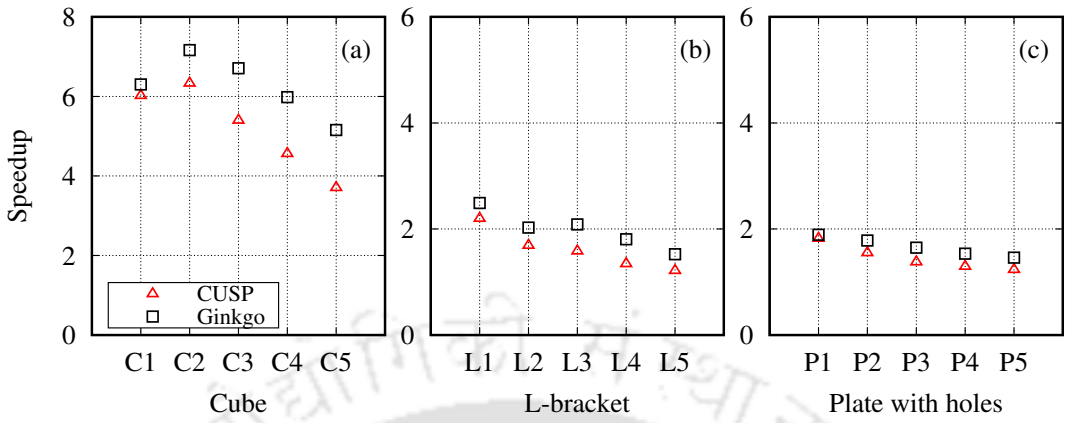


Figure 3.16: Comparison of speedup with CUSP and Ginkgo library.

problems using a GPU. The assembly of elemental matrices was performed directly into CSR sparse storage format in GPU memory, which can be readily used with any linear solver. In order to efficiently compute the stresses, a novel parallel strategy for the radial-return method was developed that makes an efficient use of the GPU memory and provides fine grain parallelism. In order to demonstrate the effectiveness of the proposed framework, computational experiments were conducted over three elastoplastic examples involving unstructured mesh. During the computation of the elemental matrices and their assembly, speedups in the range 20.4× to 69.7× were obtained over the CPU implementation. The computation of stress using the radial-return method showed speedups of 47.2× to 66.1× for all the examples. In the computation of the internal force vectors and their assembly, speedup in the range 53.7× to 67.3× was found over the CPU implementation. Further, the wall-clock timings of the GPU implementation were compared with the CPU implementation that uses a GPU-based linear solver for the solution of linear system of equations. Speedups in the range 1.4× to 7.2× were obtained in the wall-clock timings, which shows the effectiveness of the proposed strategies. Finally, as a result of the proposed strategies, the proportion of the linear solver timings in wall-clock timings of the GPU implementation reaches up to 98.9% for the finest mesh consisting of 5.1 million DOFs.

Chapter 4

Matrix-free CG Solver for Elastoplasticity using Unstructured Mesh

A fast and efficient linear solver is crucial for the performance of an elastoplastic simulation. As shown in Chapter 3, the linear solver is found to be the most time consuming step in GPU-optimized elastoplasticity solver, occupying up to 98.7% of the wall-clock timings. Hence, improving performance of the linear solver must be the sole focus of any future efforts to reduce the timings of elastoplastic simulation. Linear solvers form the core of almost all major scientific computations, and therefore attracted significant efforts from the scientific community. The iterative linear solvers are commonly used in large-scale applications for its memory efficiency and high parallelism. The main computational components of an iterative solver are sparse matrix-vector product (SpMV), vector-dot product, scalar-vector product and vector-vector addition/subtraction. Among these, SpMV is the most computationally expensive operation (Wong et al. 2015) and determine the performance of an iterative solver. The other steps are embarrassingly parallel and there exists many efficient implementations to execute on the GPU. Since SpMV is critical for performance of an iterative linear solver, its GPU implementations have been studied by many researchers (Filippone et al. 2017).

The performance of SpMV is found to be affected most by sparse formats and parallelization strategy. The selection of sparse storage formats depends on the sparsity pattern and determines the underlying data structure. Though sparse formats are essential for storage management, it introduces irregular memory access pattern detrimental to GPU performance. In addition, sparsity pattern also affects computational strategy for SpMV, highlighting the fact that there is no common strategy that guarantees best performance for all types of matrices. However, SpMV involving a sparse matrix is the only approach for those applications where generation of system matrix is necessary. For applications where generation of assembled system matrix is not explicitly required, an alternative implementation in the form of matrix-free methods exist.

Motivated by shortcomings of a conventional SpMV involving a sparse matrix, matrix-free strategies were developed. The main idea about a matrix-free solver is built over a fact that in an iterative linear solver global tangent matrix entries are not explicitly required. It is the result of SpMV with which an iterative solver works. In matrix-free FEM, the global tangent matrix is never constructed and therefore the use of sparse formats is completely avoided. The SpMV with a single large sparse matrix is replaced by its matrix-free implementation that uses many small elemental matrix-vector product. Since, elemental matrices are dense and of the same size, the matrix-free approach can provide fine level of parallelism along with regular memory access pattern suitable for a GPU. Typically, matrix-free SpMV requires more number of arithmetic operations than assembly-based SpMV. This makes matrix-free strategies less suitable for single-core computing. However, evolution of GPU as a cost-effective parallel computing tool has revived the interest in matrix-free implementation of iterative linear solvers. An added advantage of a matrix-free method is that the storage space for global tangent matrix and associated cost of assembly can be avoided. In this chapter[1], we look into the details of matrix-free strategies for all-hexahedral unstructured mesh and apply to elastoplastic simulation.

[1]The content of this chapter has been published in **Computing** (Kiran et al. 2020).

4.1 Matrix-free strategies in FEM

The discretization of a governing differential equations using FEM gives linear system of equations of the following form (see Eq. (2.27)),

$$\mathbf{K}\Delta\mathbf{u} = \mathbf{f}. \tag{4.1}$$

where $\mathbf{f}$ is the total nodal force vector. For large-scale FEM analysis, the size of matrix $\mathbf{K}$ can be very large. This leads to requirement of a large storage space in memory and large computational time to obtain solution. The solution of Eq. (4.1) using an iterative method begins with an initial guess and performs a series of algebraic operations to arrive approximate solution over a minimum number of iterations. For problems involving symmetric positive-definite global matrix, conjugate gradient (CG) solver is the most preferred iterative solver.

Algorithm 9 presents the key steps in the CG solver used in this work. It can be observed that the CG method can be entirely implemented with a series of linear algebra operations like matrix-vector product, vector-dot product, scalar-vector product and vector addition/subtraction. Among these operations, the computation of matrix-vector product is the most expensive and consequently determines the run time of the CG iterative solver. Since the global matrix $\mathbf{K}$ is sparse in nature, the matrix-vector product in line 6 of Algorithm 9 is implemented as SpMV. The computation of SpMV in assembly-based approach is implemented as

$$\mathbf{g} = (\underset{e\in\mathcal{E}}{\mathcal{A}}(\mathbf{K}^e))\mathbf{p}, \tag{4.2}$$

where $\mathbf{g}$ is the resultant vector and $\mathbf{p}$ is the vector to perform multiplication. It can be observed that elemental matrices are first assembled into a global matrix before performing multiplication. On the contrary, in matrix-free computation of SpMV, the global matrix $\mathbf{K}$ is never assembled, instead constituent elemental tangent matrices are used to obtain the result of multiplication. The matrix-free SpMV is implemented as

$$\mathbf{g} = \underset{e\in\mathcal{E}}{\mathcal{A}}(\mathbf{K}^e\mathbf{p}^e), \tag{4.3}$$

Algorithm 9 Conjugate-gradient iterative solver

Input: $\mathbf{K}$, $\mathbf{f}$, $tol, maxitr$
Output: $\mathbf{u}$

Initialize $\mathbf{u}$ to zero
$\mathbf{r} \leftarrow \mathbf{f}$
$\rho_0 = \mathbf{r}^T\mathbf{r}$
$\rho_{\text{old}} \leftarrow \rho_0$
while $t < maxitr$ **do**
    $\mathbf{g} = \mathbf{K}\mathbf{p}$
    $\alpha = \rho_{\text{old}}/(\mathbf{g}^T\mathbf{p})$
    $\mathbf{u} = \mathbf{u} + \alpha\mathbf{p}$
    $\mathbf{r} = \mathbf{r} - \alpha\mathbf{g}$
    $\rho_{\text{new}} = \mathbf{r}^T\mathbf{r}$
    $\beta = \rho_{\text{new}}/\rho_{\text{old}}$
    $\rho_{\text{old}} = \rho_{\text{new}}$
    $\mathbf{p} = \mathbf{r} + \beta\mathbf{p}$
    $t = t + 1$
    if $(\mathbf{r}^T\mathbf{r}) < (tol * tol * \rho_0)$ **then**
        break
    end if
end while

where $\mathbf{p}^e$ is the elemental sub-vector of vector $\mathbf{p}$. Here, elemental tangent matrices are first multiplied with corresponding sub-vector $\mathbf{p}^e$ and results are then assembled into the global vector $\mathbf{g}$. There are three major strategies by which a matrix-free solver can be implemented on a GPU.

1. Node-by-Node (NbN)
2. Degrees-of-Freedom -by- Degrees-of-Freedom (DbD)
3. Element-by-Element (EbE)

4.1.1 NbN strategy

In NbN strategy, the computation of matrix-vector product is done by moving through each node of the mesh. Every node has their corresponding rows in the global tangent matrix. The multiplication of each row associated with a node is done with a given vector and result are accumulated. This is represented in Fig. 4.1 by taking a 2D mesh as an example. The finite element mesh consists of quadrilateral elements where each

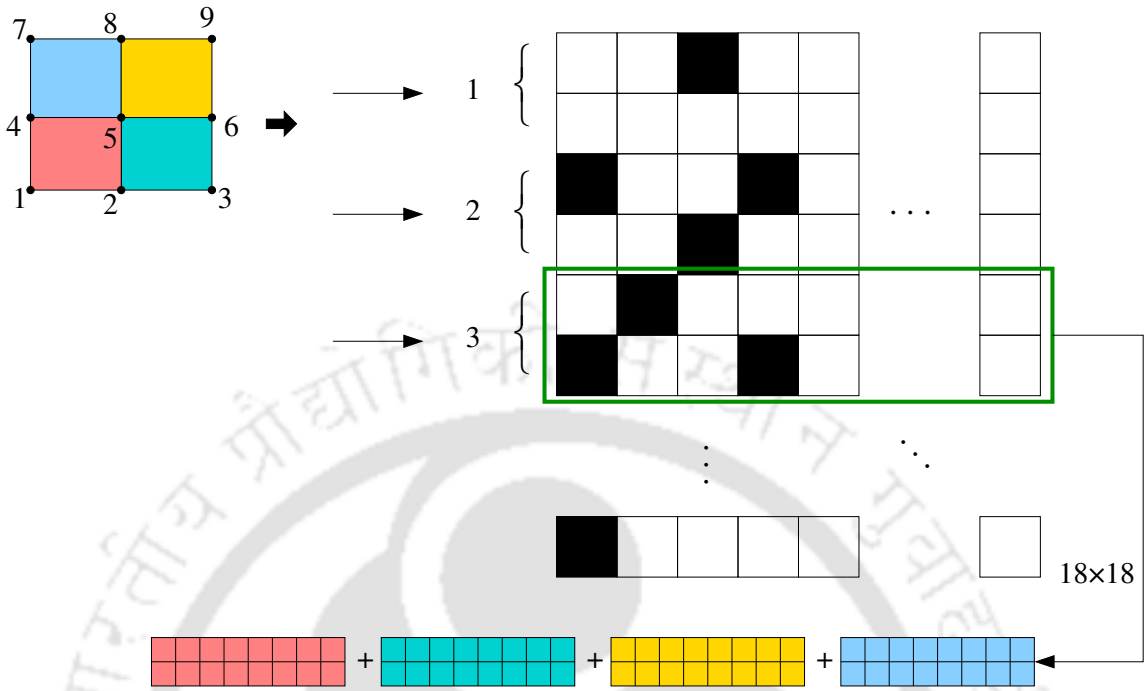


Figure 4.1: Node-by-Node matrix-free strategy.

node has two DOFs. As shown in the figure, for each node, corresponding two rows in global tangent matrix are marked with braces. The white box in global tangent matrix denotes nonzero values and the filled box denotes zero value. The multiplication of a row of assembled global tangent matrix with a multiplying vector is straightforward, and can be simply implemented like vector-dot product. However, in case of *NbN* strategy, a row of global tangent matrix is not available in assembled form. As shown in Fig. 4.1, rows corresponding to a node in the global matrix contain contributions from four elements, with each elemental contribution marked with same color as the element in the mesh. In order to assemble a row in global tangent matrix, contributions are needed from tangent stiffness matrices of elements that share the associated node. Therefore, in *NbN* strategy, multiplication is first performed with elemental contributions, and then the results are summed up to get the final value (Martínez-Frutos et al. 2015, Cai et al. 2013, Prabhune & Suresh 2020).

The *NbN* strategy can be expressed as

$$\mathbf{g}^{(n)} = \underset{e \in \mathcal{E}^{(n)}}{\mathcal{A}} (\mathbf{K}_n^e \mathbf{p}^e), \tag{4.4}$$

where $\mathcal{E}^{(n)}$ is the set of elements connected to node n, $\mathbf{K}_n^e$ is the contribution of elemental tangent matrix towards node n and $\mathbf{g}^{(n)}$ is the resultant vector for node n. Algorithm 10 shows key steps in computer implementation of NbN strategy. At first, a list of neighboring elements ($\mathcal{E}^{(n)}$) is obtained in line 2 in the form of node connectivity matrix. For each element in $\mathcal{E}^{(n)}$, an element connectivity list $\mathrm{E}^{(e)}$ is found along with local position l_index of the node. Finally, the required product is calculated in line 9 for all DOFs associated with the node.

Algorithm 10 Node-by-Node strategy.

```
1: for Node n = 1 to 𝒩 do
2:     Find Node connectivity ℰ^(n)
3:     for element e ∈ ℰ^(n) do
4:         E^(e) ← get_elementConnectivity(e)        ▷ Extract the element connectivity
5:         l_index ← get_localPosition()
6:         p^(e) ← p(E^(e))                          ▷ Obtain the sub-vector for multiplication
7:         for i = 1 to n_dof do
8:             for j = 1 to e_dof do
9:                 val[i]+ = K^e[3 * l_index + i][j] * p^e[j]
10:            end for
11:        end for
12:    end for
13: end for
```

GPU parallelization is done over the nodes of the mesh. Single thread is assigned to do computation for one node. The node connectivity array and local position array can be reordered to access in coalesced manner. The elemental stiffness matrices are read from strided locations in global memory and hence cannot be coalesced. The element connectivity matrix is arranged column-wise so that global memory transaction is minimized.

It can be observed that each node performs its computation independently and therefore the problem of data race conditions (NVIDIA Corporation (2022)) does not arise. This turns out to be a major advantage of NbN strategy, since overhead associated with synchronization mechanism like coloring can be avoided.

In case of unstructured mesh, each of the nodes can have different number of neighboring elements. This leads to an unequal amount of work load distribution on threads. A GPU warp (NVIDIA Corporation (2022)) remains active as long as any of its threads is working. In case of unbalanced work load, some threads remain idle, while others are

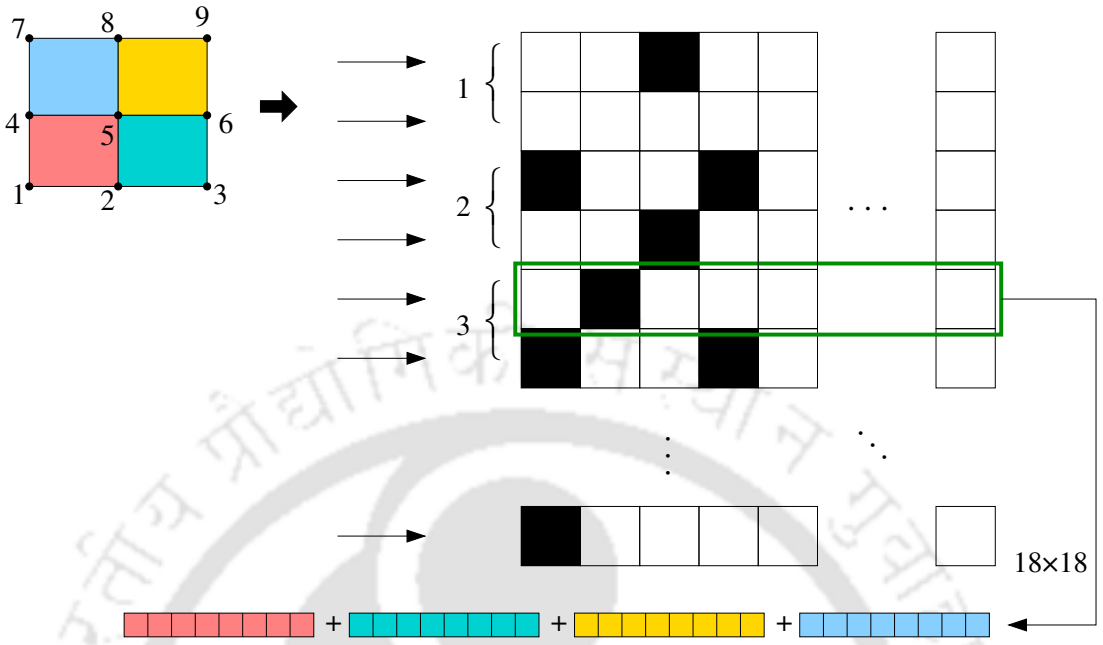


Figure 4.2: DOF-by-DOF matrix-free strategy.

active. This is not desirable for an efficient utilization of GPU resources and leads to a major disadvantage of GPU-based *NbN* strategy.

4.1.2 *DbD* strategy

The *DbD* strategy performs the matrix-vector multiplication by moving through each DOF of the system. Here, computation for each DOF associated with a node is seen as independent task. This is illustrated in Fig. 4.2 with the help of a 2D mesh having two DOFs per node. As can be seen, a thread is assigned to each row of the global tangent matrix. In *DbD* strategy, computation of matrix-vector product is implemented as (Martínez-Frutos & Herrero-Pérez 2015)

$$\mathbf{g}^{(u)} = \underset{e \in \mathcal{E}^{(u)}}{\mathcal{A}} (\mathbf{K}_u^e \mathbf{p}^e), \tag{4.5}$$

where $\mathbf{K}_u^e$ is the contribution of elemental tangent matrix toward DOF u, and $\mathcal{E}^{(u)}$ is the set of all elements connected to u. The *DbD* strategy is implemented in a way similar to Algorithm 10 except that the computation is performed inside single loop over DOFs associated with an element (e_{dof}). Single thread per DOF assignment is used to perform

computation. The input data structure remains identical as Algorithm 10, but now the same data is read by as many threads as the value of n_{dof}. Each thread performs its own computation and accumulates the result into an array in a coalesced manner. This strategy is also known as Row-by-Row solution method (van Rietbergen et al. 1996).

In terms of GPU implementation, *DbD* strategy has finer level of granularity than *NbN* strategy. However, the input data requirement remains the same as that of *NbN* strategy. Moreover, since the same data is required for all threads associated with a particular node, either it can be read redundantly from global memory or can be shared among threads using the shared memory. The limited size of shared memory restricts its use to very few cases, and generally, data is read redundantly from global memory. The *DbD* strategy also suffers from the same load imbalance problem found in case of *NbN* strategy.

4.1.3 *EbE* strategy

In the *EbE* strategy, the computation of matrix-free SpMV is done by moving through each element of the mesh. The steps involved in *EbE* strategy is expressed by decomposing Eq. (4.3) into three components as

$$\mathbf{p}^e = \mathcal{A}^{-1}(\mathbf{p}), \tag{4.6}$$

$$\mathbf{g}^e = \mathbf{K}^e \mathbf{p}^e, \tag{4.7}$$

$$\mathbf{g} = \mathcal{A}(\mathbf{g}^e), \tag{4.8}$$

where $\mathbf{g}^e$ is the result of matrix-vector product at elemental level. In the first step, a sub-vector $\mathbf{p}^e$ is obtained for each element e by applying inverse assembly operator over global vector $\mathbf{p}$. Next, a dense matrix-vector product is performed by multiplying elemental tangent matrices with $\mathbf{p}^e$ and the result is stored as $\mathbf{g}^e$. Finally, elemental resultant vectors are assembled into the global vector $\mathbf{g}$. This is illustrated in Fig. 4.3. The first and second steps are embarrassingly parallel and can be easily implemented on GPU. However, the third step involves data race conditions due to parallel assembly

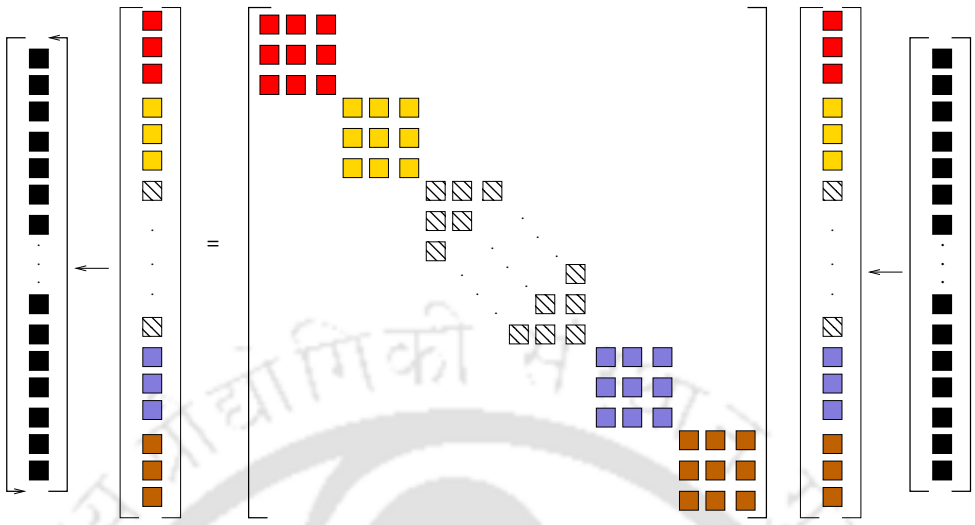


Figure 4.3: Element-by-Element matrix-free strategy.

into a global vector.

Algorithm 11 shows the computer implementation of *EbE* strategy. For each element, multiplying sub-vector $\mathbf{p}^e$ is obtained with the help of element connectivity. The multiplication of elemental tangent matrix with vector $\mathbf{p}^e$ is done inside double loops over elemental DOFs, as shown in line 6. In GPU implementation, computation for each element is performed in parallel.

Algorithm 11 Element-by-Element matrix-free strategy.

1: **for** element $e = 1$ to $\mathcal{E}$ **do**
2: $\quad \mathrm{E}^{(e)} \leftarrow$ get_elementConnectivity(e)
3: $\quad \mathbf{p}^{(e)} \leftarrow \mathbf{p}(\mathrm{E}^{(e)})$ ▷ Obtain the sub-vector for multiplication
4: $\quad$ **for** $i = 1$ to e_{dof} **do**
5: $\quad\quad$ **for** $j = 1$ to e_{dof} **do**
6: $\quad\quad\quad \mathbf{val}[i] += \mathbf{K}^e[i][j] * \mathbf{p}^e[j]$
7: $\quad\quad$ **end for**
8: $\quad$ **end for**
9: **end for**

There are three prominent ways of distributing work load among threads, which are briefly explained below.

1. Single thread per element: In single thread per element approach (Kiss, Gyimóthy, Badics & Pávó (2012)), one thread is responsible for reading input data, computing elemental matrix-vector product and accumulating calculated value to the resultant

vector. This approach is the simplest to implement. However, each element gets amount of on-chip memories (shared memory and register) corresponding to a thread only. Therefore, this approach suffers from poor utilization of fast on-chip memories.

2. Single thread per node: The single thread per node approach allocates as many threads to an element as the number of nodes (Ratnakar et al. 2021). Each thread perform computation for all DOF associated with the node. This approach provides more on-chip memory than single thread per element strategy.

3. Single thread per DOF: The finest level of granularity is achieved in single thread per DOF approach. Here, the number of threads equal to DOF associated with an element is allocated (Martínez-Frutos et al. 2015, Pikle et al. 2018). The elemental matrix-vector product is decomposed into several inner-vector product corresponding to each row of the matrix. Each thread is assigned to do computation for one inner-vector product. This approach provides the highest amount of on-chip memory per element.

After the elemental matrix-vector product is computed, it needs to be assembled into the global vector $\mathbf{g}$. Each non-zero entry in the global vector corresponds to a DOF of the system. Each DOF is shared among multiple elements of the mesh. During parallel assembly, multiple elements tend to accumulate their calculated value to the same location in global vector, simultaneously. Such operations lead to the data race condition, where the outcome of an operation remains undefined. In this work, the data race condition is avoided by the use of mesh coloring (Cecka et al. 2011, Kiran et al. 2018). The coloring method is a robust and popular synchronization technique that handles race conditions by separating a mesh into disjoint sets of elements. Elements belonging to different sets do not have any node in common, and therefore, assembly can be performed without conflict.

4.2 Proposed $EbE_{\mathbf{sym}}$ strategy

All matrix-free SpMV strategies available in the literature use full elemental tangent matrices for the computation of the matrix-vector product. The elemental tangent matrices

obtained in FEM are symmetric for most of the problems. Implementing matrix-vector product using symmetric part of elemental matrices can significantly reduce the storage requirement as well as data transfer during the computation. On memory bound architectures like GPU, reduction in data requirement is expected to improve the performance of a compute kernel, substantially. So far, matrix-free approaches have not been implemented on GPU using only symmetric part of the elemental matrices.

The proposed strategy uses symmetry property of elemental tangent matrices to compute matrix-vector product in Eq. (4.7) using only lower or upper triangular part of the matrix. The matrix-vector product in the proposed strategy is implemented as

$$\mathbf{g} = \underset{e \in \mathcal{E}}{\mathcal{A}} (\mathbf{K}^e_{\text{sym}} \mathbf{p}^e), \tag{4.9}$$

where $\mathbf{K}^e_{\text{sym}}$ is the symmetric part of elemental tangent matrix. Since the proposed strategy is based on EbE strategy, it is referred to as EbE_{sym} in this thesis, and can be represented in the graphical form as shown in Fig. 4.4. It can be seen that the dense matrix-vector product required in matrix-free solvers is replaced by dense symmetric matrix-vector product (SYMV), which requires only symmetric part of elemental matrices. In the proposed strategy, all steps mentioned in Fig. 4.4 is implemented as single computational kernel with coloring approach to avoid data race conditions.

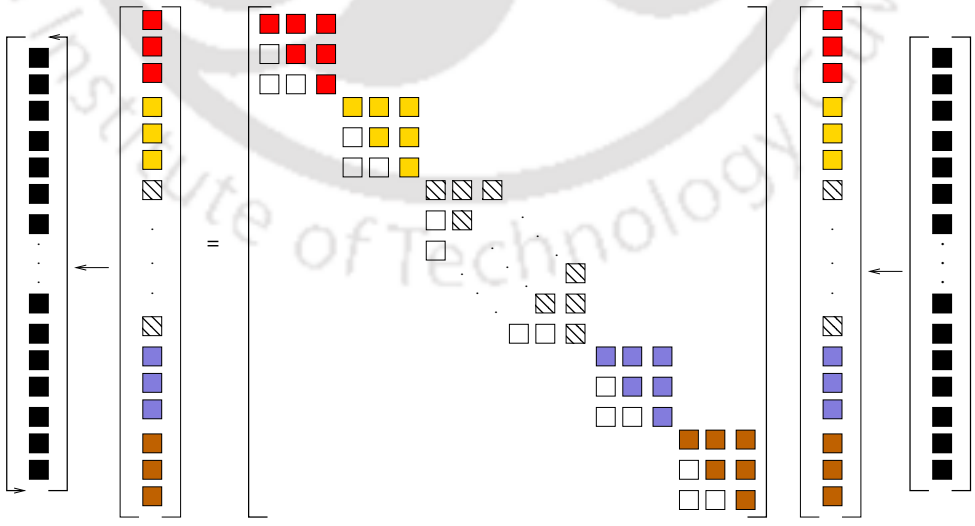


Figure 4.4: Element-by-Element matrix-free strategy using only symmetric part of elemental tangent matrix.

The computation of a SYMV operation can be performed by following Algorithm 12.

The multiplication of each row with vector $\mathbf{p}^e$ is done by looping over the total number of rows (e_{dof}), where the computation is performed first (line 3) for the upper triangular part of the matrix. In line 7 of Algorithm 12, the computation of missing symmetric part (lower triangular part) is done, where indices k and i to the matrix $\mathbf{K}^e$ always refer to the values in upper triangular part. This shows that the matrix-vector computation can be implemented by storing only upper or lower triangular part of the elemental tangent matrix.

Algorithm 12 Computation of symmetric matrix-vector product (SYMV)

```
1: for i = 1 to e_dof do
2:     for j = i to e_dof do                ▷ Computation for symmetric part
3:         g^e[i] += K^e[i][j] * p^e[j]
4:     end for
5:     if (i ≠ 1) then                      ▷ Computation for missing symmetric part
6:         for k = (i − 1) to 1 do
7:             g^e[i] += K^e[k][i] * p^e[k]
8:         end for
9:     end if
10: end for
```

For GPU implementation, the storage of symmetric part of the matrix can be done in any suitable format which facilitates uniform memory access pattern during the computation. However, the same storage format may not be suitable for both the symmetric part and missing symmetric part. As shown in step 7 of Algorithm 12, values of $\mathbf{K}^e$ are access from stored symmetric part in a strided and nonuniform manner. Such kind of access pattern wastes the memory bandwidth of the GPU and consequently degrade the performance. The optimization of memory access pattern for SYMV operation on GPU is suggested by multiple authors in (Nath et al. 2011, Abdelfattah et al. 2012, Charara et al. 2019). These studies however discuss the strategies to compute SYMV for moderate to large size matrices. The approaches available in the literature are either not applicable or not optimal for small size matrices, generally found in low order FEM. Moreover, in the current work, computation has to be performed in a batch for millions of elements present in the finite element mesh. Since matrix-vector product has very low arithmetic intensity, efficient handling of memory overhead becomes indispensable for batch implementation of SYMV. For better performance on GPU, it thus becomes extremely important to minimize the data transfer and use coalesced and localized mem-

ory access pattern. The proposed EbE_{sym} strategy addresses all these issues by adopting a novel data structure, which ensures coalesced memory access, while storing only symmetric part of the elemental matrices. In order to obtain the best performance, the EbE_{sym} strategy seeks to make an efficient use of register cache by using CUDA shuffle instruction. This not only helps in relaxing shared memory size restrictions but also avoids data movement to and fro the shared memory. Single thread per node assignment similar to EbE strategy (Section 4.1.3) is used to achieve balanced workload distribution. Also, each thread performs its task independently so that no synchronization barrier is required.

Kernel design and data structure

In the proposed strategy, elemental tangent matrix is divided into a number of sub-matrices as shown in Fig. 4.5. The figure shows nonzero entries in upper triangular part of an elemental tangent matrix for 4-noded quadrilateral element with $n_{dof} = 2$. Each node is associated with as many rows and columns in the matrix as the value of n_{dof}. The size of sub-matrices is kept equal to n_{dof} and it contains values corresponding to one node in row and one node in column. For example, the following sub-matrix contains all the computed values for node 1 in row and node 2 in column.

$$\begin{bmatrix} \mathrm{D}_1 & \mathrm{E}_1 \\ * & * \end{bmatrix}$$

Depending on the position in the matrix, the sub-matrices are categorized into two groups: diagonal and off-diagonal. The diagonal group contains all sub-matrices lying on the diagonal of the elemental matrix, such as

$$\begin{bmatrix} \mathrm{A}_1 & * \\ * & \mathrm{B}_1 \end{bmatrix}, \begin{bmatrix} \mathrm{A}_2 & * \\ * & \mathrm{B}_2 \end{bmatrix}, \text{etc.}$$

The off-diagonal group contains sub-matrices that are not unique. These sub-matrices appear in symmetric part also. In case of the diagonal group, all sub-matrices are

associated with only one node number (the rows and column nodes are same). In case of the off-diagonal group, two such numbers exist, that is, one associated with the row and other with the column. Therefore, each sub-matrix is identified with a nodal index of the form $K^e\{n, m\}$, which represents sub-matrix at n^{th} node in row and m^{th} node in column. The row node number is used to find global DOF to store result of multiplication, whereas the column node number is used to find global column indices of vector $\mathbf{p}$ to perform multiplication.

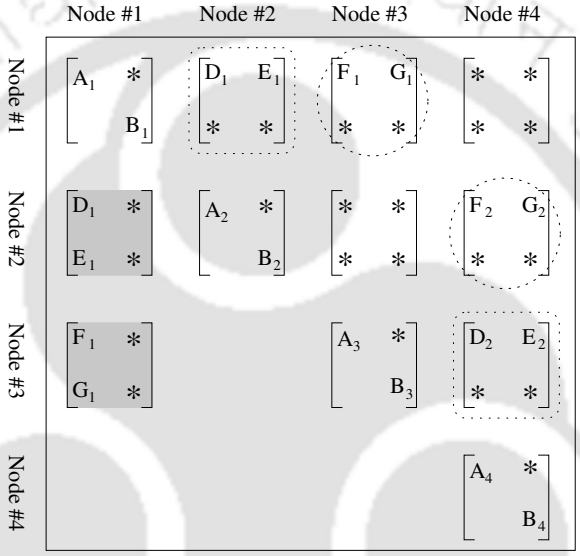


Figure 4.5: Organization of elemental stiffness matrix for a 4-noded quadrilateral element with two DOF per node.

The computation of symmetric matrix-vector product is divided into two stages. In the first stage, computation for the diagonal group is performed, whereas in the second stage, multiplication for the off-diagonal group is done.

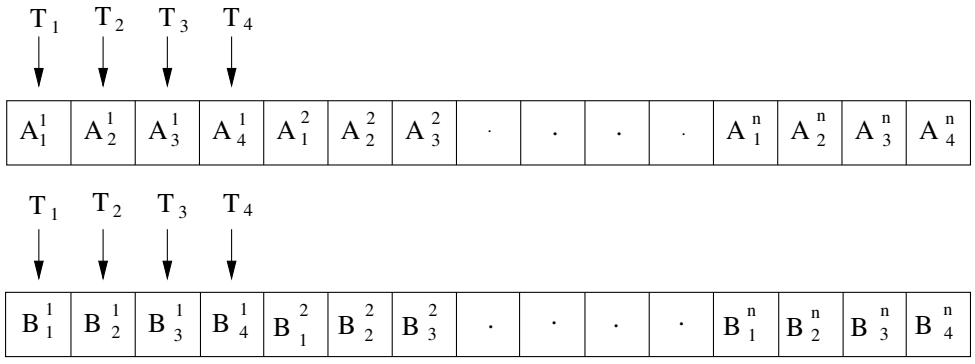


Figure 4.6: Data access pattern for diagonal group.

The sub-matrices in the diagonal group are unique. Since computation is performed in a node-wise manner, these sub-matrices contribute to the matrix-vector multiplication results for their respective node number. A thread assigned to each node reads all the unique entries from its sub-matrix and perform multiplication with $\mathbf{p}^e$ vector. The data access pattern for diagonal group is shown in Fig. 4.6. It shows four threads accessing the values marked as A and B (also shown in Fig. 4.5) from four sub-matrices of the diagonal group. Here, in Fig. 4.6, superscript represents the element number, whereas the subscript denotes the sub-matrix position in the diagonal group. The other entries of a sub-matrix are accessed in a similar way. In order to achieve coalesced access for a warp, data for other elements are stored side by side. Once these values are read, they get multiplied by the given vector and stored in the shared memory. Here, each thread uses its global node number to read values from vector $\mathbf{p}$ through read-only cache. The data access from $\mathbf{p}$ vector is not coalesced.

The off-diagonal entries in a symmetric matrix are not unique. The transpose of sub-matrices in the off-diagonal group can be obtained if row and column nodes are interchanged, as shown in Fig. 4.5. It can be seen that the sub-matrix located at $K^e\{1,2\}$ appear in its transposed form at $K^e\{2,1\}$. This implies that the same sub-matrix can be used to perform computation for both node 1 and node 2. Similarly, the sub-matrix at position $K^e\{1,3\}$ can be used for both node 1 and node 3. Thus, for a 4-noded quadrilateral element, the computation for two sub-matrices can be performed simultaneously. The computation for the off-diagonal group is implemented such that each thread is assigned with equal amount of workload. As shown in Fig. 4.5, the sub-matrices having the same type of enclosing can be processed at the same time. If chosen otherwise, any one thread can remain idle and others may have more work to do.

Since the computation of matrix-vector product uses only symmetric part of the matrix, the sub-matrices in the missing part (lower triangular part in Fig. 4.5) must be obtained separately or shared between the two threads. Due to the limited size of shared memory, values in sub-matrices are read redundantly from the global memory. However, data is arranged in a way that it results into a broadcast. The broadcast from global memory is although slower than shared memory, it has a lower overhead than reading values separately. The data access pattern for off-diagonal entries is shown in Fig. 4.7. Here, D denotes the corresponding values in sub-matrices with same type of enclosing

(refer to Fig. 4.5) in which subscript indicates the sub-matrix within an element and superscript indicates the element number. It can be seen that threads T_1 and T_2 assigned to node 1 and 2, read the same D_1^1 value, whereas threads T_3 and T_4 read D_2^1 value, which is used to perform computation for node 3 and node 4. The F values (see Fig. 4.5) are stored and read in a similar way, except now the values are required by different set of threads. The data for all elements are kept beside each other to enable coalesced access for a warp.

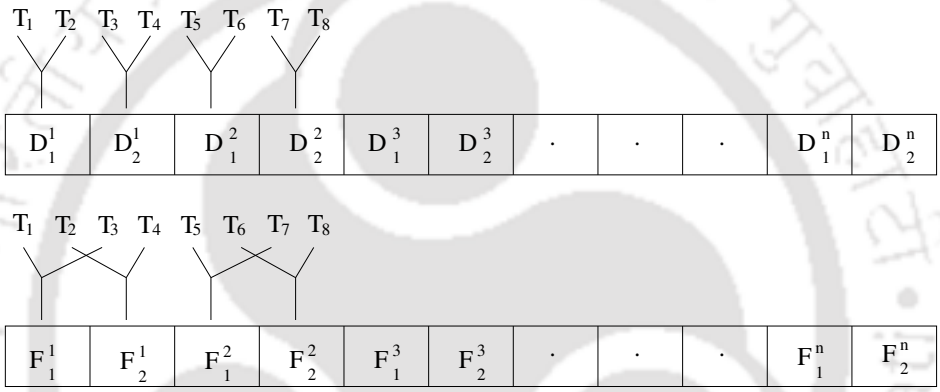


Figure 4.7: Data access pattern for off-diagonal group.

Once a value from elemental tangent matrix is read, it is multiplied with a particular value from vector $\mathbf{p}^e$. The $\mathbf{p}^e$ vector is extracted from $\mathbf{p}$ by using the column indices of the values from elemental matrix. In FEM, column indices for an elemental tangent matrix can be obtained by global node numbers and n_{dof}. In particular, the column node numbers of each sub-matrix can be used to obtain column indices of its entries. It can be observed from Fig. 4.5 that threads working over a sub-matrix either contain row node number or column node number of the sub-matrix. The row node number becomes column node number for a sub-matrix after it gets transposed. Thus, the global node number of two threads can be interchanged to get column node number of the sub-matrix. This is achieved in the proposed strategy by using warp-shuffle instruction. Using warp shuffle feature, the proposed strategy prevents the use of shared memory as well as global memory access. The warp shuffle feature has been found to be more faster than shared memory and leads to better utilization of register cache (Kiran et al. 2018).

4.2.1 3D implementation

In 3D, the proposed EbE_{sym} strategy is implemented for linear hexahedral element with three DOFs per node. For this element, the size of elemental tanget matrix is 24 and total number of nonzero values is 576. The symmetric part of the elemental tangent matrix consists of 300 values only, reducing the storage requirement by 1.92×. Here, the GPU implementation of EbE_{sym} strategy considering DOF-based thread allocation is explained.

Considering DOF-based thread allocation, 24 threads are assigned to perform computation of matrix-vector product for one element. Each thread computes an inner vector product using one row of elemental tangent matrix and a sub-vector from the multiplying vector **p**. In the symmetric part of elemental tangent matrix, each nonzero value except those lying on the diagonal contribute to two inner vector products corresponding to two rows. Therefore, each non-diagonal value is accessed by a pair of threads having thread numbers equal to row and column indices in the matrix. In the proposed strategy, symmetric part of elemental tangent matrix is divided into diagonal and off-diagonal groups as shown in the Fig. 4.8. Here, values in symmetric part of elemental tangent matrix are denoted by asterisk symbol with some of them labelled by letters to explain data arrangement. The values lying in the gray region are considered part of the diagonal group and values lying in the colored region belong to off-diagonal group.

In the diagonal group, values lying on the main diagonal of the elemental tangent matrix are denoted by A_1, A_2, ..., A_8, A_9, ..., A_{24} and stored in the consecutive memory address to access in coalesced manner. These values contribute to inner vector product of only one row and therefore accessed by one thread only. The memory layout for other values depends on combination of two threads that access a value together. For example, threads 1 and 2 access B_1, threads 3 and 4 access B_2, and so on. The storage and access pattern of value labeled as B and C is shown in Fig. 4.9. As can be seen, 24 threads access 12 values from consecutive memory locations, which is desirable for better performance on GPU. The data for other elements are stored side-by-side to increase locality. The data in off-diagonal group is divided into multiple sub-matrices as shown by colored boxes in Fig. 4.8. Each sub-matrix of size 4×4 is accessed by 8 threads and all sub-matrices with the same color is accessed simultaneously. The position of sub-matrices with the

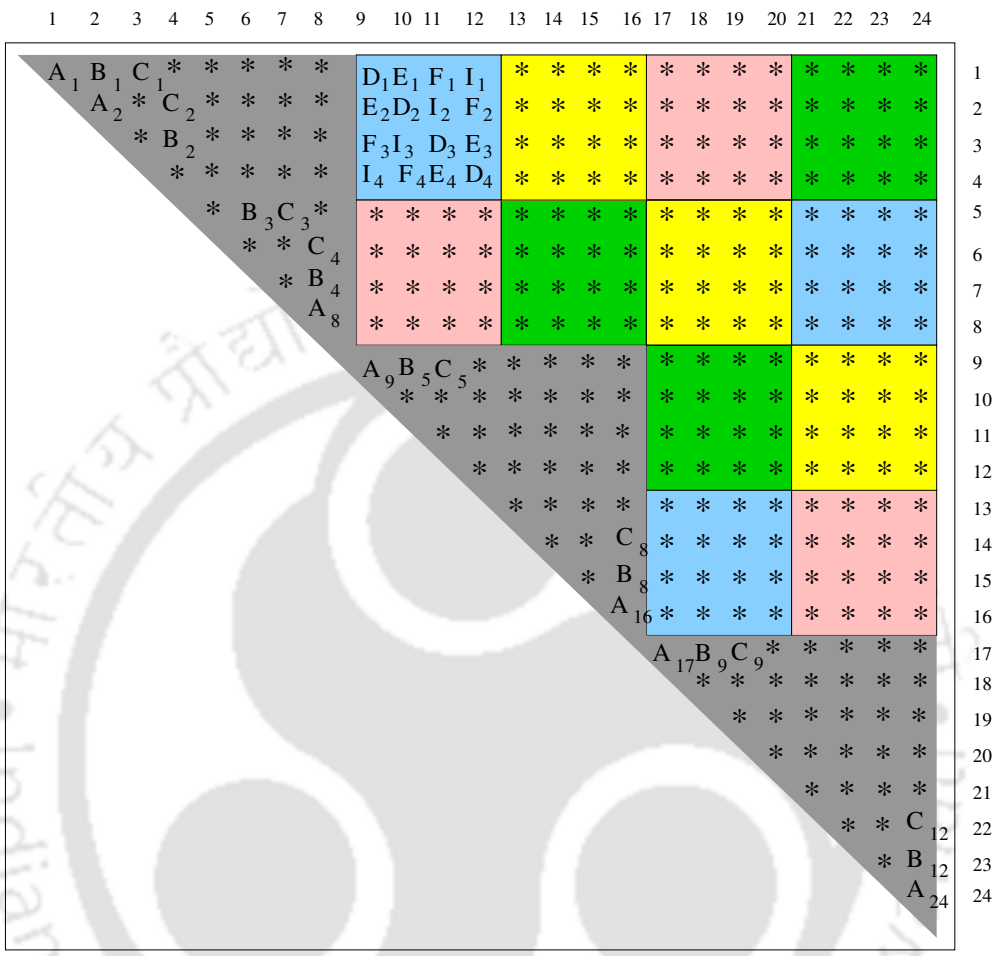


Figure 4.8: Organization of elemental tangent matrix for 8-noded hexahedral element for EbE_{sym} strategy.

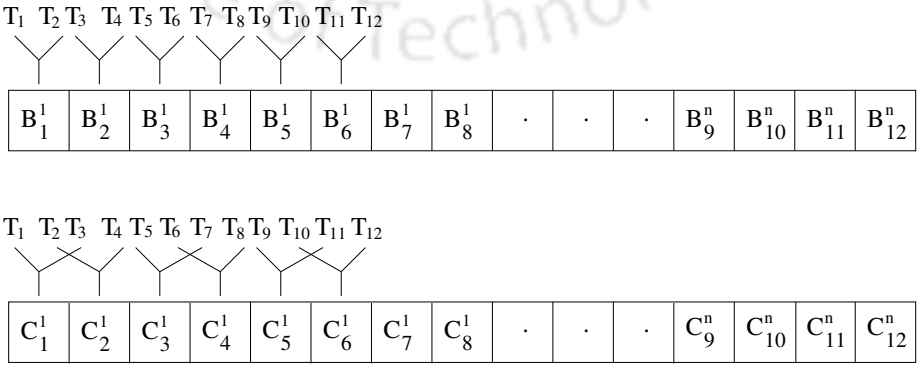


Figure 4.9: Data access pattern for diagonal group.

same color is chosen in a way that no threads remain idle and equal work distribution is achieved. Like diagonal group, data access pattern in off-diagonal group is also dictated by the combination of threads that access a value together. For example, threads 1 and 9 access D_1, threads 2 and 10 access D_2, threads 1 and 11 access F_1, and so on. Figure 4.10 shows the storage and access pattern of values labelled as D. Here, 12 values for each element(four from each sub-matrix) are accessed by 24 threads from consecutive memory locations. The other values are stored in the same way.

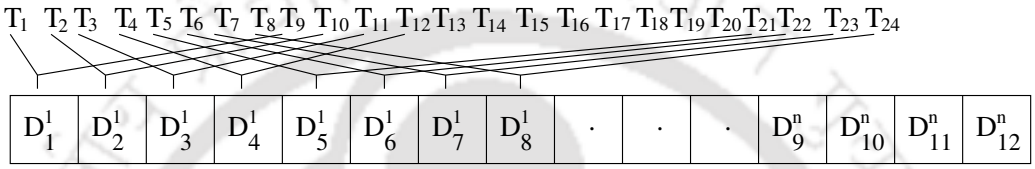


Figure 4.10: Data access pattern for off-diagonal group.

4.3 Results and Discussion

The performance of the proposed EbE_{sym} strategy is evaluated by solving both 2D and 3D problems. In 2D, elasticity and steady-state heat conduction problems are solved. The elasticity equation is solved over cantilever and L-shaped beam, and the steady-state heat equation is solved over a plate. In 3D, elastoplasticity equation is solved over L-bracket and plate with multiple holes problems. Further, the performance of the proposed strategy is compared with the existing matrix-free methods discussed in Section 4.1. Coloring method is used to handle the data race conditions associated with *EbE* and EbE_{sym} strategies. However, such data race condition is not observed with *NbN* and *DbD* strategies. It is to be noted that the proposed EbE_{sym} strategy can be used with any other synchronization techniques as well. The hardware for the numerical experiments remains the same as given in Section 3.2.

4.3.1 Elasticity problems

The governing equation for the elasticity problems is given in Section 2.1. The elasticity equation is solved for cantilever beam and L-shaped beam under plane stress condition

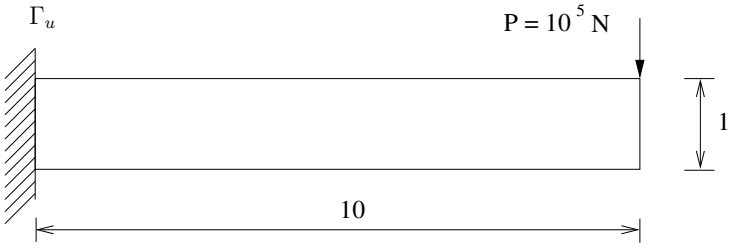


Figure 4.11: A 2D cantilever beam with end load. All dimensions are in meters (m).

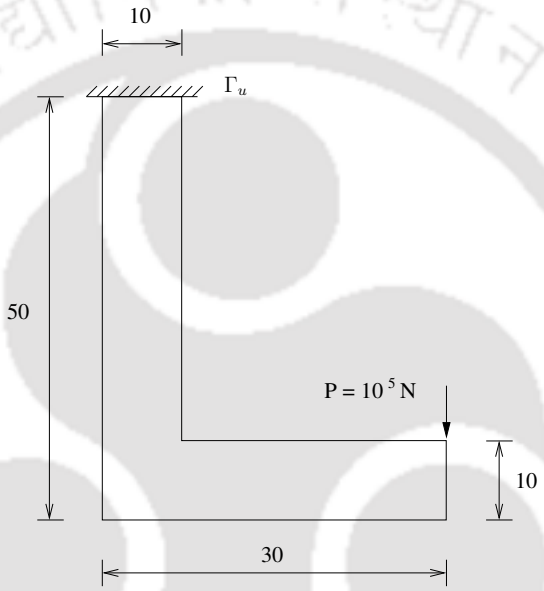


Figure 4.12: L-shaped beam. All dimensions are in meters (m).

and linear strain-displacement relation. The problem geometry along with dimensions and boundary conditions are shown in Figs. 4.11 and 4.12. The material properties are taken as: Young's modulus $E = 210$ GPa and Poisson's ratio $\nu = 0.3$. The domains are discretized with 4-noded quadrilateral elements having two degrees of freedom per node. The problems are solved for different level of mesh refinement to evaluate the performance at various workload. Tables 4.1 and 4.2 present the mesh with different number of elements and corresponding degrees of freedom for 2D cantilever beam and L-shaped beam, respectively.

Figure 4.13 shows the comparison of kernel timings for different matrix-free strategies applied to elasticity problems. Here, kernel time is referred to as the execution time of CUDA kernel in one iteration of CG solver. The *NbN* strategy consumes the highest amount of kernel time for both cantilever beam and L-shaped beam. It also has the

Table 4.1: Finite element mesh for 2D cantilever beam.

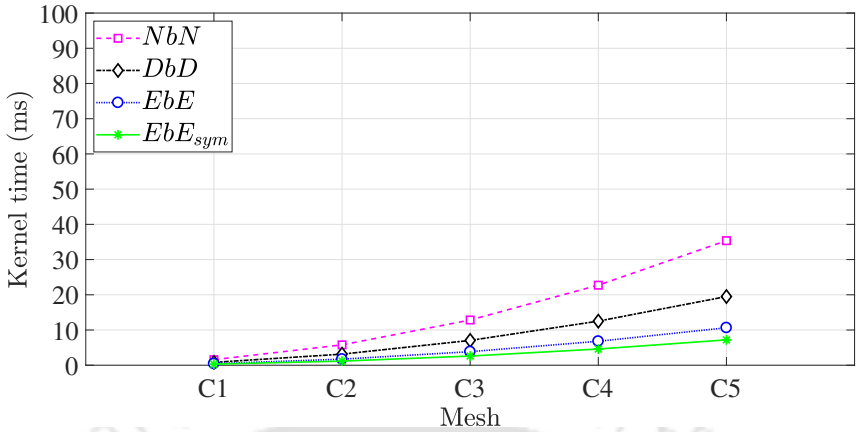
Mesh	Elements	Nodes	Degrees of freedom
C1	100,000	101,101	202,202
C2	400,000	402,201	804,402
C3	900,000	903,301	1,806,602
C4	1,600,000	1,604,401	3,208,802
C5	2,500,000	2,505,551	5,011,002

Table 4.2: Finite element mesh for L-shaped beam.

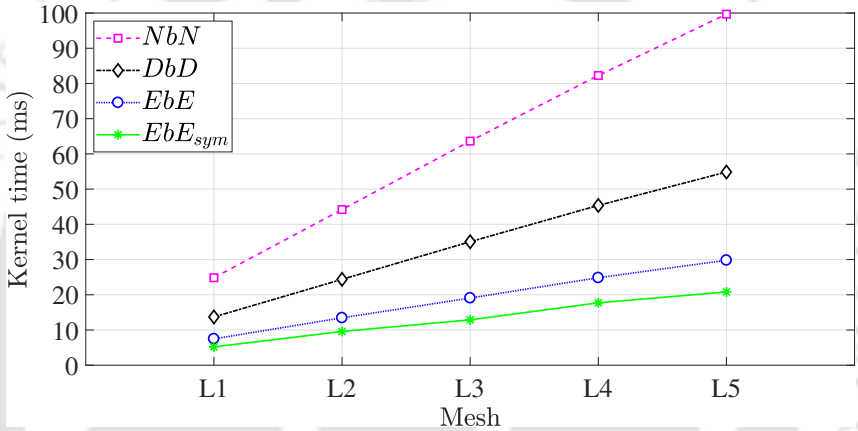
Mesh	Elements	Nodes	Degrees of freedom
L1	1,750,000	1,754,001	3,508,002
L2	3,112,889	3,118,224	6,236,448
L3	4,480,000	4,486,401	8,972,802
L4	5,783,967	5,791,240	11,582,480
L5	7,000,000	7,008,001	14,016,002

highest amount of data requirement compared to all other strategies. With the same data structure, the *DbD* strategy achieves better timings than *NbN* by just increasing the granularity of computation. The redundant access of data in case of *DbD* does not seem to have much overhead as the values are broadcasted to n_{dof} threads from the global memory. Also, the access to elemental matrix requires less number of transactions as compared to *NbN* strategy as more number of threads now access the same matrix. However, in both *NbN* and *DbD* strategies, gather operation is performed to read elemental matrices in an uncoalesced manner. The elemental matrices constitute the largest amount of data that a matrix-free solver needs to access. As evident from less kernel time of *EbE* strategy (Fig. 4.13) compared to *NbN* and *DbD* strategies, the uncoalesced access to elemental matrices has a large impact on the performance. In *EbE* strategy, the elemental matrices are accessed in coalesced manner. Apart from better memory access pattern, the *EbE* strategy has the finest level of granularity, less data requirement and equally distributed workload on each of the computational threads. With all these characteristics, the *EbE* strategy overcomes the overhead associated with race conditions handling and achieves the least kernel time among the existing matrix-free strategies.

The proposed EbE_{sym} strategy outperforms all other strategies in every test problems



(a) Kernel time for 2D cantilever beam.

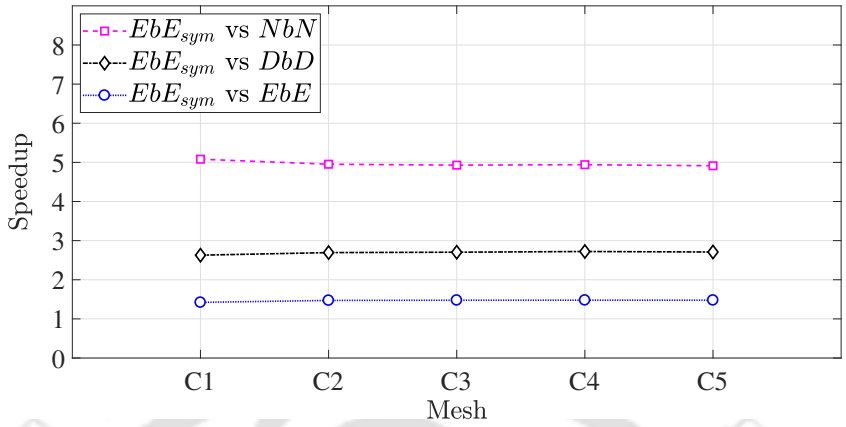


(b) Kernel time for L-shaped beam.

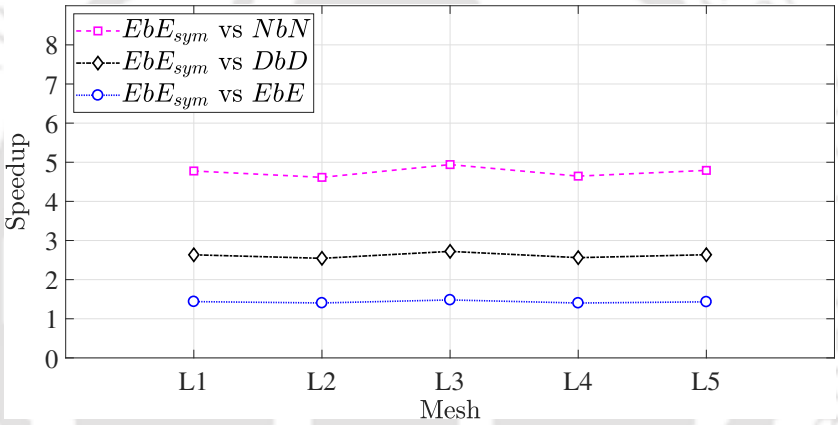
Figure 4.13: Comparison of kernel timings by matrix-free strategies.

(refer to Fig. 4.13). Since EbE_{sym} strategy inherits the major characteristic of *EbE*, better performance compared to *NbN* and *DbD* strategies is expected. However, superior performance compared to *EbE* strategy can be mainly attributed to the reduction in data movement due to use of only symmetric part of the elemental matrices. In elasticity problems, 4-noded quadrilateral elements with two DOF per node is used, which gives elemental matrix of size 8×8. The implementation and optimization of matrix-vector product for such a smaller size matrix in batch mode is extremely challenging, as it involves very low arithmetic load compared to the required amount of data movement. The proposed EbE_{sym} strategy achieves better kernel time compared to *EbE* strategy due to a unique data structure that ensures localized and coalesced access pattern using only symmetric part of the elemental matrices. The reduction in data requirement helps in maintaining a higher computation to data movement ratio than *EbE* strategy, which

is favourable for GPU implementation.



(a) Speedup for 2D cantilever beam.



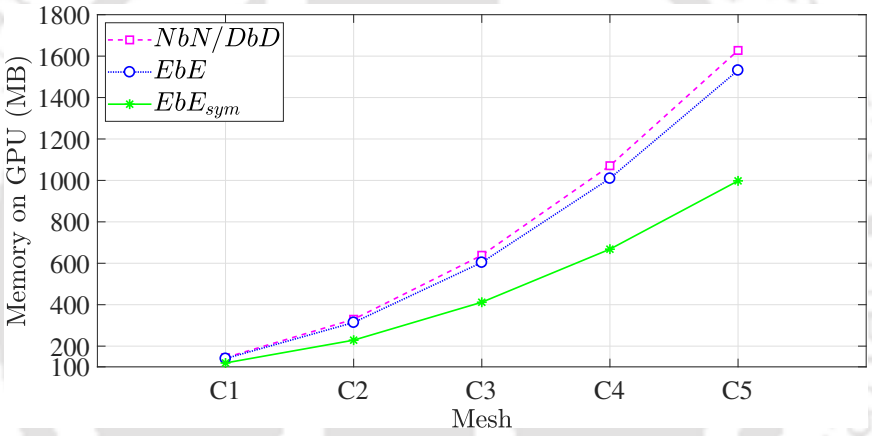
(b) Speedup for L-shaped beam.

Figure 4.14: Speedup achieved by EbE_{sym} strategy over existing matrix-free strategies.

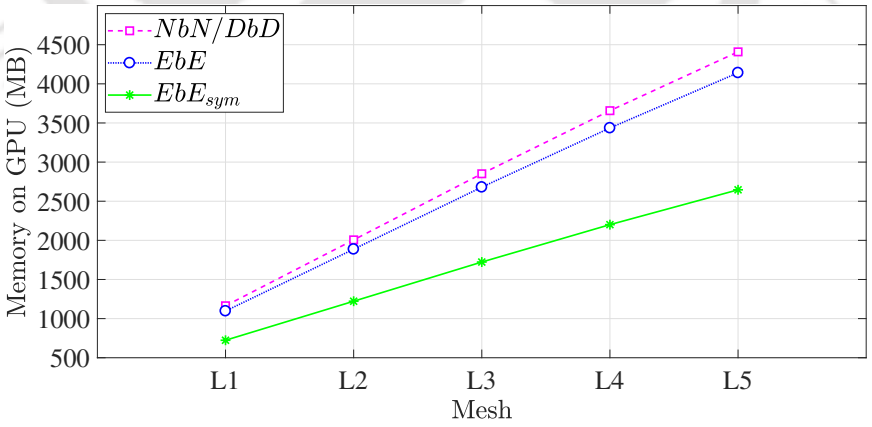
The speedup obtained by the EbE_{sym} strategy over other strategies for matrix-free solver is shown in Fig. 4.14. In both problems, a consistent speedup is observed which suggests that the EbE_{sym} strategy is able to scale well with increasing problem size. With respect to the *NbN* strategy, approximately 5× speedup is observed for both cantilever and L-shaped beam problems. In the case of the *DbD* strategy, approximately 2.8× speedup is observed for elasticity problems. With respect to the *EbE* strategy, 1.4× speedup is observed.

Figure 4.15 shows the amount of GPU memory occupied by different strategies as a function of problem size. It can be observed that the proposed EbE_{sym} matrix-free strategy consumes the least amount of GPU memory for SpMV. This suggests that a much larger problem can be solved by the proposed strategy on a given GPU hardware in a less amount of time. The *NbN* and *DbD* strategies occupy the highest amount of

memory in all the numerical problems. This is due to the dependency of these strategies on arrays like node connectivity and local position of nodes in each elements, in addition to the elemental connectivity and elemental tangent matrices. The *EbE* strategy only stores elemental connectivity and elemental tangent matrices on GPU, and therefore require lesser memory than *NbN* and *DbD* strategies. The least amount of memory consumption by EbE_{sym} strategy is due to the storage of only symmetric part of elemental matrices.



(a) GPU memory utilization for 2D cantilever beam.



(b) GPU memory utilization for L-shaped beam.

Figure 4.15: GPU memory utilization by various strategies.

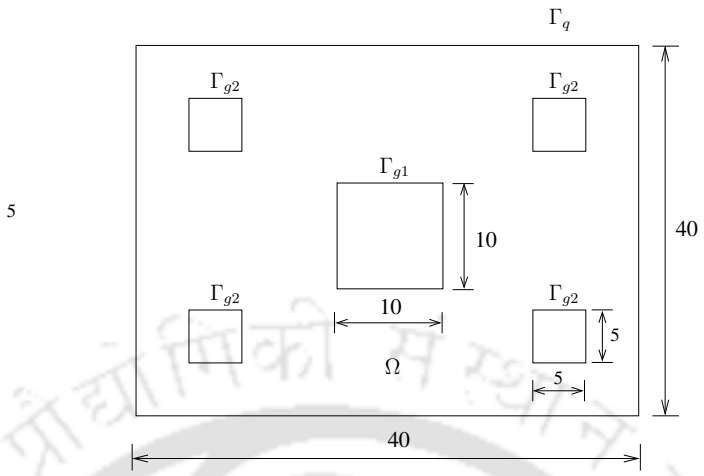


Figure 4.16: A plate with multiple holes. All dimensions are taken in meters (m).

Table 4.3: Finite element mesh for heat conduction problem.

Mesh	Elements	Degrees of freedom
H1	9,84,681	9,88,734
H2	1,938,537	1,944,226
H3	3,048,540	3,055,672
H4	4,215,044	4,223,429
H5	6,240,237	6,250,435

4.3.2 Steady-state Heat conduction

The following steady state heat conduction equation is solved for a plate with multiple holes as shown in Fig. 4.16.

$$\begin{aligned} \nabla \cdot (\boldsymbol{\kappa} \cdot \nabla T(\boldsymbol{x})) &= f(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega, \\ T(\boldsymbol{x}) &= g(\boldsymbol{x}), \quad \boldsymbol{x} \in \Gamma_g, \\ \boldsymbol{n}(\boldsymbol{x}) \cdot \boldsymbol{\kappa} \cdot \nabla T(\boldsymbol{x}) &= 0, \quad \boldsymbol{x} \in \Gamma_q, \\ \Gamma_g &= \Gamma_{g1} \cup \Gamma_{g2}, \end{aligned} \tag{4.10}$$

Here, $T(\boldsymbol{x})$ is the unknown temperature field, $f(\boldsymbol{x}) = 0$, $g(\boldsymbol{x}) = 200$ on Γ_{g2}, and 10 on Γ_{g1} and $\boldsymbol{\kappa}$ is the thermal conductivity matrix, which is taken as identity. Table 4.3 lists the mesh with various level of refinement used in this analysis. The domain is discretized with 4-noded quadrilateral element with single degree of freedom per node.

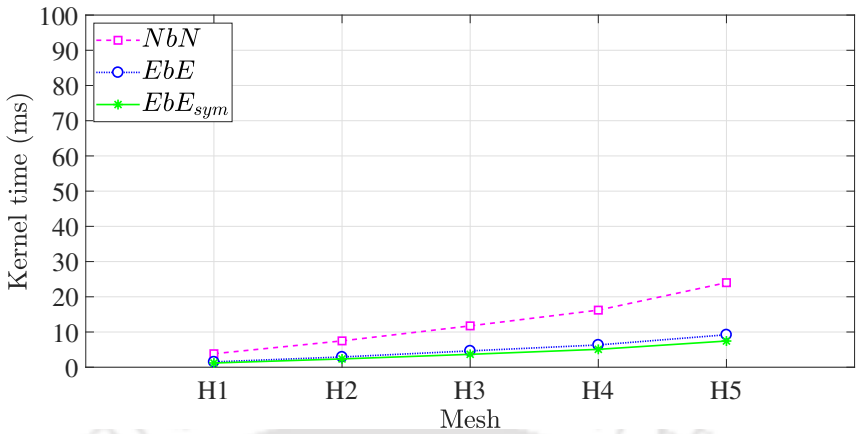


Figure 4.17: Kernel time for heat conduction over a plate.

The comparison of kernel timings for heat conduction problem is shown in Fig. 4.17. Here, *NbN* and *DbD* strategies are equivalent as each node has only one DOF. Hence, results for *DbD* strategy are not presented. Like elasticity problems, *NbN* strategy consumes the highest amount of kernel time followed by *EbE* strategy, for the heat conduction problem. In this problem, elemental tangent matrices are only of size 4×4, which is too small to efficiently compute matrix-vector product on a GPU. Still, the best kernel timings by the proposed EbE_{sym} strategy. This highlights the effectiveness of the proposed strategy in optimizing data movement using symmetric part of elemental tangent matrix.

The speedup by EbE_{sym} strategy over existing matrix-free strategies for steady-state heat conduction problem is shown in Fig. 4.18. Here 3.2× speedup is observed with respect to the *NbN* strategy and 1.3× with respect to the *EbE* strategy. Compared to elasticity problems, relatively lower speedup is observed in case of heat transfer problem because of smaller size of elemental matrices.

The GPU memory consumption for heat conduction problem is shown in Fig. 4.19. It can be observed that the proposed EbE_{sym} strategy occupies the least amount of GPU memory. For the storage of elemental tangent matrices, the EbE_{sym} strategy require 1.6× less storage space than *EbE* strategy. Overall, the EbE_{sym} strategy performs computation using 1.3× less storage space than *EbE* strategy.

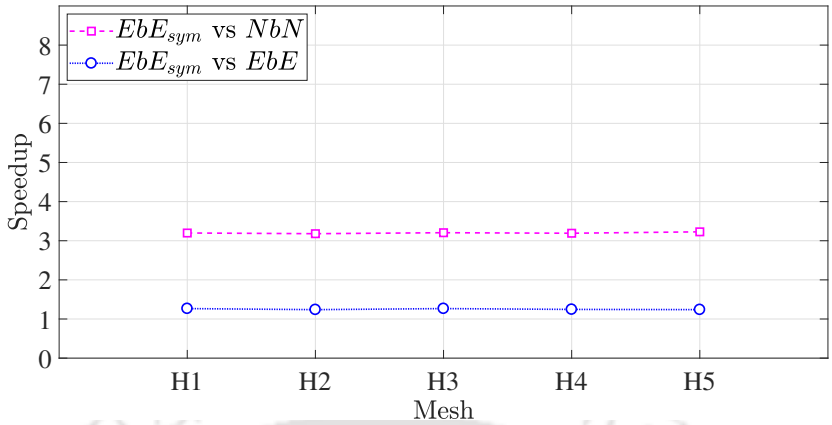


Figure 4.18: Speedup achieved by EbE_{sym} strategy over existing matrix-free strategies for heat conduction problem.

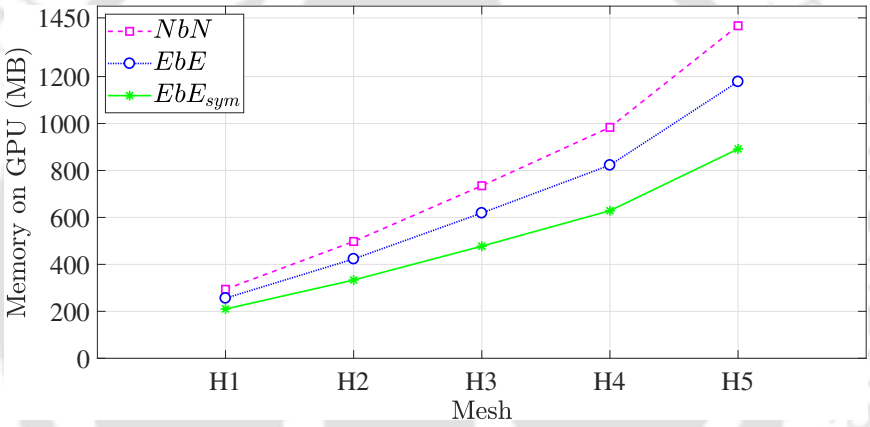


Figure 4.19: GPU memory utilization for heat conduction over a plate.

4.3.3 Elastoplastic problems

In this experiment, we solve the L-bracket and the plate with multiple holes examples as given in Section 3.2.1.2 and 3.2.1.3, respectively. The solution procedure remains the same as Chapter 3, except that the solution of system of linear equations is done by matrix-free implementation of CG solver. Figure 4.20 shows the kernel timings for EbE and EbE_{sym} strategies with percentage plasticity. The kernel timings for both the strategies are found invariant with amount of plasticity. This is expected as increasing plasticity level neither change the number of elemental tangent matrices nor access pattern. It can be observed that EbE_{sym} strategy achieves less kernel timings than EbE strategy for all mesh sizes. Like previous examples, reduction in memory footprint allows EbE_{sym} strategy achieve superior kernel timings. Similar performance gain is also observed in case of plate with multiple holes example, see Fig. 4.21. The EbE_{sym} strat-

egy achieve 1.3× speedup in case of L-bracket and 1.4× speedup in plate with multiples holes example. In addition, the use of only symmetric part of elemental tangent matrices leads to reduction of 1.92× storage requirements compared to the *EbE* strategy.

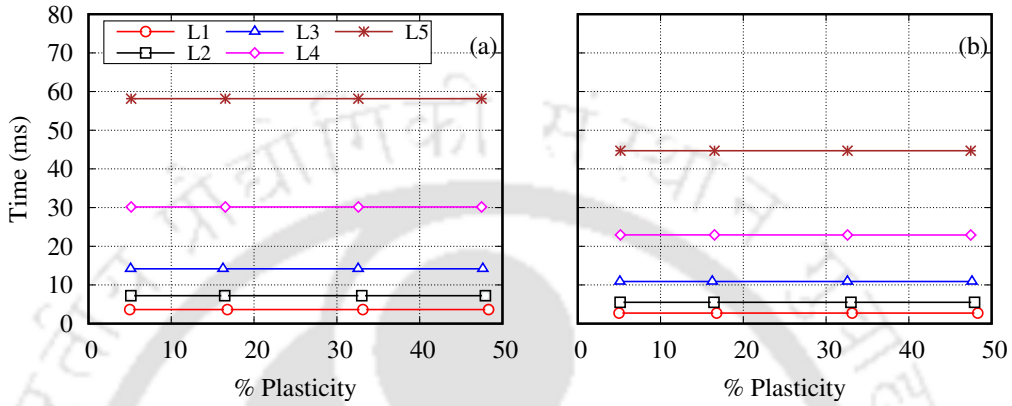


Figure 4.20: Kernel timings for L-bracket example using (a) *EbE* strategy (b) EbE_{sym} strategy.

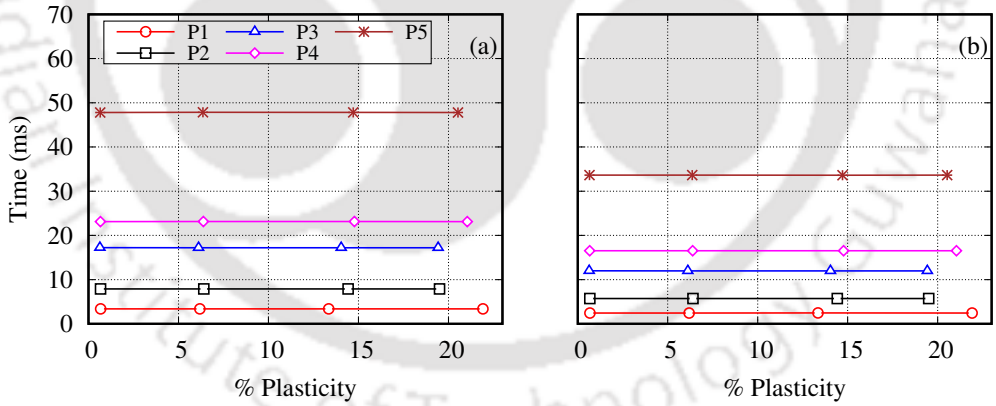


Figure 4.21: Kernel timings for plate with multiple holes example using (a) *EbE* strategy (b) EbE_{sym} strategy.

A comparison of solver timings by the fastest EbE_{sym} strategy-based matrix-free CG solver with CG implementation from Ginkgo library is presented in Fig. 4.22. In both L-bracket and plate with multiple holes examples, Ginkgo solver achieves better timings than proposed matrix-free CG solver. It can be concluded that, though the proposed matrix-free solver achieves better timings than all matrix-free strategies available in literature, fails to outperform GPU-optimized assembly-based solver.

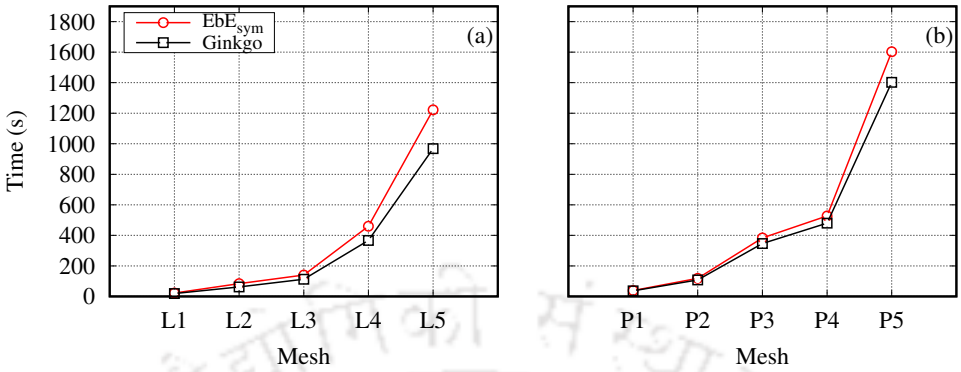


Figure 4.22: Comparison of linear solver timings for (a) L-bracket (b) plate with multiple holes.

The current results agree with Cantwell et al. (2011), which demonstrates superior performance to matrix-free strategies only for higher-order elements. However, the proposed EbE_{sym} strategy achieves significantly less memory than the classical assembly-based approach and allows us to solve bigger problems. The GPU used in this work has only 12 GB of memory, and the current comparison uses only those problems that can fit into this GPU. It would be interesting to see this comparison on recent versions of GPUs that can accommodate much bigger problems. The performance of matrix-free solvers can be improved in two ways: using preconditioners and reducing memory access. The use of preconditioners in an iterative linear solver minimizes the number of iterations needed to converge to the final solution. There are various types of preconditioners suitable for different types of linear system of equations. While simple preconditioners like diagonal or jacobi can easily be incorporated into matrix-free solvers, the absence of the global tangent matrix makes the generation of more effective preconditioners a challenging task. The development of preconditioners like LU or multigrid for matrix-free solvers is an open problem and requires further research. Another way of improving the performance of a matrix-free solver is the reduction of memory access. The matrix-free approach for unstructured mesh problems stores individual elemental matrices and leads to heavy memory traffic from the global memory. This dictates the performance of the matrix-free SpMV kernel. If the congruency of elements in a finite element mesh is leveraged, the memory requirements of a matrix-free solver can be decreased. This is discussed in the next chapter.

4.4 Closure

A new GPU-based matrix-free strategy (EbE_{sym}) for finite element method has been proposed. The developed strategy was based on an element-by-element finite element solver, which replaced SpMV operation in an iterative solution method by an element level dense matrix-vector product. A new approach to compute the elemental matrix-vector product was developed, which used only the symmetric part of the elemental matrices. The performance of the proposed solver was evaluated by solving both the elasticity and the heat transfer problems on unstructured mesh using 4-noded quadrilateral element, and comparison was made with the existing GPU-based matrix-free solvers. For the elasticity problems (two DOF per node), approximately 5× speedup was observed over the node based (NbN), 2.8× over the DOF based (DbD) and 1.4× over the element based (EbE) matrix-free strategies. In heat conduction problem (single DOF per node), 3× speedup over the NbN and 1.3× speedup over the EbE matrix-free solver were obtained. As a consequence of using symmetric part of the elemental matrices, the overall memory footprint of the proposed EbE_{sym} strategy was reduced by 1.5× in elasticity and 1.3× in heat conduction problems over the state-of-the-art EbE strategy. In elastoplastic problems, EbE_{sym} achieves 1.3× for L-bracket and 1.4× for plate with multiple holes problems, as compared to EbE strategy. The results suggest that the proposed strategy can be used to solve bigger problems on a given GPU in lesser time. The proposed strategy is applicable where symmetric elemental matrices are generated by the finite element method. The usage of matrix-free strategies with unstructured mesh demands large amount of memory to store elemental tangent matrices. This limits the size of problems as well as performance by saturating the memory bandwidth. However, recent versions of GPUs like V100 and A100 have much higher memory bandwidth than K40 used in this work. The proposed strategies are well suited to recent versions of GPUs and their performance is expected to become better due to increased memory bandwidth. In future, the proposed strategy will be applied to various element types including higher order and three-dimensional elements to study the performance and identify the limitations, if any. The outcome of the future work is expected to make this strategy more general and applicable to a broad class of problems in FEM. Moreover, the proposed strategy can also be used to develop kernel for batched symmetric matrix-vector product using small size matrices for linear algebra applications.

Chapter 5

Matrix-free CG Solver for Elastoplasticity using Structured Mesh

Several engineering problems consist of relatively simple geometry that can be discretized with structured mesh. The finite element discretization with structured mesh has many implementation advantages compared to unstructured mesh, except it doesn't conform to complex shapes. Structured meshes are easy to generate and have small memory requirement. They also offer better numerical accuracy and fast solutions for simple geometry problems. In many cases, a structured mesh can be obtained with linear cubic elements having the same shape and size, referred to as voxel-based mesh. Due to congruency of elements, voxel-based meshes have special property that a common elemental tangent matrix can be used for all elements in the mesh. This leads to significant reduction in memory requirement. However, assembly-based solvers still needs to construct the global tangent matrix that requires a large storage space. Voxel-based meshes are particularly useful with matrix-free FEM solvers that do not require to assemble the global tangent matrix. Here, computations can be performed with only one elemental tangent matrix, reducing the memory footprint to a minimum. As the dependency on data requirement reduces in a matrix-free solver, the performance is expected

to improve significantly on memory bound processor architectures like GPU. Therefore, a GPU-based matrix-free FEM solver for voxel-based mesh is an attractive option for large-scale elastoplastic problems. Though, voxel-based mesh is inherently applicable to simpler geometries with plane boundary surfaces, a recent study by Prabhune & Suresh (2020) shows sufficient accuracy for curved surfaces when using high resolution mesh. In literature, voxel-based mesh has been efficiently used to model bone failure (Nguyen & Schillinger 2018, Guha et al. 2022), topology optimization (Träff et al. 2023), simulation of material microstructures (Carter et al. 2001) and manufacturing processes (Schnös et al. 2021) to name a few.

The numerical solution procedure for elastoplasticity contains both elastic and plastic states simultaneously. In elastoplasticity, computational treatment of elastic and plastic elements are different. Unlike elastic zone where constitutive matrix remains fixed, each Gauss point in plastic zone has unique tangent modulus which depends on the state of internal variables. The unique tangent modulus leads to a unique elemental tangent matrix for elements in the plastic zone. Therefore, elemental tangent matrices must be computed individually and stored to use in matrix-free SpMV, even with voxel-based mesh. This is contrary to linear elasticity where single elemental tangent matrix is used for matrix-free SpMV. In elastoplasticity, an optimum strategy for SpMV could be the use of single common elemental tangent matrix for elements in the elastic zone and individual elemental tangent matrices for elements in the plastic zone. The elastic and plastic zones in the body evolve during the iterative solution process till final configuration is determined. This implies that the matrix-free SpMV implementation has to work with different sets of elastic and plastic elements in each iterative step. This necessitates a bookkeeping feature in computer implementation to track the state of an element. On CPU, conditional statements can be used to perform computation according to the state of an element. However, conditional statements introduces branching issues in parallel computing. It is well known that branching in parallel programs is detrimental to the performance and must be avoided. In GPU computing, branching appears in form of thread divergence (NVIDIA Corporation 2022). If all threads in a warp do not follow common execution path, the warp must execute multiple times, once for each execution path, leading to the wastage of GPU resources. Therefore, computation of matrix-free SpMV for elastoplasticity requires development of suitable strategy that can

handle elastic and plastic states efficiently and achieve better performance on GPU. In this chapter, we perform broad study into GPU implementation of matrix-free SpMV for elastoplasticity and make an effort to find the best strategy in terms of performance.

5.1 Matrix-free strategy in elastoplasticity

The elemental tangent matrix in Eq. (2.28) is given as,

$$\mathbf{K}^e = \int_{\mathcal{B}^e} \mathbf{B}^T \mathbf{D} \mathbf{B} dv. \tag{5.1}$$

Also, depending upon elastic or plastic state of deformation, the material constitutive matrix takes suitable form, as given by Eq. (2.3),

$$\mathbf{D} = \begin{cases} \mathbf{D}_e, & \text{if elastic,} \\ \mathbf{D}_p, & \text{if plastic.} \end{cases} \tag{5.2}$$

where $\mathbf{D}_e$ is the elastic constitutive matrix and $\mathbf{D}_p$ is the plastic constitutive matrix. The constitutive matrix can be split into two components and expressed as (Prabhune & Suresh 2020),

$$\mathbf{D} = \mathbf{D}_e + (\mathbf{D} - \mathbf{D}_e). \tag{5.3}$$

Here, the second component $(\mathbf{D} - \mathbf{D}_e)$ becomes zero for elastic state and remains non-zero for elements in the plastic zone. Substituting Eq. (5.3) into Eq. (5.1), the tangent matrix expression can be written as sum of two components, which is given as,

$$\begin{aligned} \mathbf{K}^e &= \int_{\mathcal{B}^e} \mathbf{B}^T \left[\mathbf{D}_e + (\mathbf{D} - \mathbf{D}_e)\right] \mathbf{B} dv, \\ \mathbf{K}^e &= \int_{\mathcal{B}^e} \mathbf{B}^T \mathbf{D}_e \mathbf{B} dv + \int_{\mathcal{B}^e} \mathbf{B}^T (\mathbf{D} - \mathbf{D}_e) \mathbf{B} dv \end{aligned} \tag{5.4}$$

$$\mathbf{K}^e = \mathbf{K}^e_{\text{elastic}} + \mathbf{K}^e_{\text{plastic}} \tag{5.5}$$

where $\mathbf{K}^e_{\text{plastic}}$ is non-zero only for elements undergoing plastic deformation. It is noted that $\mathbf{K}^e_{\text{elastic}}$ remains the same for all elements in the mesh and over all the load steps in

elastoplastic simulation. Therefore, only one tangent matrix is computed for all elastic elements and stored in the memory for SpMV computation. The plastic component $\mathbf{K}^e_{\text{plastic}}$ is computed for all elements in the plastic zone and in each NR iteration.

Let us consider the matrix-free computation of SpMV in elastoplasticity. Substituting Eq. (5.5) into Eq. (4.3), following expression is obtained,

$$\begin{aligned} \mathbf{g} &= \underset{e\in\mathcal{E}}{\mathcal{A}} \left(\left(\mathbf{K}^e_{\text{elastic}} + \mathbf{K}^e_{\text{plastic}} \right) \mathbf{p}^e \right), \\ \mathbf{g} &= \underset{e\in\mathcal{E}}{\mathcal{A}} \left(\mathbf{K}^e_{\text{elastic}} \mathbf{p}^e \right) + \underset{e\in\mathcal{E}^{(p)}}{\mathcal{A}} \left(\mathbf{K}^e_{\text{plastic}} \mathbf{p}^e \right), \end{aligned} \tag{5.6}$$

where $\mathcal{E}^{(p)}$ is the set of all elements in the plastic zone. The resultant vector $\mathbf{g}$ now contains contribution of two components. The first component computes the matrix-vector product for all elements in the mesh by considering only elastic tangent matrix. The second component performs computation using plastic tangent matrix for only those elements that undergo plastic deformation.

As suggested in (Prabhune & Suresh 2020), the computational strategy for GPU implementation uses separate CUDA kernels for the first and second components in Eq. (5.6). It is due to this reason, it is referred to as *split kernel* strategy, hereafter. A schematic representation of matrix-free SpMV expressed by Eq. (5.6) is shown in Fig. 5.1. As shown in the figure, the term 'elastic' refers to CUDA kernel for the first component and 'plastic' refers to CUDA kernel for the second component. The CUDA kernel for the elastic part uses single elemental tangent matrix to compute matrix-vector product for all the elements in the mesh. The second kernel for plastic part performs computation for only those elements that undergo plastic deformation. In Fig. 5.1, white boxes show elastic elements while colored boxes show active computation for plastic elements. The kernel for plastic part uses conditional statement to differentiate plastic elements from the bulk and access individual tangent matrices.

The kernel implementation for elastic part is efficient as it has the optimum memory access due to the use of one elemental tangent matrix. On the other hand, the kernel for plastic part suffers from thread divergence issue and leads to inefficient use of GPU resources. This drawback has been addressed in the proposed strategy, as discussed in

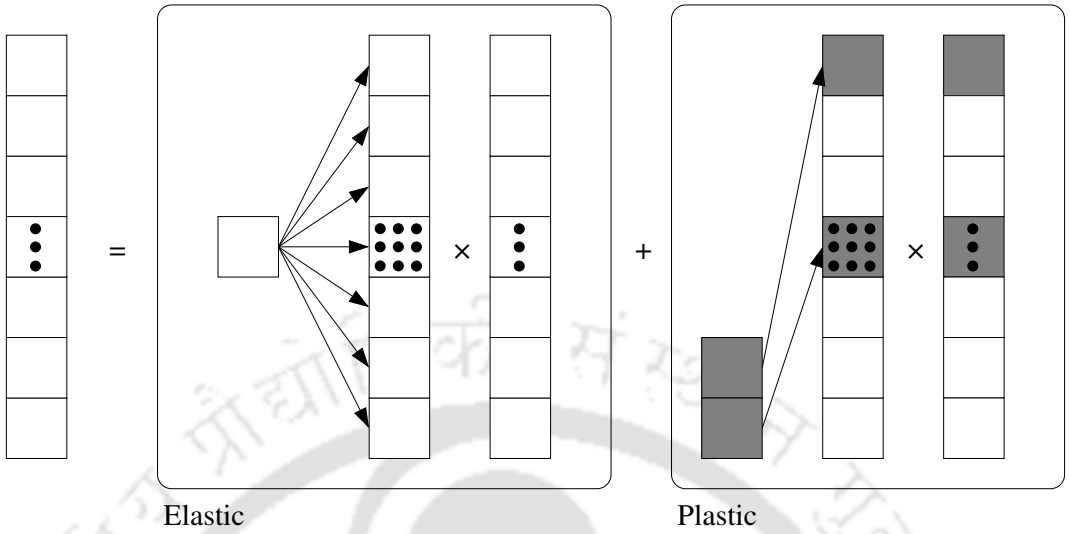


Figure 5.1: Illustration of matrix-free SpMV strategy that splits the computation into two CUDA kernels, one for the elastic part and another for the plastic part.

the following sections.

5.2 Proposed single kernel strategy[1]

In the proposed strategy, elements in the elastic and plastic zones can be separated into two mutually exclusive sets,

$$\mathcal{E} = \mathcal{E}^{(e)} + \mathcal{E}^{(p)}, \tag{5.7}$$

where $\mathcal{E}^{(e)}$ is the set of all elements in the elastic zone. This leads to reformulation of matrix-free SpMV as follows,

$$\mathbf{g} = \underset{e\in\mathcal{E}}{\mathcal{A}} \left(\mathbf{K}^e\mathbf{p}^e\right) = \underset{e\in\mathcal{E}^{(e)}}{\mathcal{A}} \left(\mathbf{K}^e_{\text{elastic}}\mathbf{p}^e\right) + \underset{e\in\mathcal{E}^{(p)}}{\mathcal{A}} \left(\mathbf{K}^e_{\text{plastic}}\mathbf{p}^e\right). \tag{5.8}$$

The proposed strategy uses the actual tangent constitutive matrix for elements in the plastic zone, rather splitting it as given in Eq. 5.3. Consequently, the expression for

[1]The content of this section has been published in **Mathematics and Computers in Simulation** (Kiran et al. 2024*b*).

$\mathbf{K}^e_{\text{plastic}}$ changes according to the following definition,

$$\mathbf{K}^e_{\text{plastic}} = \int_{\mathcal{B}^e} \mathbf{B}^T \mathbf{D}_p \mathbf{B} dv. \tag{5.9}$$

The elemental tangent stiffness matrix takes the form of $\mathbf{K}^e_{\text{elastic}}$ for the elastic state and $\mathbf{K}^e_{\text{plastic}}$ (given by Eq. (5.9)) for the plastic state. In Eq. (5.8), we observe that the two components perform computation with a mutually exclusive set of elements, unlike Eq. (5.6), where the first component performs computation for the set of all elements ($\mathcal{E}$) in the mesh. In the proposed strategy, the first component is concerned with multiplication of elastic tangent matrix only for those elements that observe elastic deformation. Similarly, the second component performs multiplication only for elements in the plastic region. The proposed strategy performs computation for plastic elements only once compared to twice (splitting leads to two multiplication in Eq. (5.6)) by the split kernel strategy, thereby reducing the number of arithmetic operations to perform matrix-free SpMV.

Though Eq. (5.8) shows two components, the GPU implementation for the proposed strategy uses single CUDA kernel to perform computations required in matrix-free SpMV for elastoplasticity. The proposed strategy is referred to as *single kernel* strategy in the subsequent discussion. An outline of the proposed strategy is shown in Fig. 5.2. In the figure, white boxes show elastic tangent matrices and grey boxes denote plastic tangent matrices. It can be seen that the computation of matrix-vector product does not differentiate between elastic and plastic elements. However, input data (elemental tangent matrices) varies according to the state of elements. The proposed strategy avoids the use of conditional statement, and uses an array of indices to identify and locate appropriate tangent matrix for the computation. As shown in Fig. 5.3, threads first access their respective indices values from an array and use them to access tangent matrices. The elastic tangent matrix is placed at 0^{th} location, whereas plastic tangent matrices are placed at locations starting from 1^{st} to n^{th}. Thus, an optimum memory access is achieved and need for conditional statements due to different state of elements can be prevented. The proposed strategy has the following advantages over the previous strategy (Prabhune & Suresh 2020):

1. An efficient utilization of GPU resources is achieved by avoiding thread divergence.

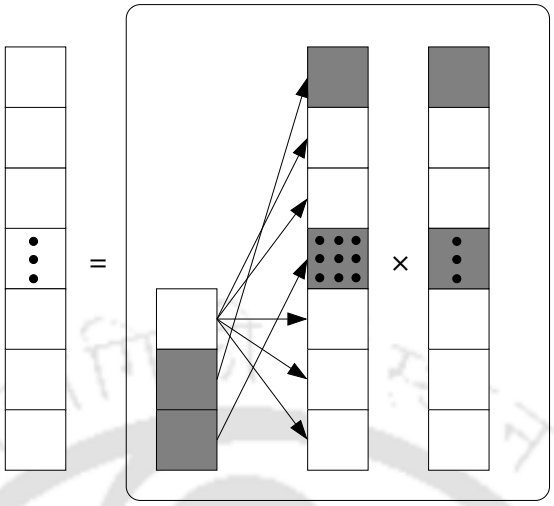


Figure 5.2: Illustration of the proposed strategy in which white boxes represent data for elements in the elastic zone and others represent data for elements in the plastic zone.

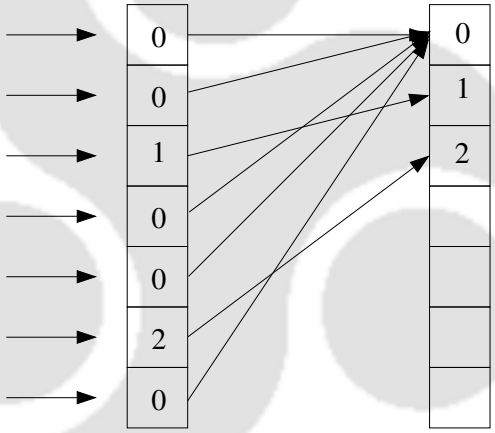


Figure 5.3: The access of elemental tangent matrices by GPU threads in the proposed strategy.

2. The redundant computation for plastic element is removed.

3. The number of kernel launches is reduced by half.

Let us take an example as shown in Fig. 5.4 to describe the single kernel strategy. The figure shows 3×3 structured mesh with a global element number marked on the top left corner. Let us assume that the shaded elements exhibit plastic deformation. In order to keep track of element status, an array of global element numbers is taken. In the initial stage of the solution process, only one elemental tangent matrix is used for the computation of matrix-vector products, as all elements are in the elastic state. However, as the plastic state appears (as shown in Fig. 5.4), tangent matrices for four elements

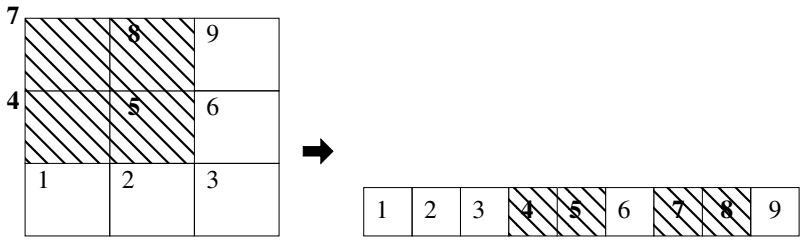


Figure 5.4: An example mesh and list of elements. The shaded elements show plastic state.

must be computed. In total, $4 + 1 = 5$ elemental tangent matrices are required for the computation of matrix-free SpMV. Since lists of elastic and plastic elements are updated in each Newton iteration, GPU implementations of computation of tangent matrices and matrix-free SpMV must be able to work efficiently with the updated list.

Figure 5.5 illustrates the GPU implementation of the single kernel strategy. First, the array of global element numbers (as shown in Fig. 5.4) is modified by adding the total number of elements (e_t) to all element numbers that are in the elastic zone. Consequently, all plastic elements now have global element numbers less than e_t. In the next step, the modified array is sorted in ascending order to segregate the elastic and plastic elements. As shown in the third array from the top, the sorted array contains plastic elements ($\mathcal{E}^{(p)}$) at the first four places, followed by elastic elements. This array is used to compute elemental tangent matrices where five threads are launched (four for plastic elements and one for elastic elements) to perform the computation for the first five element numbers. The tangent matrices for plastic elements are stored at position 1 to 4, and the common tangent matrix for elastic elements is stored at 5^{th} position. The indices to access elemental tangent matrices are stored for each element in a separate array, as denoted by 5^{th} array from the top. In the last step, the indices for elemental tangent matrices are moved to new positions by using global element numbers as indices into the final array. The final array is supplied as an input to matrix-free SpMV kernel, where threads use respective global element numbers to access the correct index for elemental tangent matrices.

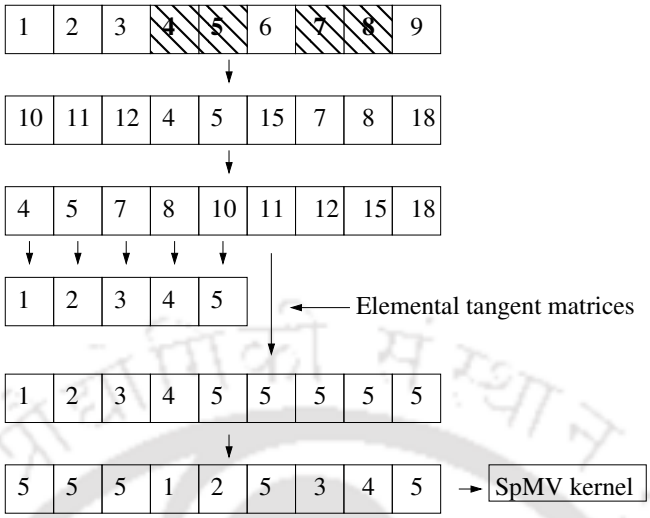


Figure 5.5: Illustration of single kernel strategy for matrix-free SpMV in elastoplasticity.

5.2.1 GPU implementation

5.2.1.1 Single kernel *NbN* strategy

In this strategy, node-based thread assignment is proposed to perform computation of matrix-free SpMV, as given by Eq. (5.8). The proposed single kernel *NbN* strategy is based on *NbN* matrix-free strategy given in Section 4.1.1 and uses only one compute kernel for elements in elastic and plastic zones.

Algorithm 13 shows the GPU implementation of single kernel *NbN* strategy. The input data required for the computation are array `y` ($\mathbf{y}$) with which multiplication is performed, elemental tangent stiffness matrices `elem_mat` ($\mathbf{K}^e$) for both elastic and plastic states, list of neighbouring elements for each node in `nelem_list`, array `localNode_pos` to store local position of each node in its neighbouring elements list, array `emat_indicies` to keep locations of elemental tangent matrices and connectivity for each element in `connectivity`. The output of the kernel is vector $\mathbf{p}$ (referred to as `p`) which keeps the result of matrix-vector product. As shown in line 2 of Algorithm 13, one thread is assigned to one node to perform its respective computation. The total number of elements that share a node is denoted by a variable $t\_neighbourElem$. Threads perform computation inside a loop over $t\_neighbourElem$. For each element that shares the node, global element number ($elem\_no$) and local position of nodes in the element connectivity (l_index) is obtained using the arrays `nelem_list` and `localNode_pos`, respectively (see

lines 5 and 6). Next, the array `emat_indices` is accessed in line 7 to find the location *e_index* of elemental tangent matrices. In the array `emat_indices`, the value of '0' refers to elastic tangent matrix, and all other values refers to specific plastic tangent matrix. The multiplication of elemental tangent matrix entries with vector **y** (y) is performed inside an outer loop over DOFs associated with a node (n_{dof}) and inner loop over total DOFs associated with the element (e_{dof}), given by line 10. The value of e_{dof} determines the size as well as the number of non-zero values (e_{nnz}) of the elemental tangent matrix. For 8-noded hexahedral element, $e_{dof} = 24$ and $e_{nnz} = e_{dof} \times e_{dof} = 576$. In array `elem_mat`, $e_{index} * e_{nnz}$ gives the location of a specific elemental tangent matrix and $(3 * l_{index} + i) * e_{dof}$ points to the location of a required row in the elemental tangent matrix, see line 11. To perform multiplication, an elemental sub-vector of **y** is required, which is obtained by global connectivity (`global_dof`) of the concerned element. Results of multiplication are initially stored in local variables (registers) and later accumulated to global vector `p` at the end of computation (see line 15). Since one thread performs computations for one node, the accumulation of computed values to `p` vector remains conflict free. It is noted that the proposed strategy does not perform any search or identification check for plastic and elastic states and presents a framework for efficient uniform treatment of elemental tangent matrices.

Since there are multiple nodes associated with an element, the `nelem_list` for all these nodes contains the same element number. This implies that the data associated with the element cannot be ordered to favour memory access for all connected nodes without redundancy. If data is ordered according to `nelem_list` of one node, it remains unordered for other nodes. Due to this reason, data like element connectivity and vector **y** can not be ordered. On the other hand, different rows of elemental tangent matrices are uniquely accessed by all connected nodes and can be ordered. However, the number of elastic and plastic elements for each node in `nelem_list` varies with each other and with Newton iterations, making it difficult to achieve any ordering scheme without redundant storage of elastic elemental tangent matrix. Therefore, in the current work, data is not ordered and access to elemental tangent matrices, elemental connectivity and vector **y** remain uncoalesced. However, as shown in line 11 of Algorithm 13, plastic elemental tangential matrices are stored side-by-side to increase locality.

Since the size of elemental tangent stiffness matrix is 24×24, the size of `elem_mat`

Algorithm 13 Single kernel Node-by-Node matrix-free strategy.

Input: `y`, `elem_mat`, `nelem_list`, `localNode_pos`, `emat_indices`, `connectivity`
Output: `p`

1: $threadId \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
2: $nodeId \leftarrow threadId$
3: **if** $nodeId < maxNodes$ **then**
4: **for** $e = 1$ to $t\_neighbourElem$ **do** ▷ Loop over all neighbouring elements
5: $elem\_no \leftarrow$ `nelem_list`
6: $l\_index \leftarrow$ `localNode_pos` ▷ $l\_index \in (0, 1, ..., 7)$
7: $e\_index \leftarrow$ `emat_indices` ▷ $e\_index \in (0, 1, ..., e_p)$
8: `global_dof` $\leftarrow$ `connectivity`$[elem\_no]$
9: **for** $i = 1$ to n_{dof} **do** ▷ Loop over DOFs associated with a node
10: **for** $j = 1$ to e_{dof} **do** ▷ Loop over DOFs associated with an element
11: `val`$[i]$ += `elem_mat`$[e\_index * e_{nnz} + (3 * l\_index + i) * e_{dof} + j]$ * `y`[`global_dof`$[j]$]
12: **end for**
13: **end for**
14: **end for**
15: `p` $\leftarrow$ `val` ▷ Accumulate to global vector **p**
16: **end if**

array becomes $(e_p + 1) * 24 \times 24$, where e_p is the number of plastic elements. The size of arrays `nelem_list`, `localNode_pos`, and `connectivity` depends on mesh topology and problem size. When the size of plastic zone is small, space occupied by elemental tangent matrices is less or comparable to other input arrays. However, for problems with the large plastic zone, elemental tangent matrices occupy the most of the storage space and become performance determiner. The use of single elemental tangent matrix for all the elastic elements and individual tangent matrices for plastic elements is the most optimum strategy and expected to provide best performance in SpMV computation.

5.2.1.2 Split kernel *NbN* strategy

In this strategy, the computation of matrix-free SpMV for elastoplaticity is split into two compute kernels, one for the elastic part and another for the plastic part of elemental tangent matrices (refer to Section 5.1). Both the kernels are implemented using *NbN* strategy. This strategy is proposed in Prabhune & Suresh (2020) and reproduced here to compare the performance of the proposed single kernel strategy.

Algorithms 14 and 15 show the GPU implementation of split kernel *NbN* strategy. In

Algorithm 14, computation for the elastic component of all elemental tangent matrices is done. The input data variables remain the same as single kernel *NbN* strategy, as discussed in Section 5.2.1.1. The computation of SpMV is done inside a loop over all neighbouring elements with one thread assigned to each node, similar to Algorithm 13. Since elastic part of elemental tangent matrices remains the same across the mesh, one elemental tangent matrix is read for all the elements, as shown in line 10. The GPU implementation for plastic part of split kernel *NbN* strategy is described in Algorithm 15. Most of the computational steps remain the same as previous strategies, except that the computation is performed only for elements undergoing plastic deformation. The information about elastic and plastic states is stored in an array `e_state`, which is accessed in line 5 to determine the state of an element. For each plastic element, respective elemental tangent matrix is accessed in line 9 and used in line 12 to perform multiplication with an elemental sub-vector from vector `y`. Finally, the computed results are accumulated into global array `p` at line 17.

Algorithm 14 GPU kernel for elastic part of split kernel Node-by-Node matrix-free strategy.

Input: `y`, `elem_mat`, `nelem_list`, `localNode_pos`, `connectivity`
Output: `p`

1: $threadId \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
2: $nodeId \leftarrow threadId$
3: **if** $nodeId < maxNodes$ **then**
4: **for** $e = 1$ to $t\_neighbourElem$ **do** ▷ Loop over all neighbouring elements
5: $elem\_no \leftarrow$ `nelem_list`
6: $l\_index \leftarrow$ `localNode_pos` ▷ $l\_index \in (0, 1, ..., 7)$
7: `global_dof` $\leftarrow$ `connectivity`$[elem\_no]$
8: **for** $i = 1$ to n_{dof} **do** ▷ Loop over DOFs associated with a node
9: **for** $j = 1$ to e_{dof} **do** ▷ Loop over DOFs associated with an element
10: `val`$[i]$ += `elem_mat`$[(3 * l\_index + i) * e_{dof} + j]$ * `y`[`global_dof`$[j]$] ▷ Same elemental tangent matrix is read
11: **end for**
12: **end for**
13: **end for**
14: `p` $\leftarrow$ `val` ▷ Accumulate to global vector **p**
15: **end if**

Algorithm 15 GPU kernel for plastic part of split kernel Node-by-Node matrix-free strategy.

Input: `y`, `elem_mat`, `nelem_list`, `localNode_pos`, `e_state`, `connectivity`
Output: `p`

1: $threadId \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
2: $nodeId \leftarrow threadId$
3: **if** $nodeId < maxNodes$ **then**
4: **for** $e = 1$ to $t\_neighbourElem$ **do** ▷ Loop over all neighbouring elements
5: **if** `e_state`$[e] == plastic$ **then** ▷ Check for plastic element
6: $elem\_no \leftarrow$`nelem_list`
7: $l\_index \leftarrow$`localNode_pos` ▷ $l\_index \in (0, 1, ..., 7)$
8: `global_dof` $\leftarrow$ `connectivity`$[elem\_no]$
9: `emat_plastic` $\leftarrow$ get_elemental_tan_mat(e, `elem_mat`)
10: **for** $i = 1$ to n_{dof} **do** ▷ Loop over DOFs associated with a node
11: **for** $j = 1$ to e_{dof} **do** ▷ Loop over DOFs associated with an element
12: `val`$[i]$ += `emat_plastic`$[(3*l\_index+i)*e_{dof}+j]$ * `y`[`global_dof`$[j]$] ▷ Individual tangent matrices are read
13: **end for**
14: **end for**
15: **end if**
16: **end for**
17: `p` $\leftarrow$ `val` ▷ Accumulate to global vector **p**
18: **end if**

5.2.1.3 Single kernel *DbD* strategy

In this strategy, DOF-based thread assignment is proposed to perform computation of matrix-free SpMV, as given by Eq. (5.8). The proposed single kernel *DbD* strategy is based on *DbD* matrix-free strategy given in Section 4.1.2.

The pseudo code for GPU implementation of single kernel *DbD* strategy is explained in Algorithm 16. Since all DOFs associated with a common node share the same neighborhood information, the input data variables in *DbD* strategy are exactly the same as the *NbN* strategy. The number of parallel threads are equal to total DOFs associated with the problem (line 2). Since, multiple threads are associated with a node, more than one thread access same data points (see lines 5–8). This leads to redundant memory access in *DbD* strategy. However, as shown in line 10, the access to elemental tangent matrices is not redundant as each DOF has a unique row associated with it. The presence of redundant memory access is detrimental to kernel performance on GPU, as long as the size of elemental tangent matrices is comparable to other variables, observed in case

of problems with small plasticity. For problems with large plastic zone, element tangent matrices becomes the largest data set and therefore redundant access to other smaller data is insignificant for performance. As shown in line 10, the result of multiplication is stored in local register variable and later accumulated to the array `p` (**p**) (see line 13), once the computation is complete. In *DbD* strategy, access to vector **p** is coalesced.

Algorithm 16 Single kernel DOF-by-DOF matrix-free strategy.

Input: `y`, `elem_mat`, `nelem_list`, `localNode_pos`, `emat_indices`, `connectivity`
Output: `p`

1: $threadId \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
2: $dofId \leftarrow threadId$
3: **if** $dofId < maxDOF$ **then**
4: **for** $e = 1$ to $t\_neighbourElem$ **do** ▷ Loop over all neighbouring elements
5: $elem\_no \leftarrow$`nelem_list`
6: $l\_index \leftarrow$`localNode_pos` ▷ $l\_index \in (0,1,...,7)$
7: $e\_index \leftarrow$`emat_indices` ▷ $e\_index \in (0,1,...,e_p)$
8: `global_dof` $\leftarrow$ `connectivity`$[elem\_no]$
9: **for** $j = 1$ to 24 **do** ▷ Loop over all DOFs associated with an element
10: val += `elem_mat`$[e\_index * e_{nnz} + (3 * l\_index + (dofId\%3)) * e_{dof} + j]$ * `y`[`global_dof`$[j]$]
11: **end for**
12: **end for**
13: `p`$[dofId] \leftarrow val$
14: **end if**

5.2.1.4 Split kernel *DbD* strategy

In this strategy, split kernel approach is implemented on GPU with DOF-based thread assignment. As discussed in Section 5.1, two separate kernels are used for the computation of matrix-free SpMV. The computation for elastic and plastic states is organized in the same way as the split kernel *NbN* strategy in Algorithms 14 and 15, except that now one thread is assigned to one DOF. The GPU implementations for elastic and plastic part of elemental tangent matrices are described in Algorithms 17 and 18, respectively. In Algorithm 17, major differences with Algorithm 14 can be seen at lines 8 and 9. Due to the DOF-based thread assignment, a thread performs computation of vector-dot product by following a single loop over all DOFs associated with an element. On the contrary, in lines 8–10 of Algorithm 14, a thread performs computation of matrix-vector

product by using double loops. In Algorithm 18, similar implementation differences with Algorithm 15 can be observed at lines 10 and 11.

Algorithm 17 GPU kernel for elastic part of split kernel DOF-by-DOF matrix-free strategy.

Input: y, elem_mat, nelem_list, localNode_pos, connectivity
Output: p

1: $threadId \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
2: $dofId \leftarrow threadId$
3: **if** $dofId < maxDOF$ **then**
4: **for** $e = 1$ to $t\_neighbourElem$ **do** ▷ Loop over all neighbouring elements
5: $elem\_no \leftarrow$ nelem_list
6: $l\_index \leftarrow$ localNode_pos ▷ $l\_index \in (0,1,...,7)$
7: global_dof $\leftarrow$ connectivity$[elem_no]$
8: **for** $j = 1$ to e_{dof} **do** ▷ Loop over DOFs associated with an element
9: val += elem_mat$[(3 * l_index + (dofId\%3)) * e_{dof} + j]$ * y[global_dof$[j]$] ▷ Same elemental tangent matrix is read
10: **end for**
11: **end for**
12: p$[dofId] \leftarrow val$
13: **end if**

Algorithm 18 GPU kernel for plastic part of split kernel DOF-by-DOF matrix-free strategy.

Input: y, elem_mat, nelem_list, localNode_pos, e_state, connectivity
Output: p

1: $threadId \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
2: $dofId \leftarrow threadId$
3: **if** $dofId < maxDOF$ **then**
4: **for** $e = 1$ to $t\_neighbourElem$ **do** ▷ Loop over all neighbouring elements
5: **if** e_state$[e]$ == $plastic$ **then** ▷ Check for plastic element
6: $elem\_no \leftarrow$ nelem_list
7: $l\_index \leftarrow$ localNode_pos ▷ $l\_index \in (0,1,...,7)$
8: global_dof $\leftarrow$ connectivity$[elem_no]$
9: emat_plastic $\leftarrow$ get_elemental_tan_mat(e, elem_mat)
10: **for** $j = 1$ to e_{dof} **do** ▷ Loop over all DOFs associated with an element
11: val += emat_plastic$[(3 * l_index + (dofId\%3)) * e_{dof} + j]$ * y[global_dof$[j]$] ▷ Individual elemental tangent matrices are read
12: **end for**
13: **end if**
14: **end for**
15: p$[dofId] \leftarrow val$
16: **end if**

5.2.2 Results and discussion

In this section, performance evaluations of the GPU-based proposed matrix-free SpMV strategies for elastoplasticity are presented over three benchmark examples. All the numerical experiments are conducted by considering full-scale 3D models. The simulations are run on a machine having specifications given in Section 3.2. All the computations have been performed in double precision arithmetic. The finite element meshes for all three examples are taken as voxel type structured meshes consisting of linear hexahedral element (8-noded). In order to show the applicability of the proposed strategies for practical purposes, meshes have been generated in ABAQUS software package and the same connectivity information is used for the computation. The reported kernel timings are average values taken over five runs.

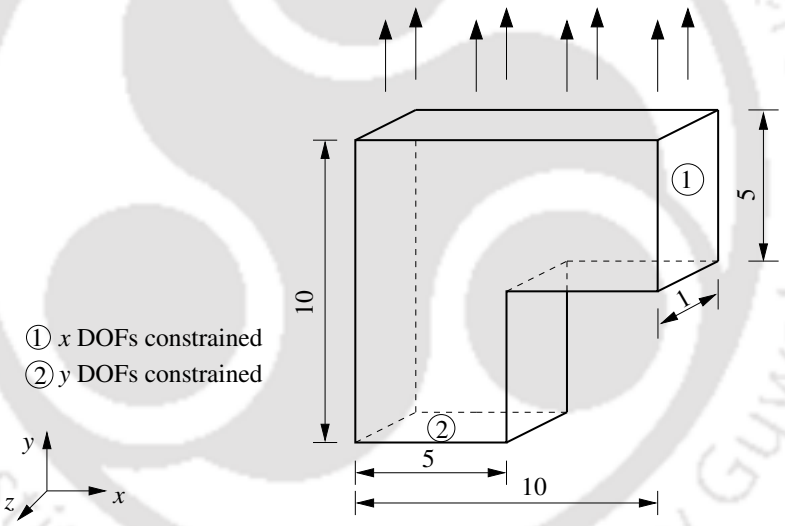


Figure 5.6: A 3D L-bracket with boundary conditions. All dimensions are taken in meters (m).

5.2.2.1 L-bracket

A 3D L-shaped bracket (Prabhune & Suresh 2020) is considered for elastoplastic analysis, as shown in Fig. 5.6 with dimensions and boundary conditions. The top surface is subjected to distributed traction in y-direction and two faces marked with ① and ② have movement constrained in x and y-directions, respectively. The force applied on the top surface can be varied to obtain different amount of plasticity in the body. The material properties are considered as: Young's modulus $E = 206900$ Pa, Poisson's ratio

$\nu = 0.29$, hardening modulus $H = 10000$ Pa, initial yield stress $\sigma_y = 450$ Pa. Table 5.1 presents the finite element mesh with different levels of refinement.

Table 5.1: Mesh for the L-bracket example.

Mesh	Elements	Nodes	DOFs
L1	103 488	115 596	346 788
L2	212 716	232 200	696 600
L3	418 176	448 115	1 344 345
L4	813 186	860 384	2 581 152
L5	1 622 964	1 697 080	5 091 240
L6	2 268 084	2 361 280	7 083 840

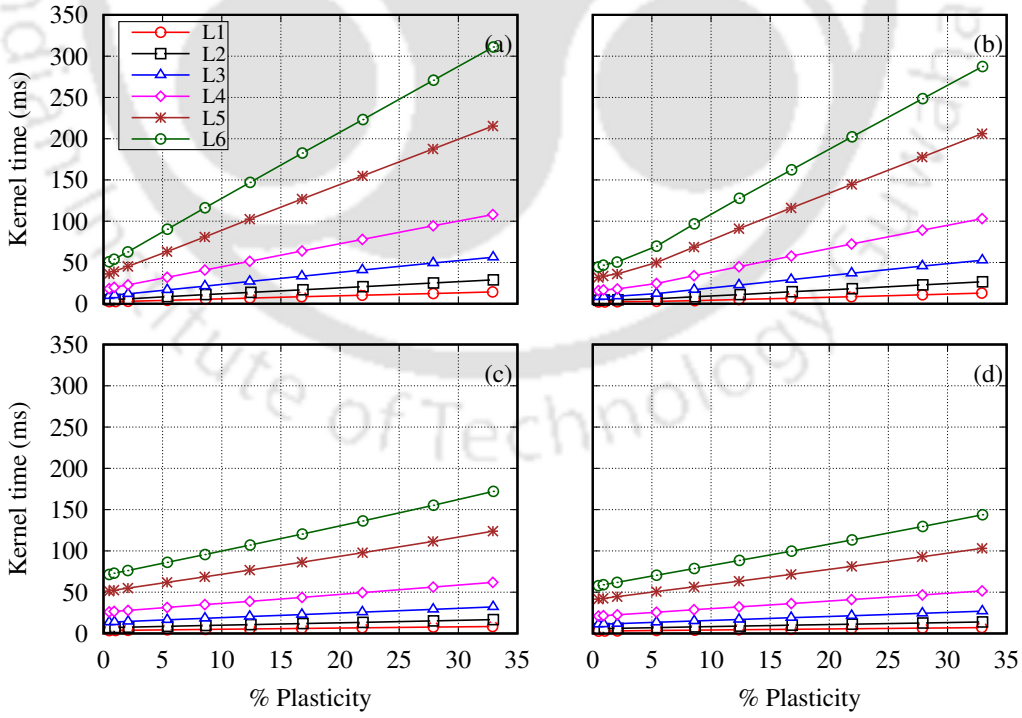


Figure 5.7: Kernel timings (milliseconds) for L-bracket example by (a) Split kernel *NbN* strategy, (b) Single kernel *NbN* strategy, (c) Split kernel *DbD* strategy and (d) Single kernel *DbD* strategy.

The kernel timings for matrix-free SPMV operation using four computational strategies are presented in Fig. 5.7. For each of the strategies, the kernel time is presented for each mesh as a function of percentage plasticity. It can be seen that the kernel timings are found varying linearly with percentage plasticity for all the strategies. As the amount of plasticity in the body increases, the number of unique plastic elemental tangent matrices also increases. This, in turn, increases the amount of memory access in matrix-free SpMV kernels. Since accessing memory on the GPU is more expensive than computation, as the amount of memory access increases, the execution timings of SpMV kernels also go up. The linear dependence of kernel time shows strong influence of memory access on the performance. One can observe that the proposed single kernel strategies perform better than split kernel strategies in both *NbN* and *DbD* settings. Comparing the performance in *NbN* strategy (see Figs. 5.7a and 5.7b), we observe that the single kernel strategy achieves least execution time for all the mesh sizes. Similar result is observed in DOF-based thread assignment (see Figs. 5.7c and 5.7d), where the proposed single kernel *DbD* strategy achieves better kernel timings. The best timings at high percentage of plasticity are achieved by single kernel *DbD* strategy.

Threads in a matrix-free SpMV kernel need to access the data from multiplying vector $\mathbf{y}$ frequently inside a loop (as discussed in Section 5.2.1.1), and therefore efficient access to vector $\mathbf{y}$ is crucial for the performance. However, access to the vector $\mathbf{y}$ is uncoalesced. The data corresponding to each node is required by threads assigned to all nodes/DOFs in its immediate neighbourhood for computation. If the access is concurrent, broadcast of the concerned value from the GPU memory would be the most efficient way. In practical scenarios, the neighbourhood of a node is determined by the topology of the mesh, which does not allow concurrent access. As discussed in Section 5.2.1.1, the access to vector $\mathbf{y}$ cannot be optimized by data ordering, and as a consequence, there are multiple threads accessing the same memory location at multiple points in time. One possible way of improving the memory access could be the use of cache memory. GPUs have a special purpose read-only cache that can be used to improve memory access to those variables that are only read but not modified. Since vector $\mathbf{y}$ and elemental tangent matrices remain read-only for the lifetime of a matrix-free SpMV kernel, it can be cached in read-only cache. Figure 5.8 shows the speedups achieved by all the matrix-free SpMV strategies when access to vector $\mathbf{y}$ and elemental tangent matrices is made

through the read-only cache of the GPU. For each strategy, the speedup is computed over the uncached version. The speedups of up to 2.5× are obtained for all the strategies when the amount of plasticity is low. The *DbD* strategy achieves more speedups in both split kernel and single kernel strategies than *NbN*. However, the speedup is found to decrease with increasing percentage of plasticity. At higher levels of plasticity, the proposed single kernel strategies achieve speedups of more than 1.5×, whereas the split kernel strategies achieve speedups up to 1.5×. In the following discussion, single kernel *NbN*, single kernel *DbD* and split kernel *DbD* strategies use read-only cache to access vector **y** and elemental tangent matrices.

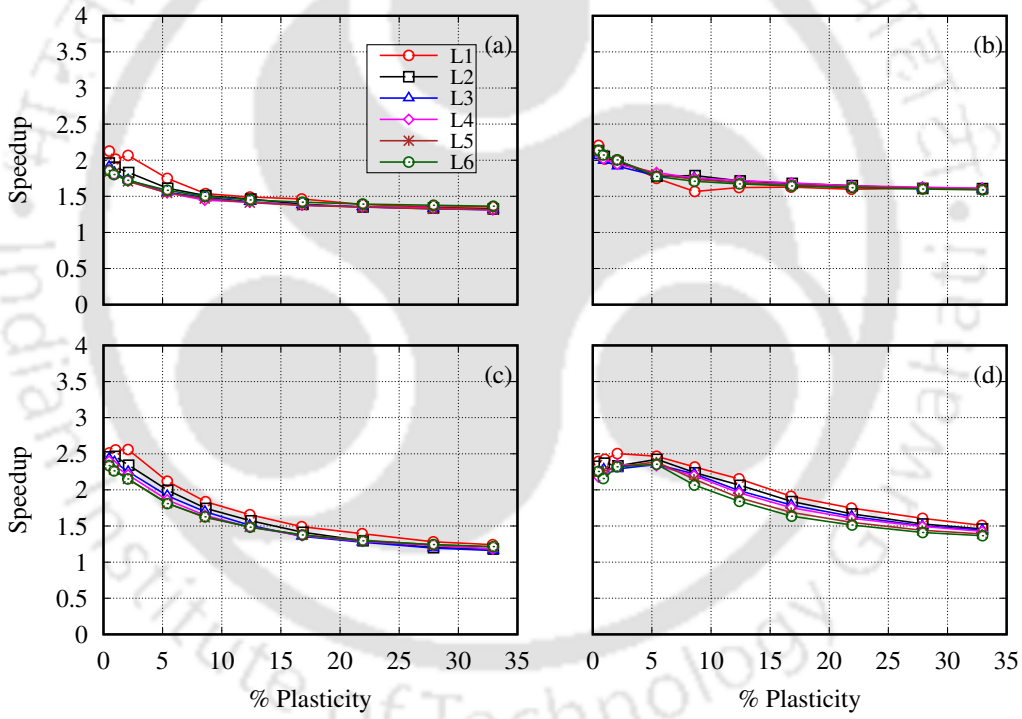


Figure 5.8: Speedups in kernel timings for L-bracket when multiplying vector (**y**) and elemental tangent matrices are cached in read-only memory. The figure shows speedups by (a) Split kernel *NbN* strategy, (b) Single kernel *NbN* strategy, (c) Split kernel *DbD* strategy and (d) Single kernel *DbD* strategy.

The usage of a single elastic elemental tangent matrix for all elements in the elastic zone provides an opportunity to use shared memory of the GPU for fast access. This approach can be advantageous for problems with a low percentage of plasticity. In Fig. 5.9, speedups are shown for the approach that stores the elastic elemental tangent matrix

in the shared memory. As can be seen, speedups are found to remain under one in both the cases. At a low percentage of plasticity, little improvement can be observed for the single kernel *DbD* strategy, but for the remaining part, the usage of shared memory has a detrimental effect on the performance. This is primarily due to the fact that data from the global memory and the shared memory cannot be accessed using the same variable. This leads to an implementation issue where a condition statement must be used to choose variables. The conditional statement leads to the problem of thread divergence on the GPU, which has a negative effect on performance.

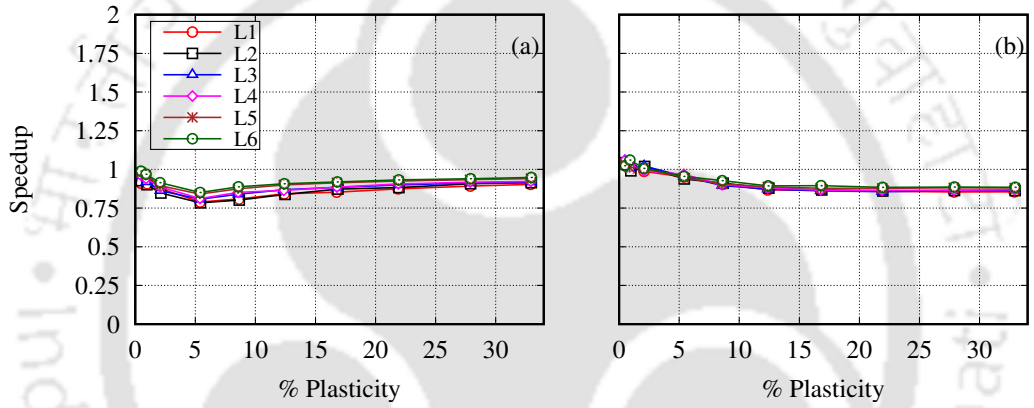


Figure 5.9: Speedups in kernel timings for L-bracket when shared memory is used to store elastic elemental tangent matrix. The figure shows speedups by (a) Single kernel *NbN* strategy and (b) Single kernel *DbD* strategy.

The speedups in kernel timings by different matrix-free SpMV strategies over the split kernel *NbN* strategy are shown in Fig. 5.10. The speedup is computed for all the mesh sizes and plotted against percentage plasticity. The speedup by single kernel *NbN* strategy (as shown in Fig. 5.10a) is found to be the highest at lower plasticity levels, achieving up to 2.5× for less than 5% plasticity. The speedups decrease sharply in the range of 5–20% plasticity, and thereafter, becomes stable at 1.7× for higher percentage of plasticity. Figure 5.10b shows speedups achieved by split kernel *DbD* strategy over split kernel *NbN* strategy, where the obtained speedup is small at lower plasticity levels but increases to 2.2× for higher percentage of plasticity. Similar speedup profile is exhibited by single kernel *DbD* strategy in Fig. 5.7c where even better speedups can be seen. The superior speedups by *DbD* strategies indicate its suitability for problems with higher amount of plasticity. Looking at Fig. 5.10, one can observe that the best speedup for

lower amount of plastic deformation is achieved by single kernel *NbN* strategy, whereas the highest speedup for moderate to large percentage of plasticity is obtained by single kernel *DbD* strategy. In order to assess the relative performance, the speedups by single kernel *DbD* strategy over single kernel *NbN* and split kernel *DbD* strategies are presented in Figs. 5.11 and 5.12, respectively. It can be seen from Fig. 5.11 that the performance of *DbD* strategy is found better only when certain value of percentage of plasticity is reached. Below this value the performance of *NbN* strategy is found better. Figure 5.12 shows speedups by single kernel *DbD* strategy over split kernel *DbD* strategy, where speedups varies from 1.2× at low plasticity levels to 1.5× at higher plasticity levels.

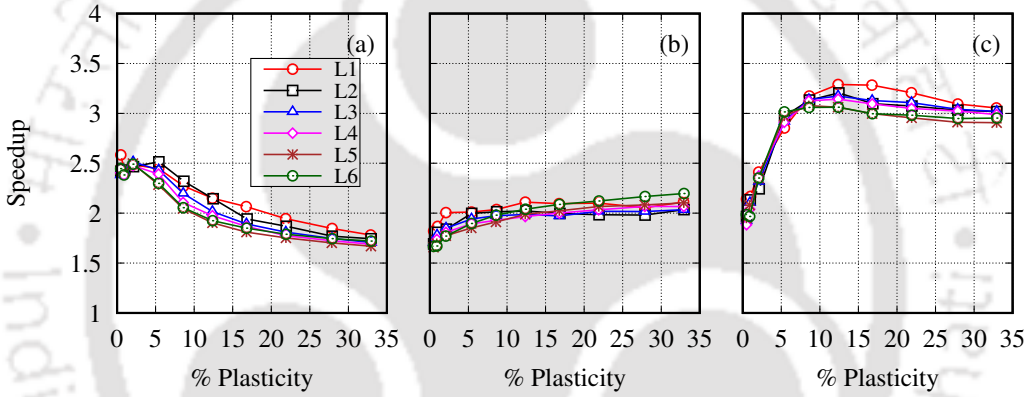


Figure 5.10: Speedups in kernel timings for L-bracket with respect to split kernel *NbN* strategy by (a) Single kernel *NbN* strategy, (b) Split kernel *DbD* strategy, (c) Single kernel *DbD* strategy.

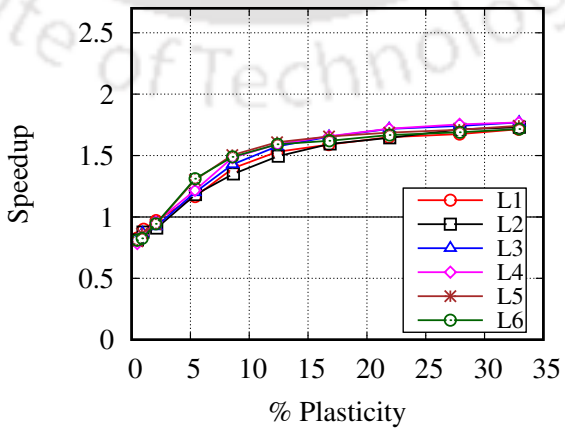


Figure 5.11: Speedups in kernel timings for L-bracket by single kernel *DbD* strategy over single kernel *NbN* strategy.

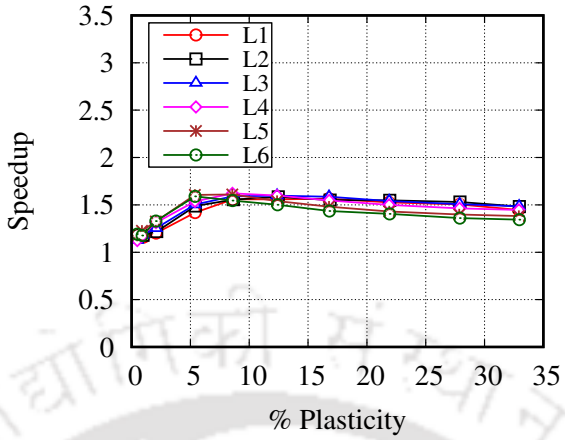


Figure 5.12: Speedups in kernel timings for L-bracket by single kernel *DbD* strategy over split kernel *DbD* strategy.

The wall-clock timings for elastoplastic analysis of L-bracket are presented in Fig. 5.13. One can observe that the least simulation timing is obtained with single kernel *DbD* strategy, followed by split kernel *DbD* strategy, single kernel *NbN* strategy and split kernel *NbN* strategy. As a result of the proposed single kernel strategy, wall-clock timing is reduced from 11153.1 sec to 4097.1 sec for the finest mesh consisting of 7.1 million DOFs and 32% plasticity. The single kernel version of both *NbN* and *DbD* strategies achieve reduced timings for all the mesh sizes from their split kernel counterparts. The speedups in wall-clock time achieved by different strategies over split kernel *NbN* strategy are displayed in Fig. 5.14. The speedup curve follows the same pattern as the speedup in kernel timings. For lower amount of plastic deformation, single kernel *NbN* strategy achieves the highest speedup of up to 2.5×. As the percentage plasticity increases, the performance of *DbD* strategy becomes better than *NbN* strategy, reaching up to 3.5× for the proposed single kernel variation. The performance of single kernel *NbN* and *DbD* strategy in wall-clock timings is compared in Fig. 5.16. The speedup curve shows better performance by *NbN* strategy at lower plasticity levels while *DbD* strategy achieves better performance at higher percentage of plasticity. This result is found to be consistent with the one observed in kernel time (refer Fig. 5.11). Figure 5.16 shows improved performance by single kernel strategy over split kernel strategy in DOF-based thread assignment.

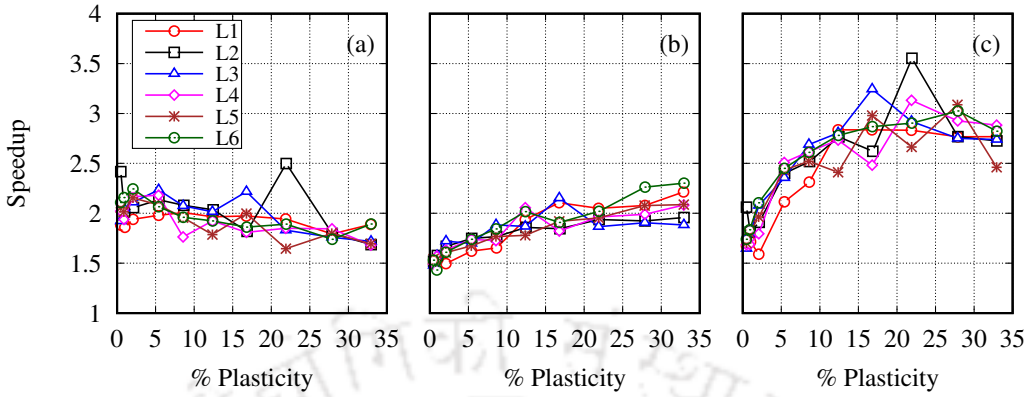


Figure 5.14: Speedups in wall-clock timings for L-bracket with respect to split kernel *NbN* strategy by (a) Single kernel *NbN* strategy, (b) Split kernel *DbD* strategy, (c) Single kernel *DbD* strategy.

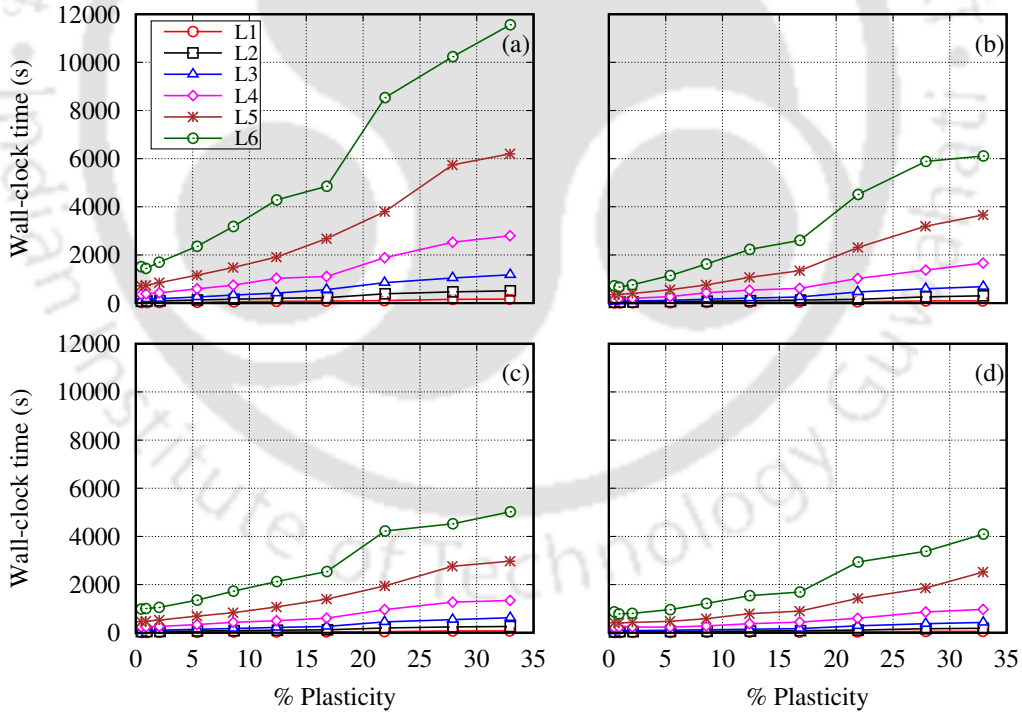


Figure 5.13: Wall-clock timings (seconds) for L-bracket by (a) Split kernel *NbN* strategy, (b) Single kernel *NbN* strategy, (c) Split kernel *DbD* strategy and (d) Single kernel *DbD* strategy.

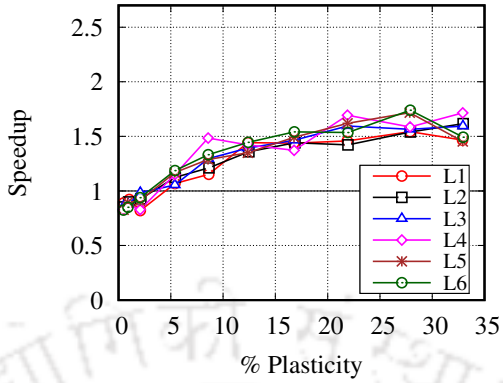


Figure 5.15: Speedups in wall-clock timings for L-bracket by single kernel *DbD* strategy over single kernel *NbN* strategy.

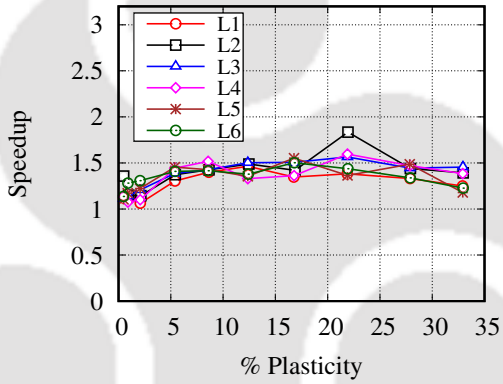


Figure 5.16: Speedups in wall-clock timings for L-bracket by single kernel *DbD* strategy over split kernel *DbD* strategy.

5.2.2.2 3D Cantilever beam

A 3D cantilever beam with end load is taken as another benchmark example (Ding et al. 2008). Figure 5.17 shows the geometry and boundary conditions of the cantilever beam. The problem parameters are considered as: Young's modulus $E = 200$ GPa, Poisson's ratio $\nu = 0.3$, hardening modulus $H = 20$ GPa and initial yield stress $\sigma_y = 250$ MPa. Four loads are taken to obtain different levels of plasticity in the body, such as 10500 N, 12500 N, 14000 N and 17000 N. The mesh for numerical simulations is shown in Table 5.2 with various levels of refinement.

The kernel timings for the matrix-free SpMV operation are presented in Fig. 5.18. It can be observed that kernel timings vary linearly with percentage plasticity for all the four strategies. The kernel time increases both with increasing mesh size and the amount

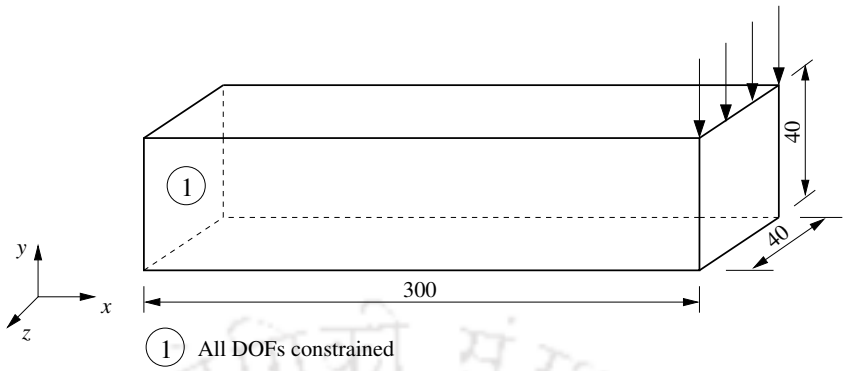


Figure 5.17: A cantilever beam with end load. All dimensions are taken in millimeters (mm).

Table 5.2: Mesh for the cantilever beam example.

Mesh	Elements	Nodes	DOFs
C1	204 300	219 108	657 324
C2	820 224	857 157	2 571 471
C3	1 612 800	1 670 729	5 012 187
C4	2 244 500	2 316 624	6 949 872

of plasticity. The least kernel timings are achieved by the proposed single kernel *DbD* strategy for higher amount of plasticity and single kernel *NbN* strategy for low amount of plasticity. The highest kernel timings are observed with split kernel *NbN* strategy. The single kernel strategy is found performing better, as both *NbN* and *DbD* variants achieve reduced kernel timings than their split kernel counterparts. The speedups in kernel timings are displayed in Fig. 5.19, where kernel timings of split kernel *NbN* strategy are divided by timings of different strategies and plotted against percentage plasticity. The speedup curves are found following the same profile as observed in the case of L-bracket benchmark. The single kernel *NbN* strategy achieves the highest speedup for low percentage plasticity, reaching up to 3.2× for less than 5% plasticity. When the amount of plasticity is high, the *DbD* strategy is found performing well. The single kernel *DbD* strategy achieves the highest speedup of up to 3.4× for higher amount of plasticity. A relative comparison of single kernel *NbN* and *DbD* strategies is presented in Fig. 5.20, where both the strategies achieve best performance in different regions of the plot, as found in L-bracket example. The comparison of single kernel *DbD* and split kernel *DbD* is presented in Fig. 5.21, where the proposed single kernel strategy achieves speedup of up to 1.6×.

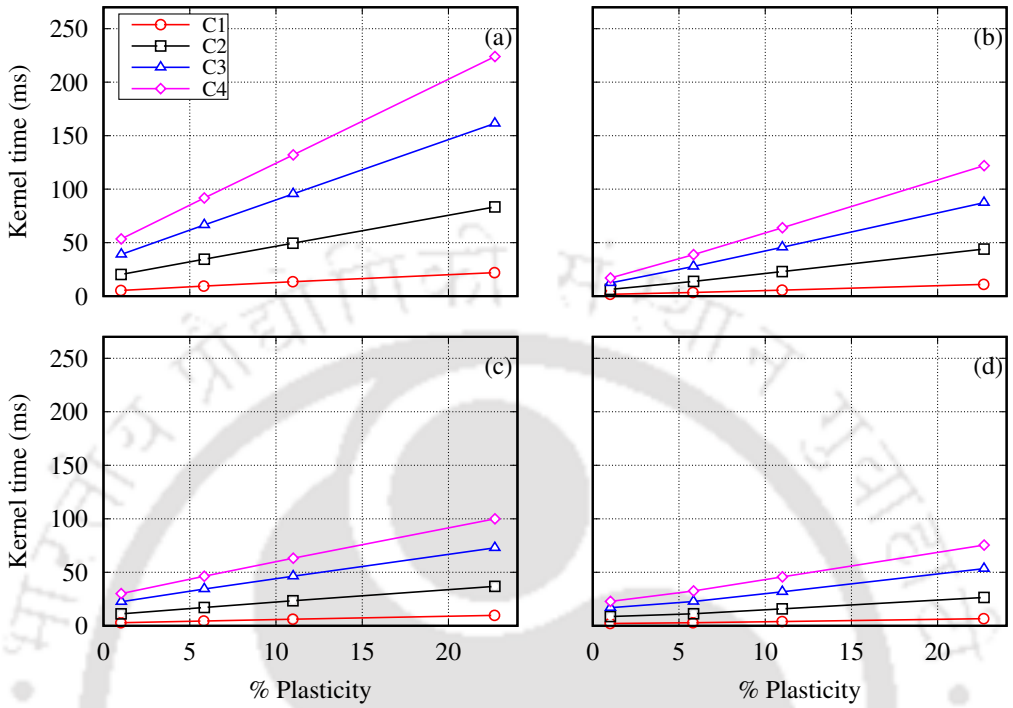


Figure 5.18: Kernel timings (milliseconds) for cantilever beam by (a) Split kernel *NbN* strategy, (b) Single kernel *NbN* strategy, (c) Split kernel *DbD* strategy and (d) Single kernel *DbD* strategy.

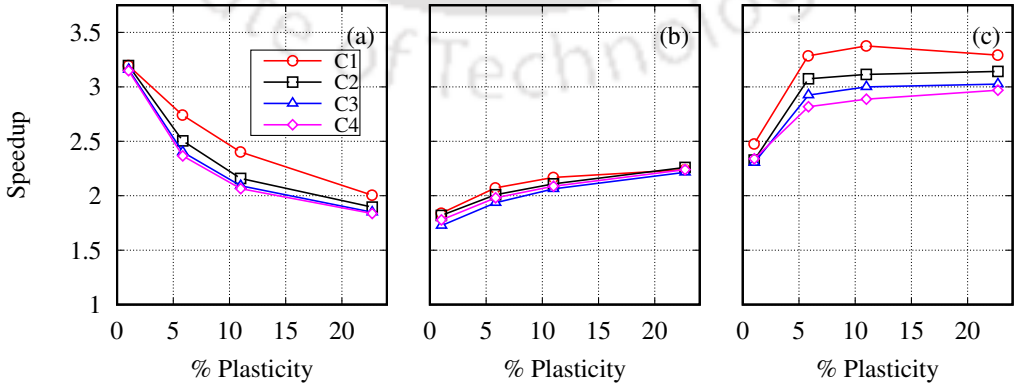


Figure 5.19: Speedups in kernel timings for cantilever beam with respect to split kernel *NbN* strategy by (a) Single kernel *NbN* strategy, (b) Split kernel *DbD* strategy, (c) Single kernel *DbD* strategy.

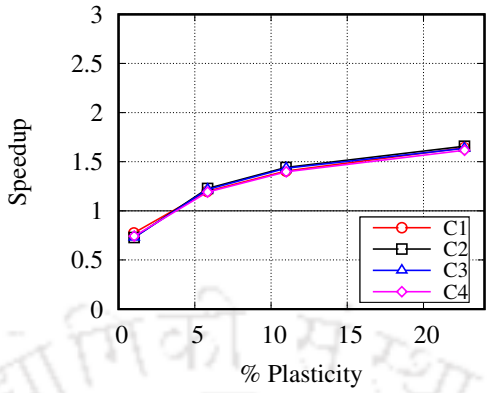


Figure 5.20: Speedups in kernel timings for cantilever beam by single kernel *DbD* strategy over single kernel *NbN* strategy.

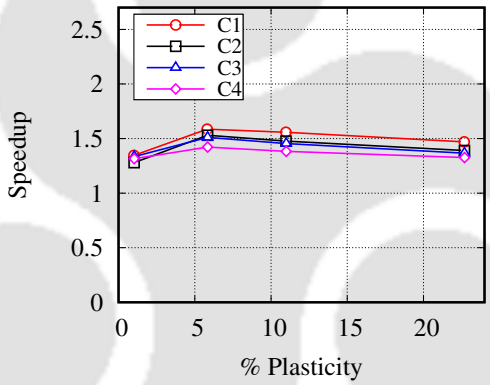


Figure 5.21: Speedups in kernel timings for cantilever beam by single kernel *DbD* strategy over split kernel *DbD* strategy.

The wall-clock timings for elastoplastic analysis of cantilever beam are presented in Fig. 5.22. As can be seen in the figure, the highest wall-clock timings are obtained with split kernel *NbN* strategy. The proposed single kernel *DbD* strategy achieves the least wall-clock timings at higher levels of plasticity and single kernel *NbN* strategy achieves the best timings for lower levels of plasticity. For the finest mesh with 6.9 million DOFs and 22.7% plasticity, the proposed single kernel strategy is able to reduce wall-clock timing from 7415 sec to 2865.18 sec. A reduction of approximately 3347.7 sec in *NbN* strategy and 1005.5 sec in *DbD* strategy is observed for the proposed single kernel strategy as compared with split kernel strategy, for the finest mesh. The speedups in wall-clock timings by different strategies over split kernel *NbN* strategy are presented in Fig. 5.23. The single kernel *NbN* strategy achieves the highest speedup of 2.6× at low

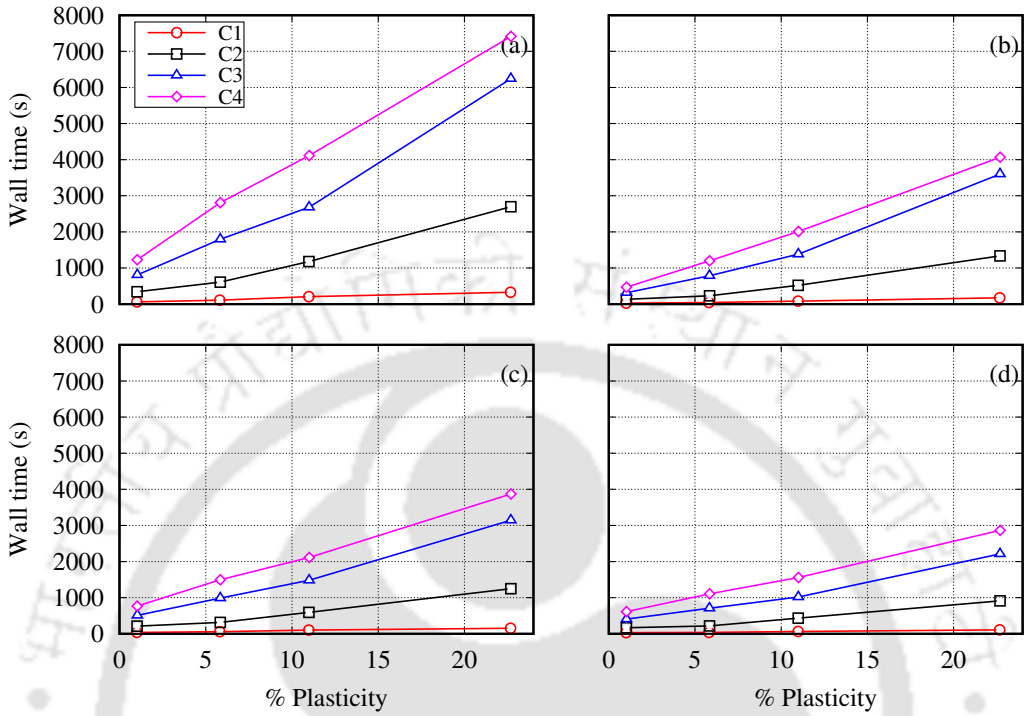


Figure 5.22: Wall-clock timings (seconds) for cantilever beam by (a) Split kernel *NbN* strategy, (b) Single kernel *NbN* strategy, (c) Split kernel *DbD* strategy and (d) Single kernel *DbD* strategy.

percentage plasticity but the speedup is found decreasing with increasing plasticity. The highest speedup of up to 3.1× at increased percentage plasticity is achieved by single kernel *DbD* strategy. In general, *DbD* strategies show better performance as the amount of plasticity increases. The speedups by single kernel *DbD* strategy over single kernel *NbN* strategy are shown in Fig. 5.24. It can be clearly observed that *NbN* strategy is most suitable for low plasticity problem and *DbD* strategy should be used for high plasticity problem. The speedups by single kernel *DbD* strategy over split kernel *DbD* strategy are presented in Fig. 5.25, where speedups of up to 1.6× is obtained at higher percentage plasticity.

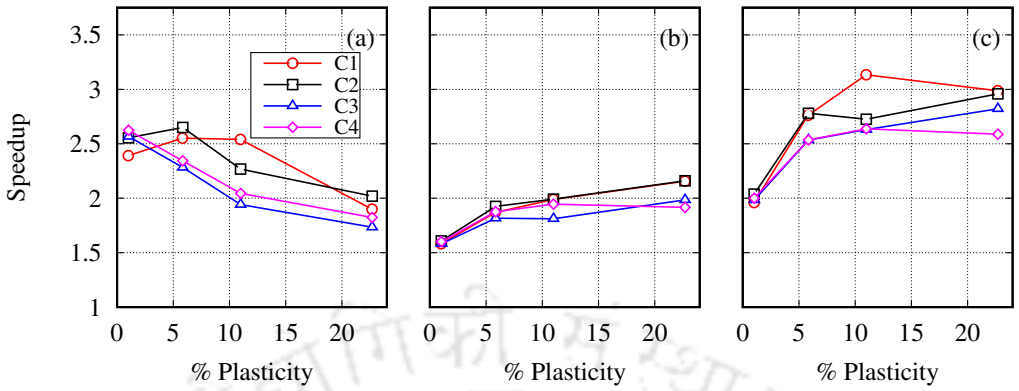


Figure 5.23: Speedups in wall-clock timings for cantilever beam with respect to split kernel *NbN* strategy by (a) Single kernel *NbN* strategy, (b) Split kernel *DbD* strategy, (c) Single kernel *DbD* strategy.

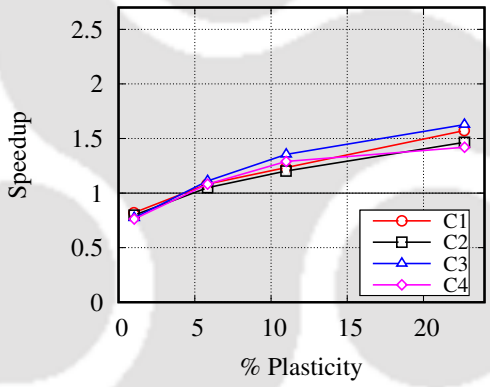


Figure 5.24: Speedups in wall-clock timings for cantilever beam by single kernel *DbD* strategy over single kernel *NbN* strategy.

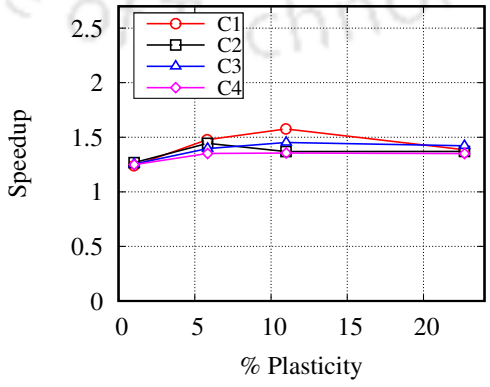


Figure 5.25: Speedups in wall-clock timings for cantilever beam by single kernel *DbD* strategy over split kernel *DbD* strategy.

5.2.2.3 Plate with multiple holes

This example is adapted from Yusa et al. (2018) where circular holes are replaced by square holes to generate voxel-based finite element mesh. The geometry of a flat plate with four square holes along with boundary conditions is shown in Fig. 5.26. The parameters for elastoplastic analysis are considered as: Young's modulus $E = 200$ GPa, Poisson's ratio $\nu = 0.3$, hardening modulus $H = 20$ GPa and initial yield strength $\sigma_{y0} = 200$ MPa. The details of finite element mesh and various levels of refinement used for the numerical experiments are given in Table 5.3.

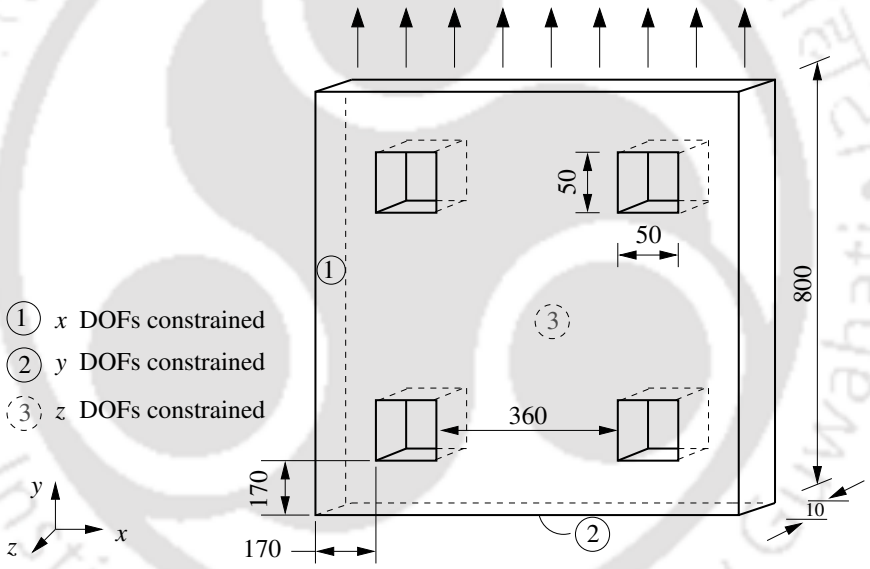


Figure 5.26: A plate with four square holes. All dimensions are taken in millimeters (mm).

Table 5.3: Mesh for the plate with multiple holes example.

Mesh	Elements	Nodes	DOFs
P1	170 100	229 188	687 564
P2	787 500	950 982	2 852 946
P3	1 360 800	1 595 979	4 787 937
P4	2 160 900	2 480 776	7 442 328

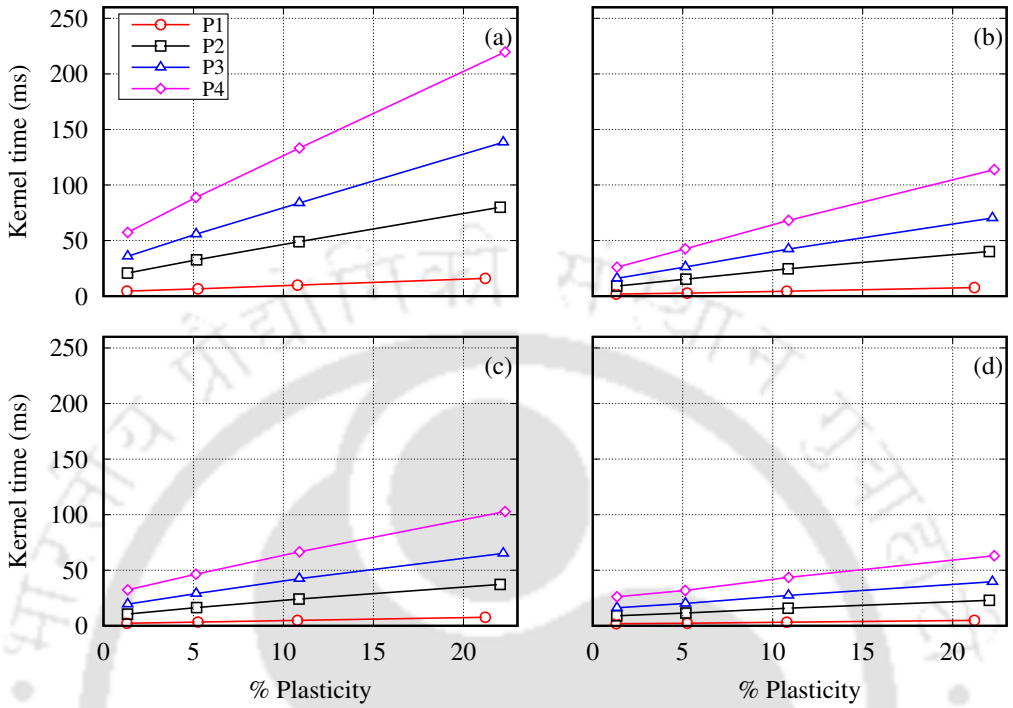


Figure 5.27: Kernel timings (milliseconds) for plate with multiple holes by (a) Split kernel *NbN* strategy, (b) Single kernel *NbN* strategy, (c) Split kernel *DbD* strategy and (d) Single kernel *DbD* strategy.

Figure 5.27 shows the kernel timings for the matrix-free SpMV, where kernel timings vary linearly with percentage plasticity for all four strategies. Similar to previous examples, the least kernel timing is achieved with the proposed single kernel *DbD* strategy. As evident by comparing Figs. 5.27a and 5.27b for *NbN* strategy, and Figs. 5.27c and 5.27d for *DbD* strategy, the proposed single kernel strategies achieve significantly less kernel timings than the split kernel strategy. The speedups in kernel timings are presented with respect to the split kernel *NbN* strategy in Fig. 5.28. Like the previous two examples, the speedup curve for the single kernel *NbN* strategy has the highest value of 2.4× at lower plasticity levels and shows a decreasing trend as percentage plasticity increases. The *DbD* strategies show increasing speedup with percentage plasticity and achieves the highest value of 3.5× at 22.3% plasticity. The speedups by single kernel *DbD* strategy over single kernel *NbN* strategy are presented in Fig. 5.29. Unlike previous examples, the single kernel *DbD* strategy shows comparable kernel timings with

the single kernel *NbN* strategy at low percentage of plasticity. The speedups by single kernel *DbD* strategy over split kernel *DbD* are shown in Fig. 5.30, where speedups of up to 1.6× are obtained.

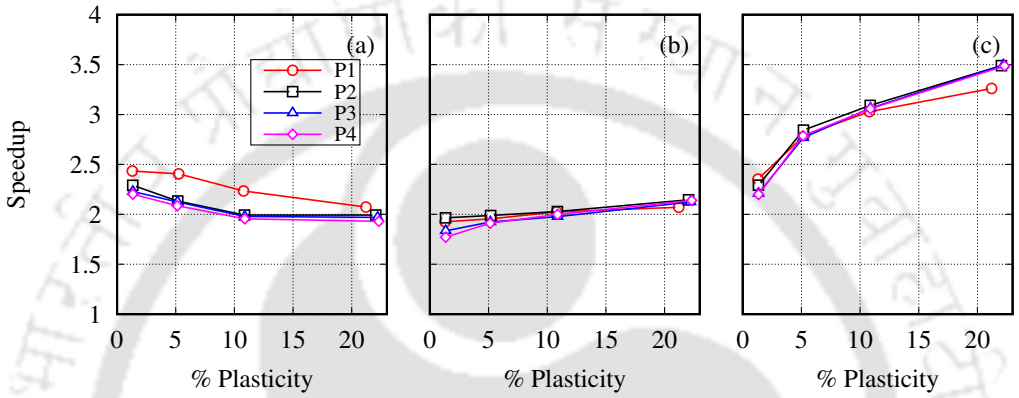


Figure 5.28: Speedups in kernel timings for plate with multiple holes with respect to split kernel *NbN* strategy by (a) Single kernel *NbN* strategy, (b) Split kernel *DbD* strategy, (c) Single kernel *DbD* strategy.

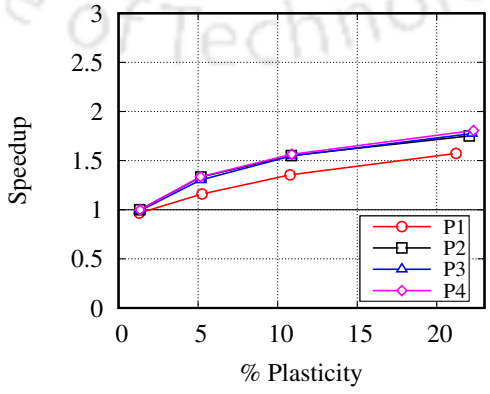


Figure 5.29: Speedups in kernel timings for plate with multiple holes by single kernel *DbD* strategy over single kernel *NbN* strategy.

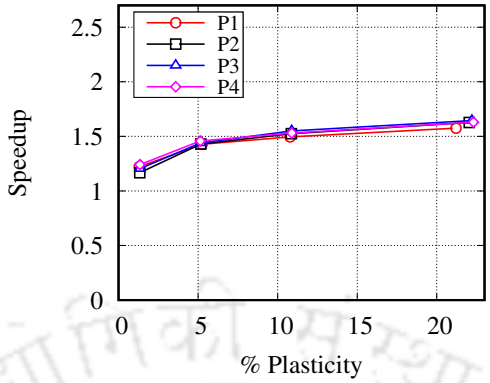


Figure 5.30: Speedups in kernel timings for plate with multiple holes by single kernel *DbD* strategy over split kernel *DbD* strategy.

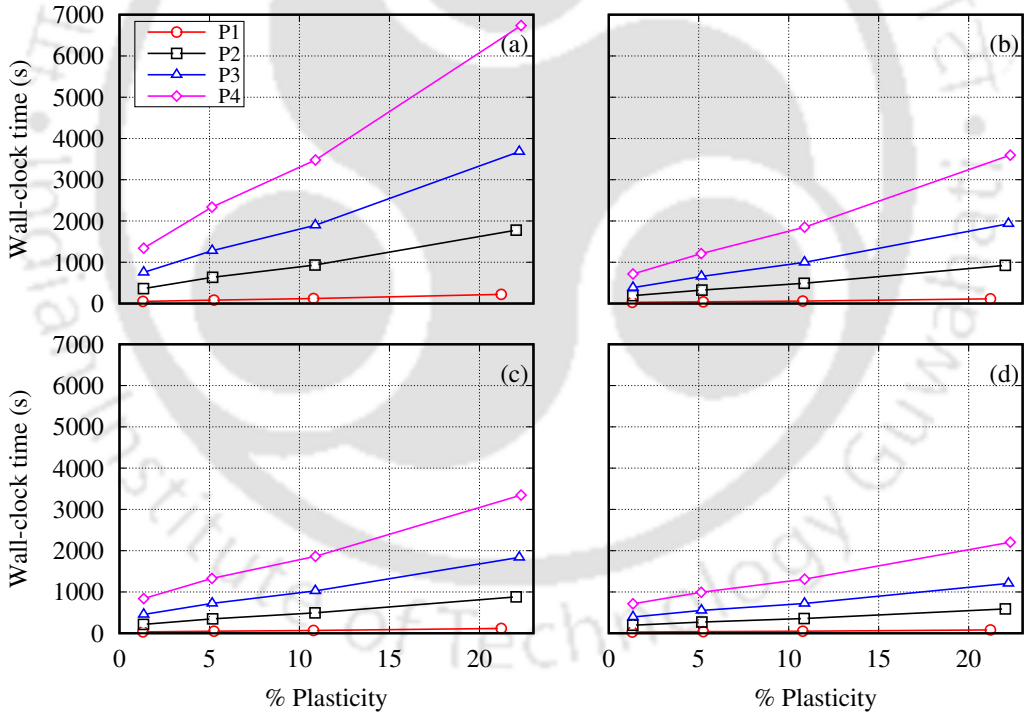


Figure 5.31: Wall-clock timings (seconds) for plate with multiple holes by (a) Split kernel *NbN* strategy, (b) Single kernel *NbN* strategy, (c) Split kernel *DbD* strategy and (d) Single kernel *DbD* strategy.

The wall-clock timings for elastoplastic analysis of the plate with multiple holes example are shown in Fig. 5.31. In accordance with previous examples, the highest wall-clock timings are observed with the split kernel *NbN* strategy, whereas the least wall-clock

timings are achieved with the single kernel *DbD* strategy. As a result of the proposed strategies, the wall-clock timing of 6736.2 sec is reduced to 2204.9 sec for the finest mesh consisting of 7.4 million DOFs at 22.3% of plasticity. Figure 5.32 shows the speedups in wall-clock timings by different strategies over the split kernel *NbN* strategy. The speedup profiles are found to be similar as previous examples, where the single kernel *NbN* strategy achieves speedup of 1.9× at low plasticity levels and the single kernel *DbD* strategy achieves speedup of 3.1× for higher percentage of plasticity. The speedups by the single kernel *DbD* strategy over single kernel *NbN* and split kernel *DbD* strategies are plotted in Figs. 5.33 and 5.34, respectively. The performance results (kernel timings, wall-clock timings and speedups) for the plate with multiple holes example are found to be consistent with previous examples of L-bracket and cantilever beam. This establishes the potential benefits of the proposed matrix-free SpMV strategy for elastoplastic problems.

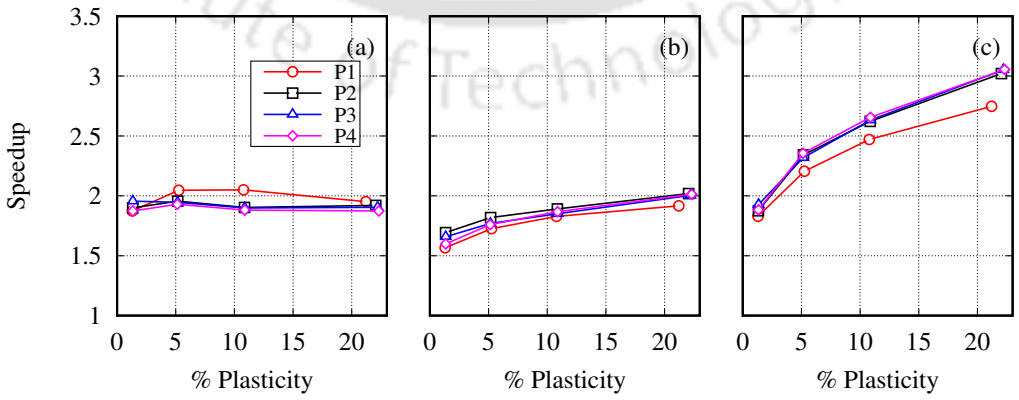


Figure 5.32: Speedups in wall-clock timings for plate with multiple holes with respect to split kernel *NbN* strategy by (a) Single kernel *NbN* strategy, (b) Split kernel *DbD* strategy, (c) Single kernel *DbD* strategy.

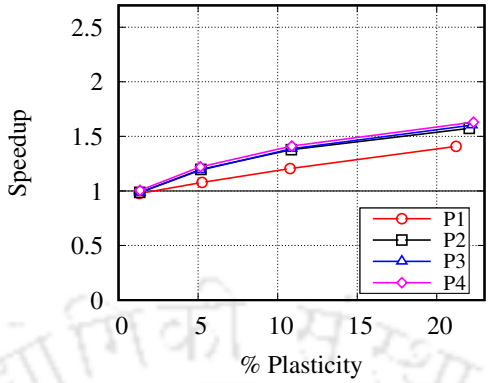


Figure 5.33: Speedups in wall-clock timings for plate with multiple holes by single kernel *DbD* strategy over single kernel *NbN* strategy.

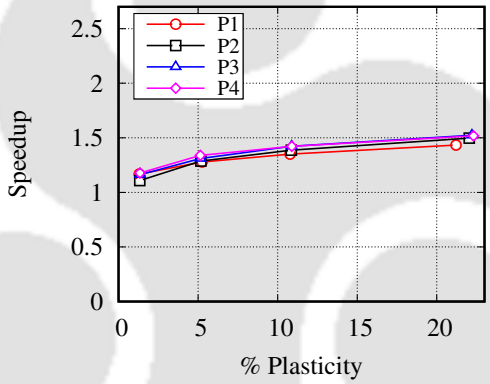


Figure 5.34: Speedups in wall-clock timings for plate with multiple holes by single kernel *DbD* strategy over split kernel *DbD* strategy.

5.2.3 Summary

In order to improve the performance of matrix-free SpMV computation, we proposed a single kernel computational strategy that prevents branching and provides efficient memory access. Further, based on the single kernel strategy, we proposed node-based and DOF-based parallel strategies to achieve efficient implementation of matrix-free SpMV on the GPU. The GPU implementations of the *NbN* and *DbD* strategies make efficient use of read-only cache and remain general enough to be applicable for voxel-based structured meshes generated by commercial software packages. Three large-scale benchmark examples in 3D for elastoplasticity were considered for the performance evaluations over different levels of mesh refinement and different amount of plasticity. Comparison

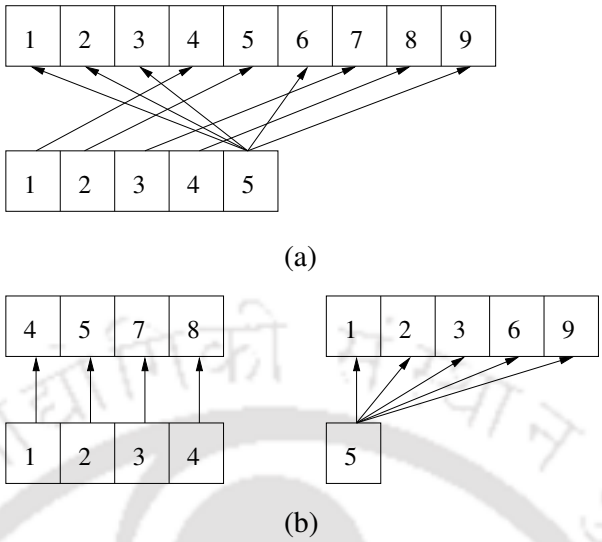


Figure 5.35: Memory access pattern in (a) Single kernel strategy (b) Split kernel strategy.

of matrix-free SpMV kernel timings revealed that the proposed *NbN* strategy achieves speedups up to 3.2× over the existing GPU-based split kernel strategy. However, the performance of *NbN* strategy was found to decrease as the amount of plasticity increased in the body. For moderate to high amount of plasticity($> 5\%$), the proposed *DbD* strategy showed the best performance achieving speedups up to 3.5× over GPU-based split kernel strategy. Based on the performance results, we conclude that the proposed matrix-free SpMV strategy makes efficient use of GPU resources and provides a uniform treatment of elastic and plastic states using a single compute kernel, unlike the previous strategy, which splits the computation into two kernels.

5.3 Proposed improved split kernel strategy [2]

The memory access pattern of the single kernel strategy is shown in Fig. 5.35. As can be seen, due to the mix of elastic and plastic elements, memory access to a warp cannot be perfectly coalesced. This increases the number of memory transactions to the expensive global memory of the GPU. An improved memory access is achieved when computations for elastic and plastic elements are performed separately. As shown in Fig. 5.35b, using separate kernels for both elastic and plastic elements enables coalesced memory access,

[2]The content of this section has been published in **International Journal for Numerical Methods in Engineering** (Kiran et al. 2024*a*).

reducing the number of transactions to the global memory. The splitting of computation into elastic and plastic parts has another advantage that specific optimizations can be applied to both kernels independently. However, kernel splitting increases the number of kernel launches.

In the improved split kernel strategy, the computation of SpMV using Eq. (5.8) uses two CUDA kernels for the GPU implementation. In the first kernel, computation of matrix-vector product for elastic zone is performed, whereas the second kernel is dedicated for computations in the plastic zone. Since the computation is performed using two mutually exclusive sets ($\mathcal{E}^{(e)}$ and $\mathcal{E}^{(p)}$), the improved split kernel strategy require segregation of elastic and plastic elements. In the numerical procedure of elastoplasticity, the number of plastic elements and their locations evolve with Newton iterations until final distribution is obtained. Therefore, the sets $\mathcal{E}^{(e)}$ and $\mathcal{E}^{(p)}$ must be updated in each Newton iterations.

The GPU implementation of the proposed split kernel strategy is illustrated in Fig. 5.36. Following the procedure for single kernel strategy (refer to Fig. 5.5), an array is obtained that contains the segregated list of plastic elements number and elastic elements number incremented by e_t (third array from the top). A new array is obtained by subtracting e_t from all entries except plastic element numbers, generating actual element numbers for elastic elements. This gives the set of elastic elements ($\mathcal{E}^{(e)}$). Next, the input data initially stored according to global element number is reordered according to elements in $\mathcal{E}^{(e)}$ and $\mathcal{E}^{(p)}$ to use in matrix-free SpMV kernels for elastic and plastic elements. This has two advantages. First, GPU kernels can be implemented without making explicit use of $\mathcal{E}^{(e)}$ and $\mathcal{E}^{(p)}$, as input data is now stored in a predetermined sequence. Second, the memory access to input data becomes coalesced. As shown in Fig. 5.37a, if input data is not reordered, both the kernels make strided access to the global memory. Reordering of data according to the list of elements given in $\mathcal{E}^{(e)}$ and $\mathcal{E}^{(p)}$ leads to access of consecutive memory locations by consecutive threads (see Fig. 5.37b). It is noted that the improved split kernel strategy does not need any additional array for matrix-free SpMV computation like the single kernel strategy.

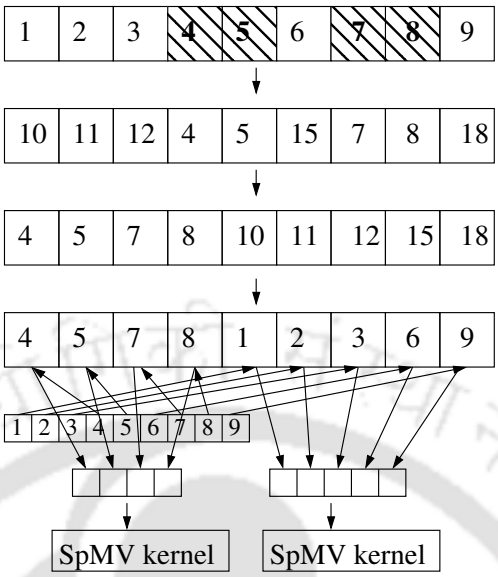


Figure 5.36: Illustration of improved split kernel strategy.

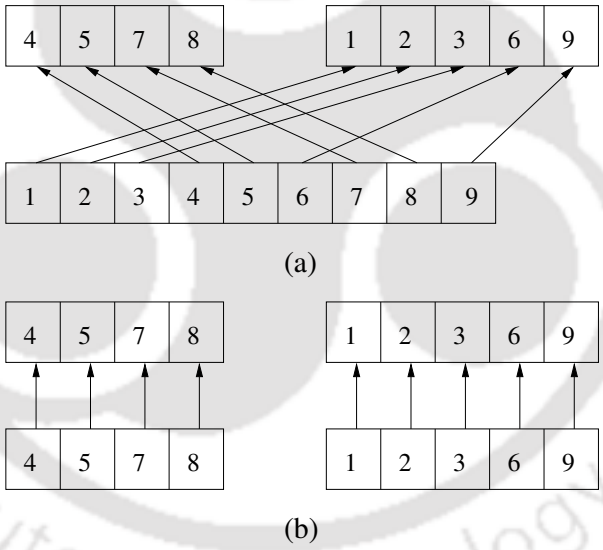


Figure 5.37: Effect of reordering of data in improved split kernel strategy.

5.3.1 Element-by-Element SpMV strategy

The GPU implementation of *EbE* strategy can vary depending on the thread assignment for each element. As shown in Section 4.1.3, three kinds of thread assignment can be used: single thread per element, single thread per node and single thread per DOF. The work presented in Ratnakar et al. (2021) show single thread per element performing poorly than the other two. Therefore,the current work explores the remaining two types of thread assignment.

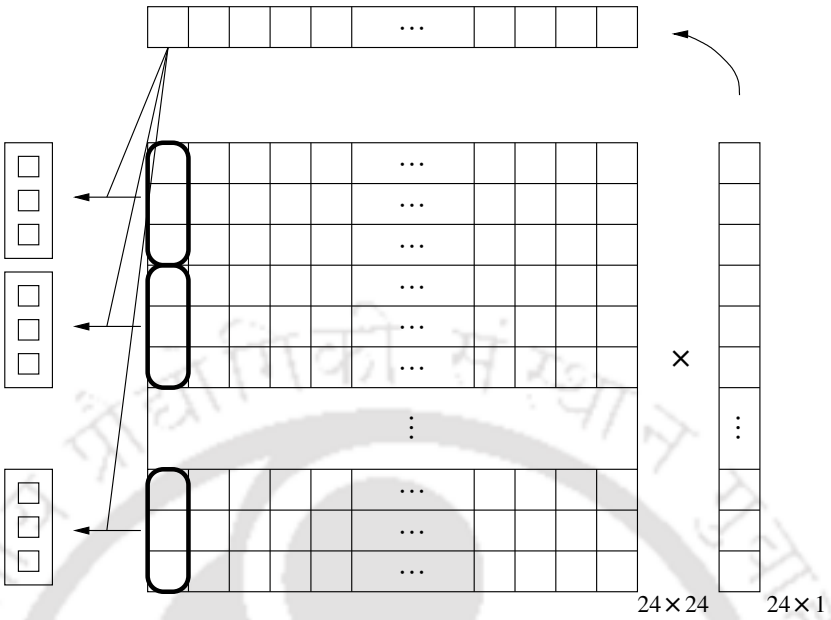


Figure 5.38: GPU implementation of EbE_{Node} strategy.

5.3.1.1 EbE_{Node} strategy

In this strategy, 8 threads corresponding to 8 nodes of a linear hexahedral element are assigned per element to compute matrix-free SpMV. Since the current work uses three DOFs per node, a thread performs multiplication for three rows in the elemental tangent matrix. The GPU implementation of EbE_{Node} strategy is illustrated in Fig. 5.38. Each thread accesses a scalar value from vector $\mathbf{p}^e$ and multiplies with a column vector (3×1) from its assigned sub-matrix. The scalar-vector product is repeated for all columns of the elemental tangent matrix, and results are accumulated into registers. It can be observed that in each scalar-vector product, one common scalar value is accessed by all 8 threads operating over an element, creating a broadcast. Since broadcast from shared memory is fast, the multiplying vector $\mathbf{p}^e$ is stored in the shared memory of each block for computation.

In the EbE_{Node} strategy, the elemental tangent matrix is reordered to achieve coalesced memory access. Figure 5.39 shows an elemental tangent matrix with non-zero values numbered in a row-wise manner. Since 8 threads assigned to an element need to access 24 values from each column, a total of 3 passes is required. The access of the elemental tangent matrix when data is not reordered is shown in Fig. 5.40a. It can be seen that the threads make strided access in the first pass. To achieve coalesced access,

1	2	3	4	⋯	23	24
25	26	27	28	⋯	47	48
49	50	51	52	⋯	71	72
73	74	75	76	⋯	95	96
97	98	99	100	⋯	119	120
121	122	123	124	⋯	143	144
⋮						
505	506	507	508	⋯	527	528
529	530	531	532	⋯	551	552
553	554	555	556	⋯	575	576

24×24

Figure 5.39: An elemental tangent matrix with non-zero values numbered in a row-wise manner.

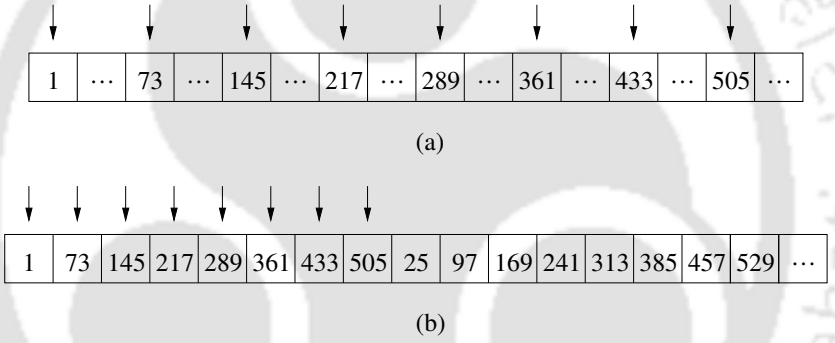


Figure 5.40: Elemental tangent matrix access pattern in EbE_{Node} strategy (a) Data is not reordered (b) Data is reordered.

column entries being read in the first pass must be stored in consecutive memory locations. The reordered elemental tangent matrix is shown in Fig. 5.40b. As can be seen, the values at locations 1, 73, 145, etc. that are read in one pass are stored in consecutive memory locations to facilitate coalesced access. If more than one element is required for matrix-free SpMV computation, corresponding entries for all elements must be stored side-by-side.

5.3.1.2 EbE_{DOF} strategy

In this strategy, the number of threads equal to the total DOFs per element is assigned to compute matrix-vector product. In the case of the linear hexahedral element with 3 DOFs per node, 24 threads are assigned to one element. The GPU implementation

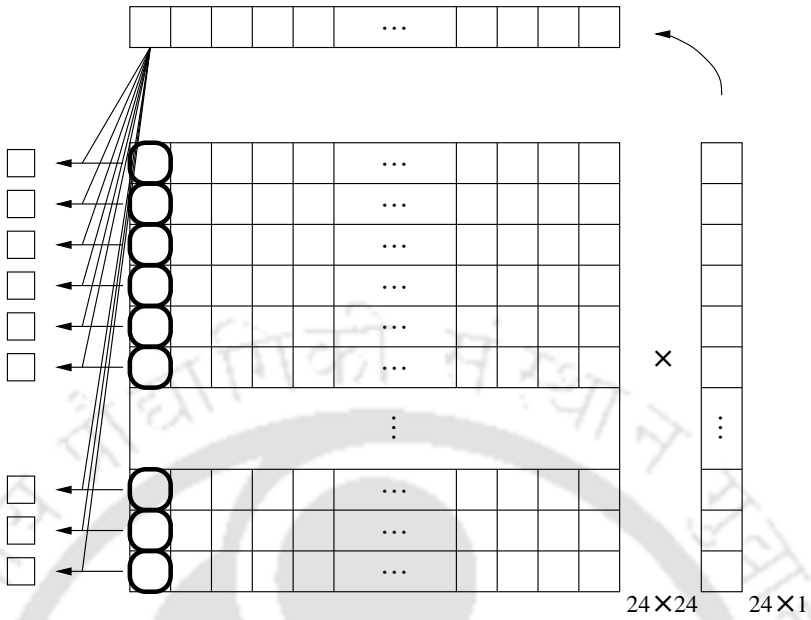


Figure 5.41: GPU implementation of EbE_{DOF} strategy.

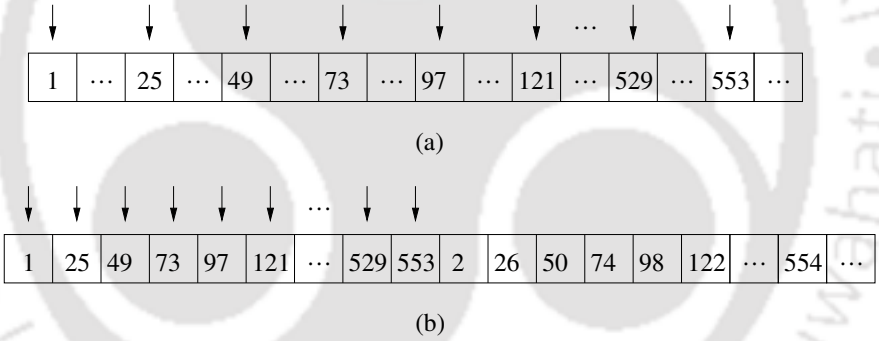


Figure 5.42: Elemental tangent matrix access pattern in EbE_{DOF} strategy (a) Data is not reordered (b) Data is reordered.

of EbE_{DOF} strategy is illustrated in Fig. 5.41. As can be seen, each thread performs multiplication for one row of the elemental tangent matrix in the form of vector-dot product. Threads read a common scalar value from vector $\mathbf{p}^e$ and multiply with a scalar value from its assigned row. The result of the multiplication is stored in a register. Similar to the EbE_{Node} strategy, $\mathbf{p}^e$ vector is cached in the shared memory for efficient broadcast access. Each thread in EbE_{DOF} strategy perform computation in a truly independent manner, and therefore, no inter-warp synchronization is required.

For efficient implementation of EbE_{DOF} strategy, the elemental tangent matrix is reordered and stored in a column-wise manner. As shown in Fig. 5.42a, if the elemental tangent matrix is stored in a row-wise manner (according to Fig. 5.39), threads make

strided access, increasing the memory transactions. The column-wise arrangement facilitates coalesced access, as shown in Fig. 5.42b. If more than one elemental tangent matrix is used for matrix-free SpMV, each corresponding column of elemental tangent matrices is stored beside each other.

5.3.2 GPU implementation

5.3.2.1 Single kernel EbE_{Node} strategy

The computation of matrix-free SpMV for elastoplasticity is done by the single kernel strategy (see Section 5.2) and implemented on GPU using EbE_{Node} strategy. Algorithm 19 shows key steps in GPU implementation of single kernel EbE_{Node} strategy. The input data for GPU kernel consists of array `P` (vector $\mathbf{p}$) to perform multiplication, elemental tangent matrices for both elastic and plastic elements in array `K_e`, `K_index` array to keep indices for accessing elemental tangent matrices, connectivity of each element in `connectivity` array and the total number of plastic elements *npelem*. The results of multiplication are stored in output array `G`. A total of 8 threads corresponding to 8 nodes are assigned to each element for the computation of matrix-vector product, as shown in line 2. For each element, global connectivity is accessed and stored in a register of each thread as *global_dof* and $\mathbf{p}$ vector is stored as `sP_e` in the shared memory. Next, a block-level synchronization barrier is called to synchronize all threads before the computation proceeds further. The computation of matrix-vector product is implemented inside an outer loop over elemental DOFs and an inner loop over nodal DOFs, as shown in line 8. In line 10, elemental tangent matrices are accessed using `K_index` array in a coalesced manner (as discussed in Section 5.2) to multiply with scalar values from `sP_e` array. The computed multiplication results are accumulated into registers and later assembled into the global vector $\mathbf{g}$ using *global_dof*.

5.3.2.2 Single kernel EbE_{DOF} strategy

In this strategy, EbE_{DOF} strategy is used to implement the single kernel matrix-free SpMV for elastoplasticity. The GPU implementation of single kernel EbE_{DOF} strategy is presented in Algorithm 20. The input data and major steps in the computation

Algorithm 19 Single kernel EbE_{Node} matrix-free strategy for elastoplasticity.

Input: `P`, `K_e`, `K_index`, `connectivity`, $npelem$
Output: `G`

1: $threadId \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
2: $elemNo \leftarrow threadId/8$
3: $elaneId \leftarrow threadId\%8$ ▷ $elaneId \in (0, 1, ..., 7)$
4: **if** $elemNo < maxElem$ **then**
5: $global\_dof \leftarrow$ `connectivity`$[elemNo * 8 + elaneId]$
6: `sP_e` $\leftarrow$ get_y_vector(`P`, $global\_dof$)
7: __synthreads()
8: **for** $i = 1$ to e_{dof} **do** ▷ Loop over DOFs per element
9: **for** $j = 1$ to n_{dof} **do** ▷ Loop over DOFs per node
10: `val`$[j]$ += `K_e`[`K_index`$[elemNo] * 8 + i * 24 * npelem + j * npelem * 8 + elaneId$] * `sP_e`$[i]$
11: **end for**
12: **end for**
13: `G` $\leftarrow$ Accumulate(`G`, `val`, $global\_dof$)
14: **end if**

remain the same as the single kernel EbE_{Node} strategy. As shown in line 2, 24 threads corresponding to 24 DOFs of an element are assigned to compute the matrix-vector product. In this strategy, 24 threads assigned to an element may spread over more than one warp. Therefore, it is essential to use block synchronization barrier, as shown in line 7, after accessing shared memory for consistent usage. Here, unlike the single kernel EbE_{Node} strategy, the computation of matrix-vector product is performed inside only one loop, as shown in line 9. The elemental tangent matrices are accessed in coalesced manner by using suitable indices as suggested in Section 5.3.1.2. The multiplication results are initially stored in a register variable and later assembled into the global vector in a coalesced manner.

5.3.2.3 Improved split kernel EbE_{Node} strategy

In this strategy, two GPU kernels are used for matrix-free SpMV as suggested by the improved split kernel strategy in Section 5.3. Both the kernels are implemented using EbE_{Node} strategy. Algorithm 21 presents the computational steps in a dedicated kernel for elastic elements. It can be seen in line 4 that the computation is performed only for elements in the elastic zone ($maxEElem$). Consequently, only one elemental tangent

Algorithm 20 Single kernel EbE_{DOF} matrix-free strategy.

Input: `P`, `K_e`, `K_index`, `connectivity`, *npelem*
Output: `G`

1: $threadId \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
2: $elemNo \leftarrow threadId/24$
3: $elaneId \leftarrow threadId\%24$ ▷ $elaneId \in (0, 1, ..., 24)$
4: **if** $elemNo < maxElem$ **then**
5: $global\_dof \leftarrow$ `connectivity`$[elemNo * 8 + elaneId/3]$
6: `sP_e` $\leftarrow$ get_y_vector(`P`, $global\_dof$)
7: __synthreads()
8: **for** $i = 1$ to e_{dof} **do** ▷ Loop over DOFs per element
9: `val` += `K_e`$[$`K_index`$[elemNo] * 24 + i * 24 * npelem + elaneId]$ * `sP_e`$[i]$
10: **end for**
11: `G` $\leftarrow$ Accumulate(`G`, `val`, $global\_dof$)
12: **end if**

matrix is used for the computation of matrix-vector product as shown in line 11. The elemental tangent matrix is accessed through shared memory for efficient memory access (see line 7). The computed values are accumulated in the global memory in line 14.

The GPU implementation of a dedicated kernel for elements in the plastic zone is explained in Algorithm 22. The computations are performed only for the maximum number of plastic elements *npelem*. Unlike the single kernel EbE_{Node} strategy, the access to `K_e` does not require the use of `K_index` array. Since elemental tangent matrices are reordered according to the list of plastic elements($\mathcal{E}^{(p)}$), indices are not required to locate an elemental tangent matrix. The array `K_e` is accessed in coalesced manner as discussed in Section 5.3.1.1.

5.3.2.4 Improved split kernel EbE_{DOF} strategy

In this strategy, both the kernels in the improved split kernel strategy are implemented on GPU using EbE_{DOF} strategy. Algorithm 23 presents the GPU implementation of the dedicate kernel for elastic elements. Since the computation for elastic elements uses only one elemental tangent matrix, it is cached in the shared memory, line 7. The matrix-vector product is computed in line 10 by reading values from the shared memory.

Algorithm 24 presents the computational steps in the dedicated kernel for plastic elements. The dependency on `K_index` is removed as elemental matrices are reordered

Algorithm 21 GPU kernel for elements in the elastic zone using improved split kernel EbE_{Node} matrix-free strategy.

Input: P, K_e, connectivity
Output: G

1: $threadId \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
2: $elemNo \leftarrow threadId/8$
3: $elaneId \leftarrow threadId\%8$ ▷ $elaneId \in (0, 1, ..., 7)$
4: **if** $elemNo < maxEElem$ **then**
5: $global\_dof \leftarrow$ connectivity$[elemNo * 8 + elaneId]$
6: sP_e $\leftarrow$ get_y_vector(P, $global\_dof$)
7: sK_e $\leftarrow$ get_k_matrix(K_e, $global\_dof$)
8: __synthreads()
9: **for** $i = 1$ to e_{dof} **do** ▷ Loop over DOFs per element
10: **for** $j = 1$ to n_{dof} **do** ▷ Loop over DOFs per node
11: val$[j]$ += sK_e$[i * 24 + j * 8 + elaneId]$ * sP_e$[i]$
12: **end for**
13: **end for**
14: G $\leftarrow$ Accumulate(G, val, $global\_dof$)
15: **end if**

Algorithm 22 GPU kernel for elements in the plastic zone using improved split kernel EbE_{Node} matrix-free strategy.

Input: P, K_e, connectivity, $npelem$
Output: G

1: $threadId \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
2: $elemNo \leftarrow threadId/8$
3: $elaneId \leftarrow threadId\%8$ ▷ $elaneId \in (0, 1, ..., 7)$
4: **if** $elemNo < nplem$ **then**
5: $global\_dof \leftarrow$ connectivity$[elemNo * 8 + elaneId]$
6: sP_e $\leftarrow$ get_y_vector(P, $global\_dof$)
7: __synthreads()
8: **for** $i = 1$ to e_{dof} **do** ▷ Loop over DOFs per element
9: **for** $j = 1$ to n_{dof} **do** ▷ Loop over DOFs per node
10: val$[j]$ += K_e$[elemNo*8+i*24*npelem+j*npelem*8+elaneId]$ * sP_e$[i]$
11: **end for**
12: **end for**
13: G $\leftarrow$ Accumulate(G, val, $global\_dof$)
14: **end if**

according to the list of plastic elements. As shown in line 9, K_e array is accessed by using suitable indices in coalesced manner, as discussed in Section 5.3.1.2. The rest of the steps remain the same as the single kernel EbE_{DOF} strategy.

Algorithm 23 GPU kernel for elements in the elastic zone using improved split kernel EbE_{DOF} matrix-free strategy.

Input: `P`, `K_e`, `connectivity`
Output: `G`

$threadId \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
$elemNo \leftarrow threadId/24$
$elaneId \leftarrow threadId\%24$ ▷ $elaneId \in (0, 1, ..., 24)$
if $elemNo < npelem$ **then**
  $global\_dof \leftarrow$ `connectivity`$[elemNo * 8 + elaneId/3]$
  `sP_e` $\leftarrow$ get_y_vector(`P`, $global\_dof$)
  `sK_e` $\leftarrow$ get_k_matrix(`K_e`, $global\_dof$)
  __synthreads()
  for $i = 1$ to e_{dof} **do** ▷ Loop over DOFs per element
    `val` += `sK_e`$[i * 24 + elaneId]$ * `sP_e`$[i]$
  end for
  `G` $\leftarrow$ Accumulate(`G`, `val`, $global\_dof$)
end if

Algorithm 24 GPU kernel for elements in the plastic zone using improved split kernel EbE_{DOF} matrix-free strategy.

Input: `P`, `K_e` `connectivity`, $npelem$
Output: `G`

$threadId \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
$elemNo \leftarrow threadId/24$
$elaneId \leftarrow threadId\%24$ ▷ $elaneId \in (0, 1, ..., 24)$
if $elemNo < maxPElem$ **then**
  $global\_dof \leftarrow$ `connectivity`$[elemNo * 8 + elaneId/3]$
  `sP_e` $\leftarrow$ get_y_vector(`P`, $global\_dof$)
  __synthreads()
  for $i = 1$ to e_{dof} **do** ▷ Loop over DOFs per element
    `val` += `K_e`$[elemNo * 24 + i * 24 * npelem + elaneId]$ * `sP_e`$[i]$
  end for
  `G` $\leftarrow$ Accumulate(`G`, `val`, $global\_dof$)
end if

5.3.2.5 Improved split kernel NbN strategy

In EbE strategies, dedicated kernels for elements in elastic and plastic zones perform computation with disjoint sets of elastic elements ($\mathcal{E}^{(e)}$) and plastic elements ($\mathcal{E}^{(p)}$). However, this is not directly applicable to node-based matrix-free strategies. Here, disjoint sets of elastic and plastic nodes are generated to perform computation of matrix-free SpMV using two separate kernels. If a node has all its connected elements in $\mathcal{E}^{(e)}$, it is

assigned to a set of elastic nodes $\mathcal{N}^{(e)}$. In all other cases, nodes are assigned to a set of plastic nodes $\mathcal{N}^{(p)}$.

In the improved split kernel *NbN* strategy, both the kernels are implemented on GPU using the *NbN* matrix-free strategy (Section 4.1.1). Algorithm 25 shows key steps in the GPU implementation for elastic nodes. As shown in line 2, using single thread per node strategy, threads are launched for the total number of elastic nodes $nENode$. For each thread, the global node number is obtained using `node_list` and used in subsequent memory access and computation. Since a common elemental tangent matrix is used for all elements in the elastic zone, computation is performed by reading values from the shared memory (see line 11). The GPU implementation of matrix-free computation for plastic nodes is presented in Algorithm 26. As shown in line 26, threads are launched considering the total number of plastic nodes. Each thread reads a global node number and uses it for further computation. It is noted that this strategy uses `K_index` array to access elemental tangent matrices like single kernel strategy. The rest of the steps remain the same as the single kernel *NbN* strategy.

Algorithm 25 GPU kernel for nodes in the elastic zone using improved split kernel *NbN* strategy

Input: `P`, `K_e`, `elem_list`, `node_pos`, `node_list`, `connectivity`
Output: `G`

1: $threadId \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
2: **if** $threadId < nENode$ **then**
3: $node\_no \leftarrow$ `node_list`
4: **for** $e = 1$ to $ctElem$ **do** ▷ Loop over elements connected to a node
5: $elem\_no \leftarrow$`elem_list`
6: $l\_index \leftarrow$`node_pos` ▷ $l\_index \in (0, 1, ..., 7)$
7: `global_dof` $\leftarrow$ `connectivity`$[elem\_no]$
8: `sK_e` $\leftarrow$ `K_e`
9: **for** $i = 1$ to n_{dof} **do** ▷ Loop over DOFs per node
10: **for** $j = 1$ to e_{dof} **do** ▷ Loop over DOFs per element
11: `val`$[i]$ += `sK_e`$[(3 * l\_index + i) * 24 + j]$ * `P`[`global_dof`$[j]$]
12: **end for**
13: **end for**
14: **end for**
15: `G` $\leftarrow$ Accumulate(`G`, `val`, $node\_no$)
16: **end if**

Algorithm 26 GPU kernel for nodes in the plastic zone using improved split kernel *NbN* strategy

Input: P, K_e, elem_list, node_pos, node_list, connectivity, K_index
Output: G

1: $threadId \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
2: **if** $threadId < nPNode$ **then**
3: $node\_no \leftarrow$ node_list
4: **for** $e = 1$ to $ctElem$ **do** ▷ Loop over elements connected to a node
5: $elem_no \leftarrow$elem_list
6: $l_index \leftarrow$node_pos ▷ $l\_index \in (0, 1, ..., 7)$
7: global_dof $\leftarrow$ connectivity[$elem\_no$]
8: **for** $i = 1$ to n_{dof} **do** ▷ Loop over DOFs per node
9: **for** $j = 1$ to e_{dof} **do** ▷ Loop over DOFs per element
10: val[i] += K_e[K_index[$elem\_no$] $*$ 576 + (3 $*$ l_index + i) $*$ 24 + j] * P[global_dof[j]]
11: **end for**
12: **end for**
13: **end for**
14: G $\leftarrow$ Accumulate(G, val, $node\_no$)
15: **end if**

5.3.2.6 Improved split kernel *DbD* strategy

In this strategy, the improved split kernel matrix-free SpMV for elastoplasticity is implemented on GPU using the *DbD* strategy. The computation is performed with the set of elastic and plastic nodes ($\mathcal{N}^{(e)}$,$\mathcal{N}^{(p)}$) as discussed in Section 5.3.2.5. Algorithms 27 and 28 show the GPU implementation of matrix-free SpMV for elastic and plastic nodes, respectively. It can be seen in both the algorithms that threads equal to DOFs associated with the number of nodes in elastic and plastic sets are launched. The rest of the computational steps remain similar as the improved split kernel *NbN* strategy.

5.3.3 Overview of the proposed matrix-free elastoplasticity solver

Algorithm 29 presents the pseudo code for Newton-Raphson iteration in elastoplasticity solver, highlighting the key computational steps and control flow. The Newton-Raphson method is implemented on the CPU with a series of calls to compute functions that executes on the GPU. Inside the Newton-Raphson loop, all computations are performed on GPU and there is no expensive data transfer with the CPU, except for tolerance value

Algorithm 27 GPU kernel for nodes in the elastic zone using improved split kernel *DbD* strategy.

Input: P, K_e, elem_list, node_pos, node_list, connectivity
Output: G

1: $threadId \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
2: **if** $threadId < 3 * nENode$ **then**
3: $node\_no \leftarrow$ node_list
4: **for** $e = 1$ to $ctElem$ **do** ▷ Loop over elements connected to a node
5: $elem_no \leftarrow$elem_list
6: $l_index \leftarrow$node_pos ▷ $l\_index \in (0, 1, ..., 7)$
7: global_dof $\leftarrow$ connectivity$[elem_no]$
8: sK_e $\leftarrow$ K_e
9: **for** $j = 1$ to e_{dof} **do** ▷ Loop over DOFs per element
10: val += sK_e$[(3 * l_index + (threadId\%3)) * 24 + j]$ * P[global_dof$[j]$]
11: **end for**
12: **end for**
13: G $\leftarrow$ Accumulate(G, val, $node\_no$)
14: **end if**

Algorithm 28 GPU kernel for nodes in the plastic zone using improved split kernel *DbD* strategy.

Input: P, K_e, elem_list, node_pos, node_list, K_index, connectivity
Output: G

1: $threadId \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
2: **if** $threadId < 3 * nPNode$ **then**
3: $node\_no \leftarrow$ node_list
4: **for** $e = 1$ to $ctElem$ **do** ▷ Loop over elements connected to a node
5: $elem_no \leftarrow$elem_list
6: $l_index \leftarrow$node_pos ▷ $l\_index \in (0, 1, ..., 7)$
7: global_dof $\leftarrow$ connectivity$[elem_no]$
8: **for** $j = 1$ to e_{dof} **do** ▷ Loop over DOFs per element
9: val += K_e$[$K_index$[elem_no] * 576 + (3 * l_index + (threadId\%3)) * 24 + j]$ * P[global_dof$[j]$]
10: **end for**
11: **end for**
12: G $\leftarrow$ Accumulate(G, val, $node\_no$)
13: **end if**

in line 20.

The proposed matrix-free SpMV strategies work with a segregated list of elastic and plastic elements or nodes and associated data. In Algorithm 29, the identification, separation and reordering of data associated with elastic and plastic states are done at

two places in the Newton iteration loop, referred to as pre-processing steps 1 and 2 (see lines 2 and 8). The GPU implementation of the single kernel and improved split kernel strategies requires an array of size e_t that stores global element number as values (see Section 5.1), referred as `e_array`. In order to differentiate between elastic and plastic elements, the global element number of elastic elements is incremented by e_t. If `e_array` is sorted in ascending order, all plastic elements are moved to the beginning of the array, followed by elastic elements. This segregation is easily achieved in Algorithm 29 by running thrust::sort() on GPU, generating $\mathcal{E}^{(e)}$ and $\mathcal{E}^{(p)}$. Next, the total number of plastic elements is obtained by counting the values less the e_t, implemented by thrust::count() in line 4. The determination of the number of plastic elements is essential to allocate memory for the storage of $(e_p + 1)$ elemental tangent matrices. In line 7, elemental tangent matrices are computed by considering a list of plastic elements from `e_array`. If the single kernel strategy is used, `K_index` must be generated as a part of pre-processing 2 (line 10) to use in matrix-free SpMV. The array `K_index` is generated on GPU by using a custom kernel where `e_array` is used to obtain indices of plastic tangent matrices. If the improved split kernel strategy is used, the input data is reordered at line 12 according to the list of elastic and plastic elements given by `e_array`. After pre-processing step 2, the solution of linear system of equations is obtained in line 15 using CG solver based on the proposed matrix-free SpMV strategies. This is followed by the computation of stress using the radial-return method. It is noted that `e_array` is also updated in the radial-return method during the evaluation of the yield function. Once the stress is evaluated, it is used in line 18 to compute internal force vectors and their assembly. The computation of stress and internal force vectors are done on GPU using strategies presented in Chapter 3. The computation of internal forces leads to the determination of an unbalanced force vector and estimation of tolerance in line 20. If tolerance is found to be less than a given value, the Newton iteration ends, and the final value is recorded; otherwise, computations continue to the next iteration.

In the *NbN* strategy, the steps mentioned in pre-processing 1 and 2 of Algorithm 29 remain the same for the single kernel strategy. However, the procedure in pre-processing step 2 change for the improved split kernel strategy to compute sets of elastic and plastic nodes, $\mathcal{N}^{(e)}$ and $\mathcal{N}^{(p)}$. $\mathcal{N}^{(e)}$ and $\mathcal{N}^{(p)}$ are determined in the same way as steps described in pre-processing step 1 with a difference that an array of global node number is taken

in place of global element number.

Algorithm 29 GPU-based implementation of Newton-Raphson (NR) iterations in the proposed framework.

Input: `coordinates`, `connectivity`, Problem parameters like material properties and boundary conditions

Output: `U, stress`

for $Itr < Max\_itr$ **do** ▷ Newton-Raphson iterations
  //- - - - - - - - - - - - - Pre-processing 1
  Apply thrust::sort with `e_array` to find $\mathcal{E}^{(e)}$ and $\mathcal{E}^{(p)}$
  Apply thrust::count to find number of plastic elements e_p
  Allocate memory to store elemental tangent matrix
  //- - - - - - - - - - - - -
  Compute elemental tangent matrices
  //- - - - - - - - - - - - - Pre-processing 2
  if $spmv\_strat ==$ '*singlekernel*' **then**
    Generate `K_index` array
  else if $spmv\_strat ==$ '*splitkernel*' **then**
    Reorder input data according to $\mathcal{E}^{(e)}$ and $\mathcal{E}^{(p)}$
  end if
  //- - - - - - - - - - - - -
  Solve on GPU: Matrix-free CG solver
  Update displacement
  Compute stress: Radial-return method
  Compute internal force vectors
  Update unbalance force vectors ▷ Compute on GPU
  $tol \leftarrow$ check_convergence() ▷ Compute on GPU
  if $tol < \xi_{NR}$ **then**
    Update variables ▷ Compute on GPU
    Break
  end if
end for

5.3.4 Results and discussion

The numerical experiments are conducted with an intent to study the performance of the proposed strategies in a comprehensive manner. The key factors that affect the performance of a matrix-free SpMV kernel are thread allocation, memory access pattern, usage of on-chip memory and presence of redundant computation. In this sub-section, we present assessment of these performance indicators by solving three large-scale benchmark examples from elastoplasticity. Towards the end of this sub-section, a comparison

with assembly-based elastoplasticity solver is also presented to highlight the advantages and to discuss limitations if any. The meshes for the numerical examples are generated in ABAQUS (Systèmes 2017) software package by considering 8-noded hexahedral elements. The hardware used for numerical experiments has the same specifications as given in Section 3.2.

5.3.4.1 Computational examples

Cantilever beam with a cut out feature

As a first example, a 3D cantilever beam with a rectangular cut out feature (Meguid 2021) is considered. Often, cut out features or openings are introduced into machine components due to specific design requirements. However, geometrical features of cut outs might lead to creation of high stress concentration zones in the body, increasing the local stress beyond the yield value. Figure 5.43 shows the geometry of the cantilever beam in 2D where the left end with larger cross section is fixed and the right end with smaller cross section is loaded on the edge. The corresponding 3D model is obtained by extrusion in z-direction with 32 mm thickness. The elastoplastic analysis is performed by considering the following material properties: Young's modulus 200 GPa, Poisson's ratio 0.3, initial yield stress 220 MPa and hardening coefficient 20 GPa. In order to obtain different amount of plasticity in the body, following loads are applied: 27000 N, 37000 N, 44000 N, 52000 N, 62000 N and 68000 N.

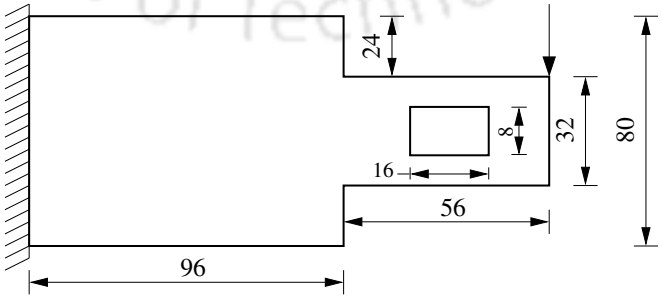


Figure 5.43: A cantilever beam with a cut out feature. All dimensions are taken in millimeters (mm).

L-bracket

This example is taken from Section 5.2.2.1.

Plate with square holes

This example is taken from Section 5.2.2.3.

The performance results for all the three computational examples are obtained for various levels of plasticity and mesh sizes. Table 5.4 presents the meshes for the numerical experiments with varying level of refinement. The number of DOFs for a mesh can be obtained by multiplying the number of nodes with three.

Table 5.4: Finite element mesh for the numerical experiment.

Cantilever			L-bracket			Plate with square holes		
Mesh	Elements	Nodes	Mesh	Elements	Nodes	Mesh	Elements	Nodes
C1	126 144	136 200	L1	103 488	115 596	H1	50 400	76 791
C2	299 008	316 800	L2	212 716	232 200	H2	170 100	229 188
C3	584 000	611 720	L3	418 176	448 115	H3	787 500	950 982
C4	1 009 152	1 048 992	L4	813 186	860 384	H4	1 360 800	1 595 979
C5	1 602 496	1 656 648	L5	1 622 964	1 697 080	H5	2 160 900	2 480 776
C6	2 392 064	2 462 720	L6	2 268 084	2 361 280			

5.3.4.2 Results for linear elasticity

The proposed improved split kernel strategy isolates the computation for elements in the elastic zone from the bulk and uses a separate kernel for the computation of matrix-free SpMV. This enables opportunities for several optimizations which are otherwise not possible with the single kernel strategy. The computation of matrix-free SpMV for elastic elements in the improved split kernel strategy has the same CUDA implementation as a matrix-free SpMV kernel in linear elasticity problems, primarily due to the same input data requirement. Therefore, a series of experiments is performed by considering linear elastic material model to assess the effect of implementation related optimizations on the performance of the proposed matrix-free strategies. In the subsequent discussion, various options for efficient usage of on-chip memory are explored, and results are discussed.

The amount of shared memory for the device (Tesla K40 GPU) used for the numerical experiment is 48 KB. Considering full occupancy, a streaming multiprocessor (SM) on Tesla K40 can launch 2048 threads, simultaneously. This implies that a thread can accommodate a maximum of $\frac{48 \times 1024}{2048 \times 8} = 3$ double precision values for full occupancy. For a thread block of 256 threads, a total of 768 values can be cached in the shared memory. The computation of matrix-free SpMV for elastic elements requires a single tangent matrix of size 24×24 and a vector of size 24×1 per element to perform multiplication. In EbE_{Node} strategy, 576 values for a tangent matrix and $24 \times \frac{256}{8} = 768$ values for multiplying vector are required for a thread block of size 256. This requirement is beyond the limit of 768 values. In such cases, either tangent matrix or multiplying vectors alone can be stored in the shared memory. In another option, both variables can be stored but at the cost of reduced occupancy. The shared memory requirement in the EbE_{DOF} strategy also remains the same as the EbE_{Node} strategy, except that the available space increases due to the allocation of 24 threads per element. Still, the EbE_{DOF} strategy requires $576 + 240 = 816$ values, which is larger than the limiting value of 768 for a thread block of 256. However, shared memory pressure in the EbE_{DOF} strategy is low compared to EbE_{Node}. In order to find the best performing strategy, numerical experiments are performed with several variations of EbE_{Node} and EbE_{DOF} strategies as presented in Table 5.5. The variation A does not use the shared memory and is implemented to provide the base timings for the comparison. The variation B stores the elemental tangent matrix in the shared memory and reads multiplying vector from the global memory. The variation C stores both the elemental tangent matrix and multiplying vector in the shared memory and has reduced occupancy.

Table 5.5: Variations in the implementation of EbE_{Node} and EbE_{DOF} strategies.

Strategy	Variations	Tangent matrix	Vector
EbE_{Node} / EbE_{DOF}	A	global	global
	B	shared	global
	C	shared	shared

In NbN and DbD strategies, one thread is assigned to perform computation for one node or one DOF. Since a thread can keep only 3 double precision values, the NbN and DbD strategies cannot make sufficient use of shared memory. For a thread block of size 256, only the common elemental tangent matrix having 576 values can be stored. The

limited shared memory per thread in *NbN* and *DbD* strategies does not allow storage of multiplying vectors associated with all neighboring elements. However, the multiplying vectors can be accessed through read-only cache memory. Table 5.6 presents the variations in the implementation of *NbN* and *DbD* strategies for the numerical experiment. Here, variation *A* refers to basic implementation with no caching. In variation *B*, the elemental tangent matrix is stored in shared memory and the vector for multiplication is read from global memory. The variation *C* stores the elemental tangent matrix in the shared memory and access multiplying vector through read-only cache.

Table 5.6: Variations in the implementation of *NbN* and *DbD* strategies.

Strategy	Variations	Tangent matrix	Vector
NbN / *DbD*	*A*	global	global
	B	shared	global
	C	shared	read-only cache

Table 5.7 presents the kernel timings of EbE_{Node} and EbE_{DOF} strategies for variations in GPU implementation as given by Table 5.5. In EbE_{Node} strategy, largest kernel timings are obtained by variation *A* that does not make any use of the shared memory. The advantage of using the shared memory for the elemental tangent matrix can be clearly seen in improved kernel timings of variation *B*. However, the variation *C* that uses the shared memory for both the elemental tangent matrix and multiplying vector shows little improvement over timings of variation *B*. This is due to the fact that variation *C* has reduced occupancy due to excessive use of the shared memory, which is detrimental to the performance. In the EbE_{DOF} strategy, variation *A* obtains the largest kernel timings, whereas variation *C* provides the smallest kernel timings. Here, variation *C* shows sufficient improvement with respect to variation B. Although variation *C* has reduced occupancy like EbE_{Node} strategy, shared memory pressure is relatively less. In both EbE_{Node} and EbE_{DOF} strategies, variation *C* is found to be the fastest implementation, highlighting the effect of optimized shared memory usage on the performance of matrix-free SpMV.

Table 5.8 presents the kernel timings of *NbN* and *DbD* strategies for different variations in implementation. In both the *NbN* and *DbD* strategies, variation *A* is found to be the slowest, and variation *C* is found to be the fastest in terms of kernel timings.

Table 5.7: Kernel timings (milliseconds) for *EbE* strategies in linear elasticity.

Mesh	EbE_{Node}			EbE_{DOF}		
Cantilever						
126 144	1.384	1.083	1.061	2.181	1.652	1.185
299 008	3.035	2.401	2.367	5.007	3.789	2.690
584 000	5.772	4.576	4.522	9.632	7.243	5.159
1 009 152	9.839	7.811	7.720	16.590	12.475	8.845
1 602 496	15.518	12.334	12.180	26.240	19.668	13.970
2 392 064	23.086	18.320	18.106	39.152	29.450	20.823
L-bracket						
103 488	1.207	0.958	0.914	1.816	1.366	0.964
212 716	2.333	1.870	1.806	3.617	2.705	1.910
418 176	4.375	3.491	3.406	6.997	5.214	3.671
813 186	8.291	6.609	6.492	13.451	10.062	7.089
1 622 964	16.163	12.851	12.680	26.577	19.715	13.976
2 268 084	22.279	17.661	17.488	37.062	27.465	19.451
Plate with square holes						
50 400	0.677	0.530	0.510	0.945	0.723	0.529
1 70 100	2.028	1.656	1.462	3.103	2.419	1.574
7 87 500	8.724	7.184	6.403	13.570	10.456	6.889
1 360 800	14.697	12.034	10.961	23.247	17.880	11.951
2 160 900	23.223	19.023	17.531	36.777	28.164	19.066

The usage of shared memory for elemental tangent matrix has a positive effect on the performance as evident by kernel timings of variation B in both NbN and DbD strategies. In both strategies, the use of read-only cache for reading multiplying vector has a huge effect on the performance, achieving approximately twice the speedup with respect to variation B.

The results presented in Tables 5.7 and 5.8 indicate significant improvement in the performance of matrix-free SpMV kernel due to optimization of memory access using shared memory of the GPU. The outcome of this study is incorporated into the proposed improved split kernel strategy for elements in the elastic zone.

Table 5.8: Kernel timings (milliseconds) for *NbN* and *DbD* strategies in linear elasticity.

Mesh	*NbN*			*DbD*		
Cantilever						
126 144	2.394	2.205	0.989	3.220	2.415	1.254
299 008	5.573	5.086	2.370	7.437	5.5889	2.968
584 000	10.662	9.755	4.458	14.331	10.748	5.699
1 009 152	18.271	16.652	7.475	24.535	18.379	9.789
1 602 496	28.812	26.270	11.563	38.758	28.966	15.349
2 392 064	43.006	39.197	16.999	57.546	43.027	22.803
L-bracket						
103 488	2.073	1.886	0.779	1.769	1.029	1.037
212 716	4.098	3.748	1.568	3.568	2.122	2.134
418 176	7.884	7.188	3.086	7.030	4.218	4.248
813 186	15.187	13.815	5.922	13.681	8.219	8.281
1 622 964	30.251	27.289	12.093	27.145	16.241	16.355
2 268 084	42.267	38.098	17.111	37.991	22.723	22.850
Plate with square holes						
50 400	1.007	0.938	0.403	1.293	0.969	0.503
1 70 100	3.396	3.105	1.420	4.570	3.493	1.614
7 87 500	15.992	14.672	7.755	21.230	16.319	7.446
1 360 800	27.273	25.118	13.002	36.614	28.112	12.937
2 160 900	43.115	39.715	20.730	57.724	44.185	20.806

5.3.4.3 Results for elastoplasticity

Single kernel strategy

The kernel timings of matrix-free SpMV for elastoplastic analysis of the cantilever beam with a cut out feature is presented in Fig. 5.44. The matrix-free SpMV is implemented using the single kernel strategy, and results are presented for all mesh sizes as a function of percentage plasticity. It can be seen that the least timings are achieved by EbE_{Node} followed by EbE_{DOF}, *DbD* and *NbN* strategies. For all the strategies except EbE_{DOF}, the kernel timings are found to be increasing linearly with the percentage plasticity. The size of the plastic zone increases with increasing percentage plasticity, implying a rise in the number of elemental tangent matrices for the computation of matrix-free SpMV. This, in turn, increases the amount of input data, which is the primary reason for increasing kernel timings. The timings of the EbE_{DOF} strategy are found to have a mild dependency on percentage plasticity. At low plasticity, timings of *NbN* strategy (Fig.

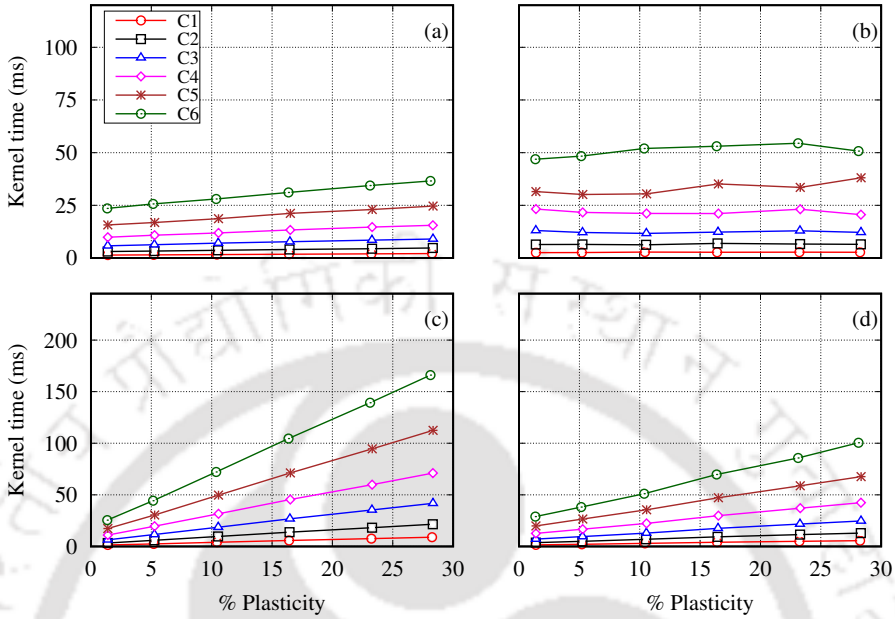


Figure 5.44: Kernel timings for cantilever beam with a cut out by (a) Single kernel EbE_{Node} strategy, (b) Single kernel EbE_{DOF} strategy, (c) Single kernel *NbN* strategy and (d) Single kernel *DbD* strategy.

5.44c) is found comparable with EbE_{Node}. However, the performance of *NbN* degrades sharply with increasing percentage of plasticity. Similar observations can be made in the case of *DbD* strategy (Fig. 5.44d) where timings depend strongly on the percentage plasticity. Compared to *EbE* strategies, node-based strategies do not access elemental tangent matrices in a coalesced manner. At low plasticity levels, the uncoalesced access has less serious effect on the performance. However, with increasing plasticity percentage, uncoalesced access has a more serious impact on the performance as the volume of input data increases rapidly. It is noted that *EbE* strategies are implemented with the mesh coloring method, which increases the number of kernel launches. The kernel timings of the single kernel strategy for the L-bracket example are shown in Fig. 5.45. It can be seen that the best timing is achieved by EbE_{Node} strategy and the worst timing is obtained by *NbN* strategy (Fig. 5.45c). The EbE_{DOF} strategy obtains higher timings than EbE_{Node} at low plasticity but achieves comparable performance at high percentage plasticity. Like the cantilever beam example, the performance of the *DbD* strategy is found better than *NbN*. The kernel timings for the plate with square holes example using single kernel strategy are shown in Fig. 5.46. Like previous examples, the best

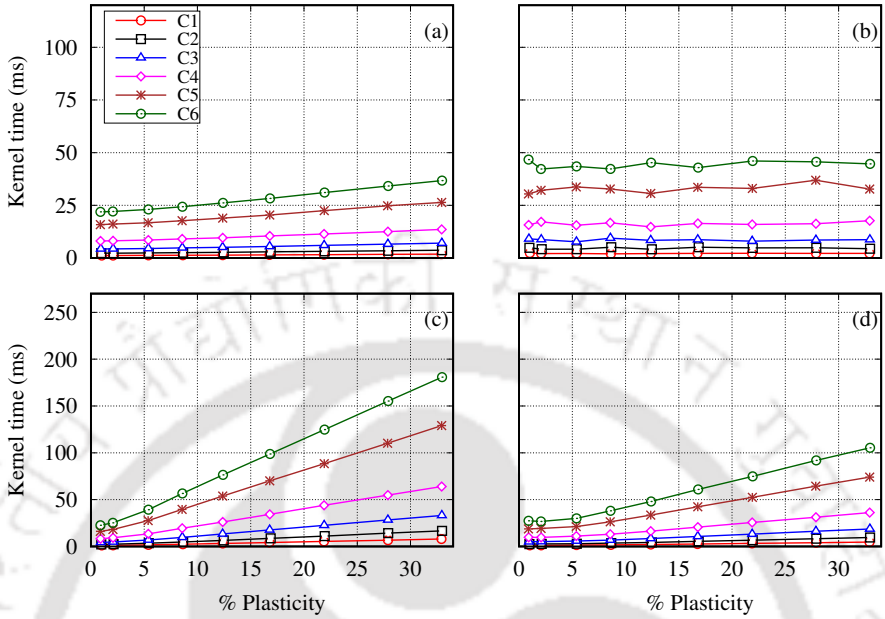


Figure 5.45: Kernel timings for L-bracket by (a) Single kernel EbE_{Node} strategy, (b) Single kernel EbE_{DOF} strategy, (c) Single kernel NbN strategy and (d) Single kernel DbD strategy.

kernel timing is obtained with the EbE_{Node} strategy followed by EbE_{DOF}, DbD and NbN strategies. The distribution of von Mises stress in all three examples using the coarsest mesh (Table 5.4) for elastoplastic analysis is shown in Fig. C.1 (see Appendix).

In the single kernel strategy, both the elastic and plastic tangent matrices are stored in the same array so that it can be accessed using suitable indices (see Section 5.2). Each matrix-free strategy is provided with a pre-computed array (referred as `K_index` in Section 5.3.2.1) that gives indices to relevant elemental tangent matrices, preventing any decision-making due to the presence of elastic and plastic states. However, this strategy also prevents any use of shared memory to store elemental tangent matrices. Since a common elastic tangent matrix is used for all elements in the mesh, it can be cached into the shared memory. An approach to matrix-free SpMV can be taken where access to the elastic elemental tangent matrix is made through the shared memory, and global memory transactions are used only for plastic tangent matrices. The use of shared memory for elastic tangent matrix has a major disadvantage that the GPU implementation requires the use of conditional statements. An experiment is performed with shared

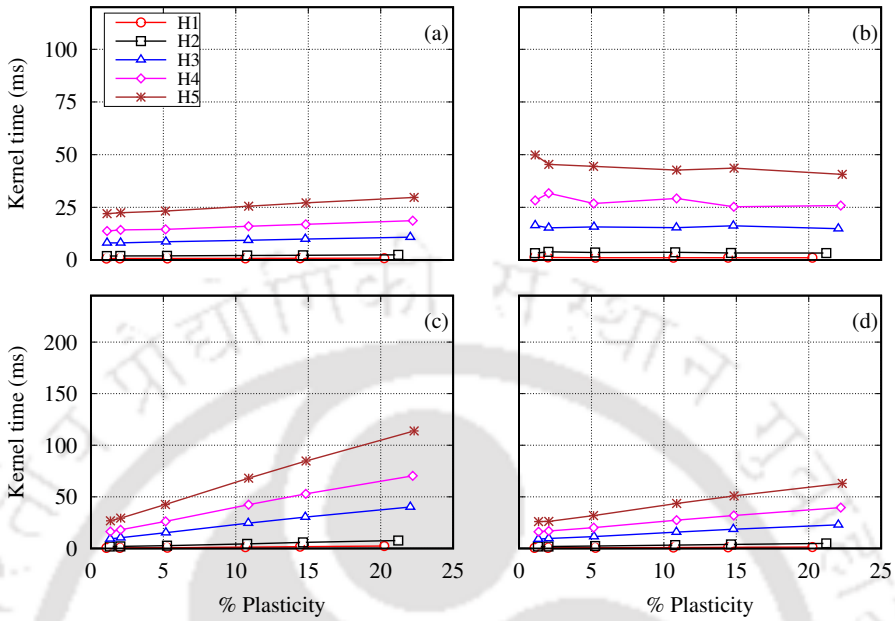


Figure 5.46: Kernel timings for plate with square holes by (a) Single kernel EbE_{Node} strategy, (b) Single kernel EbE_{DOF} strategy, (c) Single kernel NbN strategy and (d) Single kernel DbD strategy.

memory-based approach, and results are compared with implementations that do not use shared memory in Fig. 5.47. The results are presented for one mesh each from all three examples, namely C3, L3, and H3. In Fig. 5.47a, results for EbE_{Node} strategy show poor performance by shared memory-based implementations in each example, shown by dotted lines. On the other hand, the EbE_{DOF} strategy observes significant improvement in kernel timings due to the storage of elemental tangent matrix in the shared memory. The NbN and DbD strategies show negative effect of the shared memory on kernel timings at high plasticity percentage, similar as shown by Fig. 5.9 in Section 5.2.2. In the following discussion, the single kernel EbE_{DOF} strategy uses shared memory for storage of elastic elemental tangent matrix.

The GPU implementation of the single kernel strategy for elastoplasticity requires preparation of data according to elastic and plastic elements or nodes. This is denoted as pre-processing steps 1 and 2 in Algorithm 29 and implemented on GPU as discussed in Section 5.3.3. Figure 5.48 shows the combined time spent in both the pre-processing steps in Algorithm 29 for all three examples. Each sub-figure contains timings for EbE

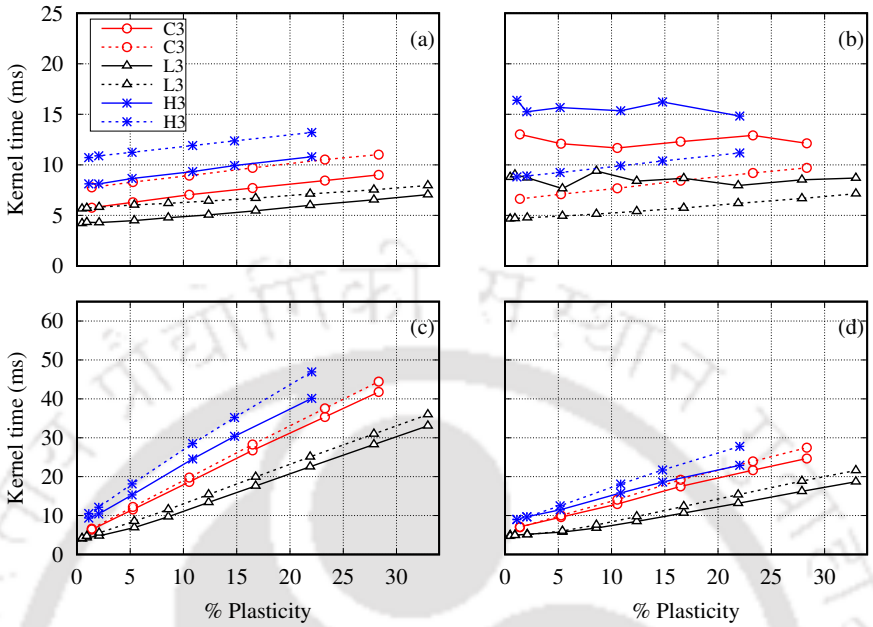
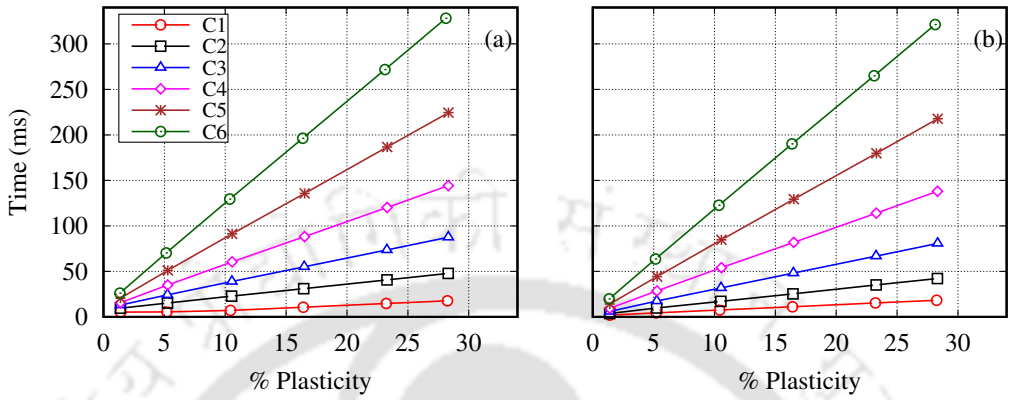


Figure 5.47: Comparison of kernel timings by (a) Single kernel EbE_{Node} strategy, (b) Single kernel EbE_{DOF} strategy, (c) Single kernel NbN strategy and (d) Single kernel DbD strategy. The dotted lines show kernel timings of implementations that use shared memory for common elastic elemental tangent matrix.

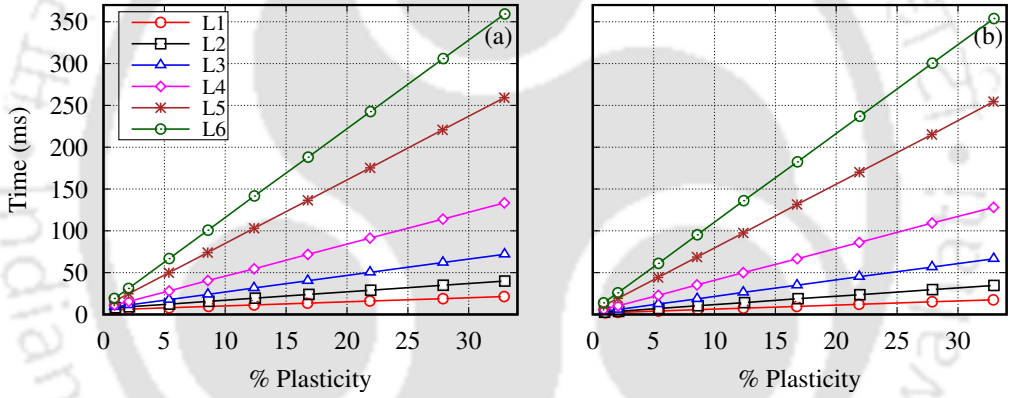
strategies in part 'a' and timings for node-based strategies in part 'b'. The EbE_{Node} and EbE_{DOF} strategies have the same GPU implementation for pre-processing, and therefore common timings are presented for them; similarly NbN and DbD have the same timings. It can be observed that the pre-processing time increases linearly with percentage plasticity in all the examples. Also, the pre-processing timings for EbE and NbN strategies are found to be similar. Since pre-processing is done once in a Newton-Raphson iteration, timings in Fig. 5.48 show a very small overhead, highlighting the effectiveness of the proposed strategy.

Improved split kernel strategy

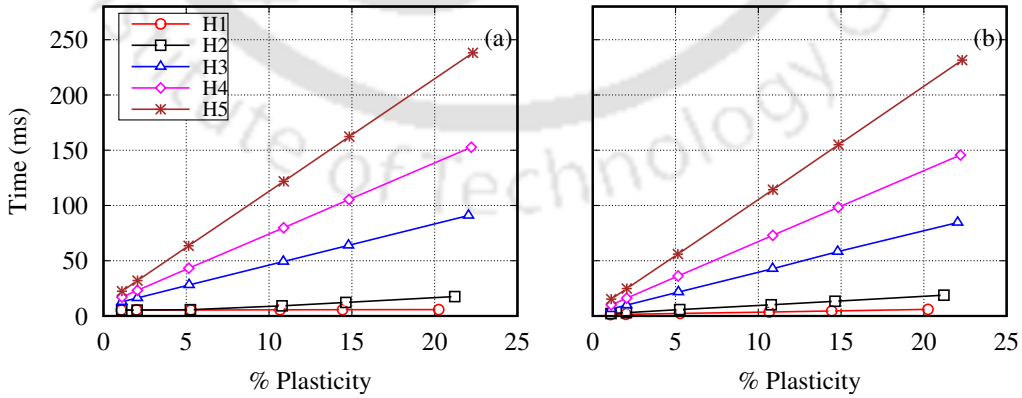
Figure 5.49 shows the kernel timings of the improved split kernel strategy for cantilever beam example. The reported kernel timings show combined execution time of both elastic and plastic kernels. The kernel timings are found increasing linearly with percentage plasticity for all the strategies. At low plasticity levels, kernel timings are dominated by



(a) Cantilever beam with a cut out.



(b) L-bracket.



(c) Plate with square holes.

Figure 5.48: Pre-processing timings for single kernel strategy. In each sub-figure, pre-processing timings are given by (a) *EbE* strategy and (b) *NbN*/*DbD* strategy.

the computation for elements in the elastic zone. As the amount of plasticity increases, more elements or nodes move into the plastic zone, and consequently, the kernel assigned to plastic elements or nodes dominates the matrix-free SpMV timings. Similar to the single kernel strategy, the best timing is achieved with EbE_{Node} followed by EbE_{DOF}, *DbD* and *NbN* strategies. It can be observed that all the strategies achieve comparable timings at low percentage of plasticity. However, as the amount of plasticity increases, the kernel timings increase at different rates. The slope of kernel time is found larger in *NbN* and *DbD* strategies due to uncoalesced access to elemental tangent matrices. The kernel timings for L-bracket and plate with square holes examples are presented in Figs. 5.50 and 5.51, respectively. In both examples, the EbE_{Node} strategy is found to have the minimum kernel timings. The kernel timings curves in all three examples are found to be similar for all the strategies.

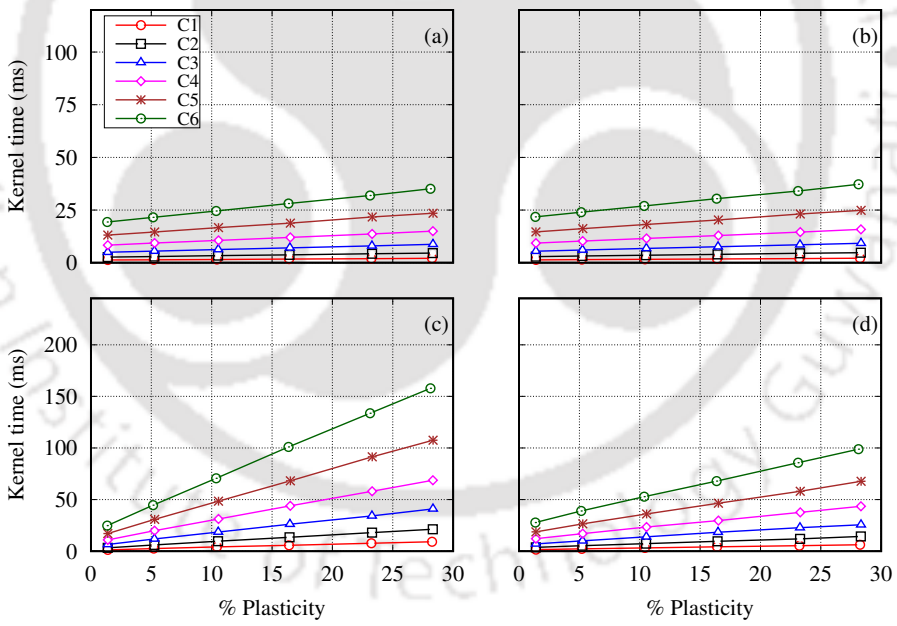


Figure 5.49: Kernel timings for cantilever beam with a cut out by (a) Improved split kernel EbE_{Node} strategy, (b) Improved split kernel EbE_{DOF} strategy, (c) Improved split kernel *NbN* strategy and (d) Improved split kernel *DbD* strategy.

The pre-processing timings for the improved split kernel strategy (as discussed in Algorithm 8) are shown in Fig. 5.52. Each sub-figure shows the combined execution timings in both pre-processing steps 1 and 2, and presents timings in part 'a' for *EbE* strategies and part 'b' for *NbN* or *DbD* strategy. It can be observed that the timings

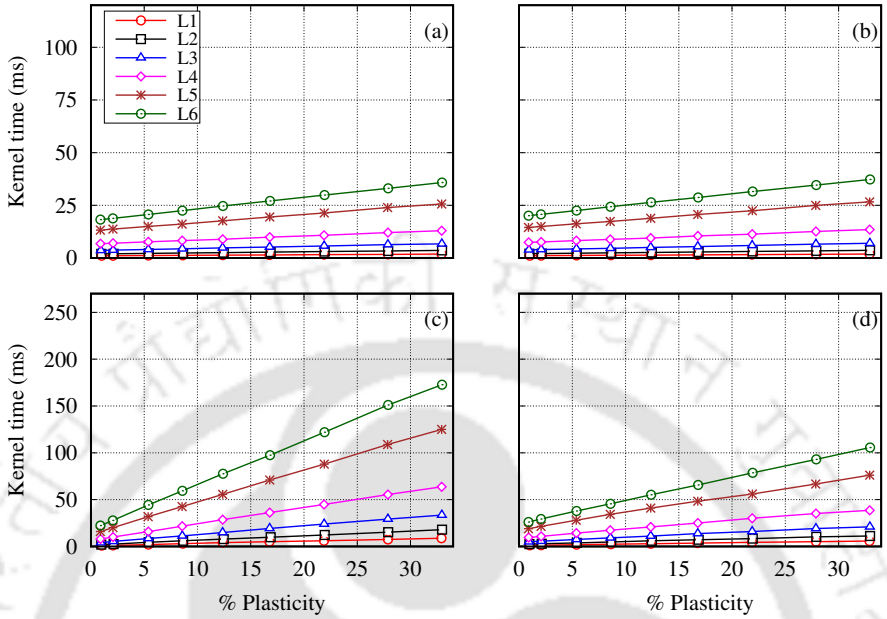


Figure 5.50: Kernel timings for L-bracket by (a) Improved split kernel EbE_{Node} strategy, (b) Improved split kernel EbE_{DOF} strategy, (c) Improved split kernel NbN strategy and (d) Improved split kernel DbD strategy.

increase linearly with percentage plasticity for all three examples. The timings of pre-processing are found similar for both EbE and NbN or DbD strategies. Comparing with Fig. 5.48, the time spent in pre-processing steps for single kernel strategy and improved split kernel strategy is found to be the same.

5.3.4.4 Performance evaluation

Comparison of kernel timings

A relative comparison of the performance by the single kernel strategy and improved split kernel strategy is presented by plotting speedups with percentage plasticity for all three examples. The speedup is computed by dividing the kernel timings of single kernel EbE_{Node}, EbE_{DOF}, NbN and DbD strategies by kernel timings of corresponding improved split kernel strategies. Figure 5.53 shows the speedup for the cantilever beam example. In the EbE_{Node} strategy, speedups up to 1.2× can be seen at low percentage plasticity that decrease up to 1.02× at high percentage plasticity. The speedups curve

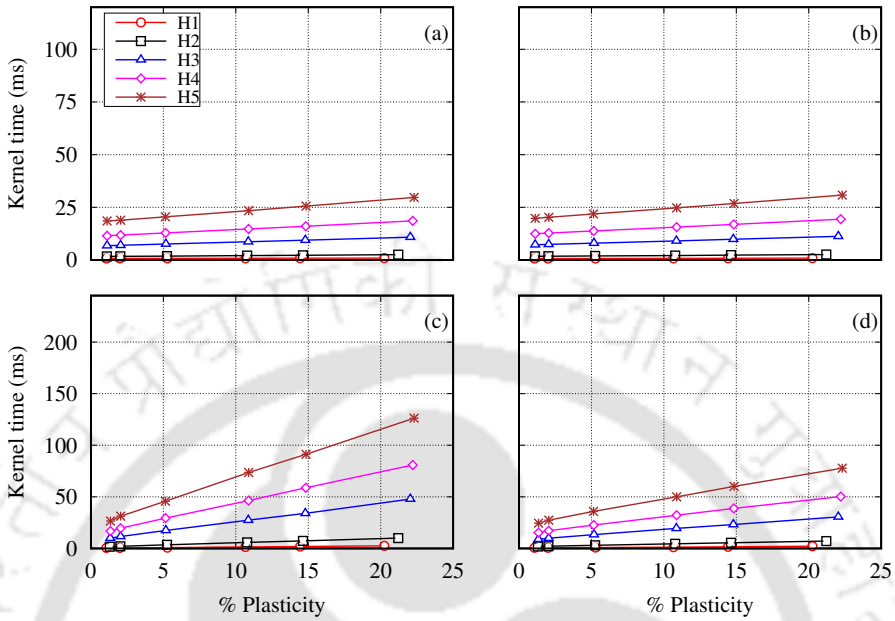
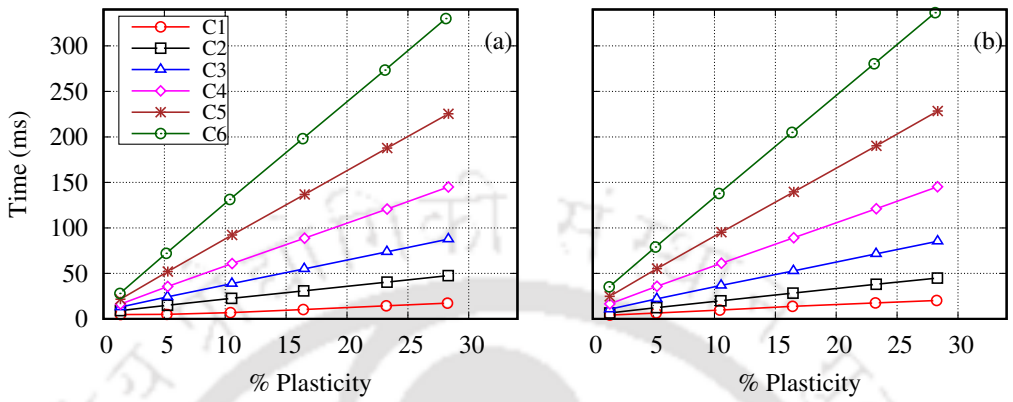


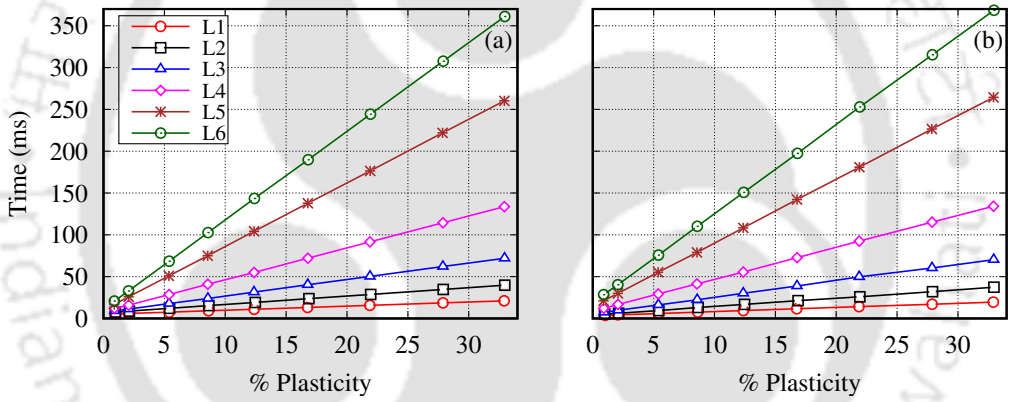
Figure 5.51: Kernel timings for plate with square holes by (a) Improved split kernel EbE_{Node} strategy, (b) Improved split kernel EbE_{DOF} strategy, (c) Improved split kernel NbN strategy and (d) Improved split kernel DbD strategy.

show similar profile for the EbE_{DOF} strategy where speedups of up to 1.25× are observed at low plasticity level that decrease to 1.05× with increasing percentage plasticity. As shown in Figs. 5.53c and 5.53d, NbN and DbD strategies show poor performance with improved split kernel strategy as obtained speedups remain below one for all percentage of plasticity, except at the lowest value. Overall, the improved split kernel variants of only EbE strategies show improved kernel timings at low percentage of plasticity. Since most of the elements remain in the elastic zone for low plasticity levels, the improved split kernel strategy obtains reduced timings due to optimized kernel for the elastic part. At high plasticity, more number of elements or nodes moves to the plastic zone and dictates the kernel timings. The speedup results by improved split kernel strategies for L-bracket and plate with square holes examples are shown in Figs. 5.54 and 5.55, respectively. The speedup curve for all the strategies are found to be similar as the cantilever beam example.

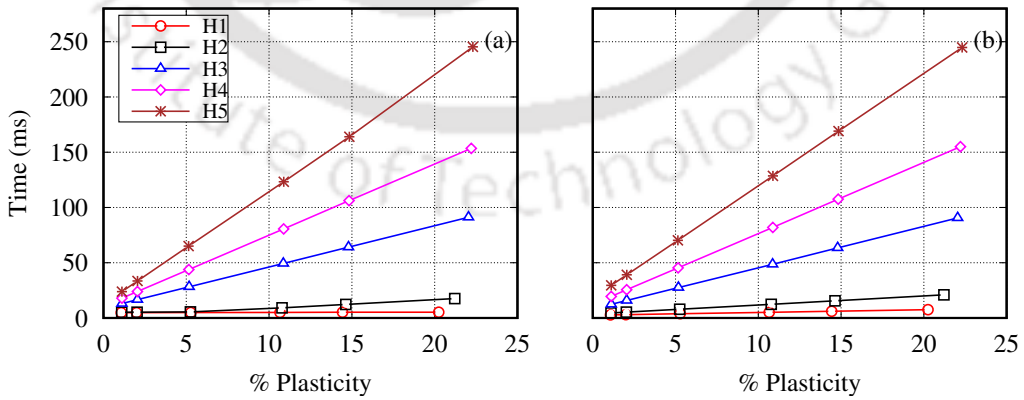
Figure 5.56 shows speedups by the best performing improved split kernel EbE_{Node} strategy over single kernel NbN strategy, for all three examples. The performance of



(a) Cantilever beam with a cut out.



(b) L-bracket.



(c) Plate with square holes.

Figure 5.52: Pre-processing timings for improved split kernel strategy. In each sub-figure, pre-processing timings are given by (a) *EbE* strategy and (b) *NbN*/*DbD* strategy.

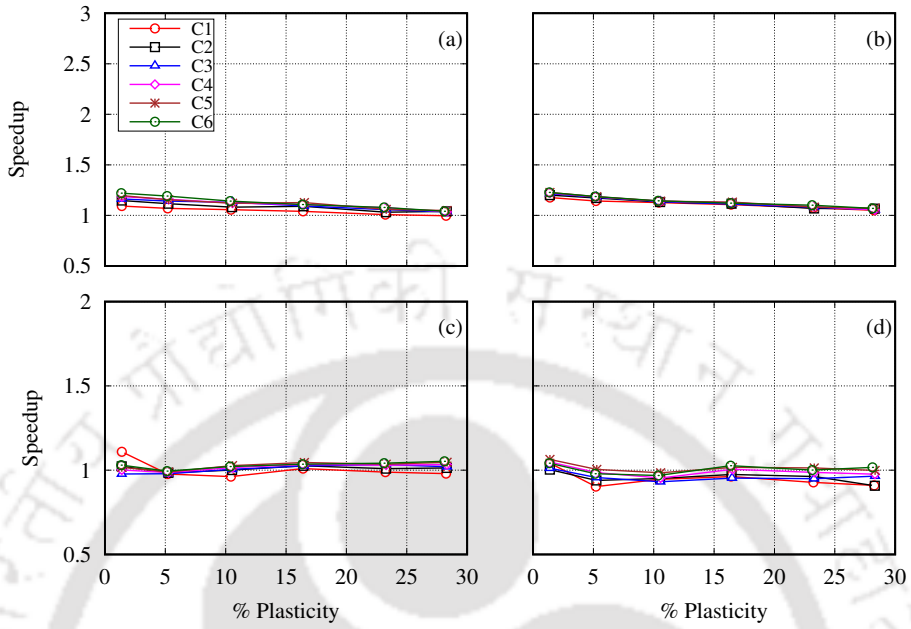


Figure 5.53: Speedup in kernel timings by improved split kernel strategies over single kernel strategies for cantilever beam example. Speedups are presented for (a) EbE_{Node}, (b) EbE_{DOF}, (c) NbN and (d) DbD strategies.

improved split kernel NbN strategy is found inferior than single kernel NbN strategy and therefore not considered for the comparison. For all three examples, a minimum speedup of approximately 1.1× (with one exception) is found at low plasticity levels that increases with increasing percentage plasticity and reaches up to 5× in case of the L-bracket. Similar speedup curves are found in Fig. 5.57 where speedup by improved split kernel EbE_{Node} strategy over the single kernel DbD strategy is found in the range 1.05–3× for all three examples. The increasing speedup profile in Figs. 5.56 and 5.57 is due to overhead associated with uncoalesced access to elemental tangent matrices in NbN and DbD strategies. As discussed in the previous section, kernel timings do increase with percentage plasticity for all the strategies, but more steeply for NbN and DbD strategies. The speedup by improved split kernel EbE_{Node} strategy over improved split kernel EbE_{DOF} strategy is shown in Fig. 5.58. At low plasticity levels, speedups up to 1.12× are obtained that decrease with increasing percentage plasticity.

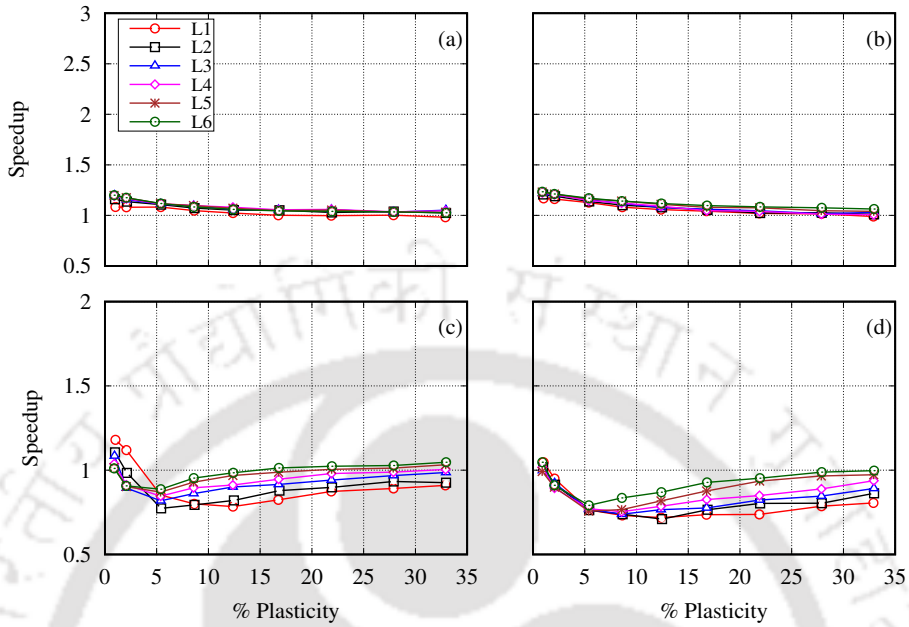


Figure 5.54: Speedup in kernel timings by improved split kernel strategies over single kernel strategies for L-bracket. Speedups are presented for (a) EbE_{Node}, (b) EbE_{DOF}, (c) *NbN* and (d) *DbD* strategies.

Comparison of wall-clock timings

The wall-clock timings also include time spent in other steps of elastoplastic analysis apart from the linear solver. The computationally expensive steps like computation of elemental tangent matrices, computation of stress and computation of internal forces are implemented as proposed in Chapter 3. In addition, CG solver may take different number of iterations depending on the matrix-free SpMV strategies. Figures 5.59, 5.60 and 5.61 show speedups in wall-clock timings by improved split kernel strategies over single kernel strategies for cantilever beam, L-bracket and plate with square holes examples, respectively. The improved split kernel EbE_{Node} strategy achieves speedup up to 1.5× for the cantilever beam (Fig. 5.59a), 1.3× for the L-bracket (Fig. 5.60a) and 1.2× for the plate with holes (Fig. 5.61a) examples. Similar speedups are obtained for the improved split kernel EbE_{DOF} strategy. In both cases, speedups are found decreasing with percentage plasticity in all three examples. The speedups by improved split kernel *NbN* and *DbD* strategies are found less than one in all the examples, except at the lowest plasticity level. This is expected as the kernel timings also achieve inferior performance

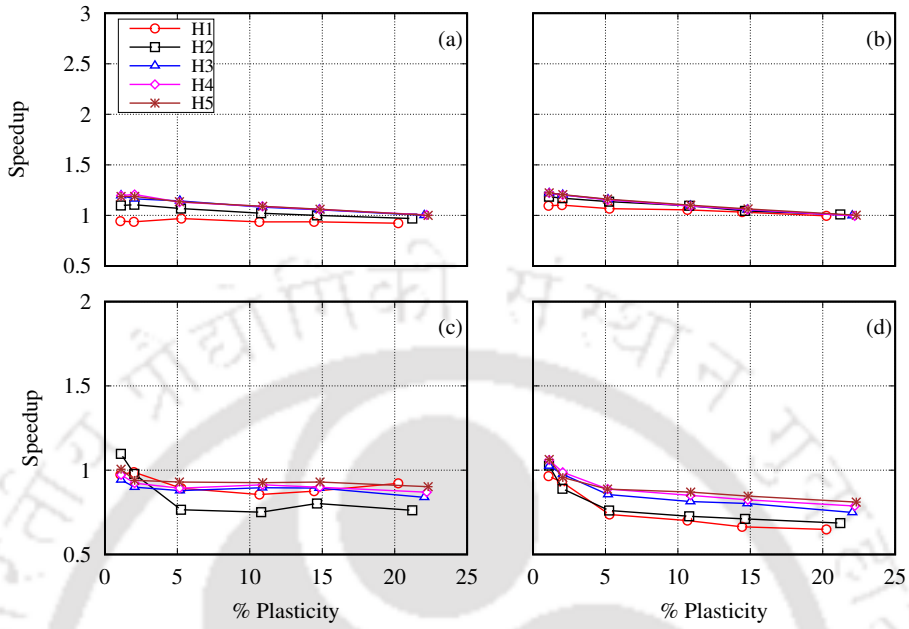


Figure 5.55: Speedup in kernel timings by improved split kernel strategies over single kernel strategies for plate with square holes. Speedups are presented for (a) EbE_{Node}, (b) EbE_{DOF}, (c) NbN and (d) DbD strategies.

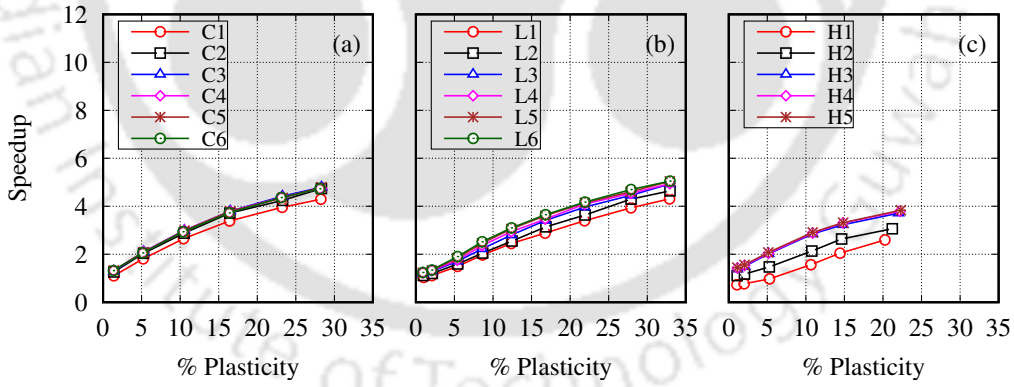


Figure 5.56: Speedup by improved split kernel EbE_{Node} strategy over single kernel NbN strategy for (a) Cantilever beam, (b) L-bracket and (c) Plate with multiple holes.

by improved split kernel strategy for NbN and Dbd strategies.

The proposed EbE strategies based on the improved split kernel strategy obtain considerable speedups at low percentage of plasticity and less or no speedups at high percentage of plasticity. In order to gain more insights into this behavior, kernel timings as a function of Newton iterations are displayed in Fig. 5.62 for all three examples. The kernel timings of two best performing strategies EbE_{Node} and EbE_{DOF} are shown in both

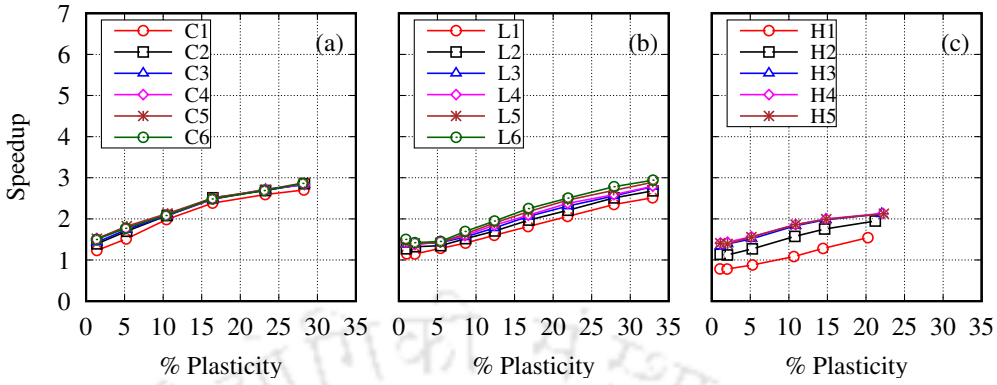


Figure 5.57: Speedup by improved split kernel EbE_{Node} strategy over single kernel DbD strategy for (a) Cantilever beam, (b) L-bracket and (c) Plate with multiple holes.

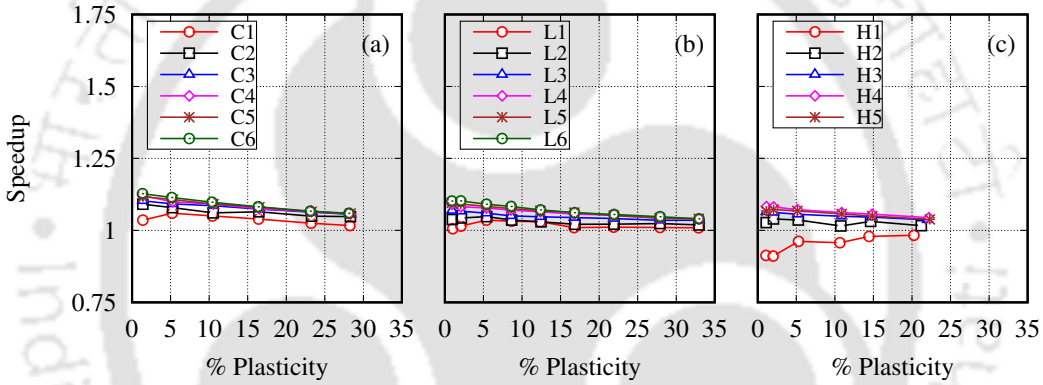


Figure 5.58: Speedup by improved split kernel EbE_{Node} strategy over improved split kernel EbE_{DOF} strategy for (a) Cantilever beam, (b) L-bracket and (c) Plate with multiple holes.

single kernel and improved split kernel variations for the finest mesh in each example. In each sub-figure, kernel timings are plotted for approximately 1% plasticity in part 'a' and for 25%, 32% and 21% plasticity for the cantilever, the L-bracket and the plate with square holes examples, respectively, in part 'b'. It is clearly seen that at low plasticity level advantage of the improved split kernel strategy persist over all Newton iterations, leading to significant reduction in wall-clock timings. The reduction is observed more in the EbE_{DOF} than EbE_{Node} strategy. At high percentage plasticity, advantage by the improved split kernel strategy is found to decrease with Newton iterations for all three examples. The improved split kernel strategies perform well during initial stages as the number of elements in the elastic zone is large. As the Newton iteration increases, more elements move to the plastic zone, and consequently, time spent in the computation for elastic elements becomes insignificant. Therefore, timings of the improved split kernel

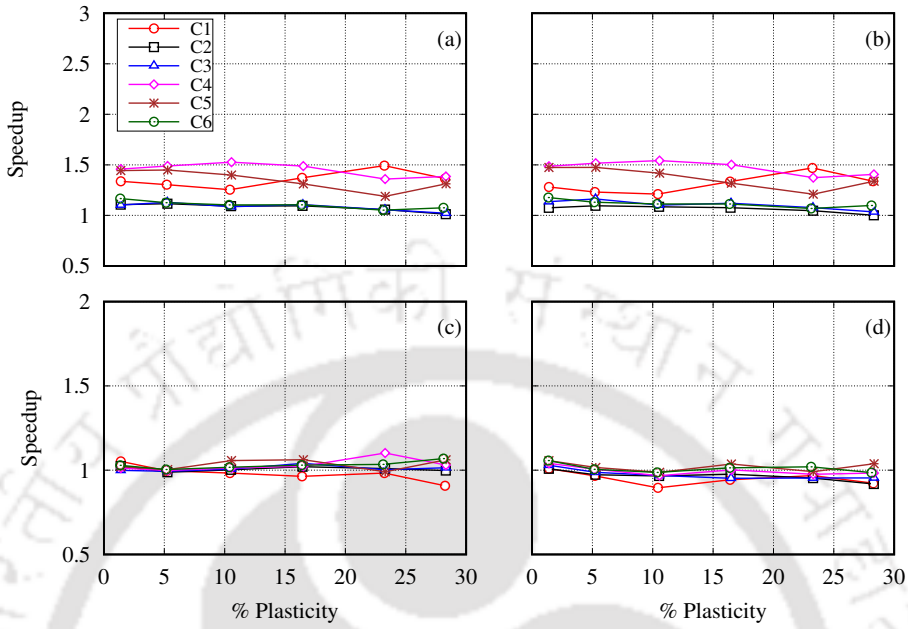


Figure 5.59: Speedup in wall-clock timings by improved split kernel strategies over single kernel strategies for cantilever beam example. Speedups are presented for (a) EbE_{Node}, (b) EbE_{DOF}, (c) NbN and (d) DbD strategies.

strategy are found close to single kernel strategy towards the end of Newton iterations. This leads to less significant reduction in wall-clock timings by the improved split kernel strategy at high percentage of plasticity.

5.3.4.5 Comparison with assembly-based solver

The performance of the best performing improved split kernel EbE_{Node} strategy is compared against assembly-based elastoplasticity solver to emphasize on advantages or limitations and determine the suitability for practical purposes. A GPU optimized assembly-based elastoplasticity solver from Chapter 3 is considered for the comparison. The difference with the assembly-based solver lies in the implementation of the linear solver step. The assembly-based elastoplasticity solver uses CG implementation from Ginkgo (Anzt et al. 2022) library to perform computation on the GPU. It is the performance of the linear solver that determines the overall performance of a GPU-based elastoplasticity solver. Figure 5.63 shows the speedup by the improved split kernel EbE_{Node} strategy over assembly-based solver for the cantilever beam example. The speedups for all meshes

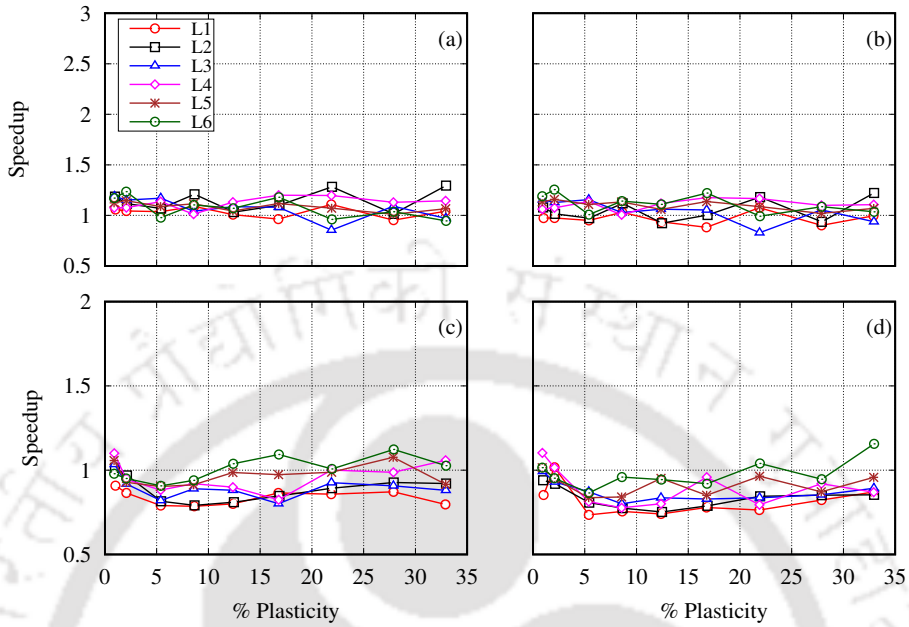


Figure 5.60: Speedup in wall-clock timings by improved split kernel strategies over single kernel strategies for L-bracket. Speedups are presented for (a) EbE_{Node}, (b) EbE_{DOF}, (c) *NbN* and (d) *DbD* strategies.

are found to decrease with increasing percentage plasticity. A maximum speedup of up to 2× is found at approximately 1% plasticity. A similar trend is observed in the case of the L-bracket and the plate with square holes examples (see Figs. 5.64 and 5.65), where speedups of up to 2× and 2.2× are obtained respectively, at low percentage of plasticity. In the plate with square holes example, speedup of 1.6× is obtained at 21% of plasticity. Overall, the proposed matrix-free solver is found twice as fast as an assembly-based solver for low plasticity percentage.

5.3.5 Summary

We proposed an improved split kernel strategy for matrix-free SpMV computation in elastoplasticity. The improved split kernel strategy separates the computation for elastic and plastic states into two GPU kernels, where each dedicated kernel is optimized to perform the assigned task efficiently. We also proposed *EbE* matrix-free strategy for computation of SpMV in elastoplasticity. Depending on the thread allocation strat-

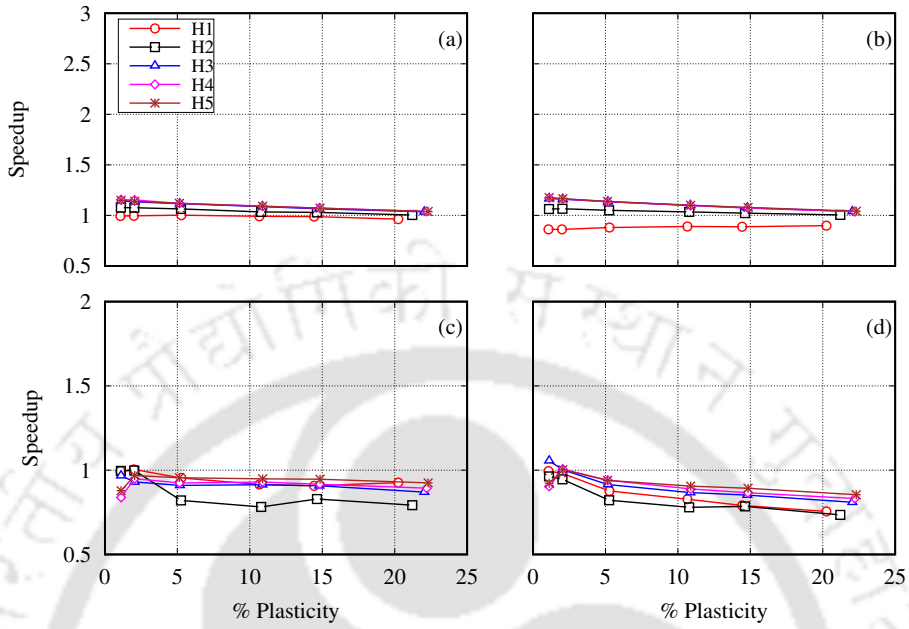
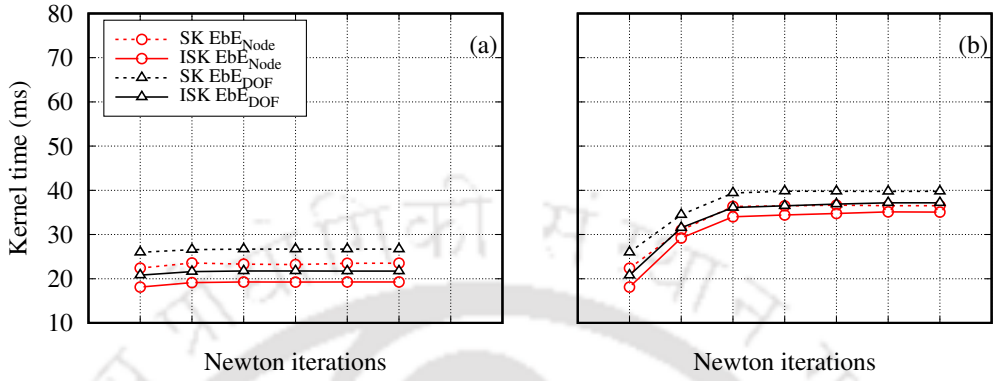
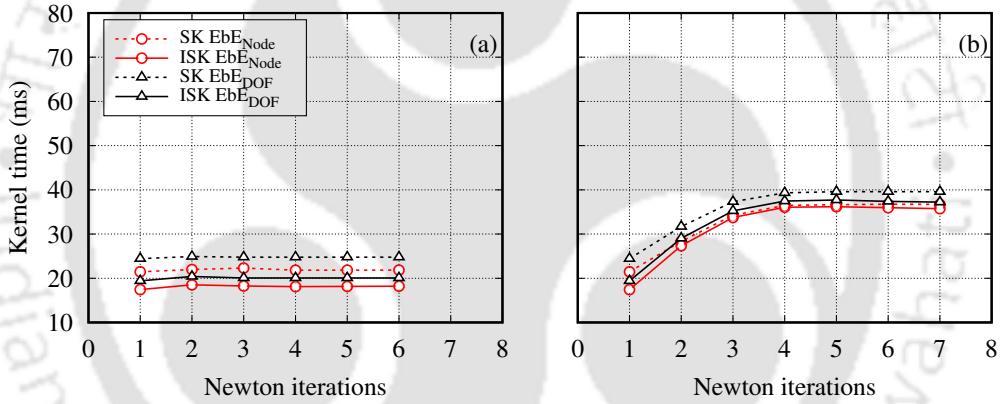


Figure 5.61: Speedup in wall-clock timings by improved split kernel strategies over single kernel strategies for plate with square holes. Speedups are presented for (a) EbE_{Node}, (b) EbE_{DOF}, (c) NbN and (d) DbD strategies.

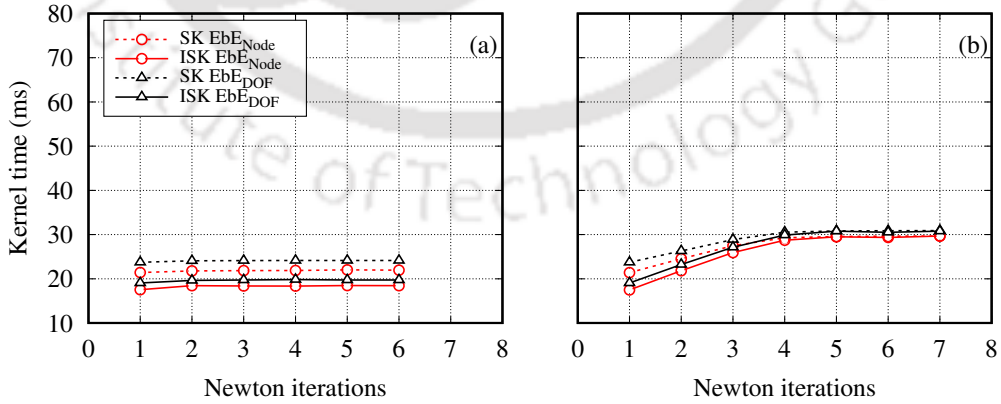
egy, the EbE strategy was implemented with two variants: EbE_{Node} and EbE_{DOF}. The EbE_{Node} uses node-based thread assignment and EbE_{DOF} uses DOF-based thread assignment. The performance of the proposed strategies were evaluated over three benchmark examples from elastoplasticity and comparison was done to find the best SpMV strategy. Compared to single kernel strategy, the improved split kernel strategy achieved 1.02–1.2× speedup for EbE_{Node} and 1.05–1.25× speedup for EbE_{DOF} strategy. In both cases, speedups are found to decrease with increasing percentage of plasticity. The NbN and DbD strategies were found performing poorly with the improved split kernel strategy. The improved split kernel EbE_{Node} strategy was found to the fastest achieving up to 5× speedup over NbN, 3× over DbD and 1.12× over EbE_{DOF} strategy. A comparison with assembly-based elastoplasticity solver revealed up to 2× speedup at low percentage of plasticity.



(a) Cantilever with a cut out.



(b) L-bracket.



(c) Plate with square holes

Figure 5.62: Variation of kernel timings with Newton-Raphson iterations in each example. In the legend 'SK' refers to single kernel and 'ISK' refers to improved split kernel.

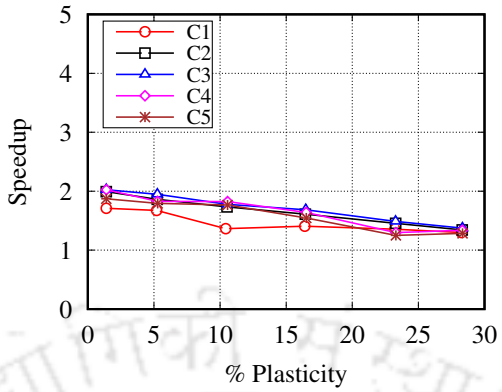


Figure 5.63: Speedup by improved split kernel EbE_{Node} strategy over assembly-based elastoplasticity solver for cantilever beam.

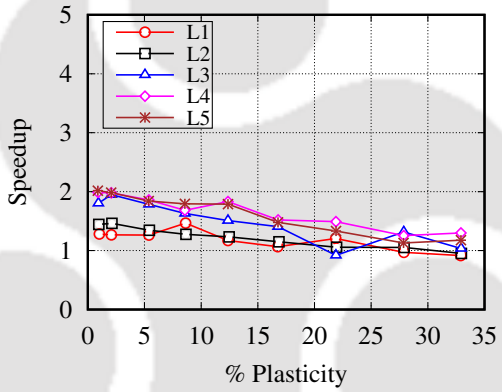


Figure 5.64: Speedup by improved split kernel EbE_{Node} strategy over assembly-based elastoplasticity solver for L-bracket.

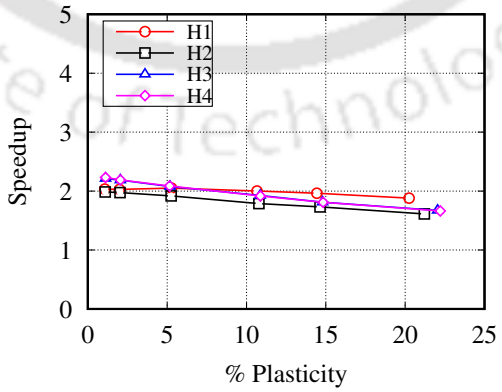


Figure 5.65: Speedup by improved split kernel EbE_{Node} strategy over assembly-based elastoplasticity solver for plate with multiple holes.

5.4 Closure

In this chapter, GPU-based matrix-free SpMV strategies have been presented to use in CG iterative solver for finite element analysis of elastoplastic problems. The parallel implementation of matrix-free SpMV in elastoplasticity is affected by branching issues due to the presence of both elastic and plastic states. In this work, two strategies were proposed, namely single kernel strategy and improved split kernel strategy, that prevents branching and provides uniform treatment of elastic and plastic states. The single kernel strategy uses one GPU kernel for the computation of SpMV, whereas the improved split kernel strategy uses separate GPU kernel for elastic and plastic states. Both the proposed strategies depend on voxel-based finite element mesh and make efficient use of elements congruency. Specifically, only one elemental tangent matrix is used for elements lying in the elastic zone and individual tangent matrices for elements in the plastic zone. The proposed strategies have been implemented on GPU with *NbN*, *DbD* and *EbE* strategies. Further, the *EbE* strategy was implemented using both node-wise (EbE_{Node} strategy) and DOF-wise (EbE_{DOF} strategy) thread assignment. The data race condition associated with the *EbE* strategy was handled by mesh coloring method.

The performance of the proposed strategies was evaluated over multiple benchmark examples from elastoplasticity. For a comprehensive study, numerical experiments were performed with different levels of mesh refinement and the applied load was varied to get different amounts of plasticity in the body. The single kernel *NbN* strategy was found to be 3.2× fast than existing GPU-based split kernel strategy at low plasticity levels. At higher levels of plasticity, the performance of single kernel *DbD* was found 3.5× better than the existing split kernel strategy. However, the performance of *DbD* and *NbN* strategies were found to degrade with improved split kernel strategy. The improved split kernel EbE_{Node} and EbE_{DOF} strategies were found to achieve up to 1.2× and 1.25× speedups, respectively over their single kernel variants. The EbE_{Node} strategy was found to be the best, obtaining speedups up to 5×, 3× and 1.12× over single kernel *NbN*, single kernel *DbD* and improved split kernel EbE_{DOF} strategies, respectively. The splitting of computations in improved split kernel strategy for elastic and plastic zones showed a clear advantage for low plasticity levels. However, at higher plasticity levels, the advantage was found confined to initial stages of Newton iterations. The comparison of wall-clock timings with GPU optimized assembly-based elastoplasticity solver using CG

implementation from Ginkgo library showed speedups of up to 2× for all computational examples.

At low plasticity levels, performance of the proposed strategies is dictated by computing power of the hardware as memory access is very small. However, as the amount of plasticity increases, the increase in the memory access becomes a performance bottleneck. In both these cases, the proposed strategies are expected to observe increased performance on recent versions of GPUs. GPUs like V100 and A100 have much higher computing power and memory bandwidth than K40. The proposed strategies can be run on recent GPUs without modification and with higher performance.

In future, the proposed matrix-free SpMV strategies can be used to make an even faster CG solver for elastoplasticity by incorporating techniques like preconditioning to improve the convergence.

Chapter 6

Conclusions and Future work

In this thesis, a GPU-accelerated framework for elastoplastic analysis was presented. In an attempt to minimize the simulation timings, GPU-based parallel strategies were proposed for all expensive steps in elastoplastic analysis except sparse linear solver. The proposed framework involves no CPU-GPU data transfer and uses an optimum amount of memory. In order to improve the performance further, matrix-free strategies were presented to compute SpMV in CG iterative solver. For problems using unstructured mesh, EbE_{sym} strategy that uses only the symmetric part of elemental tangent matrices for the computation of SpMV was proposed. For problems using voxel-based structured mesh, single kernel and improved split kernel strategies were proposed to efficiently handle branching issues due to the presence of elastic and plastic states. The GPU implementation of matrix-free SpMV for voxel-based mesh was done by DbD, NbN and EbE matrix-free strategies with proposed modifications. The proposed strategies were tested on several benchmark examples, and their performance was studied.

6.1 Conclusions

Following conclusions can be drawn from the work presented in this thesis.

- In assembly-based approach, single thread is allotted to each finite element for the computation of elemental matrices, computation of stress using radial return

method and computation of internal force vectors. The proposed strategy makes heavy use of local memory and performs assembly directly into CSR sparse storage format using pre-computed indices. The computational experiments showed significant speedups by the proposed strategies over single core CPU implementation, achieving 20.4–69.7× for computation of elemental matrices and assembly, 47.2–66.1× for computation of stress, and 53.7–67.3× for computation of internal force vector and their assembly. Comparison of wall-clock timings with CPU implementation that uses GPU-based linear solver showed speedups in the range 1.4–7.2×.

- In the computation of elemental matrices and assembly, speedups were found to increase with increasing amount of plasticity. However, speedups remained invariable with plasticity for the computation of stress and internal force vector.

- As a result of the proposed parallel strategies, the proportion of the sparse linear solver timings in wall-clock timings of the GPU implementation reached up to 98.9%. The proposed GPU-based framework is found to be memory efficient as it is able to solve problems with up to 5.1 million DOFs on a GPU with 12 GB memory.

- For problems using unstructured mesh, the performance of all standard matrix-free SpMV strategies (*NbN*, *DbD* and *EbE*) in the literature was found bound by the memory bandwidth of the hardware. The *EbE* strategy was found to be the best in terms of run time.

- For problems using unstructured mesh, a novel matrix-free strategy was proposed that uses only symmetric part of elemental matrices. The proposed strategy, referred as EbE_{sym}, was found at least 1.3× faster than *EbE* strategy for problems using unstructured mesh. The EbE_{sym} strategy occupies 1.77× and 1.92× less memory for linear quadrilateral and linear hexahedral elements, respectively.

- For problems using voxel-based structured mesh, two strategies are proposed to handle branching issues due to the presence of elastic and plastic states. The proposed single kernel strategy avoids thread divergence and prevents redundant computation as compared with split kernel strategy from the literature. At low

plasticity levels, the single kernel *NbN* strategy achieved 3.2× speedup over the split kernel *NbN* strategy. At moderate to high amount of plasticity, the single kernel *DbD* strategy showed up to 3.5× speedup over the split kernel *NbN* strategy from the literature. Further, *EbE* strategies were found to achieve the best performance than *NbN* and *DbD* strategies with the single kernel strategy.

- For problems using voxel-based structured mesh, the proposed improved split kernel strategy separates the computation of matrix-free SpMV into elastic and plastic components, providing uniform memory access and preventing thread divergence. The improved split kernel EbE_{Node} (*EbE* strategy with node-based thread assignment) was found to achieve 1.02–1.2× speedup with respect to its single kernel variant. Similarly, the improved split kernel EbE_{DOF} (*EbE* strategy with DOF-based thread assignment) was found to achieve 1.05–1.25× speedups. The *NbN* and *DbD* strategies were found performing poorly with the improved split kernel strategy.

- The improved split kernel EbE_{Node} was found to be the best performing matrix-free SpMV strategy for elastoplasticity, achieving up to 5×, 3× and 1.12× speedups over single kernel *NbN*, single kernel *DbD* and improved split kernel EbE_{DOF} strategies, respectively.

- The matrix-free elastoplastic solver, using the improved split kernel EbE_{Node} matrix-free SpMV strategy in CG linear solver, was found to achieve up to 2× speedup over the assembly-based elastoplasticity solver using the CG implementation from Ginkgo library.

6.2 Scope for Future Works

The results reported in this thesis provide a perspective for future research in many directions. They are listed below.

- All the proposed strategies in this thesis are developed for 8-noded hexahedral elements. However, many problems require the use of higher-order hexahedrals or other types of elements for accurate solutions, demanding more computational

resources and memory space. As per literature, matrix-free strategies are more efficient for higher-order elements. Therefore, the proposed strategies are expected to be more beneficial for higher-order elements. Also, applications to other types of elements is expected to make the proposed strategies more general and applicable to wide range of problems.

- In this thesis, all proposed strategies were implemented on a single GPU. However, this limits the maximum size of the problem. In order to solve a bigger problem, the proposed works can be extended to use multiple GPUs connected to the same node as well as different nodes.
- An efficient construction of advanced preconditioners can be implemented to accelerate the solution of linear system of equations in both assembly-based and matrix-free strategies.
- In this thesis, only isotropic linear hardening model is used within the context of associated flow rule. As a future research direction, the proposed strategies can be extended to anisotropic hardening model with linear or nonlinear strain hardening relation. The problems involving non-associated flow rule can also be addressed.
- The work presented in this thesis can serve as a reference for GPU acceleration of computational framework for other types of inelastic material behavior like viscoplasticity and applications like contact-impact problems as in crashworthiness.

Appendices

A GPU Architecture

GPUs were originally developed as fixed-function graphics processors. Although the sole purpose was to render images to display, it required intense computation on the parallel data set. Driven by the growing demand from the gaming industry to produce fast and high-definition graphics cards, a lot of research effort was put into innovating GPUs. As a result, GPU evolved into a massively parallel hardware capable of doing a huge amount of computation in a limited amount of time. GPUs are particularly suitable for those tasks that can be decomposed into a large number of independent tasks and require intensive computation. This is reflected by the architectural design of GPUs. Keeping in mind the throughput requirement, much of the chip area is dedicated to computational units. Figure A.1 illustrates the architectural difference between a CPU and a GPU. It can be seen that a CPU dedicates much of the chip area to RAM, control units, cache, etc., but a GPU assigns most of the chip area to house arithmetic logic units (ALU). As a result, GPU can perform more number of tasks in a given amount of time than a CPU. A GPU has a large number of massive threaded physical cores that can launch

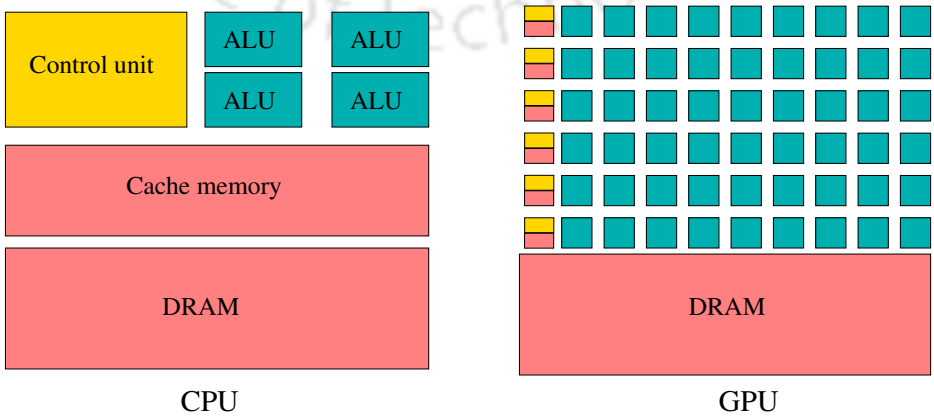


Figure A.1: Difference between CPU and GPU architectures.

thousands of threads in parallel. As a result, many threads share an instruction unit and execute in a single instruction multiple data (SIMD) fashion. Since a large number of threads execute in parallel, a GPU provides a large memory bandwidth to supply data to compute cores. The GPU developed by NVIDIA consists of a number of streaming multiprocessor (SMs). Each SM is an independent computational unit that contains a number of streaming processors (CUDA cores) to perform computations simultaneously. An SM also contains cache units for fast and efficient management of data access. GPU architecture is designed to be energy efficient and has a larger performance-per-watt ratio than a CPU. Due to this reason, GPUs are extensively used as accelerators in supercomputers.

B CUDA programming model

Compute Unified Device Architecture (CUDA) (NVIDIA Corporation 2022) is a parallel programming platform that allows usage of NVIDIA GPU for accelerated computing. CUDA provides C like programming syntax to access compute resources of a GPU and flexibility of using popular languages like C, C++, FORTRAN, Python, etc. to program the hardware.

A CUDA application consists of a sequential program running on a CPU (referred as host) that launches compute kernels to execute on a GPU (referred as device). A kernel is a special function written in CUDA using a particular syntax extension of C language. The kernel may generate a grid of thousands or even millions of threads to parallelize a given task. A grid is a set of blocks, where every block can be identified with either one or two-dimensional indices. Each Block consists of a large number of threads, where threads are arranged in one, two, or three-dimensional patterns. For a thread block, an SM schedules threads in a group of 32, called as warps. All the threads in a warp execute the same instruction, whereas warps itself may execute in any order with respect to each other. Since threads in a warp follow the same instruction at a time, all the threads must take the same execution path. If threads in a warp follow different execution paths, the warp is required to execute as many passes as a number of execution paths to complete the task. This is known as thread divergence and should be avoided whenever possible. There are some situations in which many threads working in parallel

tend to accumulate values simultaneously at the same location in GPU memory. There is no definite outcome of this operation, and hence, the result remains undetermined. This is known as race condition. The race condition is avoided by rewriting the code such that no two threads access the same piece of memory at the same time. Another way is to use expensive atomic operations provided by CUDA.

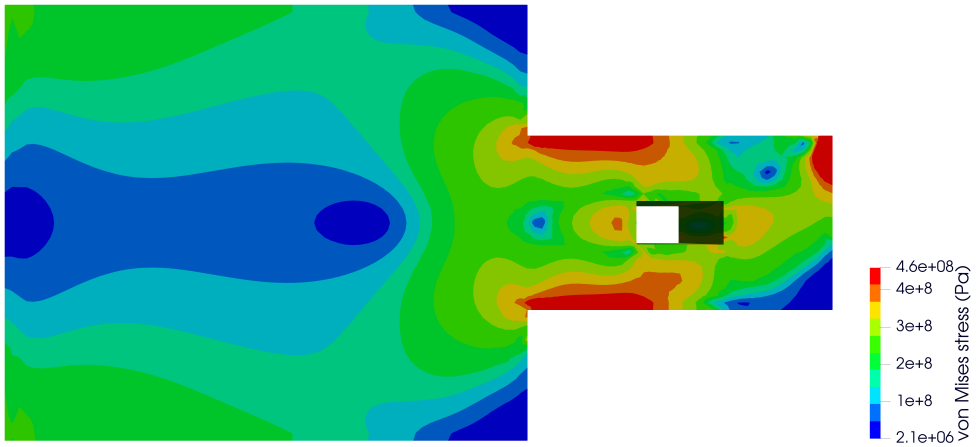
CUDA provides access to several levels of memory available in a GPU to achieve efficient data management. Three types of memory available on a GPU are global memory, shared memory and registers. Among these, the global memory is the slowest, whereas registers are the fastest memory type. However, global memory is much larger in size and can be used to store large data sets. Though small, shared memory and registers can be used judiciously to improve data reuse and reduce data traffic from the global memory. Registers are closest to CUDA cores and provide data almost instantaneously for computation. The register files are private to each thread and have the lifetime of a thread. The shared memory is accessible by all threads in a block and has the lifetime of the block. It is bigger in size than registers and can be used to share data among threads of a block. A wise use of shared memory can reduce the number of transactions to global memory. The global memory is the largest pool of memory and can be used to copy data from CPU. It has the lifetime of the application and can be accessed by all threads of the grid. However, care must be taken while reading or writing data to the global memory. Since all threads can access global memory simultaneously, synchronization techniques must be used to avoid the race condition. When all the threads in a warp read/write data to different and highly strided locations in global memory, the memory requests are serialized. If global memory is accessed in consecutive locations by consecutive threads with proper alignment, all memory requests are coalesced into one single transaction. Coalesced memory access leads to efficient use of memory bandwidth. A thread can also allocate memory statically from within a kernel, known as local memory. The local memory has the lifetime of the thread and remains private to the thread that allocates it. The data stored in the local memory resides either in registers or spilled over to the global memory.

Each kernel must specify the number of threads per block and the number of blocks in the grid. For NVIDIA Tesla K40, the maximum number of threads per SM is 2048, and there can be 1024 threads per block. This imposes a limitation on the number of blocks

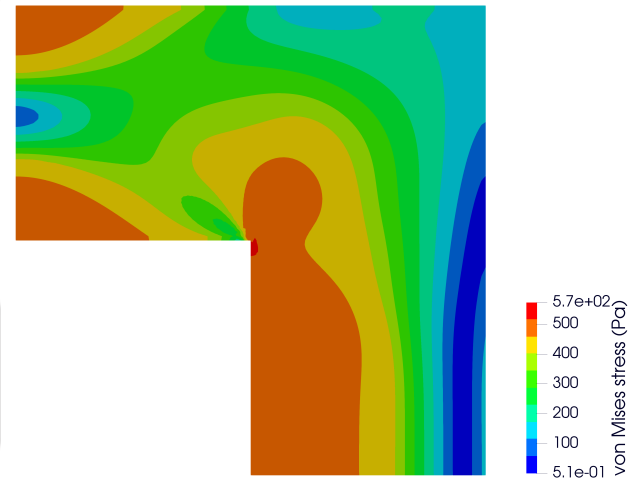
that can run concurrently on an SM. The number of threads that can run simultaneously on an SM also depends on the utilization of resources in an SM. The total amount of registers used by all the threads must not exceed the given limit for a multiprocessor. If it happens, the device automatically reduces the number of active threads to bring down the consumption. This leads to under utilization of GPU as the application is not able to fully occupy the hardware resources. The same rule also applies to the usage of shared memory. Higher occupancy of a GPU is desirable for better performance.

C Visualization

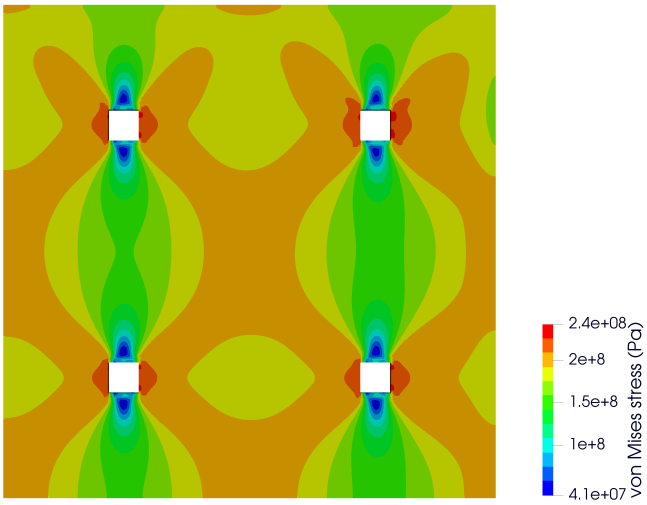
Figure C.1 shows the distribution of von Mises stress for three benchmark examples from Chapter 5.



(a) Cantilever beam with a cut out.



(b) L-bracket.



(c) Plate with square holes.

Figure C.1: Distribution of von Mises stress in each example for elastoplastic analysis.

References

Abdelfattah, A., Dongarra, J., Keyes, D. & Ltaief, H. (2012), Optimizing memory-bound SYMV kernel on GPU hardware accelerators, *in* 'International Conference on High Performance Computing for Computational Science', Springer, pp. 72–79.

Adams, M. F., Bayraktar, H. H., Keaveny, T. M. & Papadopoulos, P. (2004), Ultra-scalable implicit finite element analyses in solid mechanics with over a half a billion degrees of freedom, *in* 'SC'04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing', IEEE, pp. 34–34.

Aissa, M., Verstraete, T. & Vuik, C. (2017), 'Toward a GPU-aware comparison of explicit and implicit CFD simulations on structured meshes', *Computers & Mathematics with Applications* **74**(1), 201–217.

Anzt, H., Cojean, T., Flegar, G., Göbel, F., Grützmacher, T., Nayak, P., Ribizel, T., Tsai, Y. M. & Quintana-Ortí, E. S. (2022), 'Ginkgo: A modern linear operator algebra framework for high performance computing', *ACM Transactions on Mathematical Software* **48**(1), 2.

Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W. et al. (2006), The landscape of parallel computing research: A view from berkeley, Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

Balay, S., Abhyankar, S., Adams, M. F., Benson, S., Brown, J., Brune, P., Buschelman, K., Constantinescu, E. M., Dalcin, L., Dener, A., Eijkhout, V., Gropp, W. D., Hapla, V., Isaac, T., Jolivet, P., Karpeev, D., Kaushik, D., Knepley, M. G., Kong, F., Kruger, S., May, D. A., McInnes, L. C., Mills, R. T., Mitchell, L., Munson, T., Roman, J. E.,

Rupp, K., Sanan, P., Sarich, J., Smith, B. F., Zampini, S., Zhang, H., Zhang, H. & Zhang, J. (2021), 'PETSc Web page', https://petsc.org/. (accessed December 2021).

Banaś, K., Płaszewski, P. & Macioł, P. (2014), 'Numerical integration on GPUs for higher order finite elements', *Computers & Mathematics with Applications* **67**(6), 1319–1344.

Banaś, K., Krużel, F. & Bielański, J. (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.

Baroutaji, A., Sajjia, M. & Olabi, A.-G. (2017), 'On the crashworthiness performance of thin-walled energy absorbers: Recent advances and future developments', *Thin-Walled Structures* **118**, 137–163.

Barragy, E. & Carey, G. F. (1988), 'A parallel element-by-element solution scheme', *International Journal for Numerical Methods in Engineering* **26**(11), 2367–2382.

Bathe, K. J. (1996), *Finite Element Procedures*, Prentice Hall of India, New Delhi.

Bell, N. & Garland, M. (2009), Implementing sparse matrix-vector multiplication on throughput-oriented processors, *in* 'Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis', pp. 1–11.

Bell, N. & Hoberock, J. (2012), Thrust: A productivity-oriented library for CUDA, *in* W.-m. W. Hwu, ed., 'GPU Computing Gems Jade Edition', Applications of GPU Computing Series, Morgan Kaufmann, Boston, pp. 359–371.

Bhardwaj, M., Pierson, K., Reese, G., Walsh, T., Day, D., Alvin, K., Peery, J., Farhat, C. & Lesoinne, M. (2002), Salinas: A scalable software for high-performance structural and solid mechanics simulations, *in* 'SC'02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing', IEEE, pp. 35–35.

Bolz, J., Farmer, I., Grinspun, E. & Schröoder, P. (2003), 'Sparse matrix solvers on the GPU: Conjugate gradients and multigrid', *ACM Transactions on Graphics* **22**(3), 917–924.

Brodtkorb, A. R., Hagen, T. R. & Sætra, M. L. (2013), 'Graphics processing unit (GPU) programming strategies and trends in GPU computing', *Journal of Parallel and Distributed Computing* **73**(1), 4–13.

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

Cai, Y., Wang, G., Li, G. & Wang, H. (2015), 'A high performance crashworthiness simulation system based on GPU', *Advances in Engineering Software* **86**, 29–38.

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

Carey, G. F., Barragy, E., McLay, R. & Sharma, M. (1988), 'Element-by-element vector and parallel computations', *Communications in Applied Numerical Methods* **4**(3), 299–307.

Carey, G. F. & Jiang, B.-N. (1986), 'Element-by-element linear and nonlinear solution schemes', *International Journal for Numerical Methods in Biomedical Engineering* **2**(2), 145–153.

Carter, W., Fuller, E. R. & Langer, S. A. (2001), 'OOF: An image-based finite-element analysis of material microstructures', *Computing in Science & Engineering* **3**(03), 15–23.

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

Charara, A., Keyes, D. & Ltaief, H. (2019), 'Batched triangular dense linear algebra kernels for very small matrix sizes on GPUs', *ACM Transactions on Mathematical Software (TOMS)* **45**(2), 1–28.

Dalton, S., Bell, N., Olson, L. & Garland, M. (2014), 'CUSP: Generic parallel algorithms for sparse matrix and graph computations'. Version 0.5.0.
URL: *http://cusplibrary.github.io/*

de Souza Neto, E. A., Perić, D. & Owen, D. R. J. (2008), *Computational Methods for Plasticity: Theory and Applications*, John Wiley & Sons, Ltd.

Ding, K., Qin, Q.-H., Cardew-Hall, M. & Kalyanasundaram, S. (2008), 'Efficient parallel algorithms for elastic–plastic finite element analysis', *Computational Mechanics* **41**(4), 563–578.

Dunne, F. & Petrinic, N. (2005), *Introduction to computational plasticity*, Oxford University Press, Oxford, New York.

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

El Sayed, T., Mota, A., Fraternali, F. & Ortiz, M. (2008), 'A variational constitutive model for soft biological tissues', *Journal of Biomechanics* **41**(7), 1458–1466.

Enos, J., Steffen, C., Fullop, J., Showerman, M., Shi, G., Esler, K., Kindratenko, V., Stone, J. E. & Phillips, J. C. (2010), Quantifying the impact of GPUs on performance and energy efficiency in HPC clusters, *in* 'International Conference on Green Computing', pp. 317–324.

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

Fehn, N., Wall, W. A. & Kronbichler, M. (2019), 'A matrix-free high-order discontinuous Galerkin compressible Navier-Stokes solver: A performance comparison of compressible and incompressible formulations for turbulent incompressible flows', *International Journal for Numerical Methods in Fluids* **89**(3), 71–102.

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

Filippone, S., Cardellini, V., Barbieri, D. & Fanfarillo, A. (2017), 'Sparse matrix-vector multiplication on GPGPUs', *ACM Transactions on Mathematical Software (TOMS)* **43**(4), 1–49.

Fu, Z., Lewis, T. J., Kirby, R. M. & Whitaker, R. T. (2014), 'Architecting the finite element method pipeline for the GPU', *Journal of computational and applied mathematics* **257**, 195–211.

Gautam, S. S., Babu, R. & Dixit, P. (2011), 'Ductile fracture simulation in the taylor rod impact test using continuum damage mechanics', *International Journal of Damage Mechanics* **20**(3), 347–369.

Gautam, S. S. & Dixit, P. M. (2010), 'Ductile failure simulation in spherodized steel using a continuum damage mechanics coupled finite element formulation', *International Journal of Computational Methods* **07**(02), 319–348.

Gautam, S. S. & Dixit, P. M. (2012), 'Numerical simulation of ductile fracture in cylindrical tube impacted against a rigid surface', *International Journal of Damage Mechanics* **21**(3), 341–371.

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

Göddeke, D., Strzodka, R., Mohd-Yusof, J., McCormick, P., Buijssen, S. H. M., Grajewski, M. & Turek, S. (2007), 'Exploring weak scalability for FEM calculations on a GPU-enhanced cluster', *Parallel Computing* **33**(10-11), 685–699.

Guennebaud, G., Jacob, B. et al. (2010), 'Eigen v3', http://eigen.tuxfamily.org.

Guha, I., Zhang, X., Rajapakse, C. S., Chang, G. & Saha, P. K. (2022), 'Finite element analysis of trabecular bone microstructure using CT imaging and continuum mechanical modeling', *Medical Physics* **49**(6), 3886–3899.

He, G., Wang, H., Huang, G., Liu, H. & Li, G. (2017), 'A parallel elastoplastic reanalysis based on GPU platform', *International Journal of Computational Methods* **14**(05), 1750051.

Hong, Y., Wang, L., Zhang, J. & Gao, Z. (2020), '3D elastoplastic model for fine-grained gassy soil considering the gas-dependent yield surface shape and stress-dilatancy', *Journal of Engineering Mechanics* **146**(5), 04020037.

Hughes, T. J. R., Levit, I. & Winget, J. (1983), 'An element-by-element solution algorithm for problems of structural and solid mechanics', *Computer Methods in Applied Mechanics and Engineering* **36**(2), 241 – 254.

Irina, D., Matsuoka, S. & Toshio, E. (2011), GPU-based approach for elastic-plastic deformation simulations, Technical Report 12, Information Processing Society of Japan (IPSJ).

Jiang, M., Jasiuk, I. & Ostoja-Starzewski, M. (2002), 'Apparent elastic and elastoplastic behavior of periodic composites', *International Journal of Solids and Structures* **39**(1), 199–212.

Jones, N. (2011), *Structural Impact*, 2^{nd} edn, Cambridge University Press, New York.

Khalevitsky, Y. V., Burmasheva, N. V. & Konovalov, A. V. (2016), 'An approach to the parallel assembly of the stiffness matrix in elastoplastic problems', *AIP Conference Proceedings* **1785**(1), 040023.

Khalevitsky, Y. V., Burmasheva, N. V., Konovalov, A. V. & Partin, A. S. (2016), 'Comparative study of Krylov subspace method implementations for a GPU cluster in elastoplastic problems', *AIP Conference Proceedings* **1785**(1), 040024.

Khronos Group (2020), 'OpenCL - the open standard for parallel programming of heterogeneous systems', https://www.khronos.org/opencl/. Version 3.0.

Kim, N.-H. (2015), *Introduction to Nonlinear Finite Element Analysis*, Springer, New York.

Kiran, U., Gautam, S. S. & Sharma, D. (2020), 'GPU-based matrix-free finite element solver exploiting symmetry of elemental matrices', *Computing* **102**(9), 1941–1965.

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

Kiran, U., Sharma, D. & Gautam, S. S. (2023), 'A GPU-based framework for finite element analysis of elastoplastic problems', *Computing (2023)* **105**, 1673–1696.

Kiran, U., Sharma, D. & Gautam, S. S. (2024*a*), 'Development of GPU-based matrix-free strategies for large-scale elastoplasticity analysis using conjugate gradient solver', *International Journal for Numerical Methods in Engineering* **125**(7), e7421.

Kiran, U., Sharma, D. & Gautam, S. S. (2024*b*), 'An efficient framework for matrix-free spmv computation on gpu for elastoplastic problems', *Mathematics and Computers in Simulation* **216**, 318–346.

Kiss, I., Badics, Z., Gyimóthy, S. & Pávó, J. (2012), High locality and increased intra-node parallelism for solving finite element models on GPUs by novel element-by-element implementation, *in* '2012 IEEE Conference on High Performance Extreme Computing (HPEC)', pp. 1–5.

Kiss, I., Gyimóthy, S., Badics, Z. & Pávó, J. (2012), 'Parallel realization of the element-by-element FEM technique by CUDA', *Magnetics, IEEE Transactions on* **48**(2), 507–510.

Knaus, R. (2022), 'A fast matrix-free approach to the high-order control volume finite element method with application to low-mach flow', *Computers & Fluids* **239**, 105408.

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

Lopes, P. C. F., Pereira, A. M. B., Clua, E. W. G. & Leiderman, R. (2022), 'A GPU implementation of the PCG method for large-scale image-based finite element analysis in heterogeneous periodic media', *Computer Methods in Applied Mechanics and Engineering* **399**, 115276.

Macioł, P., Płaszewski, P. & Banaś, K. (2010), '3D finite element numerical integration on GPUs', *Procedia Computer Science* **1**(1), 1093–1100.

Maier, G. & Hueckel, T. (1979), 'Nonassociated and coupled flow rules of elastoplasticity for rock-like materials', *International Journal of Rock Mechanics and Mining Sciences & Geomechanics Abstracts* **16**(2), 77–92.

Markall, G. R., Ham, D. A. & Kelly, P. H. (2010), 'Towards generating optimised finite element solvers for GPUs from high-level specifications', *Procedia Computer Science* **1**(1), 1815–1823.

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

Markopoulos, A., Hapla, V., Cermak, M. & Fusek, M. (2015), 'Massively parallel solution of elastoplasticity problems with tens of millions of unknowns using Permoncube and FLLOP packages', *Applied Mathematics and Computation* **267**, 698–710.

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

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

Meguid, S. (2021), 'Elasto-plastic behaviour of cantilever beams containing varied stress concentration cut out features', *International Journal of Mechanics and Materials in Design* **17**(3), 453–462.

Meschke, G., Lackner, R. & Mang, H. A. (1998), 'An anisotropic elastoplastic-damage model for plain concrete', *International Journal for Numerical Methods in Engineering* **42**(4), 703–727.

Meyer, A. & Michael, D. (1997), 'A modern approach to the solution of problems of classic elastoplasticity on parallel computers', *Numerical Linear Algebra with Applications* **4**(3), 205–221.

Nath, R., Tomov, S., Dong, T. T. & Dongarra, J. (2011), Optimizing symmetric dense matrix-vector multiplication on GPUs, *in* 'Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis', SC '11, ACM, New York, NY, USA, pp. 1–10.

Nguyen, L. H. & Schillinger, D. (2018), 'A multiscale predictor/corrector scheme for efficient elastoplastic voxel finite element analysis, with application to CT-based bone strength prediction', *Computer Methods in Applied Mechanics and Engineering* **330**, 598–628.

NVIDIA Corporation (2022), 'NVIDIA CUDA C programming guide'. Version 12.0.

Ohshima, S., Hayashi, M., Katagiri, T. & Nakajima, K. (2012), Implementation and evaluation of 3D finite element method application for CUDA, *in* 'High Performance Computing for Computational Science-VECPAR 2012', Springer, pp. 140–148.

Phillips, J. C., Hardy, D. J., Maia, J. D., Stone, J. E., Ribeiro, J. V., Bernardi, R. C., Buch, R., Fiorin, G., Hénin, J., Jiang, W. et al. (2020), 'Scalable molecular dynamics on CPU and GPU architectures with NAMD', *The Journal of chemical physics* **153**(4), 044130.

Pikle, N. K., Sathe, S. R. & Vyavahare, A. Y. (2018), 'High performance iterative elemental product strategy in assembly-free FEM on GPU with improved occupancy', *Computing* **100**, 1273–1297.

Płaszewski, P., Banaś, K. & Macioł, P. (2010), Higher order FEM numerical integration on GPUs with OpenCL, *in* 'Computer Science and Information Technology (IMCSIT), Proceedings of the 2010 International Multiconference on', IEEE, pp. 337–342.

Prabhune, B. C. & Suresh, K. (2020), 'A fast matrix-free elasto-plastic solver for predicting residual stresses in additive manufacturing', *Computer-Aided Design* **123**, 102829.

Ratnakar, S. K., Kiran, U. & Sharma, D. (2022), 'Acceleration of structural topology optimization using symmetric element-by-element strategy for unstructured meshes on GPU', *Engineering Computations* **39**(10), 3354–3375.

Ratnakar, S. K., Sanfui, S. & Sharma, D. (2021), '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.

Rayhani, M. & El Naggar, M. H. (2008), 'Numerical modeling of seismic response of rigid foundation on soft soil', *International Journal of Geomechanics* **8**(6), 336–346.

Reddy, J. N. (2006), *An Introduction to the Finite Element Method*, Vol. 2, 3^{rd} edn, McGraw-Hill Education, New York.

Reese, S. & Wriggers, P. (1997), 'A material model for rubber-like polymers exhibiting plastic deformation: computational aspects and a comparison with experimental results', *Computer Methods in Applied Mechanics and Engineering* **148**(3), 279–298.

Reguly, I. & Giles, M. (2013), 'Finite element algorithms and data structures on graphical processing units', *International Journal of Parallel Programming* **43**(2), 203–239.

Rietmann, M., Grote, M., Peter, D. & Schenk, O. (2017), 'Newmark local time stepping on high-performance computing architectures', *Journal of Computational Physics* **334**, 308–326.

Rodríguez-Navarro, J. & Susín Sánchez, A. (2006), Non structured meshes for cloth GPU simulation using FEM, *in* '3rd Workshop in Virtual Reality Interactions and Physical Simulation', Eurographics, pp. 1–7.

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

Sanfui, S. & Sharma, D. (2021), '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), 011005.

Sanfui, S. & Sharma, D. (2023), 'GPU-based mesh reduction strategy utilizing active nodes for structural topology optimization', *Structures* **55**, 570–586.

Schnös, F., Hartmann, D., Obst, B. & Glashagen, G. (2021), 'GPU accelerated voxel-based machining simulation', *The International Journal of Advanced Manufacturing Technology* **115**(1-2), 275–289.

Sefidgar, S. M. H., Firoozjaee, A. R. & Dehestani, M. (2021), 'Parallelization of torsion finite element code using compressed stiffness matrix algorithm', *Engineering with Computers* **37**(3), 2439–2455.

Simo, J. C. & Hughes, T. J. R. (1998), *Computational Inelasticity*, Springer-Verlag, New York.

Systèmes, D. (2017), 'ABAQUS 2017, Documentation', Dassault Systèmes, Rhode Island.

Tadmor, E. B., Miller, R. E. & Elliott, R. S. (2012), *Continuum Mechanics and Thermodynamics: From Fundamental Concepts to Governing Equations*, Cambridge University Press.

The MathWorks Inc. (2021), 'Matlab version R2021a', https://www.mathworks.com. Natick, Massachusetts.

Trilinos Project Team, T. (2020), 'The Trilinos Project Website', https://trilinos.github.io. acccessed March, 2022.

Träff, E. A., Rydahl, A., Karlsson, S., Sigmund, O. & Aage, N. (2023), 'Simple and efficient GPU accelerated topology optimisation: Codes and applications', *Computer Methods in Applied Mechanics and Engineering* **410**, 116043.

van Rietbergen, B., Weinans, H., Huiskes, R. & Polman, B. (1996), 'Computational strategies for iterative solutions of large FEM applications employing voxel data', *International Journal for Numerical Methods in Engineering* **39**(16), 2743–2767.

Vi, F., Mocellin, K., Digonnet, H., Perchat, E. & Fourment, L. (2018), 'Hybrid parallel multigrid preconditioner based on automatic mesh coarsening for 3D metal forming simulations', *International Journal for Numerical Methods in Engineering* **114**(6), 598–618.

Wong, J., Kuhl, E. & Darve, E. (2015), 'A new sparse matrix vector multiplication graphics processing unit algorithm designed for finite element problems', *International Journal for Numerical Methods in Engineering* **102**(12), 1784–1814.

Wu, W. & Heng, P. A. (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.

Wyser, E., Alkhimenkov, Y., Jaboyedoff, M. & Podladchikov, Y. Y. (2021), 'An explicit GPU-based material point method solver for elastoplastic problems (ep2-3de v1.0)', *Geoscientific Model Development* **14**(12), 7749–7774.

Xie, F.-z., Zhao, W.-w. & Wan, D.-c. (2020), 'CFD simulations of three-dimensional violent sloshing flows in tanks based on MPS and GPU', *Journal of Hydrodynamics* **32**(4), 672–683.

Yamaguchi, T., Fujita, K., Ichimura, T., Hori, M. & Maddegedara, L. (2019), 'Acceleration of unstructured implicit low-order finite-element earthquake simulation using OpenACC on pascal GPUs', *International Journal of High Performance Computing and Networking* **13**(1), 3–18.

Yusa, Y., Murakami, Y. & Okada, H. (2019), Large-scale parallel thermal elastic-plastic welding simulation using balancing domain decomposition method, *in* 'Pressure Vessels and Piping Conference', Vol. 58936, American Society of Mechanical Engineers, p. V002T02A006.

Yusa, Y., Okada, H., Yamada, T. & Yoshimura, S. (2018), 'Scalable parallel elastic – plastic finite element analysis using a quasi-Newton method with a balancing domain decomposition preconditioner', *Computational Mechanics* **62**(6), 1563–1581.

Yusa, Y. & Yoshimura, S. (2014), 'Speedup of elastic-plastic analysis of large-scale model with crack using partitioned coupling method with subcycling technique', *CMES-Computer Modeling in Engineering and Sciences* **99**(1), 87–104.

Zayer, R., Steinberger, M. & Seidel, H.-P. (2017), Sparse matrix assembly on the GPU through multiplication patterns, *in* 'High Performance Extreme Computing Conference (HPEC), 2017 IEEE', IEEE, pp. 1–8.

Zhang, J. & Shen, D. (2013), GPU-based implementation of finite element method for elasticity using CUDA, *in* 'High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on', pp. 1003–1008.