Universidade Estadual de Campinas
Faculdade de Engenharia Civil, Arquitetura e Urbanismo

Natália Ramos Vilas Boas

A cuda accelerated numerical integration of an elastoplastic problem with the finite element method

Aceleração da integração numérica de um problema elastoplástico pelo método dos elementos finitos com CUDA

CAMPINAS
2020

Natália Ramos Vilas Boas

A CUDA accelerated numerical integration of an elastoplastic problem with the finite element method

Aceleração da integração numérica de um problema elastoplástico pelo método dos elementos finitos com CUDA

Master thesis presented to the School of Civil Engineering, Architecture and Urban Design of the State University of Campinas to obtain the degree of Master in Civil Engineering in Structures and Geotechnics area.

Dissertação de Mestrado apresentada à Faculdade de Engenharia Civil, Arquitetura e Urbanismo da Universidade Estadual de Campinas para a obtenção do título de Mestra em Engenharia Civil na área de Estruturas e Geotécnica.

Orientador: Prof. Dr. Philippe Remy Bernard Devloo

Este exemplar corresponde à versão final da dissertação defendida pela aluna Natália Ramos Vilas Boas e orientada pelo Prof. Dr. Philippe Remy Bernard Devloo.

Assinatura do Orientador

____

Campinas
2020

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Área de Engenharia e Arquitetura
Luciana Pietrosanto Milla - CRB 8/8129

V71c

Vilas Boas, Natália Ramos, 1992-
A CUDA accelerated numerical integration of an elastoplastic problem with the Finite Element Method / Natália Ramos Vilas Boas. – Campinas, SP : [s.n.], 2020.

Orientador: Philippe Remy Bernard Devloo.
Dissertação (mestrado) – Universidade Estadual de Campinas, Faculdade de Engenharia Civil, Arquitetura e Urbanismo.

1. Método dos elementos finitos. 2. Integração numérica. 3. Elastoplasticidade. 4. CUDA (Arquitetura de computador). I. Devloo, Philippe Remy Bernard, 1958-. II. Universidade Estadual de Campinas. Faculdade de Engenharia Civil, Arquitetura e Urbanismo. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Aceleração da integração numérica de um problema elastoplástico pelo Método dos Elementos Finitos com CUDA
Palavras-chave em inglês:
Finite element method
Numerical integration
Elastoplasticity
CUDA (Computer architecture)
Área de concentração: Estruturas e Geotécnica
Titulação: Mestra em Engenharia Civil
Banca examinadora:
Philippe Remy Bernard Devloo [Orientador]
Luiz Carlos Marcos Vieira Junior
Edson Borin
Data de defesa: 27-01-2020
Programa de Pós-Graduação: Engenharia Civil

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0001-8231-8239
- Currículo Lattes do autor: http://lattes.cnpq.br/9304371137146814

UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA CIVIL, ARQUITETURA E URBANISMO

A CUDA accelerated numerical integration of an elastoplastic problem with the Finite Element Method

Natália Ramos Vilas Boas

Dissertação de Mestrado aprovada pela Banca Examinadora, constituída por:

Campinas, 27 de janeiro de 2020

Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisor, Prof. Philippe Devloo, for his assistance and engagement through the learning process of this master thesis. I give my special thanks to Dr. Omar Durán for his friendship and guidance in the development of this work. Also, I would like to acknowledge my friends at LabMeC for all the time we spent together. Moreover, thanks to Petrobras and FUNCAMP for the financial support.

My very profound gratitude to my family, especially to my parents Ivone and Luiz, for providing me with continuous support and encouragement throughout my years of study. Finally, I would like to thank Rafael for keeping me harmonious and helping me putting pieces together.

This accomplishment would not have been possible without them.

Thank you.

Abstract

Finite Element Method (FEM) is a numerical technique to approximate partial differential equations. It has been widely used to approximate solutions of physical problems in different fields of research. The numerical simulation of challenging engineering problems with small error requires refined meshes and leads to high computational cost. To overcome this difficulty, parallel computing is becoming a mainstream tool. Among the techniques available to improve the performance of this type of computational application is the execution of the algorithm using Graphics Processing Unit (GPU) programming. Although GPU was initially developed for graphics processing, it has been used in the last years as a general-purpose machine with high parallelism power through the availability of platforms such as CUDA or OpenGL. The purpose of this research is to develop an efficient algorithm for the evaluation of the Jacobian matrix and residual vector arising from a FEM analysis. This work aims the particular variational formulation of an elastoplastic problem with associative plasticity, but the adopted approach can be extended to other fields and problems. The presented strategy for the calculation of the Jacobian matrix and residual vector relies on several computational ingredients such as gathering and scattering operations, matrix multiplications, and a parallel coloring scheme for the assembly process. The verification of the nonlinear approximated solution includes a comparison with regular CPU implementation in terms of numerical results and execution efficiency. For the residual vector, the GPU outperforms the CPU and the classical assembly approach by a factor of up to 6 and 24 to cubic polynomial order approximations, respectively. For the Jacobian matrix, GPU's performance overcomes CPU's for linear polynomial order, being limited by the amount of shared memory for higher orders. Finally, a Modified Initial Stiffness method is applied to accelerate the convergence of the load step using the linear elasticity stiffness matrix to take advantage of the inexpensive residual numerical integration process to achieve convergence in a very efficient manner.

Keywords: Finite Element Method; Numerical Integration; Elastoplasticity; CUDA.

Resumo

O Método dos Elementos Finitos (MEF) é uma técnica numérica para aproximar equações diferenciais parciais. Esse método tem sido amplamente utilizado para aproximar soluções de problemas físicos em diferentes campos de pesquisa. A simulação numérica de problemas de engenharia com pequeno erro requer malhas refinadas levando a um alto custo computacional. Para superar essa dificuldade, a computação paralela está se tornando uma ferramenta convencional. Entre as técnicas disponíveis para melhorar o desempenho desse tipo de aplicação está a execução de algoritmos usando a programação na unidade de processamento gráfico (GPU). Embora a GPU tenha sido desenvolvida inicialmente para processamento gráfico, ela tem sido usada nos últimos anos como uma máquina de uso geral com alto poder de paralelismo através de plataformas como CUDA ou OpenGL. O objetivo desta pesquisa é desenvolver um algoritmo eficiente para a avaliação da matriz jacobiana e do vetor residual resultante de uma análise pelo MEF. Este trabalho visa a formulação variacional específica de um problema elastoplástico com plasticidade associativa, mas a abordagem adotada pode ser estendida a outros campos e problemas. A estratégia apresentada para o cálculo da matriz jacobiana e do vetor residual baseia-se em vários ingredientes computacionais, como operações de agrupamento e dispersão, multiplicações de matrizes e um esquema de coloração para facilitar o processo de montagem dos operadores em paralelo. A verificação da solução aproximada não linear inclui uma comparação com a implementação na CPU em termos de resultados numéricos e eficiência de execução. Para o vetor residual, a GPU supera a CPU e a abordagem de montagem clássica por um fator de até 6 e 24, respectivamente, para aproximações de ordem polinomial cúbica. Para a matriz jacobiana, o desempenho da GPU supera o da CPU para ordem polinomial linear, sendo limitado pela quantidade de memória compartilhada para ordens mais altas. Finalmente, o método *Initial Stiffness* é aplicado para acelerar a convergência usando a matriz de rigidez de elasticidade linear e se beneficiar do processo de integração numérica do vetor residual para obter a convergência de uma maneira eficiente.

Palavras-chave: Método dos Elementos Finitos; Integração Numérica; Elastoplasticidade; CUDA.

List of Figures

List of Tables

List of Symbols

$(\bullet)_e$	Element entity
E	Young modulus
J_k	Jacobian matrix at integration point $\vec{\xi}_k$
N	Number of elements
Ω	Euclidean domain
Φ	Yield criterion
Ψ	Plastic flow potential
$\bar{\mathbf{B}}$	Scattered strain-displacement operator
$\bar{\mathbf{W}}$	Scattered integration rule operator
$\boldsymbol{\sigma}$	Stress tensor
$\boldsymbol{\sigma}^{\,proj}$	Projected stress tensor
$\boldsymbol{\sigma}^{\,trial}$	Elastic trial stress tensor
$\boldsymbol{\varepsilon}$	Strain tensor
$\boldsymbol{\varepsilon}^e$	Elastic strain tensor
$\boldsymbol{\varepsilon}^p$	Plastic strain tensor
$\boldsymbol{\varepsilon}^{e\ trial}$	Elastic trial strain tensor
$\delta\vec{u}$	Increment solution
$\delta\boldsymbol{\alpha}$	Set of internal state variables
$\delta\gamma$	Plastic multiplier
$\hat{\mathbf{B}}$	Global strain–displacement operator

$\hat{\mathbf{D}}$	Constitutive operator
$\hat{\mathbf{W}}$	Integration rule operator
λ	First Lamé parameter
$\mathbb{C}$	Elastic constitutive tensor
$\mathbf{A}$	Set of thermodynamical forces
$\mathbf{H}$	Hardening modulus
$\mathbf{I}$	Identity tensor
$\mathbf{K}$	Jacobian matrix
$\mathbf{K}_l$	Linear Jacobian matrix
$\mathbf{K}_\sigma$	Volumetric Jacobian matrix
$\mathbf{N}$	Plastic flow vector
$\mathbf{R}$	Residual
$\mathbf{R}_l$	Linear residual
$\mathbf{R}_\sigma$	Volumetric residual
$\mathcal{E}$	Elastic domain
$\mathcal{N}$	Number of degrees of freedom
$\mathcal{T}_h$	Geometric partition
$\mathcal{Y}$	Yield surface
μ	Second Lamé parameter
ν	Poisson ratio
ω_k	Integration point weight
$\overline{\mathcal{E}}$	Set of plastically admissible stresses
$\partial\Omega$	Euclidean boundary
ϕ	Frictional angle
σ_0	Hydrostatic stress

$\mathbf{D}_{ep}$	Elastoplastic constitutive matrix
$\vec{\mathcal{C}}$	Connectivity vector
$\vec{\xi}_k$	Integration point
$\vec{n}$	Outward normal
$\vec{t}$	Normal traction
$\vec{u}$	Displacement vector
c	Cohesion
p	Polynomial order
p_{int}	Internal pressure
r_{ext}	External radius
r_{int}	Internal radius

Contents

Chapter 1

Introduction

1.1 Motivation

The Finite Element Method (FEM) is one of the most relevant numerical techniques to find approximate solutions of partial differential equations (PDEs). According to Becker, Carey, and Oden [2], this method defines a systematic way of constructing basis functions to approximate the solution of PDEs. The underlying idea is that these functions can be defined piecewise over subregions of the domain called finite elements. The polynomial order of the functions over each element can be arbitrary. Bhavikatti [4] states that although FEM has been initially developed for approximating problems of structural mechanics, it is now widely used as a technique for solving complex problems in different fields of engineering: civil, mechanical, nuclear, biomedical, geomechanics, and others. Many problems in these fields can lead to high computational demand.

Most computer codes are written to be executed sequentially: a problem is split into instructions, and these instructions are executed one after the other. Then, the performance improvement depends on the advance in CPU efficiency: the software can achieve a significant speedup as each new generation of processors is introduced. However, Kirk and Hwu [19] highlight that since 2003 a stagnation of performance improvement of general applications has been observed because high energy consumption and heat dissipation limit the increase of the clock frequency. Therefore, the industry offers a new approach: to increase the number of cores inside each processor.

This new approach has a significant impact on the software development community, including those that use the Finite Element Method. Hence, parallel computing in high-performance computers has gradually become a mainstream tool for dealing with large and detailed numerical problems in FEM analysis. Many parallel algorithms to find the approximate solution for problems using FEM were developed. However, they may require a large number of CPUs to achieve high performance.

Graphics Processing Units (GPUs) were initially developed for image and video processing. Due to the market demand for high-quality real-time graphics in computer applications, these processors have undergone considerable technological progress. For example, in an electronic gaming application, one needs to render scenes at a resolution of 60 frames per second. According to Micikevicius [24], a GPU consists of a set of multiprocessors where each multiprocessor has its own stream processors and shared memory. All multiprocessors of a GPU have access to the global memory, and memory latency is hidden if thousands of threads are executed concurrently. The main difference between GPUs and CPUs is that CPUs may be efficient with a small number of threads per core, whereas GPUs achieve higher performance when thousands of threads are executed concurrently.

Because of the technological advance of GPUs, researchers who wanted to improve the performance of their applications started to explore their use for non-graphical ones. This trend became known as General-Purpose computation on the GPU (GPGPU). Since then, GPUs have been used for numeric simulation of problems in fields of science and engineering. According to Zhang and Shen [39], methods that use GPU's powerful computing resource to accelerate the FEM analysis have naturally emerged in the last few years. Among the steps of the FEM calculations to approximate the solution of boundary value problems, the evaluation of the elements stiffness matrix and residual vector, as well as the assembly process of the linear system, are the most time-consuming processes in terms of both memory and runtime.

The research in this thesis presents a data structure and calculation strategies to compute the residual vector and the Jacobian matrix arising from an elastoplastic FEM simulation with GPU programming in order to improve the performance to obtain the approximate solution. The target problem in this research is to perform the geomechanics analysis in a wellbore region considering the elastoplastic constitutive modeling. Nevertheless, the proposed data structure and calculation strategies can be applied to other constitutive modelings and research fields.

1.2 Previous works on FEM with GPU

Zhang and Shen [39] implement a code to approximate elasticity problems in two and three dimensions using the FEM with GPU. The authors use a coloring method to perform the assembly of the global operators. The tests are conducted on a platform composed of an Intel Core 2 Duo E7400 processor and an NVIDIA Geforce GT 430. The authors reach a speedup of 7x for approximations in two space dimensions and 10x for three-dimensional elements. For the linear system solution, the authors present speedups of 3.5x and 6x for two and three-dimensional simulations, respectively.

Mafi [23] uses a GPU-based parallel computing approach to perform real-time analysis of soft objects deformation through a nonlinear approximation using the FEM. The author uses a coalesced data structure to compute the FEM matrices in GPU. The computation time for the matrices evaluation reaches a speedup of 28x in an NVIDIA Geforce GTX 470 when compared to a sequential CPU implementation with an Intel Core i7-3770 processor.

Cecka, Lew, and Darve [8] introduce multiple strategies for evaluating the FEM global operators assembly. The authors present how to use global, shared, and local memory properly according to the polynomial order of the finite element discretization. The experimental setup consists of an NVIDIA GeForce 8800 GTX and an Intel Core 2 Quad CPU Q9450 processor. The assembly process reaches a speedup of 35x to the double-precision single-core CPU version for linear and quadratic polynomial orders.

Macioł, Płaszewski, and Banaś [22] use GPU programming to accelerate the numerical integration of the elements contributions from Laplace's equation in a 3D domain discretized in prismatic elements. Due to small resources available for a single thread for GPU architectures, the authors propose a GPU implementation of numerical integration based on the assumption that a single finite element corresponds to a single thread block, and these individual threads calculate sets of an element contribution. The platform test consists of an NVIDIA GeForce 8800 GTX and an AMD X2 processor. The speedup varies from 3.5x to 20x, depending on the approximation order.

Dziekonski *et al.* [12] implement a technique to generate the operators arising from computational electromagnetics analysis through the Finite Element Method using GPU programming. The results are obtained from tests conducted in an NVIDIA Tesla C2075 and an Opteron 6174. The authors present a series of optimizations to perform the numerical integration, such as the use of shared memory to avoid time-consuming writing and reading to and from the global memory. The authors reach speedups of 81x and 19x over an optimized single and multi-threaded CPU-only implementations, respectively.

1.3 Objectives

The main objective of the research presented in this thesis is to approximate the solution of an elastoplastic problem with the FEM using GPU programming. To accomplish the main objective, four specific objectives of this research are established:

- To understand the fundamental concepts of the elastoplastic constitutive modeling;
- To construct a data structure resulting from the Finite Element Method that allows parallel operations;
- To understand the principal concepts of GPU programming;

- To develop a numerical implementation of the global operators arising from an elastoplastic finite element problem to be executed in parallel in the GPU.

The proposed strategy for computing the residual vector and the Jacobian matrix presented in this research relies on pre-computing and storing constant data along a nonlinear FEM analysis in an aligned data structure and perform GPU parallelized operations to evaluate the global operators. A finite element coloring scheme is applied to ensure correct parallel execution of the proposed implementation since the construction of the global operations involves overlapping information corresponding to the common degrees of freedom of the system. This strategy is applied to the simulation of an elasto-plastic problem through a FEM simulation of a wellbore under internal and external stresses, as well as an initial stress presented by a hydrostatic pre-stress.

This study focuses on accelerating the numerical integration of an elastoplastic FEM simulation on the GPU. Therefore an analogous CPU implementation is developed to compare both CPU and GPU's performances. The GPU's performance is also compared with the classical assembly process of a traditional FEM simulation. Besides verifying that the results obtained by the GPU implementation are identical to the results of the CPU implementation, the accuracy of the nonlinear approximation is verified by comparing the results with an approximation obtained using a Runge-Kutta approximation on a very refined mesh.

1.4 Outline

The present work is organized as follows:

1.4.1 Body

Chapter 2 introduces the elastoplastic constitutive modeling and its main items. Also, it presents the return mapping scheme for perfect plasticity and the Mohr-Coulomb yield criterion. In Chapter 3, the Finite Element Method and the weak formulation of the elastoplastic problem are presented. Chapter 4 explores GPU programming presenting the architecture of a GPU, the CUDA programming model, and GPU memory types. In Chapter 5 is presented the structure arising from the finite element problem for the elastoplastic problem. Chapter 6 describes the main aspects of the computational implementation as well as the approach to verify the accuracy of the results obtained from the proposed structure. Also, this chapter describes a Quasi-Newton method for accelerating the convergence. Chapter 7 is dedicated to the presentation of the results and discussion. Finally, Chapter 8 presents the conclusions.

1.4.2 Appendix

Appendix A presents the Voigt notation for the stress and strain tensors. In Appendix B, the scheme for the spectral decomposition of tensors is described. Appendix C shows tables with the mean and standard deviation of the presented results.

Chapter 2

Elastoplastic constitutive modeling

A body that undergoes elastic strains is characterized by the complete recovery to its undeformed configuration upon removal of the applied loads. Moreover, this type of strain depends only on the load applied to the body. On the other hand, irreversible strains in a body subjected to a loading cycle are known as *plastic strains.* These strains occur when a body is subjected to stress intensities above a limit value known as the *elastic limit.* In this case, the total strain is formally split into elastic and plastic strains.

The concepts of plasticity are illustrated in Fig. 2.1:

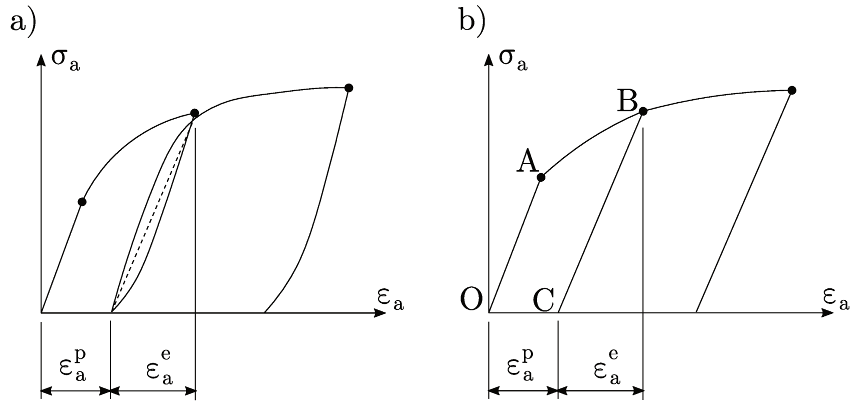

Figure 2.1: Stress-strain relationship. Extracted from Santos [30].

Figure 2.1a) presents the stress-strain curve of a uniaxial tension test of a material. Figure 2.1b) shows the abstraction of the behavior of the material, which is easy to identify the elastic and plastic portions of strain. When a bar is subjected to a loading presented by the straight line $\overline{OA}$, the unloading follows $\overline{AO}$, and there is no residual strain. ${\varepsilon_a}^e$ corresponds to the *elastic strain* and is totally recovered upon the unloading. Thus, the behavior of the material is regarded as *linear elastic* over $\overline{OA}$. If this material is subjected to a loading presented by $\overline{OAB}$, the unloading follows $\overline{BC}$, and it is observed a residual strain ${\varepsilon_a}^p$, which is called *plastic strain.*

A large number of engineering materials, such as metals, concrete, rocks, clays, and soils in general, may be modeled as plastic under a wide range of circumstances of practical interest. An incremental stress-strain relation describes the study of these materials. According to Souza Neto, Peric, and Owen [34], the basic items of the elastoplastic constitutive modeling are:

- Strain tensor decomposition;
- Elastic constitutive law;
- Yield criterion;
- Plastic flow rule;
- Hardening law.

2.1 Strain tensor decomposition

This topic consists of the decomposition of the total strain tensor ($\boldsymbol{\varepsilon}$) into elastic and plastic strains, which are related to the applied loads and the history of irreversible processes applied to the material, respectively. For the elastoplastic calculation, the strain given as input data is presented as total strain, and from the solution of the problem, the elastic ($\boldsymbol{\varepsilon}^e$) and plastic ($\boldsymbol{\varepsilon}^p$) strains are computed.

$$\boldsymbol{\varepsilon} = \boldsymbol{\varepsilon}^e + \boldsymbol{\varepsilon}^p \tag{2.1}$$

The corresponding rate form of the additive split reads:

$$\delta\boldsymbol{\varepsilon} = \delta\boldsymbol{\varepsilon}^e + \delta\boldsymbol{\varepsilon}^p$$

2.2 Elastic constitutive law

The generalized Hooke's Law is presented by:

$$\boldsymbol{\sigma} = \mathbb{C}\boldsymbol{\varepsilon}^e$$

where $\mathbb{C}$ is the fourth-order elastic constitutive tensor. In Voigt notation (see Appendix A), it is presented by:

$$\begin{pmatrix}\sigma_{xx}\\ \sigma_{yy}\\ \sigma_{zz}\\ \sigma_{xy}\\ \sigma_{yz}\\ \sigma_{zx}\end{pmatrix} = \begin{bmatrix}2\mu+\lambda & \lambda & \lambda & 0 & 0 & 0\\ \lambda & 2\mu+\lambda & \lambda & 0 & 0 & 0\\ \lambda & \lambda & 2\mu+\lambda & 0 & 0 & 0\\ 0 & 0 & 0 & 2\mu & 0 & 0\\ 0 & 0 & 0 & 0 & 2\mu & 0\\ 0 & 0 & 0 & 0 & 0 & 2\mu\end{bmatrix}\begin{pmatrix}\varepsilon^e_{xx}\\ \varepsilon^e_{yy}\\ \varepsilon^e_{zz}\\ 2\varepsilon^e_{xy}\\ 2\varepsilon^e_{yz}\\ 2\varepsilon^e_{zx}\end{pmatrix}$$

Also, the stress-strain relationship is presented as:

$$\boldsymbol{\sigma} = 2\mu\boldsymbol{\varepsilon}^e + \lambda\, tr(\boldsymbol{\varepsilon}^e)\mathbf{I} \tag{2.2}$$

where λ and μ are the first and second Lamé constants and are given as a function of the Young modulus[1] (E) and the Poisson ratio[2] (ν).

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}$$

$$\mu = \frac{E}{2(1+\nu)}$$

2.3 Yield criterion

In a uniaxial tension test, a material undergoes plastic strains when it is subjected to a specific stress limit. This principle can be extended to the three-dimensional case by stating a yield function. The yield criterion corresponds to the transition between the elastic and plastic regimes. It can be expressed by a function that is negative when only elastic strains are possible and zero when a plastic flow is imminent. The function that describes the yield criterion is given by:

$$\Phi = \Phi\left(\boldsymbol{\sigma}, \mathbf{A}\right) \tag{2.3}$$

The yield function defines the *elastic domain* as the set:

$$\mathcal{E} = \{\boldsymbol{\sigma} \,|\, \Phi\left(\boldsymbol{\sigma}, \mathbf{A}\right) < 0\}$$

of stresses for which plastic yielding is not possible. Any stress lying in the elastic domain

[1]Slope of the stress-strain curve on the elastic portion and describes the elastic properties of a solid undergoing tension or compression in one direction.

[2]Ratio between the lateral strain normal to the applied load and the axial strain in the direction of the applied load.

or on its boundary is said to be *plastically admissible*. The set of plastically admissible stresses are defined as:

$$\overline{\mathcal{E}} = \{\boldsymbol{\sigma} \,|\, \Phi\left(\boldsymbol{\sigma}, \mathbf{A}\right) \leq 0\}$$

The set of stresses for which plastic yielding may occur is called *yield locus* and corresponds to the boundary of the elastic domain, where $\Phi\left(\boldsymbol{\sigma}, \mathbf{A}\right) = 0$. The yield locus is expressed by a hypersurface in the space of stresses which is called *yield surface* and is defined as:

$$\mathcal{Y} = \{\boldsymbol{\sigma} \,|\, \Phi\left(\boldsymbol{\sigma}, \mathbf{A}\right) = 0\}$$

2.4 Plastic flow rule

The plastic flow rule assumes the existence of a plastic potential function that characterizes the tensile behavior of plastic strains in a yielding process. This potential is defined by:

$$\Psi = \Psi(\boldsymbol{\sigma}, \mathbf{A}) \tag{2.4}$$

from which the flow vector is obtained as:

$$\mathbf{N}(\boldsymbol{\sigma}, \mathbf{A}) = \frac{\partial \Psi}{\partial \boldsymbol{\sigma}}$$

Thus, the behavior of the plastic strain tensor is given by:

$$\delta \boldsymbol{\varepsilon}^p = \delta\gamma \mathbf{N}(\boldsymbol{\sigma}, \mathbf{A})$$

where $\delta\gamma \geq 0$ is a plastic multiplier.

2.5 Hardening law

The yielding process may lead to changes in size, shape, and direction of the yield surface. The hardening law determines how these modifications happen. This phenomenon can be considered isotropic or kinematic, and if there is no hardening phenomenon, the model is considered perfectly plastic. Isotropic hardening plasticity models show a uniform expansion of the initial flow surface without translation (Fig. 2.2). Whereas in kinematic hardening plasticity models, there is a translation of the surface, and its size remains constant (Fig. 2.3). In a perfectly plastic model, there is no hardening phenomenon. That is, the yield stress level does not depend in any way on the degree of plastification (Fig. 2.4).

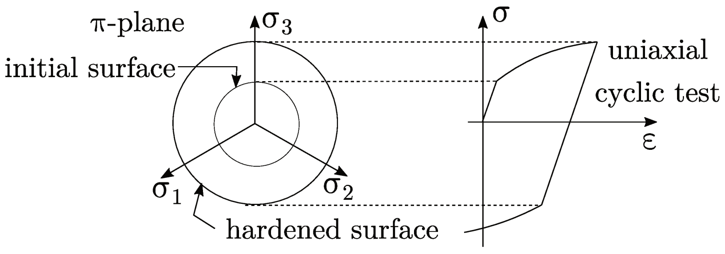

Figure 2.2: Isotropic hardening. Extracted from Souza Neto, Peric, and Owen [34].

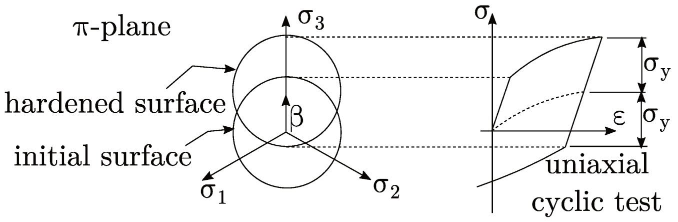

Figure 2.3: Kinematic hardening. Extracted from Souza Neto, Peric, and Owen [34].

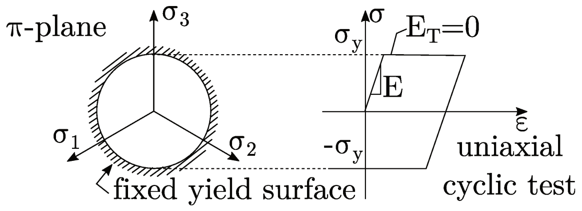

Figure 2.4: Perfect plasticity. Extracted from Souza Neto, Peric, and Owen [34].

The evolution of the internal variables associated with hardening phenomenon is given by:

$$\delta\boldsymbol{\alpha} = \delta\gamma\mathbf{H}(\boldsymbol{\sigma}, \mathbf{A})$$

where:

$$\mathbf{H}(\boldsymbol{\sigma}, \mathbf{A}) = -\frac{\partial\Psi}{\partial\mathbf{A}} \tag{2.5}$$

This work considers the perfect plasticity case. Therefore there is no hardening phenomenon ($\mathbf{A} = 0$).

2.6 Loading/unloading criterion

The loading/unloading criterion is determined by:

$$\Phi \leq 0; \qquad \delta\gamma \geq 0; \qquad \delta\gamma\Phi = 0 \tag{2.6}$$

The first statement specifies the plastic admissible region. The second assigns that the plastic multiplier must be greater or equal to zero. Finally, the last statement refers to the fact that the plastic multiplier and the yield function cannot be non-null together. If the material is in the elastic regime ($\Phi < 0$), the plastic multiplier will be zero ($\delta\gamma = 0$). However, in the plastic regime the yield function is zero ($\Phi = 0$) and the plastic multiplier is positive ($\delta\gamma > 0$) [32].

Table 2.1 summarizes the principal items of the elastoplastic constitutive model above described for the perfect plasticity case.

Table 2.1: Summary of the elastoplastic constitutive model for perfect plasticity.

Elastoplastic constitutive model for perfect plasticity	
Strain tensor decomposition	$\delta\boldsymbol{\varepsilon} = \delta\boldsymbol{\varepsilon}^e + \delta\boldsymbol{\varepsilon}^p$
Elastic constitutive law	$\boldsymbol{\sigma} = 2\mu\boldsymbol{\varepsilon}^e + \lambda\, tr(\boldsymbol{\varepsilon}^e)\mathbf{I}$
Yield criterion	$\Phi = \Phi\left(\boldsymbol{\sigma}\right)$
Plastic flow rule	$\delta\boldsymbol{\varepsilon}^p = \delta\gamma\mathbf{N}(\boldsymbol{\sigma})$
Loading/unloading criterion	$\Phi \leq 0; \quad \delta\gamma \geq 0; \quad \delta\gamma\Phi = 0$

2.7 Return mapping scheme

Considering the pseudo-time t_n the strain tensor is $\boldsymbol{\varepsilon}_n$ and the corresponding plastic portion is $\boldsymbol{\varepsilon}^p_n$. The return mapping scheme consists of two conditions that are imposed to compute $\boldsymbol{\varepsilon}^p_{n+1}$ and $\delta\gamma$: an elastic predictor and a plastic corrector. The return mapping scheme is presented by Souza Neto, Peric, and Owen [34] and is illustrated in Fig. 2.5.

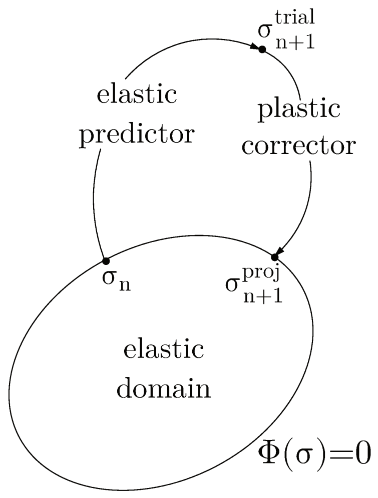

Figure 2.5: Return mapping scheme for perfect plasticity. Adapted from Souza Neto, Peric, and Owen [34].

Elastic predictor

The elastic predictor step assumes that $\delta\gamma = 0$, that is, the step $[t_n, t_{n+1}]$ is elastic. The elastic trial state is:

$$\boldsymbol{\varepsilon}_{n+1}^{e\ trial} = \boldsymbol{\varepsilon}_n^e + \delta\boldsymbol{\varepsilon}$$

$$\boldsymbol{\sigma}_{n+1}^{trial} = 2\mu\boldsymbol{\varepsilon}_{n+1}^{e\ trial} + \lambda\, tr(\boldsymbol{\varepsilon}_{n+1}^{e\ trial})\boldsymbol{I}$$

The elastic trial state occurs if $\Phi(\boldsymbol{\sigma}_{n+1}^{trial}) \leq 0$ is satisfied. Otherwise, the elastic trial state is not plastically admissible and a solution to the problem must be obtained from the plastic corrector.

Plastic corrector

The plastic corrector step uses the elastic trial state to solve the system:

$$\boldsymbol{\varepsilon}_{n+1}^{e} = \boldsymbol{\sigma}_{n+1}^{trial} + \delta\gamma\boldsymbol{N}(\boldsymbol{\sigma})$$

$$\Phi(\boldsymbol{\sigma}_{n+1}^{proj}) = 0$$

where $\boldsymbol{\sigma}_{n+1}^{proj}$ is obtained applying the stress-strain relationship presented in Eq. (2.2) where the corresponding elastic strain is $\boldsymbol{\varepsilon}_{n+1}^{e}$.

2.8 Mohr-Coulomb Yield Criterion

The Mohr-Coulomb yield criterion applies to the modeling of materials such as concrete or soil. The behavior of these materials is generally characterized by a strong dependence of the yield limit on the hydrostatic pressure. This criterion is based on the assumption that the phenomenon of macroscopic plastic yielding depends on the frictional sliding between material particles. The Mohr-Coulomb yield criterion states that plastic yielding happens when the shear stress and normal stress reach the given relationship:

$$\tau = c - \sigma \tan\phi \tag{2.7}$$

from which c corresponds to the material cohesion and ϕ is the material frictional angle.

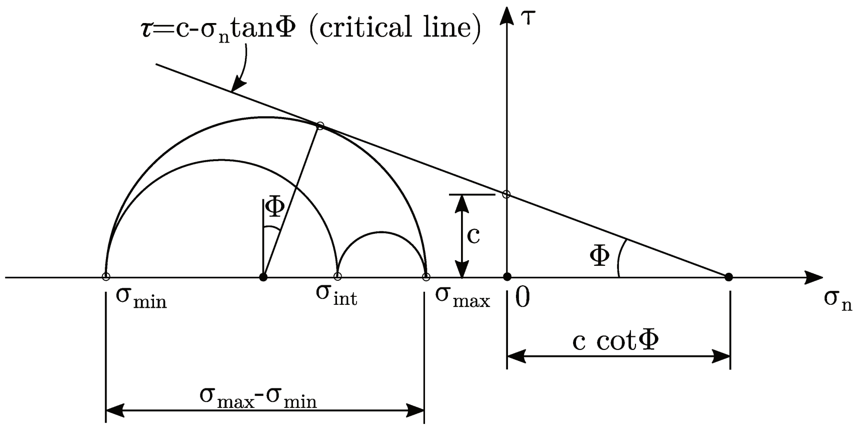

Figure 2.6: Mohr plane representation. Extracted from Souza Neto, Peric, and Owen [34].

The yield locus of the Mohr-Coulomb yield criterion is the set of all stress states such that there exists a plane in which Eq. (2.7) holds, i.e., it is the set of all stresses whose largest Mohr circle is tangent to the critical line in Fig. 2.6. The elastic domain for the Mohr-Coulomb yield criterion corresponds to the set of stresses whose all three Mohr circles are below the critical line.

In principal stresses space, Mohr-Coulomb's surface consists of six surfaces described by the following equations and is graphically presented in Fig. 2.7:

$$\begin{aligned}
\Phi_1 &= (\sigma_1 - \sigma_3) + (\sigma_1 + \sigma_3)\sin\phi - 2c\cos\phi \\
\Phi_2 &= (\sigma_2 - \sigma_3) + (\sigma_2 + \sigma_3)\sin\phi - 2c\cos\phi \\
\Phi_3 &= (\sigma_2 - \sigma_1) + (\sigma_2 + \sigma_1)\sin\phi - 2c\cos\phi \\
\Phi_4 &= (\sigma_3 - \sigma_1) + (\sigma_3 + \sigma_1)\sin\phi - 2c\cos\phi \\
\Phi_5 &= (\sigma_3 - \sigma_2) + (\sigma_3 + \sigma_2)\sin\phi - 2c\cos\phi \\
\Phi_6 &= (\sigma_1 - \sigma_2) + (\sigma_1 + \sigma_2)\sin\phi - 2c\cos\phi
\end{aligned}$$

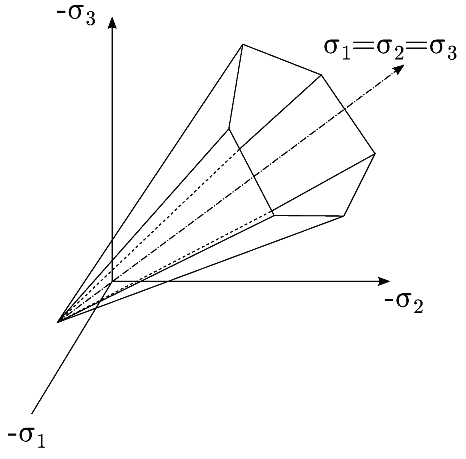

Figure 2.7: Mohr-Coulomb surface. Adapted from Souza Neto, Peric, and Owen [34].

Chapter 3

The Finite Element Method

Finite Element Method (FEM) is a technique to construct an approximate solution of partial differential equations (PDEs). The main idea of this method is to represent the problem domain as a finite number of elements and solve not the original problem, but its weak formulation. To obtain the weak formulation of a differential equation, one must replace it with an integral equation, using piecewise integration to reduce the order of the derivatives and multiply it by a test function. Three properties of the weak formulation are worth noting: the classic formulation is also a weak formulation, a weak formulation is indeed a classic formulation as long as it is regular enough, and the solution of the problem is the only solution of the weak formulation.

The Galerkin method is one approach to construct approximate solutions to boundary-value problems. This method consists in seeking an approximate solution for the weak formulation in a finite-dimensional subspace of the admissible functions space of the problem. However, Galerkin method does not provide a systematic way of constructing the basis functions for the approximate test functions. For this reason, the classical Galerkin method as described provides several possibilities, which may lead or not to reasonable approximations. Therefore, the concept of the Finite Element Method is introduced to overcome these difficulties.

The Finite Element Method provides a general and systematic technique for constructing the basis functions for Galerkin approximations of boundary-value problems. The idea is to define the basis functions piecewise over the finite elements from the discretization of the domain. Moreover, these functions can be chosen to be straightforward functions such as polynomials of low degree. Also, they are chosen in such a way that the multiplier coefficients defining the approximate solution are precisely the values of the approximate solution at the nodal points.

Having selected an appropriate set of basis functions, it is possible to calculate the operators per element. Finally, each element contribution is appropriately added to form the global approximation of the problem. This step is called assembly and consists in gathering the contributions from each element according to the connectivity of the system. This strongly characterizes the Finite Element Method, since the computations are performed per element and then incorporated into the global system of the problem [2].

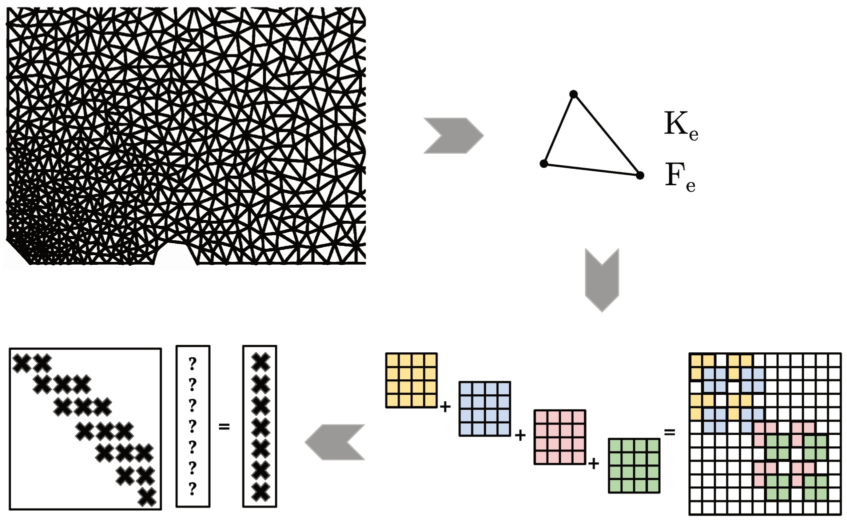

Figure 3.1: General steps in the finite element method.

The following step in FEM consists in applying the boundary constraints to the global matrix form followed by solving the system of equations using a proper numerical method. Finally, it is possible to obtain the solution post-processing and other quantities of interest, such as reproduce results in tables or graphics. Figure 3.1 illustrates the main steps in the Finite Element Method.

3.1 Finite Element basis functions

The quality of an approximate solution from a finite element problem strongly depends on the choice of the basis functions. If they are correctly selected, the determination of the coefficients is reduced to a computational problem [1]. For unidimensional domains, it is common to use linear functions presented in Fig. 3.2:

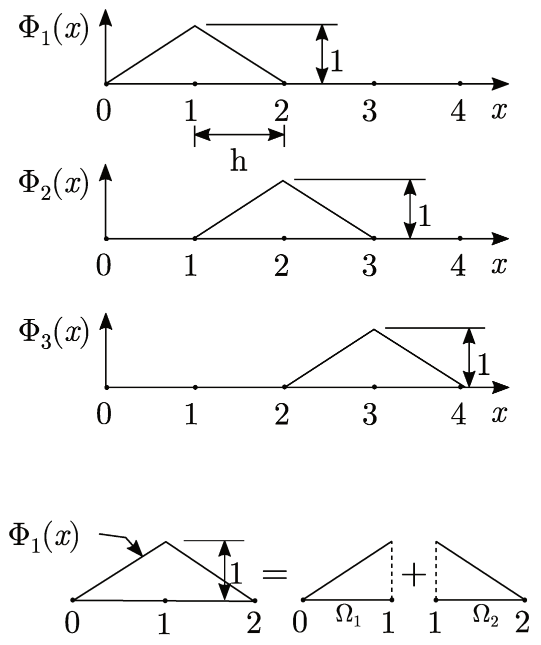

Figure 3.2: Linear basis functions for linear elements. Extracted from Becker, Carey, and Oden [2].

In the computational environment used in this research, the finite element approximation spaces are constructed using hierarchical base functions [11]. Figure 3.3 illustrates the quadratic base functions for quadrilateral elements.

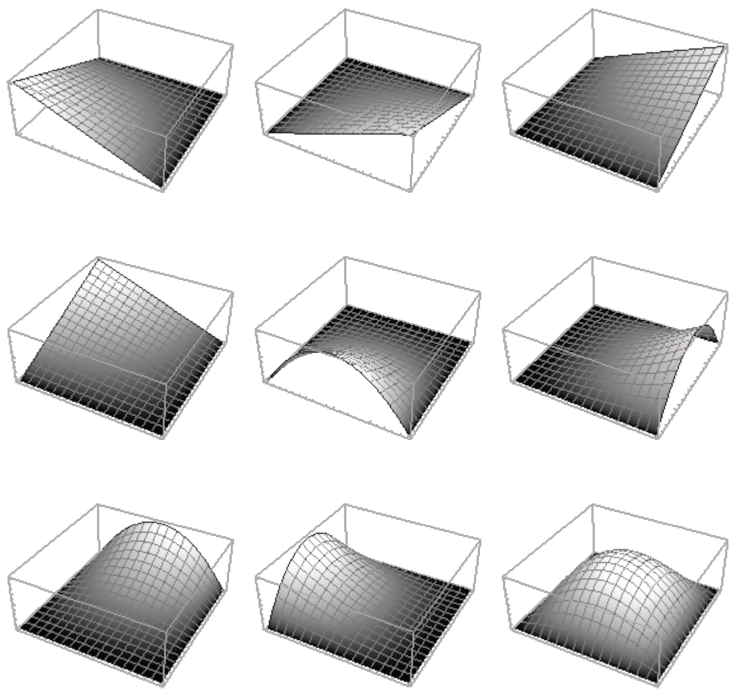

Figure 3.3: Quadratic basis functions for quadrilateral elements.

3.2 Finite Element transformations

The finite element transformations are used for mapping the coordinates of a curvilinear element Ω_e to the master element $\hat{\Omega}$. This approach allows the calculations of the elements to be performed in terms of $\hat{\Omega}$. Consider the quadrilateral element $\hat{\Omega}$ with coordinates ξ and η where $-1 \leq \xi, \eta \leq 1$ in Fig. 3.4. The transformation from $\hat{\Omega}$ onto Ω_e is presented by:

$$T_e : \begin{cases} x = x(\xi, \eta) \\ y = y(\xi, \eta) \end{cases}$$

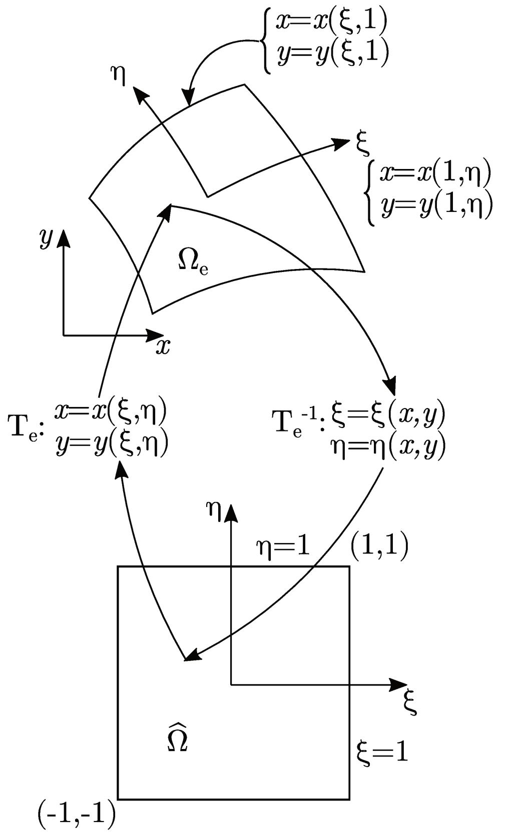

Figure 3.4: A finite element Ω_e in the x, y-plane obtained as the image under T_e of the corresponding master element $\hat{\Omega}$ in the ξ, η-plane. Extracted from Becker, Carey, and Oden [2].

Suppose the functions x and y are continuously differentiable with respect to ξ and η. Thus, $d\xi$ and $d\eta$ transform into dx and dy as:

$$dx = \frac{\partial x}{\partial \xi} d\xi + \frac{\partial x}{\partial \eta} d\eta$$
$$dy = \frac{\partial y}{\partial \xi} d\xi + \frac{\partial y}{\partial \eta} d\eta$$

In matrix form:

$$\begin{bmatrix} dx \\ dy \end{bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} \end{bmatrix} \begin{bmatrix} d\xi \\ d\eta \end{bmatrix} \tag{3.1}$$

where the 2×2 matrix of partial derivatives in Eq. (3.1) is the Jacobian of the transformation denoted $\mathbf{J}$.

3.3 Elastoplastic modeling in FEM context

The analytical solution of engineering problems governed by the elastoplastic constitutive model can only be evaluated under the assumption of very simplified conditions. The adoption of appropriate numerical techniques capable of producing approximate solutions with reasonable accuracy is required to provide more realistic analysis. In the context of FEM, the formulation and numerical solution of nonlinear problems in continuum mechanics are based on the weak formulation of the principle of virtual works. For materials governed by the elasticity law, the solution is directly obtained from the stress tensor that can be introduced into the equation of the principle of virtual work to obtain a variational equation with only the displacements and its gradient. However, the plastic behavior consists of an iterative process that necessarily has to be evaluated by the evolution of the constitutive equations in a pseudo-time (see Fig. 3.5).

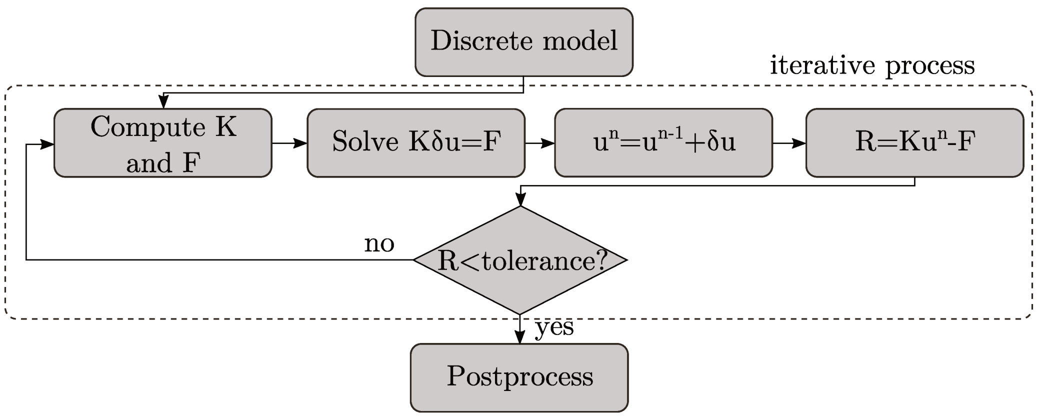

Figure 3.5: Finite element method flowchart for the iterative process.

3.3.1 Weak formulation of the elastoplastic problem

Denoting Ω as the domain for the PDE problem in $\mathbb{R}^2$ with boundaries $\partial\Omega = \partial\Omega_\mathrm{D} \cup \partial\Omega_\mathrm{N}$ where D and N stand for the boundary with Dirichlet and Neumann data, respectively. The governing equations for the elastoplastic strain consist of three parts: a conservation law, a constitutive equation, and boundary conditions.

$$\begin{aligned} \operatorname{div}(\boldsymbol{\sigma}(x)) &= 0 \ \ x \in \Omega && (3.2) \\ \vec{u}(s) &= \vec{u}_D(s) \ \ s \in \ \partial\Omega_D && (3.3) \\ \boldsymbol{\sigma}(s) \cdot \vec{n} &= \vec{t}(s) \ s \in \ \partial\Omega_N && (3.4) \end{aligned}$$

where $\boldsymbol{\sigma}$ is the stress tensor, $\vec{u}$ represents the displacement vector, $\vec{t}$ is the normal traction over $\partial\Omega_N$ and $\vec{n}$ is the outward normal. Equation (3.2) corresponds to the conservation law and Eqs. (3.3) and (3.4) are the boundary conditions.

Following the elastoplastic constitutive modeling presented in Chapter 2, the strain tensor $\boldsymbol{\varepsilon}$ is decomposed into elastic strain $\boldsymbol{\varepsilon}^e$ and plastic strain $\boldsymbol{\varepsilon}^p$. The stress tensor $\boldsymbol{\sigma}$ is a function of the elastic part of the strain tensor.

$$\boldsymbol{\sigma}(\vec{u}) = 2\mu\boldsymbol{\varepsilon}^e + \lambda\,\mathrm{tr}(\boldsymbol{\varepsilon}^e)\mathbf{I} \ \text{ in } \ \Omega \tag{3.5}$$

Under the assumption of small strains, the total strain tensor is expressed as:

$$\boldsymbol{\varepsilon}(\vec{u}) = \frac{1}{2}\left(\nabla\vec{u} + \nabla^t\vec{u}\right) \tag{3.6}$$

Consider a geometrical partition $\mathcal{T}_h = \{\Omega_e\}$ of the region Ω by convex elements Ω_e with boundaries $\partial\Omega_e$. The index h stands for the maximum diameter of the elements Ω_e. The following functional space is required:

$$\mathrm{H}^1(\Omega) = \left\{\vec{v} \in L^2(\Omega) : \nabla\vec{v} \in \mathrm{L}^2(\Omega)\right\} \tag{3.7}$$

The classical one-field weak formulation for the mechanical problem defined in Eq. (3.5) is formulated as:

Find $\vec{u} \in \mathbf{V} = \left\{\vec{v} \in \mathrm{H}^1(\Omega),\ \vec{v}\,|_{\partial\Omega_D} = 0\right\}$ such that:

$$\mathrm{R}(\vec{u},\vec{v}) = \int_\Omega \mathrm{div}(\boldsymbol{\sigma})\ \vec{v}\ \mathrm{dV} = 0 \tag{3.8}$$

The divergence theorem applied to Eq. (3.8) leads to:

$$\mathrm{R}(\vec{u},\vec{v}) = \int_\Omega \boldsymbol{\sigma} : \nabla\vec{v}\ \mathrm{dV} - \int_{\partial\Omega_N} \vec{t}\cdot\vec{v}\ \mathrm{dS} \tag{3.9}$$

For the elastoplastic case, in Eq. (3.9) the stress depends on a nonlinear response of the strain. Therefore, it must be linearized with respect to $\vec{u}$ at the point $\vec{u}^*$. The linearized problem is finding $\delta\vec{u}$ such that:

$$\mathrm{L}(\delta\vec{u},\vec{v}) = \mathrm{R}(\vec{u}^*,\vec{v}) + \mathrm{D}\ \mathrm{R}(\vec{u}^*,\vec{v})[\delta\vec{u}] = 0$$

where L is the linearization of $\mathrm{R}(\vec{u}^*,\vec{v})$ and

$$\mathrm{D}\ \mathrm{R}(\vec{u}^*,\vec{v})[\delta\vec{u}] = \frac{d}{d\epsilon}|_{\epsilon=0}\mathrm{R}_u(\vec{u}^* + \epsilon\delta\vec{u},\vec{v})$$

is the directional derivative of R_u on $\vec{u}^*$ in the direction of $\delta\vec{u}$.

$$\begin{aligned}\mathrm{D}\ \mathrm{R}(\vec{u}^*,\vec{v})[\delta\vec{u}] = \frac{d}{d\epsilon}|_{\epsilon=0}\int_\Omega \boldsymbol{\sigma}\left(\boldsymbol{\varepsilon}(\epsilon)\right) : \nabla\vec{v}\ \mathrm{dV} - \int_{\partial\Omega_N}\vec{t}\cdot\vec{v}\ \mathrm{dS} =\\ \frac{d}{d\epsilon}|_{\epsilon=0}\int_\Omega \boldsymbol{\sigma}\left(\boldsymbol{\varepsilon}(\epsilon)\right) : \nabla\vec{v}\ \mathrm{dV}\end{aligned}$$

where:

$$\boldsymbol{\varepsilon}(\epsilon) = \frac{\nabla(\vec{u}^* + \epsilon\delta\vec{u}) + \nabla(\vec{u}^* + \epsilon\delta\vec{u})^T}{2} = \nabla^s\vec{u}^* + \epsilon\nabla^s\delta\vec{u}$$

Applying the chain rule:

$$\mathrm{D}\ \mathrm{G}(\vec{u}^*,\vec{v})[\delta\vec{u}] = \int_\Omega \mathbf{D}_{ep} : \nabla^s\delta\vec{u} : \nabla\vec{v}\ \mathrm{dV}$$

where:

$$\mathbf{D}_{ep} = \frac{\partial\boldsymbol{\sigma}}{\partial\boldsymbol{\varepsilon}}$$

is the derivative of the stress tensor with respect to the strain.

Thus, the mechanical equilibrium of a body with material governed by the elastoplastic constitutive law must satisfy the following:

Find $\vec{u}\in\mathbf{V}$ such that:

$$\int_\Omega \mathbf{D}_{ep} : \nabla^s\delta\vec{u} : \nabla\vec{v}\ \mathrm{dV} = -\int_\Omega \boldsymbol{\sigma} : \nabla\vec{v}\ \mathrm{dV} + \int_{\partial\Omega_N}\vec{t}\cdot\vec{v}\ \mathrm{dS} \tag{3.10}$$

Chapter 4

GPU programming

Traditionally, computer codes were written to be executed sequentially so that a problem was partitioned into instructions and these instructions were executed one after another. However, more complex problems have emerged from technological progress in many industry's fields. The growing market and increasing demand to solve these problems more efficiently have driven the manufacturing of faster and smarter processors. For many years, the strategy for increasing the performance of the processors adopted by the industry was to develop processors with higher clock frequencies, which increased each year significantly [19].

The processor clock coordinates all Central Process Unit (CPU) and memory operations by generating a time reference signal called a *clock cycle* or *tick*. *Clock rate* is the frequency at which the clock circuit of a processor can generate pulses, which are used to synchronize the operations of its components [37]. The frequency is specified in megahertz (MHz), which corresponds to millions of clock cycles per second, or gigahertz (GHz), which specifies billions of clock cycles per second. Finally, *clock speed* determines how fast instructions are executed. Some instructions demand one clock cycle, others multiple clock cycles, and some processors execute multiple instructions during one clock cycle [37].

For more than two decades, microprocessors based on a single CPU provided fast performance increases and cost reductions in computational applications. They could achieve billions of floating-point[1] operations per second (gigaFLOPS or GFLOPS) to the desktop and trillion of floating-point operations per second (teraFLOPS or TFLOPS) to cluster servers. Thus, it was possible to deliver more functionality to applications, better user interfaces, and more useful results.

However, after some years, the performance improvement due to the manufacturing of more powerful processors experienced a stagnation. Processors' performance increased 60 percent per year in the 1990s, but slowed to 40 percent per year from 2000

[1]Method of encoding real numbers within the limits of finite precision available on computers [13].

to 2004, when performance increased by only 20 percent [15]. Faster processors heated up faster than the average fan could cool them down. Moreover, they required high energy consumption to complete their tasks [17]. These two main reasons limited the manufacturers to produce faster processors. Thus, the industry had to give up their efforts to increase clock frequency by looking for a new method: increasing the number of cores within the processor [9].

Multi-core processors consist of two or more cores or processing units that operate in parallel to read and execute instructions. The main idea of this technology is to use multiple cores instead of one at a comparatively lower frequency. However, an overall improvement in the performance is achieved through multiple cores operating simultaneously on multiple instructions [31]. For example, a dual-core chip running multiple applications is about 1.5 times faster than a chip with just one comparable core [15].

On the other hand, another type of processor emerged in terms of computational performance in the last few years. Graphics Processing Units, or GPUs, were initially developed for graphic processing of images and videos. The first GPU, a GeForce 256, was launched in 1999 by NVIDIA, the world leader in the graphics processor market [25]. GeForce 256 consists of a single-chip 3D real-time graphics processor that included almost every feature of high-end workstation 3D graphics pipelines available at that time. Due to market demand for high-quality real-time graphics in computer games and video, these processors have undergone considerable technological advancement over the years. For example, in a gaming application, one needs to render scenes at an increasing resolution at a rate of 60 frames per second [19].

Later, researchers who wanted to enhance the performance of their applications took notice of GPUs' high processing power and started to explore their use for non-graphic ones in the fields of science and engineering. This trend became known as General-Purpose computation on the GPU (GPGPU). However, developing non-graphic applications was a complex activity since GPUs had been developed to run features in graphics applications. For example, to run multiple instances of a function in parallel, one had to write them as pixel shaders[2]. The input data had to be stored in texture images and issued to the GPU by submitting triangles. Finally, the output data had to be cast as a set of pixels generated from raster operations[3].

Finally, NVIDIA focused on finding new approaches to make the development of non-graphic applications more intuitive in the GPUs. This came through Compute Unified Device Architecture (CUDA). Among the innovations contained in its architecture is the introduction of a more generic parallel programming model with parallel threads hierarchy, barrier synchronization, and atomic operations. Presently, GPUs use thousands of parallel cores that run thousands of threads in parallel, allowing the generation of

[2]Components of the GPU programmed to change pixel light and color patterns [21].

[3]Graphic operation to smooth border area colors and opacity.

parallel applications in many fields of research [38].

4.1 Architecture of a GPU

The most common approach to execute non-graphic applications on NVIDIA graphic processors is using GPUs with CUDA architecture. This architecture corresponds to the framework by which NVIDIA produces GPUs able to perform both traditional graphics rendering tasks and general tasks. In 2006 NVIDIA launched GeForce 8800 GTX, which was the first GPU with this architecture. GeForce 8800 GTX includes several new components designed strictly for GPU computing and aimed to remedy limitations that prevented previous GPUs from being legitimately useful for GPGPU [29]. Since then, more powerful graphics processors have been launched by the company.

The architecture of an NVIDIA GPU is typically composed of streaming multiprocessors (SMs). Each SM contains several streaming processor cores (SPs), in which several threads are executed. Figure 4.1 presents the first GPU with CUDA architecture. It has 16 SMs, each composed of 8 SIMD processors operating at 1350 MHz. Each multiprocessor has 8192 registers. Each thread block has 16 kB shared memory and the global memory of the GPU is 768 MB.

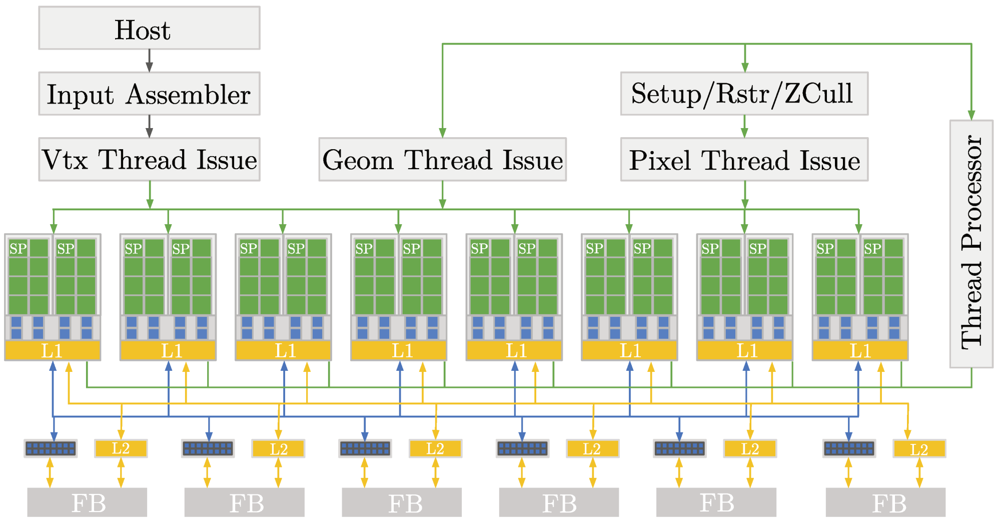

Figure 4.1: Example of the architecture of a GPU. Extracted from Svensson, Sheeran, and Claessen [35].

Developing a computing architecture that could perform non-graphic operations allowed GPUs to move from a graphic processor to a programmable parallel processor with high parallelism power. However, it was still necessary to develop a language to support user-processor interaction, or users would have to continue to treat their calculations as graphic problems. Then, NVIDIA launched CUDA parallel programming model. It is designed to overcome the challenge of writing non-graphic applications to run in the GPU while maintaining a low learning curve for programmers familiar with standard programming languages such as C. At the core of the CUDA programming model are three fundamental abstractions: a hierarchy of thread groups, shared memories, and barrier synchronization which are exposed to the programmer as a minimal set of language extensions [27].

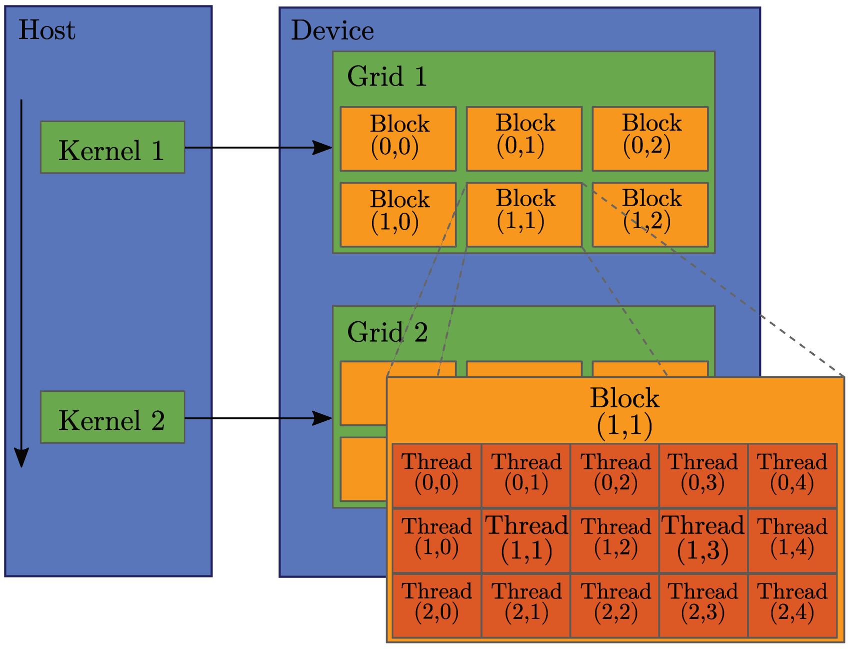

Figure 4.2: Thread hierarchy. Extracted from Kirk and Hwu [19].

CUDA applications support the coexistence of a host and one or more devices. The host is a traditional CPU where kernels are invoked. Kernels are functions that may be executed in the GPU, which is named device. Each process of a given kernel runs on a thread[4]. When launching a kernel, the CUDA runtime system generates a grid of threads organized in a two-level hierarchy. Figure 4.2 shows the hierarchy levels of a kernel launch. A grid is composed of thread blocks of the same size, while a thread block consists of a set of concurrently executing threads that communicate through barrier synchronization and shared memory.

4.2 CUDA program structure

Essentially, every CUDA application has both host and device codes. On the other hand, a traditional C/C++ application can be interpreted as a CUDA application with only the host code. Functions and data declarations that will be used by the GPU are recognized by the compiler using particular CUDA keywords. Thus, standard C/C++ compilers are not able to recognize them. Because of that, NVIDIA also developed a

[4]Simplified view of how a processor executes a program in modern computers.

compiler called NVIDIA C Compiler (NVCC). Figure 4.3 shows an overview of the compilation trajectory of a CUDA program. This trajectory separates the device code from the host code, compiles the device functions using NVCC and the host code using a C/C++ available compiler.

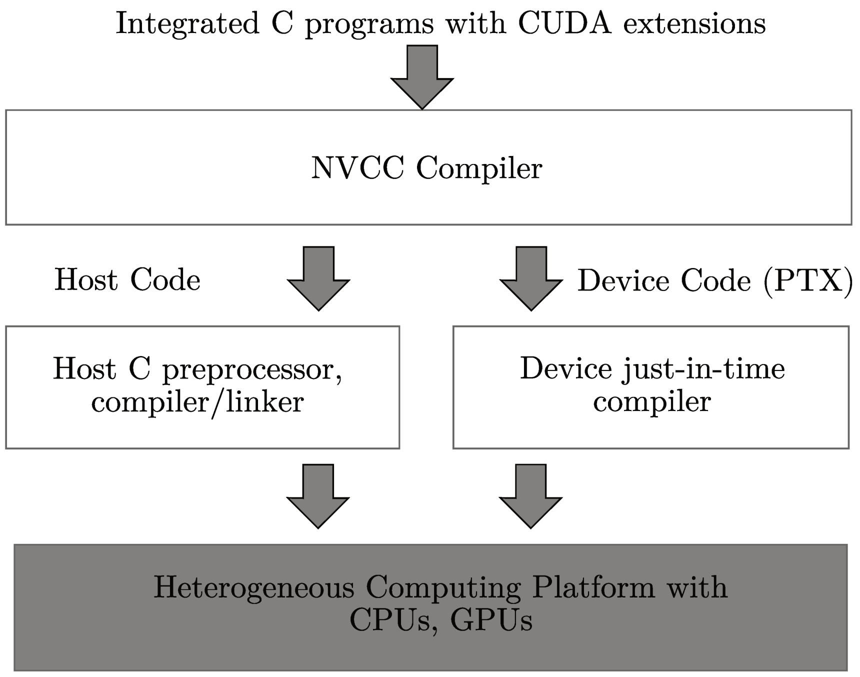
Integrated C programs with CUDA extensions
NVCC Compiler
Host Code
Device Code (PTX)
Host C preprocessor, compiler/linker
Device just-in-time compiler
Heterogeneous Computing Platform with CPUs, GPUs

Figure 4.3: Compilation trajectory of a CUDA program. Extracted from Kirk and Hwu [19].

4.3 CUDA programming model

To illustrate the GPU programming model, Code 4.1 performs the sum of the vectors A and B and returns the vector C.

Code 4.1: CUDA programming model.

```
__global__ void VectorAddKernel (int N, int *A, int *B, int *C) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < N) {
        C[tid] = A[tid] + B[tid];
    }
}

int main (void) {
    int N = 1000000;
    int A[N], B[N], C[N];
    int *dev_A, *dev_B, *dev_C;

    cudaMalloc(&dev_A, N * sizeof(int));
    cudaMalloc(&dev_B, N * sizeof(int));
15      cudaMalloc(&dev_C, N * sizeof(int));
16
17      for (int i = 0; i < N; i++) {
18          A[i] = rand() % 100;
19          B[i] = rand() % 100;
20      }
21
22      cudaMemcpy(dev_A, A, N * sizeof(int), cudaMemcpyHostToDevice);
23      cudaMemcpy(dev_B, B, N * sizeof(int), cudaMemcpyHostToDevice);
24
25      dim3 dimGrid(N, 1, 1), dimBlock(1, 1, 1);
26      VectorAddKernel<<<dimGrid, dimBlock>>>(N, dev_A, dev_B, dev_C);
27      cudaDeviceSynchronize();
28      cudaMemcpy(C, dev_C, N * sizeof(int), cudaMemcpyDeviceToHost);
29
30      cudaFree(dev_A);
31      cudaFree(dev_B);
32      cudaFree(dev_C);
33
34      return 0;
35  }
```

The parallelism on GPUs occurs through the execution of kernels. Launching a kernel typically generates several threads. The memory used by the input and output data of a kernel launch is allocated in the GPU's memory. At the end of the execution, the allocated memory has to be released. Moreover, the input data have to be transferred from the host to the device, whereas the output data is transferred from the device to the host. Also, the programmer has to set up the number of thread blocks and threads per block of the kernel.

The function *cudaMalloc* performs memory allocation. It allocates the required number of bytes of linear memory on the device and returns in a pointer[5] to the allocated memory. The allocated memory is suitably aligned for any kind of variable. Code 4.1 exemplifies this function in Lines 13 to 15. Next, the memory transfer operations are performed by the function *cudaMemcpy*. It copies the required bytes from the memory area pointed by the source pointer to the memory area pointed by the destination pointer. The direction of the copy is specified with the argument *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost*, *cudaMemcpyHostToHost* or *cudaMemcpyDeviceToDevice*. Lines 22 and 23 in Code 4.1 present the copy from the host to the device of the input arrays A and B, while Line 28 shows the copy from the device to the host of the output data stored in array C. Once the allocated memory is not to be used anymore in an application it has to be released. This operation is performed with *cudaFree*. It

[5]Variable whose value is a location in the computer's memory.

frees the memory space which is pointed by a pointer and is presented in Lines 30 to 32 of Code 4.1.

The kernel identifies and accesses threads by their identifier ID. The *blockIdx.x* variable provides the identifier of the current thread block, while *threadIdx.x* corresponds to the identifier of the current thread within a thread block and *blockDim.x* is the current block dimension. Thus, it is possible to have a unique identifier of a thread. The ID is calculated as shown in Line 2 of the code. Also, it is possible to have two and three-dimensional grids and thread blocks. Then, *blockIdx*, *threadIdx*, and *blockDim* can be determined in x, y, or z axes. This facilitates programming and provides a natural way to call computational elements in a specific domain such as vector, matrix, or volume.

Invoking a kernel is presented in Line 26 of Code 4.1. The arguments between the symbol $\lll$ $\ggg$ are the number of thread blocks and the number of threads per block of the kernel, respectively. The keyword ___*global*___ precedes the definition of the kernel. It indicates that the function is a kernel that is called from the host and is executed in the device. In addition to the ___*global*___ type, a kernel can be preceded by ___*device*___ or ___*host*___. A kernel preceded by ___*device*___ can only be called and executed in the device, while a kernel preceded by ___*host*___ is called and executed in the host. That is, it is a traditional C/C++ function. The *cudaDeviceSynchronize* function after the execution of the kernel in Line 27 ensures that the output data can only be accessed after command executions are finished.

4.4 CUDA device memory types

Having many threads available to execute a CUDA application can theoretically tolerate long memory access latency. However, one can easily run into a situation where traffic congestion in the global memory access paths prevents some threads from making progress, leaving some of the streaming multiprocessors idle. Thus, CUDA devices provide several different memory types, which may allow a significant fraction of the potential speed of the underlying hardware if used correctly. The memory types of a device are: global, local, constant, shared, and register memory. Each memory type has advantages and disadvantages. Table 4.1 summarizes the characteristics of the various CUDA memory spaces [14]:

Table 4.1: CUDA device memory types.

Memory	**Location**	**Cached**	**Access**	**Scope**
Register	On-chip	No	Read/write	One thread
Local	On-chip	Yes	Read/write	One thread
Shared	On-chip	N/A	Read/write	All threads in a block
Global	Off-chip (unless cached)	Yes	Read/write	All threads + host
Constant	Off-chip (unless cached)	Yes	Read	All threads + host

Registers: are the fastest memory on the GPU. They are a very valuable resource because they are the only memory on the GPU with enough bandwidth and a low enough latency to deliver peak performance. Registers are allocated to individual threads; each thread can only access its registers.

Local memory: local memory accesses occur for only some automatic variables. Generally, an automatic variable resides in a register except for the following: arrays that the compiler cannot determine are indexed with constant quantities; large structures or arrays that would consume too much register space; any variable the compiler decides to spill to local memory when a kernel uses more registers than are available on the SM.

Shared memory: this type of memory is allocated to thread blocks. All threads in a block can access variables in the shared memory locations allocated to the block. It is an efficient means for threads to cooperate by sharing their input data and the intermediate results of their work.

Global memory: corresponds to the “main” memory of the GPU. It has a global scope and lifetime of the allocating program. Global memory is limited by the total memory available to the GPU.

Constant memory: this type of memory is read-only capable. This lifetime is the entire application execution. Constant variables are often used for variables that provide input values to kernel functions. Constant memory resides in global memory but may be cached for efficient access.

Figure Fig. 4.4 illustrates the CUDA device memory model.

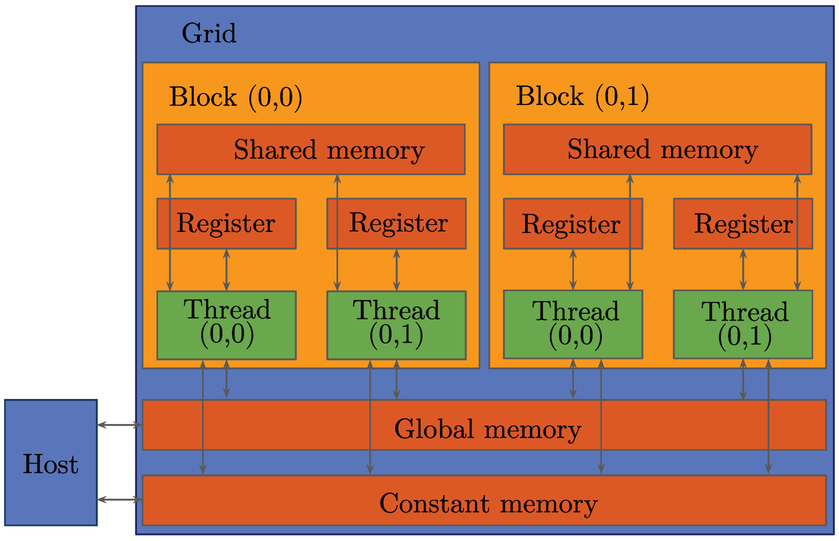

Figure 4.4: Overview of the CUDA device memory model. Extracted from Kirk and Hwu [19].

Chapter 5

Restructuring the elastoplastic finite element problem

This chapter describes the adopted structure to evaluate the global operators resulting from the Finite Element Method applied to the elastoplastic constitutive modeling with GPU programming. The structure is adapted from the Unstructured Displacement Approach (UDA), which is an algorithm for computing the global operators developed by Laouafa and Royis [20].

5.1 Matrix form of the finite element problem

The iterative process to approximate the solution for the elastoplastic problem is finding $\delta\vec{u} \in \mathbb{R}^{\mathcal{N}}$ such that:

$$\mathbf{K}\delta\vec{u} = -\mathbf{R}, \;\; \mathbf{K} \in \mathbb{R}^{\mathcal{N}\times\mathcal{N}}, \; \mathbf{R} \in \mathbb{R}^{\mathcal{N}} \tag{5.1}$$

The variable $\delta\vec{u}$ represents the finite element increment that is added to the approximate solution and $\mathcal{N}$ is the number of degrees of freedom. The variables $\mathbf{K}$ and $\mathbf{R}$ are the global Jacobian matrix and residual vector, respectively. The evaluation of the global operators is performed through the assembly process and is written as:

$$\mathbf{K} = \sum_{e=1}^{N} \mathbf{K}_e, \; \mathbf{K}_e \in \mathbb{R}^{\mathcal{N}_e\times\mathcal{N}_e}$$

$$\mathbf{R} = \sum_{e=1}^{N} \mathbf{R}_e, \; \mathbf{R}_e \in \mathbb{R}^{\mathcal{N}_e}$$

where $\mathcal{N}_e$ corresponds to element e number of degrees of freedom and N is the number of finite elements. The variables $\mathbf{K}_e$ and $\mathbf{R}_e$ are the element Jacobian matrix and residual, respectively. Following the notation introduced by Zienkiewicz [40], the evaluation of $\mathbf{K}_e$

and $\mathbf{R}_e$ is written as:

$$\mathbf{K}_e = \int_{\Omega_e} \mathbf{B}_e^t \mathbf{D}_{ep} \mathbf{B}_e$$

$$\mathbf{R}_e = \int_{\Omega_e} \mathbf{B}_e^t \vec{\sigma}_e^{proj}$$

In terms of numerical integration, $\mathbf{K}_e$ and $\mathbf{R}_e$ can be described as follows:

$$\mathbf{K}_e = \sum_{k=1}^{np_e} \omega_k |J_k| \mathbf{B}_e^t \left(\vec{\xi}_k\right) \mathbf{D}_{ep} \left(\vec{\xi}_k\right) \mathbf{B}_e \left(\vec{\xi}_k\right) \tag{5.2}$$

$$\mathbf{R}_e = \sum_{k=1}^{np_e} \omega_k |J_k| \mathbf{B}_e^t \left(\vec{\xi}_k\right) \vec{\sigma}_e^{proj} \left(\vec{\xi}_k\right) \tag{5.3}$$

where np_e is the number of integration points of element e and the variables ω_k and J_k correspond to the integration rule weight and the Jacobian of the transformation at integration point k, respectively. $\mathbf{B}_e \left(\vec{\xi}_k\right)$ is the strain-displacement matrix at integration point k. Whereas $\mathbf{D}_{ep} \left(\vec{\xi}_k\right)$ is elastoplastic constitutive matrix at the integration point k. Finally, $\vec{\sigma}_e^{proj} \left(\vec{\xi}_k\right)$ corresponds to the projection of $\vec{\sigma}_e^{trial} \left(\vec{\xi}_k\right)$ after the return mapping scheme at the integration point k.

The local stress-strain relationship, that is, the stress-strain relationship at integration point k of element e is described as:

$$\vec{\sigma}_e^{trial} \left(\vec{\xi}_k\right) = \mathbf{D}\, \vec{\varepsilon}_e^{\,e} \left(\vec{\xi}_k\right) \tag{5.4}$$

where $\mathbf{D}$ and $\vec{\varepsilon}_e^{\,e} \left(\vec{\xi}_k\right)$ are the elastic constitutive matrix and the elastic strain at integration point k, respectively. Considering a two-dimensional problem the matrix $\mathbf{D}$ is:

$$\mathbf{D} = \begin{bmatrix} 2\mu + \lambda & 0 & \lambda \\ 0 & \mu & 0 \\ \lambda & 0 & 2\mu + \lambda \end{bmatrix}$$

The process of evaluating the solution for the elastoplastic problem consists of an iterative process. Then, consider the above-mentioned statements are defined at the current pseudo-time t_{n+1}. The elastic strain at t_{n+1} is described as follows:

$$\vec{\varepsilon}_e^{\,e} \left(\vec{\xi}_k\right)^{n+1} = \vec{\varepsilon}_e \left(\vec{\xi}_k\right)^{n+1} - \vec{\varepsilon}_e^{\,p} \left(\vec{\xi}_k\right)^{n} \tag{5.5}$$

where $\vec{\varepsilon}_e^{\,p} \left(\vec{\xi}_k\right)^{n}$ and $\vec{\varepsilon}_e \left(\vec{\xi}_k\right)^{n+1}$ correspond to the local plastic strain at pseudo-time t_n and the local total strain at pseudo-time t_{n+1}, respectively. The local total strain is given by:

$$\vec{\varepsilon}_e \left(\vec{\xi}_k\right)^{n+1} = \delta\vec{\varepsilon}_e \left(\vec{\xi}_k\right) + \vec{\varepsilon}_e \left(\vec{\xi}_k\right)^{n} \tag{5.6}$$

The variable $\delta\vec{\varepsilon}_e\left(\vec{\xi}_k\right)$ corresponds to the local strain increment that is added to the total strain and is described as follows:

$$\delta\vec{\varepsilon}_e\left(\vec{\xi}_k\right) = \mathbf{B}_e\left(\vec{\xi}_k\right)\delta\vec{u}_e\left(\vec{\xi}_k\right) \tag{5.7}$$

The total strain $\vec{\varepsilon}_e$ and its decomposition ($\vec{\varepsilon}_e^{\,e}$ and $\vec{\varepsilon}_e^{\,p}$), as well as the trial stress $\vec{\sigma}_e^{\,trial}$, its projection $\vec{\sigma}_e^{\,proj}$ and the increment strain $\delta\vec{\varepsilon}_e$ are presented in Voigt notation.

$\mathbf{B}_e\left(\vec{\xi}_k\right)$ is presented as:

$$\mathbf{B}_e\left(\vec{\xi}_k\right) = \begin{bmatrix} \frac{\partial\phi_1}{\partial x} & 0 & \frac{\partial\phi_2}{\partial x} & 0 & \dots & \frac{\partial\phi_{\mathcal{N}_{ex}}}{\partial x} & 0 \\ \frac{\partial\phi_1}{\partial x} & \frac{\partial\phi_1}{\partial y} & \frac{\partial\phi_2}{\partial x} & \frac{\partial\phi_2}{\partial y} & \dots & \frac{\partial\phi_{\mathcal{N}_{ex}}}{\partial x} & \frac{\partial\phi_{\mathcal{N}_{ey}}}{\partial y} \\ 0 & \frac{\partial\phi_1}{\partial y} & 0 & \frac{\partial\phi_2}{\partial y} & \dots & 0 & \frac{\partial\phi_{\mathcal{N}_{ey}}}{\partial y} \end{bmatrix}$$

The matrix contains the values of the partial derivatives of the displacement interpolation functions at integration point k for a two-dimensional element. The variables $\mathcal{N}_{ex}$ and $\mathcal{N}_{ey}$ are the number of degrees of freedom in x and y, respectively.

5.1.1 Classical FEM assembly

The classical approach to perform the assembly of the Jacobian matrix and residual vector is to serially compute each element contribution $\mathbf{K}_e$ and $\mathbf{R}_e$ and add them to the global matrix and vector $\mathbf{K}$ and $\mathbf{R}$ according to the connectivity of the elements. Also, $\mathbf{K}_e$ and $\mathbf{R}_e$ are composed of the sum of integrals over the element. Eventually, in an iterative process $\mathbf{K}$ can be kept fixed, then only $\mathbf{R}_e$ needs to be assembled into $\mathbf{R}$. Algorithm 5.1 illustrates the classical assembly process. This algorithm has the following steps:

- Initialize the global Jacobian matrix and residual vector with zero;
- Perform a loop over the finite elements;
  - Compute $\mathbf{K}_e$ and $\mathbf{R}_e$;
  - Insert the element contribution on $\mathbf{K}$ and $\mathbf{R}$.

The characteristics of the classical FEM assembly are:

- Since the finite elements share connectivities, the contributions of different elements to one connectivity would overlap if the algorithm is parallel;
- This approach does not take advantage of the presented constant data during the evaluation of Eqs. (5.2) and (5.3);
- The classical FEM assembly uses little memory resources;

- The algorithm could be executed in parallel by coloring the elements and assemble all the elements belonging to a single color simultaneously.

Algorithm 5.1. Classical FEM assembly.

```
1: $\mathbf{K} \leftarrow 0^{\mathcal{N}\times\mathcal{N}}$ and $\mathbf{R} \leftarrow 0^{\mathcal{N}}$
2: for $k \leftarrow 1$ to $N$ do
3:     Compute $\mathbf{K}_e = \int_{\Omega_e} \mathbf{B}_e^t \mathbf{D}_{ep} \mathbf{B}_e$
4:     Compute $\mathbf{R}_e = \int_{\Omega_e} \mathbf{B}_e^t \vec{\sigma}_e^{proj}$
5:     for $i \leftarrow 1$ to $\mathcal{N}_e$ do
6:         $i_{dest}$ = connectivity$(i, k)$
7:         // Element vector assembly
8:         $\mathbf{R}\,(i_{dest})$ += $\mathbf{R}_e(i)$
9:         for $j \leftarrow 1$ to $\mathcal{N}_e$ do
10:            $j_{dest}$ = connectivity$(j, k)$
11:            // Element matrix assembly
12:            $\mathbf{K}\,(i_{dest}, j_{dest})$ += $\mathbf{K}_e(i, j)$
13:        end for
14:    end for
15: end for
```

5.1.2 FEM assembly using integration point contributions

The construction of $\mathbf{K}_e$ and $\mathbf{R}_e$, and therefore the global Jacobian matrix $\mathbf{K}$ and the global residual vector $\mathbf{R}$, are responsible for a significant part of the computational cost of the FEM assembly process presented in the previous section, especially during the recurrent assignment of information on each integration point that depends on the chosen integration rule. The computational cost tends to increase significantly in the analysis of nonlinear problems since the evaluation of the global Jacobian matrix and the global residual vector is performed at each iteration till the problem reaches convergence.

Unstructured Displacement Approach

The Unstructured Displacement Approach (UDA) is an algorithm for computing the global operators arising from a FEM problem developed by Laouafa and Royis [20]. The UDA consists of an integration point by integration point (IBI) method to perform the assembly process. The approach introduces three operators: $\hat{\mathbf{B}}$, $\hat{\mathbf{D}}$ and $\hat{\mathbf{W}}$. The first operator corresponds to the global strain–displacement operator; the second one is the 'rheological' operator, which components are the local constitutive relationships. The third operator is the one associated with the weak equilibrium and numerical integration rule. These three operators remain distinct and uncoupled during all the iterations of the nonlinear process of resolution. Thus, only one computation is required, whatever the number of iterations linked with the iterative solving process.

The algebraic problem presented in Eq. (5.1) can be rewritten as:

$$\mathbf{K}\delta\vec{u} + \mathbf{R}_l = \left(\hat{\mathbf{B}}^t\hat{\mathbf{W}}\hat{\mathbf{D}}\hat{\mathbf{B}}\right)\delta\vec{u} + \mathbf{R}_l = \mathbf{R}_\sigma + \mathbf{R}_l,\ \mathbf{R}_\sigma \in \mathbb{R}^{\mathcal{N}},\ \ \mathbf{R}_l \in \mathbb{R}^{\mathcal{N}} \tag{5.8}$$

where $\mathbf{R}_l$ is the linear residual resulting from the boundary conditions of the problem and remains constant during the iterations of the analysis of nonlinear problems. $\mathbf{R}_\sigma$ is the volumetric residual in which evaluation is required at each iteration.

The matrix $\hat{\mathbf{B}}$ is a sparse matrix of size $N_\sigma \times \mathcal{N}$, which components are the partial derivatives of the displacement interpolation functions. The size N_σ corresponds to the sum $\sum_{e=1}^{N} np_e\ n_\sigma$, where np_e is the number of integration points for each element and n_σ is the number of components of the stress tensor in Voigt notation. The size $\mathcal{N}$ corresponds to the number of degrees of freedom of the system. This operator is linear concerning the displacement and transforms the global solution vector $\delta\vec{u}$ to the values of the strain increment values at the integration points:

$$\delta\vec{\varepsilon} = \hat{\mathbf{B}}\delta\vec{u},\ \ \hat{\mathbf{B}} \in \mathbb{R}^{N_\sigma \times \mathcal{N}},\ \ \delta\vec{\varepsilon} \in \mathbb{R}^{N_\sigma}. \tag{5.9}$$

The matrix $\hat{\mathbf{D}}$ is the constitutive operator transforming the integration points elastic strain into integration point stress:

$$\vec{\sigma}^{\,trial} = \hat{\mathbf{D}}\vec{\varepsilon}_e^{\,e},\ \ \hat{\mathbf{D}} \in \mathbb{R}^{N_\sigma \times N_\sigma},\ \ \vec{\sigma} \in \mathbb{R}^{N_\sigma}. \tag{5.10}$$

where $\vec{\varepsilon}_e^{\,e}$ is the elastic strain defined in Eq. (5.5) at the integration points.

Finally, the volumetric residual $\mathbf{R}_\sigma$ is computed as:

$$\mathbf{R}_\sigma = \hat{\mathbf{B}}^t\hat{\mathbf{W}}\vec{\sigma}^{\,proj} \tag{5.11}$$

where $\hat{\mathbf{W}}$ is a diagonal matrix and contains all information about the integration rule: the weight and the determinant of the Jacobian values for all integration points. $\vec{\sigma}^{\,proj}$ is the projection of $\vec{\sigma}^{\,trial}$. Table 5.1 summarizes the UDA operators introduced by Laouafa and Royis [20]:

Table 5.1: UDA operators.

Operator	**Size**	**Description**
$\hat{\mathbf{B}}$	$N_\sigma \times \mathcal{N}$	Global strain–displacement
$\hat{\mathbf{D}}$	$N_\sigma \times N_\sigma$	Constitutive relationship
$\hat{\mathbf{W}}$	$N_\sigma \times N_\sigma$	Information of the integration rule

Some important observations can be made about the expressions in Eqs. (5.9) and (5.11):

- Once the geometrical partition and the polynomial order are assigned to every element in a finite element mesh, the global strain-displacement operator $\hat{\mathbf{B}}$ and the diagonal matrix operator $\hat{\mathbf{W}}$ with information of the integration rule are constant during the finite element computations, i.e., they only have to be evaluated once;
- The construction of $\hat{\mathbf{B}}$ involves overlapping information corresponding to common degrees of freedom. It can only be constructed in parallel if element coloring is used.
- The construction of $\hat{\mathbf{B}}^t\hat{\mathbf{W}}\hat{\mathbf{D}}\hat{\mathbf{B}}$ is implemented in two stages: first at the element level $\mathbf{B}_e^t\mathbf{W}_e\mathbf{D}\mathbf{B}_e$ for computing the element residual $\mathbf{R}_e$ and then assembling the element residuals.

5.2 Modified Unstructured Displacement Approach

In this work is explored the potential of accelerating the Jacobian matrix and residual vector evaluation by constructing a block-oriented storage pattern $\bar{\mathbf{B}} \in \mathbb{R}^{N_\sigma \times \bar{\mathcal{N}}}$. The matrix $\bar{\mathbf{B}}$ is block-diagonal and corresponds to a scattered version of $\hat{\mathbf{B}}$ presented by Laouafa and Royis [20]. While the matrix $\hat{\mathbf{B}}$ represents the global strain-displacement operator with overlapping information corresponding to common degrees of freedom, the matrix $\bar{\mathbf{B}}$ is arranged per finite element. Each block of $\bar{\mathbf{B}}$ corresponds to a finite element strain-displacement matrix $\mathbf{B}_e$, which in turn is an arrange of the strain-displacement matrices at the integration points of the element. The matrix structure for $\bar{\mathbf{B}}$ and $\mathbf{B}_e$ is shown graphically in Fig. 5.1.

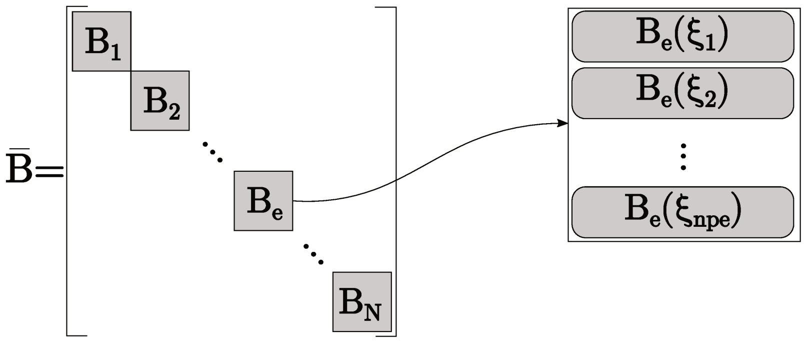

Figure 5.1: Representation of $\bar{\mathbf{B}}$ and $\mathbf{B}_e$.

The size N_σ corresponds to the number of rows of $\bar{\mathbf{B}}$ and is the sum $\sum_{e=1}^{N} np_e \; n_\sigma$ where np_e is the number of integration points for each element and n_σ is the number of components of the stress tensor in Voigt notation. Whereas $\bar{\mathcal{N}}$ is the number of columns of $\bar{\mathbf{B}}$ and corresponds to the sum of the size of each element $\sum_{e=1}^{N} \mathcal{N}_e$. $\bar{\mathbf{B}}$ contains the values of the partial derivatives of the displacement interpolation functions per element at the integration points. Using this structure, each element block $\mathbf{B}_e$ can be easily constructed in parallel.

$\bar{\mathbf{B}}$ operates on $\overline{\delta\vec{u}}$:

$$\overline{\delta\vec{\varepsilon}} = \bar{\mathbf{B}} S_c \delta\vec{u}, \quad \bar{\mathbf{B}} \in \mathbb{R}^{N_\sigma \times \bar{\mathcal{N}}}, \quad \delta\vec{\varepsilon} \in \mathbb{R}^{N_\sigma} \tag{5.12}$$

where $\overline{\delta\vec{\varepsilon}}$ is the scattered version of the global strain vector $\delta\vec{\varepsilon}$. S_c stands for the scatter operation, taking the mesh degrees of freedom to element-wise degrees of freedom. For details about gather and scatter operations, the reader can be referred to He *et al.* [18].

The global degrees of freedom vector $\delta\vec{u}$ is:

$$\delta\vec{u} = \begin{pmatrix} \delta u_1 & \delta u_2 & \dots & \delta u_{\mathcal{N}} \end{pmatrix}^T$$

The scattered version of $\delta\vec{u}$ is:

$$S_c\delta\vec{u} = \overline{\delta\vec{u}} = \Big(\underbrace{\delta u_1^1 \quad \delta u_2^1 \quad \dots \quad \delta u_{\mathcal{N}_1}^1}_{1} \quad \underbrace{\delta u_1^N \quad \delta u_2^N \quad \dots \quad \delta u_{\mathcal{N}_N}^N}_{N} \Big)^T$$

The term $\bar{\mathbf{R}}_\sigma \in \mathbb{R}^{\bar{\mathcal{N}}}$ corresponds to the scattered version of the volumetric residual vector $\mathbf{R}_\sigma$ and is expressed as:

$$\bar{\mathbf{R}}_\sigma = \bar{\mathbf{B}}^t \bar{\mathbf{W}} \overline{\vec{\sigma}}^{\,proj} \tag{5.13}$$

$\bar{\vec{\sigma}}^{\,proj}$ is the scattered version of the global stress vector $\vec{\sigma}^{\,proj}$ and $\bar{\mathbf{W}}$ carries the information of the integration rule.

The volumetric Jacobian matrix also has a scattered representation given by:

$$\bar{\mathbf{K}}_\sigma = \bar{\mathbf{B}}^t \bar{\mathbf{W}} \hat{\mathbf{D}}_{ep} \bar{\mathbf{B}} \tag{5.14}$$

where $\hat{\mathbf{D}}_{ep} \in \mathbb{R}^{N_\sigma \times N_\sigma}$ is the elastoplastic constitutive matrix at the integration points.

The scattered operators $\bar{\mathbf{K}}_\sigma$ and $\bar{\mathbf{R}}_\sigma$ are presented as the global volumetric operators $\mathbf{K}_\sigma$ and $\mathbf{R}_\sigma$ using a coloring scheme for the assembly process. Finally, the linear contributions $\mathbf{K}_l$ and $\mathbf{R}_l$ are added to the volumetric operators to form $\mathbf{K}$ and $\mathbf{R}$.

The constant preprocessing data for applying the Modified Unstructured Displacement Approach are listed as:

- Scattered strain-displacement operator;
- Integration points weight and determinant of the Jacobian operator;
- Elements connectivities;
- Finite element mesh coloring;
- Linear Jacobian matrix and residual arising from the boundary conditions of the problem.

5.2.1 Scattered strain-displacement operator

The scattered strain-displacement operator is a block-diagonal matrix with size $N_\sigma \times \bar{\mathcal{N}}$. This work presents two patterns to store $\bar{\mathbf{B}}$: sparse matrix and dense block matrices. For both approaches, all element matrices $\mathbf{B}_e$ are stored in the row-major format in a single array (*Value*) with size $N_\sigma \times \bar{\mathcal{N}}$. However, sparse matrix and dense block matrices storage patterns have different indexers for mapping $\bar{\mathbf{B}}$.

The following matrix presents a example of $\bar{\mathbf{B}}$ structure:

$$\bar{\mathbf{B}} = \begin{bmatrix} 2 & 3 & 1 & & & & & & \\ 5 & 7 & 8 & & & & & & \\ & & & 4 & 6 & 1 & & & \\ & & & 9 & 3 & 1 & & & \\ & & & & & & 2 & 7 & 5 \\ & & & & & & 3 & 4 & 2 \end{bmatrix}$$

For this example, the array *Value* is as follows:

$$Value = \begin{pmatrix} 2 & 3 & 1 & 5 & 7 & 8 & 4 & 6 & 1 & 9 & 3 & 1 & 2 & 7 & 5 & 3 & 4 & 2 \end{pmatrix}$$

Sparse matrix storage pattern

The sparse matrix storage pattern considers the compressed sparse row (CSR) format. It requires two arrays to access the scattered strain-displacement matrix. The array *RowPointer* contains the indices of the first nonzero element in the i$^{\text{th}}$ row of the array *Value* and has size $N_\sigma + 1$. The array *ColIndex* corresponds to the nonzero elements column indices and has size $N_\sigma \times \bar{\mathcal{N}}$. *RowPointer* and *ColIndex* for the example are:

$$RowPointer = \begin{pmatrix} 1 & 4 & 7 & 10 & 13 & 16 & 19 \end{pmatrix}$$
$$ColIndex = \begin{pmatrix} 1 & 2 & 3 & 1 & 2 & 3 & 4 & 5 & 6 & 4 & 5 & 6 & 7 & 8 & 9 & 7 & 8 & 9 \end{pmatrix}$$

Dense block matrices storage pattern

The dense block matrices storage pattern has five arrays associated with the scattered strain-displacement matrix. The arrays *RowSize* and *ColSize* store the number of rows and columns of each block matrix, respectively. Both arrays have size N. The array *MatrixPosition* stores the indices of the first element of a block matrix in the array *Value* and has size $N+1$. The arrays *RowFirstIndex* and *ColFirstIndex* store the row and column indices of the first element of a block matrix with size $N+1$. For the example, the presented arrays are:

$$RowSize = \begin{pmatrix} 2 & 2 & 2 \end{pmatrix} \qquad ColSize = \begin{pmatrix} 3 & 3 & 3 \end{pmatrix}$$
$$RowFirstIndex = \begin{pmatrix} 1 & 3 & 5 & 7 \end{pmatrix} \qquad ColFirstIndex = \begin{pmatrix} 1 & 4 & 7 & 10 \end{pmatrix}$$
$$MatrixPosition = \begin{pmatrix} 1 & 7 & 13 & 19 \end{pmatrix}$$

Table 5.2: Dense block matrices storage pattern arrays sizes.

Dense block matrices storage pattern	
Value	$N_\sigma \times \bar{\mathcal{N}}$
RowSize	N_e
ColSize	N_e
MatrixPosition	$N_e + 1$
RowFirstIndex	$N_e + 1$
ColFirstIndex	$N_e + 1$

Table 5.3: Sparse matrix storage pattern arrays sizes.

Sparse matrix storage pattern	
Value	$N_\sigma \times \bar{\mathcal{N}}$
RowPointer	$N_\sigma + 1$
ColIndex	$N_\sigma \times \bar{\mathcal{N}}$

5.2.2 Integration points weight and determinant of Jacobian operator

The operator $\bar{\mathbf{W}}$ corresponds to the diagonal matrix with size $N_\sigma \times N_\sigma$. Each diagonal element contains the value of the integration point weight multiplied by the determinant of the Jacobian at the integration point.

$$\bar{\mathbf{W}} = \begin{bmatrix} \omega_1 \times |\mathbf{J}_1| & 0 & \dots & 0 \\ 0 & \omega_2 \times |\mathbf{J}_2| & & \\ \vdots & & \ddots & 0 \\ 0 & \dots & 0 & \omega_{N_\sigma} \times |\mathbf{J}_{N_\sigma}| \end{bmatrix}$$

In terms of computational storage, the operator $\bar{\mathbf{W}}$ is stored as a single array with size N_{pts} where N_{pts} is the sum $\sum_{e=1}^{N_e} np_e$.

5.2.3 Elements connectivities

The elements connectivities determine where the element contributions resulting from the evaluation of $\mathbf{K}_e$ and $\mathbf{R}_e$ are added in the global Jacobian matrix and residual vector $\mathbf{K}$ and $\mathbf{R}$. Since the calculations of the proposed structure occur in the element level, an array with size $\bar{\mathcal{N}}$ containing the elements connectivities is stored to perform the scatter operations from the global system to the element level and the gather operations from the element level to the global system. Figure 5.2 shows an example of the connectivities for a two quadrilateral finite elements mesh.

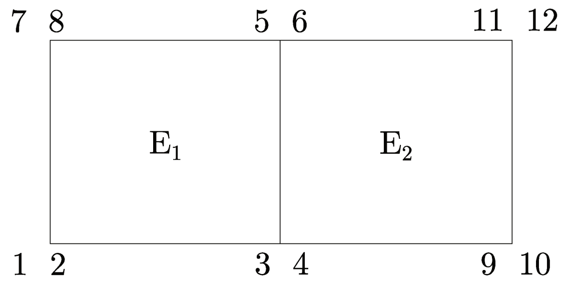

Figure 5.2: Global connectivities for a quadrilateral finite elements mesh.

For the example presented in Fig. 5.2, the global connectivities are:

$$\vec{\mathcal{C}} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{pmatrix}^T$$

The element-wise version of $\vec{\mathcal{C}}$ is:

$$\vec{\bar{\mathcal{C}}} = \big(\underbrace{1 \;\; 2 \;\; 3 \;\; 4 \;\; 5 \;\; 6 \;\; 7 \;\; 8}_{E_1} \;\; \underbrace{3 \;\; 4 \;\; 9 \;\; 10 \;\; 11 \;\; 12 \;\; 5 \;\; 6}_{E_2}\big)^T$$

5.2.4 Finite element mesh coloring

Finite element mesh coloring is used to avoid race conditions that may arise when the code execution order inadvertently affects the result. For instance, when parallel threads accumulate values in the same memory location, one may finish the computation faster than others. Thus, the result of the first threads is overlapped by the result of the last thread to finish the computation. In FEM, the elements of a mesh share connectivities, then the components of **K** and **R** are formed by the contributions from one or more elements. A coloring technique to avoid race conditions during the evaluation of **K** and **R** is to color the mesh elements such that two elements of the same color do not share any connectivity and therefore perform the assembly process of all elements of one color. Thus, the colors of a mesh are treated sequentially. Whereas, the elements of the same color are executed in parallel. Algorithm 5.2 presents the pseudo-code for the mesh coloring.

Algorithm 5.2. Mesh coloring algorithm.

```
1: $NeedsToContinue \leftarrow true$
2: $\vec{C}_{id} \leftarrow 0^N$ // Array of colors indices
3: $Id \leftarrow 0$
4: while NeedsToContinue do
5:     $NeedsToContinue \leftarrow false$
6:     for $k \leftarrow$ to $N$ do
7:         if *Element has color* then
8:             *Go to next element*
9:         end if
10:        Get connectivity of element $k$
11:        if *Any neighbour has color* $= Id$ then
12:            $NeedsToContinue \leftarrow true$
13:        else
14:            $\vec{C}_{id}[k] = Id$
15:            Connectivity of element $k$ is colored
16:        end if
17:    end for
18:    $Id = Id + 1$
19: end while
```

Figure 5.3 presents an example of a two-dimensional mesh with quadrilateral elements colored with Algorithm 5.2.

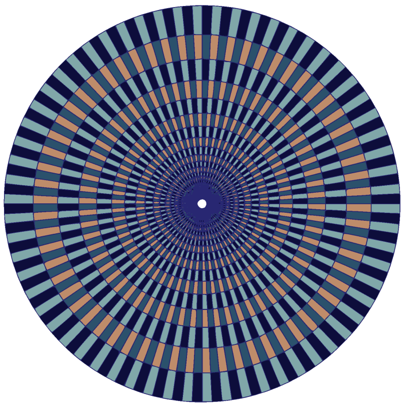

Figure 5.3: Colored two dimensional finite element mesh with quadrilateral elements.

5.2.5 Linear Jacobian matrix and residual

The linear Jacobian matrix $\mathbf{K}_l$ and residual vector $\mathbf{R}_l$ arising from the finite element discretization do not change during the iterative process to approximate the solution of the problem. Thus, it can be evaluated only once during the entire process. $\mathbf{K}_l$ and $\mathbf{R}_l$ consist of the contributions of the boundary elements of a finite element mesh. Figure 5.4 presents boundary elements for a two-dimensional finite element mesh with triangular elements.

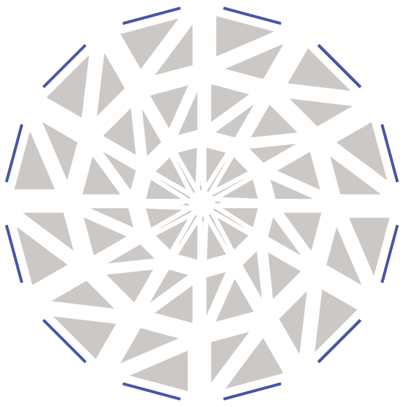

Figure 5.4: Boundary elements for a two dimensional finite element mesh with triangular elements.

For this work, the linear Jacobian matrix $\mathbf{K}_l$ and the linear residual $\mathbf{R}_l$ are constructed under the classical assembly approach presented in Algorithm 5.1. $\mathbf{K}_l$ is the global form of the linear Jacobian matrix with size $\mathcal{N} \times \mathcal{N}$. For computational aspects, it is stored in the CSR pattern $\mathbf{K}_l(\vec{K}_l, I\vec{K}_l, J\vec{K}_l)$. $\mathbf{R}_l$ corresponds to the global form of the linear residual and has size $\mathcal{N}$.

Table 5.4 summarizes the constant preprocessing data for the Modified Unstructured Displacement Approach:

Table 5.4: Summary of the constant preprocessing data.

Data	**Size**
Scattered strain-displacement operator ($\bar{\mathbf{B}}$)	Sizes in Table 5.2 or Table 5.3
Int. points weight and det. of jacobian operator ($\bar{\mathbf{W}}$)	N_{pts}
Elements connectivities ($\vec{\bar{\mathcal{C}}}$)	$\bar{\mathcal{N}}$
Finite element mesh coloring($\vec{C}_{id}$)	N
Linear Jacobian matrix and residual ($\mathbf{K}_l$ and $\mathbf{R}_l$)	$(\mathrm{NNZ}_b, \mathcal{N}+1, \mathrm{NNZ}_b)$ and $\mathcal{N}$

where NNZ_b stands for the number of nonzero entries of the linear Jacobian matrix $\mathbf{K}_l$.

5.2.6 Jacobian matrix and residual vector strategies

The evaluation of the global operators using the Modified Unstructured Displacement Approach is divided into two main folds: the computation of $\mathbf{K}$ and $\mathbf{R}$.

For the residual $\mathbf{R}$, there are two strategies of computation. The first is a block element approach where Eqs. (5.12) and (5.13) are computed in parallel at the element level with dense matrix-vector multiplications. In this case, the *dense block matrices storage pattern* is used for $\bar{\mathbf{B}}$. The second strategy consists in computing Eqs. (5.12) and (5.13) considering the *sparse matrix storage pattern* for $\bar{\mathbf{B}}$. In this case, only matrix-vector sparse operations are applied to the scattered operators. For both approaches, the evaluation of $\vec{\bar{\sigma}}^{proj}$ is performed at the integration point level following the return mapping scheme. The projection of the stress tensor is executed using principal stresses. Thus, its spectral decomposition at the integration points is performed following the method described in Appendix B. To obtain $\mathbf{R}_\sigma$, a series of gather, saxpy[1] and scatter operations are performed per color set to reduce $\bar{\mathbf{R}}_\sigma$ into $\mathbf{R}_\sigma$. Finally, the linear residual vector $\mathbf{R}_l$ contribution is added to $\mathbf{R}_\sigma$. Algorithm 5.3 presents the pseudo-code of the residual assembly process.

[1]Combination of scalar multiplication and vector addition.

Algorithm 5.3. Residual assembly process.

```
1: $\mathbf{R} \leftarrow 0^{\mathrm{N}}$
2: Scatter $\overline{\delta\vec{u}} \leftarrow \delta\vec{u}$
3: Compute $\overline{\delta\vec{\varepsilon}} = \bar{\mathbf{B}}\overline{\delta\vec{u}}$
4: for $k \leftarrow 1$ to $np$ do // Parallel execution
5:     Elastic predictor/Plastic corrector $\vec{\sigma}^{\,proj} \leftarrow \vec{\sigma}^{\,trial}$
6:     Concatenate $\overline{\vec{\sigma}}^{\,proj} \leftarrow \vec{\sigma}^{\,proj}$
7:     Compute $\vec{\varepsilon}^{\,p} = \vec{\varepsilon} - \mathbf{D}^{-1}\vec{\sigma}^{\,proj}$
8: end for
9: Compute $\bar{\mathbf{R}}_\sigma = \bar{\mathbf{B}}^t\bar{\mathbf{W}}\vec{\sigma}$
10: for $c \leftarrow 1$ to $nc$ do // Serial execution
11:     Gather $\mathbf{R}_c \leftarrow$ Color subset $\bar{\mathbf{R}}_\sigma$
12:     Add color contribution $\mathbf{R}_\sigma + = \mathbf{R}_c$
13: end for
14: Compute $\mathbf{R} = \mathbf{R}_\sigma + \mathbf{R}_l$
```

Figures 5.5 and 5.6 present both mentioned strategies to compute $\overline{\delta\vec{\varepsilon}}$. To compute $\bar{\mathbf{R}}_\sigma$ these strategies are also applied with $\bar{\mathbf{B}}$ in the transpose form.

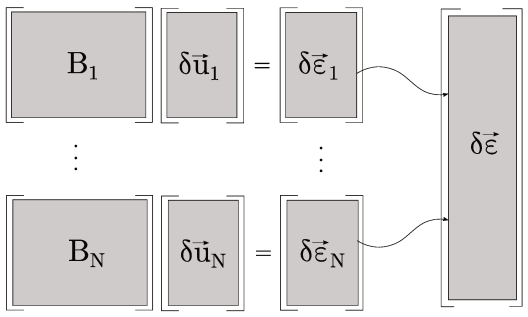

Figure 5.5: Evaluation of $\overline{\delta\vec{\varepsilon}}$ with block elements approach.

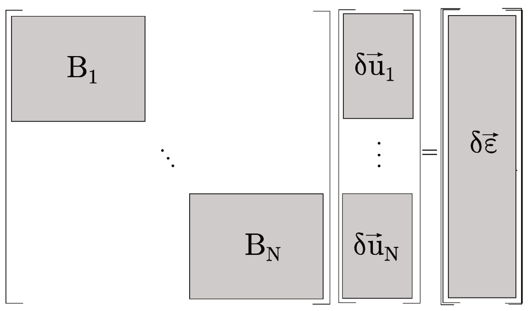

Figure 5.6: Evaluation of $\overline{\delta\vec{\varepsilon}}$ with sparse approach.

For the Jacobian matrix $\mathbf{K}$, it is only considered the block elements approach. Due to the associative characteristic for the elastoplastic equations, the matrix assembly is restricted to a sparse symmetric structure in CSR format or in other terms $\mathbf{K}(K, IK, JK)$. The element Jacobian matrices $\mathbf{K}_e$ of elements belonging to a color set are computed in parallel with dense matrix-matrix multiplications with $\mathbf{K}_e = \int_{\Omega_e} \mathbf{B}_e^t\mathbf{D}_{ep}\mathbf{B}_e$. The elastoplastic constitutive matrix is computed at the integration level with the strategy proposed by Cecílio *et al.* [7]. The contributions $\mathbf{K}_e$ are added to the array $\bar{K}_\sigma \in \mathbb{R}^{\mathrm{N\bar{N}Z}}$ of the matrix $\bar{\mathbf{K}}_\sigma(\bar{K}_\sigma, I\bar{K}_\sigma, J\bar{K}_\sigma)$. Then, using a loop over the colors, the entries of $\bar{K}_\sigma$ associated to the current color are inserted into the global vector $K_\sigma \in \mathbb{R}^{\mathrm{NNZ}}$ by gathering all the corresponding color entries, applying a saxpy operation over the color entries adding all the contributions to the color local array and finally applying a scatter operation. The size NNZ stands for the number of nonzeros entries of the global array, whereas N$\bar{\text{N}}$Z is the sum $\sum_{e=1}^{N} \mathcal{N}_e^2$ and represents the number of non overlapping nonzeros. When the loop over the colors is complete, the global $\mathbf{K}_\sigma(K_\sigma, IK_\sigma, JK_\sigma)$ is assembled. Lastly, the boundary contribution $\mathbf{K}_l$ is added to $\mathbf{K}_\sigma$. Algorithm 5.4 presents the pseudo-code of the Jacobian matrix assembly process.

Algorithm 5.4. Jacobian matrix assembly process.

```
1: $K_\sigma \leftarrow 0^{\text{NNZ}}$ and $\bar{K}_\sigma \leftarrow 0^{\text{N}\bar{\text{N}}\text{Z}}$
2: $l \leftarrow 0$
3: **for** $k \leftarrow 1$ to $N$ **do** // Parallel execution
4:     Compute $\mathbf{K}_e = \int_{\Omega_e} \mathbf{B}_e^t \mathbf{D} \mathbf{B}_e$
5:     **for** $i \leftarrow 1$ to $\mathcal{N}_e$ **do**
6:         **for** $j \leftarrow 1$ to $\mathcal{N}_e$ **do**
7:             // Element matrix scatter
8:             $\bar{K}_\sigma(l) += \mathbf{K}_e(i, j)$
9:             l++;
10:        **end for**
11:    **end for**
12: **end for**
13: **for** $c \leftarrow 1$ to $nc$ **do** // Serial execution
14:     Gather $K_c \leftarrow$ Color subset $K_\sigma$
15:     Add the color contribution $K_c +=$ Color subset $\bar{K}_\sigma$
16:     Scatter $K_\sigma \leftarrow K_c$
17: **end for**
18: Compute $\mathbf{K} = \mathbf{K}_\sigma + \mathbf{K}_l$
```

Chapter 6

Computational implementation and verification

This chapter describes the neoPZ environment, the implemented classes and kernels, as well as a Quasi-Newton method for accelerating the convergence. Also, it presents the verification method for the elastoplastic problem approximation.

6.1 neoPZ environment

neoPZ environment is an object-oriented C++ library for the development of finite element programs and is the base for this work. This library has many C++ classes to implement the main concepts for matrices, finite elements, finite element meshes, interpolation spaces, and materials constitute laws. It was created by Professor Philippe Remy Bernard Devloo and is maintained by him with the collaboration of the master and Ph.D. students in LabMeC at Unicamp. The environment is open-source and can be found at `https://github.com/labmec/neopz`. It is divided into several modules that correspond to the mentioned concepts. The main modules are presented below [32]:

- *Geom:* implements geometric abstractions for some element types (point, linear, triangle, quadrilateral, tetrahedron, hexahedron, and pyramid) and the mapping functions;
- *Topology:* contains basic definitions of a geometric element;
- *Shape:* constructs the basis functions according to the geometry of an element;
- *Material:* implements the differential equations representing the constitutive modeling of the materials supported by the environment;
- *Integral:* contains the integration rules for one, two and three dimensions and several polynomial orders;

- *Matrix:* implements different types of matrices, such as skyline, sparse, etc;
- *Mesh:* contains the concept of mesh. In neoPZ, there is a distinction between the geometric and computational meshes. The geometric mesh defines the spatial discretization of the domain and contains the geometric elements, nodes, and elementary coordinate systems. The computational mesh is related to the solution interpolation space;
- *Analysis:* manages the assembly process and the solution of the equation system related to a computational mesh;
- *StrMatrix:* is responsible for the interface between Matrix and Finite Element classes and operates with different matrix storage types;
- *LinearSolvers:* implements the preconditioning and solving methods of equation systems;
- *Post:* generates output files for visualization programs.

6.2 Implemented classes

To implement the approach presented in Chapter 5, six classes are implemented. More details of the implemented classes are described in the following items.

TPZIrregularBlocksMatrix

This class implements a matrix arranged per block. The blocks composing a *TPZIrregularBlocksMatrix* object can be either regular, i.e., the blocks have the same number of rows and columns, or irregular, where the number of rows and columns of each block can differ. This class implements the matrix $\bar{\mathbf{B}}$ storage and operations. The access of a *TPZIrregularBlocksMatrix* object is performed with the *sparse matrix* or the *dense block matrices storage pattern.* The data structure of this class consists of the array with the values of the element block matrices $\mathbf{B}_e$ as well as the arrays for accessing the matrix with the chosen storage pattern. The data structure is arranged in a *struct.* In this work, the required algebraic operation of a *TPZIrregularBlocksMatrix* is matrix multiplications. Then it is the only implemented. This class is derived from *TPZMatrix* class, in which other types of matrices are also derived from.

TPZConstitutiveLawProcessor

This class implements the elastoplastic constitutive modeling, that is, the evaluation of the elastoplastic constitutive matrix $\mathbf{D}_{ep}$ and the corresponding projection of the stress tensor $\vec{\sigma}^{\,proj}$ at the integration points of the elements of a mesh. *TPZConstitutiveLawProcessor* consists of some rewritten methods of the existing classes *TPZPlasticStepPV* and *TPZYCMohrCoulombPV* with some modifications so it can also be executed in the GPU. Both $\mathbf{D}_{ep}$ and $\vec{\sigma}^{\,proj}$ are computed using principal stresses. Thus, the spectral decomposition presented in Appendix B is also implemented in *TPZConstitutiveLawProcessor*. Moreover, the integration points weight and determinant of jacobian contributions are also applied in this class. The data structure of this class consists of the total number of integration points, the constitutive parameters, the array representing $\bar{\mathbf{W}}$ and arrays to store the history of the irreversible processes applied to the material at the integration points (total strain, plastic strain, and an identifier for the regime of the material behavior).

TPZNumericalIntegrator

This class implements the assembly process of the volumetric Jacobian matrix $\mathbf{K}_\sigma$ and residual vector $\mathbf{R}_\sigma$. The data structure of this class is a *TPZIrregularBlocksMatrix* object with the scattered strain-displacement operator, a *TPZConstitutiveLawProcessor* object for the elastoplastic constitutive information, the elements connectivities and the indices of the colors of the elements.

TPZElPlasticIPStrMatrix

This class computes the Jacobian matrix $\mathbf{K}$ and residual vector $\mathbf{R}$. It is derived from *TPZSymetricSpStructMatrix*, which implements symmetric sparse Structural Matrices. In turn, *TPZSymetricSpStructMatrix* is derived from *TPZStructMatrix*. This class is responsible for the interface between Matrix and Finite Element classes in neoPZ environment. The data structure of *TPZElPlasticIPStrMatrix* consists of a *TPZNumericalIntegrator* object encapsulating the scattered strain-displacement operator, the integration points and determinant of the Jacobian operator, the elements connectivities, the elastoplastic constitutive information and the finite element mesh coloring. Also, the linear Jacobian matrix and residual ($\mathbf{K}_l$ and $\mathbf{R}_l$) are computed and stored in this class. Finally, they are added to $\mathbf{K}_\sigma$ and $\mathbf{R}_\sigma$ to obtain $\mathbf{K}$ and $\mathbf{R}$.

TPZCudaCalls

The constant preprocessing data presented in Chapter 5 is computed using neoPZ environment. Also, the approximate solution post-process files are generated with classes of this environment. Therefore, a link between the host and device codes has to be implemented. This integration occurs in *TPZCudaCalls* class, which is responsible for encapsulating the kernels and CUDA libraries' functions calls.

TPZVecCuda

This class implements a vector scheme to manage memory transfers from the CPU to the GPU (and vice-versa). It is a template class; then, it supports any data types (*int*, *float*, *double*). This class allows the user to allocate memory of the GPU and transfer information between the GPU and CPU even in a .cpp extension file since this class wraps the CUDA API calls responsible for memory managing.

6.3 CUDA implementation

The CUDA programming model introduced by NVIDIA supports CPU and GPU execution of an application. Kernels are instructions executed in the GPU by several threads. A warp is a set of 32 threads within a thread block, while a thread block consists of a set of threads running concurrently that communicate through barrier synchronization and shared memory. To achieve the maximum performance of a kernel execution in the GPU, it is desired to maximize the parallel execution, i.e., to maximize the number of active threads in the GPU during the execution of a kernel. However, three main factors may limit better performances: registers, shared memory in a thread block, and the number of threads per thread block.

The gather and scatter operations presented in Algorithms 5.3 and 5.4 are performed with *cuSPARSE* library in the GPU. This library contains a set of basic linear algebra subroutines used for handling sparse matrices [28]. *cuBLAS* library is used for the saxpy operations in the GPU. This library corresponds to an implementation of BLAS (Basic Linear Algebra Subprograms) for NVIDIA GPUs [26]. Both *cuSPARSE* and *cuBLAS* are part of NVIDIA GPU-accelerated libraries. These libraries provide highly-optimized functions and are expected to perform 2x-10x faster than CPU-only alternatives. Every GPU architecture has its configuration of number thread blocks, shared memory, and memory bandwidth. The libraries developed by NVIDIA use a different optimization strategy for each architecture, which means that their use guarantees optimum performance of the kernels calls independent of the GPU accelerator card.

To perform the multiplication operations in Algorithm 5.3 for the evaluation of the global residual **R**, several matrix storage patterns were tested. This work presents two approaches to perform these operations. The first is to use the *sparse matrix storage pattern* and perform a sparse matrix-vector multiplication with *cuSPARSE* library. The second approach is to work with each element block in parallel. In this case, a kernel responsible for performing parallel operations where each thread is assigned to one element matrix-vector multiplication is constructed. For this approach, the *dense block matrices storage pattern* is used.

As seen in Algorithm 5.4, the global Jacobian matrix **K** assembly process consists in computing $\mathbf{K}_e$ in parallel. Thus, the kernel responsible for performing these computations assigns each element matrix computation to one thread. This kernel uses shared memory to access the element matrices in order to enhance the performance. However, there is a relation between the number of thread blocks and the amount of shared memory used by a kernel. More shared memory used by a thread block implies fewer thread blocks available by one kernel call. For the case of the present kernel, higher polynomial orders imply fewer thread blocks since the dimension of the element matrices increases demanding more shared memory.

6.4 Quasi-Newton method for accelerating the convergence

Some iterative methods have been proposed to calculate approximate solution of nonlinear finite element equations in elastoplasticity such as Newton-Raphson, modified Newton-Raphson and Initial Stiffness methods. The last mentioned uses the elastic stiffness matrix to update the solution. The following items present the Initial Stiffness Method and a corresponding modification applied to this method in order to accelerate the convergence.

Initial Stiffness Method

The method can be cast into the following steps:

1. Perform a single $\tilde{\mathbf{K}}$ assembly;
2. Decompose $\tilde{\mathbf{K}}$;
3. Compute Newton correction $\delta\vec{u}^{k-1} = -\tilde{\mathbf{K}}^{-1}\mathbf{R}\left(\vec{u}^{k-1}\right)$;
4. Perform a Newton update of $\vec{u}^{k} = \vec{u}^{k-1} + \delta\vec{u}^{k-1}$;
5. Perform 3 to 4, till the residue norm reaches the desired tolerance. i.e. $\left\|\mathbf{R}\left(\vec{u}^{k}\right)\right\| \leq \epsilon$.

Modified Initial Stiffness Method

This method uses two subsequent states or $\vec{u}^{\,k}$ and $\vec{y}^{\,k}$. Denoting the quasi Newton update $\vec{y}^{\,k}$:

$$\vec{y}^{\,k} = \vec{u}^{\,k-1} + \omega^{k-1}\,\delta\vec{u}^{\,k-1}. \tag{6.1}$$

Let be $\delta\vec{u}^{\,*} = -\tilde{\mathbf{K}}^{-1}\mathbf{R}\left(\vec{y}^{\,k}\right)$ and the new update state defined as follows:

$$\vec{u}^{\,k} = \vec{y}^{\,k} + \delta\vec{u}^{\,*} \tag{6.2}$$

where the factor ω is the so-called acceleration factor [33], defined as:

$$\omega^k = \omega^{k-1} + \frac{\delta\vec{u}^{\,k-1}\cdot\delta\vec{u}^{\,*}}{\delta\vec{u}^{\,k-1}\cdot\delta\vec{u}^{\,k-1}}. \tag{6.3}$$

The first iteration assumes $\omega^1 = 1$. This method requires a single assembly, and two linear solve and two function evaluations per update and is very robust when applied to problems with large plastic strain and complicated constitutive models.

6.5 Solution verification

The analytical solution for the linear setting is presented by Coussy [10]. The exact displacement, strain and stress fields are respectively:

$$\vec{u} = \frac{(1+\nu)\left(p_{int}-\sigma\right)}{E}\frac{r_{int}^2}{r}\mathbf{e}_r$$

$$\boldsymbol{\varepsilon} = \frac{(1+\nu)\left(\sigma-p_{int}\right)}{E}\frac{r_{int}^2}{r^2}\left(\mathbf{e}_r\otimes\mathbf{e}_r-\mathbf{e}_\theta\otimes\mathbf{e}_\theta\right)$$

$$\boldsymbol{\sigma} = \left(\sigma-p_{int}\right)\left(\mathbf{e}_r\otimes\mathbf{e}_r-\mathbf{e}_\theta\otimes\mathbf{e}_\theta\right)$$

The energy norm is one of the most natural and meaningful ways to quantify the error of an approximation. Thus, it is used to verify the solution's accuracy of the linear setting using the proposed structure. The energy norm is given by:

$$||\mathbf{e}||_E = \left\{\int_\Omega \left[\left(\mathbf{e}'\right)^2 + \mathbf{e}^2\right]\right\}^{1/2}$$

where $\mathbf{e}$ is the error and corresponds to the difference between the exact and the approximate solution.

To verify the nonlinear approximation, a Runge-Kutta solver was implemented. It is required to simplify the elastoplastic equations and recast the problem as the initial value problem:

$$\frac{d\mathbf{y}}{d\mathbf{x}} = \mathbf{f}\left(\mathbf{y}\right)$$

There are several considerations for this case:

- Describe the equations in terms of the Cylindrical coordinate system and its corresponding mixed form;

- The approximation is axisymmetric, leading to a displacement field $\vec{u}$ that depends only on the radius and just has a radial component, i.e. $\vec{u} = \Phi(r)$;
- The initial value problem is described in terms of one independent variable r, and the ODE data is prescribed at the external radius r_{ext};
- The number of discrete points is large enough to have a reasonable approximation.

The Runge-Kutta solver is implemented according to the paper presented by Büttner and Simeon [6] and neoPZ environment is used to perform the elastoplastic state update of the approximation. Moreover, the description of the equations in terms of the Cylindrical coordinate system is performed following Bradley [5].

Chapter 7

Results and discussion

The numerical results are presented in four sections organized as follow: two verifications are presented pointing to check the validity of the spatial approximation properties and elastoplastic process; the evaluation of the performance in time for CPU and GPU versions of the presented algorithms and strategies, and also for the classical residual and Jacobian matrix assembly with neoPZ environment; a memory consumption analysis; a Quasi-Newton iterative solver of the elastoplastic problem for accelerating the convergence.

7.1 Verification

A linear constitutive behavior with the physical conditions presented in Fig. 7.1 is considered to verify the approximation properties of the implementation. The elastic and elastoplastic problems associated with a circular domain representing a wellbore region with prescribed Neumann data are approximated. The internal pressure $p_{int} = \boldsymbol{\sigma}\mathbf{n}\cdot\mathbf{n}$ is applied in the wellbore walls, while an external normal stress $\sigma = \boldsymbol{\sigma}\mathbf{n}\cdot\mathbf{n}$ is applied over the external boundary. A hydrostatic pre-stress σ_0 is the initial stress.

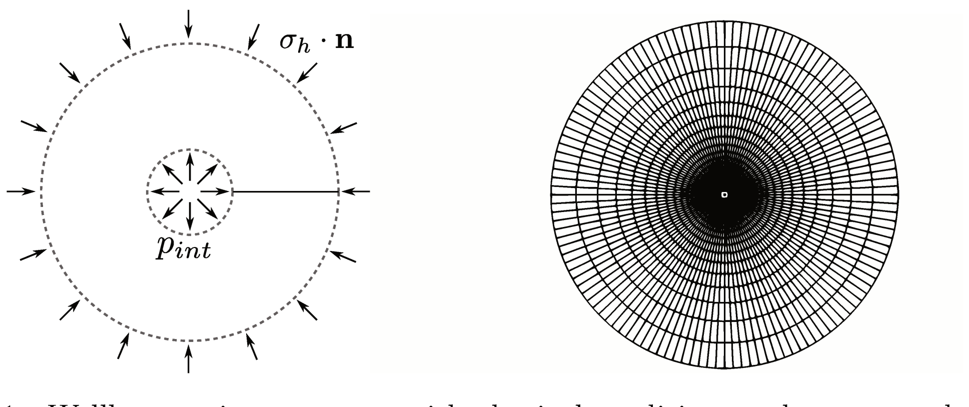

Figure 7.1: Wellbore region geometry with physical conditions and coarse mesh partition $\mathcal{T}_h|_{l=1}$.

Table 7.1 presents the set of geometrical partitions $\mathcal{T}_h$ for different refinement levels.

Table 7.1: Set of geometrical partitions $\mathcal{T}_h$ for different refinement levels.

$\mathcal{N}$	$\mathcal{T}_h\|_{l=1}$	$\mathcal{T}_h\|_{l=2}$	$\mathcal{T}_h\|_{l=3}$	$\mathcal{T}_h\|_{l=4}$	$\mathcal{T}_h\|_{l=5}$
$p = 1$	7936	32256	130048	522240	2093056
$p = 2$	31248	128016	518160	2084880	N/A
$p = 3$	69936	287280	1164336	4687920	N/A
N_e	3844	15876	64516	260100	1044484
h	0.0071	0.0041	0.0016	0.0010	0.0004

Table 7.2 shows the material properties used for the numerical results.

Table 7.2: Material properties used for numerical simulations.

Parameter	Symbol [unit]	Value
Internal pressure	p_{int} [MPa]	-40
External stress	σ [MPa]	-49.9937
Hydrostatic stress	σ_0 [MPa]	-50
Internal radius	r_{int} [m]	0.1
External radius	r_{ext} [m]	4.0
Young's modulus	E [MPa]	2000.0
Poisson's ratio	ν	0.2
Cohesion	c [MPa]	5.0
Friction	ϕ [°]	20

Linear setting verification

The energy norm is computed to verify the linear setting of the proposed structure. The polynomial order and partition $p = \{1, 2\}$ and $\mathcal{T}_h|_{l=1}$ (see Table 7.1) were selected with several uniform refinements. Disregarding the corresponding elastoplastic data, the parameters for the linear setting verification are presented in Table 7.2. It was obtained the expected approximation rate in the sense of energy norm. Figure 7.2 documents the verification of the optimal approximation properties for selected finite element discretization.

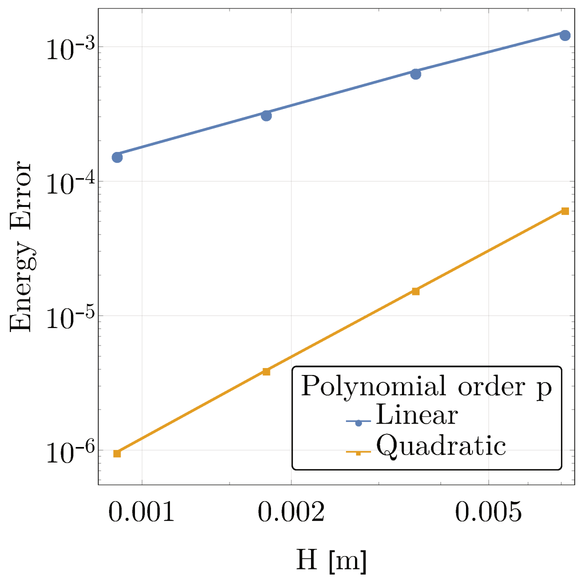

Figure 7.2: Energy error plots for $p = \{1, 2\}$ with coarse mesh partition $\mathcal{T}_h|_{l=1}$.

Nonlinear setting verification

The nonlinear setting verification of the proposed structure is performed comparing the approximate solution with the implemented Runge-Kutta solver solution. The finite element approximation was performed using a quadratic approximation with the geometrical partition $\mathcal{T}_h|_{l=1}$ and the material parameters presented in Table 7.2. Figure 7.3 shows a remarkable match between the approximations. The number of points for the Runge-Kutta approximation is 2000. The radius where the plasticity ends is approximately $r \approx 0.15\ m$ for both methods. Figure 7.3 documents the verification for the implementation considering a nonlinear setting.

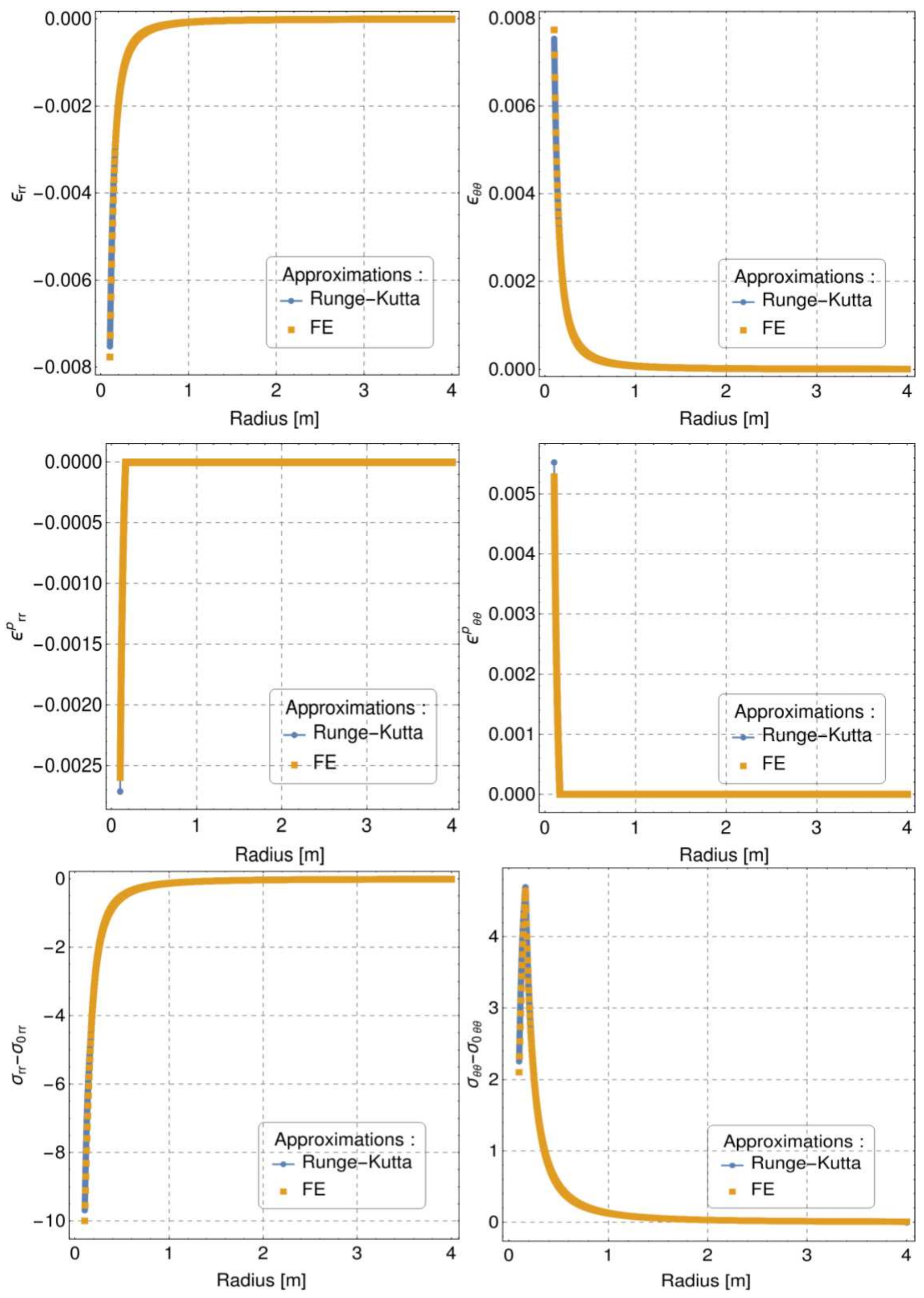

Figure 7.3: A Runge-Kutta comparison against a finite element approximation with $p = 2$ a mesh partition $\mathcal{T}_h|_{l=1}$.

7.2 Performance analysis

The numerical experiments were conducted in the High-Performance Computing Laboratory and Immersive and Interactive 3D Environment for Scientific Visualization for Petroleum Production cluster (Galileu) at the University of Campinas (UNICAMP). The cluster provides two processors Intel® Xeon® E5-2630 v3. The graphic processor is an NVIDIA® Tesla K40. Details about the tests platform are described in Table 7.3.

Table 7.3: Tests platform description.

Galileu cluster		
OS	Distribution	CentOS Linux 7 (Core)
	Kernel	3.10.0-327.el7.x86_64
CPU	Model	Intel® Xeon® E5-2630 v3
	Number of Cores	16
	Number of Threads	32
	Processor Base Frequency	2.40 GHz
	Hyper-Threading Technology	Yes
	Max Memory Size	768 GB
GPU	Model	NVIDIA® Tesla K40
	CUDA Capability	3.5
	Global Memory	12 Gb
	Multiprocessors (MP)	15
	CUDA Cores per MP	192
	Max Clock Rate	0.75 GHz
Compilation	GCC	4.8.5
	NVCC	8.0.44
	Flags	-std=gnu++11 -mtune=generic -march=x86-64 -O3 -ffast-math

For each configuration presented in Table 7.1, five executions were made for the CPU and GPU using the presented storage patterns. Also, five executions were made for the classical assembly approach. The comparison between the GPU and CPU parallel codes is required to verify the efficiency of the GPU compared with the CPU. The CPU code is implemented in C++ language and is parallelized using Thread Building Blocks, a C++ template library for task parallelism. The matrix multiplications are performed using BLAS, whereas the classical assembly approach is performed with neoPZ environment. The performance analysis takes into account the execution time of the algorithms presented. The results correspond to the average of the executions. The variability of the results is presented in Appendix C. Also, the results for the GPU's performance include the memory transfers between the CPU and GPU. At each iteration, the finite element solution is transferred from the CPU's to the GPU's memory for evaluating the residual. Later, the residual is transferred from the GPU's to the CPU's memory and the solution is updated on the CPU using the inverted stiffness matrix.

7.2.1 Residual

The performance analysis for the residual computation considers both storage patterns: *block matrices* and *sparse matrix storage patterns*. First, both storage patterns are compared for computations with the GPU and CPU. Then, a comparison between the GPU's and CPU's performance is presented. Finally, the performance for the residual assembly using the classical approach with neoPZ environment is compared with the GPU's.

GPU performance

Figure 7.4 shows the GPU's performance for the residual computation with both sparse matrix (SP) and block matrices (BL) storage patterns for the set of configurations presented in Table 7.1. The GPU's performance is slightly the same for block matrices storage pattern and sparse matrix storage pattern for partition $\mathcal{T}_h|_{l=1}$ with linear, quadratic and cubic polynomial orders. For partition $\mathcal{T}_h|_{l=5}$ with linear polynomial order, the GPU's performance with sparse matrix storage pattern is 2.490x faster than block matrices storage pattern. However, for partition $\mathcal{T}_h|_{l=4}$ with quadratic and cubic polynomial orders, the time difference goes up to 4.641x and 8.021x, respectively.

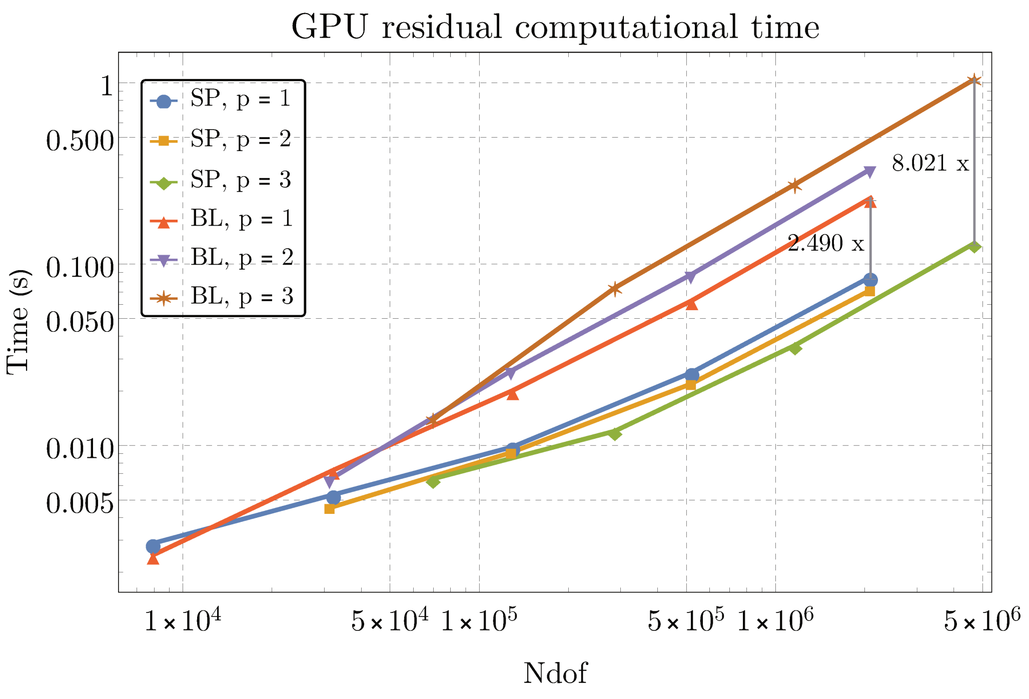

Figure 7.4: GPU residual computation performance for sparse matrix and block matrices storage patterns.

For the sparse matrix storage pattern, it is used a single sparse matrix-vector multiplication with *cuSPARSE* library. For the block matrices storage pattern, a kernel performs dense matrix-vector multiplications in parallel, where each thread executes one serial matrix-vector multiplication. Also, an alternative where each thread assigns one *cuBLAS* matrix-vector multiplication was implemented. This operation performs one parallelized matrix-vector multiplication. However, it does not take advantage of the parallelism power of the GPU for the element matrices because it is dedicated to larger dimension matrices. Moreover, to avoid race conditions, one *cuBLAS* context must be initialized for each matrix-vector multiplication. This operation is time-consuming, resulting in slow performance of the multiplication. The result shown in Fig. 7.4 presents only the kernel approach for the block matrices storage pattern.

CPU performance

Figure 7.5 presents the CPU's performance for the residual computation with the sparse matrix and block matrices storage patterns for the set of configurations presented in Table 7.1. It can be noticed that the CPU's performance for both storage patterns does not differ significantly. However, the CPU's performance for the block matrices storage pattern is slightly better than the sparse matrix storage pattern. For cubic polynomial order and partition $\mathcal{T}_h|_{l=4}$, the CPU's performance for the block matrices storage pattern is 1.579x faster than the sparse matrix storage pattern. For the sparse matrix storage pattern, a single sparse matrix-vector multiplication with BLAS is used. In contrast, for the block matrices storage pattern, parallelized dense matrix-vector multiplications are performed with BLAS. Thus, the presented results are expected since the performance for dense operators is generally faster than sparse operations on CPU architectures.

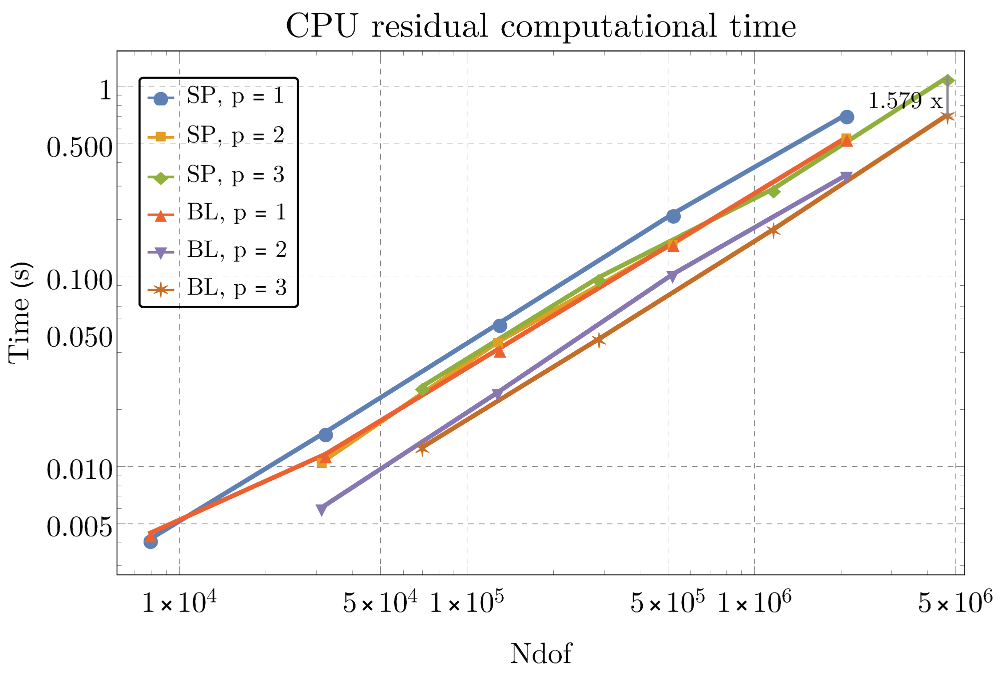

Figure 7.5: CPU residual computation performance for sparse matrix and block matrices storage patterns.

GPU and CPU comparison

Figure 7.6 presents the comparison between the CPU's and GPU's best performances. The CPU's performance is obtained using the block matrices storage pattern, while the GPU's considers the sparse matrix storage pattern. For partition $\mathcal{T}_h|_{l=1}$, the GPU and CPU's performance do not differ significantly. However, for more refined partitions, the GPU's performance is faster than the CPU's. For linear polynomial order and partition $\mathcal{T}_h|_{l=5}$, the GPU's performance is 6.394x faster than the CPU's. For partition $\mathcal{T}_h|_{l=4}$ with quadratic and cubic orders, the time ratio between the GPU and CPU is 4.797x and 5.479x, respectively.

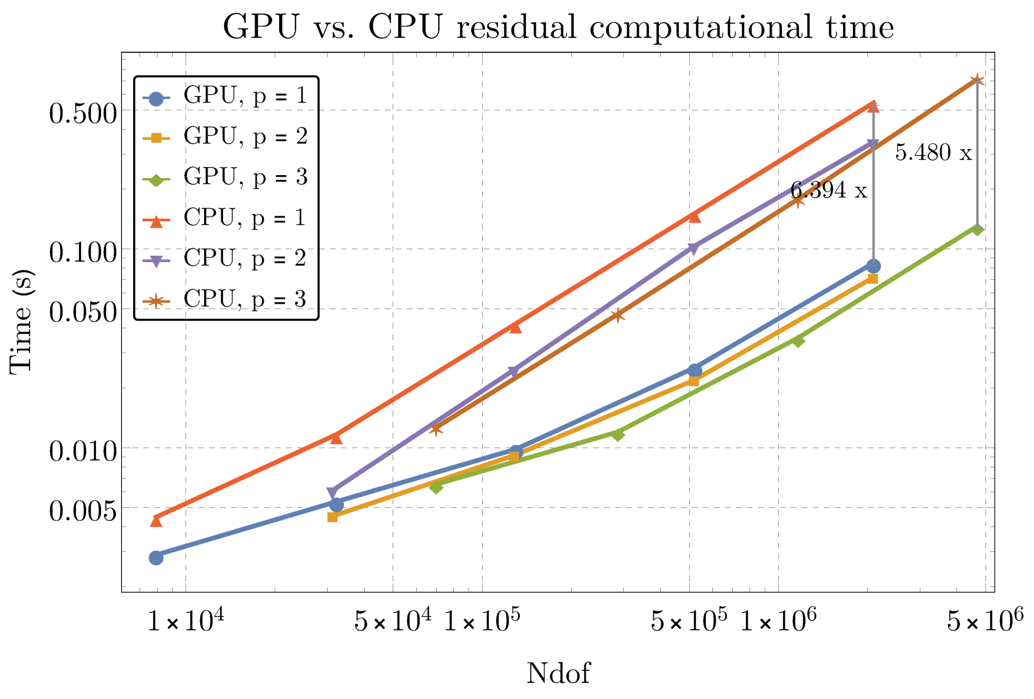

Figure 7.6: GPU and CPU residual computation performance comparison.

GPU and neoPZ comparison

Figure 7.7 shows the comparison between the GPU's performance using the sparse matrix storage pattern and neoPZ's performance using the classical assembly approach. It can be observed that for all partitions in Table 7.1 with linear, quadratic and cubic orders, the GPU's performance overcomes the classical assembly approach. For partition $\mathcal{T}_h|_{l=5}$ with linear polynomial order, the GPU's performance is 149.790x better than neoPZ's. Whereas for partition $\mathcal{T}_h|_{l=4}$ with quadratic and cubic orders, the time difference between the GPU and neoPZ is 40.060x and 23.716x, respectively. It is important to highlight that neoPZ environment performs the assembly in parallel. Therefore, it is noticed that the proposed structure leads to better performance than the classical one.

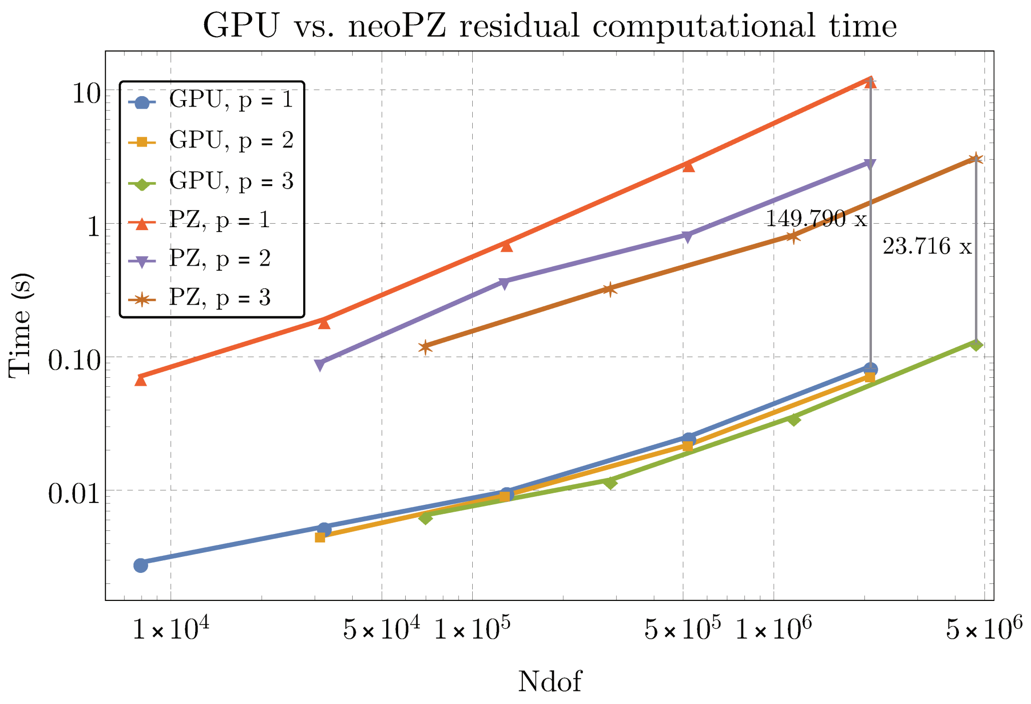

Figure 7.7: GPU and neoPZ residual computation performance comparison.

7.2.2 Jacobian matrix

The performance analysis for the Jacobian matrix computation with the GPU and CPU considers the *block matrices storage pattern*. Also, the performance for the Jacobian matrix assembly using the classical approach with neoPZ environment is compared with the GPU's performance.

GPU and CPU comparison

Figure 7.8 presents the comparison between the CPU's and GPU's performance for the computation of the Jacobian matrix. It can be observed that for linear polynomial order, the GPU's performance reaches better results when compared to the CPU's for partitions in Table 7.1. For partition $\mathcal{T}_h|_{l=5}$, the GPU's performance overcomes the CPU's by 5.360x. For quadratic polynomial order, the CPU's and GPU's performance are almost the same. However, rising the polynomial order results in a bad performance for the GPU. Several tests for different configurations of thread blocks and shared memory were implemented. Since all threads in a thread block access the same shared memory, there is a relation between the amount of available shared memory to store the element matrices and the number of active threads in a thread block. Thus, the GPU's performance does not overcome the CPU's for cubic order, being 1.937x slower than the CPU's for partition $\mathcal{T}_h|_{l=4}$.

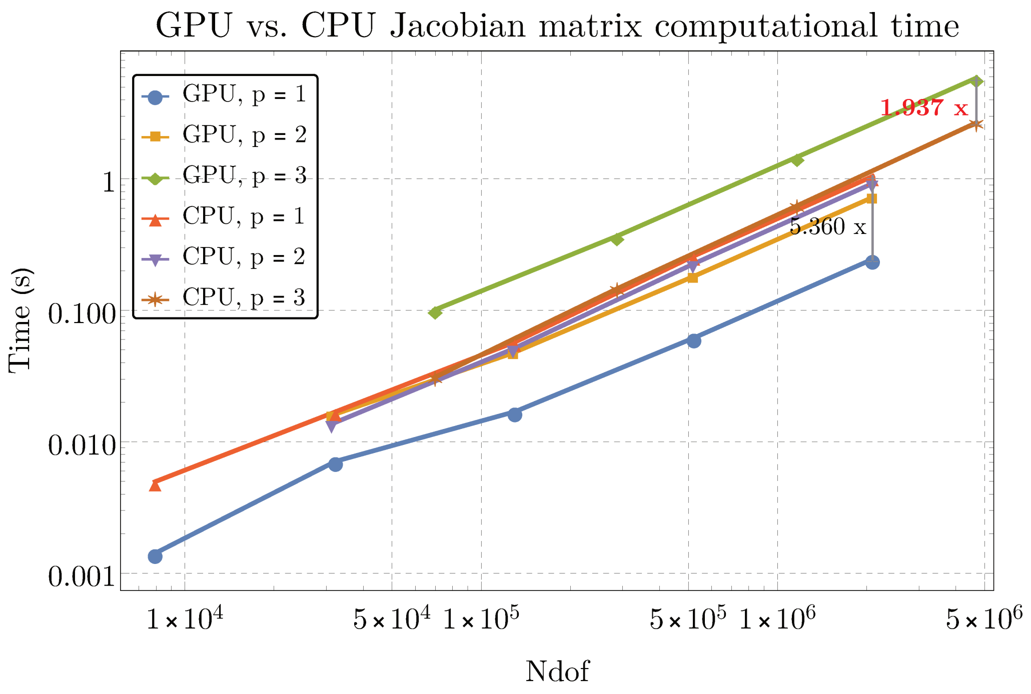

Figure 7.8: GPU and CPU Jacobian matrix computation performance comparison.

GPU and neoPZ comparison

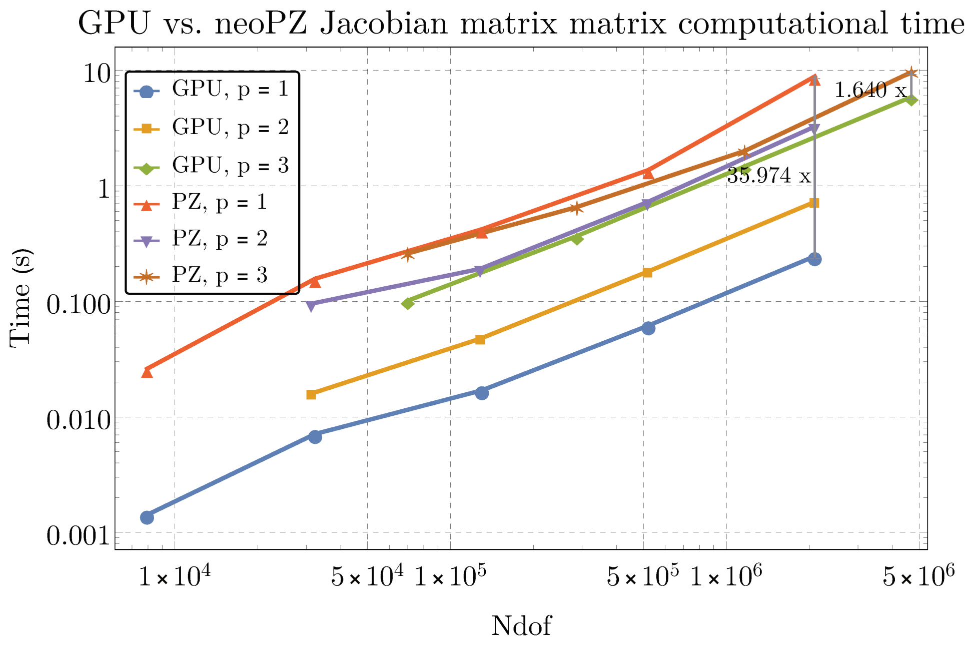

Figure 7.9: GPU and neoPZ Jacobian matrix computation performance comparison.

Figure 7.9 shows the comparison between the GPU's and neoPZ's performance for the computation of the Jacobian matrix. It is noticed that for linear and quadratic orders in partitions of Table 7.1, the GPU's performance overcomes neoPZ's. For linear and quadratic polynomial orders and partition $\mathcal{T}_h|_{l=1}$, the GPU is 18.371x and 6.000x faster, respectively. For partition $\mathcal{T}_h|_{l=5}$ and linear order, the time difference between the GPU's and neoPZ's performance is 35.974x. Whereas for quadratic order it is 4.489x. However, there is no relevant time difference between the GPU's and neoPZ's performance for cubic order. For partition $\mathcal{T}_h|_{l=4}$, for example, the GPU's performance overcomes neoPZ's 1.640x.

7.3 Memory consumption

Block matrices and sparse matrix storage patterns have different indexers to assign the scattered strain-displacement matrix. Both patterns store the values of the partial derivatives of the basis functions of each element at the integration points in an array. However, the first approach also requires the storage of the number of rows and columns of each block matrix, the position of the first element of a block matrix in the partial derivatives array as well as the row and column indices of the first element of a block matrix. The sparse matrix storage pattern considers the compressed sparse row (CSR) format. It assigns the global scattered strain-displacement matrix with an array of indices with the first nonzero element in the i$^{\text{th}}$ row of the sparse matrix and an array of the nonzero elements column indexes. Figure 7.10 shows the memory consumption per element for both sparse and blocks storage patterns for linear, quadratic and cubic polynomial orders.

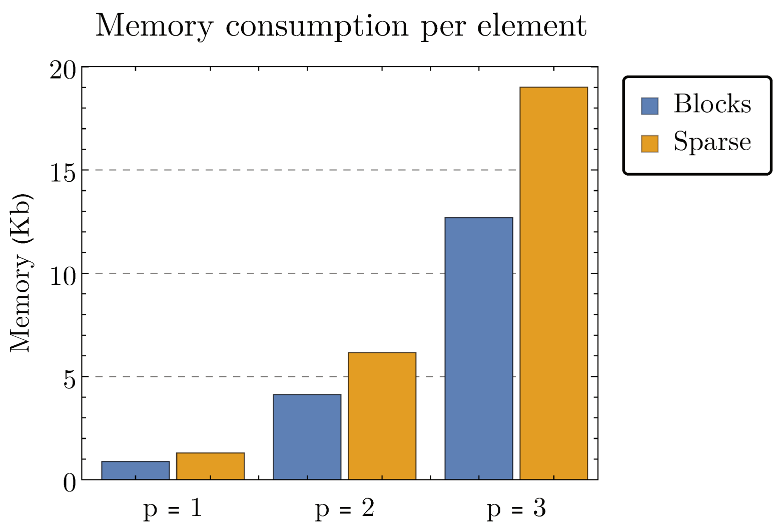

Figure 7.10: Memory consumption per element for sparse and blocks storage patterns.

Figure 7.11 compares the memory consumption for the storage patterns for partition $\mathcal{T}_h|_{l=4}$. It compares the amount of memory required for both storage patterns and also the amount of memory to store the global (assembled) Jacobian matrix in CSR format.

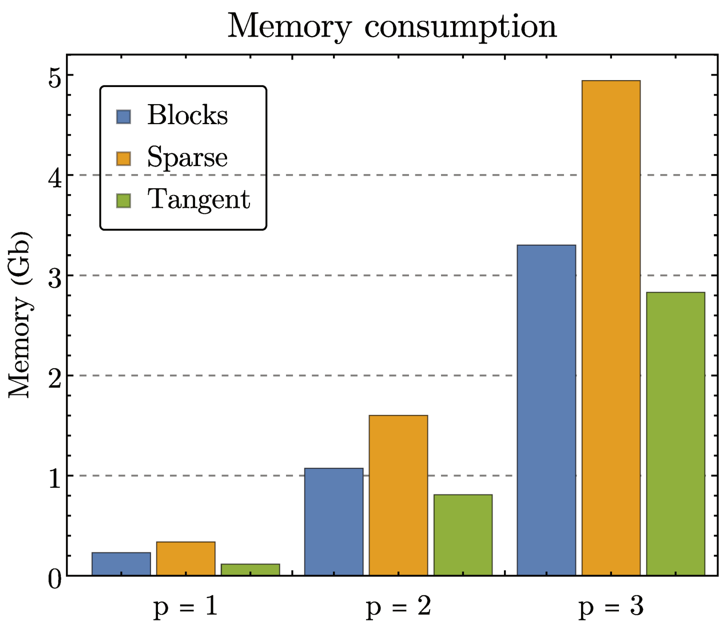

Figure 7.11: Memory consumption for partition $\mathcal{T}_h|_{l=4}$ with $p = \{1, 2, 3\}$.

7.4 Accelerating the elastoplastic convergence

The time for computing the residual of a given solution has been considerably reduced. However, the evaluation of the Jacobian matrix has performance limitations when increasing the polynomial order. Also, the time for approximating the target problem has then been shifted from a problem where the time of the residual computation dominates to a problem where the time for the global system update dominates. Table 7.4 presents the time in seconds of the global system update considering the Jacobian matrix decomposition.

Table 7.4: Solution update with Jacobian matrix decomposition (s).

	$\mathcal{T}_h\|_{l=1}$	$\mathcal{T}_h\|_{l=2}$	$\mathcal{T}_h\|_{l=3}$	$\mathcal{T}_h\|_{l=4}$	$\mathcal{T}_h\|_{l=5}$
$p = 1$	0.007	0.021	0.120	0.601	2.154
$p = 2$	0.018	0.151	3.366	2.500	N/A
$p = 3$	0.217	0.930	3.978	26.393	N/A

Seeking to reduce the time to reach the convergence, it is proposed to use the decomposed global system of the linear elastic problem to update the solution. Thus, the Jacobian matrix is computed and decomposed only once. The iterative method using the elastic stiffness matrix to update the solution is denoted Initial Stiffness Method or Elastic Stiffness Method. The main idea of the Initial Stiffness Method is to construct and approximate the Jacobian matrix as the elastic stiffness matrix to compute the iterative solution. The main advantage of this strategy is that the solution update for a single iteration is fast and stable (See Table 7.5). On the other hand, the rate of convergence for the algorithm is slow, especially when the area with plastic strain is large. Then, the Modified Initial Stiffness Method is applied based on the Thomas acceleration method presented by Thomas [36] and Sloan, Sheng, and Abbo [33] to accelerate the convergence.

Table 7.5: Solution update with no Jacobian matrix decomposition (s).

	$\mathcal{T}_h\|_{l=1}$	$\mathcal{T}_h\|_{l=2}$	$\mathcal{T}_h\|_{l=3}$	$\mathcal{T}_h\|_{l=4}$	$\mathcal{T}_h\|_{l=5}$
$p = 1$	0.003	0.009	0.056	0.232	0.950
$p = 2$	0.008	0.054	0.198	0.891	N/A
$p = 3$	0.021	0.099	0.362	2.488	N/A

Figure 7.12 shows the effect of the Modified Initial Stiffness Method on the convergence against the conventional Initial Stiffness Method for partition $\mathcal{T}_h|_{l=4}$. Fewer iterations were required to reach the same stop criterion. No kind of instability was observed during the iterative solution process. It is important to remark that by increasing the polynomial order, the Initial Stiffness Method suffers from a slow convergence due to the arrow plasticity area around the wellbore region.

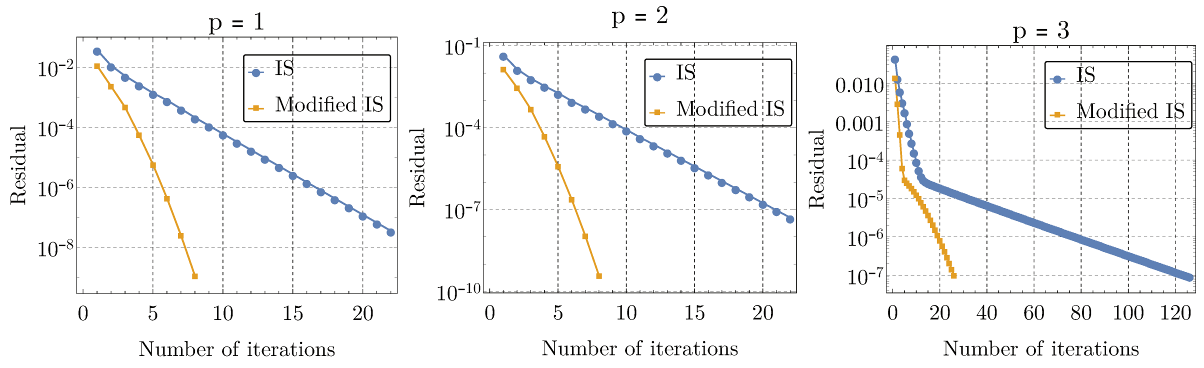

Figure 7.12: Convergence history for the vertical wellbore problem for partition $\mathcal{T}_h|_{l=4}$ with $p = \{1, 2, 3\}$.

Chapter 8

Conclusion

The Finite Element Method is one of the most important numerical techniques to find approximate solutions of partial differential equations. This method has been widely used in many different fields of science and engineering. However, growing market and increasing demand to solve more complex problems efficiently have driven to different approaches such as the manufacturing of faster and smarter processors and multi-core processors or even the use of GPUs for non-graphic applications.

Thus, the research in this thesis presents a strategy for computing the element stiffness matrix and residual vector as well as the assembly process of the global system resulting from the FEM analysis of an elastoplasticity problem using GPU programming. In the FEM analysis, the plastic behavior consists of an iterative process that is evaluated by the evolution of the constitutive equations in a pseudo-time until the problem reaches convergence. Thus, an efficient evaluation of the global system is required since it is performed at each iteration.

A data structure and associated parallel algorithm are developed to accelerate the computation of finite element residual applied to nonlinear elastoplastic problems. The data structure is designed for GPU processors but has shown to accelerate CPU computations as well. Also, it differs from the classical assembly process since constant data is computed only once and is stored globally. The presented structure has a considerable level of parallelism as the algorithms are based on matrix multiplications of highly structured matrices.

The verification of the algorithms is performed with a linear and a nonlinear setting. It is obtained the expected approximation rate in the sense of energy norm for the linear setting. For the nonlinear configuration, a Runge-Kutta solver is implemented. It is noticed a remarkable match between Runge-Kutta and the FEM approximations.

The matrix-vector multiplications for the computation of the residual are implemented using two different storage patterns. The first uses the compressed row storage pattern to perform the operations. The second approach uses an element-wise block-oriented data structure. The first approach performs better on GPU processors, while the second approach performs better when executed in CPU architectures. When compared to the classical residual computation implemented in neoPZ environment, the GPU implementation is 24 times faster for a very refined mesh with cubic polynomial order. The residual computation using the fastest algorithm in the GPU is 5.5 times faster than the fastest CPU implementation for a cubic polynomial approximation applied to the same mesh.

The evaluation of the Jacobian matrix in GPU architectures has better performance for linear polynomial order. For a very refined mesh, it overcomes the CPU's performance by 5.4 times. However, rising the polynomial order results in a bad performance for the GPU since more shared memory is necessary to evaluate the element matrices. For cubic polynomial order, the performance in CPU architectures overcomes GPU's 1.9 times.

Finally, the Modified Initial Stiffness Method is applied to reduce the number of iterations using the linear elasticity stiffness matrix. This method requires the construction of the global Jacobian matrix only once and takes advantage of the inexpensive residual numerical integration process to achieve convergence in a very efficient manner, reducing up to 83% the number of iterations. Also, it reduces the time for the solution update since the matrix is decomposed only once.

Bibliography

[1] Batalha, N. A. “Modelagem linear elástica para simulação do estado de tensão em poços de petróleo inclinados”. Master thesis. University of Campinas, School of Mechanical Engineering and Institute of Geosciences, 2017.

[2] Becker, E. B., Carey, G. F., and Oden, J. T. *Finite Elements, An Introduction: Volume I.* Englewood Cliffs: Prentice-Hall, 1981.

[3] Belytschko, T., Liu, W. K., Moran, B., and Elkhodary, K. *Nonlinear finite elements for continua and structures.* John Wiley & Sons, 2013.

[4] Bhavikatti, S. S. *Finite element analysis.* New Age International, 2005.

[5] Bradley, W. B. “Failure of inclined boreholes”. In: *Journal of Energy Resources Technology* 101.4 (1979), pp. 232–239.

[6] Büttner, J. and Simeon, B. “Runge–Kutta methods in elastoplasticity”. In: *Applied Numerical Mathematics* 41.4 (2002), pp. 443–458.

[7] Cecílio, D. L., Devloo, P. R. B., Gomes, S. M., Santos, E. S. R. dos, and Shauer, N. “An improved numerical integration algorithm for elastoplastic constitutive equations”. In: *Computers and Geotechnics* 64 (2015), pp. 1–9.

[8] Cecka, C., Lew, A. J., and Darve, E. “Assembly of finite element methods on graphics processors”. In: *International journal for numerical methods in engineering* 85.5 (2011), pp. 640–669.

[9] Correa, G. R. and Sulzbach, M. “Programação Paralela em CPU e GPU: Uma avaliação do desempenho das APIs OpenMP, CUDA, OpenCL e OpenACC”. In: *RECeT-Revista de Engenharia, Computação e Tecnologia* 1.1 (2017), pp. 19–24.

[10] Coussy, O. *Poromechanics.* John Wiley & Sons, 2004.

[11] Devloo, P. R. B., Bravo, C. M.A. A., and Rylo, E. C. “Systematic and generic construction of shape functions for p-adaptive meshes of multidimensional finite elements”. In: *Computer Methods in Applied Mechanics and Engineering* 198.21-26 (2009), pp. 1716–1725.

[12] Dziekonski, A., Sypek, P., Lamecki, A., and Mrozowski, M. “Finite element matrix generation on a GPU”. In: *Progress In Electromagnetics Research* 128 (2012), pp. 249–265.

[13] Electrical, T. I. of and Electronic Engineering Inc, T. R. *IEEE Standard for Binary Floating-Point Arithmetic*. 1985.

[14] Farber, R. *CUDA application design and development*. Elsevier, 2011.

[15] Geer, D. “Chip makers turn to multicore processors”. In: *Computer* 38.5 (2005), pp. 11–13.

[16] Gerschgorin, S. “Uber die Abgrenzung der Eigenwerte einer Matrix”. In: *Izvestija Akademii Nauk SSSR, Serija Matematika* 7.3 (1931), pp. 749–754.

[17] Gorder, P. F. “Multicore processors for science and engineering”. In: *Computing in Science & Engineering* 9.2 (2007), p. 3.

[18] He, B., Govindaraju, N. K., Luo, Q., and Smith, B. “Efficient gather and scatter operations on graphics processors”. In: *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. ACM. 2007, p. 46.

[19] Kirk, D. B. and Hwu, W.-M. W. *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann, 2012.

[20] Laouafa, F. and Royis, P. “An iterative algorithm for finite element analysis”. In: *Journal of Computational and Applied Mathematics* 164-165 (2004), pp. 469 –491.

[21] Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. “NVIDIA Tesla: A unified graphics and computing architecture”. In: *IEEE micro* 28.2 (2008), pp. 39–55.

[22] Macioł, P., Płaszewski, P., and Banaś, K. “3D finite element numerical integration on GPUs”. In: *Procedia Computer Science* 1.1 (2010), pp. 1093–1100.

[23] Mafi, R. “GPU-based Parallel Computing for Nonlinear Finite Element Deformation Analysis”. PhD thesis. McMaster University, Department of Electrical and Computer Engineering, 2014.

[24] Micikevicius, P. “3D finite difference computation on GPUs using CUDA”. In: *Proceedings of 2nd workshop on general purpose processing on graphics processing units*. ACM. 2009, pp. 79–84.

[25] Nickolls, J. and Dally, W. J. “The GPU computing era”. In: *IEEE micro* 30.2 (2010), pp. 56–69.

[26] NVIDIA. *cuBLAS Library*. 2019. URL: `https://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf` (visited on 12/02/2019).

[27] NVIDIA. *CUDA C programming model.* 2018. URL: `https://docs.nvidia.com/cuda/archive/9.1/pdf/CUDA_C_Programming_Guide.pdf` (visited on 12/02/2019).

[28] NVIDIA. *cuSPARSE Library.* 2019. URL: `https://docs.nvidia.com/cuda/pdf/CUSPARSE_Library.pdf` (visited on 12/02/2019).

[29] Sanders, J. and Kandrot, E. *CUDA by example: an introduction to general-purpose GPU programming.* Addison-Wesley Professional, 2010.

[30] Santos, E. S. R. "Simulador de meios porosos saturados elastoplásticos". PhD thesis. University of Campinas, School of Civil Engineering, Architecture and Urban Design, 2009.

[31] Sethi, A. and Kushwah, H. "Multicore processor technology-advantages and challenges". In: *International Journal of Research in Engineering and Technology* 4.09 (2015), pp. 87–89.

[32] Shauer, N. "Aproximação Numérica de Propagação de Fraturas Hidráulicas em Domínio Bidimensional com Elastoplasticidade". Master thesis. University of Campinas, School of Civil Engineering, Architecture and Urban Design, 2015.

[33] Sloan, S. W., Sheng, D., and Abbo, A. J. "Accelerated initial stiffness schemes for elastoplasticity". In: *International Journal for Numerical and Analytical Methods in Geomechanics* 24.6 (2000), pp. 579–599.

[34] Souza Neto, E. A. de, Peric, D., and Owen, D. R. J. *Computational methods for plasticity: theory and applications.* John Wiley & Sons, 2011.

[35] Svensson, J., Sheeran, M., and Claessen, K. "Obsidian: A Domain Specific Embedded Language for Parallel Programming of Graphics Processors". In: *Implementation and Application of Functional Languages.* Springer Berlin Heidelberg, 2011, pp. 156–173.

[36] Thomas, J. N. "An improved accelerated initial stress procedure for elasto-plastic finite element analysis". In: *International Journal for Numerical and Analytical Methods in Geomechanics* 8.4 (1984), pp. 359–379.

[37] Thompson, R. B. and Thompson, B. F. *PC hardware in a nutshell: a desktop quick reference.* O'Reilly Media, Inc., 2003.

[38] Wu, E. and Liu, Y. "Emerging technology about GPGPU". In: *APCCAS 2008 - 2008 IEEE Asia Pacific Conference on Circuits and Systems.* Institute of Electrical and Electronics Engineers (IEEE), 2008, pp. 618–622.

[39] Zhang, J. and Shen, D. "GPU-based implementation of finite element method for elasticity using CUDA". In: *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*. Institute of Electrical and Electronics Engineers (IEEE), 2013, pp. 1003–1008.

[40] Zienkiewicz, O. C. *The finite element method*. Vol. 36. McGraw-hill London, 1977.

Appendix A

Voigt notation

The Voigt notation is used to transform second-order symmetric tensors to vectors and forth-order symmetric tensors to square matrices. In Voigt notation the stress tensor $\boldsymbol{\sigma}$ is mapped to [3]:

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} \\ \sigma_{yx} & \sigma_{yy} \end{bmatrix} \rightarrow \vec{\sigma} = \begin{pmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{xy} \end{pmatrix} \quad \text{(A.1)} \quad \boldsymbol{\sigma} = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{xy} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{xz} & \sigma_{yz} & \sigma_{zz} \end{bmatrix} \rightarrow \vec{\sigma} = \begin{pmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{zz} \\ \sigma_{xy} \\ \sigma_{yz} \\ \sigma_{zx} \end{pmatrix} \quad \text{(A.2)}$$

for two and three-dimensional stress tensors, respectively. Similarly, the strain tensor $\boldsymbol{\varepsilon}$ is redefined in Voigt notation as:

$$\boldsymbol{\varepsilon} = \begin{bmatrix} \varepsilon_{xx} & \varepsilon_{xy} \\ \varepsilon_{yx} & \varepsilon_{yy} \end{bmatrix} \rightarrow \vec{\varepsilon} = \begin{pmatrix} \varepsilon_{xx} \\ \varepsilon_{yy} \\ 2\varepsilon_{xy} \end{pmatrix} \quad \text{(A.3)} \quad \boldsymbol{\varepsilon} = \begin{bmatrix} \varepsilon_{xx} & \varepsilon_{xy} & \varepsilon_{xz} \\ \varepsilon_{xy} & \varepsilon_{yy} & \varepsilon_{yz} \\ \varepsilon_{xz} & \varepsilon_{yz} & \varepsilon_{zz} \end{bmatrix} \rightarrow \vec{\varepsilon} = \begin{pmatrix} \varepsilon_{xx} \\ \varepsilon_{yy} \\ \varepsilon_{zz} \\ 2\varepsilon_{xy} \\ 2\varepsilon_{yz} \\ 2\varepsilon_{zx} \end{pmatrix} \quad \text{(A.4)}$$

for two and three-dimensional strain tensors, respectively. The factor of 2 on the shear strains results from the requirement that the expressions for the energy be equivalent in Voigt notation and indicial notation.

Appendix B

Spectral decomposition

B.1 Eigenvalues

The first step to evaluate the eigenvalues of the stress tensor in return mapping is the normalization of stress tensor components. After that, the theorem presented by Gerschgorin [16] is used to calculate the interval where the eigenvalues of the stress tensor are. Gershgorin's Theorem states that given a matrix $\mathbf{S}_{nxn}$, all eigenvalues of $\mathbf{S}$ lies in the union of the closed interval:

$$\left[s_{ii} - \sum_{j \neq i} |s_{ij}|\,, s_{ii} + \sum_{j \neq i} |s_{ij}| \right], \quad i, j = 1, ..., n. \tag{B.1}$$

Thus, it is possible to use Newton's method to compute the maximum and minimum roots of the characteristic polynomial of the stress tensor using the interval evaluated by Gershgorin's theorem as initial guesses. This computation corresponds to the eigenvalues λ_1 and λ_3 of the stress tensor. The intermediate eigenvalue λ_2 is evaluated using the following algebraic relationship:

$$tr(\mathbf{S}) = \lambda_1 + \lambda_2 + \lambda_3 \tag{B.2}$$

where $\mathbf{S}$ is the stress tensor and λ_i corresponds to its eigenvalues.

B.2 Eigenvectors

Given a symmetric matrix with eigenvalues and multiplicity known, the eigenvectors problem is stated by:

$$\mathbf{S} \cdot \vec{v} = \lambda \vec{v} \tag{B.3}$$

A scheme for eigenvectors calculation is defined using the matrix multiplicity.

Multiplicity 1

If the multiplicity is unitary, one can extract from the matrix $\mathbf{S} - \lambda\mathbf{I}$ a non-singular 2x2 matrix. It is possible to consider three configurations:

$$\begin{bmatrix} x_{11} & x_{12} & y_1 \\ x_{21} & x_{22} & y_2 \\ a & b & c \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} x_{11} & y_1 & x_{12} \\ a & b & c \\ x_{21} & y_2 & x_{22} \end{bmatrix} \begin{bmatrix} v_1 \\ 1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} a & b & c \\ y_1 & x_{11} & x_{12} \\ y_2 & x_{21} & x_{22} \end{bmatrix} \begin{bmatrix} 1 \\ v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Then it is chosen the configuration for which $|det\ \mathbf{X}|$ is maximum where:

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} \tag{B.4}$$

The values v_1 and v_2 are computed as follows:

$$\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = -\mathbf{X}^{-1} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \tag{B.5}$$

Multiplicity 2

In the case multiplicity is 2 the rank of $\mathbf{S} - \lambda\mathbf{I}$ is unitary and there must be at least one diagonal other than zero. It is also possible to consider three configurations:

$$\begin{bmatrix} x_{11} & y_2 & y_2 \\ a & b & c \\ d & e & f \end{bmatrix} \begin{bmatrix} v_1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} a & b & c \\ y_1 & x_{11} & y_2 \\ d & e & f \end{bmatrix} \begin{bmatrix} 1 \\ v_1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ y_1 & y_2 & x_{11} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ v_1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

and equivalently:

$$\begin{bmatrix} x_{11} & y_2 & y_2 \\ a & b & c \\ d & e & f \end{bmatrix} \begin{bmatrix} v_2 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} a & b & c \\ y_1 & x_{11} & y_2 \\ d & e & f \end{bmatrix} \begin{bmatrix} 0 \\ v_2 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ y_1 & y_2 & x_{11} \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

It is chosen the configuration for which $|x_{11}|$ is maximum. Then it is defined, for instance:

$$\vec{v_1} = \begin{bmatrix} v_1 \\ 1 \\ 0 \end{bmatrix} \qquad \vec{v_2} = \begin{bmatrix} v_2 \\ 0 \\ 1 \end{bmatrix} \tag{B.6}$$

Finally, $\vec{v_1}$ and $\vec{v_2}$ are normalized:

$$\tilde{v}_1 = \frac{\vec{v_1}}{||\vec{v_1}||} \qquad \tilde{v}_2 = \frac{\vec{v_2}}{||\vec{v_2}||} \tag{B.7}$$

Multiplicity 3

In this case the matrix is diagonal and the eigenvectors are the identity matrix.

Appendix C

Results variability

GPU residual (sparse matrix st. pat.): Mean and standard deviation					
	$\mathcal{T}_h\|_{l=1}$	$\mathcal{T}_h\|_{l=2}$	$\mathcal{T}_h\|_{l=3}$	$\mathcal{T}_h\|_{l=4}$	$\mathcal{T}_h\|_{l=5}$
$p = 1$	0.003s - 1.997%	0.005s - 1.902%	0.009s - 2.299%	0.025s - 2.856%	0.085s - 0.098%
$p = 2$	0.005s - 1.025%	0.009s - 0.834%	0.021s - 1.821%	0.071s - 0.284%	N/A
$p = 3$	0.006s - 2.690%	0.012s - 1.156%	0.036s - 1.186%	0.130s - 3.373%	N/A

GPU residual (block matrices st. pat.): Mean and standard deviation					
	$\mathcal{T}_h\|_{l=1}$	$\mathcal{T}_h\|_{l=2}$	$\mathcal{T}_h\|_{l=3}$	$\mathcal{T}_h\|_{l=4}$	$\mathcal{T}_h\|_{l=5}$
$p = 1$	0.002s - 1.446%	0.007s - 0.862%	0.020s - 0.879%	0.063s - 0.174%	0.231s - 0.306%
$p = 2$	0.006s - 2.692%	0.025s - 0.884%	0.087s - 1.753%	0.333s - 0.046%	N/A
$p = 3$	0.014s - 0.332%	0.074s - 1.384%	0.276s - 0.284%	1.045s - 0.101%	N/A

CPU residual (sparse matrix st. pat.): Mean and standard deviation					
	$\mathcal{T}_h\|_{l=1}$	$\mathcal{T}_h\|_{l=2}$	$\mathcal{T}_h\|_{l=3}$	$\mathcal{T}_h\|_{l=4}$	$\mathcal{T}_h\|_{l=5}$
$p = 1$	0.004s - 1.409%	0.015s - 1.741%	0.057s - 0.635%	0.215s - 1.147%	0.719s - 2.016%
$p = 2$	0.011s - 1.147%	0.045s - 1.106%	0.149s - 1.087%	0.543s - 2.398%	N/A
$p = 3$	0.026s - 1.001%	0.099s - 0.934%	0.291s - 0.753%	1.126s - 0.695%	N/A

CPU residual (block matrices st. pat.): Mean and standard deviation					
	$\mathcal{T}_h\|_{l=1}$	$\mathcal{T}_h\|_{l=2}$	$\mathcal{T}_h\|_{l=3}$	$\mathcal{T}_h\|_{l=4}$	$\mathcal{T}_h\|_{l=5}$
$p = 1$	0.004s - 5.525%	0.011s - 2.013%	0.042s - 2.389%	0.151s - 1.734%	0.544s - 1.077%
$p = 2$	0.006s - 6.534%	0.025s - 3.993%	0.103s - 2.967%	0.344s - 1.332%	N/A
$p = 3$	0.012s - 5.232%	0.047s - 3.890%	0.178s - 3.984%	0.714s - 3.474%	N/A

neoPZ residual: Mean and standard deviation					
	$\mathcal{T}_h\|_{l=1}$	$\mathcal{T}_h\|_{l=2}$	$\mathcal{T}_h\|_{l=3}$	$\mathcal{T}_h\|_{l=4}$	$\mathcal{T}_h\|_{l=5}$
$p = 1$	0.072s - 8.523%	0.191s - 5.629%	0.721s - 2.489%	2.851s - 4.260%	12.140s - 1.202%
$p = 2$	0.090s - 2.595%	0.369s - 4.548%	0.824s - 1.108%	2.871s - 1.993%	N/A
$p = 3$	0.121s - 5.374%	0.329s - 1.048%	0.818s - 0.842%	3.088s - 0.779%	N/A

GPU Jacobian Matrix: Mean and standard deviation

	$\mathcal{T}_h\|_{l=1}$	$\mathcal{T}_h\|_{l=2}$	$\mathcal{T}_h\|_{l=3}$	$\mathcal{T}_h\|_{l=4}$	$\mathcal{T}_h\|_{l=5}$
$p = 1$	0.001s - 1.907%	0.007s - 3.080%	0.017s - 2.036%	0.062s - 2.623%	0.245s - 2.681%
$p = 2$	0.016s - 2.368%	0.047s - 2.040%	0.180s - 0.738%	0.721s - 0.339%	N/A
$p = 3$	0.101s - 0.190%	0.367s - 0.695%	1.468s - 0.337%	5.858s - 0.138%	N/A

CPU Jacobian Matrix: Mean and standard deviation

	$\mathcal{T}_h\|_{l=1}$	$\mathcal{T}_h\|_{l=2}$	$\mathcal{T}_h\|_{l=3}$	$\mathcal{T}_h\|_{l=4}$	$\mathcal{T}_h\|_{l=5}$
$p = 1$	0.005s - 6.354%	0.016s - 7.831%	0.057s - 1.349%	0.259s - 2.375%	1.046s - 1.167%
$p = 2$	0.013s - 2.896%	0.051s - 3.166%	0.225s - 1.725%	0.920s - 0.998%	N/A
$p = 3$	0.031s - 4.692%	0.146s - 2.233%	0.625s - 0.7275%	2.677s - 1.174%	N/A

neoPZ Jacobian Matrix: Mean and standard deviation

	$\mathcal{T}_h\|_{l=1}$	$\mathcal{T}_h\|_{l=2}$	$\mathcal{T}_h\|_{l=3}$	$\mathcal{T}_h\|_{l=4}$	$\mathcal{T}_h\|_{l=5}$
$p = 1$	0.026s - 6.755%	0.157s - 4.012%	0.421s - 1.160%	1.369s - 1.603%	8.820s - 3.078%
$p = 2$	0.095s - 6.796%	0.190s - 1.141%	0.722s - 1.397%	3.238s - 1.982%	N/A
$p = 3$	0.259s - 2.318%	0.654s - 1.067%	1.998s - 2.663%	9.609s - 2.575%	N/A