UNIVERSIDADE FEDERAL FLUMINENSE

PEDRO CORTEZ FETTER LOPES

Massively Parallel Implementations of the Preconditioned Conjugate Gradient Method Applied to Image-based Numerical Homogenization with the Assembly-free Finite Element Method

> Niterói 2021

UNIVERSIDADE FEDERAL FLUMINENSE

PEDRO CORTEZ FETTER LOPES

Massively Parallel Implementations of the Preconditioned Conjugate Gradient Method Applied to Image-based Numerical Homogenization with the Assembly-free Finite Element Method

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Ciência da Computação.

Orientador: André Maués Brabo Pereira

> Co-orientador: Ricardo Leiderman

> > Niterói 2021

Ficha catalográfica automática - SDC/BEE Gerada com informações fornecidas pelo autor

L864m Lopes, Pedro Cortez Fetter Massively Parallel Implementations of the Preconditioned Conjugate Gradient Method Applied to Image-based Numerical Homogenization with the Assembly-free Finite Element Method / Pedro Cortez Fetter Lopes ; André Maués Brabo Pereira, orientador ; Ricardo Leiderman, coorientador. Niterói, 2021. 125 f. Dissertação (mestrado)-Universidade Federal Fluminense, Niterói, 2021. DOI: http://dx.doi.org/10.22409/PGC.2021.m.15789890784 1. Computação paralela. 2. Unidade de processamento gráfico. 3. Métodos numéricos. 4. Método dos Elementos Finitos. 5. Produção intelectual. I. Pereira, André Maués Brabo, orientador. II. Leiderman, Ricardo, coorientador. III. Universidade Federal Fluminense. Instituto de Computação. IV. Título. CDD -

Bibliotecário responsável: Debora do Nascimento - CRB7/6368

PEDRO CORTEZ FETTER LOPES

Massively Parallel Implementations of the Preconditioned Conjugate Gradient Method Applied to Image-based Numerical Homogenization with the Assembly-free Finite Element Method

> Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Ciência da Computação.

Aprovada em Agosto de 2021.

BANCA EXAMINADORA André Maris Brabo Perena

Prof. Dr. André Maués Brabo Pereira - Orientador, UFF Licardo /

Prof. Dr. Ricardo Leiderman - Co-orientador, UFF

Prof. Dr. Esteban Walter Gonzalez Clua, UFF

Waldeman life Rillio

Prof. Dr. Waldemar Celes Filho, PUC-Rio



Prof. Dr. Luiz Fernando Campos Ramos Martha, PUC-Rio

Niterói 2021

"Can you improve this place with the data that you gather?" Brett Gurewitz

Agradecimentos

Primeiramente, agradeço à minha namorada, Ana Carolina, por todo apoio e companheirismo. Passamos por 2020 juntos, nenhum desafio é grande demais para nós dois. Sem você, não sei o que seria de mim, e certamente não haveria este trabalho. Muito obrigado por tudo.

Agradeço à toda minha família, pelo suporte e pela educação que puderam me proporcionar. Em especial, sou grato aos meus pais, Marcos e Marcella, e à minha irmã, Isabel, que estão sempre ao meu lado.

Gostaria também de agradecer aos meus orientadores. Ao professor André, que me guia desde a graduação, e ao professor Ricardo, que me providencia ensinamentos importantes. Ambos contribuíram significativamente para a evolução dos meus estudos.

Agradeço aos membros da banca, por terem aceitado fazer parte da defesa, e pelas sugestões que contribuíram para a versão final do texto.

Sou grato à Universidade Federal Fluminense, ao Instituto de Computação, e a todos os professores da Pós com quem tive contato. Foi um prazer assistir às aulas, todos somaram ao meu desenvolvimento como aluno e pesquisador. Especialmente, agradeço ao professor Esteban pelo curso Arquitetura e Programação de GPUs, que muito me auxiliou na confecção deste trabalho.

Agradeço a todos os meus amigos, desde a época de escola, à graduação, aos meus atuais colegas de laboratório. Seja para ajudar com estudos e trabalhos, ou na hora de descontrair, todos são importantes para mim. Em especial, sou grato aos meus amigos de infância, Bernardo e Patrick. Vocês me aturam desde quando eu aprendia a resolver uma equação, ou a falar inglês. Nada mais justo que uma menção especial agora.

Por último, mas de forma alguma menos importante, agradeço ao meus amigos Rafael e professor Martha, com os quais trabalhei previamente ao meu ingresso no Mestrado. Ambos me ajudaram muito nos meus primeiros passos como pesquisador.

Resumo

O Método dos Elementos Finitos (MEF) é comumente empregado para resolver as equações governantes de vários fenômenos físicos no contexto da homogeneização numérica. Neste escopo, são utilizadas técnicas de imageamento, como a micro tomografia computadorizada, para obter modelos digitais da microescala de amostras de materiais, o que naturalmente leva a soluções baseadas em pixel e voxel. A medida que a dimensão das imagens cresce, a alocação de memória por conta da matriz global de elementos finitos, mesmo na forma esparsa, torna-se rapidamente inviável, o que dificulta a exploração dos recursos de imageamento de última geração. As estratégias "sem montagem" baseiam-se na premissa de nunca armazenar a matriz global, trabalhando apenas com as matrizes de elementos, o que alivia consideravelmente o uso de memória, mas eleva o custo computacional. Consequentemente, abordagens otimizadas de implementação são buscadas para reduzir o tempo de execução das soluções computacionais. O método dos gradientes conjugados pré-condicionado (GCP) é usado para resolver os sistemas lineares de equações algébricas, que são esparsos e de larga escala. Este trabalho foca em solvers GCP massivamente paralelos, implementados em CUDA C, aplicados ao MEF "sem montagem" para a homogeneização numérica de condutividade térmica e elasticidade de modelos baseados em imagens. São apresentadas cinco maneiras diferentes de implementar o método GCP em GPU, algumas mais eficientes em memória, outras em tempo. Todas apresentam desempenho significativamente melhor do que soluções semelhantes em CPU. Além disso, tanto quanto é do conhecimento do autor, é proposta uma nova estratégia para a obtenção de bons pontos de partida para o processo iterativo do método GCP, no âmbito das simulações baseadas em imagens. Os solvers resultantes são validados com um benchmark analítico, e pela verificação dos resultados obtidos para um modelo microtomográfico de uma amostra de ferro fundido, comparando com valores experimentais encontrados na literatura. Métricas de tempo e memória são apresentadas e discutidas. Mostra-se que as metodologias desenvolvidas permitem que simulações com cerca de 475 milhões de graus de liberdade sejam realizadas em computadores pessoais equipados com dispositivos habilitados para CUDA, levando menos de um minuto por solução do GCP. As soluções obtidas com o GCP em GPU mostraram-se até cerca de 400x mais rápidas que as alcançadas com um programa antecessor paralelizado em CPU.

Palavras-chave: elementos finitos, sem montagem, gradientes conjugados, GPU, CUDA, homogeneização baseada em imagens.

Abstract

The Finite Element Method (FEM) is commonly employed to solve the governing equations of various physical phenomena in the context of numerical homogenization. In this scope, imaging techniques, such as micro-computed tomography, are used to obtain digital models of the micro-scale of material samples, which naturally leads to pixel and voxel-based solutions. As the dimension of the images increases, memory allocation due to the finite element global matrix quickly becomes unfeasible, even in sparse form, making it harder to fully explore state-of-the-art imaging resources. Assembly-free strategies are based on the premise of never storing the global matrix, working with local element matrices instead, which considerably relieves memory usage, but increases computational cost. Hence, optimized implementation approaches are sought out to reduce runtime. The Preconditioned Conjugate Gradient (PCG) method is used to solve the large-scale sparse linear systems of algebraic equations. This work focuses on massively parallel PCG solvers, implemented in CUDA C, applied to the assembly-free FEM for the numerical homogenization of thermal conductivity and elasticity of image-based models. Five different ways of implementing the PCG method in GPU are presented, some are focused on memory efficiency, others prioritize runtime reduction. All of these implementations perform significantly better than akin solutions in CPU. Furthermore, to the best of the author's knowledge, a novel strategy for obtaining good initial guesses for the PCG method is proposed, in the scope of image-based simulations. The resulting solvers are validated with an analytical benchmark, and by comparing the obtained results for a microtomographic model of a cast iron sample against experimental values found in the literature. Time and memory metrics are presented and discussed. It is shown that the developed methodologies allow for simulations with nearly 475 million degrees-of-freedom to be conducted in personal computers equipped with CUDA-enabled devices, taking less than one minute per system solution with the PCG method. The solutions obtained with the PCG in GPU were up to 400x faster than those achieved with a predecessor program, parallelized in CPU.

Keywords: FEM, assembly-free, matrix-free, PCG, GPU, CUDA, image-based, homo-genization.

List of Figures

1.1	Digital model of a sandstone sample obtained wit μ CT. Source: Vianna et al. [76].	4
3.1	Periodic cell	12
3.2	(a) Periodic boundary conditions, source: Vianna [75]. (b) Convergence of effective properties for different boundary condition considerations, adapted from Nguyen et al. [57] and Sapucaia [70]	13
3.3	(a) Continuous domain Ω , (b) Discretization into a finite element mesh	19
3.4	Bilinear shape functions for a 1x1 quadrilateral finite element in 2D \ldots	19
3.5	8-bit grayscale representation of a 5x5 pixel-based finite element mesh	24
3.6	Periodic node (black) and element (red) numbering on a 2D image-based mesh	25
3.7	Periodic node (left) and element (right) numbering on a 3D image-based mesh	26
4.1	Convergence path of (a) steepest descent and (b) conjugate gradient methods for the minimization of a quadratic form, similar to Equation 4.2. Adapted from Shewchuk [72]	30
4.2	Element-by-element spreading of local coefficients to global DOFs	35
4.3	Concurrency on element-by-element matrix-vector product	35
4.4	Non-conflicting access of local nodes on a pixel-based mesh (left), local node numbering (right)	36
4.5	Node-by-node gathering of local coefficients to global DOFs	38
4.6	Peak Memory Allocation [MB] vs. number of DOFs, comparing vectorized assembly-free to stored matrix solutions using the PCG method	45

4.7	Time per PCG iteration [s] vs. number of DOFs, comparing vectorized assembly-free to stored matrix solutions using the PCG method \ldots .	46
4.8	Total elapsed time [s] vs. number of DOFs, comparing vectorized and sequential assembly-free solutions	46
5.1	CPU vs. GPU architecture. Based on Kirk and Hwu [38]	50
5.2	Relation of threads, blocks and grids to hardware. Source: Gupta [27]	53
5.3	Usual flow of activities of a CUDA C program	53
5.4	Memory hierarchy of a CUDA-enabled device. Based on Gupta [27]	54
5.5	Coalesced (top) and uncoalesced (bottom) memory access of an array with entries of 4 Bytes	55
5.6	16-bit node material map	59
5.7	API calls to manage the data flow in the MParPCG solver. Memory allocation and transfer to device in green, call to iterative solver in blue, freeing memory in red	62
6.1	(a) 200x200 pixel-based model of a sandstone sample [76], (b) coarsened 100x100 pixel mesh	70
6.2	$400\mathrm{x}400$ synthetic model for tests with initial guesses from coarse meshes $% 100\mathrm{x}40\mathrm{x}40\mathrm$	72
6.3	Dimensionless norms of residuals vs. number of iterations	73
7.1	Analytical benchmark	76
7.2	Mean time per PCG iteration [s] vs number of DOFs, for the analytical benchmark	77
7.3	200x200x200 voxel-based representation of the cast iron sample, consisting of graphite nodules within a ferritic matrix	78
7.4	Mean time per PCG iteration [s] vs number of DOFs, for thermal conduc- tivity analysis of the cast iron sample	81
7.5	Mean time per PCG iteration [s] vs number of DOFs, for elasticity analysis of the cast iron sample	83
7.6	Allocated memory [MB] vs number of DOFs, for thermal conductivity analysis	84
7.7	Allocated memory [MB] vs number of DOFs, for elasticity analysis	84

7.8	Total time $[s]$ vs number of DOFs, for thermal conductivity analysis $\ . \ . \ .$	85
7.9	Total time [s] vs number of DOFs, for elasticity analysis	85
7.10	Elapsed time for PCG solution [s] vs. number of recursive searches for initial guesses	87
7.11	Elapsed time for PCG solution [s] vs. DOFs, comparing $\mathbf{x}_0 = \vec{0}$ to two recursive searches for an initial guess.	88
7.12	(a) Synthetic voxel-based model generated with perlin noise, (b) details of the blue phase	89
7.13	Useful part of a sample imaged via μ CT with 1024 ³ voxels	91
C.1	(a) Pixel-based and (b) voxel-based finite elements.	103

List of Tables

2.1	Performance metrics found in the literature for massively parallel PCG	0
	solvers in GPU	9
3.1	Operations to walk on nodes of an image-based mesh	26
3.2	Data required by operations on Table 3.1	26
4.1	Memory allocated for parallel assembly-free PCG solver in CPU $\ . \ . \ .$.	48
5.1	Memory allocated in host and device for CUDA implementation of the MParPCG solver	61
5.2	Estimates of memory allocation in the device for PCG solvers with the assembly-free approach and assembled matrices in CSR format, for 3D linear elasticity simulations	61
F 0		01
5.3	Summary of the PCG solvers implemented in GPU	67
6.1	Convergence metrics for preliminary tests with initial guesses from coarse meshes	72
7.1	Specifications of the computer employed in this work	75
7.2	Results for analytical benchmark	76
7.3	Time metrics for analytical benchmark	76
7.4	Physical properties of the micro-scale phases in the cast iron sample	78
7.5	Results for orthotropic thermal conductivity of the cast iron sample \ldots .	79
7.6	Comparison of the obtained orthotropic Young's moduli with the results of Pereira et al. [61]	79
7.7	Results for elasticity constitutive tensors of the cast iron sample $\ldots \ldots$	80
7.8	Time metrics for thermal conductivity analysis of the cast iron sample	81
7.9	Time metrics for elasticity analysis of the cast iron sample \ldots .	82

7.10	Time metrics for the homogenization of elasticity of the 400^3 voxels cast	0.9
	iron sample, comparing to a previous implementation in CPU [61]	83
7.11	Time and memory metrics for tests with all implemented solvers in GPU, considering initial guesses for the PCG method	86
7.12	Metrics for thermal conductivity analyses of a synthetic model at nearly full GPU memory capacity with different solvers, using a desktop computer	90
7.13	Metrics for elasticity analyses of a synthetic model at nearly full GPU	
	memory capacity with different solvers, using a desktop computer	90
7.14	Specifications of the laptop used in the final experiment	91
7.15	Metrics for thermal conductivity analyses of a 724^3 voxels synthetic model	
	with a laptop	92

List of Abbreviations and Acronyms

FEM	:	Finite Element Method;
$\rm FE$:	Finite Element;
DOF	:	Degree-of-freedom;
PCG	:	Preconditioned Conjugate Gradient;
PDE	:	Partial Differential Equation;
μCT	:	Micro-Computed Tomography;
CPU	:	Central Processing Unit;
GPU	:	Graphics Processing Unit;
CUDA	:	Compute Unified Device Architecture;
SM	:	Streaming Multiprocessor;
SIMT	:	Single Instruction Multiple Threads;

Contents

1	Intr	oduction	1
	1.1	Motivation	3
	1.2	Goals	5
	1.3	Outline	5
2	Lite	rature Review	6
	2.1	Numerical Homogenization	3
	2.2	Assembly-free FEM	7
	2.3	Massively Parallel PCG in GPU	3
3	Nun	nerical Homogenization 1	1
	3.1	An introduction	1
	3.2	Governing equations	3
		3.2.1 Heat Conduction $\ldots \ldots 14$	4
		3.2.2 Elasticity $\ldots \ldots \ldots$	5
	3.3	Finite Element Method	3
		3.3.1 A brief introduction \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots 18	3
		3.3.2 Heat Conduction)
		3.3.3 Elasticity	1
		3.3.4 Assembly-free FEM	2
	3.4	Image-based Finite Element models	3
	3.5	An algorithm for image-based homogenization with the FEM	7

4	Prec	conditio	oned Conjugate Gradient Method	29					
	4.1	The PCG algorithm							
	4.2	Vector	r operations	33					
	4.3	Matri	x operations	34					
		4.3.1	Element-by-element	34					
		4.3.2	Node-by-node	37					
	4.4	Massi	vely parallel PCG for assembly-free FEM	40					
	4.5	Parall	el PCG solver in CPU	41					
		4.5.1	A vectorized approach	41					
		4.5.2	Memory vs. Time trade-off	44					
		4.5.3	Multi-thread implementation	47					
5	Mas	sively p	parallel implementation in GPU	49					
	5.1	Gener	al purpose programming with GPUs	49					
		5.1.1	An overview	50					
		5.1.2	CUDA programming model	51					
		5.1.3	GPU memory	52					
	5.2	PCG	applied to assembly-free image-based FEM in GPU	55					
		5.2.1	Kernels for assembly-free matrix operations	56					
		5.2.2	Memory allocation	60					
		5.2.3	Data transfer between host and device	61					
	5.3	Alterr	native strategies of implementation	63					
		5.3.1	xrd solver	63					
		5.3.2	xsd solver	64					
		5.3.3	rd solver	64					
		5.3.4	sd solver	66					
	5.4	Summ	nary of the implementations	66					

6	Find	ling ini	tial guesses in coarse meshes	68					
	6.1	$Hypothesis \dots $							
	6.2	Coarsening of image-based meshes							
	6.3	A recu	ursive algorithm for the search of initial guesses $\ldots \ldots \ldots \ldots$	70					
	6.4	A vali	dation test	71					
7	Rest	ults		74					
	7.1	Analy	tical benchmark	75					
	7.2	Cast i	ron sample	77					
		7.2.1	Homogenized physical properties	78					
		7.2.2	Metrics	80					
			7.2.2.1 Element-by-element vs. Node-by-node	80					
			7.2.2.2 Alternative implementations and initial guesses	86					
	7.3	Larges	st possible synthetic sample	88					
		7.3.1	Desktop computer	89					
		7.3.2	Laptop	90					
8	Con	clusion		93					
Re	feren	ices		95					
Ap	opend	lix A –	Isotropic materials	101					
Ap	opend	lix B –	Orthotropic materials	102					
Ap	pend	lix C –	Analytical solutions for local FE matrices	103					

Chapter 1

Introduction

Numerical homogenization is a practice within Materials Science dedicated to estimating the effective physical properties of heterogeneous materials via numerical experiments on their micro-scale. The idea is to solve the differential equations that govern a given physical phenomenon, while considering known forcing applied to either the whole domain or its borders, so that constitutive tensors can be evaluated for a macro-scale representative volume, as it were homogeneous. This is useful for various fields of science, as it allows for the study of important properties, such as thermal conductivity [41, 77], elasticity [2, 42, 61, 71, 75], permeability [2, 76] and electric resistivity [20], to cite a few.

The computational approach is an alternative to study materials without depending on physical tests, that often are destructive, non-repeatable, and fail to provide detailed results on all directions. For example, elasticity tests on a laboratory usually involve forcing the sample to its rupture, and permeability rehearsals are focused on only one direction of fluid flow. The full control of input parameters, the repeatability of analyses, and the possibilities to run simulations on all directions of interest and to generate detailed results are attractive arguments for digital testing. However, a crucial matter needs to be addressed for it to be reliable, the characterization of the model. In light of this, imaging techniques have been brought into play, technologies like micro-computed tomography (μ CT) are increasingly being employed to obtain digital models of material samples. Pereira et al. [61], Vianna [75], Sapucaia [70], Wu et al. [77], and Liu et al.[42] are some examples of image-based numerical homogenization with μ CT. The adoption of images to depict the micro-scale leads to the search for pixel and voxel-based solvers.

The Finite Element Method (FEM) is one of the most commonly employed numerical scheme to solve the differential equations related to the considered physical phenomena [2, 5, 9, 36, 41, 42, 61, 65, 71, 76, 77], generally boundary value problems described

by elliptic PDEs, although it is not the only option [45, 70]. In a nutshell, the FEM numerically characterizes the problem as a system of algebraic equations, to be solved with computational algebra techniques, for an approximate solution of the PDE in a discretized domain, denominated mesh. The standard strategies of implementation usually demand costly memory allocation, due to the global matrix, that holds the coefficients of the system of equations, and auxiliary data structures, such as a connectivity map. However, when dealing with images, that is, meshes structured in grid-like fashion, it is possible to considerably relief memory usage, as the regularity of the geometry grants sparsity to the global matrix, and allows for the connectivity to be computed on-the-fly.

A vital aspect of the FEM is the need for convergence tests. To achieve a reliable numerical solution, it is important that the same model is analyzed more than once, refining the mesh at each simulation. With image-based models, this means that it is necessary to run analyses on images with increasing resolution. Similarly, from a qualitative perspective in the context of homogenization, it is desirable to consider different cells, as big as possible, from a same material sample, to obtain trustworthy effective properties. Coupling the needs for mesh refinement and big models leads to large-scale systems of equations to be solved. For example, in 3D analyses, it is common to quickly go over 10^6 degrees-of-freedom (DOFs). This makes the storage of the global matrix unfeasible, even in sparse form, as most algorithms that do so have some sort of super-linear complexity of space. At best, efficient implementations, constrained by the assumption of structured meshes, might achieve linear tendency of memory usage, but with a high slope coefficient, so there is still high allocation demand. Available memory is a critical issue for the intended simulations. Furthermore, this is the bottleneck of the analyses. For most models, the time it takes to solve the systems of equations determines the total time of analysis, even though there are pre- and post-processing tasks.

The assembly-free, also called matrix-free, approach is based on the premise of never allocating the global matrix in memory. Nevertheless, the system of equations still conceptually exists and needs to be solved. The idea is that each time a coefficient of the matrix is requested, instead of simply performing a memory access, it must be computed on the spot, considering a stencil formed by the respective DOF and its neighbors on the mesh. This is becoming a standard practice for large-scale finite element analysis [4, 5, 24, 36, 37, 39, 47, 48, 49, 54, 56, 75]. Strategies of this sort can drastically reduce memory allocation, but they also imply that matrix operations become dependent of sweeps of the domain, meaning that computational cost is elevated. Hence, parallel implementations are sought. The Preconditioned Conjugate Gradient Method (PCG) [30, 32, 40, 72] is an iterative numerical scheme commonly employed to solve large-scale systems of equations, particularly for those from the FEM applied to elliptic PDEs, as the characteristics of the global matrix usually are favorable for fast convergence, in number of iterations [1, 4, 5, 18, 19, 23, 26, 36, 37, 56, 63, 64]. It requires significantly less memory allocation than any conventional direct method, as no factored matrix needs to be stored. Also, most of the operations on each iteration are based on vectors, so they are relatively straightforward to be implemented in massively parallel environments. Even so, there are some matrix operations that require a careful assessment in parallel solutions, especially considering an assembly-free approach.

Over the last decades, it has been observed a significant improvement in the computational power of Graphics Processing Units (GPUs) [38, 69]. So much so that, in recent years, they have become a recurring topic for academic work in Scientific Computing, mostly as a tool to speed up numerical simulations. Akbariyeh [1], Apostolou [4], Daibes [17], Madeira [44], Mirzendehdel [54], Ribeiro [67] and Vasconcellos [74] are some interesting examples of M.Sc. and Ph.D. theses that employ GPUs in such context. The massively parallel nature of GPUs, following the Single Instruction Multiple Thread (SIMT) paradigm, usually makes them more fitting to deal with vector and matrix problems than CPUs, enabling impressive performance gains. However, memory availability in most low-cost GPUs is an issue, often being a limiting factor for large-scale FEM analysis, even when accounting for sparsity or adopting assembly-free approaches. This hinders the usage of state-of-the-art μCT capabilities, in the scope of computational homogenization, as the larger models do not fit in the common 4 GB or 8 GB DRAM, leading to dependency on clusters and supercomputers. In light of this, there is a demand for implementations that exploit specificities of the simulations to obtain greater memory efficiency, so that representative models can be studied with personal computers, equipped with relatively accessible GPUs.

1.1 Motivation

The developments presented in this text follow the works of Pereira et al. [61] and Vianna [75], focusing on making feasible the numerical analysis with FEM of large-scale voxelbased models obtained with μ CT in personal computers, exploring GPU resources to speed up the computations. Figure 1.1 depicts a microtomographic model of a sandstone sample, previously studied in Vianna et al. [76]. The simulations for numerical homogenization performed in Pereira et al. [61] took place in a single 32-core CPU, making use of multithread parallelism techniques with OpenMP. Each PCG solution for a $\sim 2 \times 10^8$ DOFs (400³ voxels) elasticity analysis would take roughly 2 hours to compute, meaning that the full homogenization study would demand about 12 hours.



Figure 1.1: Digital model of a sandstone sample obtained wit μ CT. Source: Vianna et al. [76].

Having access to a CUDA-enabled Nvidia device, a GeForce RTX 2080 Super[™], it is expected that performance can be considerably improved by tackling the bottleneck of the analyses on the GPU, that is, solving the systems of equations with the PCG method. The adopted assembly-free scheme is suitable to be implemented in vector-based and massively parallel environments. However, memory requirements are high, and the 8 GB of available DRAM in the global memory of this device are a strong constraint to the dimensions of the images that can be studied. As Vianna [75] shows, the total memory requirement for the elasticity analysis in CPU of a 500³ voxels image amounts to 19.6 GB, of which 12.0 GB are taken by the arrays that store the vectors associated with the PCG method, with the pre-existing solver. Hence, a careful assessment of memory allocation, and data transfer, on the GPU is required for the new proposed solver, since it is a goal to run simulations on even larger models. In addition to this, the author has a Nvidia GeForce 940MX[™] GPU (4 GB DRAM) at personal disposal, and it is also desired that the massively parallel solution can be run on this device for relatively large models, as a proof of concept that computations for the homogenization of representative periodic cells can be carried out even on a laptop, on reasonable time.

1.2 Goals

The objectives of this work are threefold.

- 1. Implementation of a massively parallel PCG solver with CUDA, applied to assemblyfree image-based FEM problems.
- 2. Achieving significant speed-up for the elapsed time for whole numerical homogenization process, by solving the linear systems of algebraic equations in the GPU.
- 3. Making possible the thermal conductivity analysis of models with $\sim 4 \times 10^8$ DOFs, which correlates to images of 724³ voxels, with GPUs of 4 GB DRAM. This corresponds to the useful part of a material sample imaged with 1024³ voxels.

1.3 Outline

This document is divided in 8 Chapters. Chapter 1 consists of this introduction. Chapter 2 is focused on a literature review, where related work and a background to this text are explored, and a summary of the performance of similar developments is exposed. The intent is to set a goalpost on how much gain can be achieved with the present work. Chapter 3 presents the mathematical modeling behind image-based numerical homogenization and assembly-free FEM, as well as the physical phenomena analyzed in the scope of this work, thermal conductivity and elasticity. Chapter 4 aims to characterize the PCG method as a computational problem, showing how specificities of the proposed numerical simulations impact its demanded operations, and exposes preliminary CPU implementations of the parallel algorithm. Chapter 5 depicts the GPU implementation, going into details of the code in CUDA C, memory allocation on host and device, data flow between CPU and GPU and 4 alternative ways of implementing the proposed solution, each with its advantages and caveats. Chapter 6 presents a novel strategy, to the best of the author's knowledge, for obtaining good initial guesses for the PCG method from coarse meshes, in the scope of image-based FEM. Chapter 7 presents some results, metrics and discussion, validating the developed program with an analytical benchmark and analyses of a microtomographic model of a cast iron sample, and a push to the limits of the available hardware, simulating the largest synthetic sample possible. Finally, Chapter 8 brings some concluding remarks and comments on future work.

Chapter 2

Literature Review

2.1 Numerical Homogenization

The field of Materials Science advanced at large during the second half of the last century. Within this scenario, interest grew in studying heterogeneous materials at a microscopic level, to obtain homogenized effective physical properties. In this context, academic works for the modeling of the mechanical behavior of heterogeneous media, such as Hashin and Shtrikman [29], from 1962, and the development of homogenization techniques, such as Hill [33], presented in 1963, were published in Solid Mechanics journals. Advancements continued in the following decades, with works such as Cioranescu and Paulin [13], in which the homogenization of domains with holes (void/pores) was addressed. Sophisticated mathematical models were conceived, yielding analytical solutions for some geometrically well-behaved domains, as shown in 1979 by Perrins et al. [62]. However, to conduct analyses for models more representative of the micro-scale of composite and natural material samples, numerical methods and computational tools were required.

The idea of working with numerical (or computational) homogenization began to gain popularity in the 90's and early 00's. Guedes and Kikuchi [25] worked with the Finite Element Method (FEM) for the homogenization of elastic properties, Moulinec and Suquet [55], and Michel et al. [53] synthesized computational approaches to solve the governing equations of elasticity in periodic media for homogenization problems, employing the FEM or, alternatively, Fast Fourier Transforms. In 2006, Cartraud and Messager [11] presented a computational solution for periodic beam-like structures. A few years later, in 2009, Pinho-da-Cruz, Oliveira and Teixeira-Dias [15, 59] published a two part article documenting the mathematical formulation for asymptotic homogenization with FEM, considering periodic cells. Terada et al. [73] and Kanit et al. [35] studied the determination of representative volume elements (cells) for the convergence of numerical solutions, assessing different types of boundary conditions. A decade later, Nguyen et al. [57] suggested that the adoption of periodic boundary conditions accelerates convergence, in comparison to conventional Dirichlet and Neumann conditions.

In 2014, Andreassen and Andreasen [2] presented a concise educational article towards numerical homogenization with structured finite element meshes, adopting periodic boundary conditions. They provided a clear step-by-step solution that can serve as an interesting starting point for researchers to get in touch with the topic.

Over the last decade, with the development of imaging technologies, such as μ CT, in conjunction with growing computational power of relatively accessible machines, there is a notorious trend of working with image-based models for homogenization. The high resolution of the images, being able to capture details in the order of μ m of the microscale of materials, leads to large-scale FEM problems. There is extensive recent literature reporting developments in this scope, it is an active research field. References [20, 41, 42, 45, 61, 65, 68, 70, 71, 75, 76, 77] are examples of works of image-based homogenization, from the last 5 years.

2.2 Assembly-free FEM

To deal with the large-scale FEM problems that arise with high resolution image-based models, memory-efficient implementations are required. The general case finite element analysis has super-linear space complexity, due to the global matrix and required auxiliary data structures, so it quickly fills the DRAM of personal computers and workstations, as the number of DOFs increases. If the sparsity of the systems is considered, O(n) space complexity can be achieved. The assembly-free (matrix-free) strategy reduces even further the required allocation while maintaining linear complexity, thus making it possible for much larger models to be studied.

Hughes et al. (1983) [34], Carey and Jiang (1986) [10], and Erhel et al. (1991) [21] are among the first published assembly-free FEM implementations, being commonly cited as the original sources for the approach. The strategy was referred as element-byelement, as it employed loops over elements to perform sweeps of the domain, so that the coefficients of the global matrix could be computed on-the-fly. The considerable reduction in memory allocation made feasible the simulations of FEM problems with millions of DOFs in personal machines, but the limited performance of the hardware at that time hindered the exploration of the full potential of the proposed methodologies, as they are burdened with the caveat of increased computational cost.

In the last decade, considering the rise of massively parallel computation devices, namely the fast-paced improvements seen with GPUs [38, 69], the assembly-free FEM became a standard for large-scale problems, especially when dealing with image-based models and/or structured meshes. Numerous academic works documenting GPU solutions for matrix-free FEM problems can be found in the recent literature, such as Akbariyeh [1], Apostolou [4], Kiran et al. [37], Kronbichler and Ljungkvist [39], Loeb and Earls [43], Martínez-Frutos and Herrero-Pérez [47], Martínez-Frutos et al. [49], Mirzendehdel [54], Müller et al. [56], and Reguly and Giles [66]. All of these were published from 2012 to 2020. Most adopt a PCG solver.

It is interesting to notice that different fields are merging in this topic. Researchers of biomechanics need to perform large finite element analyses in microscopic images of bone samples, so they rely in not storing the global matrix, as it can be seen in works like Arbenz et al. [5], Bekas et al. [7], Flaig and Arbenz [24], and Keßler [36]. The study of topology optimization of structures requires several solutions of large-scale FEM systems, leading researchers of the area to employ the discussed approach as well, as in Duarte [18], Duarte et al. [19], and Martínez-Frutos and Herrero-Pérez [48]. In the context of image-based homogenization, the works of Liu et al. [41, 42], and Pereira et al. [61] are examples that apply the assembly-free FEM.

2.3 Massively Parallel PCG in GPU

The Conjugate Gradient (CG) method for solving linear systems characterized by positive definite and symmetric matrices was originally presented in 1952, by the works of Hestenes and Stiefel [32], and Lanczos [40]. At first, even though the scheme relies on an iterative process, it was categorized as a direct method, as it can be proved that an exact solution is found with n steps, where n is the dimension of the system. However, much faster convergence is achieved within a numerical tolerance, especially with the employment of preconditioners, as detailed by Benzi [8], so the CG was later deemed an iterative method.

The Preconditioned Conjugate Gradient (PCG) method, brilliantly described by Shewchuk [72], extends the original CG to admit preconditioning, being an iterative scheme commonly employed to large-scale FEM problems. The characteristics of the system of

	Hardware			A [GB]						
Reference (year)	CPU	GPU	CPU	GPU	Assembly- free	Structured mesh	$\begin{array}{c} \text{DOFs} \\ \times 10^6 \end{array}$	GPU Mem. Alloc.[GB]	Time [s]	Time per iteration [s]
[1] (2012)	Intel Core i5 2.66GHz	Nvidia Quadro 5000	4.0	2.5	\checkmark	\checkmark	10.1	2.5	-	0.418
[4] (2020)	$\operatorname{Intel}(\mathrm{R})$ (16 threads)	Nvidia Titan V	-	12.0	\checkmark	\checkmark	166.0	11.4	31.1	-
[19] (2015)	Intel Core i7- 2820QM (16 threads)	Nvidia GTX Titan	16.0	6.0	\checkmark	×	20.0	-	2270	22.70
[23] (2016)	Intel Xeon E5 (12 threads)	Nvidia Tesla K20m	32.0	5.0	×	×	3.2	5.0	262	0.398
[31] (2012)	Intel Xeon 2.66GHz (8 threads)	Nvidia Tesla T10	12.0	4.0	×	×	2.1	-	71.4	-
[37] (2020)	Intel Xeon E5 2.20GHz (24 threads)	Nvidia Tesla K40	-	12.0	V	√	14.0	2.6	-	0.020
[43] (2019)	Intel Xeon E5 2.70GHz	AMD FirePro D700	-	6.0	\checkmark	✓	2.0	-	26.0	0.021
[49] (2015)	Intel Xeon E5603 2.13GHz	Nvidia Tesla K40m	-	12.0	\checkmark	√	2.0	-	-	0.200
[56] (2013)	Intel Xeon E5-2620 2.00GHz	Nvidia Fermi M2090	-	6.0	\checkmark	\checkmark	8.4	~ 0.2	0.50	0.006
[64] (2019)	Intel Xeon E5606 2.13GHz	Nvidia Quadro M6000	64.0	12.0	\checkmark	\checkmark	0.9	-	3000	-

Table 2.1: Performance metrics found in the literature for massively parallel PCG solvers in GPU

equations from the FEM allow for convergence in few steps with the PCG, so solutions can be obtained much faster than with other usual iterative or direct methods.

The computations at each iteration of the PCG consist of vector and matrix operations that can be implemented in massively parallel environments. The work of Helfenstein and Koko [31], from 2012, documented a GPU implementation of the conventional form of the PCG method, as a standalone solver for large sparse systems, using the CSR format to reduce memory allocation. From this point on, many developments focus on specificities of the analysis being performed to make more efficient solvers. Many of the aforementioned references in the previous Section employ the PCG method. In addition, Fialko and Zeglen (2016) [23], and Pikle et al. (2018) [63, 64] implemented the PCG for large-scale finite element analysis in GPU.

In Table 2.1, performance metrics for several PCG solvers found in the literature are exposed, focusing on massively parallel implementations in GPUs. The idea is to associate available hardware, sizes of the simulations, peak memory allocation and elapsed time, with the objective of setting benchmarks to the solution developed in this work.

It is noticeable, from Table 2.1, that most solvers deal with about 1 million to 100 million DOFs. This range can be explained by the device available for each author and the various possibilities of implementation. For example, Apostolou [4] uses double precision floating point variables, being able to run analyses for a 166 million DOFs model allocating

11.4 GB. Oppositely, Müller et al. [56] present a solution with greater memory efficiency that employs single precision, showing the simulation of a model with ~ 8 million DOFs storing just 167 MB in DRAM, which means that this solver is expected to be able to handle models of up to 300 million DOFs with their 6 GB DRAM device. It is notorious that the assembly-free strategy is associated with larger models. Some works employ highperformance computing clusters and supercomputers to solve problems of this nature, and in doing so, are able to study models of much larger dimensions. Arbenz et al. [5] presented the solution of models of bone micro-structure with up to impressive 25 billion DOFs, using the Tödi cluster, a Cray XK7 supercomputer composed of 272 nodes, each one equipped with a Nvidia Tesla K20x GPU, of 6 GB DRAM. Duarte [18] ran a topology optimization analysis of a model with 3 billion DOFs in the Blue Waters supercomputer, a cluster of 22640 nodes, each one equipped with a 32-core CPU of 64 GB DRAM, using 2916 machines. These two references were not mentioned in Table 2.1, as we are focusing on solutions in a single GPU, but they help to contextualize the state-of-the-art of solving large-scale sparse systems with the PCG method.

Usually, PCG solvers in GPU allocate at least five arrays to store vectors of variables, when dealing with assembly-free FEM. In this work, the method is reassessed, combining some of the vector and matrix operations together, to reduce memory allocation. Solutions with four, three and two arrays stored in GPU will be presented. In addition, single precision floating points are used to characterize all field variables. The proposed implementations aim to improve on the memory efficiency currently found in the literature.

In regards to time, it is not straightforward to compare performance, from the data in Table 2.1, as different hardware are involved, and, unfortunately, not every author provides the number of iterations for convergence of the PCG method or the adopted numerical tolerance. Regardless, from the metrics observed in these references, it seems reasonable to expect that solutions for models of about 100 million DOFs, with tolerance of at least 5 digits, can be obtained in under 100 s with the most powerful device employed in this work (Nvidia GeForce RTX 2080 SuperTM). In fact, this expectation was met, as it will be shown at the end of this document.

Chapter 3

Numerical Homogenization

Numerical homogenization consists in the evaluation of effective physical properties for heterogeneous materials by establishing relationships between their behavior at a micro and a macro-scale. It is inherently a multi-scale problem, as it is admitted that two well defined different scales exist [53]. The final product of the computational experiments is a constitutive tensor that describes the physical behavior of the studied material, as it were homogeneous. The description of the homogenization process presented ahead is based on previous work such as Pereira et al. [61], Vianna et al. [76], Vianna [75], and Sapucaia [70]. Andreassen and Andreasen [2] is an interesting reference that synthesizes the methodology.

3.1 An introduction

The homogenization analyses are based on numerical simulations to solve the governing differential equations of a given physical phenomenon within a heterogeneous medium, admitting specific forcing conditions. An average of the response observed at the microscale is computed to characterize a homogenized material. Such average is described as an integral of any given field over the domain of analysis [33], as in

$$\langle \phi \rangle = \frac{1}{|\Omega|} \int_{\Omega} \phi \, d\Omega \;, \tag{3.1}$$

where ϕ is a field acting in the domain Ω . The bracket notation $\langle \dots \rangle$ is commonly adopted to denote the average [53, 70, 76].

It is assumed that the heterogeneities and localized variations of the medium at a microscopic level appear in pattern-like fashion, so that they are averaged out at a macroscale level, which, despite also being heterogeneous, can be approximately modeled as homogeneous. The numerical analysis is performed in a periodic cell, a micro-scale heterogeneous domain that represents a material point of the macro-scale, which shows little to no translational variance. This means that it is admitted that the studied image-based model could be periodically repeated to closely approximate the sample, as portrayed in Figure 3.1. The assumption of periodic media is based on previous work such as Michel et. al [53], Cartraud and Messager [11], Andreassen and Andreasen [2], and Vianna et al. [76]. In that sense, periodic boundary conditions, as illustrated in Figure 3.2, are adopted to solve the governing PDEs [35, 57, 68]. It should be noted that, for the consideration of random micro-structures, this sort of analysis could be improved by the adoption of stochastic methods [9, 65].



Figure 3.1: Periodic cell.

Two types of physical problems are considered in the scope of this work, heat conduction and elasticity, whose formulations are presented in details in Subsections 3.2.1 and 3.2.2. The homogenized constitutive tensors for each analysis are obtained, respectively, as in

$$\langle q \rangle_i = -\kappa_{ii}^h \langle \nabla T \rangle_j, \tag{3.2}$$

where $\langle q \rangle_i$ is the averaged heat flux density field, $\langle \nabla T \rangle_j$ is the averaged temperature gradient and κ_{ij}^h is the homogenized thermal conductivity tensor, and

$$\langle \sigma \rangle_{ij} = E^h_{ijkl} \langle \varepsilon \rangle_{kl}, \tag{3.3}$$

where $\langle \sigma \rangle_{ij}$ is the averaged stress field, $\langle \varepsilon \rangle_{kl}$ is the averaged strain field and E^h_{ijkl} is the homogenized stiffness tensor.



Figure 3.2: (a) Periodic boundary conditions, source: Vianna [75]. (b) Convergence of effective properties for different boundary condition considerations, adapted from Nguyen et al. [57] and Sapucaia [70].

In order for the coefficients of the constitutive tensors to be computed, it is clear in Equations 3.2 and 3.3 that two averaged fields must be known. By setting one of the two, it is possible to calculate the other by running physical simulations in the microscopic domain, followed by the employment of Equation 3.1. It is important to notice that the constitutive tensors are of a higher order than the fields to which they relate to, so multiple analyses must be performed to fully characterize them. In 3D, for thermal conductivity, three analyses must be run, while, for elasticity, six simulations are needed, accounting for the inherent symmetries associated with this constitutive law, which are further discussed in Subsection 3.2.2.

3.2 Governing equations

This Section is dedicated to expose the formulation of the physical phenomena studied in this work, going from the strong form of the governing partial differential equations to their respective weak form, suitable to be solved numerically with FEM, which will be discussed in Section 3.3. All the constitutive behavior of the different phases present in the heterogeneous domain is assumed to be linear. The mathematical expressions are depicted in tensor notation.

3.2.1 Heat Conduction

The mathematical modeling presented in this Subsection is based on the book on Heat Conduction written by Hahn and Özisik [28]. The derivation of the governing equation in weak form follows Cook [14].

Let us consider a domain Ω in \mathbb{R}^3 subjected to a heat source Q in stationary equilibrium with a heat flux density field q_i and a temperature gradient ∇T . It is known, by Fourier's law (Equation 3.4), that q_i can be written in terms of ∇T , i.e.

$$q_i = -\kappa_{ij} (\nabla T)_j, \tag{3.4}$$

where κ_{ij} is the thermal conductivity tensor. Furthermore, at any point of the domain, it must hold that

$$\nabla \cdot q_i = Q. \tag{3.5}$$

For isotropic materials (Appendix A), κ_{ij} can be substituted by a scalar, setting the subscribed indexes of ∇T as the same of q_i . From this point on, it will be referenced like so, for the sake of simplicity, but it is noteworthy that the following formulation works for anisotropic materials as well.

By combining Equations 3.4 and 3.5, Equation 3.6 is obtained. This expression is notoriously known as Poisson's Equation in Euclidean space, an elliptic PDE that appears in similar form in other physical problems, such as Newtonian Gravity and Electrostatics.

$$\kappa \nabla^2 T + Q = 0. \tag{3.6}$$

A numerical solution will be sought for the scalar field T, through the approximation of it with a finite sum of simple interpolation functions. By applying the weighted residual method, that is, imposing a null average error over the domain, Equation 3.7 is obtained.

$$\int_{\Omega} w(\kappa \nabla^2 T + Q) \, d\Omega = 0, \qquad (3.7)$$

where w is a non-null weight function, also often denominated test function.

The integral in Equation 3.7 can be separated as two, one for the $\kappa \nabla^2 T$ part, and another for Q. The divergence theorem states that the integral over a closed domain of a scalar field multiplied by the divergent of a vector field can be calculated as

$$\int_{\Omega} w\kappa \nabla^2 T \, d\Omega = \oint_{\partial\Omega} w\kappa (\nabla T)_i \hat{n}_i \, d\partial\Omega - \int_{\Omega} (\nabla w)_i \kappa (\nabla T)_i \, d\Omega , \qquad (3.8)$$

where $\partial \Omega$ corresponds to a surface that borders the domain Ω , and \hat{n} is the vector field normal to $\partial \Omega$.

The weight function is arbitrary, so it is possible to choose one that is compatible with the variable field T. In this case, this means that w can be set as δT , usually referred to as a virtual temperature field. By taking this into consideration and substituting Equation 3.8 back in Equation 3.7, we get

$$\int_{\Omega} (\nabla \delta T)_i \kappa (\nabla T)_i \, d\Omega = - \oint_{\partial \Omega} \delta T q_i \hat{n}_i \, d\partial \Omega + \int_{\Omega} \delta T Q \, d\Omega , \qquad (3.9)$$

which is the weak form of Poisson's Equation for Heat Conduction that commonly is solved with the FEM.

3.2.2 Elasticity

The mathematical modeling presented in this Subsection is based on the book on Continuum Mechanics written by Mase, Smelser and Mase [50]. The derivation of the governing equations in weak form follows literature on elasticity problems with the FEM, such as Barbero [6], Felippa [22], and Martha [46].

Similarly to what was done on the previous Subsection, admit a domain in \mathbb{R}^3 denoted by Ω , now representing a deformable body subjected to body forces f_i . Let σ_{ij} be the internal stress field acting in Ω , associated with a strain field ε_{ij} . The generalized Hooke's law for linear elasticity states that stress and strain are related, as in

$$\sigma_{ij} = E_{ijkl}\varepsilon_{kl},\tag{3.10}$$

where E_{ijkl} is called the stiffness tensor.

Assuming small displacements, that is, a geometrically linear analysis, the strain field ε_{ij} can be written in terms of the displacement field u_i , as shown in Equation 3.11. This is commonly denominated the compatibility equation.

$$\varepsilon_{ij} = \frac{1}{2}(u_{i,j} + u_{j,i}),$$
(3.11)

where the subscript notation $u_{i,j}$ denotes a partial derivative, as in $\partial u_i / \partial x_j$. It is important to notice that, according to this definition, $\varepsilon_{ij} = \varepsilon_{ji}$, meaning that the strain tensor field ε_{ij} is symmetric. In addition, by coupling 3.11 with 3.10, it is clear that σ_{ij} is also a function of u_i .

An assessment of static equilibrium, that is, making the assumption of no transient behavior, implies that Equation 3.12 must hold throughout the domain Ω .

$$\sigma_{ji,j} + f_i = 0. (3.12)$$

As it was done in Equation 3.7, the weighted residual method is applied to Equation 3.12, leading to

$$\int_{\Omega} w_i(\sigma_{ji,j} + f_i) \, d\Omega = 0, \qquad (3.13)$$

where w_i is now a collective of 3 non-null weight functions, which can be thought of as a vector field. Similarly to Equation 3.8, the divergence theorem can be applied in its tensor form to the $\sigma_{ji,j}$ part of the integral in 3.13, so that it extends to

$$\int_{\Omega} (\nabla w)_{ij} \sigma_{ji} d\Omega = \oint_{\partial \Omega} w_i \sigma_{ji} \hat{n}_j d\partial \Omega + \int_{\Omega} w_i f_i d\Omega . \qquad (3.14)$$

Analogously, the weight functions can be chosen to be compatible with the variable field u_i . However, attention must be drawn to the fact that, in this case, neither w_i or u_i are scalar fields. w_i is set as δu_i , a vector field in \mathbb{R}^3 , which can be understood as virtual displacement. In light of this, Equation 3.14 can be rewritten as

$$\int_{\Omega} (\nabla \delta u)_{ij} \sigma_{ji} \, d\Omega = \oint_{\partial \Omega} \delta u_i \sigma_{ji} \hat{n}_j \, d\partial \Omega + \int_{\Omega} \delta u_i f_i \, d\Omega \,. \tag{3.15}$$

It is interesting to notice that, by plugging Equation 3.11 into Equation 3.10 and substituting the result in the above expression, it becomes the weak form of Navier's Equation.

Even though 3.15 is a differential equation in its weak form, it still can be further simplified for a computational approach. It is desired to deal mostly with vectors, and this can be achieved by accounting for the symmetries observed in Hooke's law (3.10). In Mase et al. [50], those are explored and explained in details. It is known that the stress field σ_{ij} is symmetric, which means that, from its 9 components, 6 are actually distinct in value. The same can be affirmed for the strain field ε_{ij} . Hence, both tensor fields can be represented by pseudo-vectors, as in

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{23} \\ \sigma_{13} \\ \sigma_{12} \end{bmatrix}, \qquad \boldsymbol{\varepsilon} = \begin{bmatrix} \varepsilon_{11} \\ \varepsilon_{22} \\ \varepsilon_{33} \\ \gamma_{23} \\ \gamma_{13} \\ \gamma_{12} \end{bmatrix}, \qquad (3.16)$$

where $\gamma_{ij} = 2\varepsilon_{ij}$ are commonly referred as engineering shear strain coefficients.

The stiffness tensor E_{ijkl} also has its own symmetry properties. As σ_{ij} and ε_{ij} are symmetric, it straightforwardly follows, according to 3.10, that $E_{ijkl} = E_{jikl}$ and $E_{ijkl} = E_{ijlk}$, respectively. With this and Equation 3.16 in mind, the constitutive law (3.10) can be written as

$$\begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{23} \\ \sigma_{13} \\ \sigma_{12} \end{bmatrix} = \begin{bmatrix} E_{1111} & E_{1122} & E_{1133} & E_{1123} & E_{1113} & E_{1112} \\ E_{2211} & E_{2222} & E_{2233} & E_{2223} & E_{2213} & E_{2212} \\ E_{3311} & E_{3322} & E_{3333} & E_{3323} & E_{3313} & E_{3312} \\ E_{2311} & E_{2322} & E_{2333} & E_{2323} & E_{2313} & E_{2312} \\ E_{1311} & E_{1322} & E_{1333} & E_{1323} & E_{1313} & E_{1312} \\ E_{1211} & E_{1222} & E_{1233} & E_{1223} & E_{1213} & E_{1212} \end{bmatrix} \begin{bmatrix} \varepsilon_{11} \\ \varepsilon_{22} \\ \varepsilon_{33} \\ \gamma_{23} \\ \gamma_{13} \\ \gamma_{12} \end{bmatrix} \Rightarrow \boldsymbol{\sigma} = \mathbf{C}\boldsymbol{\varepsilon}. \quad (3.17)$$

The fourth order constitutive tensor has been reduced to a 6 by 6 matrix, which will be depicted as \mathbf{C} in the following developments. If the original tensor subscripts were switched for the pseudo-vectors and matrix indexes, Equation 3.17 would denote the Voigt notation of Hooke's law. It is important to state that all of the contractions and simplifications considered are nothing more than notation and algebraic artifices. Conceptually, the original tensor form still holds.

The compatibility equation (3.11) can also be reassessed to consider the adopted simplifications. Equation 3.18 shows how the pseudo-vector $\boldsymbol{\varepsilon}$ can be obtained in terms of the displacement field u_i .

$$\begin{bmatrix} \varepsilon_{11} \\ \varepsilon_{22} \\ \varepsilon_{33} \\ \gamma_{23} \\ \gamma_{13} \\ \gamma_{12} \end{bmatrix} = \begin{bmatrix} \partial/\partial x_1 & 0 & 0 \\ 0 & \partial/\partial x_2 & 0 \\ 0 & 0 & \partial/\partial x_3 \\ 0 & \partial/\partial x_3 & \partial/\partial x_2 \\ \partial/\partial x_3 & 0 & \partial/\partial x_1 \\ \partial/\partial x_2 & \partial/\partial x_1 & 0 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} \Rightarrow \boldsymbol{\varepsilon} = [\nabla] u_i. \quad (3.18)$$

At last, considering 3.17 and 3.18, Equation 3.15 can be expressed as in

$$\int_{\Omega} ([\nabla] \delta u_i)^T \mathbf{C} ([\nabla] u_i) \, d\Omega = \oint_{\partial \Omega} \delta u_i \sigma_{ji} \hat{n}_j \, d\partial \Omega + \int_{\Omega} \delta u_i f_i \, d\Omega \,, \qquad (3.19)$$

this is the weak form that usually is solved with the FEM.

3.3 Finite Element Method

This Section aims to describe the Finite Element Method, exposing how it can be employed to numerically solve the weak form of the governing equations derived in the previous Section, accordingly to well-known references such as Cook [14] and Felippa [22]. It is shown how the local systems of algebraic equations are formulated and it is discussed how the assembly-free strategy is adopted to characterize the global system.

3.3.1 A brief introduction

The core idea behind the FEM is that an unknown continuous field acting in a domain Ω , on a boundary value problem, can be approximated by nodal values interpolated with localized functions in a finite discretization of Ω . The finite partitions of the domain are called elements, and the collective of those is denominated mesh. Figure 3.3 depicts the discretization of a continuous domain into a finite element mesh. If Ω is in \mathbb{R}^2 the elements are surface patches, in \mathbb{R}^3 they are volumetric solids.

Each finite element has a pre-defined behavior, dictated by the adopted interpolation functions, usually referred as shape functions, as stated in Equation 3.20. These functions are constrained by the number of nodes on an element, that is, the number of discrete points where the variable field is approximated. For example, a quadrilateral element in 2D with 4 nodes, 1 at each vertex, may employ bilinear polynomials as its shape functions,



Figure 3.3: (a) Continuous domain Ω , (b) Discretization into a finite element mesh. as depicted in Figure 3.4.

$$\phi^{\rm e} \approx \mathbf{N} \mathbf{\Phi},\tag{3.20}$$

where ϕ^{e} is a variable field within the domain of an element, **N** is the matrix of shape functions for that same element, and Φ is the vector that holds the nodal values of ϕ .



Figure 3.4: Bilinear shape functions for a 1x1 quadrilateral finite element in 2D

By applying this notion to PDEs in weak form, such as Equations 3.9 and 3.19, a system of algebraic equations is achieved, at a local reference, which means it represents the approximate solution in the domain of the element in question. By coupling together all the local systems, that is, taking into consideration that neighbor elements on a mesh share some nodes and therefore are related, on a global reference, a larger system of equations is obtained, now valid for the whole mesh. The global matrix usually is ill-conditioned,
as some of its lines are linearly dependent of others, until proper boundary conditions are applied. In this work, periodic boundary conditions [35, 57, 68] are employed.

From a computational standpoint, the method consists of assembling and solving a system of algebraic equations. For general cases, data structures that store the geometric properties of the mesh and the vicinity information of each element are needed, as well as a material map, and quite often a DOF map. On large-scale problems (> 10^6 DOFs) the required memory quickly adds-up to troublesome demands for personal computers, which motivates the search for efficient implementation strategies.

3.3.2 Heat Conduction

Continuing from where Subsection 3.2.1 left off, Equation 3.9, we can now employ the notion shown in Equation 3.20 to approximate the temperature field T, as in

$$T \approx \mathbf{N}\boldsymbol{\tau},$$
 (3.21)

where τ represents the nodal temperature values. By inserting the expression above in Equation 3.9, we get

$$\delta \boldsymbol{\tau}^T \int_{\Omega_e} (\nabla \mathbf{N})^T \kappa(\nabla \mathbf{N}) \, d\Omega \, \boldsymbol{\tau} = -\delta \boldsymbol{\tau}^T \oint_{\partial \Omega_e} \mathbf{N}^T q_i \hat{n}_i \, d\partial \Omega \, + \delta \boldsymbol{\tau}^T \int_{\Omega_e} \mathbf{N}^T Q \, d\Omega \, , \qquad (3.22)$$

where $\delta \tau$ and τ can be taken out of the integrals for not being functions of Ω . It is vital to notice, in Equation 3.22, that it was assumed that the virtual temperature field δT (our weight function) can be approximated with shape functions as well. This is known as the Galerkin method, where the weight functions in a weighted residual method problem are substituted by the same interpolation functions that are applied to the variable field.

In Equation 3.22, $\delta \tau$ is a non-null vector, so it can be taken out of the expression, as the equality must hold for the remaining terms. At last, Equation 3.23 is defined.

$$\int_{\Omega_e} (\nabla \mathbf{N})^T \kappa(\nabla \mathbf{N}) \, d\Omega \, \boldsymbol{\tau} = -\oint_{\partial\Omega_e} \mathbf{N}^T q_i \hat{n}_i \, d\partial\Omega \, + \int_{\Omega_e} \mathbf{N}^T Q \, d\Omega \, . \tag{3.23}$$

In the expression above, the terms on the right-hand side are related to, respectively from left to right, heat flux at the border and a heat source, both are input parameters. The result of these integrals can be computed, being commonly denominated the forcing vector **f**. At the left-hand side of the equation, the approximate solution vector $\boldsymbol{\tau}$ is being multiplied by the result of an integral that associates the shape functions to the constitutive tensor, known as the thermal conductivity matrix **K**. Equation 3.23 can be written more concisely, as in 3.24. This is the system of algebraic equations that must be satisfied, at a local reference.

$$\mathbf{K}_{\mathrm{e}}\boldsymbol{\tau}_{\mathrm{e}} = \mathbf{f}_{\mathrm{e}},\tag{3.24}$$

where the subscript $(\ldots)_{e}$ indicates that this equation refers to a single element.

3.3.3 Elasticity

Analogously to what was done on the previous Subsection, the variable field, in this case, displacement, can be approximated with shape functions, as denoted by

$$u_i \approx \mathbf{Nd},$$
 (3.25)

where \mathbf{d} is the vector of nodal displacement values. Taking this into consideration, and applying the Galerkin method to Equation 3.19, it becomes

$$\delta \mathbf{d}^T \int_{\Omega_e} ([\nabla] \mathbf{N})^T \mathbf{C} ([\nabla] \mathbf{N}) \, d\Omega \, \mathbf{d} = \delta \mathbf{d}^T \oint_{\partial \Omega_e} \mathbf{N}^T \sigma_{ji} \hat{n}_j \, d\partial\Omega \, + \delta \mathbf{d}^T \int_{\Omega_e} \mathbf{N}^T f_i \, d\Omega \, . \tag{3.26}$$

Similarly to the case of heat conduction, for elasticity, in the expression above, the vector of virtual nodal displacements $\delta \mathbf{d}$ can be taken out of the equation, as it is non-null. Furthermore, the integrals on the right-hand side of the equation depict surface tractions and body forces, respectively, which are input information. They constitute the forcing vector \mathbf{f} . The left-hand side is characterized by the nodal displacement vector \mathbf{d} being multiplied by a matrix, resulting of an integral that associates the constitutive tensor to shape functions. This is known as the stiffness matrix, represented by \mathbf{K} . Hence, the local system of equations is defined, as written in

$$\mathbf{K}_{\mathbf{e}}\mathbf{d}_{\mathbf{e}} = \mathbf{f}_{\mathbf{e}},\tag{3.27}$$

where the subscript $(...)_e$ indicates that this equation refers to a single element.

3.3.4 Assembly-free FEM

The characteristic global system of algebraic equations associated with the FEM is the result of coupling all the local systems, as Equations 3.24 and 3.27. The domain of analysis is discretized as a mesh, so the nodal values are related to multiple elements, meaning that a global problem must be solved, as in

$$\mathbf{K}\mathbf{x} = \mathbf{f},\tag{3.28}$$

where \mathbf{K} is the global matrix, \mathbf{x} is the vector of nodal values of the approximated solution, \mathbf{f} is the global forcing vector.

The generation of the the matrix \mathbf{K} is usually called *assembly*. Authors commonly use some form of Equation 3.29 to represent it mathematically [14, 22].

$$\mathbf{K} = \sum_{e=1}^{n_{\text{elements}}} \mathbf{T}_{e}^{T} \mathbf{K}_{e} \mathbf{T}_{e}, \qquad (3.29)$$

where \mathbf{T}_e are called the incidence, or transformation, matrices, responsible for projecting the local systems into a global reference.

The assembly of the global matrix is a vital step in most FEM solvers. Admittedly, in cases where it is possible to assemble and properly store the matrix in memory, it probably is best to do it, since it makes the usual system solving techniques considerably less expensive computation-wise, be it direct or iterative. Efforts are made to explore sparsity and symmetry of FEM matrices to lighten the burden of memory allocation when dealing with large models, yet most of such strategies have some sort of super-linear (or linear with a high slope coefficient) tendency of growth of space consumption, when related to the number of DOFs a models has, which defines the size of the system to be solved. This constitutes a strong limitation to the discretization of models, or, in the context of this work, the resolution of images to be analyzed.

The assembly-free, often denominated matrix-free, approach is based on never allocating the global matrix, storing few small local matrices and a material map instead, aiming for linear complexity of space, in terms of the number of DOFs. Naturally, this does not come without a cost. Not having the global matrix stored in memory means that every time one of its coefficients is demanded by a step of the solver, it needs to be computed on the spot. In essence, the idea is that the right-hand side of Equation 3.29is substituted in whatever expression where **K** is required. For example, each matrixvector multiplication triggers extra operations analogous to an assembly on-the-fly, in computational cost. Such operations are performed through sweeps of the domain.

A straightforward manner to implement these domain sweeps would be to employ a loop over all elements of the mesh gathering their local contributions to global DOFs for the results of matrix operations with \mathbf{K} . This strategy is commonly referred as elementby-element [34, 10, 21]. This should be a fairly familiar way to do it, as it resembles the most conventional procedures for assembling the global matrix in FEM solvers, and it can be viewed as a direct translation of Equation 3.29 to code. Although this is perfectly correct, it imposes heavy computational costs to assembly-free solutions with iterative methods, as at each iteration it is commonly demanded a matrix-vector product with \mathbf{K} , meaning every element needs to be visited, one by one, at each step. In general, all sequential approaches to perform such operations share this type of issue, so it is not interesting to use them, as the trade-off between memory and time consumption may often be disadvantageous. This will be demonstrated ahead, in Subsection 4.5.1.

Aiming for more efficient solutions, by exploring computational resources to reduce total runtime, parallel algorithms are drawn out for the implementation of the assembly on-the-fly operations. The premise is to maximize the number of computations that can be performed simultaneously, on a conceptual level, avoiding potentially strenuous **for** loops. In this work, two massively-parallel approaches are explored, element-by-element and node-by-node. Both are detailed in Section 4.3 of this document.

3.4 Image-based Finite Element models

The details of the FEM analyses performed in the scope of this work are closely tied to the type of input that is considered. The micro-scale domain is depicted by an image, and it is admitted that each pixel (2D) or voxel (3D) corresponds to a finite element, or a cluster of evenly arranged finite elements, so that the mesh is always structured in grid-like fashion. Such regularity allows for significant memory relief, in regards to the amount of data necessary to fully characterize the mesh, as geometry and topology are pre-determined.

The images that represent the samples are stored in arrays of 8-bit entries, assigning a grayscale color value (an unsigned integer between 0 and 255) to each element, as presented in Figure 3.5. Each color is assumed to represent a material key on the heterogeneous micro-scale domain, so the image itself can be thought of as a material map.



Figure 3.5: 8-bit grayscale representation of a 5x5 pixel-based finite element mesh

All of the elements in an image-based mesh share the exact same geometry, squares or cubes of unitary dimensions. This means that the local matrices and forcing vectors on Equations 3.24 and 3.27 can be obtained via analytical solutions of the integral expressions, considering regular bilinear quadrilateral (2D) or trilinear hexahedral (3D) finite elements (see Appendix C). Furthermore, it is possible to work with only one local matrix for each different material in the heterogeneous domain, instead of one matrix per element, as usual in general case FEM implementations.

It is important to state that the assumption of such premises makes it impossible to guarantee smoothness for the considered meshes. As a matter of fact, in most cases, there are sharp corners in material interfaces within the cell, which is a known source of numerical errors in usual analysis with FEM. Even so, one should be mindful that, in this scope, the data available for the simulations are images obtained with μ CT, structured as a grid to begin with. The evaluation of smoothed regions in the domain of analysis could also be a possible source of error, by itself. In light of this, it is a choice to work directly on the provided input, which leads to a correlation between image resolution and quality of the mesh, in terms of convergence. It should also be highlighted that the aimed final result consists of averaged values throughout the domain, so, as image dimension increases, it is expected that the impact of local numerical errors becomes less relevant.

Problems analyzed with the FEM commonly demand auxiliary data structures to store the connectivity of the mesh. For an unstructured discretization of a randomly shaped domain, it is customary to store the indexes of the nodes that define the borders of each element in an array. This is a rather heavy memory toll that needs to be accounted for, on standard solutions. However, when restricting the input to structured meshes, space requirement can be reduced by employing rules of numbering for nodes consistent with the indexing of elements on the image array, making it possible to compute all connectivity information from a given index, when needed. Figure 3.6 illustrates the node numbering scheme adopted in the scope of this work for a pixel-based mesh, which reflects the admitted periodicity condition shown in Figure 3.1. Opposite border and corner nodes are indexed with the same number to make sure that their respective DOFs are equal, imposing the assumption that the behavior of the variable field in question is periodic in the studied domain. For 3D models, an analogous rule is employed, numbering by layers from near to far, as presented in Figure 3.7.



Figure 3.6: Periodic node (black) and element (red) numbering on a 2D image-based mesh

DOFs are numbered accordingly to nodes, that is, for models with scalar variables, the exact same index is used, otherwise, the index is computed considering the number of DOFs per node and the index of the respective node. For instance, in the case of 3D elasticity analysis, where the variables are vectors of 3 components, the DOF indexes at each node can be computed as $DOF_{id} = 3(node_{id}) + localDOF_{id}$.

Considering the numbering patterns presented in Figures 3.6 and 3.7, it is possible to characterize the vicinity of any node on an image-based mesh through a set of operations that essentially describe how to take steps, or to walk, on this type of grid. Table 3.1 presents such operations, in terms of node indexes and general geometric properties of the mesh. Table 3.2 summarizes the data required on Table 3.1. In addition, it is noticeable that the index of an element always matches its respective (left, top, near) node, so the rules stated on Table 3.1 are also sufficient to map the connectivity of every element on a structured mesh. These notions are essential for the efficient implementation of pixel and



Figure 3.7: Periodic node (left) and element (right) numbering on a 3D image-based mesh

voxel-based assembly-free strategies for the FEM.

Operation	Computational Procedure
WALK_UP	$node_{id}-1+n_{rows}*!(node_{id}\%n_{rows})$
WALK_DOWN	$node_{id}+1-n_{rows}*!((node_{id}+1)%n_{rows})$
WALK_RIGHT	$layer_{id}*n_{rowcol}+(layer_node_{id}+n_{rows})%n_{rowcol}$
WALK_LEFT	$layer_{id}*n_{rowcol}+(layer_node_{id}+(n_{columns}-1)*n_{rows})\%n_{rowcol}$
WALK_FAR	$(node_{id}+n_{rowcol})$ % n_{total}
WALK_NEAR	$(node_{id}+(n_{layers}-1)*n_{rowcol})%n_{total}$

Table 3.1: Operations to walk on nodes of an image-based mesh

Data	Description
node _{id}	Index of current node
$layer_{id}$	Index of current layer (always 0 in 2D)
layer_node _{id}	Local index of current node, within layer
n _{rows}	Number or rows on image-based mesh
$n_{columns}$	Number or columns on image-based mesh
n _{layers}	Number or layers on image-based mesh
$n_{\texttt{rowcol}}$	$n_{rows}*n_{cols}$
n_{total}	$n_{\texttt{rowcol}}*n_{\texttt{layers}}$

Table 3.2: Data required by operations on Table 3.1

3.5 An algorithm for image-based homogenization with the FEM

Given the topics explored in the previous Sections of this Chapter, it is possible to summarize the process of image-based homogenization with the FEM via a concise high-level algorithm. The idea is to compute the coefficients of the target homogenized constitutive tensor, through averaged results of the variable field in the domain of analysis, obtained by numerically solving the governing equations of a given physical phenomenon, admitting known forcing on directions of interest applied to a related field at the macro-scale.

Considering the matrix representations of the constitutive relations shown in Section 3.2, Fourier's law (Equation 3.4) and Hooke's law (Equation 3.17), it is clear that the adoption of an unitary temperature gradient, or strain field, on a direction associated with an index of those vectors leads to a vector of heat flux, or stress, that must be numerically equivalent to the correspondent column of the constitutive matrix. This notion can be exemplified as in

$$\begin{bmatrix} \langle q \rangle_1 \\ \langle q \rangle_2 \\ \langle q \rangle_3 \end{bmatrix} = -\begin{bmatrix} \kappa_{11} & \kappa_{12} & \kappa_{13} \\ \kappa_{21} & \kappa_{22} & \kappa_{23} \\ \kappa_{31} & \kappa_{32} & \kappa_{33} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \implies \begin{bmatrix} \kappa_{11} \\ \kappa_{21} \\ \kappa_{31} \end{bmatrix} = -\begin{bmatrix} \langle q \rangle_1 \\ \langle q \rangle_2 \\ \langle q \rangle_3 \end{bmatrix}, \quad (3.30)$$

and

$$\begin{bmatrix} \langle \sigma \rangle_{11} \\ \langle \sigma \rangle_{22} \\ \langle \sigma \rangle_{33} \\ \langle \sigma \rangle_{23} \\ \langle \sigma \rangle_{13} \\ \langle \sigma \rangle_{12} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} & C_{13} & C_{14} & C_{15} & C_{16} \\ C_{21} & C_{22} & C_{23} & C_{24} & C_{25} & C_{26} \\ C_{31} & C_{23} & C_{33} & C_{34} & C_{35} & C_{36} \\ C_{41} & C_{24} & C_{34} & C_{44} & C_{45} & C_{46} \\ C_{51} & C_{25} & C_{35} & C_{45} & C_{55} & C_{56} \\ C_{61} & C_{26} & C_{36} & C_{46} & C_{56} & C_{66} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \Rightarrow \begin{bmatrix} C_{11} \\ C_{21} \\ C_{31} \\ C_{41} \\ C_{51} \\ C_{61} \end{bmatrix} = \begin{bmatrix} \langle \sigma \rangle_{11} \\ \langle \sigma \rangle_{22} \\ \langle \sigma \rangle_{33} \\ \langle \sigma \rangle_{13} \\ \langle \sigma \rangle_{12} \end{bmatrix} .$$
(3.31)

Such macro-scale unitary effects can be modeled in the FE analysis at the micro-scale with forcing vectors, either by imposing temperature or displacement values at the periodic borders, or residual temperature gradients or strain fields throughout the domain.

Therefore, the methodology can actually be seen as a rather conventional FEM procedure, the main difference would be taking averages of the results, as a post-processing task. However, when dealing with increasingly large models, a trait of working with images as input, the assembly of a global matrix becomes impracticable, as it is memory-consuming, so the assembly-free strategy is proposed. The adoption of structured meshes, based on pixels or voxels, provides interesting properties in this context. Algorithm 1 shows pseudo-code for the homogenization of thermal conductivity using FEM, while Algorithm 2 depicts an analogous process for elasticity.

Alg	Algorithm 1 Image-based FEM thermal conductivity homogenization		
1:	1: Input: image, material properties, iterative solver tolerance		
2:	Output: homogenized material constitutive matrix κ		
3:	Generate node DOF map		
4:	Compute local conductivity matrices for each material		
5:	5: for $(i = 1, \ldots, \dim_{\kappa})$ do		
6:	Assemble right-hand side vector \mathbf{f} (micro-scale)		
7:	Solve system of equations $\mathbf{K}\boldsymbol{\tau} = \mathbf{f}$ (micro-scale)		
8:	Compute average heat flux field on domain $\langle \mathbf{q} \rangle$ (micro-scale)		
9:	Update column i of $\boldsymbol{\kappa}$ (macro-scale)		
10:	10: end for		
Alg	orithm 2 Image-based FEM elasticity homogenization		
1:	Input: image, material properties, iterative solver tolerance		
2:	2: Output: homogenized material constitutive matrix \mathbf{C}		
3:	3: Generate node DOF map		

- 4: Compute local stiffness matrices for each material
- 5: for $(i = 1, 2, ..., \dim_{\mathbf{C}})$ do
- 6: Assemble right-hand side vector \mathbf{f} (micro-scale)
- 7: Solve system of equations $\mathbf{Kd} = \mathbf{f}$ (micro-scale)
- 8: Compute average stress field on domain $\langle \boldsymbol{\sigma} \rangle$ (micro-scale)
- 9: Update column i of **C** (macro-scale)

10: end for

In Algorithms 1 and 2, the most demanding task time-wise is, by far, the solution of the system of equations, stated in line 7. This is the step that practically defines the total elapsed time of homogenization processes, when dealing with sufficiently large models, so it is the focus of this endeavor. To solve the system of linear algebraic equations, the PCG method is employed, which means it must be adapted to suit the considered assembly-free schemes. The following Chapter presents the PCG method in details, from a computational perspective.

Chapter 4

Preconditioned Conjugate Gradient Method

The Preconditioned Conjugate Gradient is an iterative numerical method for the solution of linear systems of algebraic equations, as presented in Equation 4.1. It is widely employed to solve large-scale FEM problems, due to its fast convergence, in number of steps, and relative low memory demand, as no factored matrix needs to be stored. Furthermore, it is fitting to be adapted for assembly-free strategies. The following developments are based on the great report on the PCG method written by Shewchuk [72]. Heath [30] is another noteworthy reference.

$$\mathbf{M}^{-1}\mathbf{A}\mathbf{x} = \mathbf{M}^{-1}\mathbf{b},\tag{4.1}$$

where **A** is a square symmetric positive matrix, **b** is a known right-hand side (or forcing) vector, and **x** is the unknown vector to be calculated. **M** is a preconditioning matrix, whose purpose is to lower the spectral radius of **A** ($\rho(\mathbf{A}) = \max|\lambda_i|$, where λ_i is the set of eigenvalues of **A**), in order to accelerate convergence of the solution. In this work, it is adopted the Jacobi preconditioner, a diagonal matrix that replicates the main diagonal of **A**, for its simplicity of implementation. Assuredly, the consideration of other kinds of preconditioning [8] is an open matter for further development, in future work.

The method is based on the minimization of the quadratic functional form presented in Equation 4.2. In matter of fact, it can be described as a problem of Optimization Without Constraints, being rooted in Quadratic Programming. The idea, at each iteration, is to take a step towards the global minimum of Π , adopting a search direction constructed from the previous residual to be **A**-orthogonal (conjugate) to all previous residuals and search directions. This is done instead of simply using the previous residual as the next

search direction, which would correspond to the Steepest Descent Method. Figure 4.1, adapted from Shewchuk [72], illustrates this.



$$\Pi(\mathbf{x}) = \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} - \mathbf{x}^T \mathbf{b}.$$
(4.2)

Figure 4.1: Convergence path of (a) steepest descent and (b) conjugate gradient methods for the minimization of a quadratic form, similar to Equation 4.2. Adapted from Shewchuk [72]

A brief assessment of Equation 4.2 can be made to clarify how the point of minimum value corresponds to the solution of $\mathbf{A}\mathbf{x} = \mathbf{b}$. It is known that local minimum (or maximum) of a function $f : \mathbb{R}^n \to \mathbb{R}^1$ can be found if there is a point that satisfies $\nabla f = 0$. As Π is quadratic, and \mathbf{A} is known to be positive (the concavity of Π is upwards), it is determined that the solution of $\nabla \Pi = 0$ yields a global minimum. In addition, \mathbf{A} is symmetric, so $\nabla \Pi = \mathbf{A}\mathbf{x} - \mathbf{b}$. Therefore, the point \mathbf{x} that minimizes Π must be the solution for the system of algebraic equations.

It is also interesting to notice that Equation 4.2 has some physical meaning to it. Linear elasticity problems are commonly formulated in terms of the minimization of the total potential energy [14] (an alternative to the modeling presented in Section 3.2), where the resulting functional form is quadratic, analogous to the presented expression.

The following Sections of this Chapter aim to characterize the PCG method as a com-

putational problem, focusing on its algorithm, and possible implementation approaches for massively parallel environments, considering the assembly-free FEM context. The last Section presents two preliminary implementations in CPU of the proposed parallel scheme.

4.1 The PCG algorithm

A high-level representation of the computations demanded by the PCG method is depicted in Algorithm 3. The portrayed pseudo-code is similar to the one presented by Shewchuk [72], a well-known scheme of implementation for the numerical method. This procedure assumes that a global matrix is assembled and stored in memory, so it needs to be modified to work with an assembly-free strategy. In the context of the proposed solution, the steps where **A** is demanded, lines 3 and 10, need to perform special operations, that account for the sweeps of the domain, discussed in Subsection 3.3.4 of this text.

Algorithm 3 Preconditioned Conjugate Gradient (PCG)

```
1: Input: \mathbf{A}, \mathbf{b}, \mathbf{x}_0, tolerance
   2: Output: x
   3: Initialize preconditioner (Jacobi): \mathbf{M} \leftarrow \operatorname{diag}(\mathbf{A})
   4: \mathbf{x} \leftarrow \mathbf{x}_0
   5: \mathbf{r} \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}
   6: \mathbf{d} \leftarrow \mathbf{M}^{-1}\mathbf{r}
   7: \delta \leftarrow \mathbf{r}^T \mathbf{d}
   8: \delta_0 \leftarrow \delta
   9: while (\delta/\delta_0 > \text{tolerance}^2) do
                    \mathbf{q} \leftarrow \mathbf{A}\mathbf{d}
10:
                    \alpha \leftarrow \delta / \mathbf{q}^T \mathbf{d}
11:
                    \mathbf{x} \leftarrow \mathbf{x} + \alpha \mathbf{d}
12:
                    \mathbf{r} \leftarrow \mathbf{r} - \alpha \mathbf{q}
13:
                    \mathbf{s} \leftarrow \mathbf{M}^{-1}\mathbf{r}
14:
                    \delta_{\text{prev}} \leftarrow \delta
15:
                    \delta \leftarrow \mathbf{r}^T \mathbf{s}
16:
                    \beta \leftarrow \delta / \delta_{\text{prev}}
17:
                    \mathbf{d} \leftarrow \mathbf{s} + \beta \mathbf{d}
18:
19: end while
```

In Algorithm 3, \mathbf{r} is the residual vector, \mathbf{d} is the search direction, \mathbf{q} is an auxiliary vector to store the result of the matrix-vector product (Ad) at each iteration, and \mathbf{s} is the result of preconditioning applied to \mathbf{r} . The scalar parameters δ are associated with the squared \mathbf{L}^2 norms of the residuals, used as stopping criteria, while α is the magnitude of each step, and β is a factor employed for computing conjugate search directions. It is important to reinforce that \mathbf{M} is admitted to be a diagonal matrix, stored in an array, so the operations in lines 6 and 14 can be seen as term-by-term divisions with vectors. No dense matrix should be inverted, as it would obviously be counterproductive.

It is also worth noticing, in Algorithm 3, that many of the computations required by the PCG method consist of vector operations that can be rather straightforwardly implemented in massively parallel environments. Tasks such as vector addition, termby-term multiplication, and multiplication by a scalar are simple to be treated as SIMT problems, as they are, in essence, characterized by applying the same instructions on multiple indexed entries, without any dependencies on the calculations performed on neighboring data. On the other hand, matrix operations pose a threat of generating concurrency issues if not properly dealt with, especially when adopting an assembly-free scheme. Considering image-based models, elegant approaches can be employed to ensure that no race conditions arise, without explicitly storing groups for parallel processing. This will be further detailed in Section 4.3.

Aiming for memory efficiency, some possibilities stand out, at a first look of Algorithm 3. By using an array to store the matrix-vector product result ($\mathbf{q} \leftarrow \mathbf{Ad}$), there is no need to allocate another array to store the entries of vector \mathbf{s} , as \mathbf{q} can be repurposed on every iteration to hold the values of \mathbf{s} without any loss of data demanded by the method. In addition, admitting that the PCG solver can modify the entries of the provided initial guess \mathbf{x}_0 and right-hand side \mathbf{b} vectors, they can be directly stored in the solution \mathbf{x} and residual \mathbf{r} arrays, respectively, thus reducing the total amount of allocated arrays, from eight to five ($\mathbf{x}, \mathbf{r}, \mathbf{M}, \mathbf{d}, \mathbf{q}$).

In terms of time complexity, it is useful to look at each iteration, at first. The majority of the operations consist of $\mathbf{O}(n)$ term-by-term vector computations, the exception being the matrix-vector product in line 10 ($\mathbf{q} \leftarrow \mathbf{Ad}$). For a dense, or randomly sparse, matrix \mathbf{A} , this step is $\mathbf{O}(n^2)$. This means that the overall time complexity, in this case, would be of the form $n_{\text{iterations}}\mathbf{O}(n^2)$. As the number of iterations grows with n, but is always less or equal to n [40, 32], it can be affirmed that the method is $\mathbf{O}(n^3)$. However, as we are dealing with structured meshes, \mathbf{A} is not dense nor randomly sparse. Every DOF has a constant number of neighbors in the mesh, meaning that every line in \mathbf{A} has a constant number of non-zero coefficients. If this is accounted for in the implementation of the matrix-vector product, its complexity drops to $\mathbf{O}(n)$. In turn, the overall time complexity of the PCG method for structured meshes (image-based models) is $\mathbf{O}(n^2)$.

4.2 Vector operations

The vector operations required by the PCG method are not dependent of the assemblyfree strategy, as the vectors involved are still properly assembled and stored in memory. Generally, these operations are quite conventional, and have well-established ways of implementation in massively parallel and vector-based environments. Term-by-term and scalar-vector computations are rather trivial to be coded. Algorithms 4 and 5, respectively, present pseudo-code for these sorts of tasks.

Algorithm 4 Massively parallel term-by-term vector operations

1:	Input:	$\mathbf{v},$	W
----	--------	---------------	---

- 2: Output: x
 3: Get thread index: id
 4: if this thread is within dimension of v and w then
- 5: $\mathbf{x}[id] \leftarrow operation(\mathbf{v}[id], \mathbf{w}[id])$
- 6: end if

Algorithm 5 Massively parallel scalar-vector operations

- Input: a, v
 Output: x
 Get thread index: id
 if this thread is within dimension of v then
- 5: $\mathbf{x}[id] \leftarrow operation(a, \mathbf{v}[id])$
- 6: end if

There is, however, one type of demanded vector operation that is not so simple, the dot products. Those are represented in Algorithm 3 as products of a transposed vector by another vector, such as $\mathbf{r}^T \mathbf{d}$, in line 7. This sort of computation can also be written as $\sum \mathbf{r}_i \mathbf{d}_i$. It is clear that a first step to compute this is to perform a term-byterm multiplication, but, after that, a parallel reduction scheme must be employed to sum up the entries of a resulting vector. As this is a very common operation in many applications, there are well-known solutions for it. It is customary, for example, that programming guides for GPUs tackle this sort of problem. The implementation adopted for dot products in this work follows what is proposed by Sanders and Kandrot [69], using the shared memory in GPU (addressed in Subsection 5.1.3), as described by the pseudocode in Algorithm 6.

```
Algorithm 6 Massively parallel dot product
 1: Input: v, w
 2: Output: res
 3: Get thread index: id, local id
 4: Initialize shared memory cache
 5: if this thread is within dimension of \mathbf{v} and \mathbf{w} then
        cache[local id] \leftarrow \mathbf{v}[id] * \mathbf{w}[id]
 6:
 7: else
        cache[local id] \leftarrow 0
 8:
 9: end if
10: Synchronize threads
11: stride \leftarrow cache dim/2
12: while stride > 0 do
        if local id < stride then
13:
            cache[local id] \leftarrow cache[local id] + cache[local id + stride]
14:
        end if
15:
        Synchronize threads
16:
        stride \leftarrow stride/2
17:
18: end while
19: res \leftarrow cache[0]
```

4.3 Matrix operations

The computations at each iteration of the PCG method that involve the global matrix, namely the initialization (or application) of the Jacobi preconditioner \mathbf{M} , and the matrix-vector product \mathbf{Ad} , demand special implementations to account for the proposed assembly-free solution. Even though the matrix is not stored in memory, it is not conceptually removed from the numerical process. The idea is that its non-null coefficients must be computed on-the-fly at each operation where they are required, via sweeps of the domain, as it has been previously discussed in Subsection 3.3.4. Two different strategies can be employed to perform such tasks in parallel, element-by-element or node-by-node. These are based on recent works for the massively parallel implementation of the PCG in GPU, such as Kiran et al. [37], Arbenz et al. [5], and Martínez-Frutos et al. [49].

4.3.1 Element-by-element

The element-by-element approach is based on the premise that every element on the mesh must provide its contributions to the DOFs of its nodes, as represented in Figure 4.2. A sequential implementation of this can be done with a loop over elements, but it is not desirable, as it is excessively intensive computation-wise. The goal is to maximize the number of elements that can be visited in parallel.



Figure 4.2: Element-by-element spreading of local coefficients to global DOFs

For the element-by-element sweep to be parallelized, it must be ensured that no concurrency might occur, that is, no more than one element can try to write data in memory spaces associated with a same global DOF at a time, as it is depicted in Figure 4.3. If this is not properly dealt with, there is no guarantee that inconsistent computations will not take place. Usually, this problem is tackled by establishing groups for parallel processing, running a coloring algorithm [12, 19, 49] to define sets of non-neighboring elements that can perform computations on their respective global DOFs without any risk of race conditions. However, focusing on the case of structured models, this solution generates additional memory allocation and pre-processing tasks that are unnecessary given that the regularity of the mesh is taken into account. Assuming this specific type of input, it is possible to work on every element simultaneously, on a conceptual level, without generating concurrency, by serializing the computations on each local DOF, or local node, which are few and constant, regardless of the size of the model. Figure 4.4 illustrates this notion.



Figure 4.3: Concurrency on element-by-element matrix-vector product



Figure 4.4: Non-conflicting access of local nodes on a pixel-based mesh (left), local node numbering (right)

Notice in Figure 4.4 that, if each element operates on the DOFs of its respective leftbottom node, no node is visited twice at that instruction, on a global level. Furthermore, only one corner and two non-opposite external edges of the model are accessed, so there is no trouble with the periodic boundary conditions. The same is valid for every other local node in 2D, and the principle is straightforwardly carried to 3D. The vital assumptions for this to work are that all elements of a given mesh share the same geometry, the mesh is structured in a grid-like fashion, and local node and DOF numbering are the same for every element. The first two are naturally met when working with image-based models, and the third is ensured, in this case, by adopting the rule that local nodes are numbered starting at the left-bottom node, growing counter-clockwise, then from near to far in 3D models.

It is clear, thus, that the element-by-element procedures can be seen as a sequence, on local nodes, of massively parallel instructions for elements. Algorithm 7 represents such notion at high-level, illustrating the process of an assembly-free matrix-vector multiplication.

Algorithm 7 can be plugged in the PCG method to meet the demands of matrix operations on an assembly-free approach. Not only does it solve matrix-vector multiplication operations, but a simplified variation of it can also be used for the assembly of the Jacobi preconditioner. It is important to state that the actual implementation of Algorithm 7 can be done as a constant, hard-coded, sequence of element-by-element operations, as the numbers of local nodes and of nodal DOFs are rather small and not dependent of the size of the model.

Algorithm 7 Image-based element-by-element matrix-vector multiplication

1:	Input: d , local matrices, material map		
2:	Output: q		
3:	for $(n = 0, 1, \dots, \text{number of local nodes} - 1)$ do		
4:	Element-by-element parallel processing of local node $n\{$		
5:	Get this element's local matrix with material map (8-bit image): \mathbf{k}		
6:	Get this element's DOFs (via computations with index): e_{dofs}		
7:	row $\leftarrow n * (number of DOFs per node)$		
8:	for i in DOFs of local node n do		
9:	$\operatorname{col} \leftarrow 0$		
10:	$\mathbf{for} \ \mathbf{j} \ \mathbf{in} \ \mathbf{e} \ \mathbf{dofs} \ \mathbf{do}$		
11:	$\mathbf{q}[\mathrm{i}] \leftarrow \mathbf{q}[\mathrm{i}] + \mathbf{k}[\mathrm{row}][\mathrm{col}] \ast \mathbf{d}[\mathrm{j}]$		
12:	$\operatorname{col} \leftarrow \operatorname{col} + 1$		
13:	end for		
14:	$\operatorname{row} \leftarrow \operatorname{row} + 1$		
15:	end for		
16:	}		
17:	end for		

4.3.2 Node-by-node

The idea behind the node-by-node strategy is that every node of the mesh must consult its respective neighboring elements to gather their contributions to its own DOFs, as depicted in Figure 4.5. The computations that must be performed are unchanged, in comparison to the element-by-element approach, however, by organizing them in terms of nodes, the possibility of concurrency is naturally eliminated. Each thread may access data associated with neighboring nodes to read, but only writes on its own respective memory spaces. As the coefficients to be calculated are related to nodal DOFs, it makes sense that nodal instructions should be responsible for the computations.

It is interesting to notice that, when working with image-based models, the node-bynode approach does not require that nodes know their respective neighboring elements, just their material keys are enough. This is due to fact that local matrices are associated with specific materials in the heterogeneous domain, instead of specific elements.

By considering the indexing rules presented in Section 3.4, each thread on a node-bynode procedure could obtain all the demanded material keys from the 8-bit image array. Even so, it often is desired to avoid doing that, as coalesced data access, an important matter to gain performance in GPUs, would be lost. Therefore, a new material map



Figure 4.5: Node-by-node gathering of local coefficients to global DOFs

can be generated, combining the color values of the elements that surround each node to compose a single material key that describes its entire vicinity. The caveat is that this usually imposes a reduction in the range of representable different materials in the micro-scale. For instance, if a 16-bit map is employed, the range reduces from 8-bit to 4-bit in 2D, or 2-bit in 3D, as there are, respectively, 4 or 8 neighboring material keys to be represented with a single 16-bit value. Analogously, to fully depict the 8-bit range in 3D, a 64-bit map would be necessary, which is also undesirable, as, for sufficiently high resolution models, it would take significant memory space on the GPU that could be better used to allocate the arrays demanded by the PCG method.

There is no definitive answer as to how the material keys around nodes should be stored. In problems where few different materials are considered in the micro-scale, it seemingly would be better to employ a data structure such as the 16-bit map. Oppositely, to analyze models that depend on the 0-255 range to fully characterize the heterogeneities of the micro-scale, it might make sense to go for the 64-bit map, or even to work directly with the element-indexed 8-bit image, making a performance compromise to spare memory. Alternatively, it should be beneficial to store the 8-bit image in texture memory caches, which will be explored in future work. Henceforth, it will be assumed that a nodal material map is used, but it should be noted that the proposed node-by-node strategy is not strongly constrained by this, and would work, with some minor adaptations, for alternative cases as well. A high-level description of the adopted node-by-node approach is presented in Algorithm 8, which exposes a procedure for matrix-vector products.

Alg	Algorithm 8 Image-based node-by-node matrix-vector multiplication		
1:	Input: d, local matrices, material map		
2:	Output: q		
3:	Node-by-node parallel processing{		
4:	Get this node's material key from map		
5:	for $(e = 0, 1,, \text{number of neighboring elements} - 1)$ do		
6:	Get local matrix of local element e using node material key: ${f k}$		
7:	Get DOFs of element e (via computations with index): e_dofs		
8:	row $\leftarrow e * (\text{number of DOFs per node})$		
9:	for i in DOFs of this node do		
10:	$\operatorname{col} \leftarrow 0$		
11:	$\mathbf{for} \ \mathbf{j} \ \mathbf{in} \ \mathbf{e}_\mathbf{dofs} \ \mathbf{do}$		
12:	$\mathbf{q}[\mathrm{i}] \leftarrow \mathbf{q}[\mathrm{i}] + \mathbf{k}[\mathrm{row}][\mathrm{col}] * \mathbf{d}[\mathrm{j}]$		
13:	$\operatorname{col} \leftarrow \operatorname{col} + 1$		
14:	end for		
15:	$row \leftarrow row + 1$		
16:	end for		
17:	end for		
18:	}		

Similarly to the element-by-element scheme, when implementing Algorithm 8, the for loops can be substituted by hard-coded steps, as they are small and not associated with the dimensions of the model. The loop over neighboring elements always runs 4 or 8 times (2D or 3D, respectively), the number of DOFs per node is a property of the physical problem being analyzed (1 for thermal conductivity, 3 for elasticity), and the dimensions of the local matrices are constant, proportional to the number of nodes that each element has.

It is worth noticing that an even more fine-grained variation of the node-by-node strategy can be achieved for models with multiple DOFs per node, implementing a DOF-by-DOF solution. Although this could be seen as a third option for the sweeps of the domain, in this work it will not be treated as such, due to it being, in essence, akin to the node-by-node strategy. The only difference, in Algorithm 8, would be the removal of the loop over "DOFs of this node", as each thread would only compute on data associated with a single DOF.

4.4 Massively parallel PCG for assembly-free FEM

Bearing in mind the topics discussed throughout this Chapter, it is possible to rewrite the well-known PCG method, presented in Algorithm 3, to better suit the considerations of the proposed assembly-free image-based finite element analysis, as portrayed in Algorithm 9. The expression shown in Equation 3.29, which can be viewed as a mathematical description of the computation of global coefficients on-the-fly in terms of local matrices, is used to represent the massively parallel matrix operations. As proposed in Section 4.1, the vector \mathbf{s} is stored in the \mathbf{q} array. All of the vector operations are admitted to be performed accordingly to Algorithms 4, 5 and 6.

Alg	Algorithm 9 Massively parallel assembly-free PCG		
1:	Input: local matrices : \mathbf{k} , material map, \mathbf{b} , \mathbf{x}_0 , tolerance		
2:	Output: x		
3:	Initialize preconditioner (Jacobi): $\mathbf{M} \leftarrow \sum \mathbf{T}_{e}^{T} \operatorname{diag}(\mathbf{k}_{e}) \mathbf{T}_{e}$		
4:	$\mathbf{x} \leftarrow \mathbf{x}_0$		
5:	$\mathbf{r} \leftarrow \mathbf{b} - (\sum \mathbf{T}_{\mathrm{e}}^T \mathbf{k}_{\mathrm{e}} \mathbf{T}_{\mathrm{e}}) \mathbf{x}$		
6:	$\mathbf{d} \leftarrow \mathbf{M}^{-1} \mathbf{r}$		
7:	$\delta \leftarrow \mathbf{r}^T \mathbf{d}$		
8:	$\delta_0 \leftarrow \delta$		
9:	while $(\delta/\delta_0 > \text{tolerance}^2)$ do		
10:	$\mathbf{q} \leftarrow (\sum \mathbf{T}_{\mathrm{e}}^T \mathbf{k}_{\mathrm{e}} \mathbf{T}_{\mathrm{e}}) \mathbf{d}$		
11:	$lpha \leftarrow \delta / \mathbf{q}^T \mathbf{d}$		
12:	$\mathbf{x} \leftarrow \mathbf{x} + lpha \mathbf{d}$		
13:	$\mathbf{r} \leftarrow \mathbf{r} - \alpha \mathbf{q}$		
14:	$\mathbf{q} \leftarrow \mathbf{M}^{-1} \mathbf{r}$		
15:	$\delta_{ ext{prev}} \leftarrow \delta$		
16:	$\delta \leftarrow \mathbf{r}^T \mathbf{q}$		
17:	$eta \leftarrow \delta/\delta_{ ext{prev}}$		
18:	$\mathbf{d} \leftarrow \mathbf{q} + eta \mathbf{d}$		
19:	end while		

In Algorithm 9, it is noticeable that allocated memory can be further reduced by not storing the preconditioner **M**. Similarly to what was done to the global matrix **A** in lines 5 and 10, the expression $\sum \mathbf{T}_{e}^{T} \operatorname{diag}(\mathbf{k}_{e})\mathbf{T}_{e}$ can be substituted in every operation where **M** is demanded, lines 6 and 14. This means that a solution with the adopted massively parallel PCG can be obtained with four arrays $(\mathbf{x}, \mathbf{r}, \mathbf{d}, \mathbf{q})$. Furthermore, it is interesting to notice that, even though the computational cost has been elevated, the time complexity of each iteration remains at $\mathbf{O}(n)$.

4.5 Parallel PCG solver in CPU

In this Section, two implementations in CPU of the proposed parallel PCG solver for assembly-free image-based FEM problems are presented. The first, discussed in Section 4.5.1, consists of a vectorized program in MATLAB, with concise script code, designed to ascertain the validity of the solution without storing a global matrix. Then, in Section 4.5.2, the same program has its performance metrics compared against a solution that assembles the system of equations, also in MATLAB, with the goal of making evident the trade-off between memory and time. Finally, in Section 4.5.3, a more efficient program is shown, written in C, employing multi-thread parallelism with OpenMP to perform the on-the-fly matrix operations. Both solvers presented here effectively implement Algorithm 9, and consider the element-by-element strategy, presented in Subsection 4.3.1.

4.5.1 A vectorized approach

Modern vector-based languages, such as MATLAB, Octave, and Python (with Numpy), are optimized to work with matrix and vector operations and indexing, instead of for loops [52, 51, 16]. The elapsed time for vectorized solutions usually is considerably reduced from sequential ones. In programming environments like these, it makes sense to write concise script dealing with large arrays to invoke computationally intensive operations with few command lines [3], oppositely to performing a loop over all instances of the problem at hand. Attention is needed, however, to ensure that the aforementioned intensive operations avoid race conditions, or else, inconsistent calculations might take place and performance can be compromised. In that sense, the vectorization of this sort of code holds some conceptual similarities to parallel programming.

The massively parallel strategies presented in Section 4.3 are fitting to be implemented with MATLAB's vectorization. Focusing on the element-by-element solution, the idea is to index local matrices with the material map array, so that, instead of explicitly calling several threads, the parallel computations can be written as large vector operations. Implicitly, CPU parallelism resources are employed. Listings with vectorized code for assembly-free matrix operations are presented ahead. It is considered the analysis of 2D heat conduction, for conciseness.

The generation of the Jacobi preconditioner consists of computing the coefficients at the main diagonal of the global matrix and storing them in an array. This operation, described in Listing 4.1, demands a sweep of the domain due to the assembly-free approach. The employment of a gatherer vector, depicted as \mathbf{g} , makes it possible to get all contributions from the local matrices \mathbf{K} of elements on each of its DOFs, accordingly to the strategy illustrated in Figure 4.4, and add them to the resulting preconditioner array \mathbf{M} without generating excessive memory overheads. It is interesting to notice, in line 2, how an array of node indexes \mathbf{n} can be computed in terms of respective element indexes, and it can be later manipulated using vector operations to access neighboring nodes. At the end of the procedure, all the coefficients in \mathbf{M} are inverted, so that the stored array represents the matrix \mathbf{M}^{-1} , needed on every iteration of the PCG method.

```
||e| = 1: nElems;
|n| = e+uint64(floor(double(e-1)/double(nRows)))+1;
|\mathbf{g}| = \mathbf{z} \mathbf{e} \mathbf{r} \mathbf{o} \mathbf{s} (\mathbf{n} \mathbf{E} \mathbf{l} \mathbf{e} \mathbf{m} \mathbf{s}, \mathbf{1}, \mathbf{d} \mathbf{o} \mathbf{u} \mathbf{b} \mathbf{l} \mathbf{e}');
_{4} M = zeros (nDOFs, 1, 'double');
_{5} g(:) = K(1, 1, elemMatMap);
_{6} M(DOFMap(n)) = M(DOFMap(n)) + g;
_{7}|n = n + nRows + 1;
|\mathbf{g}(:)| = \mathbf{K}(2, 2, \text{elemMatMap});
M(DOFMap(n)) = M(DOFMap(n)) + g;
|n| = n - 1;
|_{11}| g(:) = K(3,3,elemMatMap);
_{12}|M(DOFMap(n)) = M(DOFMap(n)) + g;
|n| = n - (nRows + 1);
_{14} g(:) = K(4,4,elemMatMap);
_{15} M(DOFMap(n)) = M(DOFMap(n)) + g;
_{16}|M = M.^{-1};
```

Listing 4.1: Vectorized program in MATLAB for the assembly of the Jacobi preconditioner

In Listing 4.1, **DOFMap** is an array that maps DOF indexes to node indexes, reproducing the periodic numbering scheme shown in Figure 3.6. **elemMatMap** essentially represents the image itself, acting as a map of material keys that links elements to their respective physical properties. The variables nRows, nElems and nDOFs, respectively denote the numbers of rows, elements and DOFs in the mesh. It should be noted that the four operations performed through lines 5 to 15 could be placed in a constant-sized **for** loop.

The implementation of the matrix-vector product, presented in Listing 4.2, follows the proposed pseudo-code on Algorithm 7. Operations are serialized in local nodes so that every element can be visited simultaneously without concurrency issues. Ideally, an infinitely parallel machine would be able to perform this operation in constant time. The gatherer vector \mathbf{g} is filled with entries from the local conductivity matrix of every element, according to the current local node, so that it can be used in a term-by-term vector multiplication with the corresponding entries of the search direction vector \mathbf{d} . The result is then added to vector \mathbf{q} .

```
1 | q(:) = 0; n = n + 1;
|_{2}|_{g}(:) = K(1, 1, elemMatMap);
||_{3}| q (DOFMap(n)) = q (DOFMap(n)) + g \cdot * d (DOFMap(n));
|_{4}|_{g(:)} = K(1, 2, elemMatMap);
[5] q(DOFMap(n)) = q(DOFMap(n)) + g \cdot * d(DOFMap(n+(nRows+1)));
_{6}|g(:)| = K(1, 3, elemMatMap);
_{7} | q (DOFMap(n)) = q (DOFMap(n)) + g . * d (DOFMap(n+nRows));
||g(:)| = K(1, 4, elemMatMap);
|q(DOFMap(n))=q(DOFMap(n))+g.*d(DOFMap(n-1));
|_{10}|_{n} = n + (nRows + 1);
||_{11}| g(:) = K(2, 1, elemMatMap);
|12|q(DOFMap(n)) = q(DOFMap(n)) + g \cdot * d(DOFMap(n-(nRows+1)));
|_{13}| g (:) = K(2, 2, elemMatMap);
|q(DOFMap(n))=q(DOFMap(n))+g.*d(DOFMap(n));
_{15} g(:) = K(2,3,elemMatMap);
|q(DOFMap(n))| = q(DOFMap(n)) + g \cdot * d(DOFMap(n-1));
_{17} g (:) = K(2, 4, elemMatMap);
|q(DOFMap(n))| = q(DOFMap(n)) + g \cdot * d(DOFMap(n-(nRows+2)));
19 n = n-1;
|g(:)| = K(3, 1, elemMatMap);
_{21} q (DOFMap(n))=q (DOFMap(n))+g.*d (DOFMap(n-nRows));
_{22} | g(:) = K(3,2,elemMatMap);
_{23} | q (DOFMap(n))=q (DOFMap(n))+g . * d (DOFMap(n+1));
_{24} | g (:) = K(3, 3, elemMatMap);
|q(\text{DOFMap}(n))| = q(\text{DOFMap}(n)) + g \cdot * d(\text{DOFMap}(n));
_{26} | g (:) = K(3, 4, elemMatMap);
_{27} | q (DOFMap(n)) = q (DOFMap(n)) + g . * d (DOFMap(n-(nRows+1)));
_{28} n = n-(nRows+1);
_{29} g (:) = K(4, 1, elemMatMap);
|q(DOFMap(n))| = q(DOFMap(n)) + g \cdot * d(DOFMap(n+1));
|g(:)| = K(4, 2, elemMatMap);
_{32} | q (DOFMap(n)) = q (DOFMap(n)) + g . * d (DOFMap(n+(nRows+2)));
```

 $\begin{array}{l} {}_{33} \\ {\rm g}(:) &= {\rm K}(4\,,\!3\,,\!{\rm elemMatMap})\,; \\ {}_{34} \\ {\rm q}({\rm DOFMap}(n\,)\,) \!=\! {\rm q}({\rm DOFMap}(n\,)\,) \!+\! {\rm g}\,.\,*\,{\rm d}\,({\rm DOFMap}(n\!+\!(n{\rm Rows}\!+\!1)\,)\,)\,; \\ {}_{35} \\ {\rm g}\,(:) &= {\rm K}(4\,,\!4\,,\!{\rm elemMatMap})\,; \\ {}_{36} \\ {\rm q}\,({\rm DOFMap}(n\,)\,) \!=\! {\rm q}\,({\rm DOFMap}(n\,)\,) \!+\! {\rm g}\,.\,*\,{\rm d}\,({\rm DOFMap}(n\,)\,)\,; \\ {}_{37} \\ {\rm n}\,=\,{\rm n}\,+\,1; \end{array}$

Listing 4.2: Vectorized program in MATLAB for the computation of matrix-vector product with an element-by-element strategy

4.5.2 Memory vs. Time trade-off

In this Section, time and memory metrics are presented for analyses made with the vectorized assembly-free solution, called **vhifem**, comparing it with a variation of the solver developed by Andreassen and Andreasen [2], which assembles the global matrix in sparse form, also using vectorization. Originally, the latter employed the **mldivide** function (commonly called the *backslash*), which is a direct solver, so it was modified to use the **pcg** function, with the Jacobi preconditioner as well. The idea is to solidify the approached concepts with data and to visualize the trade-off between allocating or not the global matrix.

The studied model is similar to the analytical benchmark that will be used ahead for validation, in Chapter 7, presented in Figure 7.1. Analyses of 2D elasticity were conducted. Adopted dimensions for the periodic cells ranged from 50×50 pixels to $500 \times$ 500 pixels. Both material phases were considered to be isotropic, with different elastic properties. A numerical tolerance of 1e-04 for dimensionless relative norms of residuals of the PCG method was admitted, which leads to equivalent results being obtained with both programs. However, the focus in this case is on performance.

For time metrics, MATLAB resources were employed. The elapsed time for each run was taken with the tic and toc functions, called right before and after the program in question was called. Then, the average time for each PCG iteration was obtained as $(total\ time)/(num\ of\ iterations)$. Memory metrics were obtained with the whos function placed at strategic points within the code, and were complemented with data from the operating system regarding allocated DRAM throughout the analyses, to account for the memory usage of built-in functions. Figure 4.6 depicts a plot of memory allocation against number of DOFs, while Figure 4.7 presents time per PCG iteration. Furthermore, Figure 4.8 shows the total elapsed time of analysis, comparing the vectorized element-by-element implementation with a sequential version of it (seqhifem), which uses for loops. All the

data were obtained from running the programs on MATLAB 2019a, on a Linux Mint 19.3 Tricia operating system, using an $\text{Intel}^{\mathbb{R}}$ Core^{\mathbb{M}} i7-7500U dual-core CPU, with a clock rate of 2.70 GHz and 8 GB of available DRAM.



Memory allocation for PCG solutions with and without a stored global matrix

Figure 4.6: Peak Memory Allocation [MB] vs. number of DOFs, comparing vectorized assembly-free to stored matrix solutions using the PCG method

In Figure 4.6, it is shown that the assembly of the global matrix is indeed memoryconsuming, even accounting for sparsity and employing the PCG iterative solver. As the meshes are structured, linear space complexity is achieved in both studied cases. The assembly-free solution is an alternative strategy to enable large-scale simulations with low memory cost, demanding more computations in exchange, so runtime is increased, as it can be seen in Figure 4.7. The presented vectorized approach is an optimized way of implementation in MATLAB that considerably improves performance for the element-byelement solution, in comparison to an akin sequential procedure, as depicted in Figure 4.8. This is an interesting indicative that a massively parallel solution in GPU should perform better than CPU solvers. However, remaining on CPU implementations for now, it is clear that solving problems with millions of DOFs is a time-consuming task. When adopting an interpreted language, even with an efficient program, the computations for the aimed model dimensions could often take days, if not weeks, to run on personal computers. Migrating to a compiled language is essential. Bearing this in mind, the next Subsection presents an analogous solution to the one considered so far, now written in C.



Time metrics with and without a stored global matrix

Figure 4.7: Time per PCG iteration [s] vs. number of DOFs, comparing vectorized assembly-free to stored matrix solutions using the PCG method



Figure 4.8: Total elapsed time [s] vs. number of DOFs, comparing vectorized and sequential assembly-free solutions

4.5.3 Multi-thread implementation

The element-by-element strategy presented in Subsection 4.3.1 was also implemented in C, using OpenMP compiler directives to perform parallel loops over elements in the assembly-free matrix. All of the vector operations are performed with parallel loops as well. The developed program was based on prior C++ code, used in works such as Pereira et al [61]. The main differences of this solution to its predecessor are not allocating a connectivity data structure, and not using coloring algorithms to establish groups of elements for parallel processing. The idea of this implementation in CPU is to serve as a ground truth when comparing performance between CPU and GPU solutions.

The matrix-vector product is analogous to the vectorized implementation presented in Subsection 4.5.1. For each local node, a parallel loop over elements is employed to gather contributions of local coefficients to the DOFs of that specific node. Listing 4.3 details this procedure. Once again, it is considered the analysis of 2D heat conduction for conciseness, but it should be noted that for all the other sorts of analyses within the scope of this work the procedure is analogous.

```
void Aprod thermal 2D(var * d, unsigned int sz, var * q){
      unsigned int n, dof; var *thisK;
2
      #pragma omp parallel for
3
      for (n=0; n < nDOFs; n++) q[n] = 0.0;
      // Constant-sized loop over local nodes
E
      for (unsigned int local node = 0; local node < 4; local node++){
6
          #pragma omp parallel for private(n,dof,local_node,thisK)
          for (unsigned int e=0; e< nElems; e++)
               this K = \&K[elemMatMap[e]*16+4*local node];
ç
              n = e+1+(e/nRows);
10
               dof = DOFMap[n+WALK 2D(local node)];
              q[dof] += thisK[0] * v[DOFMap[n]]; n+=nRows+1;
12
              q[dof] += thisK[1] * v[DOFMap[n]]; n=1;
13
              q[dof] += thisK[2] * v[DOFMap[n]]; n==nRows+1;
14
              q[dof] += thisK[3] * v[DOFMap[n]];
15
          }
16
      }
      return:
18
19 }
```

Listing 4.3: Multi-thread C program for the matrix-vector product with an element-byelement strategy

	Size of variable	Array dimension	
$\mathbf{x}, \mathbf{r}, \mathbf{d}, \mathbf{M}, \mathbf{q}$	8 Bytes	n	
DOFMap	4 Bytes	$n_{ m nodes} pprox n/n_{ m nodal DOFs}$	
elemMatMap	1 Byte	$n_{ m elems} = n/n_{ m nodalDOFs}$	
local matrices \mathbf{K}	8 Bytes	$(\text{const.})n_{\text{materials}} \ll n$	
Total [Bytes]			
$5 \times 8n + 4n_{\text{nodes}} + n_{\text{elems}} + 8(\text{small constant})$			

Table 4.1: Memory allocated for parallel assembly-free PCG solver in CPU

In Listing 4.3, the variables are equivalent to those in Listing 4.1. It stands out, however, that the local matrices **K** are now stored in a one dimensional array, which explains why the material key is multiplied by 16 (the number of coefficients in each local matrix). The operation WALK_2D is implemented as a macro, which defines the shift from local node 0 to the current node being run. Regarding OpenMP, The directive **#pragma omp parallel for** is employed to signalize to the gcc compiler that a specific for loop must be split into multiple parallel CPU threads. The private clause makes sure that each thread works with local instances of the specified variables.

Results and performance metrics for this implementation will be presented in Chapter 7. This CPU solution does improve performance from the aforementioned vectorized code, but it still does not reach the full potential of the developed methodologies. It is evident that the strategies summarized in Algorithms 7 and 8 are suitable for massively parallel environments. As we are dealing with large-scale simulations, and procedures have been conceived to avoid concurrency, it is desirable that the computations are as parallelized as made possible by the available hardware. In that sense, it is logical to expect significant runtime reduction with a solver implemented in GPU. This will be discussed in details in the next Chapter.

As memory is a limiting factor to the numerical analyses, the space efficiency of solutions is a matter of concern. Table 4.1 presents a summary of the sizes of the significant demanded arrays, in terms of n (the number of global DOFs), by the proposed multi-thread element-by-element solver. It is noticeable that memory allocation grows linearly with respect to the dimension of the model.

Chapter 5

Massively parallel implementation in GPU

This Chapter presents the implementation in CUDA C of a massively parallel PCG solver applied to assembly-free image-based FEM. Section 5.1 covers an introduction to GPU programming, pointing out some of the main concepts involved and the differences from CPU programming. Section 5.2 details the implementation of the PCG method in GPU, from the kernels for matrix operations, admitting element-by-element and node-by-node approaches, to memory allocation and data transfer, two essential topics in this work. At last, after considerations regarding the proposed solution have been made, Section 5.3 presents 4 alternative ways of implementation, with the main objective of reducing memory allocation in the GPU, to extend the limits of models that can be analyzed.

5.1 General purpose programming with GPUs

This Section is based on the books on GPU programming written by Kirk and Hwu [38], and Sanders and Kandrot [69]. The CUDA documentation is also a crucial reference [58].

Over the last two decades, a notorious evolution in GPU programming has been occurring. Interesting developments can be seen in fields such as Scientific Computing, Blockchain, and Artificial Intelligence. The last one is actually becoming so intertwined with usage of GPUs, that Nvidia itself, a company known for their efforts in Computer Graphics, is gradually moving its main focus to AI, for example. This is one of many signs that computing with GPUs is a promising trend that will continue to be explored for years to come.

5.1.1 An overview

The Graphics Processing Unit (GPU) was conceived as a massively parallel machine to accelerate rendering of images. In a nutshell, its main original purpose was to receive coalesced data referring to geometry and color, process those through an ordered set of instructions being run simultaneously on multiple Arithmetic Logic Units (ALUs), and obtain final colors to paint each pixel on screen. This is commonly denominated the graphic pipeline. It was not imperative that the ALUs were particularly powerful, as the usual computations demanded of them were not supposed to be too strenuous or complex, but it was essential that as many as possible computations could be made in parallel, so that pixels could be updated in real time. In light of this, the GPU architecture was designed as illustrated in Figure 5.1, where schematics for a CPU and a GPU are depicted. GPUs rely on parallelism, which means that increasing its capabilities is, to simply put it, a matter of fitting more ALUs into the hardware without compromising data access. On the other hand, CPUs are heavily dependent of a strong control unit, with relative little parallelism capacity, so efforts are usually focused on increasing its clock rate.



Figure 5.1: CPU vs. GPU architecture. Based on Kirk and Hwu [38].

It is important to notice, in Figure 5.1, that several ALUs in the GPU share a same control unit and a memory cache. From this stems the Single Instruction Multiple Threads (SIMT) paradigm, as the same program runs on multiple processors, who are responsible for identifying which piece of data they must work on, through software.

Returning to the graphic pipeline, the massively parallel instructions were constrained

at first, often being implemented in hardware, but as Computer Graphics developed, some of them became open to programmers, so that more sophisticated lighting and coloring techniques, for example, could be implemented in software to be run directly in GPU. This is known as *shader* programming, and it has become a standard practice. In fact, the current version of the OpenGL graphic pipeline demands vertex and fragment shaders implemented in software. These shaders receive input data such as color triplets, vertex positions in 3D, and normal vectors for lighting, outputting similar parameters. It eventually became clear that essentially any type of data could be passed to the GPU to be processed with shaders by, for the lack of a better term, tricking the hardware into assuming its working with graphic properties. This opened the door for general purpose usage of the massively parallel capabilities of GPUs, enabling drastic performance gains in non-graphic applications that demand large computations. In this context, tools were created to formalize GPGPU (General Purpose GPU) programming, ending the need for workarounds with colors and vertex positions for programmers interested in exploring GPUs as computation accelerators, rather than doing anything graphic. The most famous two would be the Open Computing Language (OpenCL) and the Compute Unified Device Architecture (CUDA), where the second is the one adopted in this work. The concept of these frameworks is to enable a CPU to communicate with GPUs, for tasks such as calling functions, allocating memory, and transferring data. The next Subsection presents some details of the CUDA environment, focusing on key topics for the implementation proposed in this text.

5.1.2 CUDA programming model

The CUDA API is a proprietary software from Nvidia dedicated to enable usage of their graphics cards for GPGPU programming. In this work, the language of focus is CUDA C, a hybrid programming language, which employs the CUDA API, for the implementation of CPU and GPU code in similar fashion to standard C/C++. It should be noted, however, that CUDA also supports FORTRAN. Although the language is aesthetically familiar to conventional CPU code in C, some specific terms and notions of working with GPUs are vital for implementing CUDA applications. These will be addressed ahead.

- The CPU is referred as **host**, while the GPU is called **device**.
- Streaming multiprocessors (SMs) are groups of CUDA cores that are used to execute multiple blocks of threads.

- Threads are independent instances of sequential flow within a program. The massively parallel nature of GPUs is due to its capacity to manage several threads at a time. In CUDA, threads are structured in blocks, and are executed in waves of 32, denominated warps.
- Blocks are groups of a maximum of 1024 threads. They can be abstractly characterized as 1D, 2D or 3D, to better represent the data worked on. Usually, they are dimensioned with multiples of 32 threads, to conform with warp sizes. Each block is assigned to a single SM. Groups of blocks are contained within a grid. Figure 5.2 depicts where threads, blocks and grids are run, in hardware.
- Kernels are functions that run in GPU. A kernel call initiates multiple blocks to run parallel instances of the programmed instructions. Generally, kernels follow a pattern of evaluating the current thread index, checking if it is within data bounds, and then doing whatever operation is required. In code, the keyword __global__ is placed before the function signature to state that it is a kernel, visible to host and device.
- Streams are queues of GPU operations to be executed orderly, employed to achieve task parallelism with GPUs. They are usually adopted to differentiate groups of calls that can be run independently of each other. It is possible, for example, to assign some streams to handle data transfers between **host** and **device**, while others deal with massively parallel computations, in an effort to avoid idleness of hardware.

The **host** controls the sequential flow of the program, being responsible for calling **kernels**, and managing memory allocations and data transfers in all of the involved hardware. It is important to be aware that many **device** calls are non-blocking to the **host**, so, for instance, calling a **kernel** does not halt the program until results are computed, nor are those automatically sent to the **host**. Explicit **host-device** synchronization and/or data transfers are commonly required, which can be done via API calls such as **cudaMemcpy**. Figure 5.3 illustrates a typical flow of action of a CUDA C program.

5.1.3 GPU memory

GPUs have a distinct memory hierarchy that impacts on how threads access data. Considerable performance gains can be obtained with proper use of local, shared and global memory. Figure 5.4 represents the memory resources in GPUs. Each of the memories are detailed in the following bullet items.



Figure 5.2: Relation of threads, blocks and grids to hardware. Source: Gupta [27].



Figure 5.3: Usual flow of activities of a CUDA C program



Figure 5.4: Memory hierarchy of a CUDA-enabled device. Based on Gupta [27].

- The global memory is accessible to every active thread in a kernel run. It corresponds to the DRAM of the GPU, having the most space, but requiring more machine cycles for data to be accessed.
- The L2 cache is shared among all SMs, so every thread can also reach it. It permits faster memory access, but has limited space.
- The L1/Shared memory is an on-chip fast-access memory space that can be reached by every thread within each specific SM. This is particularly useful for operations such as sum reductions, where threads in a block need to share some data among themselves. It is limited in size, but nevertheless a powerful tool for parallel algorithms that need to be aware of neighboring data.
- The Read-only memory is an on-chip cache commonly employed for constant or texture memory. It can speed up access to constant-value small-sized data that needs to be visible to threads.
- The Registers are used to store local variables for each thread, and have their usage controlled at compiler level.

The GPU is optimized to work with coalesced data, that is, consecutive threads accessing aligned consecutive memory spaces, as depicted in Figure 5.5. This is because consulting the global DRAM from a single processor is relatively time-consuming, and if the data is coalesced, it is possible to carry portions of it at a time, instead of just one piece of information.



Figure 5.5: Coalesced (top) and uncoalesced (bottom) memory access of an array with entries of 4 Bytes

Figure 5.5 is a common example found in GPU programming guides for memory alignment and coalesced access. The access performed at the top of the Figure requires a single operation, as the moved 128 Bytes are cached for the 32 consecutive threads, while the procedure at the bottom requires multiple memory transactions, filling the cache with unused data. Ideally, to extract the most performance from a GPU, all of the arrays stored in global memory should be arranged so that a thread identified by an index i accesses memory spaces of index i. Of course, this is desired but not always possible.

5.2 PCG applied to assembly-free image-based FEM in GPU

The pseudo-code shown in Section 4.4 is ready for massively parallel implementation. The solutions in CPU presented so far do explore some parallelism to reduce the elapsed time of the analyses, but they barely scratch the surface on the potential of performance gain with the proposed methodologies. Ideally, with an infinitely parallel machine, the element-by-element and node-by-node strategies discussed in Section 4.3 would take constant time, as they allow for parallelization in the whole domain of analysis. In that sense, GPUs appear to be far better fitting hardware for this sort of numerical simulation, in comparison to CPUs.

For the implementation in CUDA C, a slight variation of Algorithm 9 is proposed. The only difference in the adopted solution, represented by Algorithm 10, is that the preconditioner array \mathbf{M} is not assembled, as suggested in Section 4.4. Every operation $\mathbf{M}^{-1}\mathbf{r}$ now demands sweeps of the domain as well, becoming analogous to the assembly-
free matrix-vector products. This choice was based on the memory limitations of the available GPUs for this work (a maximum of 8 GB DRAM), as it reduces the number of allocated arrays in the device from five to four. The procedure described in Algorithm 10 was the first one implemented in CUDA during the development of this work, and it will be henceforth referred as the MParPCG solver in GPU for assembly-free image-based FEM. The host handles kernel calls and convergence of the numerical method.

```
Algorithm 10 MParPCG solver applied to assembly-free image-based FEM in GPU
```

```
1: Input: local matrices : \mathbf{k}, material map, \mathbf{b}, \mathbf{x}_0, tolerance
  2: Output: x
  3: \mathbf{x} \leftarrow \mathbf{x}_0
  4: \mathbf{r} \leftarrow \mathbf{b} - (\sum \mathbf{T}_{e}^{T} \mathbf{k}_{e} \mathbf{T}_{e}) \mathbf{x}
                                                                                                                                                                   ▷ Aprod kernel
 5: \mathbf{d} \leftarrow (\sum \mathbf{T}_{e}^{T} \operatorname{diag}(\mathbf{k}_{e}) \mathbf{T}_{e})^{-1} \mathbf{r}
                                                                                                                              ▷ applyPreConditioner kernel
  6: \delta \leftarrow \mathbf{r}^T \mathbf{d}
                                                                                                          \triangleright dotprod kernel, sync. host and device
  7: \delta_0 \leftarrow \delta
  8: while (\delta/\delta_0 > \text{tolerance}^2) do
                                                                                                                                                                 \triangleright check in CPU
               \mathbf{q} \leftarrow (\sum^{T} \mathbf{T}_{\mathrm{e}}^{T} \mathbf{k}_{\mathrm{e}} \mathbf{T}_{\mathrm{e}}) \mathbf{d}
                                                                                                                                                                   ▷ Aprod kernel
  9:
               \alpha \leftarrow \delta / \mathbf{q}^T \mathbf{d}
                                                                                                          \triangleright dotprod kernel, sync. host and device
10:
               \mathbf{x} \leftarrow \mathbf{x} + \alpha \mathbf{d}
                                                                                                                                                                ⊳ sumVec kernel
11:
               \mathbf{r} \leftarrow \mathbf{r} - \alpha \mathbf{q}
12:
                                                                                                                                                                ⊳ sumVec kernel
               \mathbf{q} \leftarrow (\sum \mathbf{T}_{e}^{T} diag(\mathbf{k}_{e}) \mathbf{T}_{e})^{-1} \mathbf{r}
                                                                                                                              ▷ applyPreConditioner kernel
13:
14:
               \delta_{\text{prev}} \leftarrow \delta
               \delta \leftarrow \mathbf{r}^T \mathbf{q}
                                                                                                          \triangleright dotprod kernel, sync. host and device
15:
               \beta \leftarrow \delta / \delta_{\text{prev}}
16:
               \mathbf{d} \leftarrow \mathbf{q} + \beta \mathbf{d}
                                                                                                                                                                ⊳ sumVec kernel
17:
18: end while
```

5.2.1 Kernels for assembly-free matrix operations

The kernels for assembly-free matrix operations follow Algorithms 7 and 8, implementing a hard-coded solution for each of the considered physical phenomena. As it was done in Section 4.5, code is presented referring to 2D heat conduction, for the sake of conciseness. Analogous kernels were implemented as well for 3D, and for elasticity problems.

Listing 5.1 shows how matrix-vector products are implemented in CUDA for the element-by-element strategy. Notice that a variable **n** is provided to identify which local node is being processed at the moment. This kernel is called four times (or eight, in 3D) at each matrix-vector product, to compute contributions of all local nodes.

```
// Check if this thread must work
     if (i<nElems){
       unsigned int row = (unsigned int) elemMatMap[i] *16 + n *4;
Ę
       unsigned int dof;
6
       unsigned int id = 0;
7
       var contrib = 0.0;
8
        // (bottom, left)
ç
       dof = WALK DOWN(i, nRows);
10
       contrib += K[row] * d[dof];
       id += (n==0)*dof;
12
       // (bottom, right)
13
        dof = WALK RIGHT(dof, nRows, nElems);
14
       contrib += K[row+1]*d[dof];
15
       id += (n==1)*dof;
16
       // (top, right)
17
       dof = WALK RIGHT(i, nRows, nElems);
18
       \operatorname{contrib} += K[\operatorname{row}+2] * d[\operatorname{dof}];
19
       id += (n==2)*dof;
20
        // (top, left)
21
        // dof = i;
22
       \operatorname{contrib} += K[\operatorname{row}+3]*d[i];
23
       id += (n==3)*i;
24
       q[id] += contrib;
25
     }
26
27 }
```

Listing 5.1: CUDA kernel for element-by-element matrix-vector products

In Listing 5.1, each thread is associated to one element (pixel or voxel) in the mesh, meaning the index i is equivalent to an element index. The operations WALK_DOWN and WALK_RIGHT are the ones stated in Table 3.1, implemented as macros. They satisfy the consideration of periodic boundary conditions. No **DOFMap** is required, due to the hard-coded consideration of the numbering scheme, as it can be seen in line 23, where the DOF of the (top,left) local node is identified by the element index, recalling Figure 3.6.

The node-by-node approach was also implemented in CUDA. Oppositely to the elementby-element solution, in this case, a single kernel call is able to compute all the necessary operations for an assembly-free matrix-vector product, without risk of race conditions. Listing 5.2 exposes a kernel for this task.

```
__global___void Aprod_thermal_2D_NodeByNode(var *d, unsigned int
     nDOFs, map nodeMatMap, unsigned int nCols, unsigned int nRows, var
      *q){
    // Get global thread index
2
    unsigned int i = threadIdx.x + blockIdx.x * blockDim.x;
3
    // Check if this thread must work
    if (i < nDOFs) {
5
      // Local var to store result
6
      var res;
7
      // Material map value for this node
      unsigned int map 16bit = (unsigned int) nodeMatMap[i];
ç
      // Local var to store neighbor dof indexes
10
      unsigned int dof;
11
      // Index where this material starts on K
12
      unsigned int mat = (map \ 16bit\%16)*16;
13
      // First neighbor elem
14
      res = K[mat] * d[i];
15
      dof = WALK RIGHT(i, nRows, nDOFs); // nDOFs == nElems
16
      res += K[mat+1]*d[dof];
17
      dof = WALK UP(dof, nRows);
18
      \operatorname{res} += K[\operatorname{mat}+2] * d[\operatorname{dof}];
19
      dof = WALK UP(i, nRows);
20
      res += K[mat+3]*d[dof];
21
      // Second neighbor elem
22
      map 16bit >>= 4;
23
      mat = (map \ 16bit\%16) * 16;
24
      dof = WALK LEFT(i, nCols, nRows, nDOFs);
25
      res += K[mat+4]*d[dof];
2\epsilon
      res += K[mat+5]*d[i];
27
      dof = WALK UP(i, nRows);
28
      res += K[mat+6]*d[dof];
29
      dof = WALK LEFT(dof, nCols, nRows, nDOFs);
30
      res += K[mat+7]*d[dof];
31
      // Third neighbor elem
32
      map 16 bit >>= 4;
33
      mat = (map \ 16bit\%16) * 16;
34
      dof = WALK DOWN(i, nRows); dof = WALK LEFT(dof, nCols, nRows, nDOFs);
35
      res += K[mat+8] *d[dof];
36
```

```
dof = WALK RIGHT(dof, nRows, nDOFs);
37
      res += K[mat+9] *d[dof];
38
       res += K[mat+10]*d[i];
39
      dof = WALK LEFT(i, nCols, nRows, nDOFs);
40
       res += K[mat+11]*d[dof];
41
      // Fourth neighbor elem
42
      map 16bit >>= 4;
43
      mat = (map \ 16bit\%16) * 16;
44
      dof = WALK DOWN(i, nRows);
45
      res += K[mat+12]*d[dof];
46
      dof = WALK_RIGHT(dof, nRows, nDOFs);
47
      res += K[mat+13]*d[dof];
48
      d of = WALK_RIGHT(i, nRows, nDOFs);
49
      res += K[mat+14]*d[dof];
50
      res += K[mat+15]*d[i];
51
      // Put final result in global array
52
      q[i] = res;
53
    }
54
55 }
```

Listing 5.2: CUDA kernel for node-by-node matrix-vector products

In Listing 5.2, each thread is associated to one node in the mesh, meaning the index i is equivalent to a node index. As with the element-by-element implementation, the operations WALK_UP, WALK_DOWN, WALK_RIGHT and WALK_LEFT are macros, detailed in Table 3.1, and no **DOFMap** is required. It is important to notice that materials are no longer mapped by elements, but by nodes. A 16-bit array **nodeMatMap** is employed to store neighbor material keys around each node, as illustrated by Figure 5.6.



Figure 5.6: 16-bit node material map

In Figure 5.6, the material keys are written in binary. The order of access to neighbor elements from a node is defined to match the local index of that node in the respective element. For instance, the 1st element is the one where the current node corresponds to its local node 0. In 3D, the rule is analogous, considering two layers of neighbor elements. As it has been stated previously, in Subsection 4.3.2, this mapping strategy decreases the number of different materials that can be considered, in comparison to the 8-bit element material map, but it provides coalesced data access to the image for the node-by-node approach. As it usual to work with few distinct materials at the micro-scale in homogenization analyses, such an approach meets the demands of this work, but other strategies can be explored in future developments. For example, storing the 8-bit image in texture memory caches could be an improvement.

5.2.2 Memory allocation

The MParPCG solver was implemented with space efficiency as one of its premises. The assembly-free strategy, and not storing commonly used PCG arrays (s, M) are efforts that push the limits of model sizes for FE analysis without resorting to high-performance hardware. In light of this, as another stride towards memory reduction, it was decided to test single precision floating point variables in the CUDA C program, as it was done by Müller et al. [56], so that the image dimension limit may essentially be doubled.

The choice to assess float variables was not made at random. As the numerical tolerance for dimensionless norms of residuals usually is set within 1e-04 and 1e-06, the results are considered satisfactory if they converge in 4 to 6 digits precision. Furthermore, the range of values for the coefficients of the global matrix and forcing vector generally is well within single precision range, due to the admitted physical properties at the microscale. Bearing this in mind, it was suggested that using double precision could be excessive for the problems being analyzed. In fact, the results presented in Chapter 7 confirm this notion. It is important to state that a similar version of the implementations using double variables is also maintained. This text focuses on the single precision solutions for being more efficient, both in time and memory, and effectively achieving the same results for all the conducted studies of numerical homogenization, but every development in GPU presented here also was also applied to solvers with 8 byte variables.

Another difference in the GPU implementation is that the local matrices can be stored in the read-only on-chip memory, for fast access to the CUDA threads. These matrices are not, by any means, significant to the total allocated memory in DRAM, but keeping them near the processors is interesting for performance improvement.

The significant arrays stored in global memory, of both host and device, are shown in

	Size of variable	Array dimension	host	device		
$\mathbf{r}, \mathbf{d}, \mathbf{q}$	4 Bytes	n	×	\checkmark		
x	4 Bytes	n	\checkmark	\checkmark		
RHS	4 Bytes	n	\checkmark	×		
DOFMap	4 Bytes	$n_{ m nodes} pprox n/n_{ m nodalDOFs}$	\checkmark	×		
material map	1 or 2 Bytes $n_{\text{elems}} = n/n_{\text{nodalDOFs}}$			\checkmark		
local matrices \mathbf{K}	4 Bytes $(\text{const.})n_{\text{materials}} << n$			×		
Total [Bytes]						
host	$2 \times 4n + 4n_{\text{nodes}} + (1 \text{ or } 2)n_{\text{elems}} + 4(\text{small constant})$					
device		$4 \times 4n + (1 \text{ or } 2)n_{\text{elems}}$				

Table 5.1. It is noticeable that space complexity remains linear, however the tendency of memory allocation, is reduced from the multi-thread CPU solution (Table 4.1).

Table 5.1: Memory allocated in host and device for CUDA implementation of the MParPCG solver

In order to solidify the space efficiency obtained with the proposed approach, a comparison of estimates of memory allocation in the device for assembly-free and assembled sparse matrix solutions is presented in Table 5.2. The Compressed Sparse Row (CSR) format is considered for the sparse matrices, as it was implemented for a PCG solver in GPU by Helfenstein and Koko [31]. The structured nature of the image-based meshes is taken into account. Analyses of linear elasticity in 3D are considered, for being the most memory consuming simulations in the scope of this work.

		Memory [GB]		
Dim. [voxels]	DOFs	Assembly free	CSR format	
	$\times 10^{6}$	Assembly-nee	(sparse matrix)	
50^{3}	0.375	0.006	0.25	
100^{3}	3.0	0.05	2.00	
200^{3}	24.0	0.40	16.03	
300^{3}	81.0	1.35	54.11	
400^{3}	192.0	3.20	128.3	
500^{3}	375.0	6.25	250.5	

Table 5.2: Estimates of memory allocation in the device for PCG solvers with the assembly-free approach and assembled matrices in CSR format, for 3D linear elasticity simulations.

5.2.3 Data transfer between host and device

As computationally powerful as GPUs are, programs that explore them depend on data transfers between the global memories of CPU and GPU. These memory transactions can be significantly time-consuming, to the point that, if they are repeatedly required and/or poorly timed, they can completely undermine performance improvements made with massively parallel computations. In the MParPCG solver, it is assumed that the PCG arrays \mathbf{r} , \mathbf{d} and \mathbf{q} reside solely in the device, while \mathbf{x} has copies in the host and the device. Even so, the host copy of \mathbf{x} can be updated just at the end of the PCG solver, so, in the scope of the while iterations of the numerical method, no arrays need to be transferred in either direction. At most, two float variables must be transported from device to host at each iteration, to return the results of dot products. Figure 5.7 depicts all of the memory allocation and transfer during a run of the MParPCG solver.



Figure 5.7: API calls to manage the data flow in the MParPCG solver. Memory allocation and transfer to device in green, call to iterative solver in blue, freeing memory in red.

5.3 Alternative strategies of implementation

The CUDA implementation for the PCG method presented in the previous Section performs much better than its akin CPU program, as it will be shown in Chapter 7, but, even with the reductions in memory allocation, it still can be improved in terms of space efficiency. In this Section, four alternative strategies of implementation are proposed, all with the objective of extending even further the size limits of the analysis.

5.3.1 xrd solver

At a first look in Algorithm 10, it might seem like the four arrays, \mathbf{x} , \mathbf{r} , \mathbf{d} and \mathbf{q} , are required to be stored in memory, and, in fact they are, for the original operations of the PCG method to remain unchanged. However, if some operations are mixed together, it is possible to vanish with the array \mathbf{q} , as detailed in Algorithm 11. Essentially, the idea is to substitute the operations that would attribute values to \mathbf{q} in every statement where the vector is required.

Algorithm	. xrd solver - MassPar PCG applied to assembly-free image-based FI	ΕM
1: Input: l	al matrices : \mathbf{k} , material map, \mathbf{b} , \mathbf{x}_0 , tolerance	

2: Output: x 3: $\mathbf{x} \leftarrow \mathbf{x}_0$ 4: $\mathbf{r} \leftarrow \mathbf{b} - (\sum \mathbf{T}_{e}^{T} \mathbf{k}_{e} \mathbf{T}_{e}) \mathbf{x}$ ▷ Aprod kernel 5: $\mathbf{d} \leftarrow (\sum \mathbf{T}_{e}^{T} \operatorname{diag}(\mathbf{k}_{e}) \mathbf{T}_{e})^{-1} \mathbf{r}$ ▷ applyPreConditioner kernel 6: $\delta \leftarrow \mathbf{r}^{\overline{T}}\mathbf{d}$ \triangleright dotprod kernel, sync. host and device 7: $\delta_0 \leftarrow \delta$ 8: while $(\delta/\delta_0 > \text{tolerance}^2)$ do \triangleright check in CPU $\alpha \leftarrow \delta / (\mathbf{d}^T (\sum \mathbf{T}_{\mathbf{e}}^T \mathbf{k}_{\mathbf{e}} \mathbf{T}_{\mathbf{e}}) \mathbf{d})$ \triangleright dotprod_Aprod kernel, sync. host and device 9: $\mathbf{x} \leftarrow \mathbf{x} + \alpha \mathbf{d}$ 10:⊳ sumVec kernel $\mathbf{r} \leftarrow \mathbf{r} - \alpha (\sum \mathbf{T}_{e}^{T} \mathbf{k}_{e} \mathbf{T}_{e}) \mathbf{d}$ ▷ Aprod kernel 11: $\delta_{\text{prev}} \leftarrow \delta$ 12: $\delta \leftarrow \mathbf{r}^T (\sum \mathbf{T}_{e}^T \operatorname{diag}(\mathbf{k}_{e}) \mathbf{T}_{e})^{-1} \mathbf{r} \quad \triangleright \operatorname{dotprod_precond kernel, sync. host and device}$ 13: $\beta \leftarrow \delta / \delta_{\text{prev}}$ 14: $\mathbf{d} \leftarrow (\sum \mathbf{T}_{e}^{T} \operatorname{diag}(\mathbf{k}_{e})\mathbf{T}_{e})^{-1}\mathbf{r} + \beta \mathbf{d}$ ▷ applyPreConditioner kernel 15:16: end while

It is evident that a performance price must be paid for \mathbf{q} not to be allocated, as each iteration of the PCG method, in this case, demands four sweeps of the domain, instead of two. Specific kernels, analogous to the matrix-vector product, needed to be implemented for the operations in lines 9, 11, 13 and 15 of Algorithm 11. The total allocation in global memory of the device is reduced to $3 \times 4n + (1 \text{ or } 2)n_{\text{elems}}$ Bytes, as now only three arrays of variable vectors are stored ($\mathbf{x}, \mathbf{r}, \mathbf{d}$).

5.3.2 xsd solver

In Algorithm 3, the conventional form of the PCG method, a vector \mathbf{s} is employed. It was removed in the developments of this work, as it was shown that it would allocate unnecessary memory, given that \mathbf{q} could be repurposed to store its values. Even so, \mathbf{s} can be used in alternative memory-efficient solutions, without \mathbf{q} . The statements in pseudo-code that invoke \mathbf{s} define its attributions as $\mathbf{s} \leftarrow \mathbf{M}^{-1}\mathbf{r}$. As the Jacobi preconditioner is adopted for \mathbf{M} , a diagonal matrix, it straightforwardly follows that the residual could be computed as $\mathbf{r} \leftarrow \mathbf{Ms}$. Considering this notion, the \mathbf{xrd} solver can be rewritten as in Algorithm 12.

Algorithm 12 xsd solver - MassPar PCG applied to assembly-free image-based FEM

1: Input: local matrices : \mathbf{k} , material map, \mathbf{b} , \mathbf{x}_0 , tolerance 2: Output: x 3: $\mathbf{x} \leftarrow \mathbf{x}_0$ 4: $\mathbf{s} \leftarrow (\sum \mathbf{T}_{e}^{T} \operatorname{diag}(\mathbf{k}_{e}) \mathbf{T}_{e})^{-1} (\mathbf{b} - (\sum \mathbf{T}_{e}^{T} \mathbf{k}_{e} \mathbf{T}_{e}) \mathbf{x})$ 5: $\mathbf{d} \leftarrow \mathbf{s}$ ▷ precond_Aprod kernel ▷ arrcpy kernel 5: $\mathbf{d} \leftarrow \mathbf{s}$ 6: $\delta \leftarrow \mathbf{d}^T (\sum \mathbf{T}_e^T \operatorname{diag}(\mathbf{k}_e) \mathbf{T}_e) \mathbf{s} > \operatorname{dotprod_invprecond} \operatorname{kernel}, \operatorname{sync.} \operatorname{host} \operatorname{and} \operatorname{device}$ 7: $\delta_0 \leftarrow \delta$ 8: while $(\delta/\delta_0 > \text{tolerance}^2)$ do \triangleright check in CPU $\alpha \leftarrow \delta / (\mathbf{d}^T (\sum \mathbf{T}_{e}^T \mathbf{k}_{e} \mathbf{T}_{e}) \mathbf{d})$ \triangleright dotprod_Aprod kernel, sync. host and device 9: $\mathbf{x} \leftarrow \mathbf{x} + \alpha \mathbf{d}$ ⊳ sumVec kernel 10: $\mathbf{s} \leftarrow \mathbf{s} - \alpha (\sum \mathbf{T}_{\mathrm{e}}^T \mathrm{diag}(\mathbf{k}_{\mathrm{e}}) \mathbf{T}_{\mathrm{e}})^{-1} (\sum \mathbf{T}_{\mathrm{e}}^T \mathbf{k}_{\mathrm{e}} \mathbf{T}_{\mathrm{e}}) \mathbf{d}$ 11:⊳ precond_Aprod kernel $\delta_{\text{prev}} \leftarrow \delta$ 12: $\delta \leftarrow \mathbf{s}^T (\sum \mathbf{T}_e^T \operatorname{diag}(\mathbf{k}_e) \mathbf{T}_e) \mathbf{s} \triangleright \operatorname{dotprod_invprecond} \operatorname{kernel}, \operatorname{sync.} \operatorname{host} \operatorname{and} \operatorname{device}$ 13: $\beta \leftarrow \delta / \delta_{\text{prev}}$ 14: $\mathbf{d} \leftarrow \mathbf{s} + \beta \mathbf{d}$ ⊳ sumVec kernel 15:16: end while

Memory allocation is unchanged from the **xrd** solver, but it is interesting to notice that the **xsd** solution combines two sweeps of the domain into one, in line 11, enabling the computations at each iteration to be made with three kernel calls for assembly-free matrix operations, instead of four.

5.3.3 rd solver

It is possible to go even further than the \mathbf{xrd} solver in memory reduction. At each iteration, the vector \mathbf{x} is updated, but it is not used anywhere else, so it does not need to be visible to the arrays \mathbf{r} and \mathbf{d} . In other words, \mathbf{x} is not required in the GPU, it can be updated in the CPU. To accomplish this, the search direction vector \mathbf{d} must be transferred from device to host at every iteration. In order to avoid excessive idleness of hardware, which slows down the solution, CUDA streams are employed in conjunction with asynchronous calls to transfer memory by parts, allowing the CPU to compute updated entries of \mathbf{x} while data is being received, and the GPU to keep executing the kernels for assembly-free matrix operations while sending data. This is depicted in Algorithm 13.

```
Algorithm 13 rd solver - MassPar PCG applied to assembly-free image-based FEM
```

```
1: Input: local matrices : \mathbf{k}, material map, \mathbf{b}, \mathbf{x}_0, tolerance
  2: Output: x
  3: \mathbf{x} \leftarrow \mathbf{x}_0
  4: Stream 0: {
              \begin{aligned} \mathbf{r} &\leftarrow \mathbf{b} - (\sum_{\mathrm{e}} \mathbf{T}_{\mathrm{e}}^{T} \mathbf{k}_{\mathrm{e}} \mathbf{T}_{\mathrm{e}}) \mathbf{x} \\ \mathbf{d} &\leftarrow (\sum_{\mathrm{e}} \mathbf{T}_{\mathrm{e}}^{T} \mathrm{diag}(\mathbf{k}_{\mathrm{e}}) \mathbf{T}_{\mathrm{e}})^{-1} \mathbf{r} \\ \delta &\leftarrow \mathbf{r}^{T} \mathbf{d} \end{aligned}
                                                                                                                                                          ▷ Aprod kernel
  5:
                                                                                                                       ▷ applyPreConditioner kernel
  6:
                                                                                                     \triangleright dotprod kernel, sync. host and device
  7:
  8: }
  9: for stream in (Streams > 0) do
               Asynchronous memcpy from device to host of portion of d
10:
11: end for
12: \delta_0 \leftarrow \delta
       while (\delta/\delta_0 > \text{tolerance}^2) do
13:
                                                                                                                                                        \triangleright check in CPU
               \delta_{\text{prev}} \leftarrow \delta
14:
               Stream 0: {
15:
                     \begin{array}{l} \alpha \leftarrow \delta/(\mathbf{d}^T(\sum_{\mathbf{T}}\mathbf{T}_{\mathbf{e}}^T\mathbf{k}_{\mathbf{e}}\mathbf{T}_{\mathbf{e}})\mathbf{d}) \\ \mathbf{r} \leftarrow \mathbf{r} - \alpha(\sum_{\mathbf{T}}\mathbf{T}_{\mathbf{e}}^T\mathbf{k}_{\mathbf{e}}\mathbf{T}_{\mathbf{e}})\mathbf{d} \end{array}
                                                                                     \triangleright dotprod_Aprod kernel, sync. host and device
16:
                                                                                                                                                          ▷ Aprod kernel
17:
                     \delta \leftarrow \mathbf{r}^T (\sum \mathbf{T}_{e}^T \operatorname{diag}(\mathbf{k}_{e}) \mathbf{T}_{e})^{-1} \mathbf{r}
                                                                                                            \triangleright dotprod_precond kernel, no sync.
18:
19:
               }
              for stream in (Streams> 0) do
20:
                      Wait for memcpy from device to host of portion of d
21:
                                                                                                                                                                     ⊳ in CPU
                     \mathbf{x} \leftarrow \mathbf{x} + \alpha \mathbf{d} (portion of \mathbf{d} sent by stream)
22:
               end for
23:
               \beta \leftarrow \delta / \delta_{\text{prev}}
                                                         \triangleright transfer result of dotprod_precond, sync. host and device
24:
               Stream 0: {
25:
                     \mathbf{d} \leftarrow (\sum \mathbf{T}_{\mathrm{e}}^T \mathrm{diag}(\mathbf{k}_{\mathrm{e}}) \mathbf{T}_{\mathrm{e}})^{-1} \mathbf{r} + \beta \mathbf{d}
                                                                                                                       ▷ applyPreConditioner kernel
26:
               }
27:
28:
              for stream in (Streams> 0) do
29:
                      Asynchronous memcpy from device to host of portion of d
30:
               end for
31: end while
```

Clearly, the solution presented in Algorithm 13 is not expected to outperform the MParPCG solver or the xrd solver, but it reduces memory allocation in the device even more. With the rd solver, simulations can be carried out storing just $2 \times 4n + (1 \text{ or } 2)n_{\text{elems}}$ Bytes, as only two arrays of variable vectors are allocated in the GPU (r, d).

5.3.4 sd solver

In analogous fashion to the conception of the rd solver from xrd, the xsd solution can be adapted to sd, having the array x reside solely in the host. This approach is detailed in Algorithm 14.

Algorithm 14 sd solver - MassPar PCG applied to assembly-free image-based FEM

```
1: Input: local matrices : k, material map, b, x<sub>0</sub>, tolerance
  2: Output: x
  3: \mathbf{x} \leftarrow \mathbf{x}_0
  4: Stream 0: {
             \mathbf{s} \leftarrow (\sum \mathbf{T}_{e}^{T} diag(\mathbf{k}_{e}) \mathbf{T}_{e})^{-1} (\mathbf{b} - (\sum \mathbf{T}_{e}^{T} \mathbf{k}_{e} \mathbf{T}_{e}) \mathbf{x})
                                                                                                               ⊳ precond_Aprod kernel
  5:
  6:
             \mathbf{d} \leftarrow \mathbf{s}
                                                                                                                                            ▷ arrcpy kernel
             \delta \leftarrow \mathbf{d}^T (\sum \mathbf{T}_{e}^T \operatorname{diag}(\mathbf{k}_{e}) \mathbf{T}_{e}) \mathbf{s} \triangleright \operatorname{dotprod\_invprecond kernel, sync. host and device}
  7:
  8: }
 9: for stream in (Streams > 0) do
             Asynchronous memcpy from device to host of portion of d
10:
11: end for
12: \delta_0 \leftarrow \delta
13: while (\delta/\delta_0 > \text{tolerance}^2) do
                                                                                                                                             \triangleright check in CPU
14:
             \delta_{\text{prev}} \leftarrow \delta
             Stream 0: {
15:
                                                                          \triangleright dotprod_Aprod kernel, sync. host and device
                    \alpha \leftarrow \delta / (\mathbf{d}^T (\sum \mathbf{T}_{e}^T \mathbf{k}_{e} \mathbf{T}_{e}) \mathbf{d})
16:
                    \begin{split} \mathbf{s} &\leftarrow \mathbf{s} - \alpha (\sum_{\mathbf{r}} \mathbf{T}_{\mathbf{e}}^T \mathrm{diag}(\mathbf{k}_{\mathbf{e}}) \mathbf{T}_{\mathbf{e}})^{-1} (\sum_{\mathbf{r}} \mathbf{T}_{\mathbf{e}}^T \mathbf{k}_{\mathbf{e}} \mathbf{T}_{\mathbf{e}}) \mathbf{d} \\ \delta &\leftarrow \mathbf{s}^T (\sum_{\mathbf{r}} \mathbf{T}_{\mathbf{e}}^T \mathrm{diag}(\mathbf{k}_{\mathbf{e}}) \mathbf{T}_{\mathbf{e}}) \mathbf{s} \qquad \triangleright \, \mathtt{dotpr} \end{split} 
                                                                                                                           ▷ precond_Aprod kernel
17:
                                                                                             ▷ dotprod_invprecond kernel, no sync.
18:
             }
19:
             for stream in (Streams> 0) do
20:
                    Wait for memcpy from device to host of portion of d
21:
                                                   (portion of d sent by stream)
                                                                                                                                                         ⊳ in CPU
22:
                    \mathbf{x} \leftarrow \mathbf{x} + \alpha \mathbf{d}
             end for
23:
             \beta \leftarrow \delta / \delta_{\text{prev}}
                                              ▷ transfer result of dotprod_invprecond, sync. host and device
24:
25:
             Stream 0: {
                    \mathbf{d} \leftarrow \mathbf{s} + \beta \mathbf{d}
                                                                                                                                             ▷ sumVec kernel
26:
27:
             }
28:
             for stream in (Streams> 0) do
29:
                    Asynchronous memcpy from device to host of portion of d
             end for
30:
31: end while
```

5.4 Summary of the implementations

Five solvers where implemented for the PCG method applied to assembly-free imagebased FEM problems in GPU. They vary in computational cost, memory allocation, and

Coluon	Arrays of vars.	Matrix ops.	Data transferred	Multiple
Solver	in device	per iteration	per iteration	streams
MParPCG	4	2	8 Bytes	×
xrd	3	4	8 Bytes	×
xsd	3	3	8 Bytes	×
rd	2	4	(8+4n) Bytes	\checkmark
sd	2	3	(8+4n) Bytes	\checkmark

Table 5.3: Summary of the PCG solvers implemented in GPU

amount of data transferred per iteration. Table 5.3 synthesizes the main characteristics of the proposed solutions.

From Table 5.3, it can be expected that runs with best performance are obtained with the MParPCG solver, as it allocates the result of the matrix-vector product in each iteration, meaning that less sweeps of the domain are required. However, this is the solver with the lowest image dimension limit, exactly for storing an extra array in memory. The xrd and xsd solvers seem to be the most efficient ones, as they reduce allocation from the MParPCG solver, but still demand small data transfers per iteration (two scalars, dot-product results). The xsd solution requires one less matrix operation per iteration, but in exchange it demands a relatively strenuous kernel for updating the s array, so the computational power of each CUDA core is a determinant factor for performance comparisons against the xrd. The rd and sd solvers should be slower options, as they require a whole variable vector to be transferred from device to host at each iteration. Their main purpose is to push the size limits of the analyses, aiming to maximize memory efficiency.

Chapter 6

Finding initial guesses in coarse meshes

In this Chapter, a novel methodology is proposed, to the best of the author's knowledge, to obtain good initial guesses \mathbf{x}_0 for the PCG method applied to image-based FE models, employing recursive solutions in coarsened meshes. This is an effort to accelerate the convergence of the method itself, in number of iterations, so it applies not only to assembly-free approaches or the massively parallel implementation scheme.

6.1 Hypothesis

In general case applications of the PCG to solve FEM problems, it is usual to predetermine the initial guess vector \mathbf{x}_0 as null, so \mathbf{x} is initialized with zeros in all of its entries (in line 4 of Algorithm 9). This choice can be associated with the fact that a null vector solution commonly represents a physical system in equilibrium, without any forcing applied to it. If relatively small forcing is considered, especially when working with the assumption of linear behavior, it is somewhat logical to expect that the response will be close to the null vector. However, in the majority of cases, it is obvious that the null solution is not the desired response of a physical simulation, thus, in this sense, setting \mathbf{x}_0 as $\vec{0}$ is not really adopting an initial guess, per se, but rather a non-random default starting point. This does not mean that it performs specifically bad, but it is possible to consider actual initial guesses, closer to the solution from the start, so that less iterations are needed for convergence to be obtained.

It is natural, when dealing with the FEM, to think about mesh refinement to observe convergence of the results. The idea is that, with further discretization of the domain, the numerical responses should stabilize near a good approximation of what would be an analytical solution of the governing equations. This means that solutions from coarse meshes can be thought of as approximations of the solutions from refined meshes. In the scope of pixel and voxel-based models, coarsening the mesh is equivalent to decreasing resolution of the image. It then follows that an expected good initial guess for a PCG run on an image-based FEM problem would be the solution of a similar version of that problem, with an image of lower resolution.

It is vital to be mindful that each iteration of the PCG method applied to image-based FEM has time complexity $\mathbf{O}(n)$, where *n* is the dimension of the system to be solved, that is, the number of DOFs. Considering coarsening operations that halve the resolution of the image on each of its axis, a coarsened model has 1/4 of the pixels (2D) or 1/8 of the voxels (3D), which are directly proportional to the quantity of DOFs. This means that each PCG iteration on an image with half of the original resolution should be about $4 \times$ faster in 2D, and $8 \times$ faster in 3D. Furthermore, it is observed that models of lower resolution usually demand less iterations to converge. Hence, it is clear that solving a smaller system on a coarse mesh to find an initial guess is a task that takes considerably less time than the solution of the original system itself, and, if that approximation helps in reducing the number of iterations for the convergence, it can lead to significant overall runtime reduction for a final response to be obtained.

6.2 Coarsening of image-based meshes

In this work, a straightforward strategy is employed to lower the resolution of images. The coarsening procedure, described in Algorithm 15, merges clusters of elements, 2×2 pixels in 2D, and $2 \times 2 \times 2$ voxels in 3D, into a single one, adopting the color value of the (left,top,near) element within the group. Figure 6.1 depicts this process for a 2D image. These operations can be performed in GPU.



Figure 6.1: (a) 200×200 pixel-based model of a sandstone sample [76], (b) coarsened 100×100 pixel mesh.

Assuredly, a broader discussion could be held in regards to qualitative aspects of the coarsening, considering more sophisticated image processing techniques. For the sake of conciseness, and to avoid digressing from the focus of this work, possible improvement on this matter is left as future work. Even so, it is important to recall that the main goal is to reduce the runtime of the analyses, so operations of this sort should not be computationally expensive. It might be advantageous, in fact, to stick to rather simple techniques, so that the initial guesses can be quickly obtained, even if those are not the best ones possible, but good nonetheless.

6.3 A recursive algorithm for the search of initial guesses

Once the solution with a coarse mesh is found, the results are linearly interpolated back to the original mesh, as a starting point to the PCG method. Considering this, it is noticeable that the solution for the image of lowered resolution itself can also adopt an initial guess from a coarsened version of it. This process can be recursively employed until reaching a mesh where the system can be solved with a null vector as a starting point in negligible time. The PCG can be rewritten as to take this notion into account, as described in Algorithm 16. Even though this pseudo-code was written based on Algorithm 10, it is important to state that the approach applies for any form of the PCG method.

Attention is drawn to a subtle, yet important, change made to the original form

of the method, first presented in Algorithm 3. In Algorithm 16, the parameter δ_0 is no longer associated with a provided \mathbf{x}_0 , now it is always computed as if the starting point is a null vector. This is done to relate the dimensionless stopping criteria with the adoption of $\mathbf{x}_0 = \vec{0}$ at the lowest resolution considered for the recursive searches. Mathematically, the criteria has changed from $||\mathbf{b} - \mathbf{A}\mathbf{x}||/||\mathbf{b} - \mathbf{A}\mathbf{x}_0|| < tolerance$ to $||\mathbf{b} - \mathbf{A}\mathbf{x}||/||\mathbf{b}|| < tolerance$.

Algorithm 16 MassPar PCG with recursive search for init	ial guess
1: Input: local matrices : k, material map, b, tolerance, red	cursion
2: Output: x	
3: if recursion > 0 then	
4: $\mathbf{x} \leftarrow \operatorname{interpl}(\operatorname{PCG}(\mathbf{k}, \operatorname{coarsen}(\operatorname{material}\operatorname{map}), \operatorname{coarsen}(\operatorname{material}\operatorname{map}))$	b), tolerance, recursion -1))
5: $\mathbf{r} \leftarrow \mathbf{b} - (\sum_{\mathbf{m}} \mathbf{T}_{e}^{T} \mathbf{k}_{e} \mathbf{T}_{e}) \mathbf{x}$	
6: $\mathbf{d} \leftarrow (\sum_{\mathbf{r}} \mathbf{T}_{e}^{T} \operatorname{diag}(\mathbf{k}_{e}) \mathbf{T}_{e})^{-1} \mathbf{r}$	
7: $\delta \leftarrow \mathbf{r}^T \mathbf{d}$	
8: $\delta_0 \leftarrow \mathbf{b}^T (\sum \mathbf{T}_{e}^T \operatorname{diag}(\mathbf{k}_{e}) \mathbf{T}_{e})^{-1} \mathbf{b}$	
9: else	
10: $\mathbf{x} \leftarrow 0$	
11: $\mathbf{r} \leftarrow \mathbf{b}$	
12: $\mathbf{d} \leftarrow (\sum_{\mathbf{r}} \mathbf{T}_{e}^{T} \operatorname{diag}(\mathbf{k}_{e}) \mathbf{T}_{e})^{-1} \mathbf{r}$	
13: $\delta \leftarrow \mathbf{r}^T \mathbf{d}$	
14: $\delta_0 \leftarrow \delta$	
15: end if	
16: while $(\delta/\delta_0 > \text{tolerance}^2)$ do	
17: $\mathbf{q} \leftarrow (\sum_{e} \mathbf{T}_{e}^{T} \mathbf{k}_{e} \mathbf{T}_{e}) \mathbf{d}$	
18: $\alpha \leftarrow \delta/\mathbf{q}^{I} \mathbf{d}$	
19: $\mathbf{x} \leftarrow \mathbf{x} + \alpha \mathbf{d}$	
20: $\mathbf{r} \leftarrow \mathbf{r} - \alpha \mathbf{q}$	
21: $\mathbf{q} \leftarrow (\sum \mathbf{T}_{e}^{T} \operatorname{diag}(\mathbf{k}_{e}) \mathbf{T}_{e})^{-1} \mathbf{r}$	
22: $\delta_{\text{prev}} \leftarrow \delta$	
$23: \delta \leftarrow \mathbf{r}^{T} \mathbf{q}$	
24: $\beta \leftarrow \delta / \delta_{\text{prev}}$	
25: $\mathbf{d} \leftarrow \mathbf{q} + \beta \mathbf{d}$	
26: end while	

6.4 A validation test

An experiment was conducted as proof of concept that the proposed strategy in fact works. Thermal conductivity simulations on direction x_1 were run for a synthetic 2D model of 400 × 400 pixels, illustrated in Figure 6.2. Physical properties were admitted to be isotropic, setting $\kappa^{\text{white}} = 10 \text{ W/m/K}$ and $\kappa^{\text{black}} = 1 \text{ W/m/K}$. A simple MATLAB program was designed to assemble the FEM global system of equations in sparse form

Resolution [pixels]	\mathbf{x}_0	$ \mathbf{b} - \mathbf{A}\mathbf{x} / \mathbf{b} $	Iterations	Time [s]
100×100	$\vec{0}$	8.1033e - 05	70	0.0232
200×200	$\vec{0}$	9.4370e - 05	138	0.1630
400×400	$\vec{0}$	$9.9521 \mathrm{e}{-05}$	275	1.4818
200×200	x from 100×100	9.4490e - 05	59	0.1083
400×400	x from 200×200	$9.3853e{-}05$	93	0.6116

Table 6.1: Convergence metrics for preliminary tests with initial guesses from coarse meshes

and call the pcg built-in function to solve it, providing or not an initial guess from a coarse mesh. A numerical tolerance of 1e-04 was admitted for the dimensionless norms of residuals. It should be noted that this function considers the same stopping criteria as specified in Algorithm 16, and it accepts the Jacobi preconditioner. Table 6.1 details convergence metrics for the PCG solution, considering different resolutions (coarsened versions of the 400×400 mesh).



Figure 6.2: 400x400 synthetic model for tests with initial guesses from coarse meshes

The data shown in Table 6.1 clearly denote that the adoption of initial guesses obtained from lower resolution images accelerates the PCG method, in comparison to employing the default null vector as a starting point. The solution of the system associated with the 400×400 pixels model takes roughly half the time with the calculated initial guess. It is interesting to notice, for that model, that the total number of iterations required for the solution with two recursive searches, that is 70 + 59 + 93 = 222, is not far from the 275 iterations demanded for convergence starting at $\vec{0}$, however, more than half of the iterations in the first case take place in significantly smaller meshes, so they are consistently faster.

To reinforce the notions obtained from Table 6.1, Figure 6.3 is presented. In that Figure, the dimensionless residual is plotted against the number of iterations of the PCG

method. This graphic representation of the metrics sheds light on how a good initial guess makes the solution more efficient, as it avoids possible localized instabilities in the minimization process, leading the solution on a smooth path to convergence.



Figure 6.3: Dimensionless norms of residuals vs. number of iterations

This preliminary test provided satisfactory results for the hypothesis that searching for initial guesses on lower resolution models accelerates convergence of the PCG method. In Chapter 7, results and metrics for considerably larger models studied with the proposed strategy are presented. It is shown that acceleration and significant performance gains are also obtained for the analysis of image-based models with more than 10⁸ DOFs, synthetic and of physical samples, suggesting that this is indeed a fruitful endeavor.

Chapter 7

Results

This Chapter is dedicated to presenting and discussing results and metrics obtained with the implemented solvers. Performance and memory allocation for homogenization simulations in CPU and in GPU are assessed. At first, a validation of the programs is presented by checking the obtained numerical results against a benchmark, in Section 7.1. Then, in Section 7.2, a 3D microtomographic model of a cast iron sample is studied, evaluating its thermal conductivity and elasticity constitutive tensors. Several possibilities for the solution of the large-scale system of equations with the PCG method are considered, in an effort to compare every strategy approached in this work. Performance is presented for massively parallel analyses with the element-by-element and node-by-node schemes, then for each one of the five solvers implemented in CUDA C, employing the recursive search for initial guesses discussed in Chapter 6. Finally, in Section 7.3, each GPU solver is run on the limit of memory of two different devices with simulations of synthetic models, to showcase the potential of the developed programs to deal with extremely large problems for a single computer in reasonable time.

The analyses presented in the following Sections employed a desktop computer located at the *Laboratório de Computação Científica* (Scientific Computing Laboratory), within the Institute of Computing, at Fluminense Federal University. The specifications of this machine are presented in Table 7.1. The only exception is the study presented in Subsection 7.3.2, where simulations were carried out in a laptop, with properties that will be specified at that point in the text.

Computer Specifications					
O.S.	Ubuntu 20.04.2 LTS				
CPU	AMD Ryzen 7 3700x				
Clock rate	$3.60~\mathrm{GHz}$				
DRAM	$64 \mathrm{GB}$				
Cores (threads)	8(16)				
GPU	Nvidia GeForce RTX 2080 Super				
Clock rate	$1.85~\mathrm{GHz}$				
DRAM	8 GB				
CUDA cores	3072				
Architecture	Turing				

Table 7.1: Specifications of the computer employed in this work

7.1 Analytical benchmark

The synthetic model illustrated in Figure 7.1 has a known analytical solution for thermal conductivity problems, described by Perrins et al. [62], so it was used as a validation benchmark for the developed programs. Both phases are isotropic (Appendix A), the one that configures the circles, depicted in gray, has thermal conductivity $\kappa^{\text{gray}} = 10 \text{ W/m/K}$, whilst the predominant material, represented in black, has thermal conductivity $\kappa^{\text{black}} = 1$ W/m/K. The circular regions have radii equivalent to L/8, where L corresponds to the horizontal and vertical dimensions of the image. It was admitted a tolerance of 1e-06 for dimensionless relative norms of residuals obtained with the PCG method. Initial guesses for the PCG method were all admitted as $\mathbf{x}_0 = \vec{0}$. Simulations were run with the elementby-element and node-by-node approaches to the MParPCG GPU solver, as well as with the element-by-element solution in CPU. Table 7.2 details the results, which were considered satisfactory, considering that the maximum error obtained, with a resolution of 50×50 pixels, was within a 0.2% margin of the analytical solution. It is important to remark that the increasing resolutions do not imply a mesh refinement being applied to the same original model. Each step is related to a different image that represents the domain shown in Figure 7.1 with a regular grid of pixels based on a given resolution. This means that the geometry of the model is not necessarily preserved in its entirety as resolution changes, so it is expected that a smooth convergence tendency might not occur.

From Table 7.2, it is important to see that both solutions in GPU got the same results as the CPU solver, since the latter uses double precision variables, while the two first employ single precision. A first comparison of performance was made with the metrics for this test, shown in Table 7.3. These models are significantly small, considering the context of large-scale FEM analysis, but some initial insights can be obtained from them.



Figure 7.1: Analytical benchmark

	$\kappa_{11},\kappa_{22}~[{ m W/m/K}]$			
Resolution [pixels]	CPU	GPU EbE	GPU NbN	
50x50	1.1769	1.1769	1.1769	
100 x 100	1.1755	1.1755	1.1755	
$250 \mathrm{x} 250$	1.1765	1.1765	1.1765	
$500 \mathrm{x} 500$	1.1752	1.1752	1.1752	
1000 x 1000	1.1751	1.1751	1.1751	
Perrins et al. [62]		1.1747	•	

Table 7.2: Results for analytical benchmark

In Table 7.3, and all following metrics tables, the columns headed as **PCG time** refer to the solution of each system of equations with the PCG method, while **Total time** is associated with the whole homogenization study, which demands solutions of multiple systems (2 for 2D thermal conductivity, 3 for 3D thermal conductivity and 2D elasticity, and 6 for 3D elasticity). It is noticeable that elapsed time grows rapidly with increasing resolution, as the image-based FE analysis with the PCG is $O(n^2)$. The 50x50 and 100x100 models are too small for any meaningful conclusions to be drawn, but, from the 250x250 and larger models, it is clear that the GPU can significantly accelerate the solutions. The 1000x1000 model (1 million DOFs) has its systems solved 35x faster with the node-by-node strategy in the GPU, in comparison to the solver in CPU. The total

		CPU		GPU		GPU		
			010		$_{\rm EbE}$		NbN	
Dec [pivels]	DOF	PCG its.	PCG time [s]	Total	PCG time [s]	Total	PCG time [s]	Total
Res. [pixels] DOF	DOI'S	(x_1, x_2)	(x_1, x_2)	time [s]	(x_1, x_2)	time [s]	(x_1, x_2)	time [s]
50^{2}	2500	100	0.006	0.027	0.005	0.083	0.004	0.081
100^{2}	10000	193	0.027	0.057	0.010	0.087	0.009	0.085
250^{2}	62500	420	0.205	0.442	0.029	0.132	0.024	0.124
500^{2}	250000	773	1.510	3.051	0.087	0.274	0.069	0.256
1000^{2}	1000000	1096	9.030	18.402	0.385	0.958	0.253	0.707

Table 7.3: Time metrics for analytical benchmark

time for homogenization shows speed up of 26x, in that same case. It is also notorious that the node-by-node approach outperforms the element-by-element, in GPU. These notions are strengthened by Figure 7.2, where the mean time per PCG iteration is plotted against the number of DOFs, for the analyses detailed in Table 7.3.



Figure 7.2: Mean time per PCG iteration [s] vs number of DOFs, for the analytical benchmark

It is interesting to notice, in Figure 7.2, that, in fact, the assembly-free strategy leads to O(n) time complexity per iteration of the PCG method.

7.2 Cast iron sample

In this Section, results and metrics for a 3D image-based model of a cast iron sample, obtained with μ CT, are presented to demonstrate the capabilities of the proposed implementations. The used data was previously studied by Pereira et al. [61], and is available at [60]. The micro-scale domain consists of two phases, a predominant ferritic matrix and several minor nodules of graphite. A visual representation of the sample, with a 200x200x200 voxels, is presented in Figure 7.3.



Figure 7.3: 200x200x200 voxel-based representation of the cast iron sample, consisting of graphite nodules within a ferritic matrix

	$\kappa \; [W/m/K]$	E [GPa]	ν
Graphite	129.0	39.7	0.2225
Matrix	80.4	210.0	0.3000

Table 7.4: Physical properties of the micro-scale phases in the cast iron sample

Simulations for the homogenization of thermal conductivity and elasticity of the model were performed. Each phase was admitted to be isotropic, having the physical properties shown in Table 7.4.

7.2.1 Homogenized physical properties

At first, the focus is on qualitative aspects of the numerical homogenization analysis. In Section 7.1, it was shown that the implemented 2D thermal conductivity solver was able to match a known analytical solution, but it is imperative to also validate the 3D solutions for thermal conductivity and elasticity. In that sense, the cast iron sample presented in Figure 7.3 was analyzed with the MParPCG solver in GPU, considering element-byelement, node-by-node, and DOF-by-DOF (fine-grained version of the node-by-node for elasticity analysis) strategies. The CPU solver was also used for the heat conduction

		CPU		Т	GPU	T	
					EDE, NDN		
Dim.	κ_{11}	κ_{22}	κ_{33}	κ_{11}	κ_{22}	κ_{33}	
[voxels]		[W/m/K]]		[W/m/K]]	
50^{3}	83.117	82.820	83.065	83.117	82.820	83.065	
100^{3}	85.249	85.166	85.032	85.249	85.166	85.033	
200^{3}	84.956	84.934	84.947	84.956	84.934	84.947	
300^{3}	84.799	84.809	84.791	84.798	84.808	84.791	
400 ³	84.658	84.655	84.640	84.658	84.655	84.641	

Table 7.5: Results for orthotropic thermal conductivity of the cast iron sample

problem, to compare obtained values. For elasticity, the results were checked against the original reference [61]. No recursive searches for initial guesses were employed yet, so the PCG method starts at $\mathbf{x}_0 = \vec{0}$ in all cases. A numerical tolerance of 1e-05 was adopted for dimensionless relative norms of residuals. Table 7.5 presents the obtained homogenized orthotropic conductivity coefficients (Appendix B), Table 7.6 shows the orthotropic Young's moduli, and Table 7.7 depicts the constitutive tensors for elasticity that were achieved.

	Per	eira et al.	[61]	EbE, I	GPU NbN, DOF	bDOF
Dim. [voxels]	E_1	E_2	E_3	E_1	E_2	E_3
		[GPa]			[GPa]	
100^{3}	180.757	179.338	177.976	180.748	179.420	177.933
200^{3}	180.376	180.129	179.700	180.425	179.738	180.032
300^{3}	181.090	180.876	180.790	180.966	181.045	180.776
400^{3}	182.043	181.838	181.752	182.017	182.005	181.632

Table 7.6: Comparison of the obtained orthotropic Young's moduli with the results of Pereira et al. [61]

The obtained results were considered satisfactory, as they approximated the expected values of such properties for nodular graphite cast iron with a ferritic matrix. For similar samples, Wu et al. [77] found isotropic thermal conductivity of 83.28 W/m/K, and Liu et al. [41] obtained 82.43 W/m/K, so the largest difference achieved with the developed solvers from these references is 3.3%, with an image dimension of $100 \times 100 \times 100 \times 100$ voxels, considering an average of the presented orthotropic results. The obtained orthotropic elasticity properties matched the findings of Pereira et al. [61], as it can be seen in Table 7.6. With these results and the ones shown in Section 7.1, the developed GPU programs were validated, as it was demonstrated that they are able to reproduce analytical and experimental findings.

	GPU EbE. NbN. DOFbDOF					
Dim. [voxels]		C [GPa]				
100 ³	$ \begin{pmatrix} 234.69\\ 94.19\\ 92.98\\ -0.41\\ 0.15\\ -0.48 \end{pmatrix} $	$94.20 \\ 233.16 \\ 92.88 \\ -0.78 \\ 0.23 \\ -0.22$	$\begin{array}{r} 92.98\\ 92.88\\ 230.57\\ -0.49\\ 0.17\\ -0.26\end{array}$	$-0.41 \\ -0.78 \\ -0.49 \\ 68.22 \\ -0.21 \\ 0.21$	$\begin{array}{c} 0.15 \\ 0.23 \\ 0.17 \\ -0.21 \\ 68.24 \\ -0.30 \end{array}$	$\begin{array}{c} -0.48 \\ -0.22 \\ -0.26 \\ 0.21 \\ -0.30 \\ 69.34 \end{array}$
200 ³	$ \begin{array}{c} (235.58)\\ 95.38\\ 95.48\\ 0.10\\ 0.02\\ 0.07 \end{array} $	95.38 234.69 95.30 0.21 0.01 0.11	$\begin{array}{r} 95.48\\ 95.30\\ 235.09\\ 0.07\\ -0.05\\ 0.06\end{array}$	0.10 0.21 0.07 69.56 0.05 0.01	$\begin{array}{c} 0.02\\ 0.01\\ -0.05\\ 0.05\\ 69.65\\ 0.07\end{array}$	$\begin{array}{c} 0.07\\ 0.11\\ 0.06\\ 0.01\\ 0.07\\ 69.60 \end{array}$
300^{3}	$ \begin{array}{r} 236.57\\ 96.22\\ 96.11\\ 0.06\\ -0.08\\ -0.12 \end{array} $	$\begin{array}{c} 96.22 \\ 236.69 \\ 96.15 \\ 0.05 \\ -0.02 \\ -0.09 \end{array}$	$\begin{array}{c} 96.11 \\ 96.15 \\ 236.30 \\ 0.01 \\ -0.05 \\ 0.01 \end{array}$	$\begin{array}{c} 0.06 \\ 0.05 \\ 0.01 \\ 69.98 \\ 0.03 \\ -0.01 \end{array}$	$\begin{array}{r} -0.08 \\ -0.02 \\ -0.05 \\ 0.03 \\ 69.93 \\ 0.05 \end{array}$	$\begin{array}{c} -0.12 \\ -0.09 \\ 0.01 \\ -0.01 \\ 0.05 \\ 70.02 \end{array}$
400^{3}	$\begin{pmatrix} 238.27\\ 97.12\\ 97.01\\ 0.01\\ 0.01\\ -0.10 \end{pmatrix}$	$97.12 \\238.23 \\96.97 \\-0.05 \\-0.00 \\-0.05$	97.01 96.97 237.73 $-0.05 -0.05 -0.01$	$\begin{array}{r} 0.01 \\ -0.05 \\ -0.05 \\ 70.31 \\ -0.00 \\ 0.00 \end{array}$	$ \begin{array}{r} 0.01 \\ -0.00 \\ -0.05 \\ -0.00 \\ 70.34 \\ 0.01 \end{array} $	$\begin{array}{c} -0.10\\ -0.05\\ -0.01\\ 0.00\\ 0.01\\ 70.41 \end{array}$

Table 7.7: Results for elasticity constitutive tensors of the cast iron sample

7.2.2 Metrics

After the programs were validated, the focus now shifts to our main goal: performance. In this Subsection, several possibilities for running the analyses presented in the previous Subsection are explored. Firstly, the goal is to determine the most time-efficient solution in GPU for 3D models, node-by-node or element-by-element. Then, the MParPCG, xrd, xsd, rd, and sd solvers are compared, considering the recursive search for initial guesses for elasticity simulations. The idea is to relate elapsed time to memory allocation.

7.2.2.1 Element-by-element vs. Node-by-node

In Table 7.8, time metrics are presented for the thermal conductivity homogenization studies, whose results were exposed in Table 7.5.

			CPU		GPU E	bΕ	GPU NbN	
Dim.	DOFs	PCG its.	PCG time [s]	Total	PCG time [s]	Total	PCG time [s]	Total
[voxels]	$\times 10^{6}$	(x_1)	(x_1)	time [s]	(x_1)	time [s]	(x_1)	time [s]
50^{3}	0.125	68	0.23	0.74	0.007	0.69	0.004	0.12
100^{3}	1.0	134	3.50	11.1	0.08	0.74	0.04	0.52
200^{3}	8.0	240	56.4	175	0.99	5.82	0.46	4.24
300^{3}	27.0	301	239	738	4.22	22.6	1.88	15.3
400^{3}	64.0	349	661	2010	11.4	58.6	5.16	38.6

Table 7.8: Time metrics for thermal conductivity analysis of the cast iron sample

From Table 7.8, it is clear that the GPU solvers perform significantly better than the program running in CPU. The Node-by-node strategy in GPU is once again demonstrated to be the faster option, achieving up to 128x speed up from the CPU solver in the solution of linear systems with the PCG method, for a 400³ voxels image. In that case, the speed up for the whole process of homogenization is of 52x. The element-by-element solution in GPU also performs considerably better than the CPU solver, but it is slower than the node-by-node, as portrayed in Figure 7.4, where the time per PCG iteration for each approach is plotted against the number of DOFs. A graphical comparison of elapsed times for the whole homogenization process is depcited in Figure 7.8.



Figure 7.4: Mean time per PCG iteration [s] vs number of DOFs, for thermal conductivity analysis of the cast iron sample

An analogous assessment of the performance for 3D elasticity analyses was made. Table 7.9 presents time metrics for these studies, also considering a DOF-by-DOF approach. As it can be seen from this data, the performance gains for running elasticity simulations in the GPU are even greater than what could be observed for thermal conductivity. The node-by-node solution is able to speed up the PCG method by an impressive factor of 400x, comparing to the CPU, for the 400³ voxels model. The total time for this homogenization procedure is accelerated by 290x. Furthermore, once again, the node-by-node approach provided the best performance, among the GPU solutions. This is illustrated in Figure 7.5, a plot of time per PCG iteration versus number of DOFs. Elapsed times for the whole homogenization process are portrayed in Figure 7.9.

			CPU	CPU		'U EbE	
Dim.	DOFs	PCG its.	PCG time [s]	Total	PCG time [s]	Total	
[voxels]	$\times 10^{6}$	(x_1)	(x_1)	time [s]	(x_1)	time [s]	
100^{3}	3.0	224	74.0	510	0.39	4.09	
200^{3}	24.0	376	997	7110	5.22	47.7	
300^{3}	81.0	453	4394	33723	21.6	203	
400^{3}	192.0	538	12407	91530 *	60.6	534	
			GPU NI	oN	GPU DOFbDOF		
Dim.	DOFs	PCG its.	PCG time [s]	Total	PCG time [s]	Total	
[voxels]	$\times 10^{6}$	(x_1)	(x_1)	time [s]	(x_1)	time [s]	
100^{3}	3.0	224	0.21	2.77	0.33	3.60	
200^{3}	24.0	376	2.60	29.1	4.42	41.6	
300^{3}	81.0	453	11.1	119	17.9	173	
400^{3}	192.0	538	30.9	314	50.6	462	

* The 400^3 voxels model was run in CPU on only one direction (x_1) . The total time is an estimate, considering the average time per PCG iteration, and the number of iterations observed with the other solvers for the five remaining directions.

Table 7.9: Time metrics for elasticity analysis of the cast iron sample

Recalling the motivation of this work, stated in Section 1.1, it was said that a previous CPU implementation, presented in Pereira et al. [61], was capable of running the simulations of elasticity for a 400³ voxels model in about 12h, with a 32-core CPU. The model in question is the cast iron sample studied here. This time is smaller than the one achieved with the CPU solution exposed in Table 7.9, which makes sense, as the latter was run with 16 parallel threads and is less time-efficient, in exchange for allocating less memory. However, it is interesting to see that the GPU solvers, especially the node-by-node, are able to obtain equivalent results in significantly reduced runtime. Table 7.10 synthesizes this achievement. 230x speed up was obtained for the PCG method, while total time was accelerated by 135x.



Figure 7.5: Mean time per PCG iteration [s] vs number of DOFs, for elasticity analysis of the cast iron sample

Solver	PCG time	Total time	E_{average} [GPa]
Pereira et al. [61] 32-core CPU	$\sim 2 h$	$\sim\!12~{\rm h}$	181.877
node-by-node GPU	$\begin{array}{c} 30.9 \text{ s} \\ (\sim 290 \text{x faster}) \end{array}$	$\begin{array}{c} 314 \text{ s} \\ (\sim 135 \text{x faster}) \end{array}$	181.885

Table 7.10: Time metrics for the homogenization of elasticity of the 400^3 voxels cast iron sample, comparing to a previous implementation in CPU [61]

In regards to memory allocation, it is important to observe that the intended O(n) space complexity of the assembly-free approach was achieved. In addition, recalling the dimensions of models seen being analyzed with a single GPU in the literature (Table 2.1), it is notorious that the experiments presented so far in this work reach a usual threshold of 10~100 million DOFs, with relative low memory cost. Figures 7.6 and 7.7 respectively depict the allocated DRAM for the thermal conductivity and elasticity simulations performed on the cast iron sample. For elasticity, the allocation for node-by-node and DOF-by-DOF solutions is the same. It stands out that the elasticity analysis of the 400³ voxels model, of 192 million DOFs, allocates about 3 GB in the GPU with the proposed MParPCG solver, that stores four arrays. For a finite element model of similar dimensions, Apostolou [4], for example, allocated 11.4 GB.



Memory metrics for thermal conductivity of cast iron

Figure 7.6: Allocated memory [MB] vs number of DOFs, for thermal conductivity analysis



Figure 7.7: Allocated memory [MB] vs number of DOFs, for elasticity analysis



Metrics for thermal conductivity of cast iron

Figure 7.8: Total time [s] vs number of DOFs, for thermal conductivity analysis



Figure 7.9: Total time [s] vs number of DOFs, for elasticity analysis

7.2.2.2 Alternative implementations and initial guesses

The focus is now shifted to the alternative solvers discussed in Section 5.3, and the proposed methodologies to obtain good initial guesses for the PCG method from coarse meshes, seen in Chapter 6. To this end, the elasticity analyses for the 400³ voxels model of the cast iron sample were also conducted with the **xrd**, **xsd**, **rd** and **sd** solvers, employing different initial guess considerations: $\mathbf{x}_0 = \vec{0}$, one search for initial guesses, and then two recursive searches. As the node-by-node strategy was the one with best performance in the previous tests, it was adopted for all of the following experiments. Table 7.11 details the time metrics for these simulations. The provided number of iterations refers to the steps of the PCG method in the most refined mesh, and the time to compute an initial guess from coarse meshes is added to the time metric for the solution of the system.

$3D$ elasticity - 400^3 voxels cast iron - 192 million DOFs							
	$\mathbf{x}_0 = \vec{0}$		1 recursion for \mathbf{x}_0		2 recursions		
Solver	PCG time [s]	Total	PCG time [s]	Total	PCG time [s]	Total	Memory
Solver	(x_1)	time [s]	(x_1)	time [s]	(x_1)	time [s]	(device)
	iterations = 538		iterations=254		iterations = 254		[GB]
MParPCG	30.9	314	16.9	228	16.1	222	3.20
xrd	61.9	545	31.8	354	30.9	350	2.43
xsd	62.6	547	3.19	355	31.0	351	2.43
rd	426	3251	187	1736	186	1732	1.66
sd	459	3480	217	1943	215	1941	1.66

Table 7.11: Time and memory metrics for tests with all implemented solvers in GPU, considering initial guesses for the PCG method

From Table 7.11, it is clear that there is a memory versus time trade-off between the implemented solvers. The MParPCG solver should be the preferred one in the majority of cases, as it is faster, but it also does not reach the same size limits as the others. As image dimensions increase, the xrd, xsd, rd and sd solvers become options to make the analysis feasible, without upgrading the hardware. That being said, it is evident that there is an especially heavy performance toll to the solvers rd and sd, which is expected, due to the amount of data that is transferred at each iteration. These should only be employed on extremely large-scale problems for single computers.

The recursive search for initial guesses allows for the time of each PCG run to be nearly halved, by reducing the number of iterations at the most refined mesh. However, in Figure 7.10, it is shown that the time reduction does not improve much with more than two recursive searches. This is because each successive recursion essentially halves the time of the solution in a coarsened mesh, which is relatively fast to begin with. Then, in Figure 7.11, time metrics for solutions with the PCG method are presented for the MParPCG, xrd and xsd solvers, considering images with increasing dimensions, as in Table 7.9. It is interesting to notice that the adoption of a good initial guess enables solutions with the xrd and xsd solvers, which allocate less memory, with times close to the MParPCG solution with $\mathbf{x}_0 = \vec{0}$. It is important to mention that all solutions in coarse meshes employ the MParPCG solver, as memory allocation is guaranteed to be smaller than that of any of the implemented solvers at the refined mesh.



Metrics for elasticity of 400^3 voxels - 192M DOFs - tol=1e-05

Figure 7.10: Elapsed time for PCG solution [s] vs. number of recursive searches for initial guesses.



Metrics for elasticity with 0 or 2 initial guesses

Figure 7.11: Elapsed time for PCG solution [s] vs. DOFs, comparing $\mathbf{x}_0 = \vec{0}$ to two recursive searches for an initial guess.

7.3 Largest possible synthetic sample

At last, in this Section, a push of the size limits for the image-based models is presented. The five different solvers are tested with a synthetic model that fills the global memory of the GPU, accordingly to the tendency of allocation of each solution. In Subsection 7.3.1, metrics are presented for simulations ran in the desktop computer employed so far, detailed in Table 7.1. Then, in Subsection 7.3.2, large-scale analyses are conducted in a personal laptop equipped with a CUDA-enabled device.

The synthetic image adopted for these experiments was generated with a perlin noise in two directions, similarly to what was done by Arbenz et al. [5] to create a synthetic bone micro-structure, and a periodic behavior in a third direction. The result is a model that holds some similarities to stratified composites. Figure 7.12 depicts the studied domain with 242^3 voxels. The two different phases were admitted to have the same physical properties as those of the cast iron sample, shown in Table 7.4. The phase in blue corresponds to graphite, and the one in yellow is a ferritic matrix.



Figure 7.12: (a) Synthetic voxel-based model generated with perlin noise, (b) details of the blue phase

7.3.1 Desktop computer

Thermal conductivity and elasticity analyses were performed, their metrics are exposed in Tables 7.12 and 7.13, respectively. A numerical tolerance of 1e-05 was considered for dimensionless relative norms of residuals. Two recursive searches for initial guesses in coarse meshes were adopted. The **rd** and **sd** solvers employed eight CUDA streams dedicated to the memory transfer from device to host at each iteration. Such a number of streams was chosen after some tests with other possible quantities, and verification of performance with the aid of the NsightTM tool. More thorough code profiling and fine tuning of the proposed solutions certainly are open matters for future work.

The dimensions of the simulations detailed in Tables 7.12 and 7.13 exceed the size limits of every report the author has found in the literature for large-scale finite element analysis with a single GPU. Recalling Table 2.1, in Chapter 2, the simulation that comes closer to the depicted results was carried out by Apostolou [4] in 2020, who solved a 166 million DOF problem, allocating 11.4 GB in the device. Müller et al. [56] presented the solution of a considerably smaller model, but stated that their implementation could handle problems of up to ~ 300 million DOFs with a GPU of 6 GB RAM. The programs

Column	Dimensions	DOFs	PCG its.	PCG time [s]	Total	Mem.[GB]	Mem.[GB]
Solver	[voxels]	$\times 10^{6}$	(x_1)	(x_1)	time [s]	(host)	(device)
MParPCG	$724 \times 724 \times 724$	380	277	31.6	254	5.32	6.83
xrd	$828 \times 828 \times 828$	568	273	44.1	359	7.96	7.95
xsd	$828 \times 828 \times 828$	568	273	43.5	357	7.96	7.95
rd	$925 \times 925 \times 850$	727	381	1063	3071	10.2	7.28
sd	$925 \times 925 \times 850$	727	381	1190	3342	10.2	7.28

Table 7.12: Metrics for thermal conductivity analyses of a synthetic model at nearly full GPU memory capacity with different solvers, using a desktop computer

Column	Dimensions	DOFs	PCG its.	PCG time [s]	Total	Mem.[GB]	Mem.[GB]
Solver	[voxels]	$\times 10^{6}$	(x_1)	(x_1)	time [s]	(host)	(device)
MParPCG	$724 \times 724 \times 300$	472	276	45.9	638	4.72	7.86
xrd	$724 \times 724 \times 390$	613	280	119	1372	6.14	7.77
xsd	$724 \times 724 \times 390$	613	280	117	1350	6.14	7.77
rd	$724 \times 724 \times 530$	833	276	976	9764	8.34	7.23
sd	$724 \times 724 \times 530$	833	276	1020	10402	8.34	7.23

Table 7.13: Metrics for elasticity analyses of a synthetic model at nearly full GPU memory capacity with different solvers, using a desktop computer

developed in this work have been demonstrated to be able to handle models of up to 830 million DOFs under 8 GB of allocated memory in the GPU. It is estimated that the rd and sd solvers can deal with 900 million DOFs analyses with the hardware employed here. Duarte [18] and Arbenz et al. [5] solved problems of this scale (and much larger) resorting to supercomputers and GPU clusters, respectively.

In regards to elapsed time, it is noteworthy that the MParPCG solver was able to obtain solutions for problems with nearly 475 million DOFs in under a minute, allowing for large-scale image-based numerical homogenization studies to be conducted in a matter of five to ten minutes. The predecessor solution in CPU, optimized to explore parallelism with OpenMP, would take more than a day to run such a simulation. Additionally, it is interesting to observe that, even though the xrd and xsd solvers are slower than the MParPCG implementation, they are relatively in the same order of time consumption, and enable for larger models to be analyzed. As it can be seen in Table 7.13, these two can be employed to run the PCG method for problems with more than 600 million DOFs in under two minutes, allocating less than 8 GB in the global memory of the device.

7.3.2 Laptop

The third goal of this work, stated in Section 1.2, was to enable the solution of largescale homogenization problems in a laptop. Employing the memory-efficient rd and sd

Laptop Specifications				
O.S.	Linux Mint 19.3 Tricia			
CPU	Intel Core i7-7500U			
Clock rate	$2.70 \mathrm{GHz}$			
DRAM	8 GB			
Cores (threads)	2(4)			
GPU	Nvidia GeForce 940MX			
Clock rate	$1.19 \mathrm{GHz}$			
DRAM	$4 \mathrm{GB}$			
CUDA cores	384			
$\operatorname{Architecture}$	Maxwell			

Table 7.14: Specifications of the laptop used in the final experiment

solvers, it is indeed possible to solve problems of more than 100 million DOFs in a personal computer, in relatively small time. To demonstrate this, thermal conductivity analyses of the synthetic model were performed using the computer specified in Table 7.14. It was taken as input an image-based model of 724^3 voxels, which corresponds to the useful part of a cylindrical sample imaged via μ CT with 1024^3 voxels [75], as illustrated in Figure 7.13.



Figure 7.13: Useful part of a sample imaged via μ CT with 1024³ voxels.

Numerical tolerance for the dimensionless norms of residuals from the PCG method was adopted to be 1e-05, and three recursive searches for initial guesses were employed. The rd and sd solvers used sixteen CUDA streams for the data transfer from device to host at each iteration. Time and memory metrics are presented in Table 7.15.

As it can be seen in Table 7.15, the developed methodologies in fact can be applied even to a laptop. The GPU used for the analyses in this Subsection is not, by any means, a powerful one, and regardless of that, the implemented massively parallel solvers allowed for the solutions of systems with almost 400 million DOFs in about 15 minutes,
Solver	Dimensions	DOFs	PCG its.	PCG time [s]	Total	Mem.[GB]	Mem.[GB]			
	[voxels]	$\times 10^{6}$	(x_1)	(x_1)	time [s]	(host)	(device)			
rd	724^{3}	380	277	946	3231	5.32	3.80			
sd	724^{3}	380	277	988	3380	5.32	3.80			

Table 7.15: Metrics for thermal conductivity analyses of a 724^3 voxels synthetic model with a laptop

allocating less than 4 GB in the device. The whole homogenization process took less than 1 hour. In perspective, the same studies were carried out in the CPU of the aforementioned desktop computer, specified in Table 7.1, taking about 2 hours and 15 minutes per system solution with the PCG method, and roughly 6 hours and 30 minutes for the complete homogenization procedure. Considering that it is desirable that simulations of this sort can be eventually performed in a laboratory of materials, it is very much interesting to observe that the large-scale analyses can be conducted in personal computers equipped with CUDA-enabled devices, without demand of long runtime.

Chapter 8

Conclusion

Image-based numerical homogenization is a growing trend in the field of Materials Science that is branching out and becoming intertwined with High-Performance Computing, due to the large-scale nature of the computational problems that need to be tackled. When solving the governing equations of the involved physical phenomena with the Finite Element Method, the numerical simulations essentially turn into solving large sparse systems of equations, with hundreds of millions, or even billions, of DOFs. In the recent literature, there are works that report solutions of systems of such size, and even larger, but resorting to supercomputers and clusters [5, 18]. On the other hand, there are many references for large-scale system solving in GPUs that achieve interesting performance gains [1, 4, 37, 39, 43, 47, 49, 54, 56, 66], but most are limited by the available memory of the devices.

In this work, multiple memory-efficient massively parallel PCG solvers for assemblyfree FEM applied to image-based numerical homogenization were presented. A previous C++ program in CPU was taken as starting point. The new GPU implementations in CUDA C were able to provide up to 230x speed up for the solutions of systems with the PCG method, 135x for the whole process of homogenization. Solvers were implemented to allocate four, three or only two variable arrays in the device, reducing from the five arrays commonly seen in the literature. Because of this, two of the presented solvers were able to handle analyses of up to ~800 million DOFs with a single GPU in reasonable time, less than 20 minutes per solution of the PCG method. In comparison, the previous CPU solver would take about 2 hours for the same process, with a model of 192 million DOFs [61]. Furthermore, ~400 million DOFs analyses were able to be carried out in a personal laptop, with a GPU of just 4 GB DRAM. From a literature review, no previous works were found reporting solutions for models of such dimensions in a single personal-use GPU. The developed programs were validated with an analytical benchmark and experimental results of a cast iron sample that matched previous findings [61]. This was especially important to be observed, as it was decided to switch from double to single precision floating point variables in the GPU programs. Such choice was mostly motivated by the memory efficiency.

It was shown that, for image-based simulations, the node-by-node strategy is the most fitting one to be implemented in GPUs. Authors such as Kiran et al. [37] state that the element-by-element approach is more suitable when working with unstructured meshes, as it avoids unbalanced workloads for each thread, but, when taking the regularity of the images into account, every node has the same amount of computations to perform, and race conditions are naturally eliminated. This removes the need for coloring algorithms, for example.

A methodology was proposed to obtain good initial guesses for the PCG iterative scheme, employing recursive searches in coarsened meshes. In a nutshell, the idea is to solve the system of equations on images of gradually increasing resolution, adopting the solution of an immediate coarse model as a starting point, until the target model is solved. It was observed that the presented strategy can halve the number of iterations at the refined mesh, leading to an acceleration of roughly 2x of the PCG method.

The matters addressed in this work open a path for future developments, such as:

- Implementation of the memory-efficient solutions in a multi-GPU environment. In doing so, it is expected that much larger models can be studied.
- Investigation of domain decomposition techniques, so that portions of the domain can be decoupled and analyzed in a distributed system, or by parts, in a single computer.
- Implementation of simulations for different physical phenomena, such as steady state fluid flow in porous media, so that the permeability of samples can be evaluated.
- Study of more sophisticated image processing techniques for the coarsening of imagebased meshes, in an effort to further accelerate convergence of the PCG method.
- Investigation of preconditioning better than the Jacobi. It is desired to explore other preconditioners, such as the incomplete Cholesky factorization, or the multi-grid, with an assembly-free approach.

References

- [1] AKBARIYEH, A. Large scale finite element analysis using GPU parallel computing. Dissertação de Mestrado. University of Texas, Arlington, USA, May 2012.
- [2] ANDREASSEN, E.; ANDREASEN, C. How to determine composite material properties using numerical homogenization. *Computational Materials Science* 83 (2014), 488– 495. Elsevier.
- [3] ANDREASSEN, E.; CLAUSEN, A.; SCHEVENELS, M.; LAZAROV, B.; SIGMUND, O. Efficient topology optimization in matlab using 88 lines of code. *Struct. Multidisc. Optim.* 43 (2011), 1–16. Springer-Verlag.
- [4] APOSTOLOU, P. High performance matrix-free method for large-scale finite element analysis on graphics processing units. Dissertação de Mestrado. University of Pittsburgh, Pittsburgh, USA, April 2020.
- [5] ARBENZ, P.; FLAIG, C.; KELLENBERGER, D. Bone structure analysis on multiple GPGPUs. Journal of Parallel and Distributed Computing 74, 10 (2014), 2941–2950.
- [6] BARBERO, E. J. Finite Element Analysis of Composite Materials using ANSYS, 2 ed. CRC Press, Boca Raton, USA, 2014.
- [7] BEKAS, C.; CURIONI, A.; ARBENZ, P.; FLAIG, C.; VAN LENTHE, G. H.; MÜLLER, R.; WIRTH, A. J. Extreme scalability challenges in micro-finite element simulations of human bone. *Concurrency and Computation: Practice and Experience 22*, 16 (2010), 2282–2296.
- [8] BENZI, M. Preconditioning techniques for large linear systems: A survey. Journal of Computational Physics 182, 2 (2002), 418-477.
- [9] BLAL, N.; GRAVOUIL, A. Non-intrusive data learning based computational homogenization of materials with uncertainties. *Computational Mechanics* 64, 3 (2019), 807 - 828.
- [10] CAREY, G. F.; JIANG, B.-N. Element-by-element linear and nonlinear solution schemes. Communications in Applied Numerical Methods 2, 2 (1986), 145 – 153.
- [11] CARTRAUD, P.; MESSAGER, T. Computational homogenization of periodic beamlike structures. International Journal of Solids and Structures 43, 3 (2006), 686–696.
- [12] CECKA, C.; LEW, A. J.; DARVE, E. Assembly of finite element methods on graphics processors. International Journal for Numerical Methods in Engineering 85, 5 (2011), 640–669.

- [13] CIORANESCU, D.; PAULIN, J. S. J. Homogenization in open sets with holes. Journal of Mathematical Analysis and Applications 71, 2 (1979), 590-607.
- [14] COOK, R. D. Finite Element Modeling for Stress Analysis, 1 ed. John Wiley and Sons, Madison, USA, 1994.
- [15] DA CRUZ, J. P.; OLIVEIRA, J.; TEIXEIRA-DIAS, F. Asymptotic homogenisation in linear elasticity. part i: Mathematical formulation and finite element modelling. *Computational Materials Science* 45, 4 (2009), 1073–1080.
- [16] DABROWSKI, M.; KROTKIEWSKI, M.; SCHMID, D. Milamin: Matlab-based finite element method solver for large problems. *Geochemistry Geophysics Geosystems 9*, 4 (2008). AGU and the Geochemical Society.
- [17] DAIBES, J. V. D. Implementação da solução do fluxo de potência pelo método de newton-raphson em uma arquitetura híbrida CPU-GPU. Dissertação de Mestrado. Universidade Federal Fluminense, Niterói, Brasil, 2021.
- [18] DUARTE, L. S. TopSim: A plugin-based framework for large-scale numerical analysis. Tese de Doutorado, Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brasil, September 2016.
- [19] DUARTE, L. S.; CELES, W.; PEREIRA, A.; MENEZES, I. F. M.; PAULINO, G. H. Polytop++: an efficient alternative for serial and parallel topology optimization on cpus and GPUs. *Structural and Multidisciplinary Optimization 52*, 5 (2015), 845–859.
- [20] EPOV, M.; SHURINA, E.; KUTISCHEVA, A. Computation of effective resistivity in materials with microinclusions by a heterogeneous multiscale finite element method. *Physical Mesomechanics* 20, 4 (2017), 407–416. Pleiades Publishing.
- [21] ERHEL, J.; TRAYNARD, A.; VIDRASCU, M. An element-by-element preconditioned conjugate gradient method implemented on a vector computer. *Parallel Computing* 17, 9 (1991), 1051–1065.
- [22] FELIPPA, C. Introduction to finite element methods (ASEN 5007) fall 2005. Lecture notes, 2005. Department of Aerospace Engineering Sciences, University of Colorado at Boulder.
- [23] FIALKO, S.; ZEGLEN, F. Preconditioned conjugate gradient method for solution of large finite element problems on cpu and GPU. Journal of Telecommunications and Information Technology 2 (2016), 26-33.
- [24] FLAIG, C.; ARBENZ, P. A scalable memory efficient multigrid solver for micro-finite element analyses based on ct images. *Parallel Computing* 37, 12 (2011), 846–854. 6th International Workshop on Parallel Matrix Algorithms and Applications (PMAA'10).
- [25] GUEDES, J. M.; KIKUCHI, N. Preprocessing and postprocessing for materials based on the homogenization method with adaptive finite element methods. *Computer Methods in Applied Mechanics and Engineering* 83, 2 (1990), 143–198.
- [26] GULLERUD, A. S.; DODDS, R. H. MPI-based implementation of a PCG solver using an ebe architecture and preconditioner for implicit, 3-d finite element analysis. *Computers & Structures 79*, 5 (2001), 553-575.

- [27] GUPTA, P. Cuda refresher: The cuda programming model. Website, Nvidia Developer Blog. Available at https://developer.nvidia.com/blog/ cuda-refresher-cuda-programming-model/. Last accessed in June 27th of 2021.
- [28] HAHN, D.; ÖZISIK, N. Heat Conduction, 3 ed. Wiley, Hoboken, New Jersy, 2012.
- [29] HASHIN, Z.; SHTRIKMAN, S. On some variational principles in anisotropic and nonhomogeneous elasticity. Journal of the Mechanics and Physics of Solids 10, 4 (1962), 335–342.
- [30] HEATH, M. T. Scientific Computing: An introductory survey, 2 ed. SIAM, Philadelphia, USA, 2018.
- [31] HELFENSTEIN, R.; KOKO, J. Parallel preconditioned conjugate gradient algorithm on GPU. Journal of Computational and Applied Mathematics 236 (2012), 3584–3590. Elsevier.
- [32] HESTENES, M.; STIEFEL, E. Methods of conjugate gradients for solving linear systems. Journal of Research of the National Bureau of Standards 49, 6 (Dezembro 1952), 409–436.
- [33] HILL, R. Elastic properties of reinforced solids: Some theoretical principles. *Journal* of the Mechanics and Physics of Solids 11, 5 (1963), 357–372.
- [34] HUGHES, T. J.; LEVIT, I.; WINGET, J. An element-by-element solution algorithm for problems of structural and solid mechanics. *Computer Methods in Applied Mechanics and Engineering* 36, 2 (1983), 241–254.
- [35] KANIT, T.; FOREST, S.; GALLIET, I.; MOUNOURY, V.; JEULIN, D. Determination of the size of the representative volume element for random composites: statistical and numerical approach. *International Journal of Solids and Structures* 40, 13 (2003), 3647–3679.
- [36] KESSLER, A. Matrix-free voxel-based finite element method for materials with heterogeneous microstructures. Tese de Doutorado, Bauhaus-Universitat Weimar, Weimar, Germany, December 2017.
- [37] KIRAN, U.; GAUTAM, S.; SHARMA, D. GPU-based matrix-free finite element solver exploiting symmetry of elemental matrices. *Computing 102* (2020), 1941– 1965. Pleiades Publishing.
- [38] KIRK, D. B.; MEI W. HWU, W. Programming Massively Parallel Processors, 1 ed. Morgan Kaufmann, Burlington, USA, 2010. NVIDIA.
- [39] KRONBICHLER, M.; LJUNGKVIST, K. Multigrid for matrix-free high-order finite element computations on graphics processors. ACM Trans. Parallel Comput. 6, 1 (May 2019).
- [40] LANCZOS, C. Solution of systems of linear equations by minimized iterations. Journal of Research of the National Bureau of Standards 49, 1 (Julho 1952), 33–53.

- [41] LIU, X.; RÉTHORÉ, J.; BAIETTO, M.-C.; SAINSOT, P.; LUBRECHT, A. A. An efficient strategy for large scale 3d simulation of heterogeneous materials to predict effective thermal conductivity. *Computational Materials Science 166* (2019), 265– 275.
- [42] LIU, X.; RÉTHORÉ, J.; BAIETTO, M.-C.; SAINSOT, P.; LUBRECHT, A. A. An efficient finite element based multigrid method for simulations of the mechanical behavior of heterogeneous materials using ct images. *Computational Mechanics 66*, 6 (Dec 2020), 1427–1441.
- [43] LOEB, A.; EARLS, C. Analysis of heterogeneous computing approaches to simulating heat transfer in heterogeneous material. *Journal of Parallel and Distributed Computing* 133 (2019), 1–17.
- [44] MADEIRA, D. L. A. G-MPP: Método para simulação massiva partícula-partícula de N-corpos em cluster de GPUs. Tese de Doutorado, Universidade Federal Fluminense, Niterói, Brasil, 2015.
- [45] MARINO, M.; HUDOBIVNIK, B.; WRIGGERS, P. Computational homogenization of polycrystalline materials with the virtual element method. *Computer Methods in Applied Mechanics and Engineering 355* (2019), 349–372.
- [46] MARTHA, L. F. Análise Matricial de Estruturas com Orientação a Objetos, 1 ed. Elsevier, Rio de Janeiro, Brasil, 2019.
- [47] MARTÍNEZ-FRUTOS, J.; HERRERO-PÉREZ, D. Efficient matrix-free GPU implementation of fixed grid finite element analysis. *Finite Elements in Analysis and Design 104* (2015), 61–71.
- [48] MARTÍNEZ-FRUTOS, J.; HERRERO-PÉREZ, D. Large-scale robust topology optimization using multi-GPU systems. Computer Methods in Applied Mechanics and Engineering 311 (2016), 393-414.
- [49] MARTÍNEZ-FRUTOS, J.; MARTÍNEZ-CASTEJÓN, P. J.; HERRERO-PÉREZ, D. Finegrained GPU implementation of assembly-free iterative solver for finite element problems. *Computers & Structures 157* (2015), 9–18.
- [50] MASE, G. T.; SMELSER, R. E.; MASE, G. E. Continuum Mechanines for Engineers, 3 ed. CRC Press, Boca Raton, USA, 2010.
- [51] MATHWORKS. Techniques to improve performance. Website. Available at https://www.mathworks.com/help/matlab/matlab_prog/ techniques-for-improving-performance.html. Last accessed in May 19th of 2021.
- [52] MATHWORKS. Vectorization. Website. Available at https://www.mathworks. com/help/matlab/matlab_prog/vectorization.html. Last accessed in May 19th of 2021.
- [53] MICHEL, J.; MOULINEC, H.; SUQUET, P. Effective properties of composite materials with periodic microstructure: a computational approach. *Comput. Methods Appl. Mech. Engrg.* 172 (1999), 109–143. Elsevier.

- [54] MIRZENDEHDEL, A. M. Assembly-free structural dynamics on CPU and GPU. Dissertação de Mestrado. University of Wisconsin, Madison, USA, 2014.
- [55] MOULINEC, H.; SUQUET, P. A numerical method for computing the overall response of nonlinear composites with complex microstructure. *Computer Methods in Applied Mechanics and Engineering* 157, 1 (1998), 69–94.
- [56] MÜLLER, E.; GUO, X.; SCHEICHL, R.; SHI, S. Matrix-free GPU implementation of a preconditioned conjugate gradient solver for anisotropic elliptic pdes. *Computing* and Visualization in Science 16, 2 (Apr 2013), 41–58.
- [57] NGUYEN, V.-D.; BÉCHET, E.; GEUZAINE, C.; NOELS, L. Imposing periodic boundary condition on arbitrary meshes by polynomial interpolation. *Computational Materials Science* 55 (2012), 390–406.
- [58] NVIDIA. Cuda toolkit documentation v11.3.0. Website. Available at https:// docs.nvidia.com/cuda/index.html. Last accessed in May 19th of 2021.
- [59] OLIVEIRA, J.; DA CRUZ, J. P.; TEIXEIRA-DIAS, F. Asymptotic homogenisation in linear elasticity. part ii: Finite element procedures and multiscale applications. *Computational Materials Science* 45, 4 (2009), 1081–1096.
- [60] PEREIRA, A.; ANFLOR, C.; BETANCUR, A.; LEIDERMAN, R. ds-uct-001: Cast Iron GGG40: X-Ray micro-CT of a nodular cast iron sample class GGG40., May 2020.
- [61] PEREIRA, A.; COSTA, M.; ANFLOR, C.; PARDAL, J.; LEIDERMAN, R. Estimating the effective elastic parameters of nodular cast iron from micro tomographic imaging and multiscale finite elements: Comparison between numerical and experimental results. *Metals* 695, 8 (2018). MDPI.
- [62] PERRINS, W.; MCKENZIE, D.; MCPHEDRAN, R. Transport properties of regular arrays of cylinders. Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences 369 (May 1979), 207–225.
- [63] PIKLE, N. K.; SATHE, S. R.; VYAVAHARE, A. Y. GPGPU-based parallel computing applied in the fem using the conjugate gradient algorithm: a review. Sadhana 43, 111 (2018). Springer.
- [64] PIKLE, N. K.; SATHE, S. R.; VYAVAHARE, A. Y. Accelerating the finite element analysis of functionally graded materials using fixed-grid strategy on cuda-enabled GPUs. *Concurrency and Computation: Practice and Experience 31*, 17 (2019).
- [65] PIVOVAROV, D.; STEINMANN, P. On stochastic fem based computational homogenization of magneto-active heterogeneous materials with random microstructure. *Computational Mechanics* 58, 6 (2016), 981–1002.
- [66] REGULY, I.; GILES, M. Finite element algorithms and data structures on graphical processing units. International Journal of Parallel Programming 43, 2 (2013).
- [67] RIBEIRO, M. C. Toward an ultrasonic inspecting method to detect and classify adhesive bonding defects in real time. Dissertação de Mestrado. Universidade Federal Fluminense, Niterói, Brasil, 2019.

- [68] SADABA, S.; HERRAEZ, M.; NAYA, F.; GONZALEZ, C.; LLORCA, J.; LOPES, C. Special-purpose elements to impose periodic boundary conditions for multiscale computational homogenization of composite materials with the explicit finite element method. *Composite Structures 208* (2019), 434–441.
- [69] SANDERS, J.; KANDROT, E. CUDA by example: An Introduction to General-Purpose GPU Programming, 1 ed. Addison Wesley, Boston, USA, 2011. NVIDIA.
- [70] SAPUCAIA, V. Formulação de elementos de contorno baseada em pixels para determinação de condutividade térmica efetiva de materiais heterogêneos. Dissertação de Mestrado. Universidade Federal Fluminense, Brasil, February 2021.
- [71] SEGURADO, J.; LEBENSOHN, R. A.; LLORCA, J. Computational homogenization of polycrystals. In Advances in Crystals and Elastic Metamaterials, Part 1, M. I. Hussein, Ed., vol. 51 of Advances in Applied Mechanics. Elsevier, 2018, pp. 1–114.
- [72] SHEWCHUK, J. An introduction to the conjugate gradient method without the agonizing pain. Tech. rep., Carnegie Mellon University, Pittsburgh, USA, August 1994.
- [73] TERADA, K.; HORI, M.; KYOYA, T.; KIKUCHI, N. Simulation of the multi-scale convergence in computational homogenization approaches. *International Journal of Solids and Structures* 37, 16 (2000), 2285–2311.
- [74] VASCONCELLOS, E. C. Aceleração de Modelos da Eletrofisiologia cardíaca com GPUs. Tese de Doutorado, Universidade Federal Fluminense, Niterói, Brasil, 2019.
- [75] VIANNA, R. Homogeneização em duas etapas de compósitos pultrudados reforçados com fibras de vidro. Dissertação de Mestrado. Universidade Federal Fluminense, Brasil, February 2020.
- [76] VIANNA, R.; CUNHA, A.; AZEREDO, R.; LEIDERMAN, R.; PEREIRA, A. Computing effective permeability of porous media with fem and micro-ct: An educational approach. *Fluids* 16, 5 (2020). MDPI.
- [77] WU, Y.; LI, J.; YANG, Z.; MA, Z.; GUO, Y.; TAO, D.; YANG, T.; LIANG, M. Computational assessment of thermal conductivity of compacted graphite cast iron. Advances in Materials Science and Engineering 2019 (2019). Hindawi.

APPENDIX A – Isotropic materials

A material is said to be isotropic when its physical behavior is admitted to be the same on all directions. For example, concrete and steel are commonly studied as isotropic. Taking this into account allows for simplifications to be made to the constitutive tensors, as it is shown ahead for thermal conductivity and elasticity.

Fourier's law

A single coefficient κ is enough to characterize the thermal conductivity of an isotropic material. For this reason, it is common to substitute the second order constitutive tensor for a scalar, as in

$$\begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix} = -\begin{bmatrix} \kappa & 0 & 0 \\ 0 & \kappa & 0 \\ 0 & 0 & \kappa \end{bmatrix} \begin{bmatrix} \nabla T_1 \\ \nabla T_2 \\ \nabla T_3 \end{bmatrix} \Rightarrow \begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix} = -\kappa \begin{bmatrix} \nabla T_1 \\ \nabla T_2 \\ \nabla T_3 \end{bmatrix}.$$
(A.1)

Hooke's law (Voigt notation)

Analogously, the isotropic stiffness tensor can be described in terms of a few constants. In this work, we consider the Young's modulus E and Poisson's ratio ν . The constitutive equation for elasticity, in this case, is written as

$$\begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{23} \\ \sigma_{23} \\ \sigma_{13} \\ \sigma_{12} \end{bmatrix} = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} (1-\nu) & \nu & \nu & 0 & 0 & 0 \\ \nu & (1-\nu) & \nu & 0 & 0 & 0 \\ \nu & \nu & (1-\nu) & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{(1-2\nu)}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{(1-2\nu)}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{(1-2\nu)}{2} \end{bmatrix} \begin{bmatrix} \varepsilon_{11} \\ \varepsilon_{22} \\ \varepsilon_{33} \\ \gamma_{23} \\ \gamma_{13} \\ \gamma_{12} \end{bmatrix}.$$
(A.2)

APPENDIX B - Orthotropic materials

A material is said to be orthotropic when its physical behavior varies accordingly to an orthonormal basis of directions. This is the case of materials with directional fibers and laminae in their micro-structure, for instance. Such a consideration allows for the constitutive tensors to be defined by a set of material constants, as it is shown ahead for thermal conductivity and elasticity.

Fourier's law

Three coefficients κ_{11} , κ_{22} , and κ_{33} are required to characterize the thermal conductivity of an orthotropic material. The constitutive tensor is expressed as

$$\begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix} = -\begin{bmatrix} \kappa_{11} & 0 & 0 \\ 0 & \kappa_{22} & 0 \\ 0 & 0 & \kappa_{33} \end{bmatrix} \begin{bmatrix} \nabla T_1 \\ \nabla T_2 \\ \nabla T_3 \end{bmatrix} .$$
(B.1)

Hooke's law (Voigt notation)

Analogously, the orthotropic constitutive equation for elasticity can be described in terms of constants. We consider a set of Young's moduli E_1 , E_2 , and E_3 , Poisson's ratios ν_{23} , ν_{13} , and ν_{12} , and shear moduli G_{23} , G_{13} , and G_{12} . It is usual to write this expression via a compliance matrix, which corresponds to the inverse of the matrix representation of the stiffness tensor, as in

$$\begin{bmatrix} \varepsilon_{11} \\ \varepsilon_{22} \\ \varepsilon_{33} \\ \gamma_{23} \\ \gamma_{13} \\ \gamma_{12} \end{bmatrix} = \begin{bmatrix} \frac{1}{E_1} & -\frac{\nu_{12}}{E_2} & -\frac{\nu_{13}}{E_3} & 0 & 0 & 0 \\ -\frac{\nu_{12}}{E_1} & \frac{1}{E_2} & -\frac{\nu_{23}}{E_3} & 0 & 0 & 0 \\ -\frac{\nu_{13}}{E_1} & -\frac{\nu_{23}}{E_2} & \frac{1}{E_3} & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{G_{23}} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{G_{13}} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{G_{12}} \end{bmatrix} \begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{23} \\ \sigma_{13} \\ \sigma_{12} \end{bmatrix}.$$
(B.2)

APPENDIX C – Analytical solutions for local FE matrices

When dealing with pixel and voxel-based meshes, it is possible to obtain analytical solutions for the local finite element matrices, as the geometry of every element is predetermined. The domain of each element, at a local reference, can be described as

$$\mathbf{\Omega}_{\mathbf{e}} = \{ (x_1, x_2) \mid x_1 \in [0, 1], x_2 \in [0, 1] \},$$
(C.1)

in 2D, and

$$\mathbf{\Omega}_{\mathbf{e}} = \{ (x_1, x_2, x_3) \mid x_1 \in [0, 1], x_2 \in [0, 1], x_3 \in [0, 1] \},$$
(C.2)

in 3D. These notions are illustrated in Figure C.1, where local coordinate systems and node numbering schemes are shown.



Figure C.1: (a) Pixel-based and (b) voxel-based finite elements.

In this work, linear shape functions are adopted and all materials at the micro-scale are assumed to be isotropic, which allows for the integral expressions that define the local matrices to be calculated with relative ease. The shape functions associated with each local node are

$$N_0(x_1, x_2) = (1 - x_1)(1 - x_2)$$
$$N_1(x_1, x_2) = x_1(1 - x_2)$$
$$N_2(x_1, x_2) = x_1x_2$$
$$N_3(x_1, x_2) = (1 - x_1)x_2$$

in 2D, and

$$\begin{split} N_0(x_1, x_2, x_3) &= (1 - x_1)(1 - x_2)x_3\\ N_1(x_1, x_2, x_3) &= x_1(1 - x_2)x_3\\ N_2(x_1, x_2, x_3) &= x_1x_2x_3\\ N_3(x_1, x_2, x_3) &= (1 - x_1)x_2x_3\\ N_4(x_1, x_2, x_3) &= (1 - x_1)(1 - x_2)(1 - x_3)\\ N_5(x_1, x_2, x_3) &= x_1(1 - x_2)(1 - x_3)\\ N_6(x_1, x_2, x_3) &= x_1x_2(1 - x_3)\\ N_7(x_1, x_2, x_3) &= (1 - x_1)x_2(1 - x_3) \end{split}$$

in 3D. MATLAB's symbolic math resources were employed to compute the following formulas.

2D heat conduction

The shape functions matrix is defined as

$$\mathbf{N} = \begin{bmatrix} N_0, & N_1, & N_2, & N_3 \end{bmatrix} . \tag{C.3}$$

The differential operator ∇ is applied to ${\bf N}$ as in

$$\nabla \mathbf{N} = \begin{bmatrix} \partial/\partial x_1 \\ \partial/\partial x_2 \end{bmatrix} \mathbf{N} . \tag{C.4}$$

Then, the local conductivity matrix is given by

$$\mathbf{K}_{e} = \int_{\Omega_{e}} (\nabla \mathbf{N})^{T} \kappa(\nabla \mathbf{N}) \, d\Omega = \kappa \begin{bmatrix} 2/3 & -1/6 & -1/3 & -1/6 \\ -1/6 & 2/3 & -1/6 & -1/3 \\ -1/3 & -1/6 & 2/3 & -1/6 \\ -1/6 & -1/3 & -1/6 & 2/3 \end{bmatrix} \,. \tag{C.5}$$

3D heat conduction

The shape functions matrix is defined as

$$\mathbf{N} = \begin{bmatrix} N_0, & N_1, & N_2, & N_3, & N_4, & N_5, & N_6, & N_7 \end{bmatrix} .$$
(C.6)

The differential operator ∇ is applied to ${\bf N}$ as in

$$\nabla \mathbf{N} = \begin{bmatrix} \partial/\partial x_1 \\ \partial/\partial x_2 \\ \partial/\partial x_3 \end{bmatrix} \mathbf{N} . \tag{C.7}$$

Then, the local conductivity matrix is given by

$$\mathbf{K}_e = \int_{\Omega_e} (\nabla \mathbf{N})^T \kappa(\nabla \mathbf{N}) \, d\Omega \quad ,$$

$$\mathbf{K}_{e} = \kappa \begin{bmatrix} 1/3 & 0 & -1/12 & 0 & 0 & -1/12 & -1/12 & -1/12 \\ 0 & 1/3 & 0 & -1/12 & -1/12 & 0 & -1/12 & -1/12 \\ -1/12 & 0 & 1/3 & 0 & -1/12 & -1/12 & 0 & -1/12 \\ 0 & -1/12 & 0 & 1/3 & -1/12 & -1/12 & 0 & 0 \\ 0 & -1/12 & -1/12 & -1/12 & 1/3 & 0 & -1/12 & 0 \\ -1/12 & 0 & -1/12 & -1/12 & 0 & 1/3 & 0 & -1/12 \\ -1/12 & -1/12 & 0 & -1/12 & -1/12 & 0 & 1/3 & 0 \\ -1/12 & -1/12 & 0 & 0 & -1/12 & 0 & 1/3 \end{bmatrix} . \quad (C.8)$$

2D elasticity

For elasticity problems, the variables consist of a vector field, which means that two components must be interpolated at each local node, one for each of its DOFs. The shape functions matrix is defined as

$$\mathbf{N} = \begin{bmatrix} N_0, & 0, & N_1, & 0, & N_2, & 0, & N_3, & 0\\ 0, & N_0, & 0, & N_1, & 0, & N_2, & 0, & N_3 \end{bmatrix} .$$
(C.9)

The differential operator $[\nabla]$ is applied to ${\bf N}$ as in

$$[\nabla]\mathbf{N} = \begin{bmatrix} \partial/\partial x_1 & 0\\ 0 & \partial/\partial x_2\\ \partial/\partial x_2 & \partial/\partial x_1 \end{bmatrix} \mathbf{N} .$$
(C.10)

Then, the local stiffness matrix is given by

$$\mathbf{K}_e = \int_{\Omega_e} ([\nabla] \mathbf{N})^T \mathbf{C} ([\nabla] \mathbf{N}) \, d\Omega \;\;,$$

$$\mathbf{K}_{e} = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} c_{1} & c_{4} & c_{6} & -c_{2} & c_{5} & -c_{4} & c_{3} & c_{2} \\ c_{4} & c_{1} & c_{2} & c_{3} & -c_{4} & c_{5} & -c_{2} & c_{6} \\ c_{6} & c_{2} & c_{1} & -c_{4} & c_{3} & -c_{2} & c_{5} & c_{4} \\ -c_{2} & c_{3} & -c_{4} & c_{1} & c_{2} & c_{6} & c_{4} & c_{5} \\ c_{5} & -c_{4} & c_{3} & c_{2} & c_{1} & c_{4} & c_{6} & -c_{2} \\ -c_{4} & c_{5} & -c_{2} & c_{6} & c_{4} & c_{1} & c_{2} & c_{3} \\ c_{3} & -c_{2} & c_{5} & c_{4} & c_{6} & c_{2} & c_{1} & -c_{4} \\ c_{2} & c_{6} & c_{4} & c_{5} & -c_{2} & c_{3} & -c_{4} & c_{1} \end{bmatrix}, \quad (C.11)$$

where the coefficients c_1 , c_2 , c_3 , c_4 , c_5 , and c_6 are obtained in terms of the Poisson's ratio ν , as follows.

$$c_{1} = \frac{1}{2} - \frac{2\nu}{3} \qquad c_{4} = \frac{1}{8}$$

$$c_{2} = \frac{1}{8} - \frac{\nu}{2} \qquad c_{5} = -\frac{1}{4} + \frac{\nu}{3}$$

$$c_{3} = \frac{\nu}{6} \qquad c_{6} = -\frac{1}{4} + \frac{\nu}{6}$$

3D elasticity

Similarly to the 2D problem, the variables consist of a vector field. However, in this case, there are three DOFs per node. In that sense, the shape functions matrix is defined as

$$\mathbf{N} = \begin{bmatrix} N_0, 0, 0, N_1, 0, 0, N_2, 0, 0, N_3, 0, 0, N_4, 0, 0, N_5, 0, 0, N_6, 0, 0, N_7, 0, 0\\ 0, N_0, 0, 0, N_1, 0, 0, N_2, 0, 0, N_3, 0, 0, N_4, 0, 0, N_5, 0, 0, N_6, 0, 0, N_7, 0\\ 0, 0, N_0, 0, 0, N_1, 0, 0, N_2, 0, 0, N_3, 0, 0, N_4, 0, 0, N_5, 0, 0, N_6, 0, 0, N_7 \end{bmatrix} .$$

$$\begin{pmatrix} N_0, 0, 0, 0, N_1, 0, 0, N_2, 0, 0, N_3, 0, 0, N_4, 0, 0, N_5, 0, 0, N_6, 0, 0, N_7 \\ 0, 0, N_0, 0, 0, N_1, 0, 0, N_2, 0, 0, N_3, 0, 0, N_4, 0, 0, N_5, 0, 0, N_6, 0, 0, N_7 \end{bmatrix} .$$

$$\begin{pmatrix} N_0, 0, 0, 0, N_1, 0, 0, N_2, 0, 0, N_3, 0, 0, N_4, 0, 0, N_5, 0, 0, N_6, 0, 0, N_7 \\ 0, 0, 0, 0, 0, 0, N_1, 0, 0, N_2, 0, 0, N_3, 0, 0, N_4, 0, 0, N_5, 0, 0, N_6, 0, 0, N_7 \end{bmatrix} .$$

The differential operator $[\nabla]$ is applied to **N** as in

$$[\nabla]\mathbf{N} = \begin{bmatrix} \partial/\partial x_1 & 0 & 0 \\ 0 & \partial/\partial x_2 & 0 \\ 0 & 0 & \partial/\partial x_3 \\ 0 & \partial/\partial x_3 & \partial/\partial x_2 \\ \partial/\partial x_3 & 0 & \partial/\partial x_1 \\ \partial/\partial x_2 & \partial/\partial x_1 & 0 \end{bmatrix} \mathbf{N} .$$
(C.13)

Then, the local stiffness matrix is given by

$$\mathbf{K}_e = \int_{\Omega_e} ([\nabla] \mathbf{N})^T \mathbf{C} ([\nabla] \mathbf{N}) \, d\Omega \ ,$$

which has its results presented in Equation C.14, in the following page. This matrix is also composed by coefficients (c_1 , c_2 , c_3 , c_4 , c_5 , c_6 , c_7 , c_8 , c_9 , and c_{10}) that are obtained in terms of ν , as shown below.

$$c_{1} = \frac{2}{9} - \frac{\nu}{3} \qquad c_{6} = -\frac{1}{48}$$

$$c_{2} = \frac{1}{24} - \frac{\nu}{6} \qquad c_{7} = \frac{1}{18} - \frac{\nu}{12}$$

$$c_{3} = -\frac{1}{18} \qquad c_{8} = \frac{1}{36} - \frac{\nu}{12}$$

$$c_{4} = -\frac{1}{24} \qquad c_{9} = \frac{1}{48} - \frac{\nu}{12}$$

$$c_{5} = -\frac{1}{36} \qquad c_{10} = \frac{5}{72} - \frac{\nu}{12}$$

1	-																							
	c_1	$-c_{4}$	c_4	C_3	$-c_{2}$	c_2	$-c_{10}$	c_4	c_9	$-c_{5}$	c_2	c_6	$-c_{5}$	$-c_6$	$-c_{2}$	$-c_{10}$	$-c_{9}$	$-c_{4}$	$-c_{7}$	c_6	$-c_{6}$	$-c_{8}$	c_9	$-c_{9}$
	$-c_4$	c_1	c_4	c_2	$-c_{5}$	c_6	c_4	$-c_{10}$	c_9	$-c_{2}$	c_3	c_2	$-c_6$	$-c_{5}$	$-c_{2}$	c_9	$-c_{8}$	$-c_{9}$	c_6	$-c_{7}$	$-c_{6}$	$-c_{9}$	$-c_{10}$	$-c_{4}$
	c_4	c_4	c_1	$-c_{2}$	c_6	$-c_{5}$	$-c_{9}$	$-c_{9}$	$-c_{8}$	c_6	$-c_{2}$	$-c_{5}$	c_2	c_2	c_3	$-c_{4}$	c_9	$-c_{10}$	$-c_{6}$	$-c_{6}$	$-c_{7}$	c_9	$-c_4$	$-c_{10}$
	c_3	C_2	$-c_{2}$	c_1	c_4	$-c_{4}$	$-c_{5}$	$-c_{2}$	$-c_{6}$	$-c_{10}$	$-c_{4}$	$-c_{9}$	$-c_{10}$	c_9	c_4	$-c_{5}$	c_6	c_2	$-c_{8}$	$-c_{9}$	c_9	$-c_{7}$	$-c_{6}$	c_6
	$-c_{2}$	$-c_{5}$	c_6	c_4	c_1	c_4	c_2	c_3	c_2	$-c_4$	$-c_{10}$	c_9	$-c_{9}$	$-c_{8}$	$-c_{9}$	c_6	$-c_{5}$	$-c_{2}$	c_9	$-c_{10}$	$-c_{4}$	$-c_{6}$	$-c_{7}$	$-c_{6}$
	c_2	c_6	$-c_{5}$	$-c_{4}$	c_4	c_1	$-c_{6}$	$-c_{2}$	$-c_{5}$	c_9	$-c_{9}$	$-c_{8}$	c_4	c_9	$-c_{10}$	$-c_{2}$	c_2	c_3	$-c_{9}$	$-c_{4}$	$-c_{10}$	c_6	$-c_{6}$	$-c_{7}$
	$-c_{10}$	c_4	$-c_{9}$	$-c_{5}$	c_2	$-c_{6}$	c_1	$-c_{4}$	$-c_{4}$	c_3	$-c_{2}$	$-c_{2}$	$-c_{7}$	c_6	c_6	$-c_{8}$	c_9	c_9	$-c_{5}$	$-c_{6}$	c_2	$-c_{10}$	$-c_{9}$	c_4
	c_4	$-c_{10}$	$-c_{9}$	$-c_{2}$	c_3	$-c_{2}$	$-c_{4}$	c_1	$-c_{4}$	c_2	$-c_{5}$	$-c_{6}$	c_6	$-c_{7}$	c_6	$-c_{9}$	$-c_{10}$	c_4	$-c_{6}$	$-c_{5}$	c_2	c_9	$-c_{8}$	c_9
	c_9	c_9	$-c_{8}$	$-c_{6}$	c_2	$-c_{5}$	$-c_{4}$	$-c_{4}$	c_1	c_2	$-c_{6}$	$-c_{5}$	c_6	c_6	$-c_{7}$	$-c_{9}$	c_4	$-c_{10}$	$-c_{2}$	$-c_{2}$	C_3	c_4	$-c_{9}$	$-c_{10}$
	$-c_{5}$	$-c_{2}$	c_6	$-c_{10}$	$-c_{4}$	c_9	c_3	c_2	c_2	c_1	c_4	c_4	$-c_{8}$	$-c_{9}$	$-c_{9}$	$-c_{7}$	$-c_{6}$	$-c_{6}$	$-c_{10}$	c_9	$-c_{4}$	$-c_{5}$	c_6	$-c_{2}$
	c_2	C_3	$-c_{2}$	$-c_{4}$	$-c_{10}$	$-c_{9}$	$-c_{2}$	$-c_{5}$	$-c_{6}$	c_4	c_1	$-c_{4}$	c_9	$-c_{10}$	c_4	$-c_{6}$	$-c_{7}$	c_6	$-c_{9}$	$-c_{8}$	c_9	c_6	$-c_{5}$	c_2
\mathbf{K} – E	c_6	c_2	$-c_{5}$	$-c_{9}$	c_9	$-c_{8}$	$-c_{2}$	$-c_{6}$	$-c_{5}$	c_4	$-c_{4}$	c_1	c_9	c_4	$-c_{10}$	$-c_{6}$	c_6	$-c_{7}$	$-c_{4}$	$-c_{9}$	$-c_{10}$	c_2	$-c_{2}$	c_3
$\mathbf{K}_e = \frac{1}{(1+\nu)(1-2\nu)}$	$-c_{5}$	$-c_{6}$	c_2	$-c_{10}$	$-c_{9}$	c_4	$-c_{7}$	c_6	c_6	$-c_{8}$	c_9	c_9	c_1	$-c_{4}$	$-c_{4}$	c_3	$-c_{2}$	$-c_{2}$	$-c_{10}$	c_4	$-c_{9}$	$-c_{5}$	c_2	$-c_{6}$
	$-c_{6}$	$-c_{5}$	c_2	c_9	$-c_{8}$	c_9	c_6	$-c_{7}$	c_6	$-c_{9}$	$-c_{10}$	c_4	$-c_{4}$	c_1	$-c_{4}$	c_2	$-c_{5}$	$-c_{6}$	c_4	$-c_{10}$	$-c_{9}$	$-c_{2}$	c_3	$-c_{2}$
	$-c_{2}$	$-c_{2}$	c_3	c_4	$-c_{9}$	$-c_{10}$	c_6	c_6	$-c_{7}$	$-c_{9}$	c_4	$-c_{10}$	$-c_{4}$	$-c_{4}$	c_1	c_2	$-c_{6}$	$-c_{5}$	c_9	c_9	$-c_{8}$	$-c_{6}$	c_2	$-c_{5}$
	$-c_{10}$	c_9	$-c_{4}$	$-c_{5}$	c_6	$-c_{2}$	$-c_{8}$	$-c_{9}$	$-c_{9}$	$-c_{7}$	$-c_{6}$	$-c_{6}$	C_3	c_2	c_2	c_1	c_4	c_4	$-c_{5}$	$-c_{2}$	c_6	$-c_{10}$	$-c_{4}$	c_9
	$-c_{9}$	$-c_{8}$	c_9	c_6	$-c_{5}$	c_2	c_9	$-c_{10}$	c_4	$-c_{6}$	$-c_{7}$	c_6	$-c_{2}$	$-c_{5}$	$-c_{6}$	c_4	c_1	$-c_{4}$	c_2	C_3	$-c_{2}$	$-c_{4}$	$-c_{10}$	$-c_{9}$
	$-c_{4}$	$-c_{9}$	$-c_{10}$	c_2	$-c_{2}$	c_3	c_9	c_4	$-c_{10}$	$-c_{6}$	c_6	$-c_{7}$	$-c_{2}$	$-c_{6}$	$-c_{5}$	c_4	$-c_{4}$	c_1	c_6	c_2	$-c_{5}$	$-c_{9}$	c_9	$-c_{8}$
	$-c_{7}$	c_6	$-c_{6}$	$-c_{8}$	c_9	$-c_{9}$	$-c_{5}$	$-c_{6}$	$-c_{2}$	$-c_{10}$	$-c_{9}$	$-c_{4}$	$-c_{10}$	c_4	c_9	$-c_{5}$	c_2	c_6	c_1	$-c_{4}$	c_4	C_3	$-c_{2}$	c_2
	c_6	$-c_{7}$	$-c_{6}$	$-c_{9}$	$-c_{10}$	$-c_{4}$	$-c_{6}$	$-c_{5}$	$-c_{2}$	C_9	$-c_{8}$	$-c_{9}$	c_4	$-c_{10}$	c_9	$-c_{2}$	c_3	c_2	$-c_{4}$	c_1	c_4	c_2	$-c_{5}$	c_6
	$-c_{6}$	$-c_{6}$	$-c_{7}$	C_9	$-c_{4}$	$-c_{10}$	c_2	c_2	c_3	$-c_{4}$	c_9	$-c_{10}$	$-c_{9}$	$-c_{9}$	$-c_{8}$	c_6	$-c_{2}$	$-c_{5}$	c_4	c_4	c_1	$-c_{2}$	c_6	$-c_{5}$
	$-c_{8}$	$-c_{9}$	c_9	$-c_{7}$	$-c_{6}$	c_6	$-c_{10}$	c_9	c_4	$-c_{5}$	c_6	C_2	$-c_{5}$	$-c_{2}$	$-c_{6}$	$-c_{10}$	$-c_{4}$	$-c_{9}$	c_3	c_2	$-c_{2}$	c_1	c_4	$-c_{4}$
	c_9	$-c_{10}$	$-c_{4}$	$-c_{6}$	$-c_{7}$	$-c_{6}$	$-c_{9}$	$-c_{8}$	$-c_{9}$	c_6	$-c_{5}$	$-c_{2}$	c_2	c_3	c_2	$-c_{4}$	$-c_{10}$	c_9	$-c_{2}$	$-c_{5}$	c_6	c_4	c_1	c_4
	$-c_{9}$	$-c_{4}$	$-c_{10}$	c_6	$-c_{6}$	$-c_{7}$	c_4	c_9	$-c_{10}$	$-c_{2}$	c_2	C_3	$-c_{6}$	$-c_{2}$	$-c_{5}$	c_9	$-c_{9}$	$-c_{8}$	c_2	c_6	$-c_{5}$	$-c_{4}$	c_4	c_1
										(C.14)														

.

108