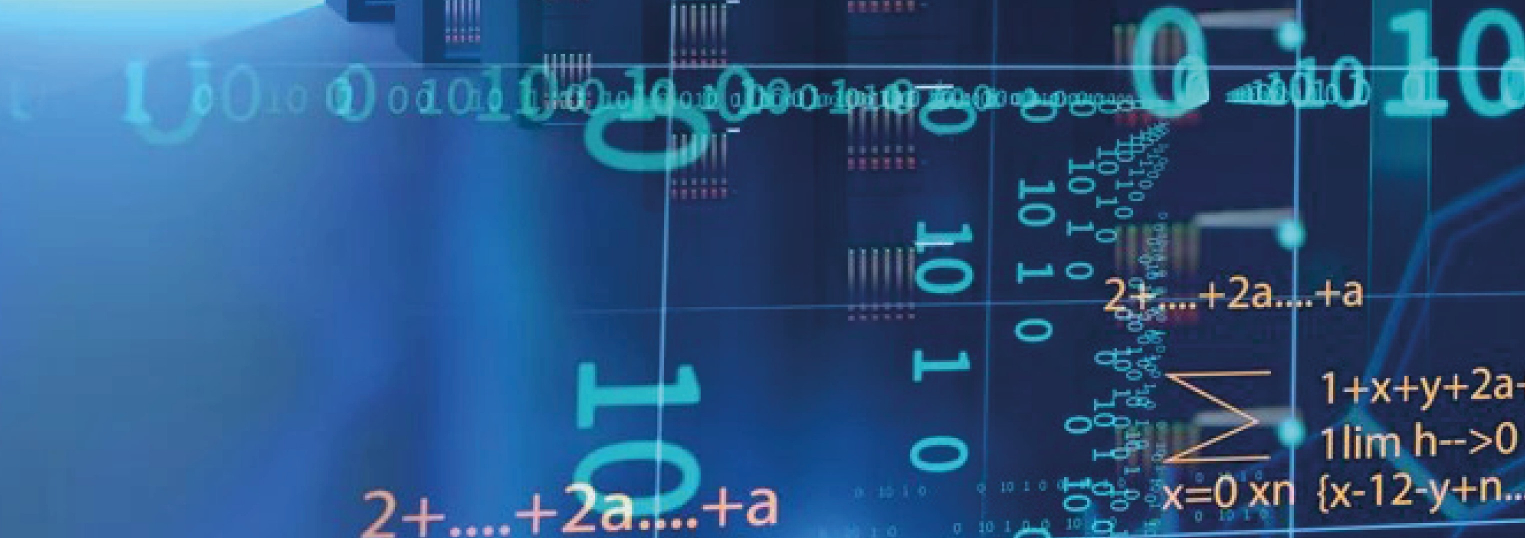
Master of Science Thesis

GPU-accelerated Finite Element Analysis for solid mechanics

S.M.N. van Paasen

July 4, 2023

Department of Aerospace Structures and Materials
Faculty of Aerospace Engineering
Delft University of Technology



GPU-accelerated Finite Element Analysis for solid mechanics

Thesis report

by

Sebastiaan van Paasen

to obtain the degree of Master of Science
at Delft University of Technology

Abstract

Finite Element Analysis (FEA) is one of the most established techniques to approximate the solution of complex nonlinear solid mechanics problems. Modeling realistic applications utilizing FEA is computationally intensive and it is often necessary to split the computing load among different processes running concurrently on a cluster of Central Processing Units (CPUs) to reduce the simulation runtime. Such parallel computing is traditionally performed by clusters of Central Processing Units (CPUs). However, the overhead associated with inter-CPU communication eventually deteriorates performance as the number of processes increases. At that point, Graphical Processing Units (GPUs) can be utilized to decrease the computation time even further. In fact, despite GPUs generally exhibit slower processing speed than CPUs, they can run thousands of threads concurrently, resulting overall in better performance. The objective of this thesis is to explore various GPU implementations aiming to determine how GPUs can act as co-accelerators alongside CPUs within a finite element calculation. To this end, the algorithmic steps of a solid mechanics finite element solver were analyzed to identify those that best lend themselves to GPU parallel computing. Consequently, two alternative GPU parallelization strategies were proposed, each of which was benchmarked on three different representative material models with the aim of comprehensively evaluating the ramifications of integrating a GPU-accelerated implementation. Results show a speed-up ranging between 31.5× and 37.7× for the evaluated material models. In addition, this thesis presents effective strategies to systematically optimize the CPU-GPU interaction in the context of FEA for solid mechanics, thus enabling highly resolved thirty million element calculations with tailored performance to heterogeneous multi-CPU and multi-GPU hardware.

Thesis committee:

Thesis supervisor:	Dr. Bianca Giovanardi
Examiner:	Dr. Matthias Möller
Chair:	Dr. ir. S. Giovani Pereira Castro
Project Duration:	September, 2022 - June, 2023
Date:	July 4, 2023
Student number:	4558944

Department of Aerospace Structures and Materials
Faculty of Aerospace Engineering
Delft University of Technology

Preface

Dear reader,

This report is the result of my master thesis and written in pursuit of the fulfillment to obtain the Master degree at Delft University of Technology. The report is the result of a study into algorithms accelerated by Graphical Processing Units that are applied to solid mechanics.

The study is performed at the Aerospace Structures and Materials department of the Aerospace Engineering Faculty at Delft University of Technology and explores how the simulation runtime of FEM simulations can be increased for aerospace applications. Theoretical technical knowledge of the Finite Element Method and solid mechanics are required, while prerequisite knowledge of Graphical Processing Units is not necessary as it is covered in the report itself.

During this study I was supervised by Bianca Giovanardi and Sai Kubair Kota. To both of you I owe many thanks for the great supervision and the constructive feedback I received to make the most of this project. I also want to thank my family and friends for the constant support during this project.

Delft, June 2023

Sebastiaan van Paasen

Summary

Phenomena driven by Partial Differential Equations (PDEs) encompass a range of examples such as heat transfer, electromagnetic potential and solid mechanics. When addressing problems associated with these phenomena, computing the exact solution is often unfeasible [1]. One of the strategies to approximate the solution of a PDE is Finite Element Analysis (FEA), which employs the Finite Element Method (FEM). The FEM is applied to various solid mechanics applications in the aerospace industry, including component design [2], topology optimization of components [3] and damage modeling [4], among others.

Modeling realistic applications utilizing the FEM is computationally intensive. A strategy to make simulation times attainable is to split the computing load among different processes running concurrently. Such parallel computing is traditionally performed by clusters of Central Processing Units (CPUs). A CPU prioritizes single-process performance and handling diverse computing tasks. However, when dealing with large-scale FEA models, the overhead associated with inter-CPU communication within a CPU cluster is detrimental for performance. In contrast to CPUs, Graphical Processing Units (GPUs) focus on massive parallelization and have received increased interest for their utilization in general-purpose parallel computations since the late 2000s. Although GPUs exhibit slower individual processing speed compared to CPUs, due to their slower clock speed and the requirement of memory synchronization, they can run thousands of processes concurrently. As a result, properly designed implementations enable GPUs to act as co-accelerators alongside CPUs [5].

In order for a GPU to act as co-accelerator alongside a CPU, the steps in the FEM that are feasible to delegate to the GPU must be identified and adjusted to enable a GPU implementation. The objective of this thesis is to explore various GPU implementations aiming to determine how GPUs can act as co-accelerators alongside CPUs within a finite element calculation. The study is conducted based on the main research question: "What speed up can be achieved in a FEA simulation applied to solid mechanics in an explicit time scheme by using massive parallelization on a heterogeneous hardware architecture?". To this end, the algorithmic steps of a solid mechanics finite element solver were analyzed to identify those that best lend themselves to GPU parallel computing. Consequently, two alternative GPU parallelization strategies were proposed, each of which was benchmarked on three different representative material models (linear elastic, Neo-Hookean, and orthotropic) with the aim of comprehensively evaluating the ramifications of integrating a GPU-accelerated implementation. An in-depth examination was conducted utilizing code profiling techniques to assess key performance metrics. These metrics served as quantitative indicators to gauge the efficacy and efficiency of the GPU-accelerated implementation.

Optimizing the GPU implementation was explored in two consecutive stages. First, remaining agnostic to the parallelization strategy and material model employed, ensuring that the achieved performance applies to all benchmarks in this *model-agnostic* GPU implementation. In addition, this thesis presents effective strategies to systematically optimize the CPU-GPU interaction in the context of FEA for solid mechanics. Subsequently, making use of the specific characteristics of each material model the optimal *model-specific* GPU implementations were obtained, exhibiting total computation speed-ups between 31.5× and 37.7×.

Exploring a multi-GPU setup with one CPU per GPU indicates that utilizing multiple GPUs is beneficial for the total simulation time as for sufficiently large problems, doubling the number of GPUs will result in half the original simulation time. It is observed that as the problem is partitioned over multiple GPUs, the problem size allocated to each GPU becomes smaller. As a result, the benefits from the GPU implementation are not fully extracted. This observation introduces a trade-off in terms of computational resources, involving the ratio of the number of CPUs and GPUs. Evaluating the described trade-off in terms of computational resources can leverage the inherent parallelism of GPUs more effectively and is left as a recommendation for future research.

Contents

Summary **ii**

List of Figures **iv**

List of Tables **iv**

1 Introduction **1**

I State of the art on GPU-accelerated Finite Element Analysis **4**

2 The power of Graphical Processing Units **5**
2.1 Computing process on a GPU . . . 5
2.2 Interaction between the CPU and GPU . . . 6
2.3 Processing on the GPU . . . 9
2.4 Event scheduling . . . 13
2.5 Conclusion . . . 14

3 A review of recent literature **15**
3.1 Speed-up strategies for GPU-accelerated FEA . . . 15
3.2 Current state of the art in solid mechanics . . . 19
3.3 High performance computing . . . 22
3.4 Conclusion . . . 23

II The Finite Element Method in solid mechanics **24**

4 Finite element formulation **25**
4.1 Review of the main FEM formulation steps . . . 25
4.2 Governing equations . . . 27
4.3 Semidiscretization . . . 29
4.4 Time integration . . . 31

5 Material models **33**
5.1 Linear material model . . . 33
5.2 Neo-Hookean material model . . . 33
5.3 Orthotropic material model . . . 34

6 Parallelization strategies **36**
6.1 Strain to stress . . . 36
6.2 Displacement to internal force . . . 36

III Accelerating Finite Element Analysis with GPUs **37**

7 Model-agnostic optimization **38**
7.1 GPU-accelerated implementations . . . 38
7.2 GPU performance metrics . . . 42
7.3 Optimizing CPU-GPU interaction . . . 43
7.4 Reducing idle GPU time . . . 47

7.5 Conclusion 49

8 Model-specific acceleration **51**
8.1 The linear elastic model 51
8.2 The Neo-Hookean model 58
8.3 The orthotropic model 62
8.4 Conclusion 67

9 Integration in a Finite Element Framework **70**
9.1 Testcase setup 70
9.2 Single-GPU integration 72
9.3 Extension to a Multi-GPU implementation 76
9.4 Conclusion 76

IV Closure **78**

10 Conclusion **79**

11 Recommendations for Future Work **82**

Bibliography **84**

A Graphical Processing Unit **85**
A.1 Specifications 85
A.2 Profiling 86

B Central Processing Unit **87**
B.1 Specifications 87
B.2 Profiling 87

Nomenclature

List of Abbreviations

API Application Programming Interface

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

DbD Degree of freedom by Degree of freedom

EbE Element by Element

FEA Finite Element Analysis

FEM Finite Element Method

FG-FEM Fixed Grid Finite Element Method

GPU Graphical Processing Unit

NbN Node by Node

PCG Preconditioned Conjugate Gradient

PCI-bus Peripheral Component Interconnect bus

PDE Partial Differential Equation

RAM Random Access Memory

SIMT Single Instruction Multiple Threads

SM Streaming Multiprocessor

TLED Total Lagrangian Explicit Dynamic

WS Warp scheduler

List of Symbols

$\mathbf{a}(\mathbf{X}, t)$ Acceleration of the body at the current time step

β Newmark time scheme stability parameter

$\mathbf{b}$ External body force tensor

$\mathbf{B}_0$ Partial derivatives of the nodal shape functions with respect to the initial configuration

c Critical material wave speed

Δt Stable time increment

δW Virtual work

δW^{ext} External virtual work component

δW^{int} Internal virtual work component

$\delta W^{kinetic}$ Kinetic virtual work component

ε	Linear strain tensor
$\mathbf{e_i}$	Principal axis tensor
$\mathbf{E}$	Green's strain tensor
$\mathbf{F}$	Deformation gradient
$\mathbf{f}^{ext}$	External nodal force vector
$\mathbf{f}^{int}$	Internal nodal force vector
$\mathbf{f}^{kin}$	Kinetic nodal force vector
γ	Newmark time scheme stability parameter
$\mathbf{\Gamma}_0$	Boundary of the body in the reference configuration
θ	Lamina angle with respect to the global coordinate frame
h	Relevant element length
$\mathbf{I}$	Identity tensor
J	Jacobian, volume ratio between initial and current configuration
J_ξ	Jacobian, ratio between natural and physical coordinates
λ	First Lamé material constant
μ	Second Lamé material constant
$\mathbf{n}_0$	Normal direction of the body in the reference configuration
N_I	Nodal shape function at node I
n_q	Number of quadrature points per element
Ω	Domain of the body under consideration
$\mathbf{P}$	First Piola-Kichhoff stress tensor
p	Shape function order
$\mathbf{Q_{global}}$	Stiffness matrix of a single lamina in the global coordinate frame
$\mathbf{Q_{local}}$	Stiffness matrix of a single lamina in its own coordinate frame
ρ	Material density
$\mathbf{S}$	Second Piola-Kirchhoff stress tensor
$\mathbf{t}_0$	Traction tensor in the reference configuration
$\phi(\mathbf{X}, t)$	Function describing the motion of the body through time
$\mathbf{u}(\mathbf{X}, t)$	Displacement of the body at the current time step
$\phi(\mathbf{C})$	Stored enenergy potential function
$\mathbf{v}(\mathbf{X}, t)$	Velocity of the body at the current time step
$\mathbf{X_0}$	Initial body configuration
$\mathbf{x}$	Current body configuration

List of Figures

1.1 Massive scalability of billion-element parallel simulations on CPU processors. 1

2.1 The main events present in the GPU computing process 5
2.2 The CPU-GPU interaction with unified global memory 7
2.3 Difference between paged and pinned global memory 7
2.4 The tasks present in the GPU processing event 9
2.5 Representation of the computing architecture of a GPU 9
2.6 A map of the different off- and on-chip memory forms of a GPU 10
2.7 A graphical representation of the streaming multiprocessor 11
2.8 The instruction process on a streaming multiprocessor 12
2.9 Coalescence in memory transactions 13
2.10 The computational process including hardware components 13
2.11 Task concurrency by utilizing streams 14

3.1 Fixed grid discretization over a physical domain 16
3.2 Different forms of assembly of the global system 19

4.1 A graphical representation of the four main steps in the FEM applied to solid mechanics . . 26

7.1 The strided memory layout 39
7.2 GPU baseline implementation event schedule and instruction schedule 39
7.3 The coalesced memory layout 39
7.4 Overload implementation event schedule and instruction schedule 40
7.5 Examination of the deformation gradient tensor 41
7.6 Component implementation event schedule and instruction schedule 41
7.7 Splitted implementation event schedule and instruction schedule 42
7.8 Performance of the implementations utilizing unified global memory 43
7.9 Performance of the implementations utilizing paged global memory 44
7.10 Performance of the implementations utilizing pinned global memory 44
7.11 Performance of the implementations utilizing mapped global memory 45
7.12 Stream implementation event schedule and instruction schedule 46
7.13 Performance of the splitted implementation utilizing streams 47
7.14 The tasks present in the GPU processing event 47
7.15 Reducing idle GPU time 48
7.16 Grid size sensitivity analysis 49

8.1 Grid size analysis for the linear elastic material in the strain to stress strategy 51
8.2 The event and instruction schedule for the diagonal splitted implementation 52
8.3 The event and instruction schedule for the stress symmetry implementation 53
8.4 The event and instruction schedule for the double symmetry implementation 53
8.5 Linear elastic material implementations in the strain to stress strategy 54
8.6 Grid size analysis for the linear elastic material in the displacement to internal force strategy 55
8.7 The event and instruction schedule for the reducing idle GPU time implementation 56
8.8 The event and instruction schedule for the reduced memory implementation 56
8.9 The event and instruction schedule for the concurrent reduced memory implementation . . 57
8.10 Linear elastic material implementations in the displacement to internal force strategy 58
8.11 Grid size analysis for the Neo-Hookean material in the strain to stress strategy 58
8.12 The event and instruction schedule for the reducing idle GPU time implementation. 59
8.13 The event and instruction schedule for the simplified implementation 60

8.14 NeoHookean material implementations in the displacement to internal force strategy 61
8.15 Grid size analysis for the Neo-Hookean material in the displacement to internal force strategy 61
8.16 NeoHookean material model implementations in the displacement to internal force strategy. 62
8.17 Grid size analysis for the orthotropic material in the strain to stress strategy 63
8.18 The event and instruction schedule for the reducing idle GPU time implementation. 63
8.19 The event and instruction schedule for the symmetry implementation. 64
8.20 The event and instruction schedule for the splitted implementation. 65
8.21 Orthotropic material implementations in the strain to stress strategy 66
8.22 Grid size analysis for the orthotropic material in the displacement to internal force strategy 66
8.23 Orthotropic material implementations in the displacement to internal force strategy 67

9.1 Subdivision of a tetrahedron element 70
9.2 A visualization of the test case in Paraview 71
9.3 Simulation performance for the linear elastic material 72
9.4 Simulation performance utilizing the Neo-Hookean material model 73
9.5 Simulation performance utilizing the orthotropic material model 74
9.6 The displacement for the simulation in Paraview. 75
9.7 Reaction force plot for the different implementations 75
9.8 Multi-GPU performance 76

A.1 Analysis tools for GPU profiling 86

List of Tables

7.1 Speed-up results for different types of global memory 45
7.2 Speed-up results for pinned global memory across different implementations 50

8.1 Speed-up results for different materials in the model-specific optimization 69

9.1 Evaluated mesh discretizations 71
9.2 The wall time for different mesh sizes utilizing the linear elastic material model 72
9.3 The constitutive time for different mesh sizes utilizing the linear elastic material model . . . 73
9.4 The wall time for different mesh sizes utilizing the Neo-Hookean material model 73
9.5 The constitutive time for different mesh sizes utilizing the Neo-Hookean material model . . 74
9.6 The wall time for different mesh sizes utilizing the orthotropic material model 74
9.7 The constitutive time for different mesh sizes utilizing the orthotropic material model . . . 75

1
Introduction

The mathematical modeling of numerous natural phenomena relies in part on Partial Differential Equations (PDEs). Phenomena driven by the understanding of PDEs encompass a range of examples such as heat transfer, electromagnetic potential, fluid flow and solid mechanics. When addressing problems associated with these phenomena, computing the exact solution is often unfeasible [1]. One of the strategies to approximate the exact solution of a PDE is Finite Element Analysis (FEA). FEA employs the Finite Element Method (FEM) that involves the subdivision of complex problems into smaller, more manageable components referred to as finite elements. These finite elements are created by partitioning the domain into a finite number of discrete elements.

A FEM implementation is computationally intensive when modeling realistic applications, for example the complex mechanical behaviors such as fracture and plasticity that occur in solid mechanics, or when multiple space- or time-scales are involved. A strategy to make simulations attainable in terms of computing time is parallel computing, which involves splitting up the computational workload in smaller parts that can run concurrently. Traditionally, Central Processing Units (CPUs) have been employed in clusters for parallel computing. CPUs prioritize single-process performance, versatility and handling diverse computing tasks. However, utilizing a CPU cluster necessitates inter-CPU communication for memory distribution and requires additional memory to support such communication. These factors contribute to a performance saturation as illustrated in Figure 1.1. Regardless of the problem size, there exists a threshold beyond which the addition of more CPUs does not yield an increase in performance.

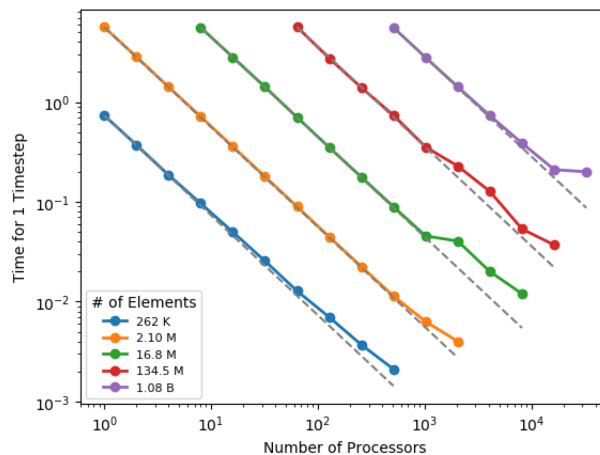


Figure 1.1: *Massive scalability of billion-element parallel simulations on up to 32,000 CPU processors. As the computational domain is split among several processors, the computational workload assigned to each processor becomes smaller. However, for each problem size the performance saturates, as it is evident the 1-billion element curve, indicating the existence of a threshold at which inter-CPU communication starts to outweigh the additional performance gained by increasing the number of processors.*

Since the latter part of the 2000s Graphical Processing Units (GPUs) received increased interest for their utilization in general-purpose parallel computations, driven by their improved speed and expanded memory capacity compared to traditional configurations [6]. GPUs are significantly slower in terms of single-process performance due to their lower clock speed and the necessity to synchronize memory between the CPU and GPU. In order for the GPU to perform computations, input data needs to be transferred towards the GPU prior to the computation. Subsequently, the results of the computation need to be transferred back

to the CPU. These memory transfers show to be detrimental in terms of performance. However, in contrast to CPUs, GPUs offer massive parallelization and can run thousands of processes concurrently. Thus, if the computation is properly designed and the performance decay due to memory synchronization can be overcome, GPUs can act as co-accelerators alongside CPUs, specifically dedicated to highly parallelize steps in the FEM suited for parallel computing [5]. Summarizing, CPUs can handle diverse computing tasks and excel in single process performance, while GPUs demonstrate exceptional performance in parallel processing, making them well-suited for computationally intensive tasks.

As a FEA problem grows in size, so do the corresponding memory requirements. For example, when considering a mesh with one billion elements, the data structures necessary for the computation would exceed the memory capacity of a single GPU. This highlights the significance of true scalability, which encompasses both time and memory, creating the possibility to solve problems of arbitrary size given sufficient computational resources. While a single GPU exhibits significant time scalability, its memory scalability is limited. On the other hand, employing multiple GPUs not only enables time scalability but also facilitates scalability in terms of memory capacity. However, utilizing both multiple CPUs and GPUs creates a trade-off in terms of computational resources. Up to a large number of processors, Figure 1.1 shows that adding more CPUs results in an increase in performance. Consequently, the portion of the problem size allocated to each CPU, and thus to each GPU, is reduced. If the reduction in problem size reaches the point where the problem size assigned to each GPU becomes too small to fully benefit from the GPU-accelerated implementation, adding more CPUs can actually result in a decay in performance way before the saturation related to communication overhead, which was described above, occurs. Additionally, the utilized hardware weighs in as well, since each CPU and GPU has its own sweet spot in terms of computational workload that results in the highest computational efficiency for that specific CPU or GPU. This highlights the importance of taking the problem size and utilized hardware into account in order to assess the suitable number of CPUs and GPUs to achieve the highest computational efficiency, even if it sacrifices the performance of a single component.

Over the past decade, scientists from different scientific disciplines have shown interest in leveraging the power of (multiple) GPUs to enhance the computational efficiency in FEA [7]. Particularly in fields such as computational electromagnetics [8] and surgical modeling [9], the employment of (multiple) GPUs has demonstrated the capability to approach near real-time solutions for large-scale FEA models [10]. In the aerospace industry, FEA is utilized in various solid mechanics applications, including component design [2], topology optimization of components [3] and damage modeling [4]. Near real-time solutions in these aerospace solid mechanics areas can offer significant benefits. For instance, this would enable the assessment of damage severity resulting from foreign object impact damage by analyzing it directly during flight. Additionally, they can aid engineers in determining the necessary reinforcement and its exact location to accelerate repair processes. Moreover, accelerating the FEA model execution time aligns with the right-first-time principle followed in aerospace structural design cycles, thereby significantly expediting the overall design process.

Examining the FEM in the context of solid mechanics and explicit dynamics, the evaluation of the material model was determined to be a suitable candidate for the GPU part of the calculation. Consequently, the objective of this thesis project is to explore GPU implementations of various material models, aiming to evaluate the performance gain achievable and bridge the research gap as GPU-accelerated algorithms have yet to be extensively used in solid mechanics. Based on the scope of this project, the following research question is formulated:

Main research question

What speed up can be achieved in a FEA simulation applied to solid mechanics in an explicit time scheme by using massive parallelization on a heterogeneous hardware architecture?

In order to answer the research question, several sub-research questions have been formulated. The sub-research questions elaborate on a part of the main research question and are formulated as follows:

Sub-research question A

How can the low-level computations in the FEM be evaluated as efficiently as possible?

Sub-research question B

To what extent is the total simulation time of a GPU-accelerated FEA model decreased compared to a single CPU FEA model?

Sub-research question C

What is the trade off, in terms of the number of CPUs and GPUs, in order to create the hardware setup that makes the best use of the resources available for a given problem?

This thesis is structured in parts, which are in turn organized into chapters. Following this introduction, the first part of the thesis covers the state of the art of GPU-accelerated Finite Element Analysis. The part contains an overview of what GPU computing entails for general purpose computations in chapter 2. Next, a concise description of recent literature for general purpose GPU computing for FEMs is given in chapter 3. Subsequently, a part on the FEM formulation for solid mechanics follows. The main steps of the employed finite element formulation are examined, and the governing equations are described in chapter 4. In the formulation, different material models will be implemented to assess the influence of material model complexity on computing performance. The equations for the different material models are presented in chapter 5. The part closes with chapter 6, proposing two parallelization strategies based on the described finite element formulation and examination of the main steps in the FEM for solid mechanics. Continuing, the results for the different parallelization strategies are presented in the next part. First, a model-agnostic optimization is performed in chapter 7, such that the obtained results hold for each material model. Subsequently, chapter 8 presents the model-specific optimization, shifting the focus to the various material models. The optimal implementation for each material model is integrated in a finite element framework in chapter 9. The final part concludes on the thesis with a conclusion in chapter 10 and recommendations for future work in chapter 11, reflecting on the GPU-accelerated implementations, the parallelization strategies that were employed, and the results that were obtained.

Part I

State of the art on GPU-accelerated Finite Element Analysis

2
The power of Graphical Processing Units

The first part of this thesis commences with a chapter that highlights different aspects of general purpose computations performed on a GPU. First the key events in the GPU computing process are explained in section 2.1. The reason to start with the main events is to familiarize the reader with the process first, before discussing the events themselves. Next, the different events are explored in section 2.2 and section 2.3, respectively. The scheduling of events is examined in section 2.4, followed by a brief conclusion on the presented content regarding general purpose computations performed on a GPU in section 2.5.

2.1 Computing process on a GPU

This section describes the essential events comprizing a computing process utilizing GPUs. The events are staged in chronological order, as depicted in Figure 2.1. It is important to emphasize that it is crucial for developers to adhere to the prescribed sequence of the events to ensure the accuracy of the obtained results. Neglecting any step within the process may engender erroneous outcomes and, in some cases, lead to code failure. Throughout this thesis a GPU developed by NVIDIA is utilized. NVIDIA offers the Compute Unified Device Architecture (CUDA) to execute code on a GPU. This Application Programming Interface (API) has several features that are highlighted throughout this chapter.

Figure 2.1: *The main events present in the GPU computing process. A GPU relies on code initialization on the CPU, followed by transferring input data towards the GPU. Utilizing the input data, the GPU performs the computation and subsequently transfers the results back to the CPU. Finally, the CPU continues with its tasks and finishes the process.*

Moving on, the shown chronological order in Figure 2.1 is explained. As mentioned in chapter 1, a GPU functions as a co-accelerator within a computing process and is reliant on the CPU for its operation. Consequently, the computing process invariably commences on the CPU. Initially, memory allocation for variables, both on the CPU and GPU side, takes place. The memory allocation is possibly followed by the execution of preliminary computations that do not lend themselves well for parallel execution on the GPU. In the context of the FEM application, this entails tasks such as mesh setup and determination of mesh connectivity, among others.

Subsequently, due to the discrete memory locations of CPUs and GPUs, particularly in the case of non-integrated GPUs, the input data for GPU computations necessitates transmission to the GPU. This transmission of data is represented by the “Transfer input data” event. Proceeding further, the computing operations are conducted on the GPU in the “Compute” event. A more in-depth discussion regarding this step will occur later in this chapter. Following the completion of the computations on the GPU, the results are transferred back to the CPU in the “Transfer results” event. The second and fourth events are further on referred to as interaction events, since they allow interaction between the CPU and GPU through memory transmission. The third event contains the actual processing of the input data and is therefore referred to as the processing event.

Once all interactions between the CPU and GPU have been concluded, the code reaches its final stages of execution on the CPU. Due to the asynchronous execution of the GPU, it is important to ensure synchronization between the CPU and GPU. This synchronization guarantees that the GPU's computations and memory operations are finished before the CPU proceeds with its own tasks. As a result, the CPU can safely continue with subsequent operations without the risk of accessing incomplete or inconsistent data.

2.2 Interaction between the CPU and GPU

This section aims to highlight the primary features of the second and fourth event of the computational process, i.e. the interaction events. To provide a comprehensive understanding, subsection 2.2.1 offers a concise explanation of the underlying hardware responsible for facilitating the memory transfer events. Continuing the discussion, subsection 2.2.2 illustrates the specific memory locations on the GPU from which the memory transfers originate. The insights of these subsections contribute to a comprehensive understanding of the interaction events and their significance in CPU-GPU data exchange.

2.2.1 PCI-bus

The memory transmission process during CPU-GPU interactions is carried out by a dedicated hardware component known as the Peripheral Component Interconnect (PCI) bus. The efficiency of the PCI bus directly impacts the volume of memory that can be successfully transferred between the CPU and GPU within a given time frame. It is crucial to acknowledge that the PCI-bus is inherently limited in its capacity to perform concurrent memory transfers, accommodating only two transfers simultaneously, with one in each direction. Consequently, it is impossible to transmit two distinct sets of data concurrently toward or from the GPU. Further exploration of the functionalities of the PCI-bus extends beyond the scope of this study. Therefore the PCI-bus is not further elaborated upon here.

2.2.2 Forms of off-chip memory

This subsection delves into off-chip memory and elaborates upon its role as the source of data being exchanged during interaction events. Off-chip memory is located outside the GPU-chip and encompasses three distinct forms: texture memory, constant memory, and global memory. Texture memory primarily finds application in visual-oriented tasks and is unsuited for FEM applications. Consequently, a detailed examination of texture memory is beyond the scope of this study and will not be elaborated upon further Constant memory, on the other hand, is a specialized cache with read-only permissions, characterized by a limited storage capacity. It is noteworthy that constant memory allows for the broadcasting of its contents to all processing events simultaneously. Consequently, scalar variables are automatically stored in constant memory to facilitate efficient data sharing among processing events. Further insights into constant memory are outside the scope of this study and will not be further explored here since read-only memory is unsuitable for the FEM application. Lastly, global memory represents the final form of off-chip memory. The global memory on the GPU encompasses four distinct types, each presenting its own set of advantages and disadvantages. These types are elaborated upon below:

- **Unified global memory**
  Unified or managed global memory is the most user-friendly global memory type for programmers. Unified memory eliminates the need for dedicated transmission instructions as it resides on both the CPU and GPU. Consequently, utilizing unified global memory is very comprehensible and an easy entry point for novice GPU-programmers.

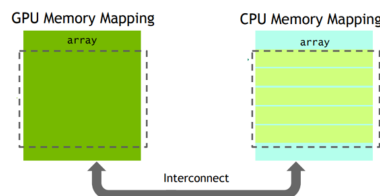


Figure 2.2: *Unified global memory allocates space for data on both CPU and GPU. The depicted array is partitioned in several blocks in CPU memory as shown on the right. Once the GPU starts to compute on a specific portion of the array, that portion is transferred from CPU to GPU. Figure credits: [11].*

The convenience comes at the cost of performance as illustrated below in Figure 2.2. Each single memory transfer introduces overhead cost, which makes unified global memory unsuited in case a large array needs to be transferred at once. Several methods exists to mitigate the performance degradation of unified global memory. For example, assuming the user knows which portion of memory shall be transferred next, the memory can be made ready to copy towards the GPU [11]. Yet, it should be noted that using unified global memory is never faster than manually copying data.

- **Paged and Pinned global memory**
  For both paged and pinned global memory separate variables are allocated for the CPU and GPU memory. Consequently, memory copies have to be made explicitly between the CPU and GPU. The difference between both memory forms lies in the allocation of the CPU memory. Both paged and pinned memory are allocated in the Random Access Memory (RAM) of the CPU. However, pageable memory can be transferred from RAM to a Hard Drive or Solid-State Drive in case the RAM becomes fully occupied. Pinned global memory can not be moved from RAM on the CPU side. As a result, the memory transfers between the CPU and GPU differ between paged and pinned memory as graphically depicted in Figure 2.3.

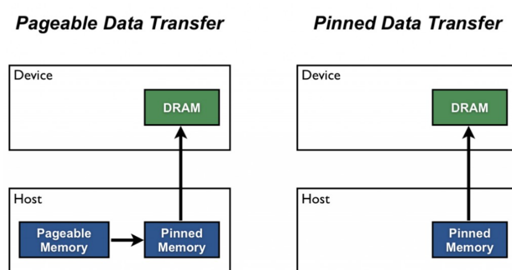


Figure 2.3: *To transfer paged memory toward the GPU, it must first be converted to pinned memory to assure the memory can not leave the RAM. In contrast, pinned memory is directly locked on allocation such that it can not leave the RAM. As a result, a pinned memory transfer bypasses the additional transfer step from pageable to pinned memory. Figure credits: [12].*

While pinned global memory provides better performance, it is limited in size as it occupies a portion of the CPU's RAM. As the RAM of a CPU is limited in size, allocating too much pinned memory can result in a shortage of pinned memory for the CPU which results in code failure. Paged global memory, although slower, is only restricted in volume by the HD or SSD size.

- **Mapped global memory**
  In contrast to paged and pinned global memory, mapped global memory does not require explicit memory copies and is therefore referred to as zero-copy memory. Mapped global memory is allocated as pinned memory on the CPU, i.e. it can not leave the RAM of the CPU. Subsequently, a variable that is available on the GPU is requested, which directs the GPU to the location of CPU memory that was allocated to the CPU variable. This way, the memory is available on both the CPU and the GPU. Consequently, the results of GPU computations are directly stored on the CPU when utilizing mapped global memory, hence the term "zero-copy memory". In setups with integrated GPUs, where the CPU and GPU share the same memory space, mapped global memory can yield performance gains.

Global memory, regardless of its type, is accessible across all processing events on the GPU and offers extensive storage capacity, along with read and write capabilities. It serves as the primary means for memory transfers between the CPU and GPU.

2.3 Processing on the GPU

In between the two interaction events, the actual processing takes place by means of the processing event. All subsequent tasks discussed in this section are exclusively carried out in the processing event of Figure 2.1. Focussing on the processing event, Figure 2.4 illustrates the tasks present in the processing event. Together, the three illustrated tasks form a single process and a GPU is capable of running thousands of these processes in parallel. Each single process gets its own unique thread assigned, which is responsible for carrying out the tasks present in the process.

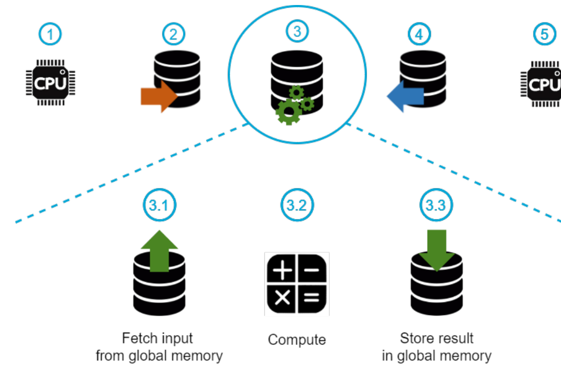


Figure 2.4: *The tasks present in the GPU processing event. First the the thread fetches the input data from the global memory. From the input data the thread computes its result which is subsequently stored in the global memory of the GPU. The tasks are executed in sequential order. Note that this is only a simple example for illustrative purposes.*

The hierarchical order in which threads are assigned is discussed in subsection 2.3.1. Prior to the actual compute task, threads first fetch the input data from the off-chip global memory and subsequently store the input data locally in the on-chip memory. The various types of on-chip memory are discussed in subsection 2.3.2. Lastly, an important note is made regarding the interaction between on- and off-chip memory in the processing event in subsection 2.3.4.

2.3.1 Computing architecture

The computing architecture of a GPU is visualized in Figure 2.5. As described before, each thread is responsible for carrying out the tasks present in the process it is assigned to. Threads within a GPU operate according to the Single Instruction Multiple Threads (SIMT) principle [13]. According to this principle, every thread executes the same code while operating on distinct values of data. Consequently, rather than utilizing a conventional "for loop" construct, all individual iterations of the loop are executed in parallel.

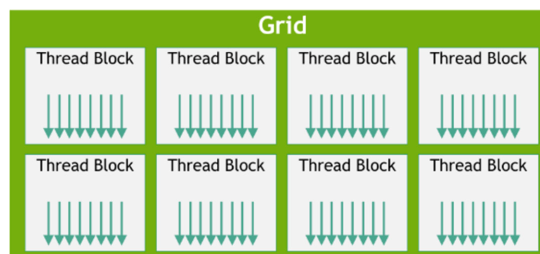


Figure 2.5: *Representation of the computing architecture of a GPU. Multiple threads collectively form a thread block. Several thread blocks combine to form a grid. The maximum number of thread blocks within a grid is GPU-specific. The grid serves as the execution unit for code execution and employs as many parallel executions of a function as there are threads in the grid. Figure credits: [13].*

Multiple threads collectively form a thread block. All threads within a individual thread block are partitioned in groups of 32 threads called warps. Regardless of the number of threads within a block, the GPU always allocates the smallest number of warps larger than the number of threads requested by the user. Consequently, utilizing a thread block size that is a multiple of 32 is recommended [14]. Several thread blocks combine to form a grid. The maximum number of thread blocks within a grid is GPU-specific. The grid serves as the execution unit for code execution. A function executed on the GPU is called a *kernel* and employs as many parallel executions of the function as there are threads in the grid.

Reflecting on the SIMT principle, the grid configuration holds information regarding the number of thread blocks and size of each thread block. Moreover, each individual thread within a thread block is assigned a distinct index within its respective thread block. By combining the information regarding the grid structure and the thread's index within its block, the possibility rises to give each thread a unique index. Consequently, each individual thread can operate on a distinct value from the same array.

The rationale behind utilizing multiple thread blocks rather than a single grid with only one thread block lies in the potential for cooperative behavior among threads within a thread block. Cooperation between threads necessitates the utilization of shared memory, which is explained in subsection 2.3.2. Leveraging thread cooperation can enhance computational speed and overall performance.

2.3.2 On-chip memory

The primary distinction of on-chip memory, in contrast to off-chip memory, is the fact that it resides within the GPU chip itself. On-chip memory offers the advantage of low latencies for data fetching and storage operations due to its proximity to the processing units. In fact, a drawback common to all forms of off-chip memory is the location outside the GPU chip, leading to significant latency when fetching and storing data in off-chip memory. Yet, on-chip memory can not communicate with the CPU and can therefore only be used to accelerate the processing event on the GPU itself. A common strategy to optimize the memory interaction of GPU computational processes is to have a single fetch and storage operation between on- and off-chip memory and use on-chip memory as frequently as possible during the processing event. The different forms of on-chip memory are visually represented inside the yellow boxes in Figure 2.6. The previously discussed forms of off-chip memory are also displayed at the bottom of the figure for reference.

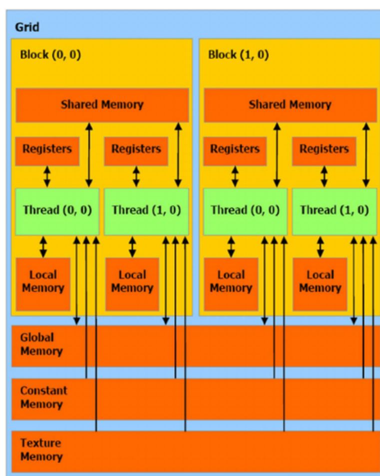


Figure 2.6: *A map of the different off- and on-chip memory forms of a GPU. The different forms of off-chip memory are shown by the large rectangular orange bars. Above it, closer to the threads and depicted inside the thread blocks, the different types of on-chip memory are shown. Figure credits: [15].*

Both registers and local memory within the GPU are exclusively accessible by individual threads. As a consequence, registers are automatically employed to store variables initialized within GPU code. Once a threads's registers are filled, any additional variables are stored in the local memory. Although local memory is slower than registers, local memory offers a larger storage capacity and remains considerably faster than the off-chip global memory.

A limitation of both registers and local memory is that it is exclusively accessible by individual threads. In contrast, shared memory is available to all threads within a thread block. Therefore, shared memory offers a larger accessibility, although this number of threads remains considerably small. Moreover, shared memory is limited in size, permitting only sparing usage. To attain many concurrent accesses, shared memory is partitioned in equally sized portions called memory banks [16]. All memory banks can be accessed simultaneously. Nevertheless, if threads try to access the same memory bank, bank conflicts arise. In this case, the accesses to the same memory bank are serialized, decreasing performance. To minimize the amount of bank conflicts, it is crucial to have knowledge of the memory bank size and size of variables stored inside the memory banks [16].

2.3.3 Computation execution

To facilitate the execution of tasks within the processing event, the allocated computation architecture is assigned to the streaming multiprocessors (SMs) of the GPU. The SMs are a key component of the GPU as they are responsible for thread scheduling and execution. The architecture of a SM is visually represented in Figure 2.7.

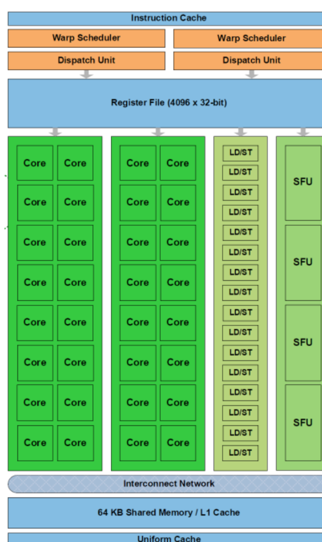


Figure 2.7: *A graphical representation of the streaming multiprocessor. When a function is executed, the thread blocks of the grid are evenly partitioned among the available SMs. Each SM has its own on-chip memory, represented by the shared memory/L1 cache at the bottom, and texture memory, represented by the Uniform Cache. Global and constant off-chip memory is accessed via the Interconnect Network which connects different SMs to the off-chip memory of the GPU. The thread blocks are further subdivided in warps that are distributed over the Warp Schedulers. When the Warp Scheduler issues a warp to execute, the Dispatch Unit take care of distributing the threads of the warp over the three different execution units. Cores perform low level arithmetic operations, Load and Storage units (LD/ST) take care of memory operations and Special Function Units (SFUs) are capable of handling special operations such as* sin, cos *and* exp. *Important to note that the amount of cores, LD/ST's and SFUs is GPU specific and serves here only to illustrate how the SM works. Figure credits: [17].*

The thread blocks of the grid are evenly partitioned among the available SMs, while further subdivision is performed to subdivide the thread blocks in warps that are assigned to Warp Schedulers. During the lifetime of the warp it stays on the assigned Warp Scheduler (WS). When the WS issues a warp to execute, the Dispatch Unit take care of distributing the threads of the warp over the different execution units. At each clock cycle, the WS can issue one instruction to a single warp [13]. Consequently, despite initializing an entire grid for code execution, the number of threads instructed to actively perform a task within a single clock cycle is significantly smaller than the total grid size. To illustrate this instruction process, a visual representation is presented below in Figure 2.8.

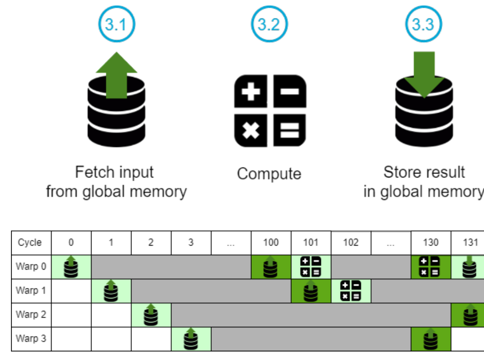


Figure 2.8: The instruction process on a streaming multiprocessor. Warp 0 *is instructed in cycle* 0*, indicated by the light-green background color, and returns in cycle* 100 *with the fetched input data, indicated by the dark-green color. The SM observes that warp* 0 *is available and proceeds to issue a new instruction to warp* 0*. Meanwhile, as warp 0 is engaged in the computation step, an increasing number of warps finalize the fetching of their respective input values. These warps are promptly instructed to commence their computations as well. This iterative process continues until all warps from all blocks conclude their respective actions. Note that the number of cycles a warp is executing a task can differ between warps for the same task. Finally, a similar series of steps is undertaken to store the resulting values back into the primary memory of the GPU. The white space present for warp* 3 *in cycle* 131 *is undesired. Although the warp is ready to be instructed, the SM is busy instructing warp* 2*. Consequently, warp* 3 *is stalled.*

Throughout the code execution process, the WS continuously observes the status of its assigned warps, identifying those that have completed their designated tasks and are ready to be re-instructed. The capability of the WS to issue an instruction to a warp in every clock cycle results in a substantial throughput of instructions. However, as highlighted in Figure 2.8, the number of clock cycles it takes to execute a task depends on the complexity of the task. Fetching and storing data in off-chip memory incur a substantial latency compared to the computational operations themselves. Moreover, if each thread utilizes Special Function units in its task and only 4 of these are present, referring to Figure 2.7, then 8 cycles are necessary to process a single warp resulting in a small latency for the warp. A similar reasoning can be followed for the amount of shared memory or number of registers [13]. Therefore, not only the complexity, but also the amount of available resources on the SM plays an important role. In case of loading or storing data in off-chip memory there are more load/storage units available. However, fetching or storing of data by threads in the off-chip memory of the GPU requires a significantly larger number of clock cycles. Consequently, these data transfer tasks result in a larger latency.

The conclusion of this subsection is the existence of a load balance problem in which the WS requires a considerable number of available warps at all times to assure that instructions can be issued continuously such that the GPU remains active and no idle time is present in the processing event.

2.3.4 Memory coalescence

An essential note regarding the interaction between on-chip and off-chip memory in GPUs is memory coalescence. In recent GPU architectures, memory transfers are subdivided into 32-byte memory transactions, regardless of the overall size of the memory transaction [14]. A memory transaction is deemed to be coalesced if all accessed values in the off-chip memory are contiguous, leading to the minimum number of memory transactions necessary to complete the overall memory transfer.

To illustrate the concept, consider Figure 2.9. If all 32 threads within a warp are tasked with fetching a 4-byte float value, the operation can be executed using four coalesced 32-byte memory transactions. In cases where not all fetched float values are required, the entire transaction is still performed. This holds true when multiple threads access the same float value or when a number of threads does not participate in the access. Additionally, it is not necessary for the threads within a warp to access the floats in contiguous order, as long as all requested values from memory are contiguous.

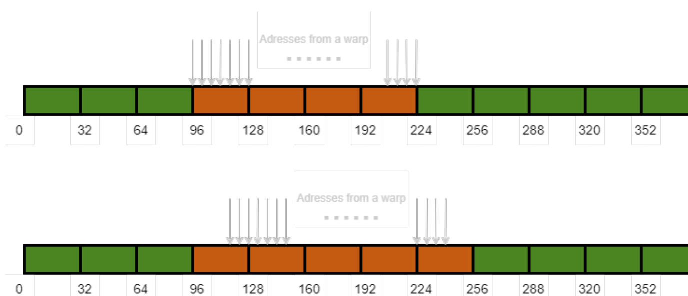


Figure 2.9: *If all 32 threads within a warp are tasked with fetching a 4-byte float value, the operation can be executed using four coalesced 32-byte memory transactions as shown in the top. However, if the requested memory is not aligned, an offset is present as illustrated in the bottom case. This implies that the first requested value can reside at any address between 96 and 128, resulting in five 32-byte memory requests due to the final address falling between addresses 224 and 256.*

However, if the requested memory is not aligned, an offset is present. The offset's size directly influences the extent of the resulting performance penalty. A relatively small offset of 2 addresses leads to a drastic reduction of over half the performance, highlighting the significant effect of offsets on the memory transfer efficiency [14].

2.4 Event scheduling

Throughout the preceding sections, the various events occurring were executed serially. The resulting process, together with the responsible hardware components are illustrated in Figure 2.10.

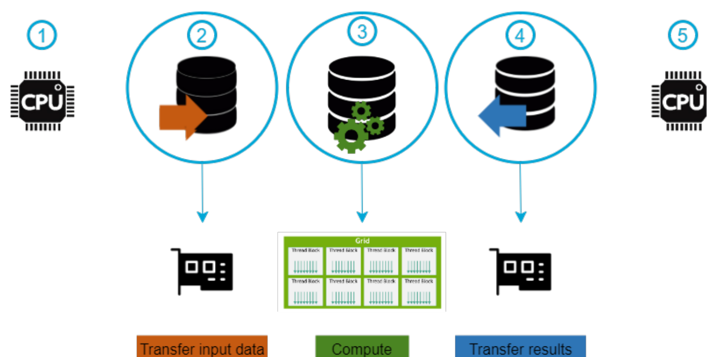


Figure 2.10: *A graphical representation of the computational process, together with the hardware components of the GPU responsible for each event. Both memory transfers are performed by the PCI-bus, while the processing is distributed over the threads in the created grid. The colored bars on the bottom will be used throughout the remainder of this report to express the occurrence of the specific event.*

To facilitate an even higher level of concurrency, i.e. executing the three events involving GPU interaction concurrently, e.g. sending input data while also computing and receiving results, NVIDIA GPUs offer the stream functionality. The utilization of streams facilitates the concurrent execution of the two interaction events, namely the memory transfers between the CPU and GPU, and the processing event on the GPU itself. From a conceptual standpoint, a stream can be perceived as a conduit within the GPU that serves as a channel to receive instructions originating from the CPU. Thus, it acts as a mechanism for asynchronous communication between the CPU and GPU.

In the computational process discussed thus far, all events have occurred in a sequential manner, forming a linear stream of operations, often referred to as the "head stream" as illustrated on the left in Figure 2.11. However, it is important to recognize that not all input data exhibits interdependencies. Consequently, the GPU remains idle, awaiting the availability of the complete input data before commencing the processing event. This scenario presents an opportunity for optimization, as the GPU could potentially initiate the

processing event as soon as a portion of the input data is present, rather than waiting for the entirety of the data to arrive. An exemplification of the streaming functionality is presented in the middle of Figure 2.11.

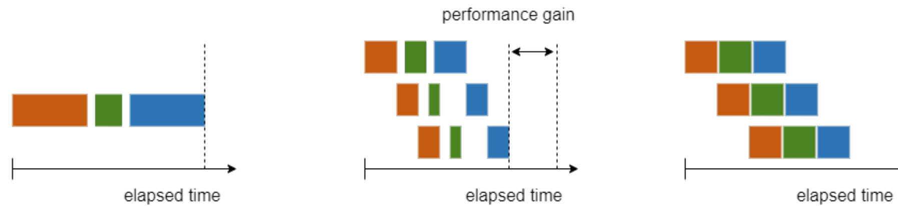


Figure 2.11: *Task concurrency by utilizing streams. On the left a general computational process is shown without using streams. In this case, the transfer of input data, the computations andthe transferring of the results occur one after the other. In the middle, three streams are used to split up the workload over different streams, thus decreasing the total process time. Note that the space in between the different tasks can not be removed since the PCI-bus allows only two memory transfers occurring simultaneously. As a result, "white space" is present, indicating idle time for the GPU. On the right the ideal case is represented. The computation takes up roughly the time it takes to compute results, making this the most efficient case.*

This implementation deviates from the conventional approach of transmitting all input data at once, opting instead to transfer only a portion of the data to the GPU. Consequently, computations are performed exclusively on the transferred segment, and the corresponding outcomes are returned to the CPU. While computations are underway for the first segment, the input data for the subsequent portion is already being dispatched to the GPU using a different stream. This mechanism effectively conceals the latency associated with memory transfer between the CPU and GPU. Important to note that higher concurrency can not be achieved since the PCI-bus allows only two memory transfers occurring simultaneously. As a result, "white space" is present, indicating idle time for the GPU. Furthermore, it is crucial to acknowledge that the feasibility of utilizing streams hinges upon the ability to disentangle the input data. If the interdependencies among the data render it inseparable, the entirety of the input data must still be transmitted to the GPU before any computation can commence, thus dissolving the benefits and any performance enhancement of the utilization of streams.

It is worth noting that achieving optimal concurrency can be contingent upon the execution times of individual kernels. The amount of kernels executed in parallel is solely limited by the GPU resources. In cases where there is significant variation in the duration of kernels, it may be advantageous to further divide the computations into multiple kernels that can be distributed across multiple streams. By doing so, the potential for enhanced concurrency and improved performance can be realized.

In addition to the aforementioned limitations, it is crucial to maintain a comprehensive understanding of the sequential order in which specific instructions need to be executed. Instructions issued to a single stream are executed in the order they are issued in. However, among different streams there is no prescribed ordering in the execution of instructions. Consequently, it is crucial to analyze interdependencies between streams and monitor those to assure accurate results. By carefully managing the issuing order of instructions and stream organization, both concurrency and correctness of results can be effectively balanced, thereby optimizing the overall performance of GPU computing processes.

2.5 Conclusion

The aim of the current chapter was to familiarize the reader with the computing process that involves GPU computing. The main events present in the computing process were presented in section 2.1, followed by an elaborate description of both the responsible hardware components and the steps present in said events in section 2.2 and section 2.3. Finally, the scheduling of events was highlighted in section 2.4, indicating that the overlap of interaction and processing events can enhance the computing performance considerably. However, a well-defined synchronization mechanism must be established, such that the overall integrity and correctness of the computing process are upheld.

3
A review of recent literature

Following the chapter on general purpose computations on a GPU, the aim of this chapter is to acquaint the reader with the utilization of GPUs for accelerating FEMs. The primary focus is to highlight the impact of using GPUs on both the algorithmic implementation and resulting performance. It is important to note that this chapter is an abbreviated version of the literature review that was conducted by the author before this thesis. For a complete overview of the literature review readers are referred to [18]. This chapter proceeds with discussing speed-up strategies employed in existing literature to enhance the performance of FEMs utilizing GPUs in section 3.1. Subsequently, a detailed investigation into GPU-accelerated FEMs applied in solid mechanics is presented in section 3.2. The added complexity associated with high performance computing utilizing GPUs is discussed in section 3.3 and the literature study concludes in section 3.4.

3.1 Speed-up strategies for GPU-accelerated FEA

This section presents the findings of speed-up strategies employed in existing literature to speed up FEA through the utilization of GPUs. The discussion begins with an exploration of strategies that have a localized influence, only affecting a single step in the full algorithms. Following that, strategies with a broader impact on the algorithm to which they are applied are addressed.

3.1.1 Localized speed-up strategies

Strategies that have a localized impact on the FEM are focused on either the first or last step, i.e. the discretization or solving step. Both steps are discussed sequentially. Since discretization is the first step, it affects subsequent steps in terms of performance and memory demand. A distinction between two types of finite element meshes is made:

- **Structured meshes**
  Structured meshes have elements that are equal in size and are ordered regularly and the node connectivity is predicted rather easily.
- **Unstructured meshes**
  In an unstructured mesh, the elements are assigned in an order that suits the geometry of the domain best, which results in elements of unequal size. The node connectivity can not be predicted for unstructured meshes. As a result, additional memory is required to store the nodal connectivity. Additionally, fetching values of a quantity through the nodal connectivity is more expensive since an additional value, the index from the node connectivity, must be fetched from memory.

Structured grid meshes are interesting due to their relatively low memory demand. Extending this further, a fixed grid FEM method (FG-FEM) was created that superimposes a fixed grid over the physical domain, neglecting the physical boundaries that are present [19]. Figure 3.1 shows the procedure as the domain in (a) is fixed-grid discretized in (b). In terms of memory, a FG-FEM simply requires preserving the elements at the boundary, one inner element, and one outside element which results in a very small memory demand. Although this method has existed for a long time already, more recent works by Jesús Martínez-Frutos and David Herrero-Pérez show that a speed-up of 8× to 20× can be achieved, depending on the hardware architecture that is used [20].

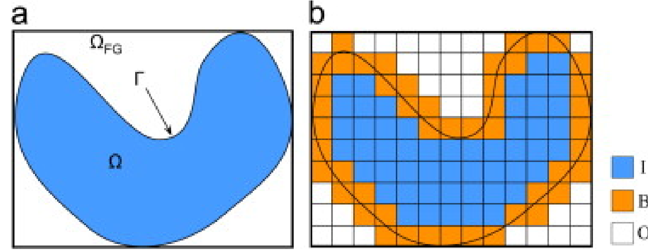


Figure 3.1: *Fixed grid discretization over a physical domain. The body is discretized in equal fixed grid squares, neglecting the physical boundary of the body. As all elements inside of the body are of the same size, only one element needs to be stored, which holds for all elements outside the body as well. All elements on the boundary, highlighted in orange, must be saved since all elements are different. Figure credits: [20].*

A drawback is that the FG-FEM can only be applied to linear elastic constitutive models at the moment. How to extend the fixed grid approach to nonlinear constitutive models is not clear as this extension would require saving all elements again, which is currently the main benefit of the method. However, it can be concluded that the fixed grid approach can handle significantly larger models as the memory demand is very low.

Unstructured meshes offer a more broad application to irregular shapes. This is the main reason why they are used more often in the found literature. A disadvantage is that, as the element size varies, so do the geometric properties of each element. As a result, the material response differs per element as well. The requirement of saving all geometric properties per element significantly increases the memory demand. On top of that, there is no regular sparsity pattern in the global system for unstructured meshes. While the irregular pattern is an effect resulting from a choice in the discretization step, i.e. a structured or unstructured mesh, it can be dealt with in the solving step by utilizing preconditioners, a specific solver or a feasible storage format for the global matrix [21]. Preconditioners and solvers are already widely used for parallel computing on CPUs and are just as relevant for GPUs since they aid in decreasing the computing time on GPUs as well. Feasible storage formats can be considered even more relevant for a GPU as the memory on a GPU is limited and should therefore be used as efficiently as possible.

Preconditioners can be used to adjust the linear system such that the solution is preserved while the eigenvalues of the solution are brought closer to each other. As a result, the system becomes easier to solve. Direct preconditioners exist, such as the Jacobi and block-Jacobi preconditioners. Directly derived from the global matrix, direct preconditioners are easy to construct and parallelize but show poor convergence rates [21]. The block-Jacobi preconditioner is well suited for assembly-free systems as it can be used on an element-level scale [1]. In the iterative domain, incomplete factorized preconditioners, such as ILU, and sparse approximate inverse preconditioners are commonly used. The first shows lesser performance when increasing the number of thread blocks as the number of couplings between thread blocks increases as more thread blocks are used [22]. The latter utilizes sparse matrix-vector products, which are feasible to compute on GPUs. However, results of sparse approximate inverse preconditioners show that they are less robust [21].

In terms of solvers, iterative solvers are common due to the large sparse systems that arise in FEM [21]. Most often the Conjugate Gradient method is utilized. While the Conjugate Gradient method is mostly used for positive definite matrices, it can be expanded to non-symmetric matrices [23] and to nonlinear problems as well [24]. Another method is the dynamic relaxation algorithm, in which a convergence criterion is based on the current iteration of the desired solution variable (e.g. the displacement in solid mechanics). The dynamic relaxation algorithm is very flexible as the order of the steps in the calculation is flexible to suit the hardware setup and application [5].

The sparse matrix-vector product is the most often occurring computation for globally assembled systems. That is why the sparse matrix-vector product received increased interest in the literature. The broad attention to sparse matrix-vector products has led to the development of different library packages that can be used to optimize the sparse matrix-vector product computation such as MAGMA [25] and more

recently HyMV [26]. Specialized storage formats aid in decreasing the required memory for large matrices. Even though additional computations must be executed [27], the lower memory requirement results in a significant decrease in computing time which makes simulations more feasible. Examples of storage formats include the ELLPACK, COO and CSR formats. Another strategy is to utilize multiple storage formats to reduce the number of variables that require storage space, which can result in significant speed-up results [8]. Different options are explained by Vazquez that show applying the most feasible storage format for the hardware resources available can lead to a speed-up of 30× [28].

The focus of the achieved speed-up will be on the computational steps before solving the system. This is because the solving step is often handed to advanced software libraries that already do an excellent job in terms of computational performance [21, 29]. However, the knowledge of preconditioners, solvers and storage formats is included here to serve as background information on the solving step in general.

3.1.2 Global speed-up strategies

A lot of strategies in literature affect the FEM on a more global scale in their approach. In most cases the computation of elemental contributions and assembly are the affected steps as both steps operate on the same data. Both the computation of elemental contributions and assembly have a considerable memory demand. Since a GPU offers several memory types, there are many different speed-up strategies. The intensive use of memory is also the reason why broad attention is given to speeding up the computation of elemental contributions and assembly step in literature.

The computation of elemental contributions always involves numerical integration of the shape functions used to model the element geometry. As the nature of this step is on an element level, parallelization is possible on multiple levels and in different ways. As a simple example take a relation between stress and strain, which can be evaluated using Equation 3.1.

$$\int_{\Omega_e} \mathbf{B}^T \mathbb{C} \mathbf{B} d\Omega_e \tag{3.1}$$

Here $\mathbf{B}$ represents the shape function derivatives, $\mathbb{C}$ the tangent tensor to the constitutive stress-strain relation and Ω_e the element domain. To parallelize the calculation in Equation 3.1, one can loop twice over the shape function derivatives, i.e. once over the quadrature points, and once over the elements for the material properties. Kruzel and Banaś concluded that the computation of elemental contributions is a straightforward procedure for low-order polynomial shape functions. However, it can become the dominating step in terms of performance in case the approximation order becomes five or higher [30]. The main conclusion from Kruzel and Banaś was that, from a performance point of view, it is best to compute shape function derivatives beforehand and store them in global memory on the GPU compared to computing them in the shared memory of the GPU. While global memory is much slower, it is also significantly larger than shared memory. Komatitsch *et al.* adopted a similar approach by storing element data in the global memory [31]. The time penalty taken was outweighed by the higher element order that was realized since shared memory did not offer enough space to store all shape function derivatives. Still, a speed-up of 25× was obtained compared to a single CPU implementation in the field of seismic wave propagation.

Executing element computations in shared memory is much faster and does not imply using a single thread per element. Maciol proposed a technique allowing multiple threads to work on a single element [32]. Threads in the same warp can access the same shared memory to store partial computations and accelerate the computation. Recently, Kiran used the same principle by assigning a full warp to compute a single element [33]. By assigning a warp to a single element, it is assured that enough shared memory space is available for all elemental computations. Different implementations were tested and the largest speed-up achieved was between 6.73× and 8.21×. Although shared memory was used in the computation of elemental contributions step in the approaches by Maciol and Kiran, global memory is required for the assembly step as shared memory does not offer the space necessary for this step. Dziekonski *et al.* exploited the use of global memory with a streaming functionality [8]. The slow global memory transactions were overlapped by the element computations that are performed in shared memory. The strategy of Dziekonski led to a speed-up of 81× compared to a single CPU in the field of computational electromagnetics.

A radically different approach is to solve the memory issue by dividing the computation over two different kernels. Kiran showed a two-kernel approach as one of the possible implementations of assigning a warp per element [33]. The first kernel computes geometrical parameters and the second kernel computes the element stiffness matrices and accumulation into the global matrix. In the end, the two-kernel implementation was deemed the slowest since more kernels need to be launched. Dziekonski *et al.* used the two-kernel approach differently by using it only in the assembly part [34].

From the previously discussed literature it can be concluded that the decision between partly using local/shared and global memory or global memory only is one that comes forward very often in literature. Furthermore, the bottleneck for using shared memory is its small size, while the high latency is the bottleneck for utilizing global memory. Furthermore, it can be concluded that the most optimal strategy on the GPU is depending on the problem size and available hardware as different applications show different strategies to perform best. This conclusion is further supported by Cecka who proposed several memory management strategies and came to a similar conclusion [35]. Finally, tracing back to the initial statement by [30] that global memory is preferred over local shared memory, Banaś showed in [36] that for varying approximation orders of the shape functions the best performing solution differs per hardware setup. In fact, the total amount of threads available, in combination with the available memory, drives the performance of a GPU algorithm. This highlights the influence of the problem size even more.

What all literature so far has in common is that global memory is used to store assembled parameters of the model, which gives rise to a problem in the form of a race condition. The race condition occurs when multiple threads handling different elements want to access data at the same memory address simultaneously. To avoid the race condition, a graph coloring method was applied by Komatitsch *et al.* [31]. The graph coloring method creates distinct element sets such that elements handled in parallel have no underlying dependencies on each other. In general, the graph coloring method is found often in literature as it is suited for unstructured meshes due to the unpredictable connectivity of the mesh [8, 32, 33]. Another option is to use the "Addto" strategy proposed by Markall [37]. The Addto-strategy creates a direct link between the local element matrix and the global matrix with a mapping function. The link is created by either the use of the graph coloring method explained above or atomic updates, which handle attempts to update the same global matrix entry sequentially. While this seems more computationally expensive at first, the atomic update approach is preferred in literature [38].

To avoid the need to take the race condition into account an assembly-free FEM can be used which reduces memory requirements. Additionally, this strategy has more coalesced memory transactions as the matrix-vector products have regular-sized elemental matrices that result in a regular memory access pattern [7]. The difference in terms of assembly between regular and assembly-free FEM is illustrated in Figure 3.2.

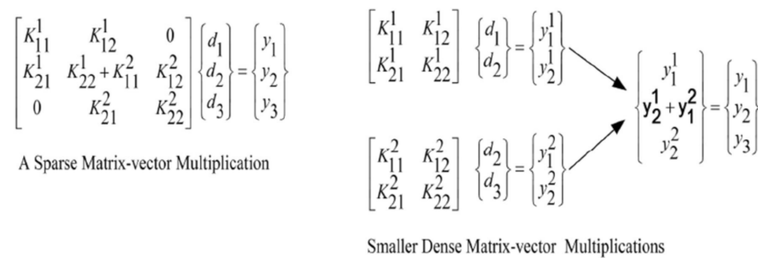


Figure 3.2: *Different forms of assembly of the global system. On the left a sparse matrix-vector product is depicted. Only a small system is depicted to serve as an example. In larger system the percentage of non-zero terms is much smaller resulting in a much more sparse system. On the right the matrix-vector product is split up into smaller, denser parts that are subsequently summed. Figure credits: [21].*

Assembly-free FEM exists in different forms including Element by Element (EbE) and Degree of freedom by Degree of freedom (DbD) methods. For EbE methods, the domain is split into elemental subdomains that do not have common degrees of freedom, removing the costly storage of large (sparse) global matrices as explored by Kiss *et al.* [23].

In the case of a DbD method, the contribution of different elements does not need to be tracked as each thread is assigned to a single unknown, i.e. a degree of freedom [38]. More recently, an iterative DbD approach was used by Pikle [7]. This approach launches multiple kernels that each take a certain amount of columns from the local element matrices to update the global matrix, making sure that memory transactions stay coalesced. Compared to a serial implementation, the DbD method achieved a 30× speed-up, while a 90× speed-up was achieved with the iterative DbD method [7]. Interestingly, the observation by Kiran in [33] that multiple kernel launches are an unfeasible option does not apply here, which can be attributed to the absence of the assembly part in the iterative DbD method.

The literature on EbE and DbD methods is promising as the race condition is avoided. However, the load balance over different subdomains is something that needs to be handled carefully as a load imbalance can lead to very inefficient implementations. Additionally, as noted by Pikle *et al.* [21], preconditioners are no longer easily implemented as preconditioners assume the global system to be assembled. In case the use of a preconditioner is desired, the preconditioner needs to be developed as shown by Kiss *et al.* [23]. Finally, EbE and DbD methods contain redundant computations as local element matrices need to be computed for different subdomains.

3.2 Current state of the art in solid mechanics

Apart from the GPU-accelerated implementations in FEMs for general applications, a lot of knowledge can be gained from GPU-accelerated implementations of FEMs applied specifically to solid mechanics. Therefore the current section only highlights the global level of FEMs and more emphasis is laid on the mechanics and the utlized constitutive material models.

3.2.1 Mechanical models on GPUs

First, a soft tissue modeling approach is analyzed. The underlying mechanical model is based on the Total Lagrangian formulation that is used in combination with an explicit time domain method (TLED) [39]. Important to note is the use of the lumped mass approximation in conjunction with mass-proportional damping. These assumptions yield diagonal matrices, allowing for the creation of a GPU implementation based on the fact that nodal force contributions are derived from local stiffness matrices. The same is true for nodal displacements, which are independent of one another. Based on these two facts, it is evident that both computations for each element and subsequently each node may be done in parallel, as demonstrated by Taylor *et al.* [10]. The algorithm is split into two computational kernels. The first kernel computes the elemental nodal force contributions, while the second kernel computes the new nodal displacements. A quasi-static approach is used for the loading in combination with the Neo-Hookean constitutive material model. Although it is considered a simple nonlinear constitutive formulation, the model is able to capture geometric non-linearities. In total, the GPU implementation was 16.8× faster compared to a serial CPU implementation.

A year later this model was tested further for multiple constitutive formulations [9]. The new tests resulted in a further 2.5× speed-up of the model which the authors attribute to memory access performance. More recently, the aforementioned TLED is rewritten to a form that is purely based on the Jacobian in different configurations [40]. This formulation contains strain- and time-invariant tensors that can all be computed beforehand based on the constitutive formulation that is used. The revised implementation resulted in a 1.42× speed-up compared to the original TLED algorithm. Comparing the computing time between the GPU and serial CPU implementation, speed-ups of 120× for the tetrahedral elements and 90× for hexahedral elements were obtained. A disadvantage is that both the stress and strain are now implicitly defined, necessitating post processing operations in case these parameters are desired.

All aforementioned models employ an explicit time scheme. Therefore, it is interesting to investigate implicit time scheme models such that a comparison can be made. Mafi and Sirouspour developed a model using the Newmark implicit time scheme [24]. The implicit time scheme ensures the stability of the model at the cost of solving a linear system of equations at each time step. From a mechanical point of view, it is based on the Total Lagrangian formulation and compares the conventional Preconditioned Conjugate Gradient(PCG) solver with an EbE implementation that is developed by the authors. Since the PCG solver is iterative, the linear system needs to be solved multiple times at each time step to reach a

converged solution. The GPU is in both methods used to compute element matrices, where each thread is assigned to a single element. Afterward, the PCG uses CUDA libraries to assemble the global matrix and subsequently compute the global sparse matrix-vector product and solve for the displacement. The EbE implementation skips the assembly step and replaces the sparse matrix-vector product with three kernels. The first kernel disassembles the external force vector, the second performs the element matrix-vector product and the third assembles the solution vector that contains the displacement. While specific speed-up numbers are absent, the results show that the EbE model is faster for small numbers of convergence iterations at each time step. For large convergence iteration numbers, the EbE method is slower than the conventional PCG solver.

Cai *et al.* developed a Node by Node (NbN) based conjugate gradient solver [41]. Due to the NbN fashion, no preconditioner is applied and the model is assembly-free. In terms of GPU usage, threads are first assigned to elements to compute elemental stiffness matrices. Next, each thread is assigned to a node and serially computes all contributions of other nodes to its own node. For a 2D and 3D cantilever beam speed-ups of 174× and 167× were obtained compared to a serial CPU implementation. It must be noted that in case the number of elements is decreased, the speed-up will decrease as well.

Comparing the different models in solid mechanics, several conclusions can be made. First, it is observed that in both implicit and explicit models the Total Lagrangian formulation is used. In contrast to the Updated Lagrangian formulation and computing spatial derivatives with respect to the configuration of a previous time step, the Total Lagrangian formulation computes spatial derivatives with respect to the initial configuration. This strategy saves memory and decreases the number of computations. Moving on, the EbE approach works for explicit time schemes regardless of the number of elements. For implicit time schemes, the number of elements is of influence as it affects the number of iterations required by the solver. Finally, the explicit models neither use preconditioners nor linear solvers. This significantly speeds up the total computation and makes explicit schemes more suited for parallelized GPU implementations. Consequently, a much larger speed-up with respect to a serial CPU implementation is anticipated in case a GPU implementation is utilized in conjunction with an explicit time scheme.

3.2.2 Constitutive material models

The constitutive material formulation dictates the computational cost of the computation of elemental contributions of the FEM for explicit dynamics. In implicit dynamics, the linear and nonlinear solvers of the FEM can be the bottleneck as well. Therefore it is interesting to investigate which material formulations are utilized in the literature.

Taylor included different hyper-elastic formulations ranging from isotropic to anisotropic versions [9, 10]. Additionally, a visco-hyper-elastic constitutive model was formulated. Incorporating visco-elasticity itself showed to only have 4.5% additional computational cost, while both visco-elastic and anisotropic properties together resulted in 5.1% additional computational cost. The models utilizing a implicit time scheme use a hyper-elastic material formulation, for which the additional cost of different implementations is not explicitly mentioned. All these hyper-elastic formulations are based on the Neo-Hookean constitutive material formulation [5], which is a hyper-elastic constitutive material formulation that captures geometric non-linearities.

Summarizing, GPU-accelerated solid mechanics models utilizing Neo-Hookean constitutive material formulations have been more widely studied in the literature. It is expected that larger speed-ups can be achieved in case a more complex constitutive model is chosen as this increases the computational intensity of the FEM [21]. Moreover, this can be combined with the observation of Banaś [36] presented in section 3.1 regarding the importance of the order of the shape functions. Combined with material non-linearities, a higher-order shape function will significantly increase the computational cost of the part computing the elemental contributions.

3.3 High performance computing

In the context of true scalability, the findings of an exploration into high performance computing are presented in this section. The initial focus is primarily on the load balance for single GPU implementations in case the memory demand exceeds the memory capacity of the GPU, which is often the case. Subsequently, the focus shifts toward multi-GPU implementations and the added complexity associated with balancing the load between different GPUs.

3.3.1 Single-GPU load balancing

Based on the prior performed investigations into GPU algorithms, it was found that the manner in which loops are handled on a GPU is very important for the performance that is achieved. Two ways that are frequently encountered in the literature regarding the utilization of loops in GPU algorithms are presented:

- **Grid-strided looping**
  It often occurs that the computational demand is larger than can be handled in one single loop on the GPU. The best strategy to work with this is by using a grid-strided loop, which means that after the first loop iteration each thread shifts by the total number of threads that is utilized and performs the same computation, only for the new memory address as explained in [34]. Such grid-strided looping occurs when dealing with large arrays as well. The grid size is chosen such that one row of the array is covered and the grid loops over the subsequent rows and performs the calculation.

- **Loop unrolling**
  Ideally, to achieve the highest performance on a GPU, a thread should never have to wait on other threads if the threads are in the same warp. However, conditional statements (e.g. if-statements or switch-statements) give threads the option to choose a different path and diverge from each other. Additionally, threads can follow a path in which variables are read or evaluated that will not influence the final outcome of a computation. By unrolling loops the compiler can optimize conditional statements such that threads can not diverge and threads that will not aid the outcome will not read variables or store intermediate results [14].

Taking the aforementioned ways of looping into account is beneficial for the total computing time of the computation and therefore aids in achieving a higher speed-up in comparison with a serial implementation.

3.3.2 Multi-GPU computing

Apart from single-GPU implementations, multiple GPUs can also be implemented together to speed up the total computation. Using multiple GPUs is a relatively new topic in literature as it requires a lot of synchronization and data transfer overhead between GPUs [21]. Still, attempts are made in different fields of study to develop implementations for multi-GPU hardware setups.

Dziekonski *et al.* investigated the conversion of the FEM from an iterative single GPU implementation to a parallel multi-GPU implementation [34]. The domain is decomposed into different subsets, which are subsequently allocated to different GPUs. Through assuring the subsets are decoupled, all subsets can be handled in parallel. For optimal performance, the CPU must always finish its tasks before the GPU finishes the parallel computation such that the CPU can carry on with the computation. Utilizing multiple GPUs decreases the GPU computing time significantly and can result in a situation in which the GPU has a faster computing time than the CPU. If this situation occurs, the computing time of the CPU becomes the main obstruction for a higher speed-up.

More recently, the overhead communication and load balance of using multiple GPUs was investigated [42]. GPU communication can be minimized by decomposing the domain over different GPUs directly after the discretization step of the FEM. Letting each GPU compute the elementary contributions and assemble its own subdomain results in an almost equal load balance between different GPUs. Using direct solvers leads to a speed-up of 5× and 10× for a single and multi-GPU implementation. A decrease in speed-up is observed for iterative solvers, in which case the single GPU implementation yields a 3.5× speed-up and the multi-GPU implementation a 5× speed-up respectively.

An important consideration regarding multi-GPU setups is that communication between GPUs should be minimized. Additionally, the decomposition of the domain in different subsets should be handled carefully since each GPU should get an equal load assigned. Based on the literature, it can be concluded that on top of the speed-up that can be gained in a single-GPU implementation, using multiple GPUs will provide another significant speed-up. This additional speed-up comes from the fact that a multi-GPU setup offers true scalability as previously explained in chapter 1.

3.4 Conclusion

In this chapter, a brief overview was given regarding the current state-of-the-art GPU-accelerated implementations. First, speed-up strategies affecting a single step in the FEM were examined. It was determined that structured meshes can provide significant speed-ups and handle large problems more easily than unstructured meshes. The predictability of the node connectivity results in a much lower memory demand for structured meshes. Furthermore, higher speed-ups can be achieved in case optimized storage formats, preconditions and linear solvers are used. Regarding speed-up strategies affecting multiple steps, it is concluded that the most optimal strategy for GPU-accelerated algorithms is depending on the problem size and available hardware as different applications show different strategies to attain the highest speed-up.

For the application to solid mechanics, the Total Lagrangian formulation is most commonly used as it has the lowest memory demand of the explored formulations. The low memory demand leads to the highest speed-up in combination with an explicit time scheme. Formulations including the Neo-Hookean constitutive material model have been more widely studied in the literature than other constitutive models. The geometric non-linearities the Neo-Hookean material model is able to capture, in conjunction with higher-order shape functions, can significantly increase the computational cost of the step that computes the elemental contributions.

The exploration into high performance computing resulted in several observations. In single GPU implementations the memory demand often exceeds the memory capacity of a single GPU and several strategies in literature exist to handle the resulting load-balancing problem. A multi-GPU setup offers true scalability and has been successful in an application to computational electromagnetics. Therefore, it is a promising approach to accelerate solid mechanics simulations as well.

Part II

The Finite Element Method in solid mechanics

4
Finite element formulation

Following the state of the art on GPU-accelerated FEA, the part on the FEM in solid mechanics commenes with a chapter explaining the finite element formulation applied to solid mechanics. First, the main steps in the finite element formulation for solid mechanics are examined in section 4.1. Subsequently, the governing equations are presented in section 4.2 for the finite element formulation utilized. As the governing equations in itself are continuous, the conversion to a discrete system of equations is described in section 4.3. Finally, the integration through time is explained in section 4.4.

4.1 Review of the main FEM formulation steps

In general, the FEM can be split up into four distinct steps [21, 31, 32]. Each of these four steps is separately discussed in this section. In solid mechanics, the system that is solved with the FEM, often called the body, comprises of a set of algebraic equations that yield the motion of the body. The external and internal forces acting on a body result in a inertial movement of the body in case of a dynamic system. In case there is no inertia term, the system is quasi-static. An input to the FEM is the mesh. In the mesh, the full domain of the body is discretized into smaller elements, each element consisting of a number of so called discretization nodes at its edges. Each node has a number of degrees of freedom, collectively representing the deformation of the body. Figure 4.1 illustrates the different steps of the finite element method in solid mechanics. These steps are described in more detail below.

1. **Shape function construction**
   The first step is to construct shape functions for the body under consideration. The shape functions can be of any order, depending on the application and type of element. The input for this step is the mesh of the domain. Based on the order of the shape functions, a certain number of integration points, or quadrature points, is required in each element as shown in the enlarged element in Figure 4.1. The solution on each element (e.g. the values of the deformation at the nodal degrees of freedom) is then represented as a linear combination of these shape functions with unknown coefficients which will be determined in subsequent steps. The construction of the shape functions is only computed once at the beginning of the FEM, making the computational load of this step relatively small.

2. **Computing elementary contributions**
   Based on the problem domain, the material parameters and the boundary conditions, a set of algebraic equations for the unknown coefficients can be assembled. For each element in the FEM, the elementary contribution to the set of algebraic equations is evaluated using numerical integration, graphically depicted by the enlarged element and the link to the quadrature rule and material model in Figure 4.1. Depending on the order of the shape functions and the complexity of the material model, this step can be computationally expensive. In fact, in contrast to the shape function construction, this step must be performed at least once per time step. In addition, this computation is performed for each quadrature point of each element. This results in a multiplier effect: In case higher-order elements and complex material models are considered in combination with a fine mesh comprising of many elements, the computational load becomes extremely high.

3. **Assembling the global system**
   Once all elementary contributions are evaluated, the partial results for each element can be assembled in the global system of equations that represents the entire body. This step is necessary because multiple elements may contribute to the algebraic equation for the same nodal degree of freedom. The assembly is in Figure 4.1 shown by the assembler and the connectivity of the local and global nodes for the enlarged element. The computational load of the assembly is similar to that of computing the elementary contributions as the assembly is done for each element as well. A distinction can be made between explicit and implicit time integration methods. For explicit time schemes, there is no need to assemble the global matrix as the solution depends only on the previous time step, which is not the case for implicit time schemes.

4. **Solving the global system**
   The solution step is shown on the top left of Figure 4.1. Typically there is an inertia term present, which makes the solving step more complex. Similar to the assembly step, a distinction can be made between explicit and implicit time schemes. With explicit schemes, the solution of the next time step is a function of the solution at the current time step, while for implicit schemes the solution of the next time step is a function of the next and current time steps and requires the solution of a (possibly non)linear system to perform the time update. Nowadays many different types of solvers exist to solve large-scale linear systems. Therefore, finite-element codes typically delegate this step to advanced open-source libraries for high-performance numerical linear algebra.

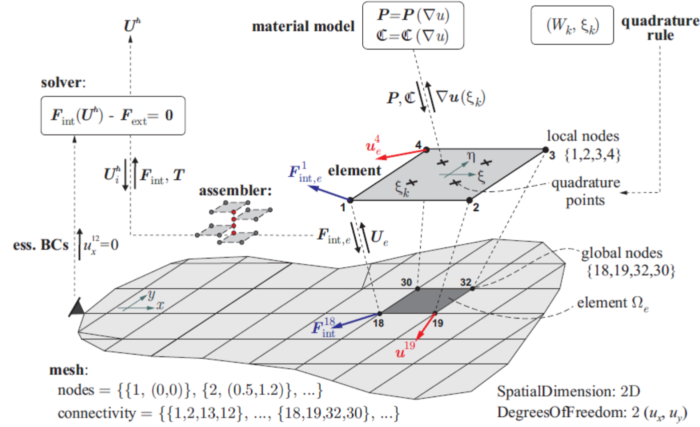


Figure 4.1: *A graphical representation of the four main steps in the FEM applied to solid mechanics. Starting from the discretized form of the domain, shape functions are constructed to form a linear combination that forms the solution. The coefficients of the linear combination are determined by first computing the elementary contributions, which depend on the shape functions and material model employed. Subsequently, the elementary contributions are assembled, as multiple elements may contribute to the algebraic equation for the same nodal degree of freedom. Following the assembly, the system is solved as depicted on the top left. Note that, in the figure, a quasi-static system is considered as there is no inertia term present. Figure credits: [43].*

Based on the description of the four main steps in the FEM, it is clear that the highest computational load arises in computing the elementary contributions and assembling the global system. Also, because this step consists of evaluating expensive calculations for each quadrature point in the domain independently, this step lends itself well to GPU parallelization. Consequently, the main focus in this study is to parallelize the computations present in computing the elementary contributions and assembling the global system as there the largest enhancement in performance is anticipated when utilizing a GPU-accelerated parallel implementation.

4.2 Governing equations

The current study utilizes the Total Lagrangian formulation. The choice to use the Total Lagrangian formulation stems from the fact that Lagrangian measures of stress and strain are utilized. In these measures, derivatives and integrals are obtained with respect to the Lagrangian material coordinates $\mathbf{X}$. Consequently, the mesh has the ability to follow material points through time and can deal with complex boundaries. First the description of a Lagrangian formulation is presented in subsection 4.2.1, followed by the explanation of the various balance principles the formulation adheres to in subsection 4.2.2. Finally, the Strong Form of the balance principles are converted into the Weak Form in subsection 4.2.3, such that the equations can be transformed into a discrete form for the FEM. The remainder of this chapter follows closely Belytschko's book on Nonlinear Finite Elements for Continua and Structures, which is summarized here for the reader's convenience [44].

4.2.1 Lagrangian description of the motion and measures

In solid mechanics, as the domain deforms in time, it is important to distinguish the current configuration of the domain from a reference configuration. The state of the reference configuration is known and often the initial domain is used as the reference configuration, which is the case in this study as well. The motion of the body Ω is described by Equation 4.1. Here $\mathbf{X_a}$ indicates the starting configuration of the body and $\mathbf{X_b}$ the final configuration of the body. The initial domain, depicted as $\mathbf{X_0}$, is the position of the body at $t = 0$. The system is dynamic, hence there is a dependency on time in the motion of the body.

$$\mathbf{x} = \phi\left(\mathbf{X}, t\right), \quad \mathbf{X} \in \left[\mathbf{X_a}, \mathbf{X_b}\right] \tag{4.1}$$

Moving on, the displacement $\mathbf{u}\left(\mathbf{X}, t\right)$ is described by Equation 4.2 and equals the difference between the current and initial position of the material point. Taking derivatives with respect to time, the expressions for the velocity and acceleration are obtained as depicted in Equation 4.3 and Equation 4.4 respectively.

$$\mathbf{u}\left(\mathbf{X}, t\right) = \phi\left(\mathbf{X}, t\right) - \phi\left(\mathbf{X}, 0\right) = \mathbf{x} - \mathbf{X_0} \tag{4.2}$$

$$\mathbf{v}\left(\mathbf{X}, t\right) = \frac{\partial \mathbf{u}\left(\mathbf{X}, t\right)}{\partial t} \tag{4.3}$$

$$\mathbf{a}\left(\mathbf{X}, t\right) = \frac{\partial^2 \mathbf{u}\left(\mathbf{X}, t\right)}{\partial t^2} \tag{4.4}$$

The deformation gradient is defined as shown in Equation 4.5, which is the gradient of the current configuration with respect to the reference configuration. Furthermore, the determinant of $\mathbf{F}$ is denoted by $J\mathbf{F}$ and is computed according to Equation 4.6. The Jacobian serves as a link between the current and reference configuration and transforms volume integrals from one configuration to the other, as shown by the example function f in Equation 4.7.

$$\mathbf{F} = \frac{\partial \phi}{\partial \mathbf{X}} = \frac{\partial \mathbf{x}}{\partial \mathbf{X}} = \left(\nabla_{\mathbf{0}} \phi\right)^T \tag{4.5}$$

$$J = \det\left(\mathbf{F}\right) \tag{4.6}$$

$$\int_{\Omega} f(\mathbf{x}, t) d\Omega = \int_{\Omega_0} f(\phi(\mathbf{X}, t), t) J d\Omega_0 \tag{4.7}$$

There are several conditions prescribed for the motion as listed below. The first condition ensures that the compatibility is satisfied such that there are no gaps or overlaps in the deformed body. The second requirement assures that there is a unique point in Ω for each point in Ω_0. The second and third requirement are related as the second conditions results in the fact that $\mathbf{F}$ is invertible, which is possible if and only if J is not equal to zero. The fact that J must be larger than zero follows from the Balance of Mass presented later on.

1. The function $\phi\left(\mathbf{X}, t\right)$ is continuously differentiable.
2. The function $\phi\left(\mathbf{X}, t\right)$ is one-to-one.
3. The determinat of $\mathbf{F}$ satisfies the condition $J > 0$.

The measure of strain is defined by Equation 4.8, which shows that there is no strain in case there is no deformation present as both the deformation gradient and its transpose are then equal to unity. Furthermore, the equation for Green's strain $\mathbf{E}$ can be written in a displacement format as well as depicted in Equation 4.9.

$$\mathbf{E} = \frac{1}{2}\left(\mathbf{F}^T \cdot \mathbf{F} - \mathbf{I}\right) \tag{4.8}$$

$$\mathbf{E} = \frac{1}{2}\left((\nabla_\mathbf{0}\mathbf{u})^T + \nabla_\mathbf{0}\mathbf{u} + \nabla_\mathbf{0}\mathbf{u} \cdot (\nabla_\mathbf{0}\mathbf{u})^T\right) \tag{4.9}$$

The stress measures considered in a Lagrangian description are the first Piola-Kirchhoff stress tensor $\mathbf{P}$ and the second Piola-Kirchhoff stress tensor $\mathbf{S}$. The first Piola-Kirchhoff stress tensor expresses the traction in terms of the area and normal of the reference configuration, which is the initial configuration in this study, and is defined as shown in Equation 4.10. The second Piola-Kirchhoff stress has no physical meaning and differs from the first Piola-Kirchhoff stress in the transformation of the force by $\mathbf{F}^{-1}$ as illustrated in Equation 4.11. Due to this transformation, the second Piola-Kirchhoff is a symmetric tensor.

$$\mathbf{n}_0 \cdot \mathbf{P} d\mathbf{\Gamma}_0 = \mathbf{t}_0 d\mathbf{\Gamma}_0 \tag{4.10}$$

$$\mathbf{n}_0 \cdot \mathbf{S} d\mathbf{\Gamma}_0 = \mathbf{F}^{-1} \cdot \mathbf{t}_0 d\mathbf{\Gamma}_0 \tag{4.11}$$

4.2.2 Balance principles

Following from the balance principles, the governing equations can be stated for the Total Lagrangian Formulation. There are three balance principles which are subsequently discussed, starting with the the Conservation of Mass in Equation 4.12.

$$\rho(\mathbf{X}, t) J(\mathbf{X}, t) = \rho_0(\mathbf{X}) \tag{4.12}$$

From Equation 4.12 it follows that J must be larger than zero. The mass of the material domain has to be constant since no material leaves the domain and mass to energy conversion is not considered. Moving on, the conservation of linear momentum is depicted in Equation 4.13. The left hand side consists of two terms. The first term represents the net internal force per unit volume of the initial configuration. The second term consists of the external body forces, $\mathbf{b}$, acting on the volume of the initial configuration. The right hand side represents the change in momentum, often called the inertial term, as it is the product of the initial density and acceleration.

$$\nabla_0 \cdot \mathbf{P} + \rho_0 \mathbf{b} = \rho_0 \frac{\partial \mathbf{v}(\mathbf{X}, t)}{\partial t} \tag{4.13}$$

The third governing equation is the Conservation of Angular Momentum. The equation is obtained by taking the cross-product of each term in Equation 4.13. Subsequently, it results in the fact that the product of the deformation gradient and nominal stress tensor should adhere to the symmetry conditions imposed by the balance of angular momentum on the Cauchy stress.

$$\mathbf{F} \cdot \mathbf{P} = \mathbf{P}^T \cdot \mathbf{F}^T \tag{4.14}$$

It is important to note that the current study only explores isothermal systems. Therefore, the equation for the Conservation of Energy is left out. To complete the set of governing equations, the boundary and initial conditions are described. The boundary conditions can be either a prescribed displacement or traction. It is impossible to have both a prescribed displacement and traction at the same boundary point, but one of both must be prescribed at each boundary point. The boundary of the body is indicated with Γ, where a boundary displacement is represented by Γ_u and a traction boundary by Γ_t respectively. The zero superscript indicates that the boundary of the initial configuration is considered. For a traction boundary the product of the nominal stress and the outward normal of the undeformed body should equal the imposed traction along the principal axes, as depicted in Equation 4.15. A displacement boundary is simply the imposed displacement value itself as indicated by Equation 4.16.

$$\mathbf{e}_i \cdot \mathbf{n}_0 \cdot \mathbf{P} = \mathbf{e}_i \cdot \bar{\mathbf{t}}_0 \qquad \text{on} \quad \Gamma^0_{t_i}, \tag{4.15}$$

$$u_i = \overline{u_i} \qquad \text{on} \quad \Gamma^0_{u_i} \tag{4.16}$$

Initially the body is undeformed and at rest. Consequently, the stress and the velocity of the body are zero in the initial state as depicted by Equation 4.17 and Equation 4.18.

$$\mathbf{P}\left(\mathbf{X}, 0\right) = \mathbf{0} \tag{4.17}$$

$$\mathbf{v}\left(\mathbf{X}, 0\right) = \mathbf{0} \tag{4.18}$$

Finally, the internal continuity, shown in Equation 4.19 must be satisfied as well. The internal continuity follows from the balance of linear momentum and dictates that there is no jump present in the stress measure, which holds for the entire interior of the body, represented by the boundary Γ_{int}, in the initial state.

$$[\![n^0_j P_{ji}]\!] = 0 \quad \text{on} \quad \Gamma^0_{int} \tag{4.19}$$

4.2.3 Derivation of the Weak Form

The described governing equations in the previous subsection are of the so-called Strong Form, meaning that the equations are required to hold for each and every material point. However, in FEA the body is discretized into a number of elements. A suitable Weak Form for FEA can be derived by multiplying the balance of linear momentum with a virtual displacement. The application of such a virtual displacement is called the principle of virtual work as shown in Equation 4.20. The different components of the virtual work are listed below in Equation 4.21 up to Equation 4.22 respectively.

$$\delta W = \delta W^{\text{int}}\left(\delta \mathbf{u}, \mathbf{u}\right) - \delta W^{\text{ext}}\left(\delta \mathbf{u}, \mathbf{u}\right) + \delta W^{\text{kin}}\left(\delta \mathbf{u}, \mathbf{u}\right) = 0 \tag{4.20}$$

$$\delta W^{\text{int}} = \delta \mathbf{u} \mathbf{f}^{\mathbf{int}} = \int_{\Omega_0} \delta \mathbf{F}^T : \mathbf{P} d\Omega_0 \tag{4.21}$$

$$\delta W^{\text{ext}} = \delta \mathbf{u} \mathbf{f}^{\mathbf{ext}} = \int_{\Omega_0} \rho_0 \delta \mathbf{u} \cdot \mathbf{b} d\Omega_0 + \sum_{i=1}^{n_{\text{sD}}} \int_{\Gamma^0_{ij}} \left(\delta \mathbf{u} \cdot \mathbf{e}_i\right) \left(\mathbf{e}_i \cdot \bar{\mathbf{t}}^0_t\right) d\Gamma_0 \tag{4.22}$$

$$\delta W^{\text{kin}} = \delta \mathbf{u} \mathbf{f}^{\mathbf{kin}} = \int_{\Omega_0} \delta \mathbf{u} \cdot \rho_0 \ddot{\mathbf{u}} d\Omega_0 \tag{4.23}$$

Important to highlight are the restrictions implied on the virtual displacement functions, which assure continuity. The first condition is that the virtual displacement functions must be continuous functions with piecewise continuous derivatives, pertaining to the continuity in X. Additionally, the virtual displacement functions are required to vanish on displacement boundaries. If the above conditions are met, both equilibrium and the described boundary conditions of the problem are satisfied.

4.3 Semidiscretization

To approximate the governing equations of the motion described in the previous subsection, the domain is partitioned in a number of elements that encompass the whole domain. The motion itself is approximated with the function shown in Equation 4.24 and the trial displacement by Equation 4.25. The variations utilized in the principle of virtual work are not a function of time and can be written as shown in Equation 4.26.

$$\mathbf{x}\left(\mathbf{X}, t\right) = \mathbf{x_I}\left(t\right) N_I\left(\mathbf{X}\right) \tag{4.24}$$

$$\mathbf{u}\left(\mathbf{X}, t\right) = \mathbf{u_I}\left(t\right) N_I\left(\mathbf{X}\right) \tag{4.25}$$

$$\delta \mathbf{u}\left(\mathbf{X}\right) = \delta \mathbf{u_I} N_I\left(\mathbf{X}\right) \tag{4.26}$$

In the above equations N represents the shape functions, which are interpolants functions of space only. Important to note is that indicial notation is employed, where all repeated indices are summed. Upper case indices represent nodal values and are summed over all nodes of the element, while lower case indices pertain to components and are summed over the number of dimensions. The shape functions satisfy the condition shown in Equation 4.27, where δ_{IJ} is the Kronecker delta which equals one at $I = J$ and zero if $I \neq J$. The subscript I ranges from zero to the number of nodes per element, while the subscript J indicates the location of the node in its element.

$$N_I\left(X_J\right) = \delta_{IJ} \tag{4.27}$$

Quadratic shape functions are examined, which require six nodal coordinates per element respectively. The reason for considering quadratic shape functions is the fact that it is the lowest order that contains multiple quadrature points per element, which will be elaborated upon later. The equations governing the shape functions are shown in Equation 4.28 up to Equation 4.33 [45].

$$N_1 = \left(1 - \xi - \eta\right)\left(1 - 2\xi - 2\eta\right) \tag{4.28}$$
$$N_2 = \xi\left(2\xi - 1\right) \tag{4.29}$$
$$N_3 = \eta\left(2\eta - 1\right) \tag{4.30}$$
$$N_4 = 4\xi\left(1 - \xi - \eta\right) \tag{4.31}$$
$$N_5 = 4\xi\eta \tag{4.32}$$
$$N_6 = 4\eta\left(1 - \xi - \eta\right) \tag{4.33}$$

With the description of the shape functions, the expression for the components of the deformation gradient can be written as shown in Equation 4.34. Here $\mathbf{B}$ represents the partial derivative of the nodal shape function with respect to the component of the initial deformation gradient. Transforming the equation to tensor notation results in Equation 4.35. Similar, the variation of the deformation gradient can be computed for a component according to Equation 4.36, which results in Equation 4.37 for the tensor notation. Note that the upper case subscript represents the nodes of the element under consideration, while the lower case subscripts are utilized for components of the tensor.

$$F_{ij} = \frac{\partial x_i}{\partial X_j} = \frac{\partial N_I}{\partial X_j} x_{iI} = B^0_{jI} x_{iI} \tag{4.34}$$

$$\mathbf{F} = \mathbf{x}\mathbf{B_0^T} \quad \text{where} \quad \mathbf{B_0} = \nabla_0 N_I \tag{4.35}$$

$$\delta F_{ij} = \frac{\partial N_I}{\partial X_j}\delta x_{iI} = \frac{\partial N_I}{\partial X_j}\delta\left(X_{iI} + u_{iI}\right) = \frac{\partial N_I}{\partial X_j}\delta u_{iI} \tag{4.36}$$

$$\delta\mathbf{F} = \delta\mathbf{u}\mathbf{B_0^T} \tag{4.37}$$

The derivatives of the shape functions are defined with respect to the reference configuration in the above equations. However, in the shape function expressions of ?? to Equation 4.33, the shape functions are expressed in terms of natural coordinates. Consequently, a conversion is necessary to establish a relationship between the shape function derivatives in the global coordinate system, denoted by (x, y), and the natural coordinate system, denoted by (ξ, η). This conversion is achieved by employing the Jacobian matrix, $\mathbf{J}_\xi$, that is obtained through the computation of Equation 4.38 [43]. It is important to note that the Jacobian matrix captures the transformation between the global and natural coordinate systems, as indicated by Equation 4.39, which transforms the nodal coordinates in the reference configuration between the natural and global coordinate system.

$$\mathbf{J}_\xi = \begin{bmatrix} \frac{\partial N_{iI}}{\partial \xi} x_i & \frac{\partial N_{iI}}{\partial \xi} y_i \\ \frac{\partial N_{iI}}{\partial \eta} x_i & \frac{\partial N_{iI}}{\partial \eta} y_i \end{bmatrix} \tag{4.38}$$

$$\mathbf{B_0} = \nabla_0 N_I = \mathbf{J}_\xi{}^{-1}\nabla_\xi N_I \tag{4.39}$$

As stated before, quadratic shape functions are examined, which require six nodal coordinates respectively. Referring to Equation 4.40, it is determined that quadratic shape functions necessitate three quadrature points, n_q, per element. Note that each quadrature point has a different $\mathbf{J}_\xi$, and therefore $\mathbf{B_0}$, as the coordinates of each quadrature point differ. The weights of the quadrature points and natural coordinates of each quadrature point in terms of ξ and η have been obtained from [46].

$$n_q = 2p - 1 \tag{4.40}$$

Employing Gaussian quadrature, the influence of each nodal point on the respective quadrature point is evaluated. Subsequently, summing the values of all elements contributing to a nodal degree of freedom by summing the quadrature points of each element, the found expressions for the variation of displacement and deformation gradient can be implemented in the expressions for the virtual work. Doing so for Equation 4.21 up to Equation 4.23 results in Equation 4.41 up to Equation 4.43 as expressions for the different force components, summed over the number of quadrature points of the element. Note that now a component notation is used, opposed to the tensorial notation in Equation 4.21 up to Equation 4.23, to clearly indicate the computation of a single component.

$$\delta W_{iI}^{\text{int}} = \sum_{q=1}^{n_q} \int_{\Omega_e} \delta u_{iI} B_{jI}^0 P_{ji} w_q d\Omega_e = \delta u_{iI} \sum_{q=1}^{n_q} \int_{\Omega_e} B_{jI}^0 P_{ji} w_q d\Omega_e$$

$$f_{iI}^{int} = \sum_{q=1}^{n_q} \int_{\Omega_e} B_{jI}^0 P_{ji} w_q d\Omega_e \tag{4.41}$$

$$\delta W_{iI}^{\text{ext}} = \sum_{q=1}^{n_q} \left(\int_{\Omega_e} \delta u_{iI} N_I \rho_0 b_i w_q d\Omega_e + \int_{\Gamma_{ti}^0} \delta u_{iI} N_I \overline{t_i^0} w_q d\Gamma_0 \right) = \delta u_{iI} \sum_{q=1}^{n_q} \left(\int_{\Omega_e} N_I \rho_0 b_i d\Omega_e + \int_{\Gamma_e} N_I \overline{t_i^0} d\Gamma_e \right)$$

$$f_{iI}^{ext} = \sum_{q=1}^{n_q} \left(\int_{\Omega_e} N_I \rho_0 b_i w_q d\Omega_e + \int_{\Gamma_e} N_I \overline{t_i^0} w_q d\Gamma_e \right) \tag{4.42}$$

$$\delta W_{iI}^{\text{kin}} = \int_{\Omega_e} \delta u_{iI} \rho_0 N_I N_J \ddot{u}_{jJ} d\Omega_e = \delta u_{iI} M_{ijIJ} \ddot{u}_{jJ} \quad \text{where} \quad M_{ijIJ} = \delta ij \int_{\Omega_e} \rho_0 N_I N_J d\Omega_e$$

$$f_{iI}^{kin} = M_{ijIJ} \ddot{u}_{jJ} \tag{4.43}$$

Important to highlight is the utilization of a diagonal, or lumped, mass matrix in Equation 4.43. Utilizing a lumped mass matrix avoids the need to solve a linear system when integrating over time as each row contains only a single non-zero entry. The non-zero entries are present on the diagonal of the matrix and are calculated by summing the values in the entire row. Finally, the expressions for the various force components for each node can be inserted in Equation 4.20. Shifting the term of the external force to the right hand side, the equation to solve for each nodal degree of freedom becomes Equation 4.44.

$$f_{iI}^{kin} + f_{iI}^{int} = f_{iI}^{ext}$$

$$M_{ijIJ} \ddot{u}_{jJ} + \sum_{q=1}^{n_q} \int_{\Omega_0} B_{jI}^0 P_{ji} w_q d\Omega_0 = \sum_{q=1}^{n_q} \left(\int_{\Omega_0} N_I \rho_0 b_i w_q d\Omega_0 + \int_{\Gamma_{ti}^0} N_I \overline{t_i^0} w_q d\Gamma_0 \right) \tag{4.44}$$

4.4 Time integration

The equation to solve for each nodal degree of freedom has been derived. Subsequently, the integration through time is discussed to present. In this study the explicit Newmark model is utilized to integrate through time. Explicit methods are computationally efficient, but subject to stability constraints. Implicit methods include a dependency on the solution at the next time step and have a increased computational load due to the computation of tangent matrices. Although implicit methods guarantee unconditional stability, which usually outweighs the more complex implementation, the focus in this study is on the GPU implementation and therefore only explicit methods are considered. Explicit methods are restricted in terms of stability by the critical time step, which is the largest possible time step that guarantees a stable solution at the next time step. The critical time step is computed by computing Equation 4.45 for each quadrature point of each element and selecting the smallest value. In Equation 4.45, h represents the relevant element length and c represents the critical wave speed, which is material model dependent. The equations for the critical wave speed are presented in chapter 5 for each material model respectively. .

$$\Delta t = \frac{h}{c} \tag{4.45}$$

With the time step known, the explicit Newmark model integrates the displacement and velocity through time according to Equation 4.46 and Equation 4.47. The predictor step for the values of the displacement, velocity and acceleration at each node. These predictions are made according to the following equations. Note that $\mathbf{v}$ has been replaced by $\dot{\mathbf{u}}$ and $\mathbf{a}$ with $\ddot{\mathbf{u}}$.

$$\mathbf{u}_{n+1} = \mathbf{u}_n + \Delta t \dot{\mathbf{u}}_n + \frac{\Delta t^2}{2} \left[(1 - 2\beta) \ddot{\mathbf{u}}_n + 2\beta \ddot{\mathbf{u}}_{n+1} \right] \tag{4.46}$$

$$\dot{\mathbf{u}}_{n+1} = \dot{\mathbf{u}}_n + \Delta t \left[(1 - \gamma) \ddot{\mathbf{u}}_n + \gamma \ddot{\mathbf{u}}_{n+1} \right] \tag{4.47}$$

The integration through time is split in two parts. First a so-called predictor step is taken by updating the displacement and velocity based on the part of the equation that depends on the current configuration of the body as shown below in Equation 4.48 and Equation 4.49. Subsequently, based on the values of the displacement and velocity at the next time step, updated values of the internal and external force contributions are computed. Utilizing Equation 4.44, the value of the acceleration at the next time step is determined.

$$\mathbf{u_{n+1}}^{pred} = \mathbf{u}_n + \Delta t \dot{\mathbf{u}}_n + \frac{\Delta t^2}{2} (1 - 2\beta) \ddot{\mathbf{u}}_n \tag{4.48}$$

$$\dot{\mathbf{u}}_{\mathbf{n+1}}^{pred} = \dot{\mathbf{u}}_n + \Delta t (1 - \gamma) \ddot{\mathbf{u}}_n \tag{4.49}$$

Finally, the values for the displacement and velocity are corrected with the acceleration at the next time step, as depicted in Equation 4.50 and Equation 4.51. In the current study the value of γ is equal to 0.5 and β is set to zero. Consequently, there is no corrector step present for the displacement.

$$\mathbf{u}_{n+1} = \mathbf{u}_{n+1}^{pred} + \Delta t^2 \beta \ddot{\mathbf{u}}_{n+1} \tag{4.50}$$

$$\dot{\mathbf{u}}_{n+1} = \dot{\mathbf{u}}_{n+1}^{pred} + \Delta t \gamma \ddot{\mathbf{u}}_{n+1} \tag{4.51}$$

It is important to highlight that the framework presented here is still agnostic with respect to the specific material constitutive behavior. The material constitutive behavior completes these equations by prescribing the relation between the kinetic quantities (in this case the Piola-Kirchhoff stress tensor) and the kinematic ones (in this case the deformation). The material models utilized will be described in the next chapter.

5
Material models

The framework that was presented in the previous chapter remains agnostic with respect to the material model. The material model prescribes a relation between a deformation measure (ε, $\mathbf{F}$ or $\mathbf{E}$) and a stress measure (σ, $\mathbf{P}$ or $\mathbf{S}$). This chapter presents a comprehensive overview of the material models utilized in this study. Each material model has a dedicated section describing the equations and underlying assumptions that govern the behavior of the model. The various material models are ordered based on computational complexity starting with the linear elastic material in section 8.1. Subsequently, the Neo-Hookean material model is discussed in section 5.2. Finally, the orthotropic composite material model is explained in section 8.3.

5.1 Linear material model

The linear elastic material model is perhaps te simplest material model in solid mechanics. The properties of the material are the same in each direction, i.e. the material is isotropic. For this case the second Piola-Kirchhoff stress can be described by Equation 5.1 [44].

$$\mathbf{S} = \lambda \mathrm{tr}\left(\varepsilon\right) \mathbf{I} + 2\mu\varepsilon \tag{5.1}$$

$$\varepsilon = \nabla_0 \mathbf{u} + \nabla_0 \mathbf{u}^{\mathbf{T}} \tag{5.2}$$

In the context of Equation 5.1, ε represents the linear strain part of Green's strain $\mathbf{E}$ and is computed with Equation 5.2. The parameters λ and μ are the first and second Lamé constants, respectively. Due to the linear nature of the model, the Cauchy stress collapses with the second Piola-Kirchhoff stress, removing the necessity of any additional transformation to the first Piola-Kirchhoff stress, $\mathbf{P}$. Finally, the critical wave speed can be computed with Equation 5.3. Utilizing the critical wave speed, the stable time step can be determined with Equation 4.45 for the linear elastic material.

$$c = \sqrt{\frac{\lambda + 2\mu}{\rho}} \tag{5.3}$$

5.2 Neo-Hookean material model

The Neo-Hookean material model is a hyper-elastic material model featuring geometric nonlinearity. The hyper-elastic behavior of the Neo-Hookean material model is characterized by an equation for the strain energy w, or stored energy potential ψ . Consequently, the derivative of the stored energy potential can be taken to yield the second Piola-Kirchoff stress tensor $\mathbf{S}$, described by Equation 5.4 [44].

$$\mathbf{S} = \frac{\partial w}{\partial \mathbf{E}} = 2\frac{\partial \psi}{\partial \mathbf{C}} \tag{5.4}$$

The equation defining ψ can be observed in Equation 5.5. By differentiating ψ with respect to the right Cauchy-Green strain $\mathbf{C}$, the expression for $\mathbf{S}$ is obtained as depicted in Equation 5.6. The conversion of $\mathbf{S}$ into the first Piola-Kirchhoff stress tensor $\mathbf{P}$ is accomplished by multiplication with $\mathbf{F}$ as illustrated in Equation 5.7.

$$\psi\left(\mathbf{C}\right) = \frac{\lambda}{2}\left(lnJ\right)^2 - \mu lnJ + \frac{\mu}{2}\left(\mathrm{tr}\left(\mathbf{C}\right) - 3\right) \tag{5.5}$$

$$\mathbf{S} = \left(\lambda lnJ - \mu\right)\mathbf{C}^{-1} \tag{5.6}$$

$$\mathbf{P} = \mathbf{F}\mathbf{S} \tag{5.7}$$

The computational complexity is governed by the presence of the determinant of the deformation gradient J, as it relies on all components of $\mathbf{F}$. Moreover, the computation of the inverse of $\mathbf{C}$ and the multiplication of F and S make the complexity rise further. Finally, the critical wave speed is computed according to Equation 5.8. Comparing the equation with Equation 5.3, it can be observed that the formulation for the Neo-Hookean material model is more complex.

$$c = \sqrt{J \frac{\lambda + 2\left(\mu - \lambda \log\left(J\right)\right)}{\rho}} \tag{5.8}$$

5.3 Orthotropic material model

The primary distinction between the orthotropic material model and the other models is the composite nature of the material, which makes it no longer isotropic. A composite material is build up in layers. Such a laminate contains different layers, or lamina, consisting of two different constituents. These constituents, often referred to as matrix and reinforcement, are combined such that the specific individual properties contribute to the overall performance of the material. The reinforcement, or fiber, has a high strength along its axial direction, but can not take up load in transverse direction. The matrix serves as a continuous medium that surrounds and binds the fibers, providing structural integrity and load transfer capabilities. The fibers are typically carbon, glass or aramid fibers, while the matrix is most often a polymeric material.

An orthotropic composite exhibits different mechanical properties in three mutually perpendicular directions. The longitudinal direction of the fibers is often referred to as the “fiber direction” or “1-direction”. The transverse direction, perpendicular to the fibers, is referred to as the “2-direction”, and the direction perpendicular to both the fibers and the transverse direction is known as the “3-direction” or “thickness direction”. By rotating the fiber direction through the different lamina, the properties of the laminate can be tailored to a specific application. However, the variation of material properties across different directions leads to a substantial increase in the computational workload. The stiffness of a single lamina is called a local stiffness matrix, or $\mathbf{Q_{local}}$, and is depicted in Equation 5.9.

$$\mathbf{Q_{local}} = \begin{bmatrix} q_{11} & q_{12} & q_{13} & 0 & 0 & 0 \\ q_{21} & q_{22} & q_{23} & 0 & 0 & 0 \\ q_{31} & q_{32} & q_{33} & 0 & 0 & 0 \\ 0 & 0 & 0 & q_{44} & 0 & 0 \\ 0 & 0 & 0 & 0 & q_{55} & 0 \\ 0 & 0 & 0 & 0 & 0 & q_{66} \end{bmatrix} \tag{5.9}$$

The components of $\mathbf{Q_{local}}$ are governed by the material properties as illustrated by Equation 5.10 up to Equation 5.17 [47]. Note that the matrix is symmetric.

$$q_{11} = \frac{1 - \nu_{23}\nu_{32}}{E_2 E_3 \Delta} \tag{5.10}$$

$$q_{12} = \frac{\nu_{21} - \nu_{31}\nu_{23}}{E_2 E_3 \Delta} = q_{21} \tag{5.11}$$

$$q_{13} = \frac{\nu_{31} - \nu_{21}\nu_{32}}{E_2 E_3 \Delta} = q_{31} \tag{5.12}$$

$$q_{22} = \frac{1 - \nu_{13}\nu_{31}}{E_1 E_3 \Delta} \tag{5.13}$$

$$q_{23} = \frac{\nu_{32} - \nu_{12}\nu_{31}}{E_1 E_3 \Delta} = q_{32} \tag{5.14}$$

$$q_{33} = \frac{1 - \nu_{12}\nu_{21}}{E_1 E_2 \Delta} \tag{5.15}$$

$$q_{44} = G_{23} = q_{55} = q_{66} \tag{5.16}$$

$$\Delta = \frac{1 - \nu_{12}\nu_{21} - \nu_{23}\nu_{32} - \nu_{31}\nu_{13} - 2\nu_{21}\nu_{32}\nu_{13}}{E_1 E_2 E_3} \tag{5.17}$$

In order to combine the stiffness of different layers, the local stiffness matrix is transformed to the global coordinate-system. The resulting global stiffness matrix, $\mathbf{Q_{global}}$ is depicted in Equation 5.18.

$$\mathbf{Q_{global}} = \begin{bmatrix} Q_{11} & Q_{12} & Q_{13} & 0 & 0 & Q_{16} \\ Q_{21} & Q_{22} & Q_{23} & 0 & 0 & Q_{26} \\ Q_{31} & Q_{32} & Q_{33} & 0 & 0 & 0 \\ 0 & 0 & 0 & Q_{44} & Q_{45} & 0 \\ 0 & 0 & 0 & Q_{54} & Q_{55} & 0 \\ Q_{61} & Q_{62} & 0 & 0 & 0 & Q_{66} \end{bmatrix} \tag{5.18}$$

The components of the global stiffness matrix of each layer, or ply, takes the ply angle, θ, into account according to Equation 5.21 up to Equation 5.32. Here m and n are the cosine and sine of the ply angle as illustrated in Equation 5.19 and Equation 5.20. Similar as for the local stiffness matrix, the global stiffness matrix contains symmetric components.

$$m = \cos(\theta) \tag{5.19}$$
$$n = \sin(\theta) \tag{5.20}$$
$$Q_{11} = n^4 q_{11} + 2m^2n^2 (q_{11} + 2q_{66}) + n^4 q_{22} \tag{5.21}$$
$$Q_{12} = m^2n^2 (q_{11} + q_{22} - 4q_{66}) + \left(m^4 + n^4\right) q_{22} = Q21 \tag{5.22}$$
$$Q_{13} = m^2 q_{13} + n^2 q_{23} = Q_{31} \tag{5.23}$$
$$Q_{16} = m^3 n q_{11} - mn^3 q_{22} + \left(mn^3 - m^3 n\right)(q_{11} + 2q_{66}) = Q_{61} \tag{5.24}$$
$$Q_{22} = m^4 q_{11} + 2m^2n^2 (q_{11} + 2q_{66}) + m^4 q_{22} \tag{5.25}$$
$$Q_{23} = n^2 q_{13} + m^2 q_{23} = Q_{32} \tag{5.26}$$
$$Q_{26} = mn^3 q_{11} - m^3 n q_{22} + \left(m^3 n - mn^3\right)(q_{11} + 2q_{66}) = Q_{62} \tag{5.27}$$
$$Q_{33} = q_{33} \tag{5.28}$$
$$Q_{44} = m^2 q_{44} + n^2 q_{55} \tag{5.29}$$
$$Q_{45} = mn (q_{55} - q_{44}) = Q_{54} \tag{5.30}$$
$$Q_{55} = n^2 q_{44} + m^2 q_{55} \tag{5.31}$$
$$Q_{66} = m^2n^2 (q_{11} + q_{22} - 2q_{12}) + \left(m^2 - n^2\right)^2 q_{66} \tag{5.32}$$

Although the mathematical operations involved are quite straightforward, the extensive number of computations can lead to increased computational cost. Following the transformation, the stress tensor can be computed by multiplication of the global stiffness matrix with the linear strain vector depicted by ε in Equation 5.33. The different components of ε are illustrated in Equation 5.34. Linear strain is used as solely small deformations are considered in compliance with the linear strain assumption for composite materials. Furthermore, due to small deformations, there is no conversion from $\mathbf{S}$ to $\mathbf{P}$ as the two measures are equivalent in the small-deformations regime.

$$\mathbf{S} = \mathbf{Q_{global}} \mathbf{C} \tag{5.33}$$
$$\varepsilon = \begin{bmatrix} \varepsilon_{xx} & \varepsilon_{yy} & \varepsilon_{zz} & 2\varepsilon_{yz} & 2\varepsilon_{xz} 2\varepsilon_{xy} \end{bmatrix}^T \tag{5.34}$$

Finally, the critical wave speed for the orthotropic material can be determined with Equation 5.35. Important to highlight is that for the composite material, only the speed of sound across the fibers is considered.

$$c = \frac{E_1}{\rho} \tag{5.35}$$

6
Parallelization strategies

Following the finite element formulation employed, the two parallelization strategies that are examined in this study are presented. The first parallelization strategy focuses solely on the conversion of the strain tensor into the stress tensor. In contrast, the second parallelization strategy incorporates additional steps, enveloping the conversion from displacement to strain and subsequently from stress to internal force. Consequently, the second parallelization strategy can be regarded as an extension of the first, as it entails more calculations within the finite element formulation. Both strategies have been implementated and results obtained with both will be presented in chapters chapter 7 and chapter 8.

6.1 Strain to stress

The first parallelization strategy encompasses only the computation of elementary contributions. Consequently, there is no dependency between different quadrature points as the assembly of the global system is still performed on the CPU. As a result, all the quadrature points present in the mesh can be evaluated simultaneously. This clearly results in a relatively simple, embarrassingly parallel GPU implementation.

The input toward the GPU is the deformation gradient tensor $\mathbf{F}$ for each quadrature point at the current time step. Subsequently, Green's strain, $\mathbf{E}$, is computed on the GPU and converted into the first Piola-Kirchhoff stress tensor $\mathbf{P}$. The relation between $\mathbf{E}$ and $\mathbf{P}$ is material dependent and the equations employed for each material model were presented in chapter 5.

6.2 Displacement to internal force

As mentioned earlier, the second parallelization strategy represents an extension of the first. Instead of considering only the computation of elementary contributions, the assembly of the global system is incorporated as well. As a result, there is a dependency between different elements and their quadrature points since, as explained in section 4.1, different elements may contribute to the algebraic equation for the same nodal degree of freedom. Consider the case in which two threads simultaneously want to update the result for the same nodal degree of freedom. If both threads fetch the same value from global memory, the contribution of one of the elements will be overwritten, leading to an incorrect result. Consequently, the dependency between different elements should be taken account in the parallelization strategy on the GPU.

In the second parallelization strategy the input to the GPU changes to the value of the nodal degrees of freedom at the current time step. As a result, both the shape function derivatives, $\mathbf{B}$, and $\mathbf{E}$ can be computed on the GPU. Subsequently, the first parallelization strategy is applied that computes $\mathbf{P}$ based on $\mathbf{E}$. Utilizing $\mathbf{B}$ and $\mathbf{P}$, the internal force component of Equation 4.44 can be computed. This parallelization strategy contains a considerably larger number of computations compared to the first parallelization strategy at the advantage of lighter memory transfers between CPU and GPU. Both the input and output consist of only a single value per nodal degree of freedom instead of a three-by-three tensor for each quadrature point.

Part III

Accelerating Finite Element Analysis with GPUs

7 Model-agnostic optimization

The previous part of the report has provided the equations for the finite element formulation and the material models that are considered. Additionally, the different parallelization strategies that will be examined have been explained. The third part of this report continues with optimizing the GPU-accelerated implementations,starting with optimizing the GPU implementation in a general, model-agnostic, sense. In section 7.1 the different GPU implementations are described, followed by the utilized performance metrics in section 7.2. The optimization of the CPU-GPU interaction is performed in section 7.3, exploring different implementations utilizing various memory types and examining the event scheduling. Subsequently, a novel approach to reduce the idle GPU time is considered in section 7.4. Finally, based on the aforementioned sections the optimal model-agnostic implementation is presented in section 7.5.

7.1 GPU-accelerated implementations

To determine the optimal GPU implementation for the different parallelization strategies, it is crucial to first optimize the GPU computation process in a general sense. This section elaborates on the different GPU implementations that were implemented and executed to assess their performance. To evaluate the performance of various implementations, different approaches were employed in the context of the strain-to-stress finite element implementation for the linear elastic material model. The selection of this particular combination was motivated by its relative simplicity, allowing for a focused analysis of the different implementation strategies. Importantly, the current optimization efforts remain agnostic to the specific material model and finite element implementation, ensuring that the findings can be generalized and applied to other constitutive material models and alternative finite element implementations.

For the GPU implementations, a Quadro RTX 5000 graphics card from NVIDIA was utilized. More information regarding the GPU can be found in Appendix A and in Appendix A for the CPU. A single CPU is considered without utilizing distributed memory. This way, no inter-CPU communication has to be performed for the GPU implementation, lowering the initial computational complexity of the implementation. If multiple CPUs were to be used with distributed memory, this would significantly increase the initial complexity and make the development of a GPU implementation more difficult. As mentioned before, NVIDIA offers the CUDA API. CUDA utilizes the *C++* programming language to execute code on GPUs. For the CPU portion of the code, the code developed by the research group, summit, is utilized which is written in *C++* as well.

Each of the following subsections provides a quick summary of one of the implementations evaluated. Each implementation builds on the preceding one, such that each change to the implementation may be evaluated separately in comparison to the prior one. The performance of a GPU implementation is driven by the instruction process and event schedule which were highlighted in section 2.3 and section 2.4 respectively. Consequently, for each implementation a instruction schedule and an event schedule are graphically illustrated. The instruction schedule indicates the number of instructions and order in which the instructions are executed, while the event schedule depicts the order and number of events that is executed.

7.1.1 GPU baseline implementation

The GPU baseline implementation involves the conversion of the CPU baseline implementation to a GPU-based implementation. Following a similar memory layout as the CPU counterpart, the memory organization within the GPU is arranged such that each deformation gradient tensor or stress tensor follows sequentially after one another. This memory layout is illustrated below in Figure 7.1.

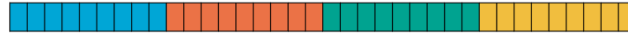


Figure 7.1: *A strided memory layout, where each color represents the components of a single deformation gradient tensor stored. Aligned with the SIMT principle, each thread concurrently performs the same task. Therefore, if each thread is assigned to a single quadrature point, all threads will first load the first component of their respective deformation gradient tensor. In the strided memory layout, deformation gradient tensors are ordered one after another. Consequently, eight memory addresses are present between the values that the threads of a warp fetch.*

Aligned with the SIMT principle, each thread concurrently performs the same task. Therefore, if each thread is assigned to a single quadrature point, all threads will first load the first component of their respective deformation gradient tensor. In the strided memory layout, deformation gradient tensors are ordered one after another. Consequently, eight memory addresses are present between the values that the threads of a warp fetch.

The purpose of the baseline implementation is to facilitate a direct comparison between the CPU and GPU implementation. Such a comparison highlights the effect on the performance of utilizing a GPU compared to a CPU. The event schedule and instruction schedule of the baseline implementation are graphically visualized in Figure 7.2. No streams are utilized, hence the events are executed sequentially. The issuing of instructions proceeds as follows: first all nine components of the deformation gradient tensor, $\mathbf{F}$, are fetched to local memory. Continuing, the trace of $\mathbf{F}$ is computed. Subsequently, each stress component is computed and stored in global memory. These two events are repeated nine times to compute all stress components.

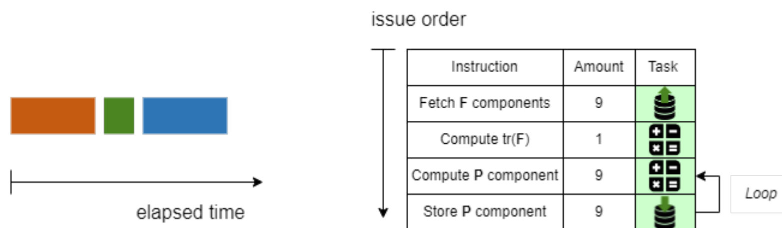


Figure 7.2: *A graphical representation of the GPU baseline implementation. The left image shows the event schedule, while the right image shows the instruction schedule. No streams are utilized, hence the events are executed sequentially. The issuing of instructions proceeds as follows: first all nine components of the deformation gradient tensor,* $\mathbf{F}$*, are fetched to local memory. Continuing, the trace of* $\mathbf{F}$ *is computed. Subsequently, each stress component is computed and stored in global memory. These two events are repeated nine times to compute all stress components.*

7.1.2 Coalesced implementation

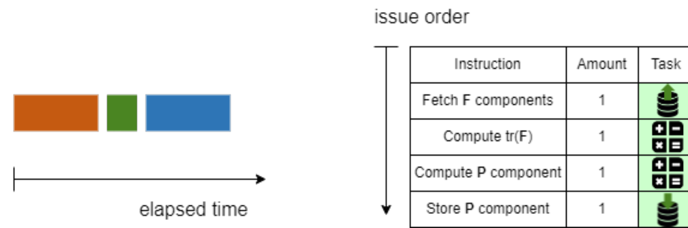
The objective of the coalesced implementation is to investigate the influence of the memory layout on the performance of the implementation. The coalesced memory arrangement optimizes memory access patterns by maximizing memory coalescence, thereby enhancing memory transfer efficiency. In the coalesced memory layout, the individual components of the deformation gradient tensors are stored contiguously as illustrated below in Figure 7.3. All other aspects of the implementation remain identical with the baseline implementation. The event and instruction schedule remain as depicted in Figure 7.2.

Figure 7.3: *A coalesced memory layout, where each color represents the components of a single deformation gradient tensor stored. When fetching global memory utilizing the coalesced memory layout, the values of the memory addresses that the threads of a warp visit are contiguous. Consequently, the time to retrieve values from global memory is reduced.*

7.1.3 Overload implementation

The overload implementation introduces a new approach where the computation of a single quadrature point is divided among nine threads, one for each component of the first Piola-Kirchhoff stress tensor, **P**. The primary objective of this implementation is to assess whether subdividing the workload in smaller portions, distributed over a larger number of threads, can lead to improved performance in the implementation.

The memory layout is kept as shown in Figure 7.3. Note that though the workload per thread is significantly smaller, the number of allocated threads is significantly larger. Additionally, partitioning the workload necessitates code branching using *if-statements* to ensure each thread computes the correct stress tensor component. As discussed in chapter 3, code branching acts as a significant performance inhibitor. The event and instruction schedule for this implementation are visualized in Figure 7.4. The event order is in this implementation sequential. In terms of instructions each thread loads a single component of **F** to local memory, computes the corresponding stress component and stores the value in global memory. Note that the computation of the trace only occurs in case the thread is assigned to a diagonal component of a quadrature point.



Instruction	Amount	Task
Fetch F components	1	
Compute tr(F)	1	
Compute P component	1	
Store P component	1	

Figure 7.4: *The visualization of the overload implementation. The left image shows the event schedule, while the right image shows the instruction schedule. No streams are utilized, hence the events are executed sequentially. In terms of instructions each thread loads a single component of* **F** *to local memory, computes the corresponding stress component and stores the value in global memory. Note that the computation of the trace only occurs in case the thread is assigned to a diagonal component of a quadrature point.*

7.1.4 Decoupled implementation

Even within the constitutive update at a quadrature point, certain parts of the calculation are independent of each other and can be decoupled. Consequently, the decoupled implementation focuses on adjusting the order of tasks within the instruction schedule based on an examination of the tensor **F**. Equation 5.1 is repeated below for convenience.

$$\mathbf{S} = \lambda \mathrm{tr}\left(\varepsilon\right)\mathbf{I} + 2\mu\varepsilon \tag{7.1}$$

Note that the equation can be simplified for the different components of **S**. For the diagonal components Equation 7.2 can be extracted, while Equation 7.3 is valid for the off-diagonal components. Below the equations the formulation of a single component is depicted to clarify the dependency

$$\mathbf{S}_{diagonal} = \lambda \mathrm{tr}\left(\varepsilon\right) + 2\mu\varepsilon_{diagonal} \tag{7.2}$$
$$S_{ii} = \lambda \mathrm{tr}\left(\varepsilon_{00} + \varepsilon_{11} + \varepsilon_{22}\right) + 2\mu\varepsilon_{ii}$$
$$\mathbf{S}_{off-diagonal} = 2\mu\varepsilon_{off-diagonal} \tag{7.3}$$
$$\mathbf{S}_{ij} = 2\mu\varepsilon_{ij} = 2\mu\left(\nabla_0 u_{ij} + \nabla_0 u_{ji}\right)$$

The components of the linear strain and displacement gradient correspond to the same components of **F** that are required for the calculation. In Figure 7.5 the deformation gradient tensor for a single quadrature point is represented as a three-by-three grid. The different colors illustrate the internal coupling that can be distinguished between different components of the deformation gradient based on the above stated simplified forms of Equation 7.1.

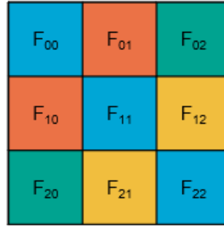


Figure 7.5: *A graphical representation of a deformation gradient tensor in terms of its components. The different colors illustrate the internal coupling between the diagonal components to compute the respective stress components. As a result, no diagonal stress component can be computed before all diagonal components are fetched from global memory. A similar coupling is present between three pairs of off-diagonal components.*

Benefitting from the coupling in the deformation gradient tensor, while maintaining the coalesced memory layout from Figure 7.3, the component implementation is visualized in Figure 7.6. Instead of fetching all nine components of **F**, only the diagonal components are fetched initially. Subsequently, computation and storage of the diagonal stress tensor components is performed. This process is then repeated sequentially for the three off-diagonal pairs. Through this investigation, it is tested whether the order of tasks in the processing event influences the performance of the implementation.

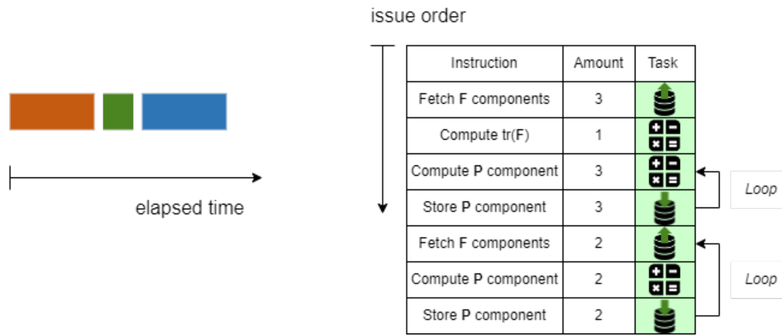


Figure 7.6: *The event and instruction schedule for the component implementation. Instead of fetching all nine components of* **F***, only the diagonal components are fetched initially. Subsequently, computation and storage of the diagonal stress tensor components is performed. This process is then repeated sequentially for the three off-diagonal pairs.*

7.1.5 Splitted implementation

The final implementation builds upon the previous implementation by further dissecting the processing event, while also partitioning the event schedule. Separate kernels are created for the diagonal and off-diagonal components, enabling the possibility to send a portion of the input data, subsequently followed by the computation of that specific portion. A visual representation of the event and instruction schedule is depicted in Figure 7.6 The sequence of tasks in the processing events remains identical to the previous implementation. However, instead of sending all input data initially in a single memory transfer, the input data is split up into four portions. Each portion contains the components of **F** that are represented by the same color in Figure 7.5, i.e. first the diagonal components, followed by the three off-diagonal pairs. Consequently, four kernel executions are present. The top instruction schedule represents the first kernel computing the diagonal stress components, while the remaining three kernels utilize the bottom instruction schedule.

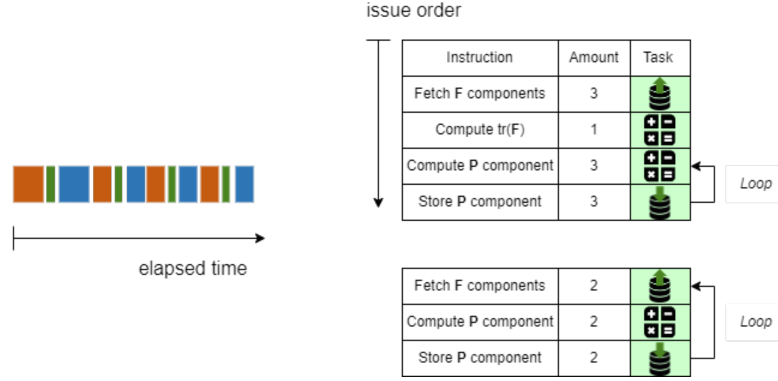


Figure 7.7: *The event and instruction schedule for the splitted implementation. The event schedule is partitioned based on the internal coupling of the deformation gradient tensor to first send and receive the diagonal components, followed by the three off-diagonal pairs. Consequently, four kernel executions are present. The top instruction schedule represents the first kernel computing the diagonal stress components, while the remaining three kernels utilize the bottom instruction schedule.*

The underlying objective of this implementation is to examine the impact of splitting computations across multiple kernels on the overall performance of the implementation. Additionally, through the partitioning of the interaction and processing events, streams can be utilized to perform the different events concurrently as discussed in section 2.4.

7.2 GPU performance metrics

To evaluate the aforementioned implementations, three different performance metrics are considered which will be utilized in the model-agnostic and model-specific optimizations in this study. Each of the performance metrics is explained below:

- **Total computation time**
  The computing time entails the CPU-GPU interaction, i.e. the memory transfers between CPU and GPU, as well as the GPU processing of the data, containing the fetching, compute and storage operations. The timing of results is performed via GPU profiling techniques that are explained in Appendix A. Important to note is that memory (de)allocation is not included in the total computation time. As de(allocation) occurs only once during the process, the effect on performance is negligible, especially in case the number of time steps becomes considerably large.

- **Total computation speed-up**
  Based on the total computation time, the total computation speed-up is computed with respect to the original CPU implementation. For the CPU implementation the timing of results is obtained through CPU profiling techniques described in Appendix B.

- **Computing speed-up**
  The computing speed-up focuses solely on the speed-up achieved by the processing event with respect to the CPU implementation. By disregarding the time spent on memory transfers, the performance of the CPU and GPU computation can be compared.

The GPU and CPU implementations are examined across various problem sizes, ranging from 1 million to 34 million computations of elementary contributions, where each computation belongs to a unique quadrature point. The inclusion of different problem sizes is motivated by the significant impact of memory transfers on GPU performance. For larger problem sizes, the performance can be heavily influenced by the efficiency of memory transfers. The smallest problem size was chosen to ensure sufficient workload for efficient GPU utilization, while the largest problem size was selected as a representative estimate of the total number of computations of elementary contributions in a single time step for a large (FEM) simulation. Each problem size was evaluated six times. The first run is used as a so-called "warm-up run" for the GPU, followed by five iterations. The average of the five iterations was computed for each performance metric to determine the performance at the evaluated problem size.

7.3 Optimizing CPU-GPU interaction

To optimize the interaction between the CPU and GPU, the aforementioned implementations are evaluated for each global memory type available on a GPU. First the performance of each global memory type is assessed in subsection 7.3.1. Subsequently, overlapping events by utilizing streams is explored in subsection 7.3.2. From both examinations key findings are extracted to determine the optimal model-agnostic GPU implementation.

7.3.1 Assessing memory transfer performance

The performance results for the unified global global memory type are depicted in Figure 7.8. Both the CPU and GPU implementations show to scale according to a linear trend. Such scaling was expected beforehand: i.e. when the problem size becomes twice as large, the computing time increases by a factor two as well. The findings demonstrate a significant improvement in performance compared to the baseline implementation across all the tested problem sizes. However, for larger problem sizes, a slight waviness can be observed for all implementations. This behavior can be attributed to the large number of memory transfers that occurs as the GPU iterates over the problem size, as discussed in subsection 2.2.2. Comparing the different implementations, both the coalesced and decoupled implementation exhibit the best performance, with a total computation speed-up of approximately 2.85×. As unified global memory resides on both the CPU and GPU, there is no distinction between the total computation speed-up and computing speed-up as explicit memory transfers can not be identified.

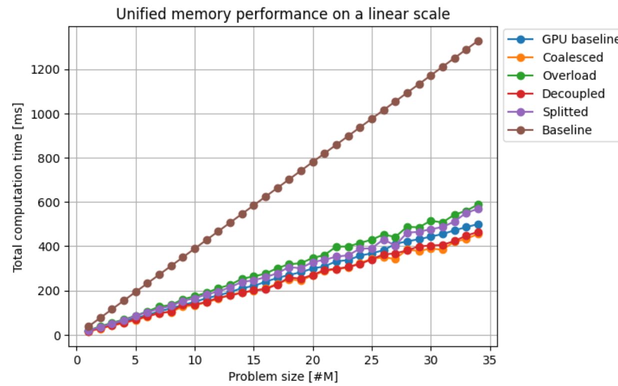


Figure 7.8: *Performance of the implementations utilizing unified global memory, demonstrating a significant improvement in performance compared to the baseline implementation across all the tested problem sizes. For larger problem sizes, a slight waviness can be observed for all implementations. Comparing the different implementations, both the coalesced and decoupled implementations exhibit the best performance, with a total speed-up of approximately* 2.85×.

The results for the implementations utilizing paged global memory are presented in Figure 7.9. At first glance, there is no noticeable difference in performance among the various implementations. Furthermore, the overall performance of the paged global memory is inferior to that of the unified global memory, as the vertical difference between the CPU baseline implementation and GPU implementations is smaller. Examining the computing speed-up, significant disparities among the implementations become evident. The computing speed-up comparison highlights the substantial impact of memory coalescence, increasing the computing speed-up from 75× to 102×. This observation emphasizes the importance of organizing memory transactions in a coalesced manner such that the highest performance is achieved. Another important observation to highlight is the fact that the lowest speed-up is demonstrated by the overload implementation which is an indication that code branching is detrimental for performance. Analyzing the figures, the total computation speed-up for paged global memory is 1.55×, while the computing speed-up is approximately 102×.

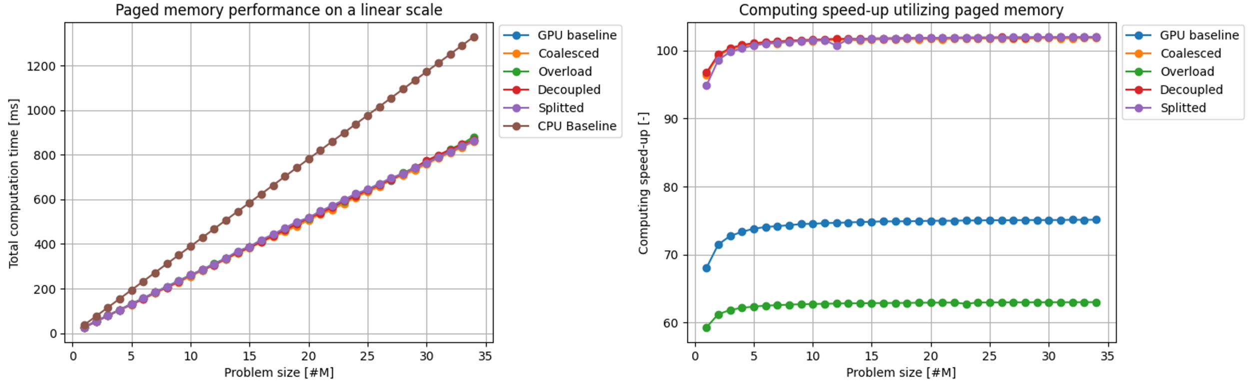


Figure 7.9: *Performance of the implementations utilizing paged global memory, the left figure shows the total computing time, while the right figure depicts the computing speed-up. At first glance, there is no noticeable difference in performance among the various implementations. Examining the computing speed-up, significant disparities among the implementations become evident. The computing speed-up comparison highlights the substantial impact of memory coalescence, increasing the computing speed-up from approximately* $75\times$ *to* $102\times$*. Comparing the figures, the total computation speed-up for paged global memory is* $1.55\times$*, while the computing speed-up is approximately* $102\times$*.*

Examining the results for the pinned global memory implementation in Figure 7.10 a similar trend to the paged global memory implementation is observed. However, the vertical separation between the CPU and GPU implementations is larger for pinned global memory, resulting in a larger speed-up equal to $3.43\times$. Moving on to the computing speed-up on the right, a strong similarity to the results for the paged global memory becomes evident as the computing speed-up is equal to $102\times$ as well, suggesting that the performance of the computation itself remains unaffected by the choice of memory type.

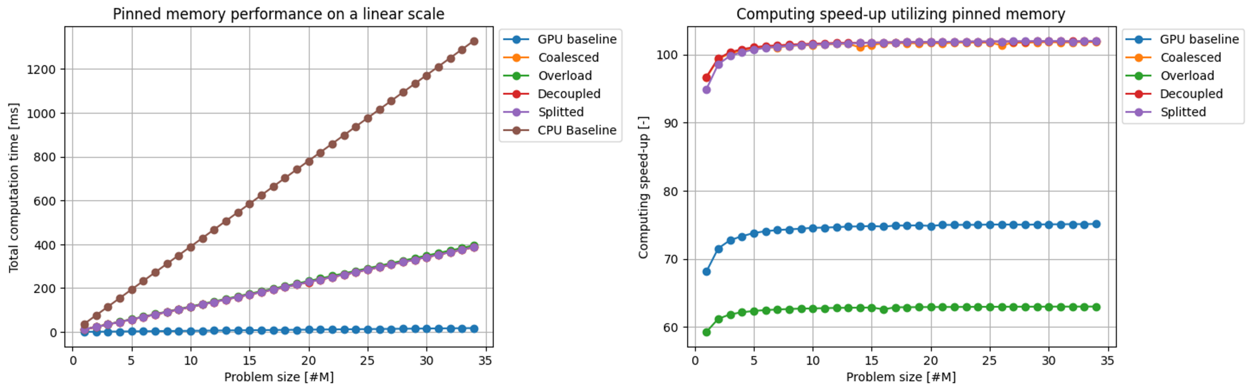


Figure 7.10: *Performance of the implementations utilizing pinned global memory, the left figure shows the total computation time, while the right figure depicts the computing speed-up. Both the total computation time and computing speed up show a strong similarity to the results for the paged global memory, suggesting that the performance of the computation itself remains unaffected by the choice of memory type. The total speed-up for pinned global memory is* $3.43\times$*, while the computing speed-up is approximately* $102\times$*.*

Turning to the mapped memory implementation presented in Figure 7.11, it becomes apparent that the strided implementation performs worse than the CPU implementation. The lesser performance of the strided implementation can be attributed to the zero-copy nature of the mapped global memory as explained in subsection 2.2.2. Furthermore, the concurrent and splitted implementations show similar performance in the mapped global memory scenario, suggesting that the order of tasks in the processing event has less impact on performance compared to the influence of memory access patterns and code branching when utilizing mapped memory. The total computation speed-up for mapped global memory is approximately $3.66\times$. Similar as for the unified global memory, no distinction is present between the total computation speed-up and computing speed-up due to the absence of explicit memory transfers.

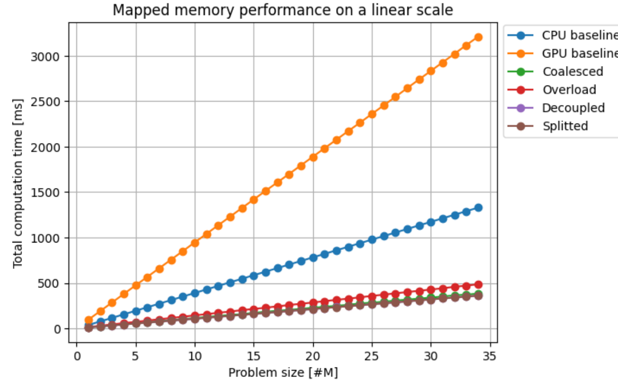


Figure 7.11: *Performance of the implementations utilizing mapped global memory. The strided implementation shows lesser performance compared to the CPU implementation. The concurrent and splitted implementations show similar performance in the mapped global memory scenario, suggesting that the order of tasks in the processing event has less impact on performance compared to the influence of memory access patterns and code branching when utilizing mapped global memory. The total computation speed-up for mapped global memory is approximately* 3.66×

Table 7.1 summarizes the best performing implementation for each global memory type. The coalesced implementation exhibits the largest total computation and computing speed-up for the different global memory types. However, it must be noted that the difference between the coalesced, concurrent and splitted implementations is only marginal, both in terms of total computation speed-up and computing speed-up. Overall, these findings highlight that the pinned and mapped global memory types show the best performance for the different implementations utilizing a single stream.

Memory type	Implementation type	Computing speed-up	Total computation speed-up
Unified	Coalesced	2.90×	2.90×
Paged	Coalesced	102×	1.53×
Pinned	Coalesced	102×	3.43×
Mapped	Coalesced	3.66×	3.66×

Table 7.1: *A summary of the speed-up results for different types of global memory, showing the best performing implementation, total computation speed-up, including the CPU-GPU interaction, and computing speed-up, excluding the CPU-GPU interaction. Both speed-up factors are with respect to the CPU implementation. The coalesced implementation exhibits the largest speed-up for each global memory type. Overall, these findings highlight that the pinned and mapped global memory types show the best performance for the different implementations utilizing a single stream.*

As the overload, concurrent and splitted implementation all utilize the coalesced memory layout, the difference in the event and instruction schedule drive the differences observed in the performance between the different implementations. Based on this observation, several conclusions can be drawn:

- Creating a larger grid to split the computation of a single quadrature point over multiple threads yields a decrease in performance, as tested with the overload implementation.
- Issuing all memory fetching tasks first results in similar performance as spreading the memory fetching tasks equally over the processing event. This is concluded based on the similar performance of the coalesced implementation and the decoupled implementation.
- Splitting kernels in a single stream results in similar performance as executing a single kernel, as indicated by the results of the splitted implementation.

7.3.2 Optimizing the event scheduling order

In the pursuit of further increasing the performance of the GPU implementations, streams are incorporated to facilitate concurrency in the event schedule as highlighted in Figure 7.12. The exploration of streams focuses specifically for the splitted implementation. As mentioned before, the splitted implementation facilitates a relative ease to incorporate streams in the implementation. The event schedule on the right indicates concurrent execution of the different events. As a result, the processing event commences at an earlier stage, which leads in a gain in performance. The instruction schedule remains the same as for the splitted implementation. The top instruction schedule represents the first kernel computing the diagonal stress components, while the remaining three kernels utilize the bottom instruction schedule.

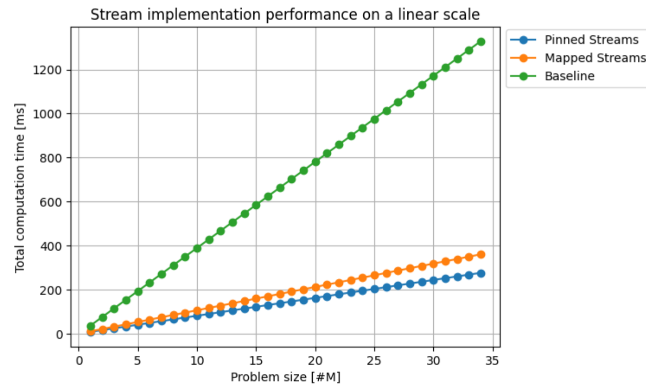


Figure 7.12: *The stream implementation event schedule and instruction schedule. The event schedule on the right indicates concurrent execution of the different events, facilitating an earlier start of the processing event. The instruction schedule remains the same as for the splitted implementation. The top instruction schedule represents the first kernel computing the diagonal stress components, while the remaining three kernels utilize the bottom instruction schedule.*

Unified and paged global memory can not be explored as both global memory types are incompatible with concurrent operations and stream utilization [13]. The results of the evaluation are depicted in Figure 7.13, where a slight distinction in performance is revealed between pinned and mapped global memory. The explicit memory copies present for pinned memory ensure that the data required for kernel execution is readily available. In contrast, mapped memory involves continuous swapping of data between the GPU and CPU throughout the kernel computation. This dynamic data swapping process increases the computing time and results in a reduced ability to hide latency compared to the pinned memory implementation. A total computation speed-up of 4.79× and 3.68× is achieved by the pinned and mapped global memory, respectively. The computing speed-up for pinned global memory is equal to approximately 98×.

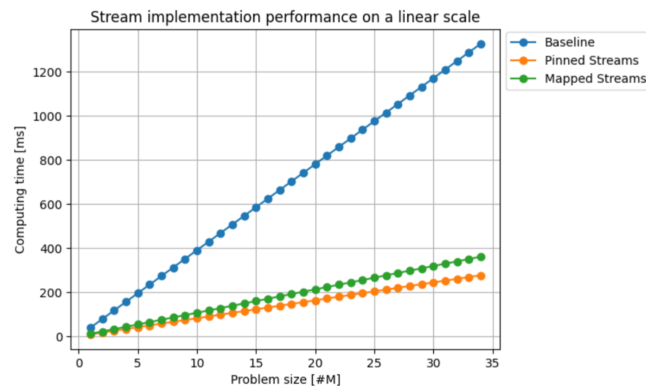


Figure 7.13: *The performance of the splitted implementation utilizing streams for pinned and mapped global memory reveals a slight distinction in performance between the two. A total computation speed-up of* 4.79× *and* 3.68× *is achieved by the pinned and mapped global memory, respectively. The computing speed-up for pinned global memory is equal to approximately* 98×.

7.4 Reducing idle GPU time

Building forth on the concurrency facilitated by streams, this section defines a novel concept to reduce idle GPU time. The reasoning behind the concept to reduce idle GPU time is presented in subsection 7.4.1 and the performance of the implementation is assessed in subsection 7.4.2.

7.4.1 Defining the concept

The exploration of streams established that overlapping interaction and processing events leads to optimal performance. However, it is possible to further increase the level of overlap between events, thereby reducing the time which the GPU is idle. To illustrate this concept, let us reconsider the example described in subsection 2.3.3, with Figure 7.14 provided below.

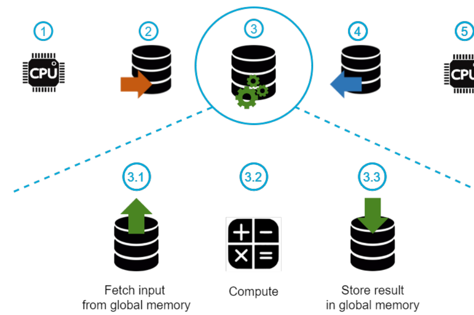


Figure 7.14: *The tasks present in the GPU processing event. First the the thread fetches the input data from the global memory. From the input data the thread computes its result which is subsequently stored in the global memory of the GPU. The tasks are executed in sequential order.*

In the current approach, the total number of threads, i.e. the grid size, is determined based on the number of quadrature points that needs to be evaluated. Yet, as explained in subsection 2.3.3 there is a limitation on the number of warps that can be instructed per clock cycle, which is determined by the number of Streaming Multiprocessors (SMs) available on the GPU. In each clock cycle an instruction is issued to a number of warps equal to the number of SMs. Figure 7.14 indicates that the first set of warps completes the fetching task, i.e. retrieving values from local memory, in approximately cycle 100. Consequently, the threads that received their fetching instruction in the first cycle can already commence their computation after 100 cycles.

In the current approach, all warps are first issued the instruction to fetch their values from memory before any warp is instructed to perform the computation task. As a result, the warps that received their instructions in the first cycle are stalled until all other warps complete their fetching task. This situation can be described as an excess of warps leading to idle GPU time. To fully utilize the capabilities of the GPU and minimize idle time, it is necessary to determine the grid size required to ensure that each SM has a sufficient number of warps to overcome the latency cycles of the initially instructed warps. Therefore, instead of creating a grid consisting of a number of threads equal to the total number of quadrature points that needs to be evaluated, a grid can be sized such that it contains enough threads to keep all SM's busy until the first warps complete their fetching task. After that, the SM can start issuing computational instructions to the warps that have finished their fetching task.

If the problem size is bigger than the number of threads present in the grid, the whole process can be repeated in a loop to process the entire problem size as illustrated in Figure 7.15. As a smaller grid is utilized in this novel concept, the amount of memory that initially needs to be transferred to the GPU is significantly reduced as highlighted in Figure 7.15. As a result, the computation commences earlier and results can be transferred to the CPU at an earlier stage. Reducing the GPU's idle time reduces the latency between the start of computation and the receipt of results, contributing to improved performance and efficiency of the GPU implementation. As a result, more overlap is created between the CPU-GPU interaction, i.e. the memory transfers, and the computations performed on the GPU.

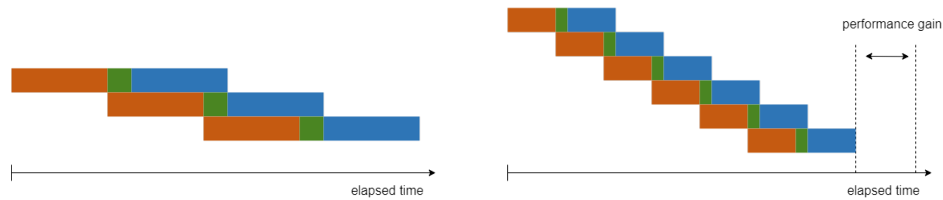


Figure 7.15: By considering the amount of threads the GPU can instruct per clock cycle, the grid size is reduced. Subsquently, the amount of memory that initially needs to be transferred to the GPU is significantly reduced. As a result, the computation commences earlier and results can be transferred to the CPU at an earlier stage. Reducing the GPU's idle time reduces the latency between the start of computation and the receipt of results, contributing to improved performance and efficiency of the GPU implementation.

7.4.2 Sensitivity analysis

In order to assess the impact of the grid size on the performance of the GPU implementation, a small sensitivity analysis is conducted. By varying the grid size, the impact on the performance of the implementation can be observed. This analysis allows for the identification of the optimal grid size to reduce idle GPU time and plays a role in assessing the memory type that should be used.

The results in Figure 7.16 show that the pinned memory outperforms the mapped global memory and stream implementation in reducing the idle GPU time as the total computation speed-up reaches 5.80×, leveling of to 5.43× for larger problem sizes. It is observed that for smaller problem sizes, the 400K grid size results in better performance compared to the 800K grid size. This observation makes sense as the 800K grid requires two loops to process the entire problem size in which the second loop contains only a small amount of quadrature points that still needs to be evaluated. Although the 400K grid requires more loop iterations, the workload per loop is more balanced, resulting in a better overall performance. The gain in performance associated with the concept to reduce idle GPU time demonstrates the effectiveness in maximizing GPU utilization and minimizing idle GPU time. The same reasoning as for the stream implementation in subsection 7.3.2 can be followed to explain the difference in results between the pinned and mapped global memory. On top of that, memory transfers, though smaller in size, occur more frequently, introducing more overhead cost. As a result, inconsistencies appear in the results for the largest grid size that was evaluated for mapped global memory.

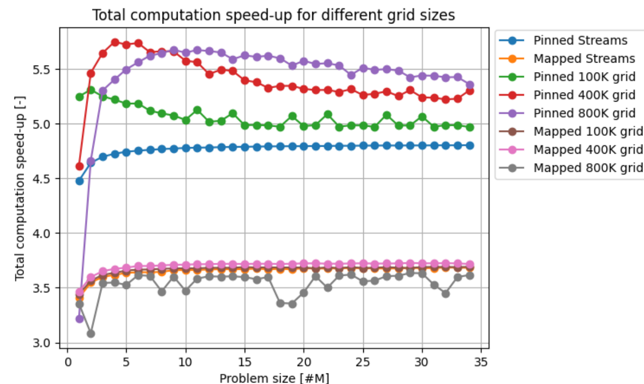


Figure 7.16: The results of the sensitivity analysis on the grid size in order to reduce idle GPU time. The pinned memory outperforms the mapped global memory and stream implementation in reducing the idle GPU time as the total computation speed-up reaches 5.80×, leveling of to 5.43×. Smaller memory transfers that occur more frequently introduce more overhead cost, leading to inconsistencies for the largest grid size that was evaluated for mapped global memory.

To implement the concept to reduce idle GPU time, a sensitivity analysis is performed for each material model in the model-specific optimization in chapter 8. In such manner, the varying memory transfer requirements and computational complexity can be taken into account, such the best performance for each individual material model can be achieved.

7.5 Conclusion

The evaluation of the different GPU implementations reveals several key findings. Firstly, all GPU implementations, with the exception of the strided implementation using mapped global memory, exhibit superior performance compared to the CPU implementation even after considering the time required by the CPU-GPU interaction. Moreover, a consistent linear relationship between the total computation time and problem size is observed across all GPU implementations. A notable difference in performance is observed between the implementations utilizing pinned or mapped memory, and those employing unified or paged memory. The implementations using pinned or mapped memory demonstrate a significantly lower total computation time compared to the unified and paged memory implementations. This difference can be attributed to the additional overhead associated with paged and unified global memory, as explained in chapter 2. The findings align with the expectations, as paged memory and unified memory incur additional memory transfers and exhibit limitations in terms of memory access efficiency.

Moving on to the optimization of the event scheduling order, the utilization of streams further increase the performance of pinned and mapped memory. The difference in performance for both memory types can be attributed to the inherent characteristics and handling of memory transfers within each approach. Yet, the resulting difference in performance is only small. Moreover, the computing architecture of the implementation is tailored to a specific grid size to minimize idle GPU time. Instead of executing a single kernel that covers the entire problem size, the implementation loops over the problem size. As a result, more overlap is created between the CPU-GPU interaction, i.e. the memory transfers, and the computations performed on the GPU.

In terms of verification, the results of the GPU-accelerated implementations have been compared with the results of the CPU implementation and were found to be equivalent. This shows that the GPU-accelerated implementations are capable of computing accurate results with enhanced performance compared to the CPU implementation. The results utilizing the pinned memory, which is determined to be the best performing type of global memory, across various implementations are summarized below in Table 7.2.

Implementation type	**Computing speed-up**	**Total computation speed-up**
Baseline GPU	74.85×	3.39×
Coalesced	101.63×	3.43×
Splitted including streams	97.66×	4.79×
Splitted reducing idle GPU time	100.48×	5.45×

Table 7.2: *The speed-up results for pinned global memory across different implementations. The optimal GPU implementation utilizes pinned memory with a coalesced memory layout. Furthermore, streams and the concept to reduce idle GPU time are utilized to further increase performance.*

In conclusion, the optimal GPU implementation, which is agnostic to the material model, utilizes pinned memory with a coalesced memory layout to maximize memory access efficiency and minimize data transfer overhead. The use of pinned memory ensures faster data transfers compared to other memory types, contributing to improved overall performance. The coalesced memory layout optimizes memory access patterns by aligning the memory layout with the underlying hardware architecture. Finally, maximal performance is achieved by tailoring the grid size to the computing architecture such that idle GPU time is minimized.

8
Model-specific acceleration

In the optimization of chapter 7, the GPU implementations have remained agnostic to the constitutive material model utilized. However, it is recognized that each constitutive model possesses distinct characteristics and potential benefits that can be leveraged to further optimize the kernels and memory transfers associated with each particular model. This chapter follows the model agnostic optimization by incorporating additional improvements for each constitutive material model specifically. Sections section 8.1 up to section 8.3 delve into the specifics of each material model and present the optimizations implemented to enhance the performance of the GPU implementation for the two parallelization strategies that were presented in chapter 6. In conclusion, section section 8.4 presents a comprehensive analysis and comparison of the performance achieved for each material model, highlighting the most effective GPU implementation for each parallelization strategy for each material model.

8.1 The linear elastic model

In this section the results of the optimization process conducted specifically for the linear elastic material model are presented. Both parallelization strategies are presented starting with the strain to stress strategy in subsection 8.1.1, followed by the displacement to internal force strategy in subsection 8.1.2. For each of these parallelization strategies the optimization starts with a sensitivity analysis to identify the optimal grid size at which the idle GPU time is reduced. Subsequently, the different implementations that were explored are explained and the results are presented.

8.1.1 Strain to stress parallelization strategy

To initiate the optimization process, the optimal grid size for the linear elastic material model in the strain to stress parallelization strategy is determined. The results of the sensitivity analysis are depicted in Figure 8.1. Based on the obtained results, the optimal grid size consists of $400K$ threads. This choice strikes a balance between the improvement in performance and the additional computational resources required as a larger number of threads only leads to a marginal increase in performance. Choosing a larger grid size results in larger memory transfers, as the number of threads in the grid increases, which increases the idle GPU time, while a smaller grid size would result in a substantial loss of performance for larger problem sizes. Subsequently, this optimal value was employed for the subsequent implementations.

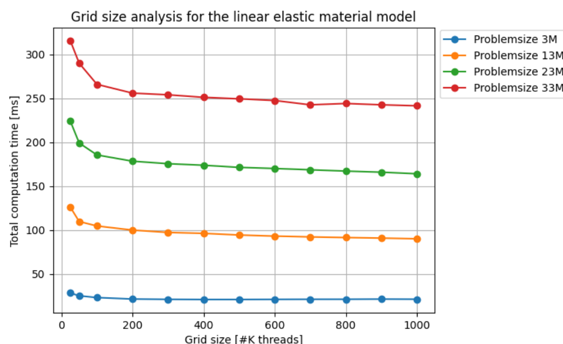


Figure 8.1: *The results of the grid size analysis across different problem sizes for the strain to stress parallelization strategy. Based on the obtained results, the optimal grid size consists of* $400K$ *threads. This choice strikes a balance between the improvement in performance and the additional computational resources required as a larger number of threads only leads to a marginal increase in performance.*

Continuing, four distinct implementations were tested for the strain to stress parallelization strategy. The GPU implementation exhibiting superior performance in the model-agnostic optimization, as presented in chapter 7, was employed as the foundation for these four implementations. Each of the different implementations is briefly described below. Similar as for the model-agnostic optimization, each implementation builds forth on the previous one. In order to assess the increase in performance that is achieved solely by the GPU, the GPU baseline implementation from chapter 7 is displayed in the results as well.

It is important to clarify the input and output data exchanged between the CPU and GPU in the various implementations. In all cases, unless explicitly stated otherwise, the input to the GPU consists of the deformation gradient tensor, $\mathbf{F}$. Conversely, the result returned from the GPU to the CPU corresponds to the first Piola-Kirchhoff stress tensor, $\mathbf{P}$.

Diagonal splitted implementation

Considering the internal coupling of $\mathbf{F}$, as indicated in Figure 7.5, an alternative approach can be employed to improve computational efficiency. Instead of calculating all three diagonal components sequentially and subsequently storing the results in global memory, each component of $\mathbf{P}$ can be computed independently in separate streams on the GPU. This approach allows for concurrent execution of the computations and independent storage of each component of $\mathbf{P}$. The event and instruction schedule are illustrated in Figure 8.2. The event schedule contains a larger initial memory transfer representing the three diagonal components of $\mathbf{F}$. Subsequently the top instruction schedule is executed three times to compute each diagonal component of $\mathbf{P}$, after which each computed component of $\mathbf{P}$ is transferred back to the CPU. Afterwards, the off-diagonal components of $\mathbf{F}$ are transferred to the GPU, the corresponding components of $\mathbf{P}$ are computed in pairs utilizing the bottom instruction schedule and the components of $\mathbf{P}$ are transferred back to the CPU. The process is repeated in a looping fashion to cover the full problem size.

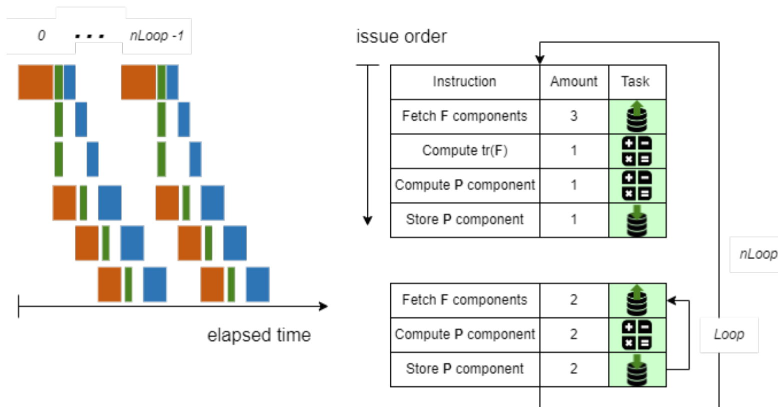


Figure 8.2: *The event and instruction schedule for the diagonal splitted implementation. The event schedule contains a larger memory transfer representing the three diagonal components of* $\mathbf{F}$*. Subsequently the top instruction schedule is executed three times to compute each diagonal component of* $\mathbf{P}$ *and transfer the component back to the CPU. Afterwards, the off-diagonal components of* $\mathbf{F}$ *are transferred to the GPU, the corresponding components of* $\mathbf{P}$ *are computed in pairs utilizing the bottom instruction schedule and the components of* $\mathbf{P}$ *are transferred back to the CPU.*

The present implementation investigates the potential performance improvement resulting from partitioning the calculation of the diagonal stress components into different streams. It is crucial to note that, to guarantee accurate and reliable results, the computation can only commence once all three diagonal components of $\mathbf{F}$ have been stored in the global memory of the GPU. Appropriate CUDA functions have been utilized to assure this order in the event schedule.

Stress symmetry implementation

The second implementation capitalizes on the symmetry of the first Piola-Kirchhoff stress tensor $\mathbf{P}$ for the linear elastic model as previously discussed in chapter 5, which allows to limit the memory transfer from nine tensor components to six. It is anticipated that this implementation will yield an overall performance

improvement compared to the previous implementation. The exploitation of symmetry not only reduces the computational workload but also minimizes the size of the memory transfer, thus enhancing the computational efficiency of the GPU implementation. The implementation is visualized in Figure 8.3. Compared to the previous implementation the number of interaction events returning results has been reduced. This reduction is visible in the the bottom instruction schedule as well since the number of tasks is reduced.

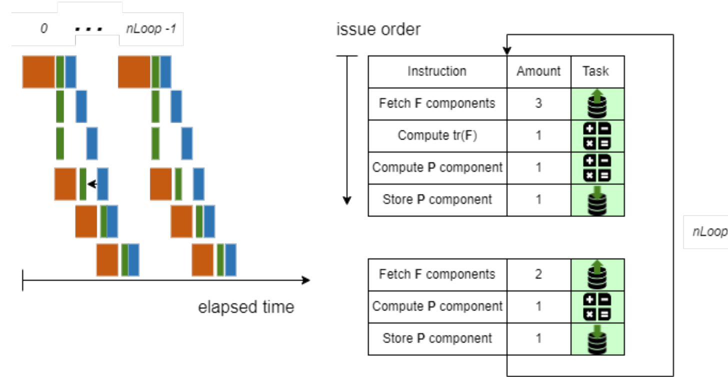


Figure 8.3: *The event and instruction schedule for the stress symmetry implementation. Compared to the previous implementation the number of interaction events returning results has been reduced. This reduction is visible in the the bottom instruction schedule as well since the number of tasks is reduced.*

Double symmetry implementations

In this implementation, a symmetry benefit is explored by examining the effect of utilizing ε as the input data instead of $\mathbf{F}$. Prior to transferring data to the GPU, ε is precomputed from $\mathbf{F}$ on the CPU. Due to the symmetric nature of ε, only six components need to be transmitted to the GPU, in contrast to the nine components required when using $\mathbf{F}$. The revised event and instruction schedule are depicted in Figure 8.4. Due to the decrease in memory transfers towards the GPU, the event schedule becomes much more compact as the second loop can start earlier in time. Additionally, the number of fetching tasks in the bottom instruction schedule is decreased.

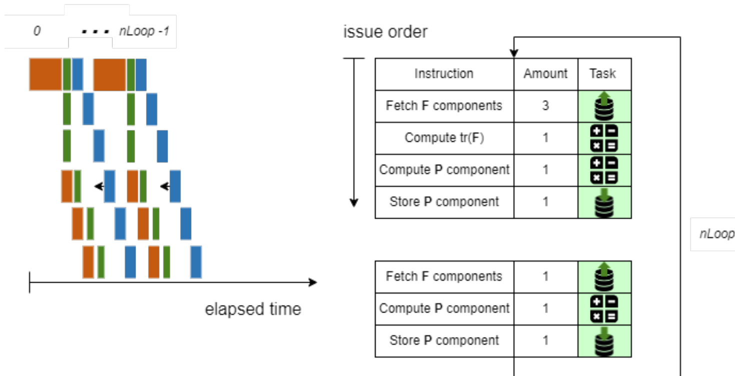


Figure 8.4: *The event and instruction schedule for the double symmetry implementation. Due to the decrease in memory transfers towards the GPU, the event schedule becomes much more compact as the second loop can start earlier in time. Additionally, the number of fetching tasks in the bottom instruction schedule is decreased.*

The aim of this implementation is to investigate whether the prior computation of the symmetric strain tensor ε leads to performance improvements. It is important to consider that if the computation time required for ε exceeds the time needed for memory transfer of the additional three components of $\mathbf{F}$, the implementation may fail to deliver performance gains.

Implementation results

The outcomes of the implementations are depicted in Figure 8.5. From the 2.90× total computation speed-up achieved by the baseline implementation, an increase to a speed-up of 5.76× is realized by utilizing the characteristics of the linear elastic material model. Both the diagonal splitted and stress symmetry implementations show a significant increase in total computation speed-up reaching $5.55 times$ and $5.76 times$, respectively. The computing speed-up is constantly 74.8× for the diagonal splitted implementation and 86.3× for the stress symmetry implementation. A striking observation is the presence of distinct peaks in the double symmetry implementation, which are attributed to the computation of ε on the CPU. The amount of memory that is required on the CPU is determined based on the problem size. For certain problem sizes a large stride exists resulting in a significant decrease in performance at those specific problem sizes. Even though the computing speed-up is significantly larger compared to other implementations, the addition of the CPU computing time of the tensor ε results in a lesser overall performance for the double symmetry implementation with a total computation speed-up of 4.45× on average and a computing speed-up of approximately 171×.

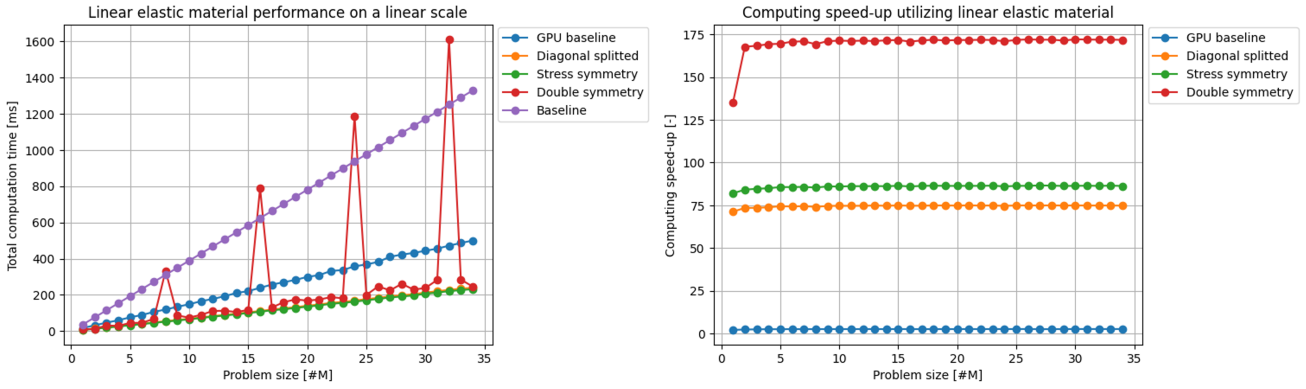


Figure 8.5: *Performance of the linear elastic model implementations. From the 2.90× total computation speed-up achieved by the baseline implementation, an increase to a speed-up of 5.76× is realized. Both the diagonal splitted and stress symmetry implementations exhibit a considerable increase in both total computation speed-up and computing speed-up, with the total computation speed-up equal to 5.55× and 5.76× respectively. The computing speed-up is constantly 74.8× for the diagonal splitted implementation and 86.3× for the stress symmetry implementation. The addition of the CPU computing time of the tensor ε results in a lesser overall performance for the double symmetry implementation with a total computation speed-up of 4.45× on average and a computing speed-up of approximately 171×.*

Continuing, the visible difference between the diagonal splitted and stress symmetry implementation when examining the computing speed-up almost completely vanish when examining the total computation time. This observation can be ascribed to the linear elastic material model's small computation demand relative to the memory demand. Given that each memory transfer necessitates more time than the execution of a computational kernel, the GPU often awaits the availability of new data, leading to limited performance gains. This is highlighted in the event schedules presented for the different implementations by the large amount of white space present between the different events. In the ideal implementation, no white space, i.e. idle time, would be present as explained in section 2.4.

8.1.2 Displacement to internal force parallelization strategy

For the displacement to internal force strategy it is important to note that the equations are evaluated for a 2D case. Since there exist nodal dependencies between different quadrature points, a simple finite element solver was developed. To limit the time spent on the development of the finite element solver, as that is not the scope of the current study, a 2D case was implemented for each material model utilizing quadratic triangular elements following the shape function formulation described in chapter 4.

The results for the grid size analysis are shown below in Figure 8.6. Examining the results, the optimal grid size is determined to be equal to $700K$ threads. Similar to the strain to stress strategy, this grid size

provides a considerable increase in performance compared to smaller grid sizes for all problem sizes, while not increasing the grid size too far. The optimal grid size is as almost twice as large as for the strain to stress strategy, which can be attributed to the significantly larger number of computations present in this parallelization strategy.

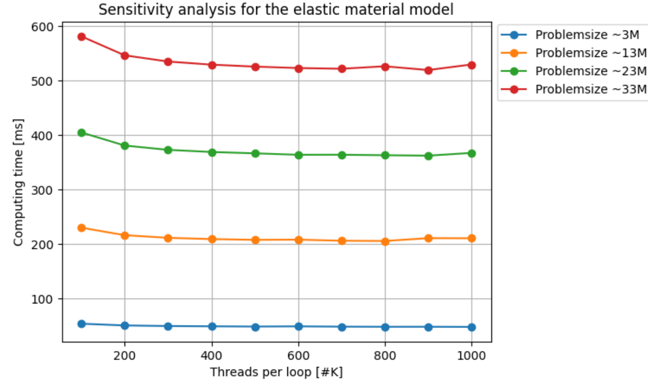


Figure 8.6: *The results of the grid size analysis for the linear elastic material model in the displacement to internal force strategy. Based on the results, the optimal grid size is determined to be equal to* $700K$ *threads. Larger grid sizes do not yield a further increase in performance compared to the optimal value.*

Three different implementations are examined for this parallelization strategy. Note that the implementation using the concept to reduce idle time is evaluated utilizing the optimal value that was previously determined. However, a potential benefit of the displacement to internal force strategy is the option to drastically decrease the memory demand, by transferring only the nodal displacements from CPU to GPU, instead of a three-by-three tensor for each quadrature point of each element. As there exists a coupling between different nodes, all nodes have to be send at once in order to assure all required nodal values are present on the GPU. If only a single memory transfer is utilized, the benefit from the concept to reduce idle GPU time vanishes since the loop would only contain the computation kernel and no memory transfers. Consequently, implementations without the concept to reduce idle GPU time are evaluated as well.

An important note to make is the fact that several memory transfers are only made once during the entire simulation in the displacement to internal force parallelization strategy. These memory transfers are not taken into account for the total computation time. The reason for this is that the number of time steps is envisioned large enough that the influence of the initial memory transfers becomes negligible. These memory transfers include the shape function derivative values, $\mathbf{B}$, the conversion jacobian, $\mathbf{J}_\xi$, of each quadrature point and the weights of the quadrature points w_q. Similar to the strain to stress strategy, an implementation utilizing unified global memory with a strided memory layout is used as GPU baseline implementation.

Reducing idle GPU time implementation

The first implementation that is examined contains the concept to reduce idle GPU time. In order to apply the concept, the nodal displacements are structured in an array in such a way that for each quadrature point of each element the nodal displacements are present. Consequently, the input data toward the GPU consists of twelve nodal displacement values per quadrature point. The output data retrieved from the GPU are the internal force components at each quadrature point for each element. The event schedule and instruction schedule for this implementation are shown below in Figure 8.7. The event schedule remains relatively simple with a single memory transfer in each direction for each loop. Interpreting the instruction schedule, it is clear that the number of computations is significantly increased for the displacement to internal force implementation. First the nodal displacement and shape function derivative values are fetched from global memory. Subsequently, $\mathbf{F}$ and the corresponding $\mathbf{P}$ are computed. Finally, after fetching J_ξ and w_q, $\mathbf{P}$ is contracted with the shape function gradient in order to obtain the internal force, f, for the nodal degree of freedom at the quadrature point under consideration. The steps starting from fetching J_ξ are repeated for each nodal degree of freedom of the quadrature point.

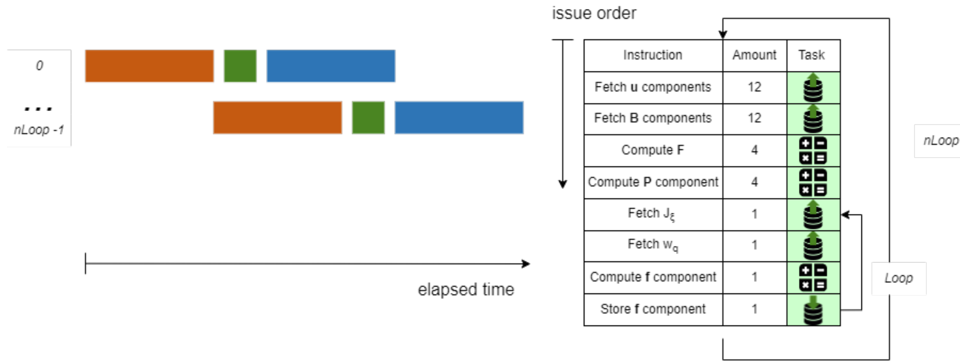


Figure 8.7: *The event and instruction schedule for the reducing idle GPU time implementation. The event schedule remains relatively simple with a single memory transfer in each direction for each loop. Interpreting the instruction schedule, it is clear that the number of computations is significantly increased for the displacement to internal force implementation. First the nodal displacement and shape function derivative values are fetched from global memory. Subsequently,* $\mathbf{F}$ *and the corresponding* $\mathbf{P}$ *are computed. Finally, after fetching* J_ξ *and* w_q*,* $\mathbf{P}$ *is contracted with the shape functions gradient in order to obtain the internal force,* f*, for the nodal degree of freedom at the quadrature point under consideration. The steps starting with fetching* J_ξ *are repeated for each nodal degree of freedom of the quadrature point.*

The goal of this implementation is to assess the performance of the concept to reduce idle GPU time in the context of a parallelization strategy that in itself would require less memory to be transferred between the CPU and GPU if the concept would not be utilized.

Reduced memory implementation

The second implementation does no longer utilize the concept to reduce idle GPU time. Instead, the amount of memory that is transferred between the CPU and GPU is significantly reduced. A single memory transfer is made initially to transfer the nodal displacements toward the GPU. In order to make sure that for each quadrature point the correct nodal displacement is utilized, the element connectivity matrix is transferred toward the GPU as well, such that each thread can access the nodal displacements for its corresponding quadrature point via the connectivity array. Note that, since the element connectivity is send only once before the first time step, it is regarded as an initial memory transfer and not taken into account for the total computation time. The event and instruction schedule are illustrated in Figure 8.8 for this implementation. The amount of memory transferred in the event schedule is significantly reduced, while the instruction schedule itself does not change. Through the significant reduction in memory transfers it is evaluated whether such a reduction in memory is more beneficial for the total compared to the implementation that reduces the idle GPU time.

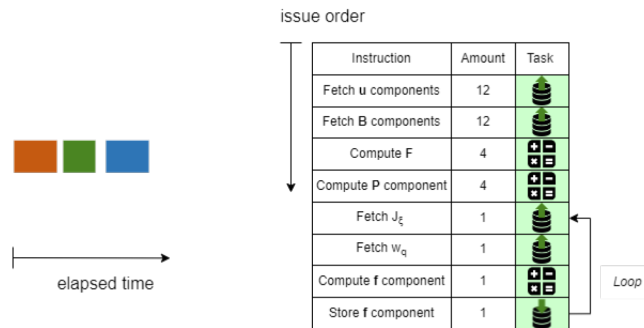


Figure 8.8: *The event and instruction schedule for the reduced memory implementation. A single memory transfer is made to transfer the nodal displacements toward the GPU. Subsequently, the instruction schedule is only executed once for a larger grid compared to the previous implementation.*

Concurrent implementation

The final implementation applies streams to concurrently execute different tasks to overlap memory transfers with computations, with the goal to reduce the total computation time. The event and instruction schedule are depicted in Figure 8.9. The transfer of the nodal displacements is split into two parts. First, the nodal displacements in x-direction are transferred such that the deformation gradient components depending on these displacements can already be computed. Subsequently the procedure is repeated for the nodal displacements in y-direction. Similarly, the computation of the internal forces is split into two parts to overlap the computation of the internal forces with transferring the memory toward the CPU. The instruction schedule contains two parts, one for computing the components of $\mathbf{F}$ and one for computing the components of the internal forces. Note that in the bottom schedule only six instead of twelve components are computed compared to the previous implementation. Important to highlight is the fact that the computation of internal forces in x-direction can only start once the components of $\mathbf{F}$ that depend on the nodal displacements in y-direction have been computed. Consequently, there is a gap present in the top stream in the event schedule.

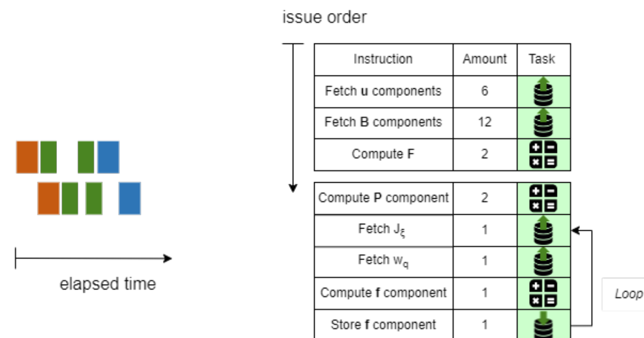


Figure 8.9: *The event and instruction schedule for the concurrent reduced memory implementation. The transfer of the nodal displacements is split into two parts. First, the nodal displacements in x-direction are transferred such that the deformation gradient components depending on these displacements can already be computed. Subsequently the procedure is repeated for the nodal displacements in y-direction. Similarly, the computation of the internal forces is split into two parts to overlap the computation of the internal forces with transferring the memory toward the CPU. The instruction schedule contains two parts, one for computing the components of $\mathbf{F}$ and one for computing the components of the internal forces.*

Implementation results

Examining the results for the displacement to internal force strategy depicted in Figure 8.10, several observations can be made. First, the gain in performance compared to the strain to stress strategy is increased significantly. The total computation speed-up is approximately 32.5×, indicating that the significantly larger amount of computations is beneficial for the GPU implementation. Furthermore, the concept to reduce idle GPU time is not as effective as for the strain to stress strategy, with a total computation speed-up of only 4.06× due to its larger memory demand compared to the other GPU implementations. Comparing the concurrent and reduced memory implementations, it becomes evident that overlapping events does not yield a gain in performance for the displacement to internal force strategy. The separate computations of the deformation gradient components results in double the amount of global memory transfers, resulting in a significantly slower processing event. As the computation of the stress components only commences once all deformation gradient components have been computed, the concurrent implementation is slower than the reduced memory implementation with a total computation speed-up of 24.3× compared to 32.5× for the reduced memory implementation. The computing speed-up levels off at approximately 89.4× for the reduced memory implementation and 31.5× for the concurrent implementation. In between lies the reducing idle GPU time implementation, showing a computing speed-up of 72.5×.

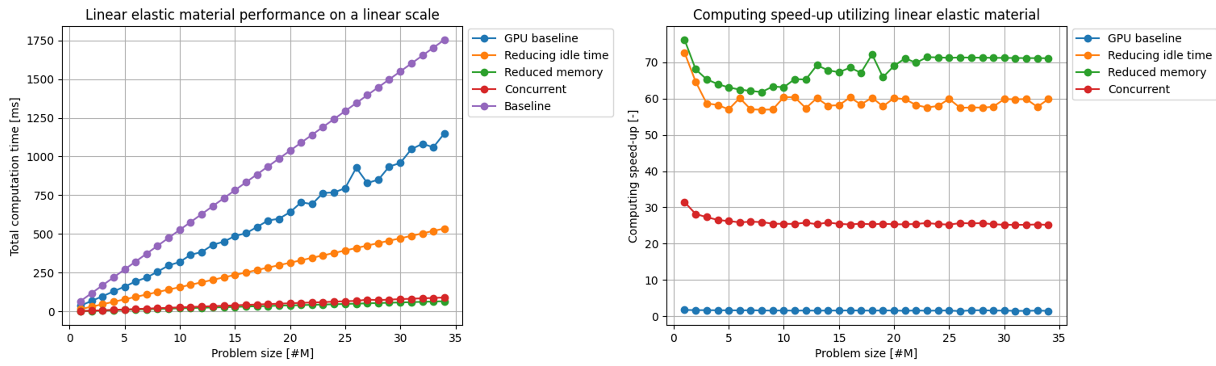


Figure 8.10: *Performance of the linear elastic model implementations for the displacement to internal force strategy. The total computation speed-up is approximately* $32.5\times$ *for the reduced memory implementation, while the concurrent implementation exhibits a total computation speed-up of* $24.3\times$*. Furthermore, the concept to reduce idle GPU time is not as effective as for the strain to stress strategy, with a total computation speed-up of* $4.06\times$*. The computing speed-up levels off at approximately* $89.4\times$ *for the reduced memory implementation and* $31.5\times$ *for the concurrent implementation. In between lies the reducing idle GPU time implementation, showing a computing speed-up of* $72.5\times$*.*

8.2 The Neo-Hookean model

In continuation of the previous section on the linear elastic model, this section focuses on the optimization specifically for the Neo-Hookean material model. The results of the strain to stress parallelization strategy are presented in subsection 8.2.1. The different implementations that were tested are explained by discussing the event and instruction schedules. Afterwards, the displacement to internal force parallelization strategy is discussed only in terms of results in subsection 8.2.2 as the tested implementations are the same as for the linear elastic material model.

8.2.1 Strain to stress parallelization strategy

For the Neo-Hookean material model, a distinct approach is adopted compared to the linear elastic model. Several important distinctions arise due to the nonlinear nature of the material model. Firstly, the computational workload increases significantly, necessitating a higher number of computations compared to the linear elastic model. Moreover, the stress tensor $\mathbf{P}$ is no longer symmetric, which leads to an increase in memory transfers. Given the different computational and memory transaction requirements compared to the linear elastic model, a new sensitivity analysis is performed to ascertain the optimal grid size. The outcome of this analysis is depicted in Figure 8.11.

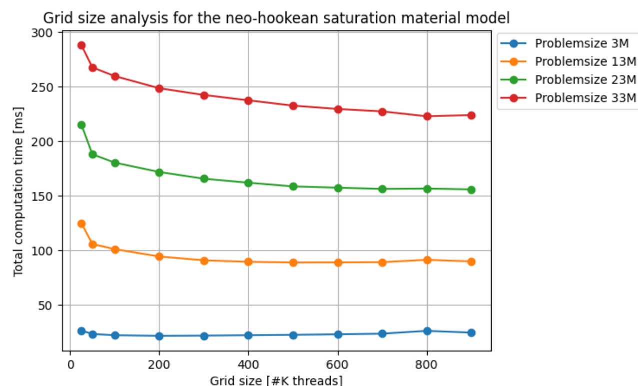


Figure 8.11: *The Neo-Hookean model performance for different grid sizes across different problem sizes. The gain in performance for larger grid sizes is considerably larger with respect to the smallest grid size. It is determined that a grid size consisting of* $800K$ *threads yields the most optimal performance for the Neo-Hookean constitutive model.*

Compared to the linear elastic material model the gain in performance for smaller grid sizes is considerably larger with respect to the smallest grid size. It is determined that a grid size consisting of $800K$ threads yields the most optimal performance for the Neo-Hookean constitutive model. Continuing, the three different implementations that were tested are briefly described. For each implementation the input to the GPU is the deformation gradient tensor, $\mathbf{F}$. The computed result returned to the CPU is the first Piola-Kirchhoff stress tensor, $\mathbf{P}$. For the Neo-Hookean material model the computation of $\mathbf{P}$ involves the determinant of $\mathbf{F}$, rendering it impossible to overlap memory transfers from and toward the GPU as all components of $\mathbf{F}$ are coupled with all components of $\mathbf{P}$. As a result, all memory transfers toward the GPU must be completed before any component of $\mathbf{P}$ can be computed. Similar as for the linear elastic material model a GPU baseline implementation utilizing unified global memory and a strided memory layout is used.

Reducing idle GPU time implementation

The goal of the first implementation is to reduce idle GPU time. The event and instruction schedule are depicted in Figure 8.12. The implementation consists of only a single memory transfer in each direction for each loop. The instruction schedule highlights the increased complexity compared to the computation of the linear elastic material model. After fetching the components of $\mathbf{F}$, a significantly larger number of computations has te be evaluated before the stress can be computed. Based on $\mathbf{F}$ the right Cauchy-Green strain tensor, $\mathbf{C}$, is computed and subsequently its inverse. Next, the second Piola-Kirchhoff stress, $\mathbf{S}$ is computed. Since $\mathbf{S}$ is a symmetric tensor, a potential avenue to explore is to transfer $\mathbf{S}$ directly to the CPU instead of converting it to $\mathbf{P}$ and transfer $\mathbf{P}$ toward the CPU. In the first case, only six tensor components need to be transferred to the CPU, compared to nine tensor components that need to be send in case $\mathbf{P}$ is transferred to the CPU. However, results for the linear elastic model have shown that a larger number of memory transfers takes a smaller amount of time compared to letting the CPU perform additional calculations. Consequently, this avenue is not explored further. Therefore, $\mathbf{S}$ must be converted to $\mathbf{P}$ by multiplication with $\mathbf{F}$ before the results are stored in global memory and subsequently returned towards the CPU.

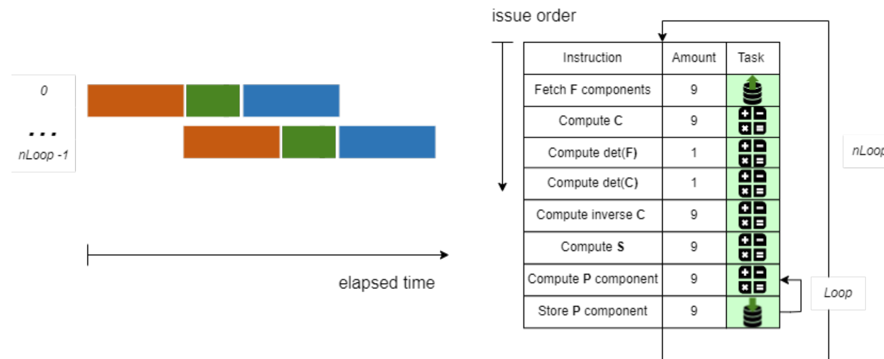


Figure 8.12: *The event and instruction schedule for the reducing idle GPU time implementation. The implementation consists of only a single memory transfer in each direction for each loop. The instruction schedule highlights the increased complexity compared to the computation of the linear elastic material model. After fetching the components of* $\mathbf{F}$*, a significantly larger number of computations has to be evaluated before the stress can be computed. Based on* $\mathbf{F}$ *the right Cauchy-Green strain tensor,* $\mathbf{C}$*, is computed and subsequently its inverse. Next, the second Piola-Kirchhoff stress,* $\mathbf{S}$*, is computed and converted to* $\mathbf{P}$ *by multiplication with* $\mathbf{F}$*. Subsequently, the results are stored in global memory and returned towards the CPU.*

Simplified computation implementation

Turning to the second implementation, as the input consists of $\mathbf{F}$, the calculation of $\mathbf{C}$ is employed on the GPU. However, multiple redundant computations are conducted during the processing event as the symmetry of $\mathbf{C}$ and $\mathbf{S}$ is not utilized. The revised and reduced number of tasks due to these simplifications are illustrated in Figure 8.13. Omitting redundant computations and minimizing the number of global memory transfers is expected to enhance performance of the processing event.

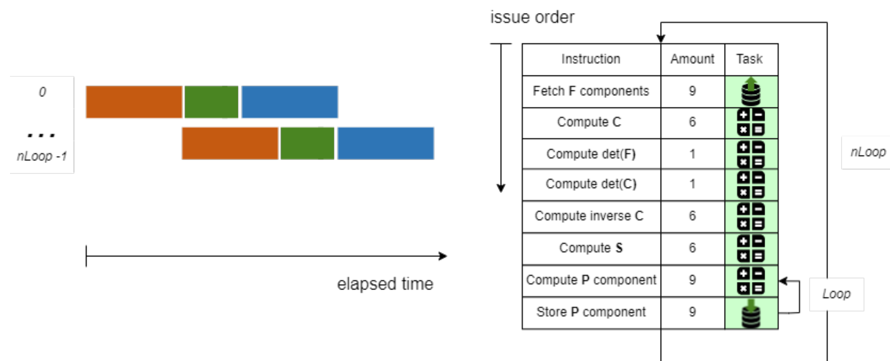


Figure 8.13: *The event and instruction schedule for the simplified implementation. Note that the event schedule remains the same as in Figure 8.12, while the number of tasks is slightly decreased in the instruction schedule. Specifically, the computations for* **C** *and* **S** *have been reduced.*

Splitted implementation

The final implementations explores a similar concept to the linear elastic model by trying to split up the computation in as many concurrent tasks as possible. The dissection of the processing event significantly alters both the event and instruction schedule, increasing their complexity significantly. Therefore the event and instruction schedule have been omitted for this implementation. Splitting up the interaction and processing events significantly increases the number of events that can occur simultaneously. However, the number of events occurring concurrently remains limited, which is highlighted by the fact that the components of **P** are only computed after the last components of **F** has been sent toward the GPU. The goal of this implementation is to explore whether splitting up the computing process across multiple events and instruction schedules will result in an increase in performance.

Implementation results

Upon examining the results presented in Figure 8.14, several observations can be stated. First, the gain in performance achieved by the baseline implementation is significantly larger compared to the gain that was achieved for the linear elastic material model. The GPU baseline implementation exhibits a total computation speed-up of approximately 3.11×, which increases slightly further to 3.82× for the splitted implementation. The implementation reducing the idle GPU time and the simplified computation exhibit similar performance with a total computation speed-up of approximately 7.82× and 7.94× respectively. Turning to the computing speed-up, it can be seen that the splitted implementation shows a considerably lower computing speed-up due to the large amount of different instruction schedules, necessitating a significantly larger amount of global memory transfers. As a result, the computing speed-up converges to a value of 11.4×, while the simplified computation and reducing idle GPU time implementations demonstrate a computing speed-up of 66.5×. Furthermore, it can be observed that all three implementations show a larger speed-up for small problem sizes. This can be related to the number of loops that increases with the problem size and thus increases the number of instruction schedules executed.

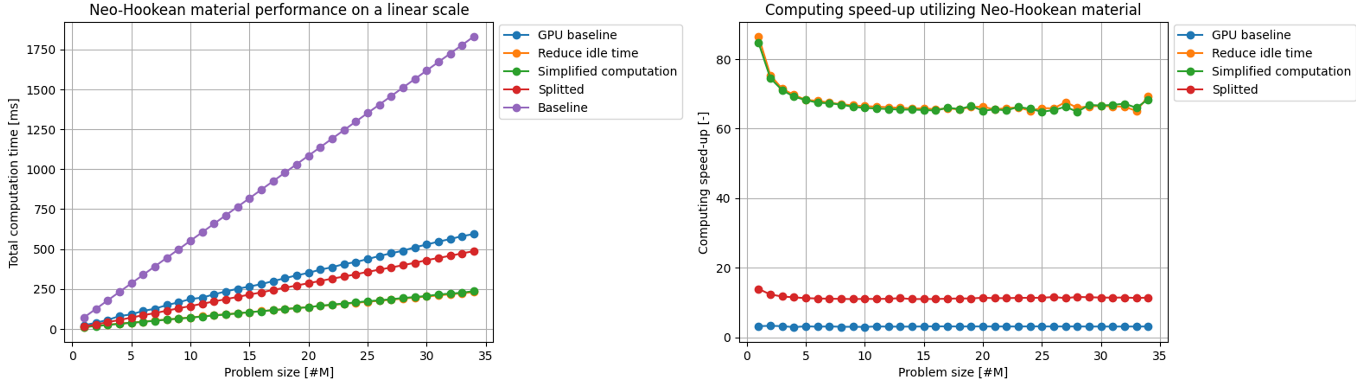


Figure 8.14: *Performance of the NeoHookean material model specific implementations. The GPU baseline implementation exhibits a total computation speed-up of approximately* $3.11\times$*, which increases slightly further to* $3.82\times$ *for the splitted implementation. The implementation reducing the idle GPU time and the simplified computation exhibit similar performance with a total computation speed-up of approximately* $7.82\times$ *and* $7.94\times$ *respectively. Turning to the computing speed-up, it can be seen that the splitted implementation shows a considerably lower computing speed-up, converging to a value of* $11.4\times$*, while the simplified computation and reducing idle GPU time implementations demonstrate a computing speed-up of* $66.5\times$*.*

8.2.2 Displacement to Internal force implementation

For the displacement to internal force strategy, the same GPU implementations as for the linear elastic material model are examined. The reason to keep the evaluated implementations the same is the fact that the additional steps that are present for the displacement to internal force strategy are agnostic with respect to the material model that is utilized. Consequently, the reader is referred to section 8.1 for more information on the utilized GPU implementations.

A sensitivity analysis was performed to determine the optimal grid size for the Neo-Hookean material model for the displacement to internal force strategy and is depicted in Figure 8.15. Compared to the strain to stress strategy, there is no increase in computing time for larger grid sizes. The optimal grid size is determined to contain $600K$ threads as the increase in performance for larger grid sizes is negligible, while smaller grid sizes result in a lower performance for larger problem sizes.

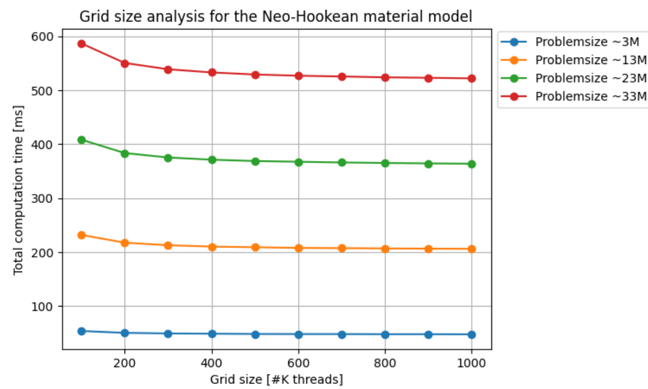


Figure 8.15: *The Neo-Hookean model results for different grid sizes across different problem sizes for the displacement to internal force strategy. Compared to the strain to stress strategy, there is no increase in computing time for larger grid sizes. The optimal grid size is determined to contain* $600K$ *threads as the increase in performance for larger grid sizes is negligible, while smaller grid sizes result in a lower performance for larger problem sizes.*

Looking at the results for the different implementations in Figure 8.16, the following observations can be made. First, the GPU baseline implementation exhibits a total computation speed-up of approximately

2.30×, which increases slightly further to 4.64× for the reducing idle GPU time implementation. The implementation reducing memory transfers and the concurrent implementation perform better with a total computation speed-up of approximately 31.7× and 25.9× respectively. Turning to the computing speed-up, it can be seen that the concurrent implementation shows a considerably lower computing speed-up, converging to a value of 33.8×, while the reducing idle GPU time and reduced memory implementation demonstrate a computing speed-up of approximately 74.4× and 73.9× respectively. The lower performance of the concurrent implementation is attributed to the same reason as for the linear elastic material model, which is a larger number of global memory transfers.

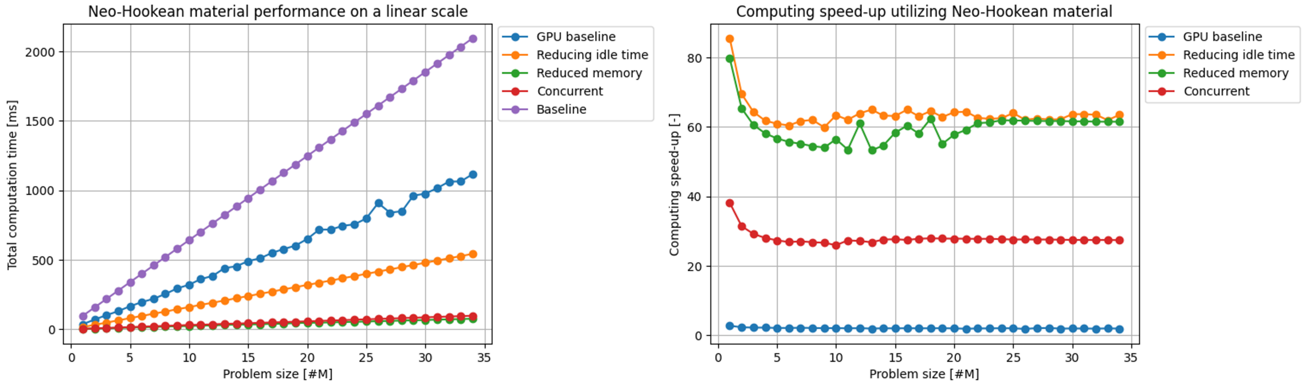


Figure 8.16: *Performance of the NeoHookean material model specific implementations for the displacement to internal force strategy. The GPU baseline implementation exhibits a total computation speed-up of approximately* 2.30×*, which increases slightly further to* 4.64× *for the reducing idle GPU time implementation. The implementation reducing memory transfers and the concurrent implementation exhibit similar performance with a total computation speed-up of approximately* 31.7× *and* 25.9× *respectively. Turning to the computing speed-up, it can be seen that the concurrent implementation shows a considerably lower computing speed-up, converging to a value of* 33.8×*, while the reducing idle GPU time and reduced memory implementation both demonstrate a computing speed-up of approximately* 74.4× *and* 73.9× *respectively.*

8.3 The orthotropic model

The last material that is discussed is the orthotropic material model. The strain to stress parallelization strategy is examined in subsection 8.3.1, presenting the different implementations that were explored to increase the performance of the GPU implementation. Subsequently, the results for the displacement to internal force parallelization strategy are discussed in subsection 8.3.2. Note that, similar to the Neo-Hookean material model, the explored implementations for the displacement to internal force strategy are not explained when they are the same for each material model.

8.3.1 Strain to stress parallelization strategy

In the case of the orthotropic material model, a similar approach was undertaken as in the analysis of the Neo-Hookean material model. Given the small deformations typically associated with composites, a linear strain assumption, akin to the linear elastic model, was adopted. Consequently, the computation entails a matrix-vector multiplication involving the stiffness matrix of the composite laminate and the linear strain vector as explained in chapter 5. The performed sensitivity analysis for the concept of reducing idle GPU time results in Figure 8.17 for the orthotropic material model. Particularly noteworthy is that, compared to the other material models for the strain to stress strategy, the initial descent, or increase in performance, is larger for the orthotropic material model. The optimal grid size was determined to consist of $400K$ threads.

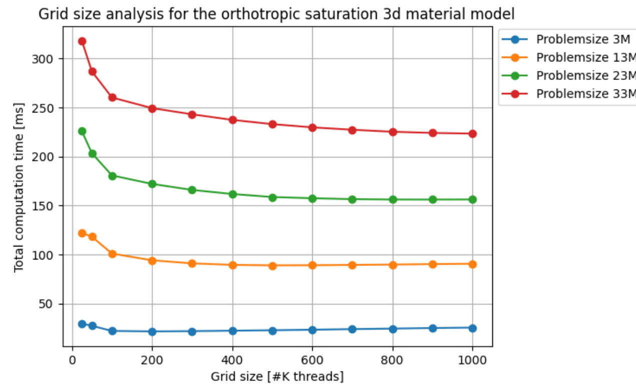


Figure 8.17: *The 3D orthotropic model performance for different grid sizes across different problem sizes. Compared to the other material models for the strain to stress strategy, the initial descent, or increase in performance, is larger for the orthotropic material model. The optimal grid size was determined to consist of* $400K$ *threads.*

Subsequently, the three implementations that were tested for the orthotropic material model are explained. For each implementation the input data towards the GPU is the deformation gradient tensor, $\mathbf{F}$, and the returned result towards the CPU is the first Piola-Kirchhoff stress tensor, $\mathbf{P}$. Similar as for the displacement to internal force strategy in general, several memory transfers are only made once during the entire simulation. In case of the orthotropic material model this entails the material properties, which is not taken into account in the total computing time for similar reasons stated before. Finally, note that again a baseline implementation utilizing unified global memory and a strided memory layout is used.

Reducing idle GPU time implementation

Similar as for the Neo-Hookean material model, the first implementation of the orthotropic material model focuses on reducing the idle GPU time. In terms of the number of tasks in the processing event, the orthotropic material model falls in between the linear elastic and Neo-Hookean material models. The event schedule and instruction schedule are depicted in Figure 8.18. The event schedule consists of a single memory transfer followed by a processing event and another memory transfer. This order is looped utilizing the optimal grid size until the entire problem size is covered. Moving on to the instruction schedule, first the material properties for the lamina are fetched from global memory to compute the stiffness of the laminate. Next, $\mathbf{F}$ is fetched from global memory such that the strain and subsequently the stress can be computed which is then stored in global memory.

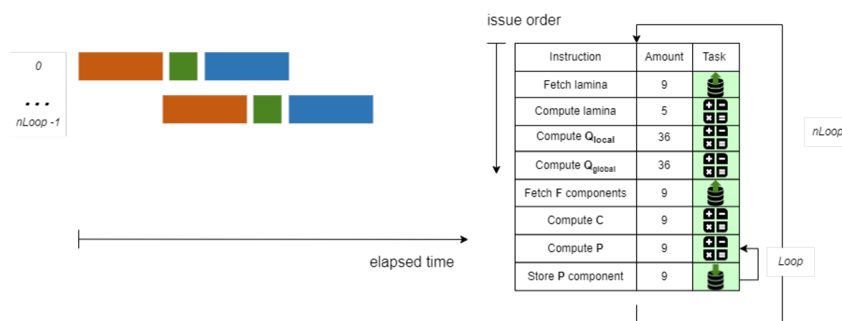


Figure 8.18: *The event and instruction schedule for the reducing idle GPU time implementation. The event schedule consists of a single memory transfer followed by a processing event and another memory transfer. This order is looped utilizing the optimal grid size until the entire problem size is covered. Moving on to the instruction schedule, first the material properties for the lamina are fetched from global memory to compute the stiffness of the laminate. Next,* $\mathbf{F}$ *is fetched from global memory such that the strain and subsequently the stress can be computed which is then stored in global memory.*

Symmetry implementation

Similar as for the linear elastic model, the stress symmetry implementation leverages the symmetry of the Lagrangian stress tensor due to the linear strain. Consequently, the amount of memory transferred back toward the CPU decreases. Note that the computation remains in a single stream, resulting in the event order as depicted in Figure 8.19. However, the gain in computing time only holds for the last loop since the amount of memory transferred towards the GPU is too large to have increased performance for each loop. Additionally, symmetry is utilized in the computation of the local and global stiffness matrices of the laminate, resulting in a considerably lower amount of tasks in the processing event. Instead of 36 computations for both the local stiffness and global stiffness matrix of the laminate, only 9 computations for the local and 13 computations for the global stiffness matrix have to be performed.

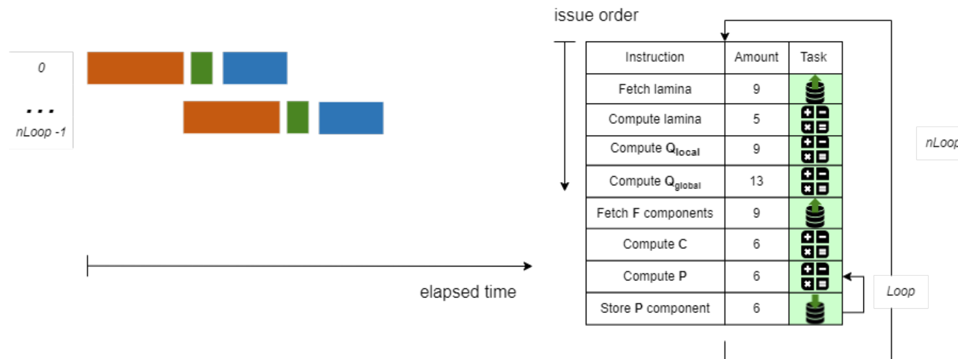


Figure 8.19: *The event and instruction schedule for the symmetry implementation. Leveraging the symmetry of* $\mathbf{P}$*, the amount of memory transferred back to the CPU is decreased. However, the gain in computing time only holds for the last loop since the amount of memory transferred towards the GPU is too large to have increased performance for each loop. Additionally, symmetry is utilized in the computation of the local and global stiffness matrices of the laminate, resulting in a considerably lower amount of tasks in the processing event. Instead of* 36 *computations for both the local stiffness and global stiffness matrix of the laminate, only* 9 *computations for the local and* 13 *computations for the global stiffness matrix have to be performed.*

Splitted implementation

The splitted implementation of the orthotropic material model aims to exploit the strain dependencies present in the computation of the stress tensor, $\mathbf{P}$. Examining Equation 5.18, two observations can be made. First, it becomes apparent that the stiffness matrix can be divided into two distinct parts: one for the axial components and in-plane shear component, and the other for the out-of-plane shear components. Moreover, the computation of each stress component comes down on a summation of the row values of the stiffness that are non-zero multiplied with a strain component. By splitting the matrix-vector product into its individual components. Each component is computed separately as soon as its corresponding strain component is transferred to the GPU as depicted in the event schedule in Figure 8.20.

The out-of-plane shear components can be returned earlier, as these only depend on four components of $\mathbf{F}$. For the remaining components of $\mathbf{P}$ two instruction schedules are executed. For the axial strain components only a single component of $\mathbf{F}$ is needed to update the stress component. For the in-plane shear component two components of $\mathbf{F}$ are required. After summing al four contributions, the last four components of $\mathbf{P}$ can be returned towards the CPU. This implementation explores whether splitting the different strain dependencies over different streams results in the optimal concurrency for the memory transfers. Additionally, it is investigated whether a significantly larger number of executed kernels in combination with a lower computational workload per kernel results in an increase in performance.

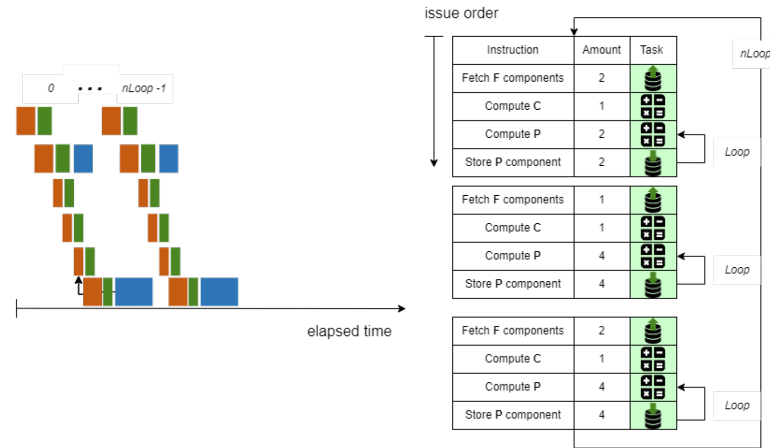


Figure 8.20: *The event and instruction schedule for the splitted implementation. The computation of each stress component comes down on a summation of the row values of the stiffness that are non-zero multiplied with a strain component. Consequently, the computation of each stress component can be split up such that all computations are performed as soon as the necessary component of* **F** *is present on the GPU. The out-of-plane shear components can be returned earlier, as these only depend on four components of* **F**. *For the remaining components of* **P** *two instruction schedules are executed. For the axial strain components only a single component of* **F** *is needed to update the stress component. For the in-plane shear component two components of* **F** *are required. After summing al four contributions, the last four components of* **P** *can be returned towards the CPU. This implementation explores whether splitting the different strain dependencies over different streams results in the optimal concurrency for the memory transfers.*

Finally, note that the computation of the local and global stiffness matrix of the laminate no longer occurs on the GPU. Since the computation only needs to be executed once at the beginning of the simulation, the influence on the total performance is negligible when distributed over multiple time steps. Consequently, the computation and corresponding memory transfer to the GPU have been removed from the instruction schedule.

Implementation results

The results for the orthotropic material model are presented in Figure 8.21. The findings indicate that utilizing a GPU, even only the simple baseline implementation, already results in a speed-up of 4.86×. Reducing idle GPU time further enhances performance to yield a speed-up of 9.57×, which is the largest gain achieved by the reducing idle GPU time implementation. Subsequently, utilizing the stress symmetry and overlapping memory transfers with kernel executions result in a decrease in total computation time. The speed-up for the symmetry implementation is 10.2× while the splitted implementation reaches 12.2×. Considering the computing speed-up, it is evident that utilizing the symmetry in the computations of the laminate stiffness and the computations of the stress tensor result in a significant increase of computing speed-up to approximately 205×. Furthermore, the larger number of kernel executions in the splitted implementation results in more overhead and thus a lower computing speed-up of approximately 56.5×. The same argument can be made for the reducing idle GPU time implementation, which shows a computing speed-up of approximately 48.6×.

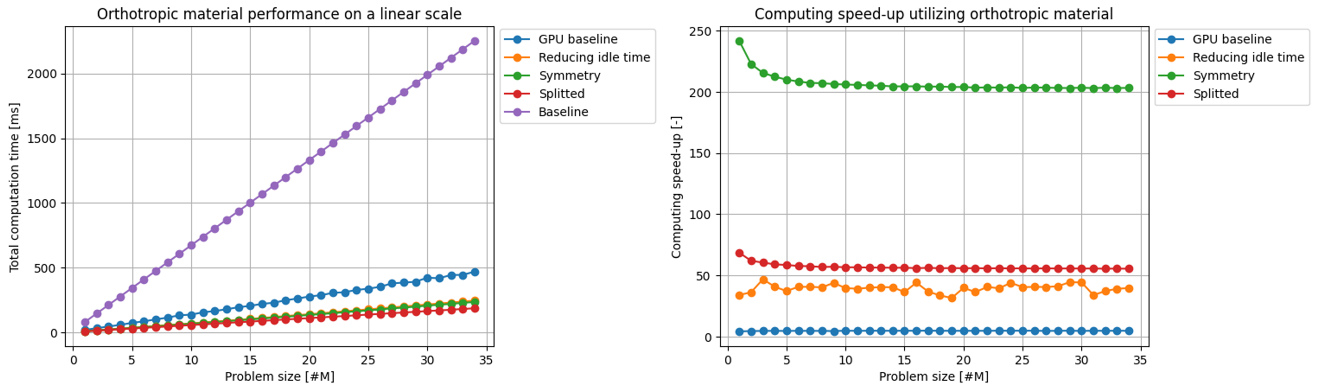


Figure 8.21: *Performance of the orthotropic model implementations. The findings indicate that the utilizing a GPU results in a speed-up of* $4.86\times$ *for the baseline implementation. Reducing idle GPU time further enhances performance to yield a speed-up of* $9.57\times$. *Subsequently, utilizing the stress symmetry and overlapping memory transfers and kernel executions result in a decrease in total computation time. The speed-up for the symmetry implementation is* $10.2\times$ *while the splitted implementation reaches* $12.2\times$. *Considering the computing speed-up, the symmetry implementation yields a significant increase in computing speed-up to approximately* $205\times$. *Furthermore, the larger number of kernel executions in the splitted implementation results in more overhead and thus a lower computing speed-up of approximately* $56.5\times$. *The same argument can be made for the reducing idle GPU time implementation, which shows a computing speed-up of approximately* $48.6\times$.

8.3.2 Displacement to internal force parallelization strategy

Moving on to the displacement to internal forces strategy, the results of the sensitivity analysis are depicted in Figure 8.22. Comparing to Figure 8.17, the behavior for the different parallelization strategies is almost identical, except for the steeper initial decay for the displacement to internal force strategy. As a result, the optimal result for the displacement to internal force strategy is also obtained with a grid size of $400K$ threads.

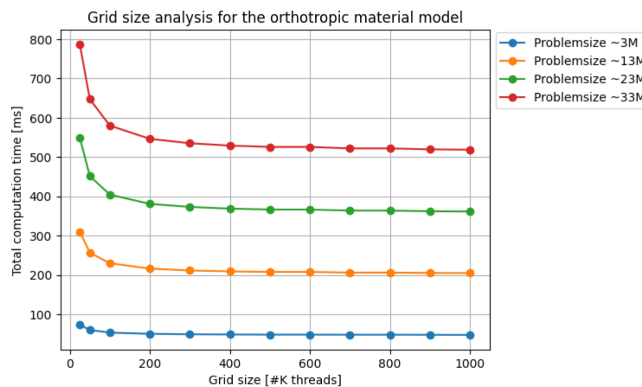


Figure 8.22: *The 3D orthotropic model results for different grid sizes across different problem sizes for the displacement to internal force strategy. The behavior for the different parallelization strategies is almost identical, except for the steeper initial decay for the displacement to internal force strategy. As a result, the optimal result for the displacement to internal force strategy is also obtained with a grid size of* $400K$ *threads.*

The performance results for the displacement to internal force strategy are depicted in Figure 8.23. The baseline GPU implementation shows a lower total computation speed-up of $2.36\times$, which is considerably lower than the speed-up that was obtained by the GPU baseline implementation in the strain to stress strategy. Reducing the idle GPU time, the total computation speed-up increases to $4.58\times$ and the computing speed-up levels of at approximately $81.6\times$. Reducing memory transfers in the reduced memory

implementation results in a total computation speed-up of 37.7× and a computing speed-up that levels off at 109×. Overlapping memory transfers and kernel executions yields a total computation speed-up of 28.8×, while the computing speed-up is equal to 37.9×.

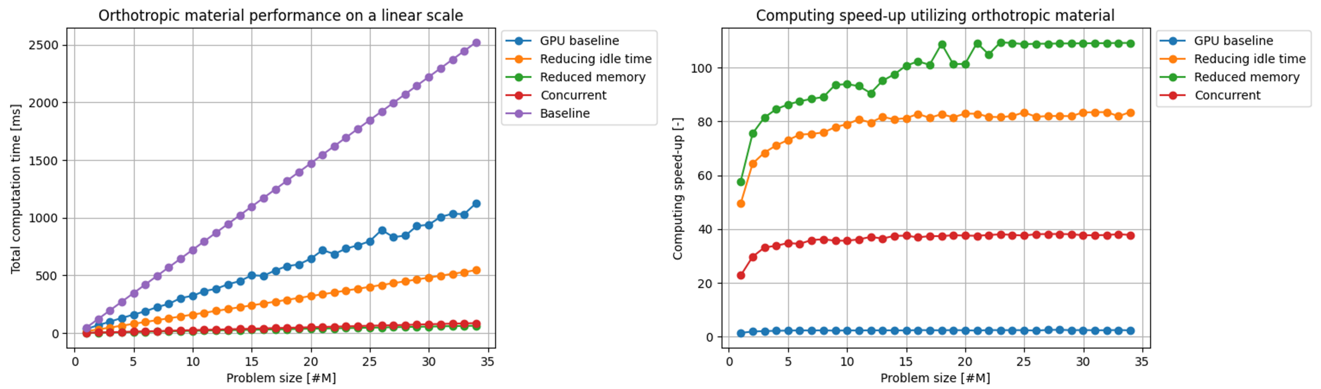


Figure 8.23: *Performance of the orthotropic model implementations. The baseline GPU implementation shows a lower total computation speed-up of* 2.36×*, which is considerably lower than the speed-up that was obtained by the GPU baseline implementation in the strain to stress strategy. Reducing the idle GPU time, the total computation speed-up increases to* 4.58× *and the computing speed-up levels of at approximately* 81.6×*. Reducing memory transfers in the reduced memory implementation results in a total computation speed-up of* 37.7× *and a computing speed-up that levels off at* 109×*. Overlapping memory transfers and kernel executions yields a total computation speed-up of* 28.8*, while the computing speed-up is equal to* 37.9×*.*

Compared to the reduced memory implementation, the concurrent implementation yields a slight decrease in both performance metrics. It becomes apparent that utilizing more kernels results in too much overhead, decreasing the computing speed-up and consequently the total computation speed-up as well. The reduced memory implementation results in a considerable increase in total computation speed-up compared to the reducing idle GPU time implementation, which was seen for both the linear elastic and Neo-Hookean material model as well. Finally, note that the reduced memory implementation computing speed up keeps increasing for a larger range of problem sizes compared to the other implementations. This makes sense as for each problem size only one kernel is executed, resulting in minimal overhead. As there is no idle time to reduce since there are only two memory transfers in the implementation, the only limiting factor on the computing speed-up are the GPU resources. If the GPU resources are fully utilized during the computational process, the computing speed-up can not increase further. This cross-over point is reached at a problem size of approximately $23M$.

8.4 Conclusion

Based on the results in previous sections, the most efficient implementation for each material model within the considered parallelization strategies can be determined. This section concludes on the results that have been obtained for each material model in the different parallelization strategies. Furthermore, a summary of the optimal performance is given in Table 8.1 for each material model in the different parallelization strategies.

Based on the two parallelization strategies and three material models that have been examined, there are several general conclusions that can be drawn. These are summarized below:

- The amount of memory transfers should be limited as much as possible, as memory transfers are taking up more than 70% of the total computation time. Overlapping memory transfers with computations by utilizing streams has exhibited the ability to hide the latency considerably. On the other hand, referring to the double symmetry implementation for the linear elastic material, performing portions of the implementation on the CPU to decrease the size of memory transfers has proven not to be beneficial for the total computation time. The CPU time for performing the computations increases the total computation time more than the memory transfers.

- A distinction can be made between *memory-driven* and *computation-driven* implementations. The performance of *memory-driven* implementations is heavily reliant on the amount of memory transfers that are necessary. It is observed that there are too many memory transfers to overlap all with computations as the amount of computations is not large enough for these implementations. In case of *computation-driven* implementations, either the computations take too long to be overlapped with memory transfers, or there exists a coupling between model parameters that results in computations that must be executed sequentially.

Apart from general conclusions, some observations were made specifically for the different parallelization strategies.

- For the strain to stress strategy, it is evident that the reducing idle GPU time implementation results in the most significant gain in performance. Although other implementations yield a larger gain in computing speed-up, reducing idle GPU time implementation focuses solely on overlapping as many memory transfers with computations. Since the strain to stress strategy is *memory-driven*, this approach leads to the largest gain in the total computation speed-up.
- It has been discovered that overlapping memory transfers and calculations are detrimental to performance in the displacement to internal force strategy. Because of the overlap, the number of global memory transfers increases, lowering performance.
- The coupling between nodal displacement and deformation gradient components results in a necessary sequential order of a part of the computation. Hence, the displacement to internal force strategy is *computation-driven* and a sequential order without utilizing streams shows the best performance. This implementation exhibited the best performance for each material model, as can be seen in Table 8.1, and does not build further upon the best performing model-agnostic GPU implementation.

Continuing with the different material models for the strain to stress strategy, the optimal performing implementation for each material model in each parallelization strategy is briefly summarized:

- The optimal ipmlementation for the linear elastic material model in the strain to stress strategy builds further on the model agnostic implementation by applying the symmetry of the stress tensor and splitting of the diagonal stress component calculation across different streams. To reduce idle GPU time, a grid size of $400K$ threads is utilized.
- In the context of the Neo-Hookean material model in the strain to stress strategy, the best performing implementation turned out to be a combination of the reducing idle GPU time implementation for small problem sizes and the simplified implementation for larger problem sizes. Splitting up the constitutive model is decreasing performance rather than enhancing it, as all components of $\mathbf{F}$ are coupled with all components of $\mathbf{P}$. The optimal grid size was found to contain $800K$ threads.
- Concluding on the orthotropic material model utilized in the strain the stress strategy, the results indicate that the orthotropic material model presents a combination of the optimization challenges presented by the linear elastic and Neo-Hookean material models. On one hand the symmetry of the stress tensor due to linear strain can be utilized to enhance performance. On the other hand, the internal couling of some strain tensor components limit the overlap of different memory transfers. In the end, the best performing implementation was the splitted implementation, utilizing a grid size of $400K$ threads.
- For the displacement to internal force strategy, the optimal GPU implementation is the reduced memory implementation for each material model. The enormous decrease in memory demand results in a significantly faster computational process and thus a larger total computation speed-up.

Similar verification as in chapter 7 has been applied. The obtained results were again equivalent to the CPU implementation, verifying that the results computed by the GPU-accelerated implementation are correct.

Finally, Table 8.1 summarizes the results of the model-specific optimization. For each implementation it becomes clear that the computing speed-up is significantly larger compared to the total computation speed-up. The strain to stress parallelization strategy shows a total computation speed-up ranging between $5.76x$ and $12.2x$, while the range for the displacement to force parallelization strategy is $31.7x$ to $37.7x$. In the strain to stress strategy, the computing speed-up ranges between $56.5x$ for the orthotropic material model and $86.3x$ for the linear material model. For the displacement to internal force strategy this range starts at $73.9x$ up to $109x$.

Material	**Parallelization strategy**	**Implementation type**	**Computing speed-up**	**Total computation speed-up**
Linear elastic	Strain to stress	Stress symmetry	$86.3x$	$5.76x$
	Displacement to force	Reduced memory	$89.4x$	$32.5x$
Neo-Hookean	Strain to stress	Symmetry computation	$66.5x$	$7.94x$
	Displacement to force	Reduced memory	$73.9x$	$31.7x$
Orthotropic	Strain to stress	Stress symmetry	$56.5x$	$12.2x$
	Displacement to force	Reduced memory	$109x$	$37.7x$

Table 8.1: *Performance summary of the model-specific optimization. The strain to stress parallelization strategy shows a total computation speed-up ranging between* $5.76x$ *and* $12.2x$*, while the range for the displacement to force parallelization strategy is* $31.7x$ *to* $37.7x$*. The computing speed-up ranges between* $56.5x$ *for the orthotropic material model and* $86.3x$ *for the linear material model in the strain to stress strategy. For the displacement to internal force strategy this range starts at* $73.9x$ *up to* $109x$*.*

9 Integration in a Finite Element Framework

Building upon the specific optimizations targeting each constitutive model in chapter 8, this chapter proceeds with the integration of the GPU-accelerated constitutive update within a Finite Element Framework. The test case, formulated to facilitate a comprehensive comparison between the CPU and GPU-accelerated implementations, is presented in detail in section 9.1. Subsequently, the results of the integration within a single-GPU framework is examined in section 9.2. Furthermore, the integration is expanded to encompass a multi-GPU implementation, exploring the potential performance gains achieved by increasing the number of GPUs utilized in section 9.3. Finally, the findings and implications drawn from the integration process are consolidated and summarized in section 9.4.

9.1 Testcase setup

The exemplary test case encompasses a 3D rectangular beam that undergoes elongation under a uniform strain rate, while being clamped at one end. Along the longitudinal outer boundary of the beam Dirichlet boundary conditions are prescribed, meaning that there the displacement value is fixed, while the lateral boundaries are stress free. Based on the value of the displacement and the current time step, the velocity and acceleration of the nodes belonging to the outer boundary are determined. In the interior of the beam the displacement is obtained through the finite element formulation as described in chapter 4.

The beam is discretized using first order tetrahedral elements. The mesh size is adjusted by employing a subdivision algorithm as depicted below in Figure 9.1 [48]. Along the edge of each tetrahedron the midpoint of the edge is utilized as the new corner for a smaller tetrahedron. As a result, each large tetrahedron element is partitioned into eight smaller tetrahedra. Note that this procedure can be recursively performed an arbitrary number of times, thus enabling the evaluation of the CPU and GPU implementations' efficiency across different mesh sizes.

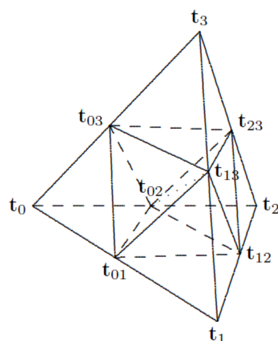


Figure 9.1: *Subdivision of a tetrahedral element. Along the edge of each tetrahedron the midpoint of the edge is utilized as the new corner for a smaller tetrahedron. As a result, each large tetrahedron element is partitioned into eight smaller tetrahedra. Figure credits: [48].*

To visually portray the test case, refer to the graphical representation presented in Figure 9.2. Furthermore, the impact of augmenting the total simulation time by incorporating additional time steps is explored by varying the number of time steps utilized in the analysis.

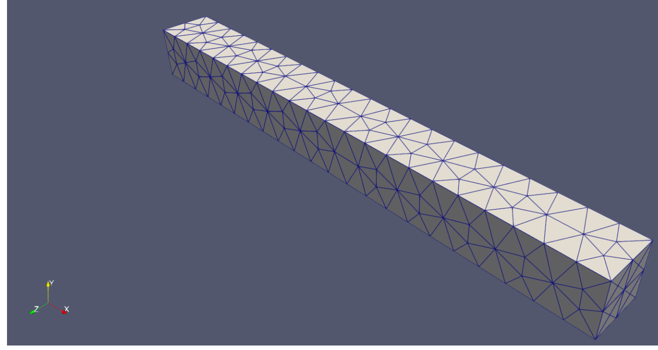


Figure 9.2: *A visualization of the test case in Paraview. A rectangular beam is partitioned in multiple tetrahedral elements, visible as triangular edges along the outer contour of the beam. Along the outer longitudinal boundary of the beam Dirichlet boundary conditions are prescribed, while the lateral boundaries are stress free.*

The different mesh sizes that are considered in this study are depicted below in Table 9.1. As each tetrahedron is subdivided in eight subtetrahedra, the number of elements increases exponentially per subdivision. The nodal distance decreases as the mesh is subdivided further. The considered average nodal distance is relevant for a realistic solid mechanics application.

Mesh subdivision	Number of elements	Average nodal distance [mm]
0	694	9.87
1	5552	4.93
2	44416	2.47
3	355328	1.23
4	2842624	0.61

Table 9.1: *Information regarding the mesh for the utilized test case. The number of elements in the mesh increases exponentially by the applied subdivision algorithm. The considered average nodal distance is relevant for a realistic solid mechanics application.*

The performance evaluation of the GPU-accelerated implementation encompasses two key metrics. These are described as follows:

- **Simulation wall time**
  The choice for the simulation wall time is motivated by its ability to reflect the overall speed-up achieved of the simulation duration. This metric accounts for the performance impact resulting from adjustments made to the memory layout pattern, including any associated effects on methods outside the portion of the code affected by the parallelization strategy. The wall time measurement is started just before the computation of the first time step commences. As a result, the generation of the mesh, loading the material properties and setting up the initial and boundary conditions is not taken into account in the simulation wall time. Once the calculation of the last time step has finished the wall time measurement is stopped.

- **Simulation constitutive time**
  The constitutive time is employed as a performance metric to specifically assess the efficiency of the parallelization strategy in isolation. By focusing solely on this step, it provides insights into the extent of improvement achieved in the portion of the code itself. The constitutive time is measured utilizing the same profiling techniques as in chapter 7 and chapter 8. In line with the model optimizations presented in chapter 7 and chapter 8, the (de)allocation time of memory is not included in the performance metrics. This omission is again attributed to the fact that memory (de)allocation transpires only once during the entire simulation process. Consequently, its impact on the overall performance is considered negligible and does not necessitate inclusion in the performance evaluation.

Important to note is that no output results, such as the render in Paraview, have been generated during the runs that were performed to measure performance, as writing results to output files can significantly influence the results in terms of performance. Furthermore, in order to integrate the GPU implementation in the finite element framework, a contribution has been made to the open source software Pyre [49]. Through the contribution in Pyre, GPU memory is facilitated within the finite element framework. As a result, the GPU-facing data-structures can be allocated via CUDA functionalities in a coalesced memory layout. Data-structures that are not GPU-facing remain allocated in a strided memory layout.

9.2 Single-GPU integration

The outcomes derived from the integration process are expounded upon separately for the different material models. The detailed results of the integration pertaining to the linear elastic model can be found in subsection 9.2.1, while the outcomes associated with the Neo-Hookean model are presented in subsection 9.2.2, followed by the results of the orthotropic material model in subsection 9.2.3.

9.2.1 Linear elastic results

The wall time for both GPUs is graphically illustrated in Figure 9.3 for the linear elastic material model. The performance ranges between 676 and 714 seconds, showing a decrease in wall time of 2.6%. The minor difference in wall time can be attributed to the small mesh sizes that were evaluated. The mesh sizes evaluated in the model-agnostic and model-specific optimizations contained a considerably larger amount of computations, ranging between $1M$ and $34M$. Considering the time spend on the constitutive time only, a discernible trend becomes visible for the different implementations. The GPU-accelerated constitutive update exhibits improved performance that increases from 2.89× decrease in computing time up to a 2.98× decrease in computing time after taking the memory transfers into account.

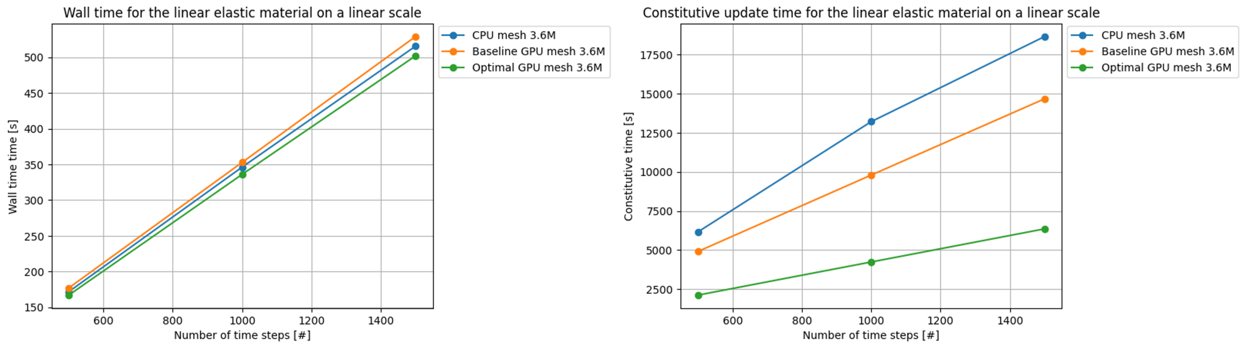


Figure 9.3: *Simulation results utilizing the linear elastic material model. The performance ranges between* 676 *and* 714 *seconds, showing a decrease in wall time of* 2.6%*. The GPU-accelerated constitutive update exhibits improved performance that increases from* 2.89× *decrease in computing time up to a* 2.98× *decrease in computing time.*

The mesh size has been varied in order to examine the effect of increasing the computational workload on the performance of the different implementations. The results for different mesh sizes are summarized below in Table 9.2 for the wall time. Examining the wall time, a decrease in wall time is only observed for the larger numbers of subdivisions.

Mesh subdivision	CPU wall time [s]	GPU wall time [s]	Percentual difference [%]
0	1.32	1.45	+9.85
1	10.59	10.8	+1.98
2	84.9	84.4	-0.59
3	694	676	-2.60

Table 9.2: *The wall time for different mesh sizes utilizing the linear elastic material model. A decrease in wall time is only observed for the larger numbers of subdivisions.*

Table 9.3 depicts the results for different mesh sizes in terms of the constitutive time. As the number of elements in the mesh increases, a significant improvement in constitutive update time can be observed.

Mesh subdivision	CPU constitutive time [ms]	GPU constitutive time [ms]	Percentual difference [%]
0	55	83	+50.9
1	470	192	-49.2
2	3170	1112	-64.9
3	25260	8475	-66.4

Table 9.3: *As the number of elements in the mesh increases, a significant improvement in constitutive update time can be observed.*

9.2.2 Neo-Hookean results

The performance for Neo-Hookean material model is depicted in Figure 9.4. Compared to the linear elastic material model, all GPU implementations perform better with respect to the CPU implementation. This observation makes sense as the computational complexity for the Neo-Hookean material model is significantly increased which is beneficial for the GPU implementation. The baseline implementation increases performance by 3.2%, while the optimal implementation results in a 5.6% improvement in simulation wall time. In terms of time spent in the constitutive time, the optimal GPU implementation is approximately 2.4× faster compared to the CPU implementation.

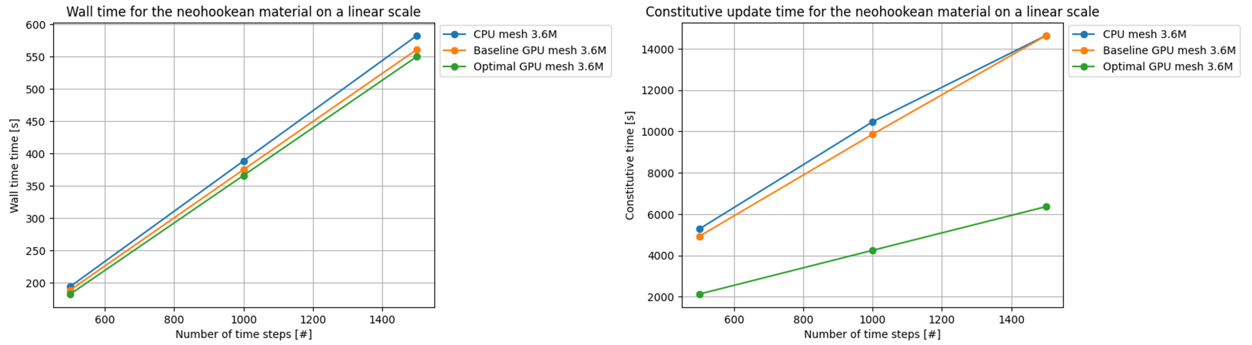


Figure 9.4: *Simulation performance utilizing the Neo-Hookean material model. The baseline implementation increases performance by* 3.2%*, while the optimal implementation results in a* 5.6% *improvement of performance.*

Similar as for the linear elastic material model, the mesh size has been varied in order to examine the effect of increasing the computational workload on the performance of the different implementations. The results for different mesh sizes in terms of the wall time are summarized below in Table 9.4. For the Neo-Hookean material a larger decrease in wall time is observed

Mesh subdivision	CPU wall time [s]	GPU wall time [s]	Percentual difference [%]
0	1.51	1.57	+3.97
1	9.00	11.4	+26.7
2	96.5	90.8	-6.00
3	772	731	-5.32

Table 9.4: *The wall time for different mesh sizes utilizing the Neo-Hookean material model. For a finer mesh a larger decrease in wall time can be observed compared to the linear elastic material.*

The constitutive time for the different mesh sizes is summarized below in Table 9.5. The increase in performance regarding the constitutive update time is comparable with the increase observed for the linear elastic material model.

Mesh subdivision	CPU constitutive time [ms]	GPU constitutive time [ms]	Percentual difference [%]
0	30	83.4	+178
1	320	192	-40.0
2	2395	1112	-53.6
3	19235	8474	-55.9

Table 9.5: *The constitutive time for different mesh sizes utilizing the Neo-Hookean material model. A significant increase in performance is observed for the constitutive time as the number of elements in the mesh increases.*

9.2.3 Orthotropic results

Important to highlight for the orthotropic material model is the fact that the code of the research group currently only supports axial loading for composite materials. For the orthotropic material the performance is illustrated in Figure 9.5. With respect to the other material models the obtained speed-up values are considerably larger for the orthotropic material model as the optimal implementation increases performance by 15.3%. Examining the results in terms of the constitutive time, it is observed that approximately 2.78× less time is spent on the constitutive update compared to the CPU implementation.

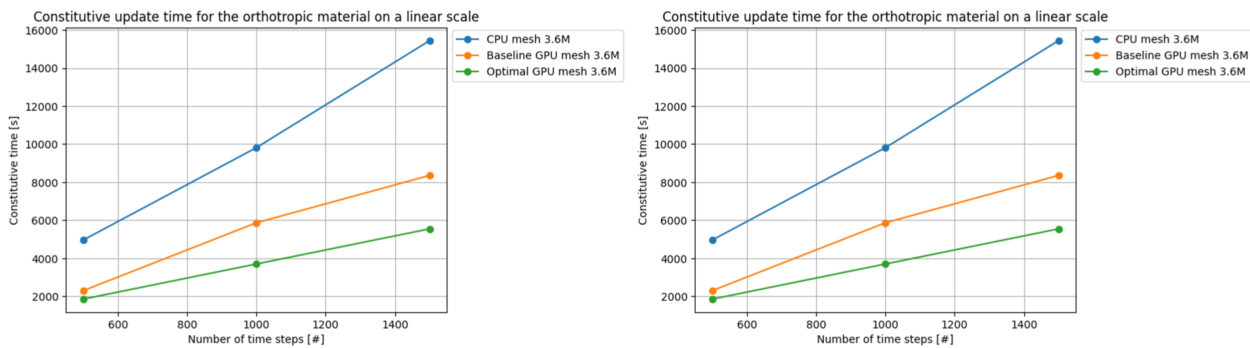


Figure 9.5: *Simulation performance utilizing the orthotropic material model. With respect to the other material models the obtained speed-up values are considerably larger for the orthotropic material model as the optimal implementation increases performance by 15.3%. Examining the results in terms of the constitutive time, it is observed that approximately 2.78× less time is spent on the constitutive update compared to the CPU implementation.*

The variation of the mesh size resulted in Table 9.6. The decrease in wall time is significantly larger compared to the other material models. In terms of the constitutive time the results for the orthotropic material model align with the results of the other material models. The results are summarized below in Table 9.7.

Mesh subdivision	CPU wall time [ms]	GPU wall time [s]	Percentual difference [%]
0	1.51	1.49	-1.13
1	11.8	10.47	-11.3
2	95.6	81.3	-15.0
3	774	656	-15.3

Table 9.6: *The wall time for different mesh sizes utilizing the orthotropic material model. The decrease in wall time is significantly larger compared to the other material models.*

Mesh subdivision	CPU constitutive time [s]	GPU constitutive time [ms]	Percentual difference [%]
0	70	62.6	-10.6
1	295	148	-48.9
2	1865	707	-62.1
3	15440	5546	-64.1

Table 9.7: *The constitutive time for different mesh sizes utilizing the orthotropic material model. The decrease in constitutive time aligns with the results of the other material models.*

9.2.4 Verification

In order to ascertain the correctness of the computed results obtained through the GPU-accelerated simulation, a meticulous verification process was undertaken by comparing the displacement of the rectangular beam in Paraview. The comparative analysis yielded the depicted results illustrated in Figure 9.6. Upon careful examination, it becomes evident that the magnitude of displacement along the longitudinal direction of the beam remains consistent and equivalent for both the CPU and GPU implementations.

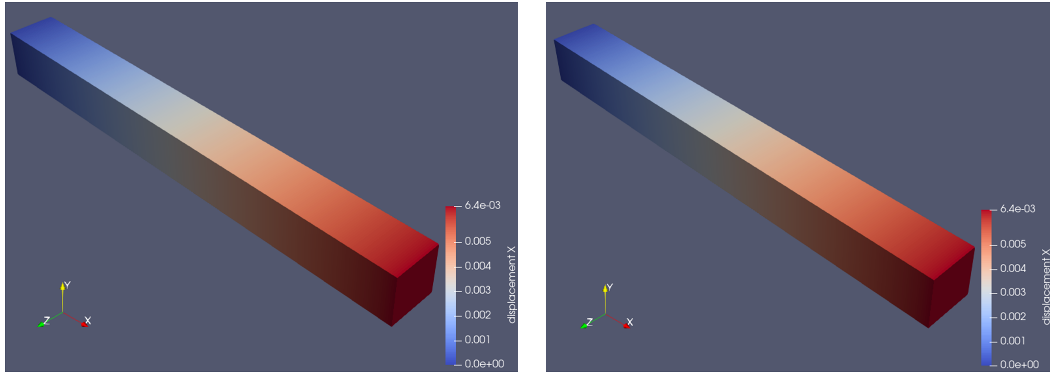


Figure 9.6: *The displacement for the simulation of the elastic material visualized in Paraview. The left image shows the result of the standard CPU implementation, while the right image depicts the result for the GPU-accelerated implementation. The displacement magnitude in the displayed x-direction is equal for both simulations.*

Furthermore, to provide further verification of the GPU-accelerated implementation, the reaction force at the clamped end of the beam was computed by aggregating the internal force contributions from all nodes located along the clamped edge. The resulting reaction forces for both the linear elastic and Neo-Hookean constitutive models are illustrated in Figure 9.7. To ensure the utmost accuracy in the comparison of results, the finest mesh evaluated was employed. As expected, there is no discrepancy between the two implementations. Based on the identical magnitude of displacement and reaction force at the clamped edge, it can be concluded that the GPU-accelerated implementation has been successfully verified.

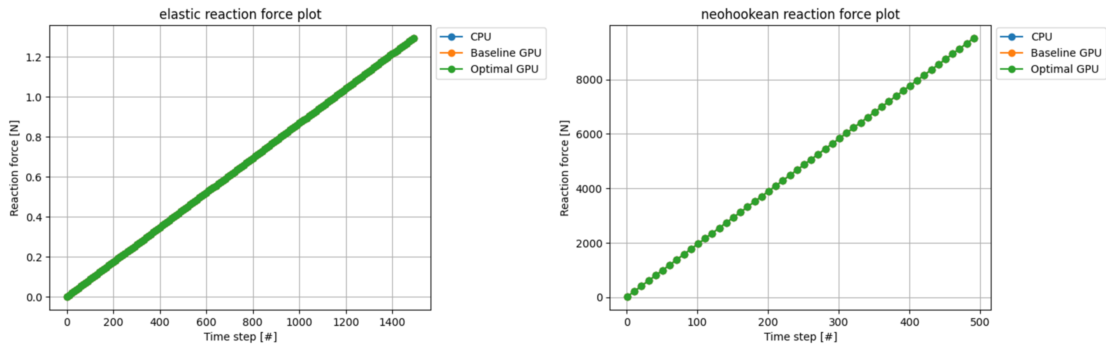


Figure 9.7: *A plot of the reaction force for the different implementations. For both material models the resulting reaction force is identical. As a result, only one curve is visible in the plot.*

9.3 Extension to a Multi-GPU implementation

To expand the single GPU implementation to a multi-GPU implementation, the High Performance Cluster Delft Blue is utilized. Delft Blue provides 10 GPU nodes, each consisting of 2 CPUs and 4 GPUs. Specifications of the hardware can be found at [50]. Due to the limited availability of utilizing multiple GPUs on Delft Blue, it was decided to operate with a single GPU per node for simplicity. Furthermore, the results depicted consists of only a single run per mesh size. The main challenge to expand a single GPU implementation to a multi-GPU implementation relies in the communication between different hardware components. As the Summit finite-element framework already provides support for CPU communication, there was no need to implement inter-GPU communication. Furthermore, by maintaining a one-to-one ratio between the number of CPUs and GPUs, no further adjustments need to be made to the GPU implementation as each CPU directly communicates with its own GPU.

Below, the performance for the single and multi-GPU implementation on the hardware setup of Delft Blue are depicted in Figure 9.8 on a logarithmic scale. The computation time decreases from 831 to 423 seconds when increasing the utilized number of GPUs to two, which is approximately twice as fast. The significant increase in performance makes sense as the amount of computational resources is doubled as well. Increasing the number of GPUs further, the increase in performance quickly levels off. Adding a fifth GPU only leads to a 18.2% improvement in performance. The small gain in performance can be attributed to the fact that the problem size evaluated is still relatively small. Consequently, as the problem is partitioned over multiple GPUs, the problem size allocated to each GPU is even smaller. As a result, the benefits from the GPU implementation are not fully extracted.

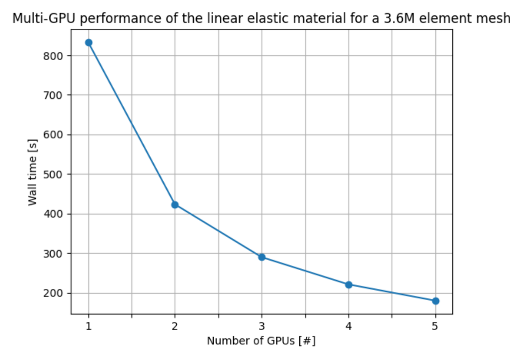


Figure 9.8: *The performance in terms of total simulation time for an increasing number of GPUs. Initially the gain in performance is approximately* 50%, *which levels off to* 18.2% *for the largest GPU amount that is evaluated.*

9.4 Conclusion

Analyzing the results of the previous sections, the GPU implementation can be assessed for both the single and multi-GPU setup. This section concludes on the results that have been obtained for the different material models in the integrated GPU implementation. Examining the results of the single GPU implementation for different hardware setups, the following conclusions can be drawn:

- The smaller effect on wall time compared to the change in constitutive time can be partially attributed to the substantial adjustment required in the memory layout pattern due to the GPU implementation. A coalesced memory layout as depicted in Figure 7.3 is utilized for efficient memory transfers between CPU and GPU. Consequently, all parameters interacting with the GPU are stored according to the coalesced memory layout, whereas in the CPU implementation these parameters are stored according to the strided memory layout as illustrated in Figure 7.1. As a result, the performance of CPU code outside the constitutive update step is affected.
- The computational complexity of the material model utilized drives the gain in performance of the single GPU implementation. For the same mesh size and number of time steps the gain in performance is larger for a computationally more expensive material model.

The multi-GPU implementation was only evaluated for different mesh sizes for the largest number of time steps considered. Investigating the results, it is observed that as the number of GPUs increases, the gain in performance levels off earlier than expected. This can be attributed to the fact that the problem size evaluated is still relatively small. Consequently, as the problem is partitioned over multiple GPUs, the problem size allocated to each GPU is even smaller. As a result, the benefits from the GPU implementation are not fully extracted.

Important to note is the fact that a further expansion in terms of the number of CPUs and GPUs could not be obtained due to the limited availability of computational resources. It is anticipated that at some point increasing the number of GPUs will not lead to an increase in performance for two reasons. First, the portion of the problem size allocated to each GPU can become too small to fully benefit from the GPU-accelerated implementation. Additionally, communication overhead between the increased number of CPUs can lead to a increase in simulation time that is larger than the gain in performance achieved by adding more computational resources in terms of the number of GPUs utilized.

Part IV

Closure

10 Conclusion

Following the part on accelerating Finite Element Analysis with GPUs, this part concludes the thesis, starting with a conclusion on the obtained results. In accordance with the primary and secondary research questions presented in chapter 1, the overarching research objective of this study was formulated as follows:

Main research objective

To develop a GPU-accelerated implementation of the constitutive update in an explicit time scheme and investigate the impact on simulation time upon integration within a Finite Element Framework.

The present study has successfully accomplished the aforementioned research objective through the development of a GPU-accelerated implementation of the constitutive update within an explicit time scheme for different material models in the context of solid mechanics. Furthermore, each material model's implementation has been seamlessly integrated into a Finite Element framework, allowing for a comprehensive assessment of its influence on the overall simulation time. Finally, the implementation has been extended to a multi-GPU setup, through which it was verified that adding additional computational resources increases the total speed-up that can be achieved. The remainder of this chapter serves as a summary of the results of this study, through which the sub-research questions are answered, followed by the answer to the main research question.

Sub-research question A

How can the low-level computations in the FEM be evaluated as efficiently as possible?

The optimal evaluation of the constitutive update encompasses two distinct facets. Firstly, an exploration of the GPU implementation that remained agnostic to the material model was undertaken in chapter 7. From this model-agnostic optimization it was deduced that the most efficient GPU implementation entails the utilization of pinned memory combined with a coalesced memory layout.

The second facet involves a targeted optimization process that primarily focuses on the specific material model employed, as demonstrated in chapter 8. This model-specific optimization is intricately linked to the nature of the particular material model. Subsequently, the constitutive model is thoroughly analyzed, and its distinctive features are leveraged to further enhance performance.

Two different parallelization strategies were explored. For the strain to stress strategy the computing architecture of the implementation is tailored according to the concept to reduce idle GPU time, defined in section 7.4, to efficiently iterate over the problem size, thereby facilitating concurrency resulting in a total computation speed-up ranging from 5.76× up to 12.2×. The displacement to internal force strategy is less demanding in terms of memory transfers as it uses nodal data as opposed to quadrature point data. However, because of the coupling between nodal degrees of freedom of different elements, this strategy does not utilize the concept to reduce idle GPU time. Hence, all memory is transferred at the beginning of the computation. The significantly lower memory demand resulted in a total computation speed-up ranging from 31.7× up to 37.7×.

In terms of material models, the simplicity of the linear elastic material model results in a larger overlap of memory transfers and computations. However, the low computational complexity results in the smallest increase in performance for the strain to stress strategy. The Neo-Hookean material model shows a larger increase in performance compared to the linear elastic material model in the strain to stress strategy

due to its increased computational complexity. A further increase in speed-up is mainly limited by the coupling between all components of the deformation gradient tensor $\mathbf{F}$ and the first Piola-Kirchhoff stress tensor $\mathbf{P}$. This becomes more evident from the results of the displacement to internal force strategy, where the different material models have the same memory demand. For this parallelization strategy, the increase in performance for the linear elastic material model is larger than for the Neo-Hookean material model. This result highlights the detrimental effect of the coupling in the Neo-Hookean material model as described above. The orthotropic material model has the largest increase in performance for both parallelization strategies. The orthotropic material model benefits from the symmetry of the stress tensor, enabling optimizations that decrease memory transfers. Additionally, the computation of the local and global laminate stiffness matrices significantly increases the computational complexity for this material model.

Answer sub-research question A

By harnessing the full potential of the underlying GPU functionalities and subsequently tailoring the implementation to align with the distinct characteristic features inherent to the specific material model the optimal GPU-accelerated implementation is obtained, resulting in a total computation speed-up ranging from $5.76\times$ to $12.2\times$ for the strain to stress strategy and from $31.2\times$ up to $37.7\times$ for the displacement to internal force strategy.

Sub-research question B

To what extent is the total simulation time of a GPU-accelerated FEA model decreased compared to a single CPU FEA model?

The integration of the GPU-accelerated constitutive update implementation within the Finite Element framework, as outlined in chapter 9, facilitates a comprehensive assessment of its impact on a complete FEM simulation. Through the contribution to the open source software Pyre, GPU memory is facilitated in the finite element framework. The alteration in memory access patterns resulting from the GPU implementation impacts the performance of methods that rely on the data utilized by the GPU, thereby influencing the overall simulation time.

The single GPU implementation exhibited a significant increase in performance in terms of the time spent on the constitutive update. The time spent on the constitutive update itself is inherently tied to the specific material model employed. As a result, the magnitude of the decrease in the total simulation time varies for each material model.

In terms of total simulation time, a small increase in performance was achieved for the linear elastic and Neo-Hookean material models. The gain in performance for the orthotropic material model was considerably larger as that specific material model is more balanced in terms of computational complexity and the required amount of CPU-GPU interaction compared to the other material models. However, the evaluated simulations were not large enough to exhibit an increase in performance as obtained through the model-agnostic and model-specific optimization. Through the adjustment in memory layout calculations outside the constitutive update are affected and show lesser performance. The verification of results establishes the capability of the GPU-accelerated implementation to generate identical results to the baseline single CPU implementation, while decreasing the constitutive and wall time significantly.

Answer sub-research question B

The employed GPU-accelerated implementations demonstrate their ability to generate accurate results within the context of a Finite Element Method (FEM) simulation. The increase in performance in the constitutive update step is clearly visible. However, it does not always translate to a significant gain in total simulation time due to the impact of the GPU implementation on methods outside the constitutive update step.

Sub-research question C

What is the trade off, in terms of the number of CPUs and GPUs, in order to create the hardware setup that makes the best use of the resources available for a given problem?

The extension of the Finite Element framework to encompass a multi-GPU implementation offers the potential for true scalability, as it enables the scaling in terms of both memory and time. By incorporating multiple GPUs, it becomes possible to assess the trade-off between the number of CPUs and GPUs in order to optimize the utilization of available computational resources. This trade-off analysis commences at the single-GPU level, where the concept to reduce idle GPU time, as explained in section 7.4, indicates that the performance for the GPU implementation becomes optimal for a relatively small grid size. Utilizing larger grid sizes yield only a marginal improvement in performance.

Due to the limited availability of computational resources, only a small exploration into multi-GPU setups was made. It was observed that as the number of GPUs increases, the gain in performance levels off earlier than expected. This can be attributed to the fact that the problem size evaluated is still relatively small. Consequently, as the problem is partitioned over multiple GPUs, the problem size allocated to each GPU is even smaller. It is expected that in case larger problem sizes are evaluated a linear gain in performance is realized as the number of GPUs increases.

It must be noted that, for an arbitrarily large number of GPUs, performance will eventually saturate for two reasons. First, the portion of the problem size allocated to each GPU can become too small to fully benefit from the GPU-accelerated implementation. Additionally, communication overhead between the increased number of CPUs can lead to a increase in simulation time that is larger than the gain in performance achieved by adding more computational resources in terms of the number of GPUs utilized.

Answer sub-research question C

The trade-off is contingent upon both the specific material model employed and the size of the problem being solved. The problem size should be partitioned in portions large enough to fully benefit from the GPU-accelerated implementation.

Main research question

What speed up can be achieved in a FEA simulation applied to solid mechanics in an explicit time scheme by using massive parallelization on a heterogeneous hardware architecture?

Finally, the main research question is addressed. Gathering the findings that resulted in the answers to the sub-research questions, the answer to the main research question can be formulated.

Main research question

The achieved speed-up is directly related to the constitutive model employed. Utilizing the discovered optimal GPU-accelerated implementations in the displacement to internal force strategy, single-GPU performance is improved in the constitutive update, with a total computation speed-up ranging from $31.7\times$ to $37.7\times$ and a computing speed up that varies from $74\times$ to $109\times$. The translation to a significant decrease in total simulation time requires further investigation. Exploring a multi-GPU setup indicates that utilizing multiple GPUs is beneficial for the total simulation time as for sufficiently large problems, doubling the number of GPUs will result in half the original simulation time.

11
Recommendations for Future Work

Based on the conducted research and the findings presented in this study, several recommendations can be proposed for future research and development. These recommendations are organized according to the order in which the research questions have been addressed in chapter 10. By addressing these recommendations, future research and developments can further advance GPU-accelerated implementations in FEA applied to solid mechanics and contribute to the optimization and efficiency of such finite element simulations.

Optimization of low-level computations in the constitutive update

On the low-level computation scale, there are potential avenues for further optimization that could lead to additional speed-ups in terms of computing speed. These optimizations include both computational modeling and GPU functionalities.

- **Tiling the problem size**
  The displacement to internal force parallelization strategy exhibited the largest enhancement in performance. However, in the reduced memory implementation all nodal degree of freedom values at a time step are transferred in a single memory transfer to assure the value of all nodal degrees of freedom are available as a coupling exists between different nodes. However, an additional optimization strategy that could be explored is the application of a tiling scheme. By determining which nodes are coupled, a smaller portion of the problem size could be transferred to the GPU. Consequently, the concept to reduce idle GPU time can be applied to enhance performance, resulting in an improved GPU implementation.
- **Utilizing shared memory**
  Further investigation into the utilization of shared memory in the GPU-accelerated implementation is warranted to determine if the current implementation is indeed the most efficient approach. Currently, due to the limited cooperation between threads, shared memory is not extensively utilized. However, given the significant impact of memory fetching tasks on processing events, it is worth exploring if shared memory can be incorporated in a more effective manner. To explore the incorporation of shared memory, it is essential to analyze the memory access patterns and data dependencies within the constitutive update algorithm. Additionally, proper synchronization mechanisms need to be implemented to ensure data integrity and consistency among threads.
- **Simplify arithmetic operations**
  The computational cost associated with various arithmetic operations can vary significantly, and taking advantage of this knowledge can lead to performance improvements in the GPU-accelerated implementation. One such example is the divide operation, which is relatively expensive compared to other arithmetic operations. To mitigate the computational cost of division, alternative formulations such as the co-factor formulation for tensors can be considered. By avoiding explicit divisions and instead leveraging alternative mathematical formulations, the overall computational efficiency can be enhanced. Furthermore, GPUs offer fused multiply-add instructions that can expedite both multiplication and summation operations. Fused multiply-add instructions allow for the simultaneous execution of a multiplication and addition in a single instruction, reducing the number of separate operations required and potentially improving performance. By leveraging fused multiply-add functions provided by the GPU, multiple arguments can be multiplied and added together in a more efficient manner, leading to faster computation. However, it is important to note that the effectiveness of these optimizations may vary depending on the specific algorithms, data structures, and hardware architecture involved. Thorough performance analysis and benchmarking are necessary to evaluate the impact of these optimizations and ensure their suitability for the given problem.

Extending the FEM modeling capacity

The current research focused primarily on accelerating the constitutive update across the whole domain, while only considering explicit time schemes. The capacity of problems that can be modeled with the GPU-accelerated implementation can be extended with the following recommendations:

- **Implicit time schemes**
  Investigate the integration of implicit time integration schemes within the GPU-accelerated implementation. Implicit schemes offer superior stability properties and can efficiently handle larger time steps, thereby enabling the simulation of more challenging problems. However, it is important to carefully consider the additional computational requirements and algorithmic complexities associated with implicit schemes, including the solution of large linear systems.

- **Different material models**
  Explore additional material models beyond the linear elastic, Neo-Hookean, and orthotropic models covered in this thesis to evaluate the generalizability and performance implications of the GPU-accelerated constitutive update across a broader range of material models.

- **Localized behavior**
  The current GPU-accelerated implementation does not incorporate plasticity or crack propagation, as these phenomena typically occur at a more localized scale. However, a promising direction for future research is to explore methods for incorporating these localized phenomena into the GPU-accelerated implementation. Depending on the scale at which crack propagation or plasticity occurs, the subsequent computations can be performed either on the GPU or the CPU, depending on what is more efficient. Since these phenomena involve a significant change in material behavior, incorporating them into the Finite Element framework would make it more comprehensive and capable of capturing a wider range of material responses. Investigating methods to account for localized phenomena such as plasticity and crack propagation, and determining the appropriate computational strategies to handle these phenomena efficiently, would be valuable for extending the capabilities of the GPU-accelerated implementation and enabling more comprehensive simulations.

- **Other phenomena**
  In addition to phenomena occurring on a local scale, there are also global-scale phenomena that can be considered for future research and development of the GPU-accelerated implementation. Two such phenomena specifically highlighted here are contact and composite material failure:

  - Contact algorithms play a crucial role in simulating the interaction between bodies or surfaces. Integrating efficient contact algorithms into the GPU-accelerated implementation would enable the simulation of interacting bodies, which is important in various engineering applications. Developing specialized contact algorithms that can fully leverage the computational power of GPUs would significantly enhance the applicability of the implementation in contact-rich simulations.

  - Composite material failure is another global-scale phenomenon that can be explored within the GPU-accelerated implementation. Incorporating material degradation mechanisms, such as progressive damage or failure criteria, can be achieved with relative ease in the GPU implementation. By efficiently modeling and simulating composite material failure, the implementation can provide valuable insights into the behavior of composite structures under various loading conditions. It is anticipated that the GPU acceleration would result in a substantial performance improvement for simulations involving material degradation compared to a CPU implementation.

Extending the GPU utilization

From the profiling of the CPU implementation, it was observed that apart from the constitutive update there are several other steps present in the FEM that can benefit from a parallel implementation. These steps are listed below:

- **Internal force computation**
  The computation of the internal force was already explored as a parallelization strategy. However, it was not implemented in the finite element framework. As the CPU profiling indicated that the step in which the internal forces are computed proves to be a bottle neck in the overall performance of the code, it is recommended to perform the calculation of the internal forces in parallel as well.

- **Computation of the stable time step**
  After taking a step in time in the simulation, the time increment for the next time step is determined by evaluating the equation that determines the stable time step, which differs per material. As this computation is evaluated for every element at every time step, it provides another opportunity to apply a parallel implementation to increase the performance of the implementation further.

Parallelizing other steps that lend themselves well for parallel execution can result in a larger enhancement in performance than is currently realized by parallelizing the constitutive update only.

Investigating the optimal heterogeneous hardware setup

Even though a multi-GPU implementation has been developed, the study was conducted with a limited amount of computational resources. The following recommendations can facilitate useful insights in determining the optimal heterogeneous hardware setup:

- **Extending the range of evaluated problem sizes**
  By extending the range of evaluated problem sizes two insights can be gained. First, by evaluating larger problem sizes the scalability of the multi-GPU implementation can be investigated. For example, whether the enhanced performance of adding additional GPUs levels off for a arbitrarily large problem size. Exploring smaller problem sizes can result in the case where not all GPU resources are used by the same kernel at the same time. This means that multiple kernels can execute simultaneously, enabling a higher concurrency, and thus more performance. Important to note is that, in order to be of effect, the model must be computation driven since executing multiple kernels simultaneously does not yield an improved performance if the memory transfers are dictating the performance.

- **Varying the CPU to GPU ratio**
  The current study only employed a single CPU for each GPU. By exploring different combinations of GPUs and CPUs, it is possible to identify the most efficient heterogeneous hardware setup that maximizes the individual computational load and minimizes communication bottlenecks to achieve a well-balanced and efficient multi-GPU system. Different CPU-to-GPU ratios significantly influence the overall performance of the system. One avenue to explore is utilizing a single GPU in conjunction with multiple CPUs. In a simulation the computational workload is partitioned among the number of CPUs utilized. In chapter 9 it was observed that as the portion of the computational workload is split over more GPUs the GPU performance saturates as well. If multiple CPUs are utilized in conjunction with a single GPU, the GPU resources are utilized more efficiently, resulting in improved overall performance. Referring to Figure 1.1, the number of CPUs should stay in the range of optimal performance for the given problem size. In this range, the number of GPUs should be chosen such that the GPU resources are efficiently utilized. In case of a low computational workload per CPU, multiple CPUs should be used per GPU, while for a large computational workload per CPU, multiple GPUs per CPU should be used. Important to note is that increasing the number of GPUs without a proportional increase in CPU resources may lead to a higher demand on the CPU, potentially resulting in resource contention and decreased performance. Evaluating the described trade-off in terms of computational resources can leverage the inherent parallelism of GPUs more effectively.

Bibliography

[1] D. M. Fernández et al. "Alternate parallel processing approach for FEM". in: *IEEE Transactions on Magnetics* 48 (2 Feb. 2012), pp. 399–402. DOI: 10.1109/TMAG.2011.2173304.

[2] O. A. Bauchau et al. *Structural Analysis: With Applications to Aerospace Structures.* Ed. by G. Gladwell. 1st ed. Springer, 2009. DOI: https://doi.org/10.1007/978-90-481-2516-6.

[3] G. H. Paulino et al. "A general topology-based framework for adaptive insertion of cohesive elements in finite element meshes". In: *Engineering with Computers* 24 (1 Mar. 2008), pp. 59–78. DOI: 10.1007/s00366-007-0069-7.

[4] L. Wu et al. "A micro-meso-model of intra-laminar fracture in fiber-reinforced composites based on a discontinuous Galerkin/cohesive zone method". In: *Engineering Fracture Mechanics* 104 (May 2013), pp. 162–183. DOI: 10.1016/j.engfracmech.2013.03.018.

[5] G. R. Joldes et al. "Real-time nonlinear finite element computations on GPU - Application to neurosurgical simulation". In: *Computer Methods in Applied Mechanics and Engineering* 199 (49-52 Dec. 2010), pp. 3305–3314. DOI: 10.1016/j.cma.2010.06.037.

[6] Q. Huang et al. "GPU as a General Purpose Computing Resource". In: Jan. 2008, pp. 151–158. DOI: 10.1109/PDCAT.2008.38.

[7] N. K. Pikle et al. "High performance iterative elemental product strategy in assembly-free FEM on GPU with improved occupancy". In: *Computing* 100 (12 Mar. 2018), pp. 1273–1297. DOI: 10.1007/s00607-018-0613-x.

[8] A. Dziekonski et al. "Finite Element Matrix Generation on a GPU". in: *Progress In Electromagnetics Research* 128 (2012), pp. 249–265. DOI: 10.2528/PIER12040301.

[9] Z. A. Taylor et al. "On modelling of anisotropic viscoelasticity for soft tissue simulation: Numerical solution and GPU execution". In: *Medical Image Analysis* 13 (2 Apr. 2009), pp. 234–244. DOI: 10.1016/j.media.2008.10.001.

[10] Z. A. Taylor et al. "High-speed nonlinear finite element analysis for surgical simulation using graphics processing units". In: *IEEE Transactions on Medical Imaging* 27 (5 May 2008), pp. 650–663. DOI: 10.1109/TMI.2007.913112.

[11] B. Crovella. *CUDA Unified Memory.* NVIDIA Tutorial. 2020.

[12] N. M. Harris. *How to Optimize Data Transfers in CUDA C/C++.* Dec. 2012. URL: https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/ (visited on 03/05/2023).

[13] NVIDIA. *CUDA C++ Programming Guide Design Guide.* Oct. 2022. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 10/04/2022).

[14] NVIDIA. *CUDA C++ Best Practices Guide Design Guide.* Oct. 2022. URL: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html (visited on 10/24/2022).

[15] M. Nobile et al. "cuTauLeaping: A GPU-Powered Tau-Leaping Stochastic Simulator for Massive Parallel Analyses of Biological Systems". In: *PloS one* 9 (Mar. 2014), e91963. DOI: 10.1371/journal.pone.0091963.

[16] N. M. Harris. *Using Shared Memory in CUDA C/C++.* Jan. 2013. URL: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/ (visited on 03/10/2023).

[17] P. N. Glaskowsky. *NVIDIA's Fermi: The First Complete GPU Computing Architecture.* Oct. 2009. URL: https://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIAFermi-TheFirstCompleteGPUComputingArchitecture.pdf (visited on 10/04/2022).

[18] S. van Paasen. *GPU-accelerated Finite Element Analyses for nonlinear solid mechanics.* Tech. rep. Delft University of Technology, Nov. 2022.

[19] M. J. García-Ruíz et al. "Fixed grid finite elements in elasticity problems". In: *Engineering Computations* 16 (Dec. 1999), pp. 145–164. DOI: 10.1108/02644409910257430.

[20] J. Martínez-Frutos et al. "Efficient matrix-free GPU implementation of Fixed Grid Finite Element Analysis". In: *Finite Elements in Analysis and Design* 104 (July 2015), pp. 61–71. DOI: 10.1016/j.finel.2015.06.005.

[21] N. K. Pikle et al. *GPGPU-based parallel computing applied in the FEM using the conjugate gradient algorithm: a review.* Visvesvaraya National Institute of Technology, 2018. DOI: https://doi.org/10.1007/s12046-018-0892-0.

[22] R. Li et al. "GPU-accelerated preconditioned iterative linear solvers". In: *Journal of Supercomputing* 63 (2 Feb. 2013), pp. 443–466. DOI: 10.1007/s11227-012-0825-3.

[23] I. Kiss et al. "Parallel realization of the element-by-element FEM technique by CUDA". in: *IEEE Transactions on Magnetics* 48 (2 Feb. 2012), pp. 507–510. DOI: 10.1109/TMAG.2011.2175905.

[24] R. Mafi et al. "GPU-based acceleration of computations in nonlinear finite element deformation analysis". In: *International Journal for Numerical Methods in Biomedical Engineering* 30 (3 2014), pp. 365–381. DOI: 10.1002/cnm.2607.

[25] V. Kindratenko. *Numerical Computations with GPUs*. Springer, 2012, pp. 3–28. DOI: 10.1007/978-3-319-06548-9.

[26] H. D. Tran et al. “A scalable adaptive-matrix SPMV for heterogeneous architectures”. In: Institute of Electrical and Electronics Engineers Inc., 2022, pp. 13–24. DOI: 10.1109/IPDPS53621.2022.00011.

[27] M. Geveler et al. “Towards a complete FEM-based simulation toolkit on GPUs: Unstructured grid finite element geometric multigrid solvers with strong smoothers based on sparse approximate inverses”. In: *Computers and Fluids* 80 (1 2013), pp. 327–332. DOI: 10.1016/j.compfluid.2012.01.025.

[28] F. Vázquez et al. “Improving the performance of the sparse matrix vector product with GPUs”. In: 2010, pp. 1146–1151. DOI: 10.1109/CIT.2010.208.

[29] L. Li et al. “The sedimentation of flexible filaments”. In: *Journal of Fluid Mechanics* (2013). DOI: 10.1017/jfm.2013.512.

[30] F. Kruzel et al. “Vectorized OpenCL implementation of numerical integration for higher order finite elements”. In: *Computers and Mathematics with Applications* 66 (10 Dec. 2013), pp. 2030–2044. DOI: 10.1016/j.camwa.2013.08.026.

[31] D. Komatitsch et al. “Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA”. in: *Journal of Parallel and Distributed Computing* 69 (5 May 2009), pp. 451–460. DOI: 10.1016/j.jpdc.2009.01.006.

[32] P. Macioł et al. “3D finite element numerical integration on GPUs”. In: vol. 1. Elsevier B.V., 2010, pp. 1093–1100. DOI: 10.1016/j.procs.2010.04.121.

[33] U. Kiran et al. “GPU-warp based finite element matrices generation and assembly using coloring method”. In: *Journal of Computational Design and Engineering* 6 (4 Oct. 2019), pp. 705–718. DOI: 10.1016/j.jcde.2018.11.001.

[34] A. Dziekonski et al. “Generation of large finite-element matrices on multiple graphics processors”. In: *International Journal for Numerical Methods in Engineering* 94 (2 Apr. 2013), pp. 204–220. DOI: 10.1002/nme.4452.

[35] C. Cecka et al. “Assembly of finite element methods on graphics processors”. In: *International Journal for Numerical Methods in Engineering* 85 (5 Feb. 2011), pp. 640–669. DOI: 10.1002/nme.2989.

[36] K. Banaś et al. “Numerical integration on GPUs for higher order finite elements”. In: *Computers and Mathematics with Applications* 67 (6 Apr. 2014), pp. 1319–1344. DOI: 10.1016/j.camwa.2014.01.021.

[37] G. R. Markall et al. “Finite element assembly strategies on multi-core and many-core architectures”. In: *International Journal for Numerical Methods in Fluids* 71 (1 Jan. 2013), pp. 80–97. DOI: 10.1002/fld.3648.

[38] J. Martínez-Frutos et al. “Fine-grained GPU implementation of assembly-free iterative solver for finite element problems”. In: *Computers and Structures* 157 (June 2015), pp. 9–18. DOI: 10.1016/j.compstruc.2015.05.010.

[39] K. Miller et al. “Total Lagrangian explicit dynamics finite element algorithm for computing soft tissue deformation”. In: *Communications in Numerical Methods in Engineering* 23 (2 Feb. 2007), pp. 121–134. DOI: 10.1002/cnm.887.

[40] J. Zhang. “A direct Jacobian total Lagrangian explicit dynamics finite element algorithm for real-time simulation of hyperelastic materials”. In: *International Journal for Numerical Methods in Engineering* 122 (20 Oct. 2021), pp. 5744–5772. DOI: 10.1002/nme.6772.

[41] Y. Cai et al. “A parallel node-based solution scheme for implicit finite element method using GPU”. in: vol. 61. Elsevier Ltd, 2013, pp. 318–324. DOI: 10.1016/j.proeng.2013.08.022.

[42] A. Dziekonski et al. “Communication and Load Balancing Optimization for Finite Element Electromagnetic Simulations Using Multi-GPU Workstation”. In: *IEEE Transactions on Microwave Theory and Techniques* 65 (8 Aug. 2017), pp. 2661–2671. DOI: 10.1109/TMTT.2017.2714670.

[43] D. M. Kochmann. *Computational Solid Mechanics (Fall 2017)*. ETH Zürich, Dec. 2017.

[44] T. Belytschko, W.K. Liu, and B. Moran. *Nonlinear Finite Elements for Continua and Structures*. John Wiley & Sons, 2000.

[45] Y. Lyu. “Finite Element Analysis Using Triangular Element”. In: *Finite Element Method: Element Solutions*. Singapore: Springer Nature Singapore, 2022, pp. 93–118. DOI: 10.1007/978-981-19-3363-9_5.

[46] F. Hussain et al. “Appropriate Gaussian quadrature formulae for triangles”. In: (Jan. 2012).

[47] C. Kassapoglou. *Design and analysis of composite structures : With applications to aerospace structures*. John Wiley & Sons, Incorporated, 2013.

[48] A. Liu et al. *Quality local refinement of tetrahedral meshes based on 8-subtetrahedron subdivision*. 1996, pp. 1183–1200.

[49] M. Aivazis. *Pyre*. 2023. URL: https://github.com/pyre/pyre.

[50] T. Delft. *DelftBlue: the TU Delft supercomputer*. 2023. URL: https://www.tudelft.nl/dhpc/system (visited on 05/15/2023).

[51] NVIDIA. *The World's first ray tracing GPU: NVIDIA QUADRO RTX 5000*. Mar. 2019. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/quadro-product-literature/quadro-rtx-5000-data-sheet-us-nvidia-704120-r4-web.pdf (visited on 04/12/2023).

[52] NVIDIA. *CUDA C++ Programming Guide Design Guide*. July 2019. URL: https://developer.nvidia.com/blog/migrating-nvidia-nsight-tools-nvvp-nvprof (visited on 12/09/2023).

[53] AMD. *AMD Ryzen™ Threadripper™ PRO 3975WX*. 2020. URL: https://www.amd.com/en/product/10176 (visited on 04/12/2023).

[54] J. Fenlason et al. *GNU Gprof, the gnu profiler*. 1998. URL: https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_node/gprof_toc.html (visited on 05/15/2023).

A
Graphical Processing Unit

This appendix contains information regarding the NVIDIA Quatro 5000 RTX Graphical Processing Unit (GPU). The specifications of the GPU are presented in section A.1 which have been obtained from NVIDIA [51]. Furthermore, information on GPU code profiling can be found in section A.2.

A.1 Specifications

REAL TIME RAY TRACING FOR PROFESSIONALS

Shatter the boundaries of what's possible with the NVIDIA® Quadro RTX™ 5000, powered by NVIDIA Turing GPU to bring real-time ray tracing and accelerated AI to next-generation workflows. Creative and technical professionals can supercharge demanding design and visualization workloads and make more informed decisions faster than ever before. Equipped with 3072 CUDA cores, 384 Tensor cores, 48 RT Cores and 16GB GDDR6 memory, Quadro RTX 5000 can render complex models and scenes with physically accurate shadows, reflections, and refractions to empower users with instant insight. Support for NVIDIA NVLink[1] enables applications to scale memory and performance with multi-GPU configurations[2]. And with the industry's first implementation of the new VirtualLink®[3], Quadro RTX 5000 provides connectivity to the next-generation of high-resolution VR head-mounted displays to let designers view their work in the most compelling virtual environments possible.

Quadro cards are certified with a broad range of sophisticated professional applications, tested by leading workstation manufacturers, and backed by a global team of support specialists. This gives you the peace of mind to focus on doing your best work. Whether you're developing revolutionary products or telling spectacularly vivid visual stories, Quadro gives you the performance to do it brilliantly.

To learn more about the NVIDIA Quadro RTX 5000 visit **www.nvidia.com/quadro**

FEATURES

- Four DisplayPort 1.4 Connectors
- VirtualLink Connector[3]
- DisplayPort with Audio
- VGA Support[4]
- 3D Stereo Support with Stereo Connector[4]
- NVIDIA GPUDirect™ Support
- Quadro Sync II[5] Compatibility
- NVIDIA nView® Desktop Management Software
- HDCP 2.2 Support
- NVIDIA Mosaic[6]

SPECIFICATIONS

GPU Memory	**16 GB GDDR6**
Memory Interface	**256-bit**
Memory Bandwidth	**Up to 448 GB/s**
ECC	**Yes**
NVIDIA CUDA Cores	**3,072**
NVIDIA Tensor Cores	**384**
NVIDIA RT Cores	**48**
Single-Precision Performance	**11.2 TFLOPS**
Tensor Performance	**89.2 TFLOPS**
NVIDIA NVLink	**Connects 2 Quadro RTX 5000 GPUs[1]**
NVIDIA NVLink bandwidth	**50 GB/s (bidirectional)**
System Interface	**PCI Express 3.0 x 16**
Power Consumption	**Total board power: 265 W Total graphics power: 230 W**
Thermal Solution	**Active**
Form Factor	**4.4" H x 10.5" L, Dual Slot, Full Height**
Display Connectors	**4xDP 1.4, 1x USB-C**
Max Simultaneous Displays	**4x 4096x2160 @ 120 Hz, 4x 5120x2880 @ 60 Hz, 2x 7680x4320 @ 60 Hz**
Encode / Decode Engines	**1X Encode, 2X Decode**
VR Ready	**Yes**
Graphics APIs	**DirectX 12.0[7] Shader Model 5.1[7], OpenGL 4.6[8], Vulkan 1.1[8]**
Compute APIs	**CUDA, DirectCompute, OpenCL™**

[1] NVIDIA NVLink sold separately | [2] Connecting two RTX 5000 cards with NVLink to scale performance and memory capacity to 32 GB is only possible if your application supports NVLink technology. Please contact your application provider to confirm their support for NVLink | [3] In preparation for the emerging VirtualLink standard, Turing GPUs have implemented hardware support according to the "VirtualLink Advance Overview". To learn more about VirtualLink, please see www.virtuallink.org | [4] Via adapter/connector/bracket | [5] Quadro Sync II card sold separately | [6] Windows 7, 8, 8.1, 10 and Linux | [7] GPU supports DX 12.0 API, Hardware Feature Level 12_1 | [8] Product is based on a published Khronos Specification, and is expected to pass the Khronos Conformance Testing Process when available. Current conformance status can be found at www.khronos.org/conformance

© 2018 NVIDIA Corporation. All rights reserved. NVIDIA, the NVIDIA logo, Quadro, nView, CUDA, and NVIDIA Turing are trademarks and/ or registered trademarks of NVIDIA Corporation in the U.S. and other countries. OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc. All other trademarks and copyrights are the property of their respective owners.

QUADRO RTX 5000 | DATA SHEET | MAR19

A.2 Profiling

NVIDIA offers several profiling tools that can be utilized to optimize GPU code. The current state-of-the art profiling tools are shown in Figure A.1. The main analysis tool for GPU code is Nsight Systems, which provides a detailed overview of the implementation and can be used to determine overall limitations for the performance of code, identify slow functions or unbalanced workload divisions on the GPU. Streams can be individually tracked, which allows the developer to easily optimize the overlap between kernel executions and data transfers.

Apart from Nsight Systems NVIDIA offers other GPU analysis tools. While Nsight Graphics is more applicable to graphical programs, Nsight Compute is relevant for the current project. The Nsight Compute tool analyzes code on a kernel level, identifies the amount of registers used by each thread and shows line by line the amount of cycles that is required to complete an instruction. The insight in the used resources and their efficiency provides information to extract the last few percentages of performance. In general however, the obtained additional performance by Nsight Compute is expected to be significantly smaller compared to the optimization using Nsight Systems. Therefore, Nsight Compute will only be used as a final check.

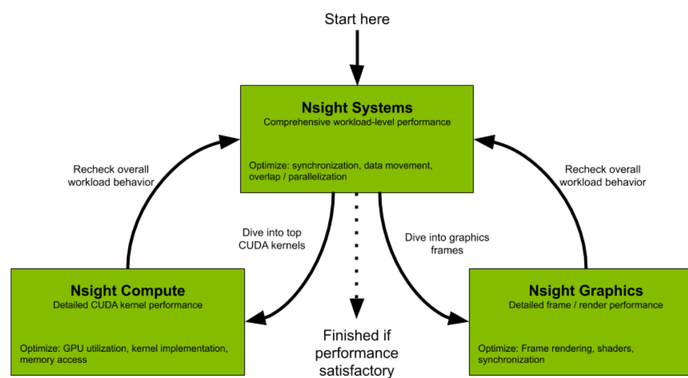


Figure A.1: *Analysis tools offered by NVIDIA to analyze GPU code. The main analysis tool for GPU code is Nsight Systems, which provides a detailed overview of the implementation and can be used to determine overall limitations for the performance of code. Apart from Nsight Systems NVIDIA offers other GPU analysis tools. While Nsight Graphics is more applicable to graphical programs, Nsight Compute can be used to extract the last bits of performance.Figure credits: [52].*

B
Central Processing Unit

This appendix contains information regarding the AMD Ryzen Threadripper PRO 3975 WX, which is the utilized Central Processing Unit (CPU). section B.1 contains specifications of the CPU, which have been obtained from the AMD website [53]. The CPU code profiling process is explained in section B.2.

B.1 Specifications

AMD Ryzen™ Threadripper™ PRO 3975WX

General Specifications

Platform:	Desktop
Product Family:	AMD Ryzen™ PRO Processors
Product Line:	AMD Ryzen™ Threadripper™ PRO Processors
Former Codename:	"Castle Peak"
# of CPU Cores:	32
# of Threads:	64
Max. Boost Clock:	Up to 4.2GHz
Base Clock:	3.5GHz
L1 Cache:	2MB
L2 Cache:	16MB
L3 Cache:	128MB
Default TDP:	280W
Processor Technology for CPU Cores:	TSMC 7nm FinFET
Unlocked for Overclocking:	No
CPU Socket:	sWRX8
Socket Count:	1P
Max. Operating Temperature (Tjmax):	90°C
Launch Date:	7/14/2020
***OS Support:**	Windows 10 - 64-Bit Edition *Operating System (OS) support will vary by manufacturer.

Connectivity

PCI Express® Version:	PCIe 4.0
System Memory Type:	DDR4
Memory Channels:	8
System Memory Specification:	Up to 3200MHz

Graphics Capabilities

Integrated Graphics:	No

B.2 Profiling

For CPU code *Gprof* is the standard profiler that comes with the *gcc* compiler. To obtain the profiling report the code should be implemented using the *-pg* flag, which enables the compiler to perform line by line profiling. In the profiling report information is present over each method that is called in the program. The main limitation for *Gprof* is the timing resolution. Methods with a shorter execution time than 0.01s can not be timed, which means that in case the computational complexity is low, there is a minimum problem size required to actually time the CPU implementation. To use *Gprof*, the implementation should first build and execute a single time. After that *Gprof* analyzes the executable that was ran and generates the profiling report. Further documentation of *Gprof* can be found at [54].