Master in High Performance Computing

GPU accelerated contact simulations

Supervisor(s):
Luca Heltai Supervisor,
Teseo Schneider Supervisor

Candidate:
Alexander Cristhoper Trujillo Ochoa

8th edition
2021–2022

Contents

List of Figures

Acknowledgments

To my parents, Carmen Rosa Ochoa Cabrera and Leoncio Zenon Trujillo Aquino for their unconditional support throughtout my career.

I am truly thankful to MHPC program along with Luca Heltai and Teseo Schneider for their guidance and pieces of advice which pointed this project in the right direction. As well to Daniele Panozzo for its weekly recommendation about computational toolkit details and Yibo Liu for her joint collaboration during this year.

Thanks to my friends and MHPC colleagues, Giulio Malenza, Serafina Di Gioia , Debarshi Banerjee for the constant discussion during the courses where we learned from each other.

This work presented and travel allowances were supported by the Abdus Salam International Centre for Theoretical physics (ICTP) which made possible this opportunity.

Abstract

This project aims at implementing a GPU accelerated non-linear solver for hyperelastic neo-Hookian models with contact. The project will build on the PolyFEM [11], a C++ and Python finite element library (https://polyfem.github.io/), coupled with the Incremental potential contact method which allows accurate simulation of surface-to-surface contact (https://ipc- sim.github.io/).

Currently, PolyFEM relies on multithreading libraries like TBB, the strategy is to reengineer the non-linear solver section of the code to make it suitable for CUDA [10] device functions taking advantage of libraries such as cuBLAS, cuSPARSE, Thrust and external sparse linear solvers with GPU support. Our goal is to obtain a performance improvement of stress tests in comparison with multithreading implementation from low to high end CPUs.

The project will be conducted in collaboration with Teseo Schneider (University of Victoria).

Chapter 1

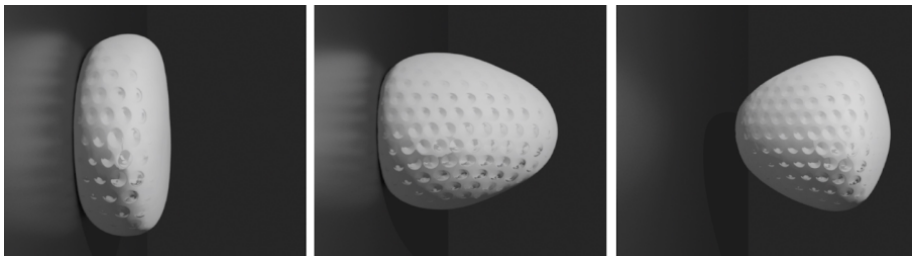
Introduction

It is no surprise that numerical methods applied to specific physics-based simulations could be computationally expensive, but nowadays GPU computational power could improve the performance in general along with multi-threading implementations.

The current project aims to implement a GPU version of the PolyFEM library, which could speed-up up to 3 times simulations using non-linear models in comparison with the current version using Intel threading building blocks or C++ threads, those tests were run on CPU models as AMD(R) EPYC(TM) 7452 with NVIDIA GPU RTX A5000 and Intel Core(TM) i7-5930K with NVIDIA GeForce GTX 1080 Ti.

We focused on Neo-Hookean models (specifically to the code section that assembles and solves this model) as PolyFEM could be linked with the Incremental potential contact (IPC) library to obtain accurate simulation of surface-to-surface contact which uses this mentioned part of the code, in the following chapters we point the hot-spots and parallelizable parts that could take advantage of a GPU implementation.

(a) Video footage



(b) Simulation using PolyFEM and IPC

Figure 1.1: We show key frames from a highspeed video capture of a foam practice ball fired at a fixed plate. This shows a high fidelity behavior with the actual physical event, stating the importance to obtain a faster implementation for this library. [1]

This thesis is structured in the following way:

- Chapter two will contain the theoretical framework to understand the underlying working of PolyFEM and IPC library.
- Chapter three will focus on the benchmarking of the current implementation and will point out portions of code to be suitable for GPU computation.
- Chapter four will present a restructuration of the multithreading parts and the steps to convert them as CUDA kernels, as well managing external libraries as EIGEN for CUDA code.
- Chapter five will cover about Polysolve which is a cross-platform Eigen wrapper for many different external linear solvers, for this project we will modify the library compilation process to activate the CUDA backend for the AMGCL library.
- Chapter six will show the final results and benchmark in comparison with original implementation for computer graphics stress and small tests.

Chapter 2

Theory overview

In order to grasp the underlying work of PolyFEM we present a condensed overview of how finite element and Incremental Potential contact method are used for hyperelastic model simulations and implemented on this library.

2.1 Kinematics of hyper-elastic models

The mathematical framework needed for this simulations starts with defining how it changes the state of the material from time to time. For this we consider $\mathbf{X}$ as the configuration of several points in the material at certain point. We will quantify the deformation of the objects of interest in terms of a deformation mapping (ϕ) between an initial configuration $\mathbf{X_0}$ and a deformed configuration $\mathbf{X_d}$, consider this two configurations we will define the displacement mapping $\mathbf{u}$ as the difference between the deformed configuration and the original one [12].

$$\phi(\mathbf{X_0}, t) = \mathbf{X_d} \tag{2.1}$$

$$\mathbf{u}(\mathbf{X_0}, t) = \phi(\mathbf{X_0}, t) - \mathbf{X_0} \tag{2.2}$$

One important notation that arises is the deformation gradient, which will contain the deformation information of the material.

$$\mathbf{F} = \frac{\partial \phi}{\partial \mathbf{X_0}} \tag{2.3}$$

And more generally

$$\mathbf{F} = \nabla \mathbf{u} + I \tag{2.4}$$

Once presented this notation, here is the governing equation of motion gave by the basic principles of continuum mechanics, so to speak an extension of the Newton second law but applied over the continuum of the material.

$$\rho_0(\mathbf{X_0}) \frac{\partial^2 \mathbf{u}}{\partial t^2} + \mathbf{f}^{\text{int}} = \mathbf{f}^{\text{ext}} \tag{2.5}$$

Where ρ_0 is the mass density of the material and $\mathbf{f}^{\text{int}}$ is normally stated as the divergence of the first Piola-Kirchoff stress tensor describing the forces of interaction in the material.

$$\mathbf{f}^{\text{int}} = -\nabla^{\mathbf{X}_0} \cdot \mathbf{P} \tag{2.6}$$

To connect the stress tensor and configuration of the material, we obtain this relation from the definition of the stress tensor which is the derivative of the strain energy density function (Ψ) with respect to the deformation gradient [12].
This strain energy is known as the constitutive law for hyperelastic materials, which depends on a specific model of the elastic material.

$$\mathbf{P} = \frac{\partial \Psi}{\partial \mathbf{F}} \tag{2.7}$$

2.1.1 Neo-Hookean model

We start with the definition of neo-hookean elasticity, which we can consider as an extension of the linear Hookean model, as we can observe in the Fig. 2.1 generally for elastic materials the relation stress-strain is linear at low stress, but as we continue to exert more stress it becomes a non-linear relation which is important to take into account for close to real life simulations.

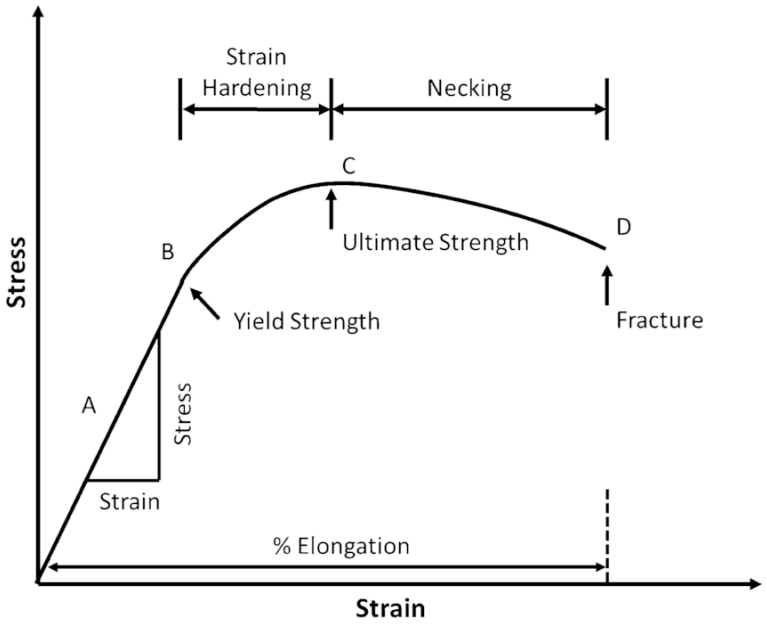


Figure 2.1: Stress-Strain curve for polymer film [13].

Another important characteristic is the hyperelastic strain energy density which is stated by the next formula, specifically for 3D models:

$$\Psi(\mathbf{F}) = \frac{\mu}{2}(\mathbf{Trace}(\mathbf{F^T F}) - 3) - \mu \log(\mathbf{det}(\mathbf{F})) + \frac{\lambda}{2}\log(\mathbf{det}(\mathbf{F}))^2 \tag{2.8}$$

The λ, μ terms arises from the linear elasticity construction which depends of the properties of the material such as $\mathbf{Y}$, the Young modulus, and ν, the Poisson ratio.

$$\lambda = \frac{\mathbf{Y}\nu}{(1+\nu)(1-2\nu)} \tag{2.9a}$$

$$\mu = \frac{\mathbf{Y}}{2(1+\nu)} \tag{2.9b}$$

Now for this specific model we obtain the corresponding stress tensor $\mathbf{P}$

$$\mathbf{P}(\mathbf{F}) = \mu\mathbf{F} + (\log(\mathbf{det}(\mathbf{F}))\lambda - \mu)\mathbf{F}^{-\mathrm{T}} \tag{2.10}$$

2.2 Finite element method

This method is used to solve continuous linear mappings, considering $\mathbf{A}$ a linear mapping which maps from a Hilbert space V to V:

$$\mathbf{Au} = f \tag{2.11}$$

A solution $\mathbf{u}$ can be approximate by means of using the weak formulation, considering a test function v and applying a bi-linear form in both sides.

$$< \mathbf{Au}, v > = < f, v > \tag{2.12}$$

The next step is to determine the basis functions ϕ that belongs to space V, in this way $\mathbf{u}$ can be expressed as

$$\mathbf{u} = \sum_i u_i \phi_i \tag{2.13}$$

Finally we replace the expansions, this is known as Galerkin method.

$$\sum_j u_j < \mathbf{A}\phi_j, \phi_i > = < f, \phi_i > \tag{2.14}$$

For a material with volume Ω and boundary surface $\partial\Omega$, the chosen bi-linear form is the following:

$$< f, \phi_i > = \int_\Omega f\phi_i d\Omega \tag{2.15}$$

To approximate the result of the previous integral we use the Gaussian quadrature method, where with a set of n points and n weights the integral of a function f could be computed, for example the 1D version:

$$\int_a^b f(x)dx = \sum_i^{n-1} w_i f(x_i) \tag{2.16}$$

$$w_i = \int_a^b \prod_{j \neq i} \frac{(x - x_j)}{(x_i - x_j)} dx \tag{2.17}$$

This last values w could be computed beforehand and stored in a table for further use, as well this method is known to solve exactly integral for polynomials with degree less than 2n-1.

With gaussian quadrature and the decomposition in the finite functional space we could rewrite equation 2.12 as:

$$\sum_j u_j \sum_k w_k^2 \, \mathbf{A}(x_k)\phi_j(x_k)\phi_i(x_k) = \sum_k w_k f(x_k)\phi_i(x_k) \tag{2.18}$$

One important consideration for this method is the presence of the following:

$$M_{i,j} = \int_\Omega \phi_i \phi_j d\Omega = \sum_k w_k \, \phi_j(x_k)\phi_i(x_k) \tag{2.19}$$

This is known as the mass matrix, and from now on we will refer to it as $\mathbf{M}$.

2.3 Incremental potential method

Considering the equation 2.5, by using the Galerkin method approach we can express it as:

$$M\ddot{\mathbf{u}} + \mathbf{f}^{\text{int}}(\mathbf{u}) = \mathbf{f}^{\text{ext}}(t) \tag{2.20}$$

For this example we will use implicit Euler for the time discretization, which is the default used in PolyFEM. The set of equations for this method is shown in 2.21a where h is the user-defined time-step.

$$\mathbf{x}^{t+1} = \mathbf{x}^t + \mathrm{h}\dot{\mathbf{x}}^{t+1} \tag{2.21a}$$

$$\dot{\mathbf{x}}^{t+1} = \dot{\mathbf{x}}^t + \mathrm{h}\ddot{\mathbf{x}}^{t+1} \tag{2.21b}$$

According to [2], it is postulated the existence of a conservative potential *V* related to the internal forces of the material, therefore possible to formulate the time step update for new positions x^t as the minimization of an Incremental Potential which is the Euler-Lagrange equation of the function 2.20

$$x^{t+1} = \text{argmin}_{x^{t+1}} E(x^{t+1}, x^t, v^t) \tag{2.22}$$

$$E(x^{t+1}, x^t, v^t) = \frac{1}{2}(x^{t+1} - \hat{x})^T M(x^{t+1} - \hat{x}) - h^2 f_{ext}.x^{t+1} + h^2 V \tag{2.23}$$

Where $\hat{x} = x^t + hv^t$ for implicit Euler, for this thesis we specify for Neo-Hookeans models, where Ψ is its strain energy density function :

$$E(x^{t+1}, x^t, v^t) = \frac{1}{2}(x^{t+1} - \hat{x})^T M(x^{t+1} - \hat{x}) - h^2 f_{ext}.x^{t+1} + h^2 \Psi(x^{t+1}) \tag{2.24}$$

2.4 Newton method for minimization

In order to find the minimum of 2.24 , the default non-linear solver implementation in polyFEM is the multi-dimensional version of the Newton method for root-finding ($\nabla E(x^t) = 0$)

$$x^{t+1} = x^t - (\mathbf{H}_x E(x^t))^{-1} \nabla_x E(x^t) \tag{2.25}$$

In this way, we obtain directly Δx by solving the following linear equation system.

$$\mathbf{H}_x E(x^t) \Delta x^t = -\nabla_x E(x^t) \tag{2.26}$$

This linear system is solved iteratively until tolerance regarding Δx is met. Considering the previous model, the following needs to be computed per each iteration

$$\nabla_x(\Psi(\mathbf{F})) = \mu \mathbf{F} - \frac{\mu}{\mathbf{det}(\mathbf{F})} \mathbf{Adjugate}(\mathbf{F}) + \frac{\lambda \, log(\mathbf{det}(\mathbf{F}))}{\mathbf{det}(\mathbf{F})} \mathbf{Adjugate}(\mathbf{F}) \tag{2.27}$$

$$\mathbf{H}_x(\Psi) = \mu \mathbf{I} + (\frac{\mu \; + \; \lambda(1 - log(\mathbf{det}(\mathbf{F})))}{\mathbf{det}(\mathbf{F})^2}) \mathbf{Adj}(\mathbf{F}) \mathbf{Adj}(\mathbf{F})^T + (\frac{\lambda log(\mathbf{det}(\mathbf{F})) - \mu}{\mathbf{det}(\mathbf{F})}) \frac{\partial}{\partial \mathbf{F}} \mathbf{Adj}(\mathbf{F}) \tag{2.28}$$

2.4.1 Line search method

In order to improve and obtain the estimation of the solution of equation 2.22 in less steps it is required to compute a step length s, this parameter scales the iterative method $x_n = x + sg(x)$, for example g(x) for newton method is stated in 2.25, and approximately optimizes the target function along the line $x + sg(x)$.
For this thesis we will focus on the Backtracking line search coupled with newton descend method as it is the default non-linear solver. We add the step-size parameter s to equation 2.25

$$x^{t+1} = x^t - s(\mathbf{H}_E(x^t))^{-1} \nabla E(x^t) \tag{2.29}$$

To find the adequate step-size per iteration we use the backtracking search, first we consider the following parameters $0 < \alpha < 0.5, 0 < \beta < 1$, then we start the iterations with $s = 1$ and while the criteria 2.30a is true, we shrink $s = \beta s$ and continue to calculate $E(x^t) + \alpha s \nabla E(x^t)^T v$

$$E(x^t + sv) > E(x^t) + \alpha s \nabla E(x^t)^T v \tag{2.30a}$$
$$v = -(\mathbf{H}_E(x^t))^{-1} \nabla E(x^t) \tag{2.30b}$$

2.5 Incremental potential contact (IPC)

Li et. al [1] proposes a modification of IP 2.24 to handle large deformation dynamics with frictional contact by adding the barrier ($\mathbf{B}$) and friction ($\mathbf{D}$) potential terms.

$$x^{t+1} = \text{argmin}_{x^{t+1}} E(x^{t+1}, x^t, v^t) + \mathbf{B}(x^t, \hat{d}) + \mathbf{D}(x^t, \hat{d}) \tag{2.31}$$

Where $\hat{d}$ is a user-defined parameter called geometric accuracy, which defines how close objects can come to touching.
The computation of these two later potentials is done by the IPC library toolkit [3], theoretical details of these are beyond the scope of this thesis, but it is worth to mention that the computation of those takes less that 2% of the total execution time per time-step so we will focus entirely on optimizing PolyFEM library which is the one that will do the minization solving, gradient and hessian computation of the Incremental potential contact 2.31. Once we ported polyFEM on GPU, IPC could benefit entirely on it as the minimization problem and hessian assembly are the bottleneck of the entire simulation for accurate surface-to-surface contact.

In order to keep track of the computation of the gradient and hessian of the incremental potential contact, we will divide 2.31 into several forms named as:

- **Inertia form:** $\frac{1}{2}(x^{t+1} - \hat{x})^T M(x^{t+1} - \hat{x})$
- **Body form:** $f_{ext}.x^{t+1}$
- **Elastic form:** $\Psi(x^{t+1})$
- **Friction form:** $\mathbf{D}(x^t, \hat{d})$
- **Contact form:** $\mathbf{B}(x^t, \hat{d})$

With these two additional potentials, these are the gradient and global hessian of the system respectively:

$$\nabla_x E(x^{t+1}) = \frac{1}{2} M(x^{t+1} - \hat{x}) - h^2 f_{ext} + h^2 \nabla_x \Psi(\mathbf{F}) + \nabla_x(\mathbf{D}(x^t, \hat{d}) + \mathbf{B}(x^t, \hat{d})) \tag{2.32}$$

$$\mathbf{H}_x E(x^{t+1}) = \frac{1}{2} M + h^2 \mathbf{H}_x \Psi(\mathbf{F}) + \mathbf{H}_x(\mathbf{D}(x^t, \hat{d}) + \mathbf{B}(x^t, \hat{d})) \tag{2.33}$$

Chapter 3

PolyFEM library

As stated on the introduction of the thesis, PolyFEM provides the computational framework and user interface for IPC toolkit. In order to start a simulation, the user must give a set of meshes and parameters regarding the linear solver method, boundary conditions, time steps, material properties, etc, which should be contained in a text-based format supported by polyFEM such as JSON.
To manage vectors and matrices (dense and sparse) allocations and operations along the source code, Eigen 3.4.0 [4] is used, this later version is prefered due to its new implementation for CUDA kernel compatibility.
All of this information is managed and validated by the main function which will initialize the respective parameters and deploy the diverse polyFEM submodules in the following order:

- **Mesh loading:** This module will look for mesh object files mentioned by the user, which contains the information of the cells, vertices and faces of a certain discretized structure, all of this will be stored in arrays of matrices for further computation.

- **Build basis functions and quadrature vector:** It is in charge of setting the corresponding basis to approximate the displacement vector as stated in 2.13 and the quadrature values to approximate integral calculations 2.17. By default spline basis are built following the Poly-spline method [5] which has demonstrated to be robust against interpolation errors for bad element quality by increasing basis construction complexity of spline method. It is important to mention that the mesh cells are hexahedra which works with node positions for quadratic bases ($\mathbf{Q}_2$) in 2D or 3D as shown in Fig. 3.1.

- **Assemble mass matrix and right hand side:** Both are necessary to complete the computation of the term M and f_{ext} present in equation 2.24. For the default basis construction, these functions are locally supported, as a result most of the pairwise integrals are zero, leading to a sparse matrix.
  This non-zeros integral could be expressed as a sum of integrals over the mesh elements, allowing that for a given element a local mass matrix to be assembled. A global mapping defined on a reference element has to be taken into account to yield the global mass matrix entries, for this reason a transpose inverse Jacobian matrix should multiply each basis due

to the change on the local variable (translation).
Regarding the right hand side assembler, this contains the information of the Neumann and Dirichlet boundary conditions, commonly presence of external forces. The construction of the latter is a independent per-element assembly.

- **Solving problem:** This module will check for which kind of problem we are trying to solve, if it is time dependent and its formulation. As we will focus on the solution of Neo-Hookean simulations by means of incremental potential approach, the class which we will work on this thesis is the transient non-linear tensor problem. This class will perform the following initialization:
  - **It will check if the initial mesh configuration contains intersections.**
  - **Initializes the classes which solves the value, gradient and hessian of each form as expressed in equation 2.5, specifically for Neo-Hookean formulation**
  - **It will deploy the time integrator module, which by default is the Implicit Euler, by setting the updates of positions and velocities as it is in 2.21a**

  Once done this, for each time-step it instantiates a non-linear solver (Default: Newton descent method) which will perform the minimization algorithm 3. Once the updated value is computed, the time integrator will update as well the velocities and acceleration.
- **Export data:** This last step manages all the displacements data along time-steps in a visual tool markup format to be reproduced by software like Paraview [6].

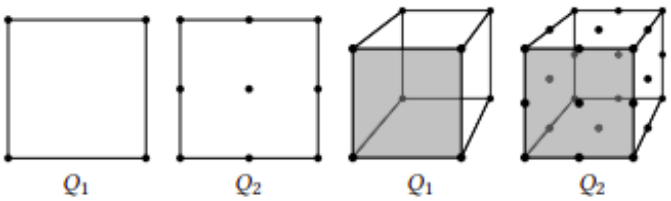


Figure 3.1: Node positions for each mesh element, each point represents a basis function which explicit formulation can be found on appendix C of [5].

Algorithm 1 Non-linear solver

SparseMatrix init $H(E)$ **Vector init** $\nabla E, \Delta x$ **Scalar** E , $stepsize$, tol
Data: Vector x , ProblemClass $Prob$, LinearSolver LS , LineSearch BT
ConstrainUpdate(x) `// Updates the constrain set for the friction form`
∇E <- $Prob$. ComputeGradient(x)
  CheckConvergence(∇E)
  `/* Checks if` ∇E `is not small enough to meet convergence criteria */`
while $\Delta x < tol$ **do**
  SystemComputation(x)
  E <- $Prob$. ComputeValue(x)
  ∇E <- $Prob$. ComputeGradient(x)
  $H(E)$ <- $Prob$. ComputeHessian(x)
  Δx <- LS. solve(E, ∇E,$H(E)$)
  LS . CheckSolution()
  if $\Delta x.\nabla E > 0$ **then**
    print("Not a descent direction")
    continue
  $stepsize$ <- BT. BacktrackingLineSearch(x, Δx,E)
  x <- $x + stepsize\ \Delta x$
`/* LS.solve solves` $H\Delta x = -\nabla E$ `*/`
return x

3.1 Benchmarking and design

In this section we will show which sections of the code takes the most time to compute, currently parallelized by using C++ threads or threading building blocks oneAPI software, as well to give an insight of this library design. Once we determined the hot spots and bottlenecks of the implementation, we filtered which functions has the higher compute-to-memory ratio, suitable for GPU heavy computation.
For this, we used a stress case of an armadillo creature with 65K mesh elements falling to a couple of rolling rods to exert extreme compression to the figure. To solve linear systems we relied on PARDISO MKL external library which uses LDL^T factorization. Coupled with Valgrind we ran 3 time-steps of the stress case to obtain a profile report generated by Callgrind.

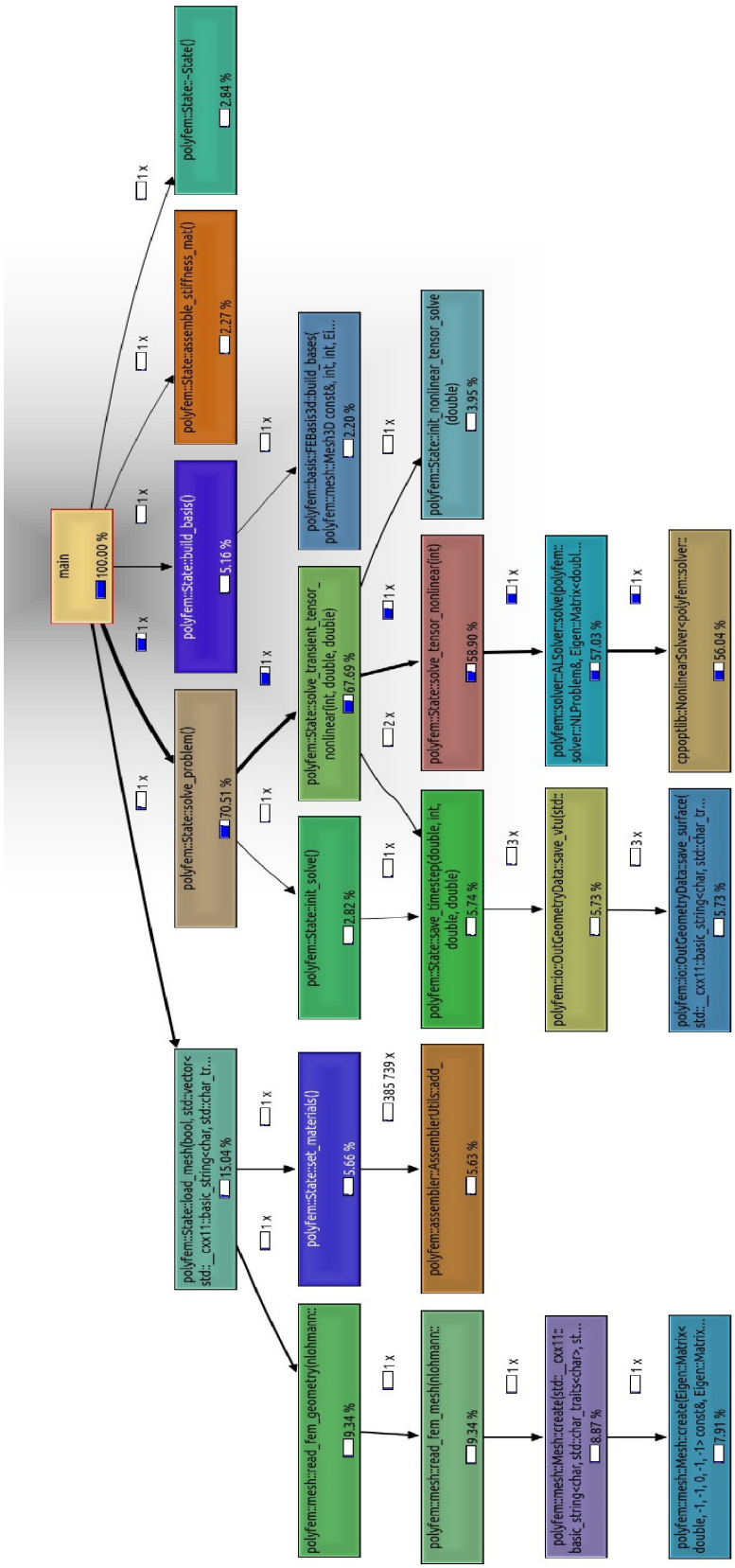


Figure 3.2: Call graph of the main modules for PolyFEM library.

According to Fig 3.2, solve problem class is the one that takes ∼70% of the computation (measured in CPU cycles) and it has been run for 3 time-steps (original is 400 time-steps) so it is guaranteed this relative percentage will increase up to 95% because mesh loading, mass matrix assembly and basis construction are a one time computation modules during the whole run. From now on we totally focus on the solve problem class implementation to find where are the hot-spots.

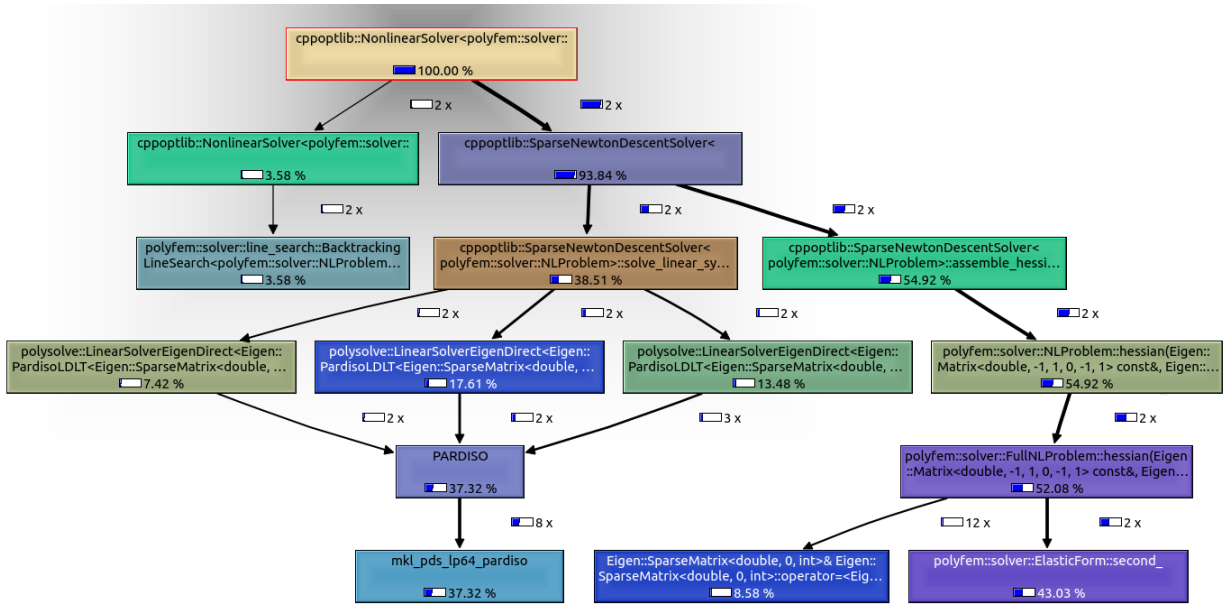


Figure 3.3: Call graph of the non-linear solver class of PolyFEM library.

The solve problem class follows the steps written in items 3, the initialization process and time integrator set up are negligible according to the report, therefore the non-linear solver takes almost the whole time execution of this.
As shown in 3.3, the non-linear solver has two main components, line search algorithm and newton descent for the minimization problem, despite the line search for this report takes barely ∼4% of the computation, it is important to mention this report was run for few timesteps, later benchmarks will show that line search could take up to ∼15% - ∼20% of the non-linear solver, for this reason we will treat this component as well.
Firstly the important improvements should be on the newton descent solver, where we can find two hotspots:

- **Linear solver:** PolyFEM relies on external libraries to solve Sparse linear systems, for this reason a toolkit called PolySOLVE [7] was implemented to take care of the necessary wrappers for integrating these solvers to Eigen vectors and sparse matrices.
  On CUDA implementation chapter we will return to this point to make use of libraries that contains CUDA backends such as AMGCL [8].

- **Hessian Assembly:** To solve equation 2.22 it is required to obtain the global sparse hessian matrix of that function which is in the order of Nº mesh elements x Nº mesh elements. As stated in chapter 3, it is possibly to assembly this matrix by computing independently each local hessian matrix for mesh element and set a global mapping to obtain the full matrix.
  This approach makes the hessian assembly embarrassingly parallel as we can take advantage of a GPU by allocating thousand of threads per block (NVIDIA current technology allows max 1024 threads per block) and each block could compute 1024 local hessian elements, let's set an example to appreciate GPU's computational power, currently this mesh test has 65K elements , for a 16 threads CPU this means that each thread have to compute 4K local hessian, a GPU instead could allocate 65 blocks which will compute the entire amount of local hessians for this mesh at once.

3.1.1 Hessian matrix for elastic form

Algorithm 2 Assembly Hessian for Elastic form

SparseMatrix init H , $localH$ **Vector init** Qda , $Jacit$, $LocalGrad$, x
Data: Displacement u , AssemblyUtils $AUtils$, MeshCache $Mcache$
TBBThreadAllocation() `// Start of the multithreading function`
`/* Each thread will have a local storage containing the respective computed data as local Hessian or gradient */`
for *MeshElement* **in** *MeshCache* **do**
  SystemComputation(u, $MeshElement$)
  $JacIT$ <- $Autils$. ComputeLocalMapping(MEl)
  Qda <- $Autils$. ObtainQuadratureVector(MEl)
  $LocalGrad$ <- $Autils$. ObtainLocalGradient(MEl)
  x <- $MeshCache$. LocalDisplacement(MEl, u)

  $localH$ <- LocalHessianComputation(x ,$LocalGrad$,Qda , $JacIT$)

  $row, col, value$ <- GlobalMapping($MeshElement$,$LocalH$)

  AddValues($row, col, value, H$)
TBBThreadFree() `// End of the multithreading function`
H.CompressedSparseMatrix()
`/* Converts the sparse matrix from a triplet version to a compressed column storage */`
return H

Additionally, Callgrind gave us a insight, as the computation of the hessian matrix is made by parts regarding each form as stated in the list 2.5, the one that is the most computational heavy is the elastic form, this means the computation of the second derivative of the strain density energy function for NeoHookean models 2.8. The algorithm followed by the polyFEM multithreading implementation is shown in algorithm 2, in the next chapter we will modify this approach to make it suitable for a entire computation on GPU.

The function LocalHessianComputation in algorithm 2 refers to compute equation 2.28, which as well it needs to be multiplied by the quadrature vector to obtain the approximation of the integral. Therefore is no surprise that this form is the heaviest to compute, due to several matrix operations for each mesh element, this is perfect to implement and compute in the GPU.

3.1.2 Backtracking line search

This implementation tries to find the step-size in which the energy decreases in a time-step as stated in equation 2.30a, as we increase the number of time-steps and the possible number of iterations to achieve convergence of the minimization of IPC, the line search becomes more expensive computationally as it is constantly evaluating the value of IPC 2.31 and its gradient for cases where certain tolerance is met, as shown in Fig. 3.4. The steps to compute this section can be found on algorithm 3

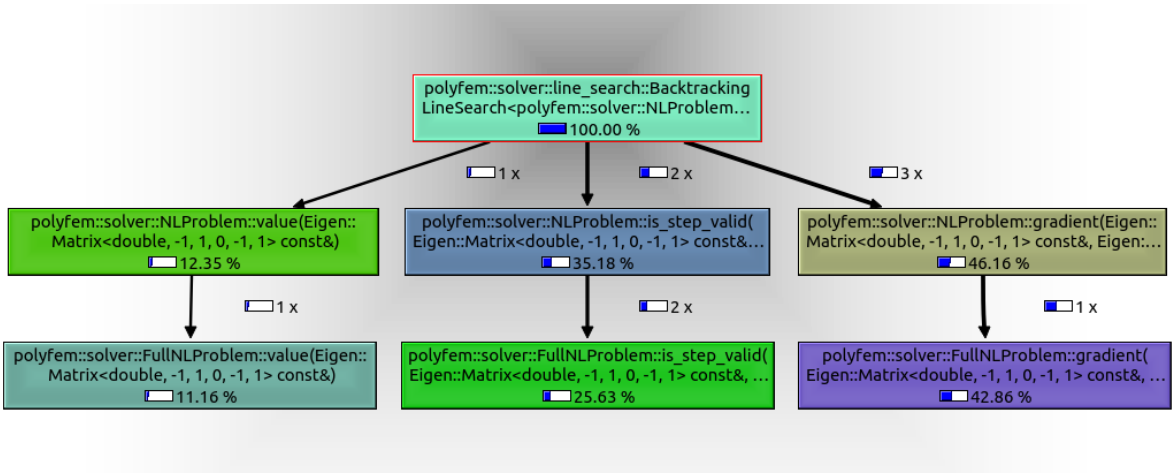


Figure 3.4: Call graph of the backtracking line search class of PolyFEM library.

Algorithm 3 Backtracking line search

```
Vector init ∇E , NEWx
Data: Vector Δx x , Scalar OldEnergy , stepsize =1, ProblemClass Prob
∇E <- Prob . ComputeGradient(x)
   /* If the norm of the gradient is less certain tolerance it will be used
   instead of the energy value for updating the line search              */
bool UseGradNorm <- ∇E < tol
 OldEnergy <- UseGradNorm ? ∇E.norm : OldEnergy
 while iterations < MAX ITERATIONS do
    NEWx <- x + stepsizeΔx
     if UseGradNorm then
        ∇E <- Prob.ComputeGradient(NEWx)
         Energy <- ∇E .norm()
    else
        Energy <- Prob.ComputeValue(NEWx)
    if Energy > OldEnergy then
        stepsize <- stepsize/2
return stepsize
```

Chapter 4

CUDA implementation

In the previous chapter we found the algorithms suitable for GPU computation, they are heavy on computation with respect of memory access and takes most of the total execution time, now it is time to show the CUDA kernel design for each of them, source code of the GPU implementation can be found here : https://github.com/AlexTru96/polyfem.
Before moving on, it is worth to mention the following characteristics of the software building tools used for this project:

- **CMAKE:** Minimum version : 3.19 , currently used for developing: 3.24.1
- **GCC compiler:** Version used for developing: 9.4.0
- **NVCC compiler:** Version used for developing: 11.7

4.1 CUDA tools and libraries

The main strategy for this project is to re-engineer the sections mentioned before to make it suitable for CUDA device functions taking advantage of libraries such as CuSparse, Eigen and Thrust.

- **CuSparse:** Will be used for sparse matrices sum, as each form will compute its own global hessian matrix it will be at least 5 matrices to sum of order Nº mesh elements x Nº mesh elements for each iteration in the newton descent solver.
- **Eigen:** 3.4 release [4] contains a unified GPU backend which allows support for matrix allocations and operations inside CUDA kernels (tested on this project for dense matrices with size less than 20x20).
- **Thrust:** A CUDA Toolkit for data transfer management and provides data parallel primitives for reduce and sort operations. Regarding data, Thrust vector framework has similar interface as standard library vector for C++ to send data structures to GPU or CPU, additionally it is compatible with std::vector allocations which provides flexibility and allows a

better approach to port already existent data structures to GPU, for example Eigen::Vector and Eigen::Matrix.

4.2 Sending data to GPU

There are parameters, vector and matrices that will be present during the whole simulation as they are needed in every time-step, for this reason we created a new function in charge to send to the GPU these variables that will last the lifetime of the simulation. We got the following list of data that is moved forward to the GPU at the start, all of them are vector of size Nº of mesh elements. (Considering 3D dimension)

- **Vectors of parameters (Vector <Double>, Size: nº mesh elements):** μ , λ, ρ
- **External forces (Structure: Vector<Eigen::Matrix(3,3)>):** f_{ext}
- **Quadrature (Vector <Eigen::Matrix(3,1)>):** $\mathbf{Qda}$
- **Gradient of the basis (Structure: Vector<Eigen::Matrix(3,3)>):** $\mathbf{Grad}$
- **Evaluation of the basis on the quadrature points(Structure: Vector <Eigen::Matrix(3,1)>):** $\mathbf{Val}$
- **Index and values of the basis according to global mapping (Structure: Vector <(int , double)>):** $\mathbf{GlobalData}$

To keep track of all this data once sent, a data structure is required to store the GPU pointers and to use it freely as an argument in any module that could require it for further computation, this structure is called DATA_POINTERS_GPU on the source code

Algorithm 4 Solve transient non-linear tensor problem

SendingDataToGPU *()*
    $\mathbf{ThrustVector}\ \mu, \lambda, \rho <- \mathbf{ThrustCopy}(\mathbf{std} :: \mathbf{vector} \mu, \lambda, \rho)$
    $\mathbf{ThrustVector}\ forcesGPU <- \mathbf{ThrustCopy}(\mathbf{std} :: \mathbf{vector}\ forces)$
    $\mathbf{ThrustVector}\ QdaGPU <- \mathbf{ThrustCopy}(\mathbf{std} :: \mathbf{vector}\ Quadrature)$
    $\mathbf{ThrustVector}\ GradGPU <- \mathbf{ThrustCopy}(\mathbf{std} :: \mathbf{vector}\ Grad)$
    $\mathbf{ThrustVector}\ ValGPU <- \mathbf{ThrustCopy}(\mathbf{std} :: \mathbf{vector}\ Val)$
    $\mathbf{ThrustVector}\ GlobalDataGPU <- \mathbf{ThrustCopy}(\mathbf{std} :: \mathbf{vector}\ GlobalData)$

```
/* After sending the data, the following function will update the data
   structure containing all the allocated pointers                    */
```

$\mathbf{Update}(\mathbf{DataPointersGPU})$
 $\mathbf{InitializeSolver}()$
 for *t* in *TOTAL TIMESTEPS* **do**
    NewtonMinimize(DataPointersGPU)
    UpdateQuantities()

4.3 Hessian computation

Once the data is already allocated in the GPU we can use the pointers directly as arguments for CUDA kernels, but first we set the strategy to change a thread parallelization in the algorithm 2 to a kernel function.
As each thread will store its own local hessian, sending this data back to CPU would create a bottleneck as this transference has to be repeated every newton descend iteration, the ideal scenario is to send back only the vector of non-zero values and compress the sparse hessian matrix back on the CPU.

4.3.1 Mapping for hessian computation

A solution for the last was to send the mapping of the already assembled hessian matrix on CPU, taking advantage of the constant mapping of elastic form hessian matrix, this means the rows and columns index of the non zero values does not vary throughout time-steps. Therefore, we call the hessian computation only once as multithreading and we store the row and column index for the global mapping and the order of computation which indicates the corresponding mesh element.
The obtained structure (mapping) is a vector of vector of pairs, if we consider i as a row and j as a column, mapping[i] is the vector of pairs that contains the index of the columns of non zeros values(first) and index counter (second), counting from 0 to the number of nonzero values considering a sweep from row to columns.

Extraction and allocation of the mapping structure to the GPU can be found in the function assemble_energy_hessian_GPU on AssemblerUtils.cpp and algorithm 5 can be found on the file NeoHookeanElasticity.cu

Algorithm 5 GPU Assembly Hessian for Elastic form

SparseMatrix init H **Vector init** $NonZeroValues$
Data: Displacement u , DataPointerToGPU $GPUData$, Vector< Vector<Pair> > $Mapping$
/* Mapping is allocated and copied in CPU and GPU */
$\mathbf{ThrustVector}\ u' <- \mathbf{ThrustCopy}(\mathbf{Eigen} :: \mathbf{vector}\ u)$
$\mathbf{ThrustVector}\ NZeroValGPU <- \mathbf{ThrustCopy}(\mathbf{std} :: \mathbf{vector}\ NonZeroValues)$

NT<- 32 // A warp
NB<- NMeshElements / 32 + 1 // Blocks required to compute each local hessian
HessianComputationKernel< < <NT,NB> > > $(GPUData,\ Mapping,\ u',\ NZeroValGPU)$
  Each GPU thread
  $localH$ <- LocalHessianComputation(u' ,$LocalGrad$,Qda , $JacIT$)
  $\mathbf{ComputedValuesReduction}$:
  $NZeroValGPU(Mapping : Index) <- -\mathbf{ATOMICADD}(localH, Mapping : (i, j))$
NonZerosValues <- CopyToCPU(NZeroValGPU)
H.ReplaceNonZeroValues(NonZerosValues)
H.CompressedSparseMatrix()
return H

4.4 Backtracking line search

According to the valgrind [14] report on 3.4 and the algorithm shown on 3, this method makes use constantly of the value and gradient computation of the IPC, for this reason the quickest way to optimize this portion of the code is by directly implementing a GPU version of those, taking into account that elastic and body form are the hot-spots for this calculation.

4.4.1 Value and gradient computation

Unlike the hessian assembler, this computation does not require a mapping and the sum reduction operation can be done straightforward inside the kernel

Algorithm 6 GPU Gradient and value computation for elastic form

Vector init ∇E **Scalar init** E
Data: Displacement u , DataPointerToGPU $GPUData$
ThrustVector $u' <-$ **ThrustCopy**(**Eigen** :: **vector** u)
ThrustVector $GradEGPU <-$ **ThrustCopy**(**Eigen** :: **vector** ∇E)

NT<- 32 `// A warp`
NB<- NMeshElements / 32 + 1 `// Blocks required to compute each local hessian`
GradientComputationKernel< < <NT,NB> > > $(GPUData,\ u',\ GradEGPU)$
  Each GPU thread
  $localGrad$ <- LocalGradComputation(u' ,$Grad$,Qda , $JacIT$)
  ComputedValuesReduction :
  $GradEGPU <-$ **ATOMICADD**($localGrad$)
ValueComputationKernel< < <NT,NB> > > $(GPUData,\ u',\ EGPU)$
  Each GPU thread
  $localEnergy$ <- LocalEnergyComputation(u' ,$Grad$,Qda , $JacIT$)
  ComputedValuesReduction :
  $EGPU <-$ **ATOMICADD**($localEnergy$)
∇E <- CopyToCPU(GradEGPU)
E <- CopyToCPU(EGPU)
return $\nabla E, E$

CAVEAT: The reduction for summing between and within blocks used on the latter is only available for GPU's with NVIDIA compute capability 6.0 or higher.

Chapter 5

Polysolve library

One main step of newton descent method is to solve the linear system for each iteration as shown in 2.25, polyFEM works with structures as Eigen::Vector, Eigen::SparseMatrix for that some external libraries provides Eigen backend to operate directly with these structures such as: **PARDISOMKL**, **SuperLU**, **CholmodSupernodalLLT**, but it is not always the case and wrappers from Eigen to the required data structure are necessary to make use of these libraries. For this reason to create a standard linear solver function able to use any external library independent if it has a Eigen backend or not, polySOLVE toolkit has been developed to take care of this issue.
On this project we will modify this library to make use of CUDA backend available on some external projects such as **AMGCL** [8]. Source code used on this project can be found on : https://github.com/AlexTru96/polysolve/tree/polyFEM_GPU

5.1 Linear solvers available

Currently polySOLVE contains wrappers for a bundle of external sparse linear solvers, direct and iterative ones. Which has its own pros and cons respectively

5.1.1 Direct solvers

Those work mostly on a decomposition of the linear system matrix and apply forward or backward substitutions. Are known to consume more memory and have little room to parallelization strategies because of data dependencies, but these methods are very robust for any kind of matrix independent of its sparsity pattern.

- **PARDISO MKL** : Contains factorizations of kind LDL^T, LL^T, LU
- **SuperLU** : LU factorization
- **UmfPackLU** :LU factorization
- **CholmodSupernodalLLT** :LL^T factorization

5.1.2 Iterative solvers

On the contrary, these methods are more easy to parallelize and make good usage of GPU computation, but these methods not will always converge and it is heavily dependent of a good preconditioner.

- **AMGCL** : Focus on the algebraic multigrid method
- **HYPRE** : Contains solvers such as generalized minimal residual method, conjugate gradient, Jacobi method, etc.
- **Eigen :** Inner sparse linear solvers such as conjugate gradient and stabilized bi-conjugate gradient

5.1.3 Eigen sparse matrix wrapper for AMGCL

For example **AMGCL** library does not contain a Eigen backend to manage the original variables of polyFEM directly, for this reason on the algorithm 7 is shown how this is managed for this specific case:

Algorithm 7 Eigen wrapper for AMGCL linear solver

Data: Eigen::SparseMatrix A , Eigen::Vector b, x JSON PropertiesList

CreateLinearSolver *()*

$\mathbf{Array} < \mathbf{StorageIndex} > OuterVector <- \mathbf{A}.outerIndexPointer()$
$\mathbf{Array} < \mathbf{StorageIndex} > InnerVector <- \mathbf{A}.innerIndexPointer()$
$\mathbf{Array} < \mathbf{Scalar} > ValueVector <- \mathbf{A}.valuesPointer()$
$\mathbf{AdquireProperties}(PropertiesList)$
$\mathbf{auto}\ A' <- \mathbf{std} :: \mathbf{tie}(numRows, Inner, Outer, Values)$
$\mathbf{auto}\ Am <- \mathbf{AMGCL}.Adapter(A')$
$\mathbf{AMGCLSolver}\ S <- \mathbf{MakeSolver}(Am)$

```
/* Once created the solver, we use the backend to copy b and x vectors     */
```

$\mathbf{Vector} < \mathbf{Scalar} > b\_ <- \mathbf{amgcl} :: \mathbf{BackendCopy}(\mathbf{b})$
$\mathbf{Vector} < \mathbf{Scalar} > x\_ <- \mathbf{amgcl} :: \mathbf{Solve}(S, b\_)$

5.2 Using AMGCL CUDA backend

For this project we take advantage of the **AMGCL** CUDA backend, which uses cuBLAS for the matrix operations throughout the solving process. In the algorithm 8 we point out how to use this backend for polySOLVE.

Algorithm 8 AMGCL CUDA backend

Data: Eigen::SparseMatrix A , Eigen::Vector b, x JSON PropertiesList

$\mathbf{using\ Backend} = \mathbf{amgcl} :: \mathbf{backend} < \mathbf{CUDA} >$

$\mathbf{using\ Prop} = \mathbf{amgcl} :: \mathbf{preconditioner}, \mathbf{amgcl} :: \mathbf{solver}$

$\mathbf{using\ AMGCLSolver} = \mathbf{amgcl} :: \mathbf{make\_solver} < \mathbf{Prop}, \mathbf{Backend} >$

$\mathbf{CreateLinearSolver}()$

$\mathbf{Vector} < \mathbf{Scalar} > b\_ < - \mathbf{amgcl} :: \mathbf{BackendCopy}(\mathbf{b})$

$\mathbf{Vector} < \mathbf{Scalar} > x\_ < - \mathbf{Thrust} :: \mathbf{copy}(\mathbf{amgcl} :: \mathbf{Solve}(S, b\_))$

Chapter 6

Results and GPU benchmarking

We will evaluate time execution performance of the CUDA implementation using 2 CPU models to test how much impacts GPU computation in comparison with only a multi-threaded(MT) version. For the MT case we will use PARDISO with LDL factorization as sparse linear solver as this was the fastest solver tested for this specific scenario, and AMGCL with CUDA backend to benchmark the new implementation.

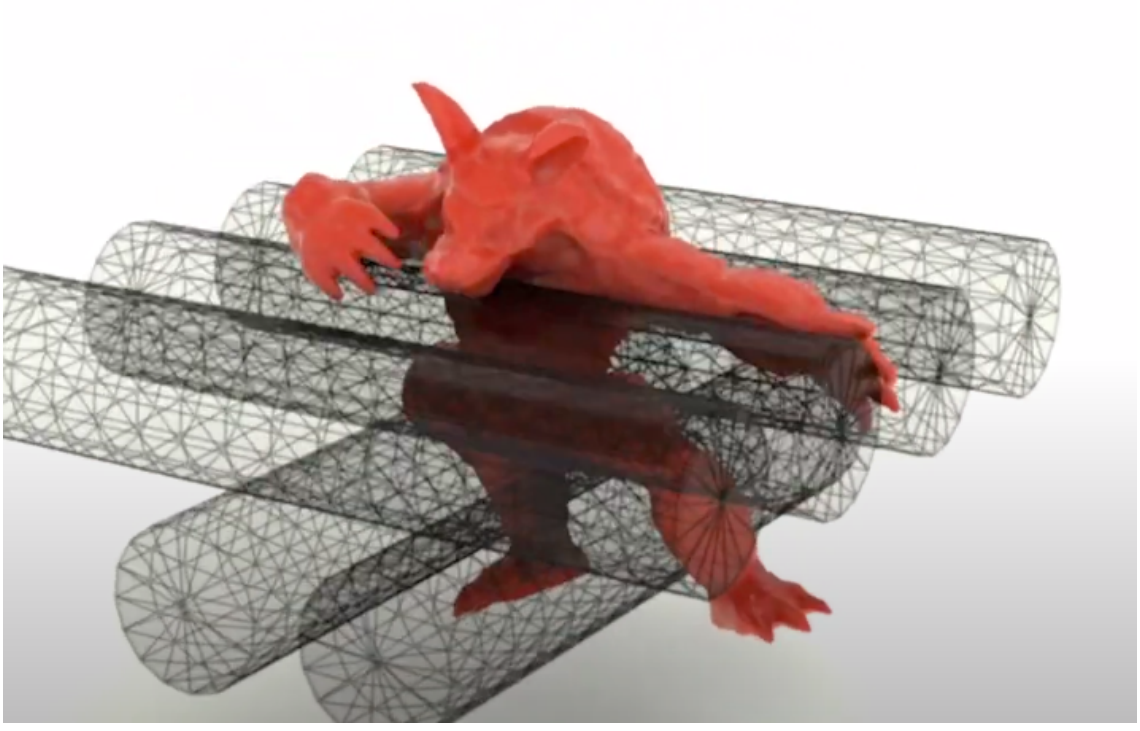


Figure 6.1: Stress case for benchmarking: armadillo compression by falling to rollers. Software: Paraview.

6.1 Stress benchmarking

Implementation	Hardware Specifications	Case simulation
Only multithreading (4T)	CPU: AMD(R) EPYC(TM) 7452 CPU: Intel Core(TM) i7-5930K	Armadillo creature with 63K elements compressed by rollers (40 time-steps, solver: Pardiso LDL^T)
Multithreading (4T) + GPU	CPU: AMD(R) EPYC(TM) 7452 / GPU : NVIDIA RTX A5000 CPU: Intel Core(TM) i7-5930K / GPU: NVIDIA GeForce GTX 1080 Ti	Armadillo creature with 63K elements compressed by rollers (40 time-steps, solver: Algebraic multigrid with sparse approximation inverse as preconditioner (AMGCL))

Table 6.1: Stress cases to test GPU performance

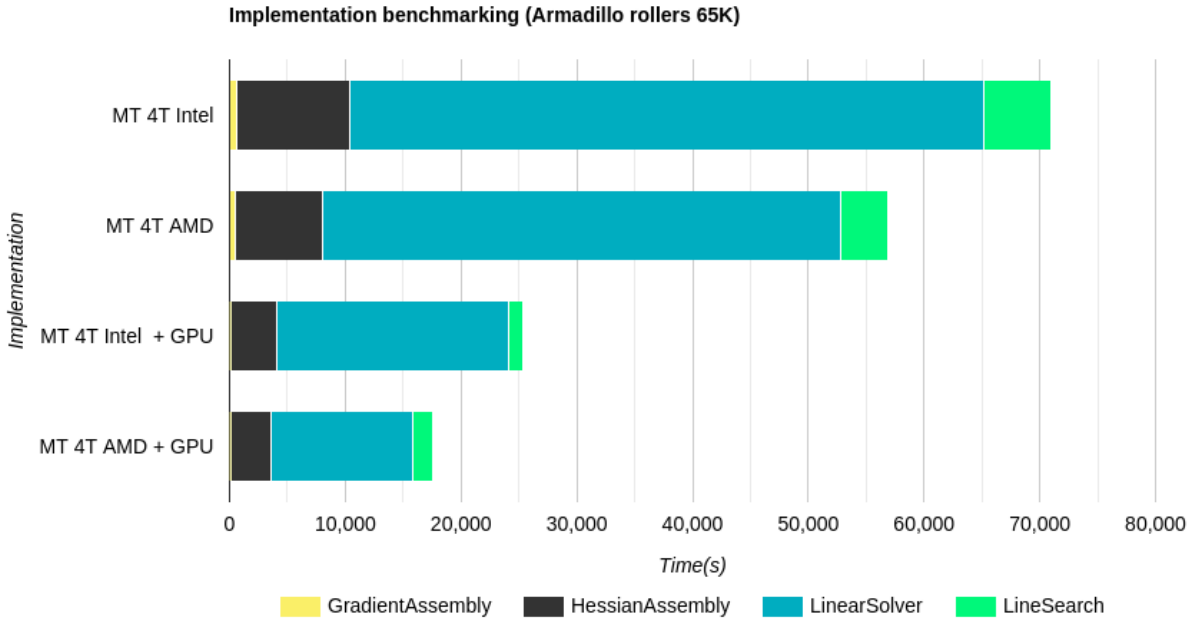


Figure 6.2: Comparison between multithreading (MT) and MT + GPU implementation for cases shown in table 6.1.

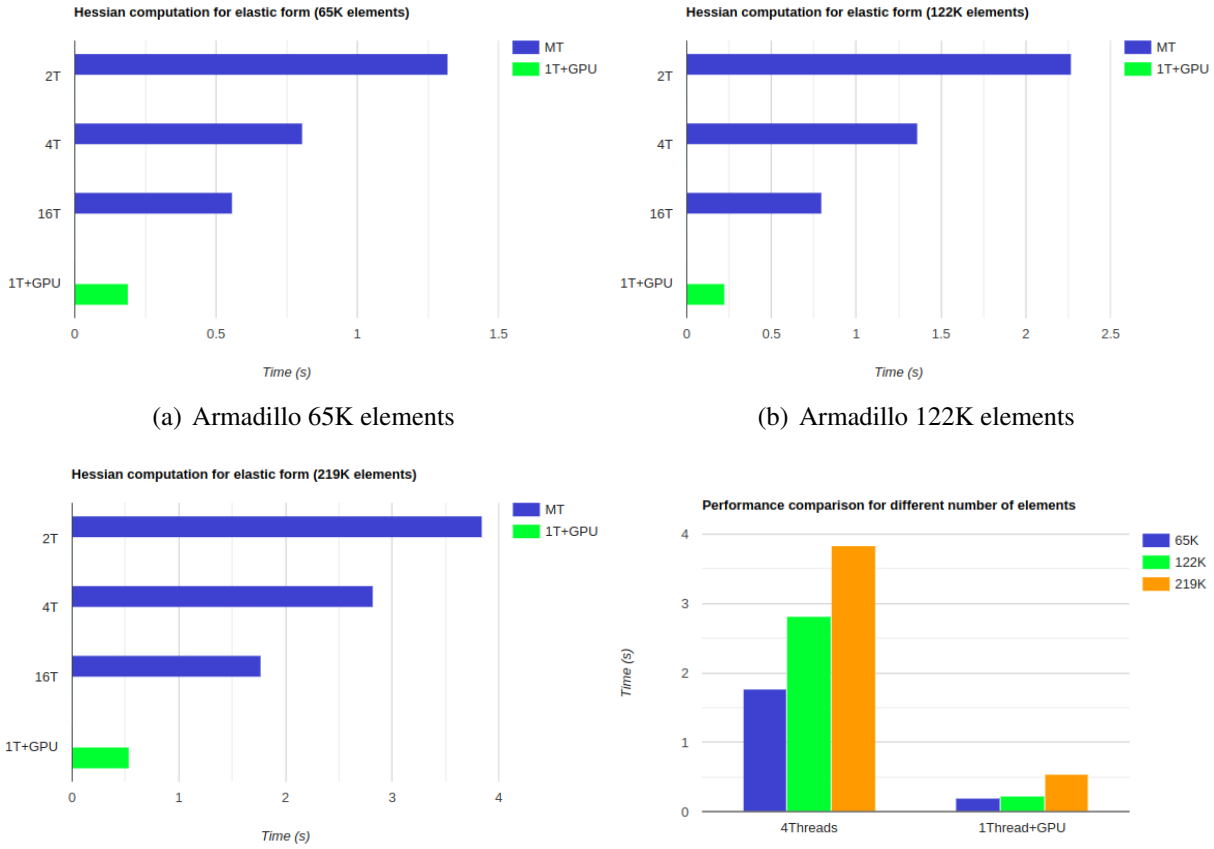
As shown in Fig. 6.2 the sections of the code that takes the most advantage of the new implementation are the hessian assembly and the linear solver required for newton descent solver. A important aspect to highlight is related to execution time of the assembly hessian as it equalizes between two CPU models, this mostly because the whole computation is being done in GPU leaving only the compression of the hessian matrix to CPU. This indicates great news for users with a GPU, a low or medium end CPU could achieve the same performance as an expensive high-end CPU as shown on this benchmarking.

6.2 Size benchmarking (hessian computation)

As mentioned on chapter 4, the CUDA implementation of the Hessian computation for the elastic form it allows to allocate one thread per mesh element, in certain sense it means that the computational time complexity of this algorithm is constant but bounded by the number of CUDA cores and memory that GPU has. This is a considerable change in comparison with the linear complexity of the previous parallel implementation, it is expected to have an overhead but as long as the number of elements are less than the maximum concurrent number of GPU threads the computation time should be constant plus thread allocation latency.

Implementation	Hardware Specifications	Case simulation
Only multithreading (2T , 4T , 16T)	CPU: AMD(R) EPYC(TM) 7452	Armadillo creature with 63K elements Armadillo creature with 122K elements Armadillo creature with 219K elements
Single thread + GPU	CPU: AMD(R) EPYC(TM) 7452 / NVIDIA RTX A5000	Armadillo creature with 63K elements Armadillo creature with 122K elements Armadillo creature with 219K elements

Table 6.2: Performance test for different number of mesh elements for the hessian computation of the elastic form



(a) Armadillo 65K elements

(b) Armadillo 122K elements

(c) Armadillo 219K elements

(d) Multithreading vs GPU

Figure 6.3: Performance benchmarking of the hessian computation of strain density energy function for cases shown in table 6.2

As shown in the Figure d on 6.3, the explanation of the performance behavior for GPU implementation for different sizes is that computation time is constant but as the number of threads required increases as well it did the thread allocation overhead.

6.3 Counterproductive for small cases

One important aspect for GPU computation is to make sure the ratio between computation and memory read-write is high enough but as well we must consider the data transfer bandwidth between CPU and GPU, even if the data is small enough it exists a latency to take into account. According to the algorithm 5 and 6, the displacement , gradient and non zero values vector needs to be transferred to GPU to compute the corresponding function or copied back to CPU in each iteration. To prove the impact of this data transfer overhead we will show a test simulation of a walking elephant, which has approximate 7K elements.

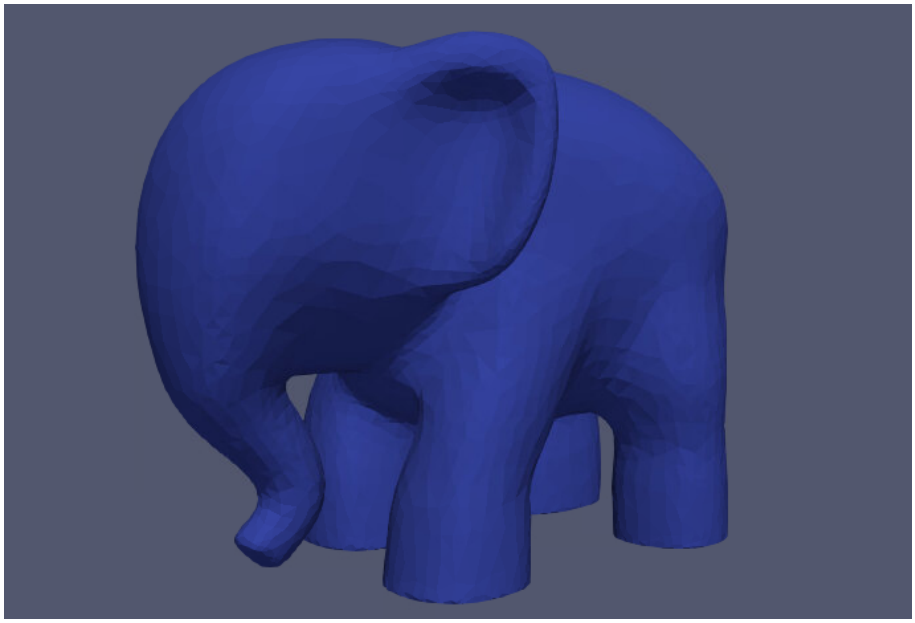


Figure 6.4: Walking elephant simulation, current object has 7K mesh elements. Software: Paraview.

Implementation	Hardware Specifications	Case simulation
Only multithreading (4T)	CPU: AMD(R) EPYC(TM) 7452	Elephant creature with 7K mesh elements 40 time-steps, linear solver: PARDISO LDL
4 threads + GPU	CPU: AMD(R) EPYC(TM) 7452 , NVIDIA RTX A5000	Elephant creature with 7K mesh elements 40 time-steps, linear solver: AMGCL with CUDA

Table 6.3: Small case to test GPU performance

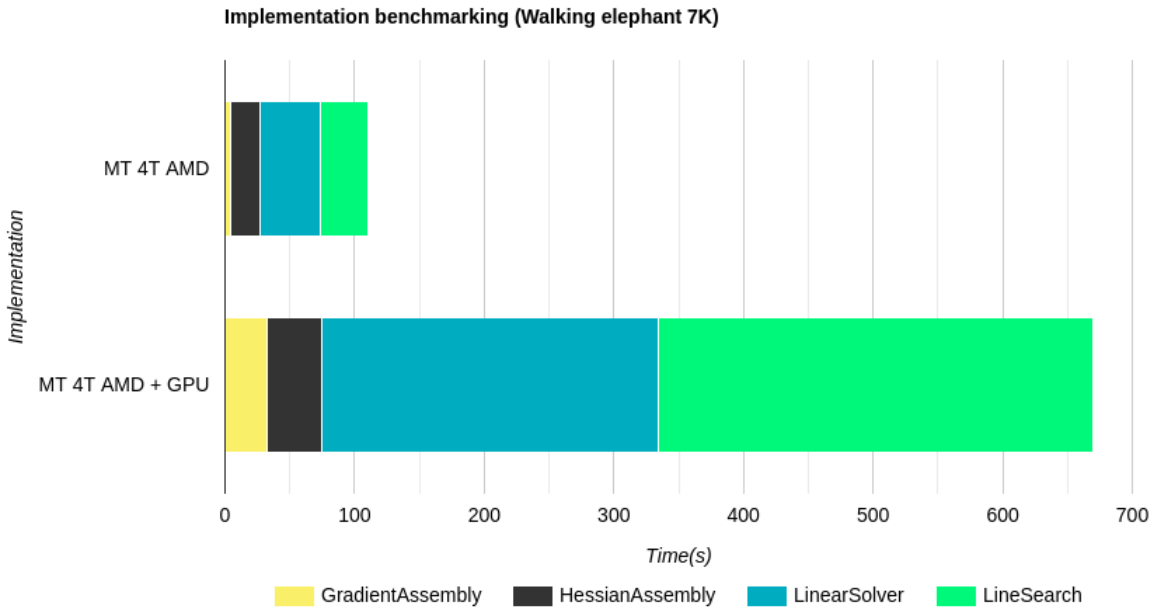


Figure 6.5: Downgrade of the performance of GPU implementation when the number of mesh elements is less than 7K.

As shown in Fig. 6.5 for this case GPU implementation was six times slower, the major slowdown occurred on line search computation as this currently calls the gradient and value computation and for those functions the computation is not as heavy as hessian one, the data transfer took a toll on the performance in general as the computation load was not enough to get benefit of a GPU.

Conclusions

- As shown in Fig. 6.2, polyFEM with CUDA implementation was able to speed up four times the original implementation considering parallelization with 4 threads for a AMD(R) EPYC(TM) 7452 model, it is worth to mention this high end CPU it has a cost market of approximate $2000 with a GPU NVIDIA RTX A5000 of $2500 and has similar performance in comparison with a Intel Core(TM) i7-5930K with NVIDIA GeForce GTX 1080 Ti which both together cost less than $1000.
  This proves that our implementation aims to low-medium end CPU users who can obtain great computational performance with a GPU.

- Fig. 6.3 demonstrates potential to further speedup for simulations of objects with higher number of elements as this is only bounded by memory and number of parallel processors a GPU has. But even if the number of elements is greater than the maximum number of GPU thread allocations this has to be order of magnitudes superior to create an important bottleneck on the computation.

- AMGCL with CUDA backend was up to 3 times faster than PARDISO MKL with LDL factorization (fastest multithreading implementation for this stress test), this by using sparse approximate inverse as preconditioner and relaxation for the linear solver. An important caveat arises, this preconditioner would not work for other simulations as it is highly dependent of the sparsity pattern and matrix conditioning.

- The performance shown in Fig 6.5 for small cases (<20K mesh elements), reveals that it is still preferred to work with a multi-thread implementation when the object has few elements, for that reason it is planned to implement a function to determine which version to run for a certain simulation according to the computational load.

Future work

- **PolySOLVE coupled with PETsc :** Currently a interface for PETsc is being implemented for PolySOLVE, providing a bundle of linear solvers available as well PETsc latest development [9] provides support for GPU accelerated sparse linear solvers, which some of them are direct solvers as CHOLMOD, in this way that solver could be used for any kind of simulation.

- **Generic GPU backend:** This project was only using CUDA as programming interface for GPU computation, so one of the future plans is to generalize this support for NVIDIA/CUDA and AMD/HIP.

Bibliography

[1] **Li, M., Ferguson, Z., Schneider, T., Langlois, T., Zorin, D., Panozzo, D., Jiang, C., amp; Kaufman, D. M. (2020).** *Incremental potential contact. ACM Transactions on Graphics, 39(4).* , https://doi.org/10.1145/3386569.3392425

[2] **Couro Kane, Jerrold E Marsden, Michael Ortiz, and Matthew West. 2000.** *Variational integrators and the Newmark algorithm for conservative and dissipative mechanical systems.*, Int. J. for Numer. Meth. in Eng. 49, 10 (2000)

[3] **Minchen Li; Ferguson Zachary.** *IPC-SIM/IPC: Incremental Potential Contact (IPC)* Retrieved December 3, 2022, from https://github.com/ipc-sim/IPC

[4] **Gaël Guennebaud and Benoît Jacob and others (2010)** *Eigen v3* Retrieved December 3, 2022, from https://eigen.tuxfamily.org/index.php?title=3.4

[5] **Schneider T., Dumas J., Gao X. , Botsch M, Panozzo D. and Zorin D. (2019).** *Poly-Spline Finite-Element Method. ACM Transactions on Graphics, 38(3).* , https://doi.org/10.1145/3313797

[6] **Henderson, A. (2007).** *ParaView Guide, A Parallel Visualization Application.* , Kitware Inc., 2007.

[7] **Teseo Schneider; Ferguson Zachary.** *Polysolve: Easy-to-use wrapper for linear solver* Retrieved December 3, 2022, from https://github.com/polyfem/polysolve

[8] **Denis Demidov.** *AMGCL - a C++ library for solving large sparse linear systems with algebraic multigrid method* Retrieved December 3, 2022, from https://github.com/ddemidov/amgcl

[9] **Richard Tran Mills and Mark F. Adams and Satish Balay and Jed Brown and Alp Dener and Matthew Knepley and Scott E. Kruger and Hannah Morgan and Todd Munson and Karl Rupp and Barry F. Smith and Stefano Zampini and Hong Zhang and Junchao Zhang (2021).** *Toward performance-portable PETSc for GPU-based exascale systems. Parallel Computing(108)*, https://doi.org/10.1016/j.parco.2021.102831

[10] **NVIDIA and Vingelmann, Péter and Fitzek, Frank H.P. (2020).** *CUDA, release: 10.2.89*, https://developer.nvidia.com/cuda-toolkit

[11] **Teseo Schneider; Ferguson Zachary.** *PolyFEM: A polyvalent C++ FEM library* Retrieved December 3, 2022, from https://github.com/polyfem/polyfem

[12] **J. Bonet and R.D. Wood, (1997).** *Nonlinear continuum mechanics for finite element analysis.*, Cambridge University Press, 1997.

[13] **Lim, H.; Hoag, S. W. (2013).** *Plasticizer effects on physical–mechanical properties of solvent cast Soluplus® Films. AAPS PharmSciTech, 14(3), 903–910.*, https://doi.org/10.1208/s12249-013-9971-z

[14] **Valgrind. (n.d.)** Retrieved December 3, 2022, from https://valgrind.org/info/developers.html