TRA220 GPU-accelerated Computational Methods using Python and CUDA 2024

GPU-Accelerated Computational Methods for FEM Using Python and CUDA

Afroditi Tzanetou
Arik Ben-Shabat
Oweis Al-Karawi
Róbert F. Birkisson
Simon Riis

Supervisor:

Fredrik Larsson

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2024

Abstract

In this project, CPU- and GPU-based numerical solvers for simulating elastic deformable objects under forces are developed using methods based on the Finite Element Method (FEM). The objects are discretized into a finite number of elements and the deformation is calculated, either using static methods or dynamic methods and either run sequentially on the CPU or in parallel on the GPU. The performance of the CPU-based implementations is then compared with the GPU-based ones for various problem sizes and models using FEM, including static and dynamic implementations for linear models.

Keywords: GPU, GPU-Acceleration, FEM, CUDA, CuPy

Contents

1 Introduction

1.1 Background

In this project, the use of parallel computing capabilities of Graphics Processing Units (GPUs) and CUDA is studied with the aim of making the Python-based Finite Element Method (FEM) programs faster and more efficient. FEM is a powerful tool for solving engineering problems and is utilized to handle static and dynamic scenarios. In the rapidly evolving landscape of computational methods, the integration of the advanced computing abilities of GPUs can be employed to unlock unprecedented computational power for scientific simulations and enhance the performance and speed of the FEM simulations. Therefore, the solution to complex problems in engineering and scientific applications can be handled in a robust and scalable manner, advancing the state-of-the-art in numerical analysis.

1.2 Goals

The main objectives of this project are:

- Introduction and formulation of one static and one dynamic structural problem and development of two FEM programs in Python for solving the two problems, respectively.
- Acceleration of the developed FEM programs by performing parallel computing in GPU using CUDA.
- Investigation of the efficiency of the developed FEM programs in CPU and GPU in terms of computational time.

1.3 Limitations

The study is performed under the following assumptions:

- The geometry domain is considered two-dimensional. The plane stress assumption is adopted.
- The material of the domain follows linear elastic isotropic behavior.
- Linear element approximations are utilized for solving the finite element model.

1.4 Resources

The evaluation of the developed code has been performed using the following computer system:

- CPU: 12 12th Gen Intel i5-12500
  - Memory: 128 GB
- GPU: NVIDIA GeForce RTX 3050
  - Memory: 8192 MiB

2 Theory

First, some relevant theory about the Finite Element Method (FEM) is presented in order to help understand the problem and the applied solutions.

2.1 Finite Element Problem - Static

In this section a description of the problem being studied is given. For the retrieval of the solution the Finite Element Method is utilized, and as an initial approach a static load is considered. Gravitational effects are neglected.
The geometry of the problem is an axisymmetric disc of outer radius 50 mm and inner radius 30 mm. For simplicity, one quarter of the disc is simulated as presented in Figure 2.1, applying the appropriate boundary conditions to simulate the axisymmetric geometry.

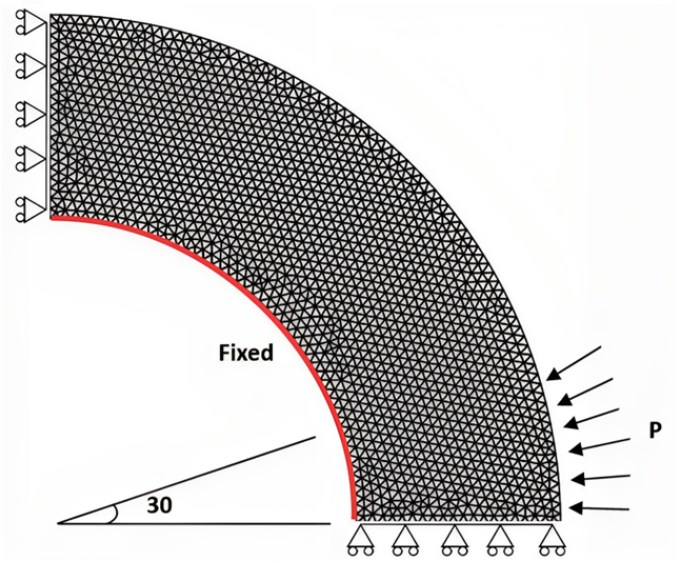


Figure 2.1: Axisymmetric disc problem.

The disc of the problem is made of steel with Young's modulus equal to 200 GPa and Poisson's ratio equal to 0.3, simulated to follow a linear elastic behavior. The plane stress assumption is considered for a plate with thickness 2 mm.
The disc is clamped at the edge of its inner radius. A static pressure is applied to the outer edge of the disc at an arc of 30 degrees in the static case, while in dynamic case, initial velocity is considered at the same position.
For the problem of the axisymmetric disc of volume V, the governing equations need to be solved as described in [1]:

$$\begin{cases} -\boldsymbol{\sigma} \cdot \nabla = \bar{\boldsymbol{b}} \; in \; V & (2.1) \\ \boldsymbol{t} = \bar{\boldsymbol{t}} \; on \; \Gamma_{\bar{t}} & (2.2) \\ \boldsymbol{u} = \bar{\boldsymbol{u}} \; on \; \Gamma_{\bar{u}} & (2.3) \end{cases}$$

where $\bar{\boldsymbol{b}}$ and $\bar{\boldsymbol{t}}$ are the prescribed volume and boundary force vectors, respectively.

In order to numerically solve the problem for the displacement field $\boldsymbol{u}$, the weak formulation of the governing equations need to be performed. The displacement field equilibrium Eq.(2.1) is integrated over the volume and multiplied with a kinematically admissible virtual displacement $\delta\boldsymbol{u}$. Using the Cauchy's stress theorem $\boldsymbol{t} = \boldsymbol{\sigma}\boldsymbol{n}$, the Gauss divergence theorem as well as taking into account that $\delta\boldsymbol{\epsilon} = \nabla^{sym}\,\delta\boldsymbol{u}$ for the strains $\boldsymbol{\epsilon}$, the weak form of the displacement field becomes:

$$\int_V \boldsymbol{\sigma}\;\delta\boldsymbol{\epsilon}\;dV = \int_V \bar{\boldsymbol{b}}\;\delta\boldsymbol{u}\;dV + \int_\Gamma \bar{\boldsymbol{t}}\;\delta\boldsymbol{u}\;d\Gamma \tag{2.4}$$

For the derivation of the finite element form of the equations, the weak form equation need to be expressed in Voigt format. For a 2-dimensional problem, the displacement and strain terms as well as their virtual components can be described as:

$$\boldsymbol{u}(x,y) = \begin{bmatrix} u_x(x,y) \\ u_y(x,y) \end{bmatrix}, \quad \delta\boldsymbol{u}(x,y) = \begin{bmatrix} \delta u_x(x,y) \\ \delta u_y(x,y) \end{bmatrix}$$

$$\boldsymbol{\epsilon}(x,y) = \nabla^{sym}\boldsymbol{u}(x,y) = \begin{bmatrix} \epsilon_{xx}(x,y) \\ \epsilon_{yy}(x,y) \\ \gamma_{xy}(x,y) \end{bmatrix}, \quad \delta\boldsymbol{\epsilon}(x,y) = \nabla^{sym}\delta\boldsymbol{u}(x,y) = \begin{bmatrix} \delta\epsilon_{xx}(x,y) \\ \delta\epsilon_{yy}(x,y) \\ \delta\gamma_{xy}(x,y) \end{bmatrix}$$

The displacement field of the disc domain is discretized and computed in $\boldsymbol{u}$ nodes using the Galerkin method such as:

$$\boldsymbol{u}(x,y) = \sum_{k=1}^{n} N_k^u(x,y) \begin{bmatrix} u_{x,k} \\ u_{y,k} \end{bmatrix} = \boldsymbol{N}^u\;\boldsymbol{a}^u \tag{2.5}$$

where

$$\boldsymbol{N}^u = \begin{bmatrix} N_1^u(x,y) & 0 & ... & N_n^u(x,y) & 0 \\ 0 & N_1^u(x,y) & ... & 0 & N_n^u(x,y) \end{bmatrix}, \quad \boldsymbol{a}^u = \begin{bmatrix} u_{x,1} \\ u_{y,1} \\ . \\ . \\ . \\ u_{y,k} \end{bmatrix}$$

Using the small strain definition, the strain field is expressed in terms of the displacement field as:

$$\boldsymbol{\epsilon} = \nabla^{sym}\boldsymbol{u} = \nabla^{sym}\boldsymbol{N}^u\;\boldsymbol{a}^u = \boldsymbol{B}^u\;\boldsymbol{a}^u$$

for

$$\boldsymbol{B}^u = \begin{bmatrix} \frac{\partial {N_1}^u(x,y)}{\partial x} & 0 & \frac{\partial {N_2}^u(x,y)}{\partial x} & 0 & ... & \frac{\partial {N_n}^u(x,y)}{\partial x} & 0 \\ 0 & \frac{\partial {N_1}^u(x,y)}{\partial y} & 0 & \frac{\partial {N_2}^u(x,y)}{\partial y} & ... & 0 & \frac{\partial {N_n}^u(x,y)}{\partial y} \\ \frac{\partial {N_1}^u(x,y)}{\partial y} & \frac{\partial {N_1}^u(x,y)}{\partial x} & \frac{\partial {N_2}^u(x,y)}{\partial y} & \frac{\partial {N_2}^u(x,y)}{\partial x} & ... & \frac{\partial {N_n}^u(x,y)}{\partial y} & \frac{\partial {N_n}^u(x,y)}{\partial x} \end{bmatrix}$$

The Galerkin's method is also utilized for the virtual components of the displacement and strain fields, which can be described as:

$$\delta\boldsymbol{u} = \boldsymbol{N}^u\delta\boldsymbol{c}^u \tag{2.6}$$

$$\delta\boldsymbol{\epsilon} = \boldsymbol{B}^u\delta\boldsymbol{c}^u \tag{2.7}$$

Inserting the approximation of the displacement variable and its virtual component into the weak form equation, the system of equations can be written in a matrix form as following, in order to be solved in an iterative manner:

$$\boldsymbol{K} \cdot \boldsymbol{a^u} = \boldsymbol{f_{ext}} \tag{2.8}$$

where

$$\boldsymbol{K} = \int_V \boldsymbol{B}^{u^T} \boldsymbol{D}\, \boldsymbol{B}^u \, dV \tag{2.9}$$

for the material stiffness tensor $\boldsymbol{D}$, and:

$$\boldsymbol{f_{ext}} = \int_V \boldsymbol{N}^{u^T} \bar{\boldsymbol{b}} \, dV + \int_\Gamma \boldsymbol{N}^{u^T} \bar{\boldsymbol{t}} \, d\Gamma \tag{2.10}$$

It is mentioned that the gravitational forces $\bar{\boldsymbol{b}}$ are neglected for our study.
For the 2-dimensional problem Constant Strain Triangular (CST) linear elements are utilized with one Gauss integration point, and the system of equations is solved for the unknown displacements at the degrees of freedom of the nodes.

2.2 Finite Element Problem - Explicit Dynamic

The axisymmetric disc problem as presented in the previous section is solved by considering the case of initial velocity at the same position the force is applied in the static case. The assumptions and parameters set for the material of density 0.00785 kg/mm^3 and boundary conditions are considered as for the static case. The domain is discretized using CST elements as well.
For solving the finite element problem, an explicit dynamic solver is developed. Discretizing the simulation time of 0.003 sec and a time step which varies differently with the mesh size between $\Delta t = 1 \cdot 10^{-4}$ - $1 \cdot 10^{-7}$ sec. The displacement at the degrees of freedom is computed explicitly at a time step n, as:

$$\boldsymbol{a}^n = \boldsymbol{a}^{n-1} + \Delta t \cdot \boldsymbol{v}^{n-1} + \frac{\Delta t^2 M^{-1}}{2} (\boldsymbol{f_{ext}} - \boldsymbol{f_{int}}) \tag{2.11}$$

Where f_{int} is computed as Ka^{n-1}, and a^{n-1} is the displacement of a previous time increment which is updated after each time step, and v^{n-1} , the initial velocity which is updated as:

$$v^n = 2\frac{a^n - a^{n-1}}{\Delta t} - v^{n-1} \tag{2.12}$$

2.3 Solvers

For the purposes of FEM SA there are two different and relevant solvers. These are sparse direct solvers and iterative solvers[2]. The sparse direct solvers use the Gaussian elimination method or an equivalent method like LU or Cholesky decomposition to solve the system of equation for the displacement vector. It does so in quadratic time complexity. The sparse direct solvers guarantee a converged solution which is accurate, but is limited to an overwhelmingly serial execution due to the

dependencies in the system of equations, making it an unviable target for large scale GPU-acceleration. Nevertheless, these solvers are still good in the context of a small to mid scale problem due to their accuracy. The iterative solvers on the other hand solve the system of equation utilizing that the system of equation problem can be transformed to a quadratic formula representation

$$g(a) = (1/2)a^T Ka - a^T f \tag{2.13}$$

upon which various steepest descent inspired optimization methods can be invoked to solve for the solution displacement vector.

2.3.1 Conjugate Gradient Iterative Search

One such is the conjugate gradient[3] method which based on an initial guess a_0 caluclates an initial residual vector

$$R_0 = f - Ka_0 \tag{2.14}$$

and sets it as the initial search direction:

$$P_0 = R_0 \tag{2.15}$$

For the iterative steps, the step size is calculated as

$$\alpha_k = \frac{R_K^T R_K}{P_k^T K P_k} \tag{2.16}$$

afterwhich the solution can be updated iteratively

$$a_{K+1} = a_k + \alpha_k P_k \tag{2.17}$$

as well as the residual

$$R_{k+1} = R_k - \alpha_k K P_k \tag{2.18}$$

The quick convergence compared to the steepest descent method comes from the following iterative step ensuring that the new search direction is conjugate to the previous

$$\beta_{k+1} = \frac{R_{K+1}^T R_{K+1}}{R_K^T R_K} \tag{2.19}$$

in the following search direction

$$P_{k+1} = R_{k+1} + \beta_{k+1} P_k \tag{2.20}$$

2.3.2 Conjugate Gradient Squared Iterative Search

The conjugate gradient squared method works in a similar manner to CG from an initial guess a_0. The difference is that it utilizes two auxillary vectors to steer the search direction[4]. The initial search direction and residual is the same as in CG

$$R_0 = f - Ka_0 \tag{2.21}$$

$$P_0 = R_0 \tag{2.22}$$

In the iteration step, the step size, as well as solution and residual update are also the same as in CG

$$\alpha_k = \frac{R_K^T R_K}{P_k^T K P_k} \tag{2.23}$$

$$a_{K+1} = a_k + \alpha_k P_k \tag{2.24}$$

$$R_{k+1} = R_k - \alpha_k K P_k \tag{2.25}$$

What's different is that the following two residual vectors are calculated

$$V_{k+1} = R_{k+1} - \frac{R_{k+1}^T K P_k}{P_k^T K P_k} V_k \tag{2.26}$$

$$W_{k+1} = K^T R_{k+1} - \frac{R_{k+1}^T K P_k}{P_{k+1}^T K P_k} W_k \tag{2.27}$$

including the now modified conjugacy scalar

$$\beta_k = \frac{R_{k+1}^T V_{k+1}}{V_k^T V k + 1} \tag{2.28}$$

upon which the search direction is updated as

$$P_{k+1} = V_{k+1} + \beta_k P_k - \frac{W_{k+1}\beta_k}{\alpha_k} \tag{2.29}$$

While the conjugate gradient method is well suited for when the K matrix is symmetric and positive definite, the conjugate gradient squared method is a more appropriate choice when K is non-symmetric.

2.3.3 Minimal Residual Iterative Search

Lastly we have the minimal residual iterative search method which approximates the solution $u_k \epsilon$ ($u_0 + \langle r_0, Kr_0, K^2 r_0, \ldots, A^{k-1} r_0 \rangle$), by minimizing the norm of the residual $r_k = f - Ku_k$ in the subspace to the right of the epsilon-sign[5]. This is in fact the general approach for General Minimal Residual methods, but since the vectors in the subspace might be linearly dependent, introducing numerical instability in the solving procedure for large and sparse matrixes, an orthonormal basis $q0, q1, ..., qk$ found via the Arnoldi method is instead used which enables the solution vector $u_k \epsilon$ ($u_0 + \langle r_0, Kr_0, K^2 r_0, \ldots, A^{k-1} r_0 \rangle$) to be obtained by $u_k = u_0 +$

$Q_k y_k$, where y is a real valued vector and Q is the matrix formed by the basis vectors. The problem is now solved by finding y_k for which the system $KQ_k = Q_{k+1}H_k$ helps in doing so since the Hessenberg matrix H is also obtained by the Arnoldi method. In the symmetric case, a symmetric triagonal matrix is in fact achieved, from which the MINRES method can be invoked to minimize the residual which now looks like the following $\|H_n y_n - \beta e_1\|$. This is the equivalent of solving the least squares problem which can be done by QR decomposition or an equivalent method such as gaussian elimination. The GMRES method is good when the matrix K is non-symmetric and indefinite, while the MINRES method is good for solving large and sparse systems that are symmetric but still indefinite.

2.4 GPU-acceleration in Python

The appeal of General Purpose GPU is based on the fact that GPUs use a much larger fraction of the silicon for computation than CPUs. Because of this they can use far less energy per unit of computation and offer a much larger degree of parallelism than CPUs. Therefore programs with a high level of concurrency and simple control logic are ideal for execution on a GPU. Thanks to the large number of threads that are spawned on the GPU, it is able to hide latencies that arise from for example cache misses by switching to another thread. Switching threads on the GPU is supported by hardware and is therefore very fast.
The computational model that GPUs are based on is called Single Instruction, Multiple Threads (SIMT). Following this model, General Purpose GPUs are programmed using CUDA by writing a single program that is executed by all threads. The number of threads is specified when launching the program on the GPU. Before launching the program, the required data needs to be transferred to the GPUs memory from the main memory. After computation the result has to be transferred back to main memory.
Python is an interpreted language, and is therefore generally thought not to be suitable for high-performance computing. There are however libraries that offer just-in-time (JIT) compilation of your code, and also pre-compiled functions. Numba is Python library which offers JIT compilation of Python code. Among other things it has support for CUDA, and can compile Python functions into CUDA, for acceleration on GPUs.
Cupy is another alternative for GPU-acceleration in Python. Cupy offers pre-compiled functions with an interface that is highly compatible with Numpy and Scipy. Thanks to this, it can often be used through a simple drop-in replacement with Numpy or Scipy.

3 Method

In the following section, the process of accelerating the FEM method on GPUs is presented.

3.1 GPU-accelerated static FEM

The profiling of the static FEM code identified two main kernels contributing the majority of the execution time. These were found to be the stifness assembly and solver. In order to focus the development efforts to be as effective as possible, only these two computational kernels were accelerated with the GPU.

3.1.1 Stiffness assembly

The stiffness assembly function was accelerated on the GPU by parallelizing the loop over all elements over multiple GPU threads. This was done using Numba for CUDA.
In order to use the matrix operations that need to be performed inside of the loop iterations it was necessary to implement these from scratch for CUDA. The following matrix operations were implemented: matrix multiplication, matrix-vector multiplication, inversion of two-by-two matrix, transposition of two-by-two matrix and transposition and scaling of three-by-six matrix. The dimensions of the matrices in all operations are known beforehand and therefore several of the functions were hardcoded to those specific dimensions. Because the loop is what is parallel on the GPU, the implementations of the operations that are performed inside the loop iterations are serial.
The reason that this approach was chosen, even though it led to a lot of extra implementation work, over simply swapping the NumPy function calls inside the loop for the corresponding functions from CuPy, is the following. The number of loop iterations grows proportionally to the problem size, because it iterates over all elements. The matrix operations inside of the loop however, operate on small, fixed-size arrays. Therefore it is more effective to parallelize the loop instead of parallelizing the operations inside of the loop with CuPy.
Because the K matrix, which is the result of the stiffness assembly, will be very large for big problems, it would be infeasible to store the entire matrix on the GPU. Therefore it is represented on the GPU as a value-table where the threads store the produced values as pairs of matrix coordinates and values. Each value is stored in a new entry to avoid having to use synchronization to handle race conditions. The sparse matrix constructors in SciPy are however able to handle this so it is not a large issue, but it leads to slightly increased memory consumption on the GPU.

3.1.2 Solvers

The solvers are SciPy functions, and could therefore easily be accelerated using CuPy. It was done by simply swapping the SciPy function with the corresponding CuPy function. Additionally, the input matrices had to be converted to CuPy types,

i.e. transferred to GPU. The same goes for the resulting matrix, in the opposite direction.

3.2 Explicit Dynamics Parallelization

Since the explicit is computed every time step of every degree of freedom it means that the whole system's displacement is evaluated at once and the next time step is dependent on the previous iteration, which means that the time-stepping loop cannot be parallelized. Instead, the explicit dynamics formula 2.11 could be rewritten to be more friendly for parallelization by evaluating the elementwise contribution of the displacement as shown in the following formula:

$$a^n = a^{n-1} + v^{n-1}\Delta t + A_{e=1}^{nel}\left\{\underbrace{\frac{\Delta t^2}{2}E_{e}{}_{e=1}^{nel}(M^{-1})(f_{\text{ext},e} - K_e E_{e}{}_{e=1}^{nel}(a^{n-1}))}_{\Delta a_e}\right\} \tag{3.1}$$

Where the symbol $A_{e=1}^{nel}$ indicates that the value is assembled, and the symbol $E_{e}{}_{e=1}^{nel}$ indicates that the value is extracted, meaning extracting the element contribution from a global matrix.

3.2.1 GPU-accelerated Explicit Dynamics

In a similar fashion to the static code, the efforts for the GPU-accelerated dynamics were concentrated on the part of the code where most of the execution time was spent. The time-stepping loop could not be parallelized, but the elementwise loop within could be parallelized in the same fashion, i.e. using Numba, as the stiffness assembly of the static GPU-accelerated code. Here too, the majority of the code is run on the CPU, except for the elementwise loop, for which the required data is transferred to the GPU and the loop is run in parallel. For this again, it was needed to develop several array operations from scratch and the same value table sparse matrix representation, as in the static stiffness assembly, is used.
Once the results of the elementwise loop have been produced and inserted into the value table, the value table has to be transferred back to the CPU as the rest of the time-stepping loop is performed on the CPU. This is a downside with this implementation, as it incurs some overhead, for transferring data back and forth between the CPU and GPU, each loop iteration. A possible improvement to this would be to implement also the rest of the time loop for the GPU as to avoid the data transfers.

4 Results & Discussion

In this section the results of the evaluation of the developed programs, along with discussions of the results, are presented.

4.1 Static Problem

As mentioned before, there are two main parts of the static problem that each take a significant amount of execution time and are prime for GPU acceleration. The first and the most time-consuming part of the code in this implementation is the assembly of the stiffness matrix and the second is the solving of the displacement matrix. Note that the assembly could likely be optimized further and be a smaller factor in other implementations but the solvers are highly optimized. Both of these parts were profiled and successfully accelerated with a GPU implementation.

The stiffness assembly execution times for the CPU and GPU implementations are shown in Figure 4.1.

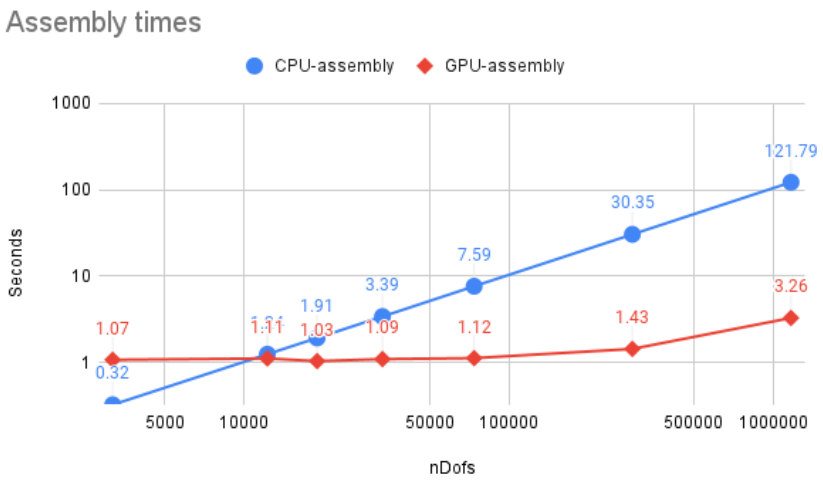


Figure 4.1: Execution time in seconds of stiffness assembly for both CPU and GPU.

Figure 4.1 illustrates the relationship between the number of degrees of freedom and the computation time for both GPU and CPU. It's crucial to note that the stiffness matrix has dimensions $(nDofs \times nDofs)$, indicating that, for the largest case tested, the matrix size corresponds to $10^6 \times 10^6$ cells. Also, Figure 4.2 shows the speedup due to GPU implementation for different mesh sizes.

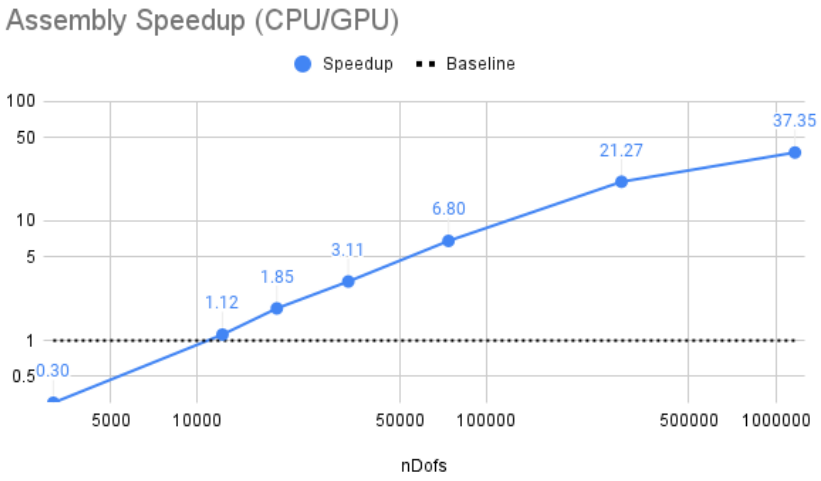
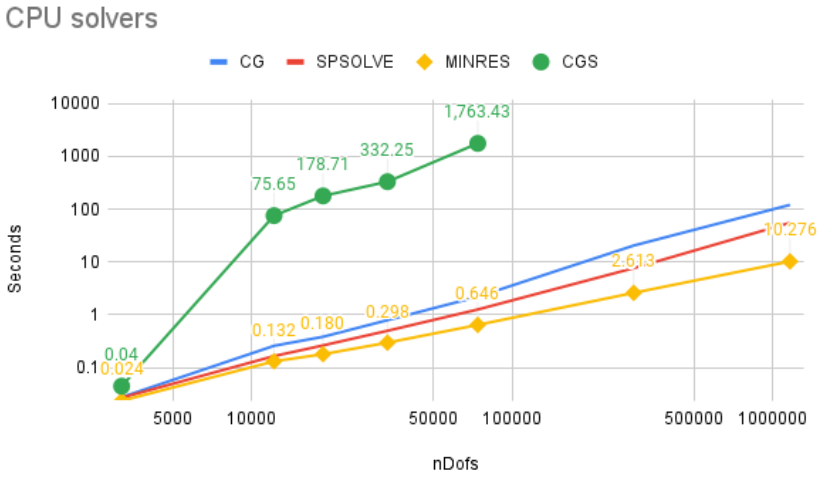
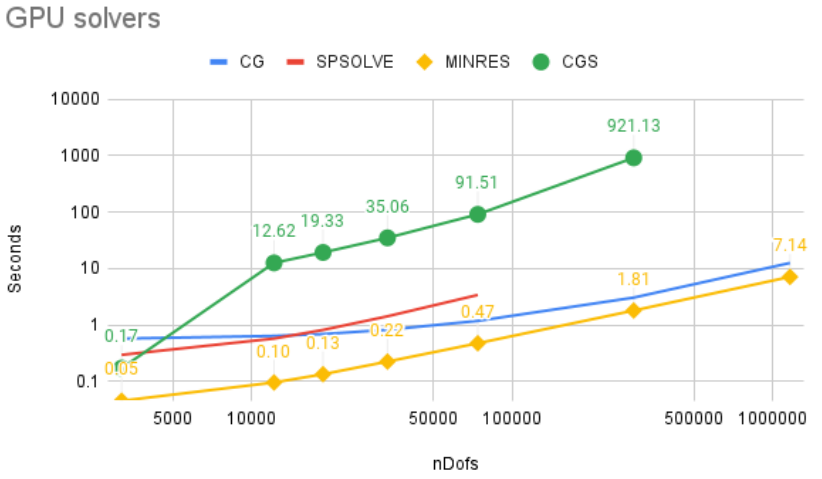


Figure 4.2: The speedup due to GPU implementation for different mesh sizes. Note the dotted line, points under it mean the the GPU implementation was slower than the CPU and vice versa.

As for the solving of the displacement matrix, four different types of solvers were benchmarked, both on the CPU and GPU, with one of them being a direct solver (SPSOLVE) and the other three being iterative solvers (CG, CGS and MINRES). On the CPU, solvers from the SciPy library were used and for the GPU implementation CuPy's implementation of the same solvers were instead used.



(a) CPU solvers



(b) GPU solvers

Figure 4.3: Execution times for the different solvers both on the CPU and GPU

Figure 4.3 illustrates that MINRES performed the best in both implementations, while CGS performed the poorest, which is supported by the fact that the problem at hand includes the K matrix of symmetric characteristic, which was well suited for MINRES, whereas CGS was more suited to a non-symmetric matrix. Something particularly noteworthy is MINRES's consistent performance, only obtaining a 1.4x acceleration when transitioned on the GPU. This could be explained by the tolerance levels being unchanged and therefore some slight data-loss due to early stopping but it did not seem to affect the final deformation significantly.

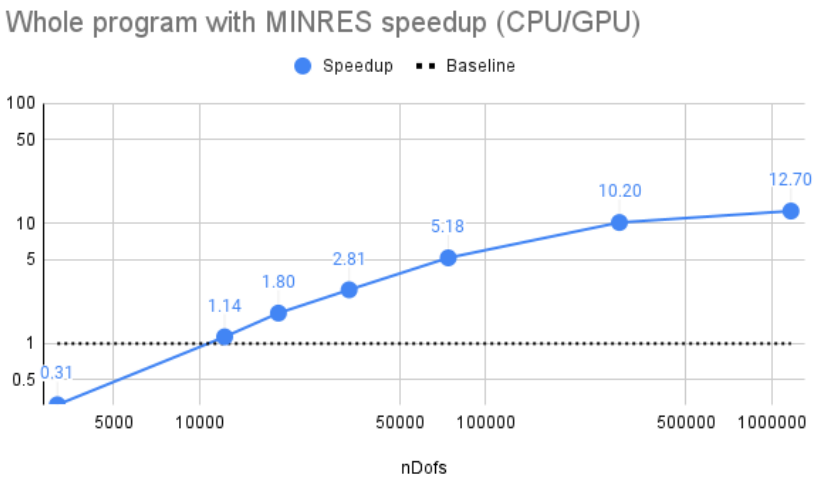


Figure 4.4: Speedup of whole program using MINRES as a solver.

As for the performance boost of the program on the whole, we used MINRES as the solver for both implementations to accurately compare them. Similarly to the assembly speedup, we see that the GPU implementation was quite slower on lower nDofs (degrees of freedom), which was expected because of the overhead of moving data to the GPU and back. However, with larger nDofs, the GPU performed quite a bit better.

4.2 Dynamic Problem

In the dynamic problem, we disregard the idea of having to solve the displacement at the end and instead use a single assembly of the masses and then an iterative assembly of the displacement. As such, almost all of the computation time is spent in these assemblies, which can be sped up with a GPU implementation.

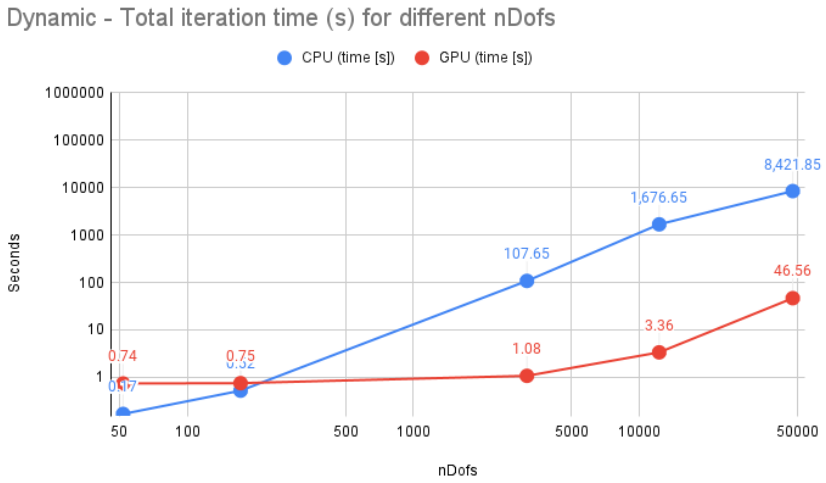


Figure 4.5: Execution time in seconds of total iteration time for displacement assembly for both CPU and GPU.

In Figure 4.5, there is a clear distinction in the performance of the CPU and GPU implementation. This difference is logical, given that parallelization benefited the assembly process the most in the static problem. The speedup for the assembly process (Figure 4.6) follows a slight S-curve which dips at the end. It is unclear why this dip appeared but it could be explained by memory limitations and cost of transferring memory to and from the GPU. Despite this, a speedup of around 500x was achieved in the case of 10^3 nDofs, which is $10^3 * 10^3 = 10^6$ cells.

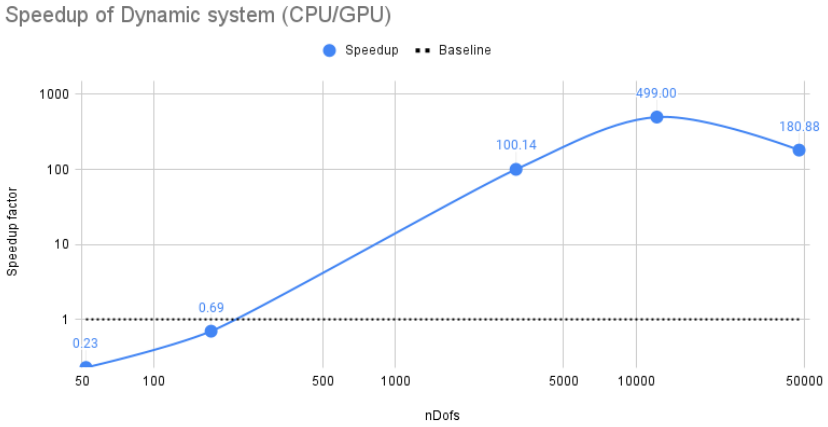


Figure 4.6: The speedup due to GPU implementation for different mesh sizes. Note the dotted line, points under it mean the the GPU implementation was slower than the CPU and vice versa.

As can be seen, the GPU implementation generally performed worse than the CPU at smaller *number of degrees of freedom* (nDoFs) and better than the CPU at larger nDofs, both in the Static and Dynamic problem. This performance difference matches the intuition that the GPU implementation is more efficient but has a larger over-

head.

However, this performance boost came at the cost of implementation time and complexity, since all matrix operations had to be implemented from scratch to work with CUDA on the GPU. Additionally, the sparse matrix data type had to be implemented from scratch as well to reduce memory consumption and to allow the GPU to write to it concurrently without race conditions. This implementation ended up being more efficient on the CPU as well, slashing computing time in half.

5 Conclusions

The optimization of the GPU implementation could be generalized further to problems of a similar nature, problems with large sparse matrices that need to be assembled from many small matrices, which unfortunately aren't many outside of FEM. Additionally, the problem size is constrained by the memory available on the GPU, although using a more efficient sparse matrix implementation helps. Overall, it could be difficult to adapt the project to other problems without some major rewrites and changes. That said, this project lends itself to be extended to non-linear FEM problems, especially the dynamic implementation, which could be used in more advanced simulations.

As seen in the report above, all goals mention in Section 1.2 were achieved. Ultimately, speedups of 12.7 for the static problem and 499.0 for the dynamic problem, were achieved. This project underlines the importance of GPU acceleration in the realm of Computer Aided Engineering (CAE). The performance gains, especially from the assembly, emphasize the critical role of parallelization in reducing simulation times and allowing for larger, more accurate simulations. This is especially significant for complex scenarios, like crash simulations with numerous degrees of freedom and intricate contact dynamics, where the possibility for an accurate simulation makes vehicles safer for both drivers and pedestrians.

6 Contributions

During the project, the team has been divided into two teams. One team was responsible for the implementations of the static and dynamic FEM codes and the other team was responsible for accelerating those codes in GPU. Performed activities for each group member follows below:

Afroditi: Initially, me and Oweis were working on the static FEM code. My contribution to that was mainly in the beginning, in determining and formulating the problem to be solved (geometry, boundary conditions, applied load etc). Oweis was handling the code and I offered my assistance in developing some of the functions, such as the function that created the external force vector. For the dynamic problem that followed after the static one, Oweis was the main responsible for the development of the code. My contribution in the dynamic code was towards the end of the process, in offering assistance in troubleshooting and debugging the code that Oweis developed. Regarding the project report, I suggested and formulated the basic structure of it, and wrote Chapters 1 and 2.1. For the Power-Point-Presentation, I suggested the initial format during mid-term presentation, wrote and presented the introductory slides as well as the ones related to the FEM problem formulation and FEM theory.

Oweis: Afroditi and I teamed up to define the FEM problem and kickstarted the code by laying out the first steps. Specifically, we tackled the construction of the external force function. As things progressed, I took the lead on building the rest of the code. I was also pretty hands-on when it came to implementing any tweaks or suggestions from the GPU team to boost CPU efficiency. On top of that, I ran my own benchmarks for CPU performance, making sure our results matched up with what we got on the university computers. Afroditi and then I completed the dynamics code through to the end. Plus, I contributed to both the mid-term and final presentations, not to mention putting in work on the final report in the theory, methods, and results chapters.

Róbert: Over the course of the project, the GPU team has been working closely together to develop the GPU implementations and so most of the code is attributed to the whole team. That being said, in addition to taking part in the coding sessions above, I also worked on implementing our own version of sparse matrices, which ended up providing a significant performance boost and were the key to work with the large matrices on the GPU. I also performed most of the benchmarking on both CPU and GPU implementations and created all of the graphs used. In the presentations, I wrote and presented our results and comparisons for the static problem. Similarly, I wrote most of the "Results" and "Conclusions" chapters in the final report. Overall, I've been so happy to be able to take part in such a fun project and work with such a talented team.

Simon: During the project I have, together with Róbert and Arik, developed the GPU accelerated versions of the FEM method programs. Between sessions I also

did some coding and a lot of debugging on my own. In the final report, I wrote the method sections describing how the GPU acceleration was done, as well as the theory section on GPU-acceleration in Python. Similarly, in the presentation I made the slides about how we accelerated the programs on GPU. I also presented it for the static version in both the midway and final presentations.

Arik: Over the course of the project I have contributed by helping the team develop the GPU accelerated versions of the FEM SA programs. In particular i helped develop and debug some of the static GPU version of the code, and a larger part in the explicit dynamic GPU version. I believe I helped myself and the team understand the ins and outs of GPUs and why we at times had errors in our implementations and how we could go about to solve them. In addition, i took notes of essential parts during weekly meetings and distributed these within the group. I helped derive most of the theoretical work on the theory on solvers for the presentation and report as an underlying fundamental to help the team draw conclusions as to why our results came out as they did with that as a support. I helped in structuring the slides for the midway and final presentation, as well as taking part in them.

References

[1] Martin Fagerström and Magnus Ekh. *Finite Element Method – Structures Lecture notes*. March 6, 2023.

[2] Hiroshi Okuda Serban Georgescu, Peter Chow. Gpu acceleration for fem-based structural analysis. 20:114–115, 2013.

[3] Wikipedia. Conjugate gradient method, 2024. Accessed: January 10, 2024.

[4] A. Author. *An introduction to continuous optimization : foundations and fundamental algorithms*. Niclas Andréasson, Anton Evgrafov, Michael Patriksson, 2016.

[5] Wikipedia. Generalized minimal residual method, 2024. Accessed: January 25, 2024.