Chalmers University of Technology

GPU Accelerated Computational Methods Using Python and CUDA

GPU Accelerated FEM Calculations for Stationary Heat Flow

Author:
Carl Gillmert
Sebastian Kvaldén
Erik Henriksson
Yanchen Lin

Supervisor:
Prof. Fredrik Larsson

February 2025

Abstract

The Finite Element Method (FEM) is a widely used computational tool in the mechanical engineering field for computing stresses on objects and heat transfers. However, it comes with high computational demands, especially in large-scale simulations. This project evaluates the efficiency of FEM calculations for stationary heat flow across three platforms: a CPU, a local GPU, and a GPU cluster such as the Vera computing cluster at Chalmers University of Technology. Benchmark tests were used to analyze the performance of assemblers and solvers, focusing on speed with memory constraints in mind. The results show significant improvements in speed with GPU acceleration, where the fastest GPU solver/assembly combination finished its calculations well above three orders of magnitude faster than the slowest combination, and just below one order of magnitude faster than the best CPU-only version.

The memory limitations of standalone GPUs caused some challenges, which were solved with specialized algorithms together with extended RAM available on the Vera cluster. The findings highlight the potential of GPU acceleration for FEM and its use in larger computational problems. Ideas for future improvements, using multiple GPUs and continued optimization of assemblers, using different elements for the mesh, and importing already-made mesh nets, are also presented.

Contents

1 Introduction

A common issue in mechanical engineering is computing heat flow in objects. One common method to solve this is with the so called the Finite Element Method (FEM). FEM breaks down the object into several smaller elements which together are called a mesh. The mesh represents the object with a finite number of elements. While FEM is a great method for calculating heat flow in objects, it also requires a great amount of computational power. This becomes a significant challenge in large-scale problems.

Historically, FEM calculations were performed on Central Processing Units (CPUs). However, since the mid-2010s, Graphics Processing Units (GPUs) have been the preferred Processing Unit due to their significant performance advantages. GPUs can offer enormous speed increases for both the assembly and solving processing thus enabling faster and more efficient computations, especially for large-scale and parallelizable tasks [1], [2].

1.1 Aim of the Project

This project will evaluate and compare the performance of FEM calculations for stationary heat flow. The project will compare calculations of calculations on a CPU, GPU, and the computer cluster Vera at Chalmers University of Technology [3] with memory constraints in mind. The goal is to find which solvers and assemblers perform better in a 2D environment.

1.2 Limitations

Several factors limit this project. First, it focuses only on 2D FEM models, and not 3D simulations due to the limited time frame. Second, the scope of the project was also limited to only focus on the linear stationary model problem of heat flow. Third, GPU tests are conducted using quadrilateral mesh elements, chosen for their scalability, with limited exploration of triangular elements. Finally, memory constraints on standalone GPUs restrict the scale of certain simulations, although these challenges are alleviated in cluster-based tests.

2 Theoretical Background

In this section the basis of the heat flow problem which is the problem to be solved using the FEM method will be covered. Together with a simple introduction to GPU accelerated computation.

2.1 Stationary Heat Flow

Consider a two-dimensional surface $\Omega \in \mathbb{R}^2$ with constant heat conductivity k being heated by a source $f : \Omega \to \mathbb{R}$. Let $\partial\Omega$ denote the boundary of Ω and let Γ_N, Γ_D be a partition of $\partial\Omega$, such that $\Gamma_N \cup \Gamma_D = \partial\Omega$, $\Gamma_N \cap \Gamma_D = \emptyset$. Along Γ_N the temperature is kept constant according to $T : \mathbb{R}^2 \to \mathbb{R}$ while along Γ_D the heat flux is zero which means that Ω is isolated along Γ_D. In the stationary heat problem we are interested in solving for the temperature distribution $u : \Omega \to \mathbb{R}$ when the system has reached equilibrium and is therefore time-independent. The problem can be written as a partial differential equation according to,

$$\nabla \cdot (-k\nabla u(x,y)) = f(x,y), \qquad (x,y) \in \Omega, \tag{1}$$

$$u(x,y) = T(x,y), \qquad (x,y) \in \Gamma_D, \tag{2}$$

$$\nabla_{\boldsymbol{v}} u(x,y) = 0, \qquad (x,y) \in \Gamma_N. \tag{3}$$

which is known as the strong form. Here, ∇ denotes the gradient and $\nabla_{\boldsymbol{v}}$ the directional derivative. To numerically solve (1)-(3) the system needs to first equivalently be rewritten on its weak form. We introduce a so called test function $\delta u(x,y)$, that satisfies $\delta u(x,y) = 0$ on Γ_D. By multiplying (1) with δu and integrate over the surface Ω we get,

$$\int_\Omega \delta u(\nabla \cdot (-k\nabla u))\ d\Omega = \int_\Omega \delta u f\ d\Omega. \tag{4}$$

By Greens first identity, the left hand side of (4) can be rewritten as

$$\int_\Omega \delta u(\nabla \cdot (-k\nabla u))\ d\Omega = \int_{\partial\Omega} \delta u(-k\nabla_{\boldsymbol{n}} u) d\Gamma + \int_\Omega \nabla\delta u \cdot k\nabla u\ d\Omega, \tag{5}$$

where $\Gamma = \Gamma_D \cup \Gamma_N$. For the boundary integral in the right hand side of (5) we have

$$\int_{\partial\Omega} \delta u(-k\nabla_{\boldsymbol{n}} u)\ d\Gamma = \int_{\Gamma_D} \delta u(-k\nabla_{\boldsymbol{n}} u)\ d\Gamma + \int_{\Gamma_N} \delta u(-k\nabla_{\boldsymbol{n}} u)\ d\Gamma = 0 + 0, \tag{6}$$

since $\delta u = 0$ on Γ_D and $\nabla_{\boldsymbol{n}} u = 0$ on Γ_N. At last, by substituting (6) into (5) we arrive at the weak form

$$\int_\Omega \nabla\delta u \cdot k\nabla u\ d\Omega = \int_\Omega \delta u f\ d\Omega. \tag{7}$$

2.2 Discretization

To solve (7) numerically, we introduce a discretization of the domain Ω. Let Ω be divided into N_e elements $\Omega^1, \Omega^2, ..., \Omega^{N_e}$ such that $\Omega^1 \cup \Omega^2 \cup ... \cup \Omega^{N_e} = \Omega$. Each element is corresponds to a set of nodes which are the edges of the elements. The discretization is also known as a mesh. For simplification, we will consider quadrilateral meshes where each element consist of four nodes. The theory can simply be extended to triangular meshes as implemented in section 4.5. Let N_{dof} be the number of nodes in the mesh and therefore the number of degrees of freedom (Ndofs) in the system. The solution can then be approximated according to

$$u(x,y) \approx u_h(x,y) = \sum_{i=1}^{N_{dof}} N_i(x,y)a_i, \tag{8}$$

where a_i are node values and N_i are basis function satisfying

$$\delta u(x,y) = \sum_{i=1}^{N_{dof}} N_i(x,y)c_i, \tag{9}$$

with c_i being arbitrary coefficients. The basis function are defined such as they take the value one on its corresponding node and zero in every other node. Substituting (8) and (9) into the weak form (7) gives us the approximation

$$\int_\Omega \nabla \left(\sum_{i=1}^{N_{dof}} N_i(x,y)c_i \right) \cdot k\nabla \left(\sum_{j=1}^{N_{dof}} N_j(x,y)a_j \right) \, d\Omega = \int_\Omega \left(\sum_{i=1}^{N_{dof}} N_i(x,y)c_i \right) f \, d\Omega. \tag{10}$$

Since c_i and a_i are independent of x and y (10) can be rewritten as

$$\sum_{i=1}^{N_{dof}} \sum_{j=1}^{N_{dof}} c_i \int_\Omega \nabla N_i \cdot k\nabla N_j \, d\Omega \, a_i = \sum_{i=1}^{N_{dof}} c_i \int_\Omega N_i f \, d\Omega. \tag{11}$$

We identify (11) as a linear system of equations

$$c^T Sa = c^T f \Leftrightarrow Sa = f, \tag{12}$$

where

$$S_{ij} = \int_\Omega \nabla N_i \cdot k\nabla N_j \, d\Omega, \tag{13}$$

$$f_i = \int_\Omega N_i f \, d\Omega. \tag{14}$$

The matrix $S \in \mathbb{R}^{N_{dof} \times N_{dof}}$ is called the stiffness matrix and the vector $f \in \mathbb{R}^{N_{dof}}$ is called the load vector. Notice that if N_i and N_j corresponds to basis functions of non neighboring nodes the product of their gradients becomes zero and thus also S_{ij}. This means that the stiffness matrix will be sparse which is an important property to keep in mind. Two other important properties is that S is symmetric and is known to be semi-definite.

2.3 Assembly Process

The discretization of Ω can be represented with a so called adjacency matrix $M \in \mathbb{N}^{N^e \times 4}$. Each row of M corresponds to an element and stores the set of nodes the element corresponds to. For each element e, we can calculate its contributions to the stiffness matrix and load vector independently. We denote these local matrices and vectors by S^e and f^e, according to

$$S^e_{ij} = \int_{\Omega^e} \nabla N^e_i \cdot k\nabla N^e_j \, d\Omega^e, \tag{15}$$

$$f^e_i = \int_{\Omega^e} N^e_i \, d\Omega^e. \tag{16}$$

Since quadrilateral elements are used, $S^e \in \mathbb{N}^{4\times4}$, and $f^e \in \mathbb{N}^4$. The stiffness matrix (13) and load vectors (14) can then be calculated according to

$$S_{ij} = \sum_{e=1}^{N_{dof}} S^e_{ij}, \tag{17}$$

$$f_i = \sum_{e=1}^{N_{dof}} f^e_i. \tag{18}$$

To simplify calculations further, each element can Ω_e can be transformed to the square $[0,1]^2$ by a change of variables. This introduces the calculation of the Jacobian determinant of the transformation. At last, the integrals (15) and (16) can for example be approximated with Gaussian quadrature with some finite number of weights. A common choice of basis functions are bilinear. Bilinear basis functions are one at their

corresponding node, decreases linearly to zero in the direction of its two neighboring nodes and are 0 in the opposite node. The gradient of the basis functions will therefore be constant.

Once the stiffness matrix and load vector is assembled the boundary conditions can be imposed. Since the temperature along Γ_D are known, the corresponding rows and columns that corresponds to the nodes can be removed from the stiffness matrix and load vector. This reduces the NDofs in the system but also makes the stiffness matrix positive definite. Moreover, to impose some predetermined heat flux on Γ_N. The heat flux can simply be added to the load vector and since its 0 along Γ_N for this specific problem, no changes needs to be made. At last, (12) can be solved with an appropriate solver.

2.4 Solving Systems of Linear Equations

There are several different approaches for solving the linear system (12). The algorithms for solving linear systems are either direct or iterative methods. Direct methods solves systems analytically by for example Gaussian elimination or matrix decomposition. Iterative methods solve systems by given an initial guess, generates a sequence of approximated solutions that converges to the analytic solution. This means that they have some stopping criteria to determine when the solution has converged to stop iterating. The stopping criteria may vary for different methods which may make comparing them unfair. Generally, direct solvers needs more memory than iterative solvers which means iterative solvers could be more suitable for systems with greater amount of NDofs. Moreover, iterative algorithms relies heavily on matrix multiplications which may make them more favorable when being run on the GPU. A number of pre-written solvers used in the benchmark is presented in 4.4.

2.4.1 Conjugate Gradient Method

To illustrate how iterative algorithms works, we will describe the conjugate gradient method in detail. Other iterative methods later used in the benchmark will not be described as thoroughly but share similarities with the method. The method assumes that the stiffness matrix is symmetric and positive definite to assume convergence [4]. If the stiffness matrix is symmetric, then solving (12) is equivalent to solving the minimization problem

$$\min \frac{1}{2} a^T S a - f^T a, \tag{19}$$

since taking the derivative of (19) gives us that the solution is given by solving

$$Su - f = 0, \tag{20}$$

since we know S is positive definite. The algorithm is similar to gradient decedent method but instead of moving in the direction of the negative gradient in each iteration we move in the so called conjugate direction which improves convergence. Let a_i denote the solution in iteration i. The initial residual r_0 is given by

$$r_0 = f - S a_0, \tag{21}$$

given an initial solution a_0. Moreover, the initial search direction p_0 is simply set as r_0. Then, in each iteration i, the step size α_i is calculated and the solution updated according to

$$\alpha_i = \frac{r_i^T r_i}{p_i^T S p_i}, \tag{22}$$

$$a_{i+1} = a_i + \alpha_i p_i. \tag{23}$$

The residual r_i can then be updated according to

$$r_{i+1} = r_i - \alpha_i S p_i. \tag{24}$$

If $||r_i||_2$ is sufficiently small, the algorithm terminates. At last the step direction p_i is updated according to

$$p_{i+1} = r_{i+1} + \beta_i p_i, \tag{25}$$

where

$$\beta_i = \frac{r_{i+1}^T r_{i+1}}{r_i^T r_i}. \tag{26}$$

It is the calculation of β_i that ensures that the new search direction p_i is so called S-conjugate to all previous directions. Note that the method performs a lot of matrix and vector multiplications in each iteration which has the potential of being speed up when being run on the GPU. Moreover, the amount of data being stored is independent of how many iterations the algorithm performs.

2.5 GPU Acceleration & CUDA

While there are many similarities between CPUs and GPUs there are some differences that makes GPUs extremely effective for some computationally intensive tasks. More specifically tasks that perform similar operations to a wide range of values/variables that acts semi-independent and can thus be computed in parallel. Such as matrix computations.

This is possible because GPUs consist of a higher number of *Arithmetic Logic Units (ALUs)*, which are the fundamental computing units in any processing unit, compared to CPUs, which allocate more space to *Control Units (CUs)* and other components to handle the broader set of tasks that CPUs are designed to perform. The GPU on the other hand were specifically designed to only perform computations and can thus allocate (almost) all its available silicon for that specific task. It is not uncommon for a GPU to have above a thousand ALUs.

In essence, this allows for the programmer to write well structured routines that can run in parallel on the ALU units of the GPU to take full advantage of this processing power. To enhance the usability of GPUs for a broader range of tasks, some manufacturers have developed APIs that can be easily integrated into high-level programming languages. An example of this is the CUDA API which is available on all modern NVIDIA GPUs which give the programmer access to the so called *CUDA cores* which each consists of a small number of ALUs toghether with a shared memory.

2.6 The Vera Cluster

The Vera Cluster is a PC-cluster owned and managed by C3SE, which is the Chalmers e-Commons e-Infrastructure group at Chalmers University of Technology in Gothenburg Sweden, and it serves researchers at Chalmers.

The Vera cluster contains several hardware models. It runs Intel Xeon Gold 6130 (code-named "Skylake") CPU's and newer Intel(R) Xeon(R) Gold 6338 CPU and Platinum 8358 (code-named "Icelake") CPUs. All nodes have dual CPU sockets. It has T4, A40, V100, A100 NVidia GPUs and Infiniband network.

The Vera Cluster also provides commonly used software like programming tools, simulation environments and various python packages. We can access the hardware and software using command line tools and web interface. The personal files on Vera can be accessed on the website with GUI or via command line tools like **scp**.

For this project, we mainly want to accelerate the calculation with the help of Vera's GPU resources using CUDA.

3 Problem Description

To be able to solve the stationary heat flow problem described by equations (1)-(3) the problem needs to be further specified. For the results presented in 5 we have considered the square $\Omega = [-1, 1]^2$ with heat conductivity $k = 0.1$. Moreover, Γ_N consists of two of the opposite sides of the square, $\Gamma_N = \{(x, y),\ y \in [-1, 1], x \in \{-1, 1\}\}$ and Γ_D is the other two opposite sides of the square, $\Gamma_D = \{(x, y),\ x \in [-1, 1], y \in \{-1, 1\}\}$. We let the two partitions of Γ_D have different constant temperatures according to

$$T(x, y) = \begin{cases} -1000, & (x, y) \in \{(x, 1), x \in [-1, 1]\}, \\ 0, & (x, y) \in \{(x, -1), x \in [-1, 1]\}. \end{cases} \tag{27}$$

The problem with its boundary conditions is shown in figure 1.

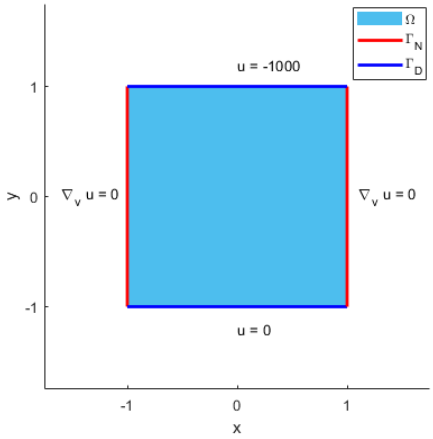


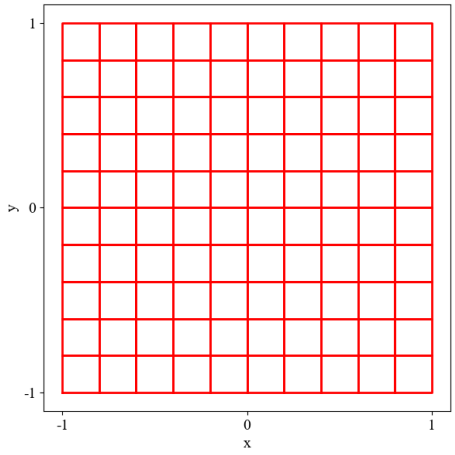
Figure 1: Illustration of the set and the boundary conditions.

At last, the heat source is given by,

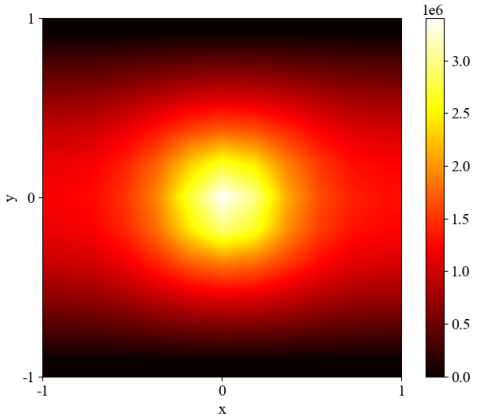
$$f(x, y) = \frac{1000}{0.0001 + 10x^6 + 10y^6}. \tag{28}$$

Notice that the heat source has it's maximum in the center of the square with value 10^7 and decays rapidly to approximately 50 on $\partial\Omega$.

An initial code was given that solves the problem using a structured quadrilateral mesh which means that all elements has the same size and dimensions. The integrals (15) and (16) were calculated using Gaussian quadrature with two points and bilinear basis functions as described in section 2.3. The system (12) was solved using `numpy.linalg.solve`. An example mesh with its corresponding solution is shown in figure 2. Notice that due to the relatively high maximum of f, the relative difference between the two constant boundary temperatures described by (27) isn't seen in figure 2b.



(a) Structured mesh with quadrilateral elements.



(b) Approximated solution of the temperature distribution.

Figure 2: A solution using 100 quadrilateral elements and 99 NDofs.

4 Method

The following section will give an outline of the project structure. In particular the design of the test environment and its most prominent features. This will be followed by information about the assemblers and solvers that will be tested in section 4.3 and 4.4.

4.1 Benchmark Environment

In order to perform reasonable tests on the variety of assemblers and solvers for the *FEM*-system a test environment were set up that allows for any combination of assembler and solver to be tested. The timing of the individual assembler and solver were then saved together with the profiling of the methods in order to analyze potential bottlenecks in the functions. The environment was split into two parts, the first one was for pre-compiling the assemblers and solvers that were going to be tested. And the second one is to time their performance with different mesh sizes as input.

The benchmark environment has the following input flags to specify all necessary input variables for the tests, together with some extra options such as plotting results and importing mesh files.

Flag	Type	Required	Description
`-s`, `-solver`	String	Yes	FEM solver to run, e.g., `GPU_cg`
`-a`, `-assembler`	String	Yes	Assembler method to run, e.g., `basic`
`-l`, `-linux`	Boolean	No	Specifies if the script is running on a Linux machine
`-i`, `-iterations`	Integer	Yes	Number of test iterations
`-s`, `-scaleFactor`	Integer	Yes	Scale factor between iterations
`-d`, `-data`	String	Yes	Array of dimension data, e.g., `[10,10]`
`-f`, `-figure`	Boolean	No	Specifies whether a figure should be plotted
`-o`, `-onlyAssemble`	Boolean	No	Runs only the assembler without the solver
`-e`, `-externalFigure`	Boolean	No	Imports a mesh file instead of creating a new one

Table 1: Input flags

4.1.1 Pre-Compiling

To ensure accurate results, it is essential to pre-compile the assemblers and solvers. This is because *Numba* employs *Just-In-Time (JIT)* compilation, where functions are compiled at their first invocation [5]. Without pre-compilation, the compilation time would be included in the test timing results, leading to skewed measurements for the first iteration of calculations. To achieve this, the test environment runs the assembler and solver that are about to be tested with a very small input, eg a 2x2 element matrix, and ignores the timing results. This ensures that all methods have been invoked before the first iteration of measurements is recorded.

4.1.2 Timing Measurements

The timing measurements for the assembler and solvers performance are as previously mentioned measured separately to enable deeper analysis of their performance. Except for the actual assemble and solver time each individual measurement also includes the transfer time to the desired computational block. For instance, if we use the GPU to assemble the S-matrix and f-vector, the transfer time needed to place the required data into the GPU memory as well as the time to transfer the S, and f variables back into main memory will be included. The reasoning for this is that if the GPU shall work as a viable option to the CPU the total computational time from initiation until the results can be used by other methods or processes must be accounted for in order to achieve a total time reduction of the system when compared to the CPU-only version. In other words, it is not sufficient for the GPU assembly to be faster on its own if the results are delayed for subsequent processes.

4.2 Profiling

To identify what parts of the original code occupy most of the computing time, and thus being able to benefit the most from parallel execution via GPU the program was profiled with the *Line-Profiler* package. This enabled profiling of the entire code for each new version of the assembler and solver that were tested to identify the system's current bottleneck.

In the original code, the primary bottleneck was the solver, which occupied well above 70% of the total time for large mesh sizes. By replacing the original solver, `numpy.linalg.solve`, the effect of this bottleneck could be limited. For instance, by replacing the solver with a conjugate gradient solver its time share can be limited to 1.2% without involving the CPU. This can most likely reduced further with GPU acceleration which will be tested during the project. All solvers that will be tested as a replacement for the original one will be able to run on the GPU which potentially can further improved their timings even more. More details about the tested solvers will be presented in section 4.4.

With the new and more efficient solvers, a new bottleneck in the assembler was identified. With the conjugate gradient solver presented in 4.4.3 the assembler now occupies well above 90% of the time. New assemblers will thus also be tested in order to improve the time even more. More information about the specifics of these new assemblers and solvers will follow in the next section.

4.3 The Assembler Functions

As mentioned earlier the assembler function were one of the major bottleneck of the system that were able to benefit the most from GPU acceleration. And to address this problem a few different solutions were tested. The first one was simply moving the original assembler function on to the GPU by using the Numba decorator, which converts Python code into a GPU compatible machine code together with some alterations to the code to match the Numba specific requirements. For instance, not using `SciPy` functions as well as other math functions from the `Math` library. Furthermore, a more specialized assembler that takes advantage of the fact that the data is represented by sparse matrices has been constructed. The main difference compared with the original assembler is that the specialized assembler utilizes lists to store the data and the corresponding x, y coordinates which are used to construct the matrix at the very end of the assembly. The idea of the specialized assembler is built upon the same principals as `Gustavssons` algorithm [6]. In contrast, the original assembly works directly with the stiffness matrix. This is cumbersome when the calculation is off-loaded to the GPU since the entire matrix must be moved from main memory to the GPU memory, even though all the information in the matrix are concentrated along its diagonal. This means that most of the transferred data remain unused while still slowing down the transfer time and occupying valuable memory on the GPU. This specialized assembly will be referred to as the *Efficient* assembler in the test data.

In order to conduct proper tests two more assemblers were constructed. Algorithmically these algorithms are the same as the ones already mentioned, the difference is that they are pre-compiled with Numba which allows them to run directly on the kernel without passing through the Python interpreter. This can potentially speed them up and will make them more similar to the GPU version of the algorithms. The pre-compilation will thus make the algorithms be closer to the peak optimization that is possible to achieve on the CPU and will serve as a better comparison against the GPU versions when comparing CPU optimization against GPU optimization.

The assemblers that will be compared are thus the following:

Each assembler will be paired with a transformer capable of converting dense matrices to sparse ones and vice versa. This is necessary in order to be able to test all assemblers with any type of solver indifferent of the desired input of the solver.

Version	Hardware	Precompiled
Original	CPU	No
Efficient	CPU	No
Original PreComp	CPU	Yes
Efficient PreComp	CPU	Yes
Original GPU	GPU	Yes
Efficient GPU	GPU	Yes

Table 2: List of Assemblers

4.4 Solvers

A number of different pre-written solvers were considered from the NumPy and SciPy libraries to solve the linear system (12). Since the stiffness matrix is known to be sparse, solvers from the submodule `scipy.sparse` were mainly considered. To run the solvers on the GPU their corresponding functions from the CuPy library where used. Two different types of solvers where used, iterative and direct solvers as described in 2.4. Existing solvers where used since they are well-written and optimized in contrast to if we where to implement our own. However, one drawback becomes that we don't know in-depth how they work and all their sub-routines which makes analyzing them more difficult. All solvers used in the benchmark is presented in table (3).

Solver	Type	Method	Stopping tolerance
`linalg.solve`	Direct	LU-decomposition	-
`sparse.linalg.spsolve`	Direct	LU-decomposition	-
`sparse.linalg.lsqr`	Iterative	LSQR	$a_{tol} = 10^{-6}$, $b_{tol} = 10^{-6}$
`sparse.linalg.cg`	Iterative	conjugate gradient method (cg)	$r_{tol} = 10^{-5}$
`sparse.linalg.cgs`	Iterative	conjugate gradient squared method (cgs)	$r_{tol} = 10^{-5}$
`sparse.linalg.minres`	Iterative	minimal residual method (MINRES)	$r_{tol} = 10^{-5}$
`sparse.linalg.lsmr`	Iterative	LSMR	$a_{tol} = 10^{-6}$, $b_{tol} = 10^{-6}$

Table 3: Solver routines used in the benchmark

4.4.1 Linalg.solve

The solver used in the initial code described in 3 is NumPy's direct solver `linalg.solve`. The method uses the LAPACK routine *_gesv* which solves the system by LU-decomposition [7]. This is the only solver considered that isn't from SciPy's Sparse submodule. The performance from this solver is expected to be poor since it's made to solve systems with dense matrices. Nevertheless it is still used as a indication of the worst case scenario to emphasize the importance of choosing a somewhat optimized algorithm to increase performance.

4.4.2 Sparse.linalg.spsolve

The other direct solver considered is `sparse.linalg.spsolve`. The complexity of the function heavily depends on structure of the matrix. The method uses UMFPACK which solves the system by LU-decomposition [8].

4.4.3 Sparse.linalg.cg

The function `sparse.linalg.cg` implements the iterative conjugate gradient method which is described in 2.4.1. The method assumes that the matrix is symmetric and positive definite to guarantee convergence. The convergence rate of the conjugate gradient method relies on the condition number of S. To reduce the condition number and also the convergence rate, preconditioning is often applied. The stopping criteria is given by

$$||f - Sa||_2 \leq r_{tol}||f||_2. \tag{29}$$

4.4.4 Sparse.linalg.cgs

The function `sparse.linalg.cgs` implements the conjugate gradient squared method and unlike the conjugate gradient method S does not need to be symmetric or positive definite [9]. The method is also sensitive for ill-conditioned S which affects the convergence rate. The stopping criteria is given by (29).

4.4.5 Sparse.linalg.lsqr

The function `sparse.linalg.lsqr` implements the LSQR algorithm which is an iterative method for solving $\min ||Sa - f||_2^2$ and hence also (12) [10]. The algorithm is based on bidiagonalization, which is a form of matrix decomposition, using Lanczos algorithm. Unlike the conjugate gradient method, the LSQR algorithm performs well when S is ill-conditioned and S doesn't have to be symmetric or positive definite. The stopping criteria is given by

$$||r||_2 \leq a_{tol}||S||_2||a||_2 + b_{tol}||f||_2, \tag{30}$$

which means that if $a_{tol} = b_{tol} = 10^{-6}$ then $||r||_2 \sim 10^{-6}$.

4.4.6 Sparse.linalg.minres

The function `sparse.linalg.minres` implements the iterative minimal residual method which solves (12) by solving $\min ||Sa - f||_2^2$ [11]. Unlike the conjugate gradient method, S does not need to be positive definite, but still symmetric. The algorithm is based on Lanczos tridiagonalization. The stopping criteria is given by (29).

4.4.7 Sparse.linalg.lsmr

The function `sparse.linalg.lsmr` implements the LSMR algorithm which solves the system by solving $\min ||Sa - f||_2^2$ [12]. The algorithm is like LSQR based on bidiagonalization but also uses regularization to improve stability. The solver is therefore suitable for ill-condition S. The method doesn't require S to be symmetric or positive definite. The stopping criteria is given by (30).

4.5 Mesh with Triangular Elements

The mesh created with the triangular elements used the same solvers as the quadrilateral mesh net. However, the assembler function used for this net had to be reworked since the assembly process differs when there less amount of nodes per element. The chosen assemblers for the triangular elements were a subset of those in the quadrilateral solution. The assemblers were:

Version	Hardware	Precompiled
Triangular Original	CPU	No
Triangular Original GPU	GPU	Yes
Triangular Efficient GPU	GPU	Yes

Table 4: List of Triangular Assemblers

4.6 Running on Vera

The assembler functions and solver found during this project that seem to work well with GPU acceleration will be run in the Vera cluster in order to test their scalability in a larger system. One way to access Vera is through SSH, we can use SSH to log in to the login nodes of Vera, then we can use Sbatch to submit a task to the actual performing nodes that has the powerful GPUs.

For convenience, we write the commands in a shell file so that we can easily run them again and replicate the tests if needed.

4.7 Imported Mesh Nets

The original mesh net with quadrilateral elements was built by this project code, however, there is a solution to import triangular mesh nets via the *GMSH* python library and a *.msh* file. The smallest triangular mesh net used was simple and consisted of 4 equilateral triangles joined into a square. While this mesh was simple the implications of the ability to import mesh nets means that more complex mesh nets could be used as well. The project currently only supports importing mesh nets with triangular elements, however allowing it to import mesh nets with quadrilateral elements would be a trivial improvement.

4.8 Hardware Specifications

The benchmarks were run on a system with the following specifications:

OS: Linux Ubuntu
CPU: 12th Gen Intel(R) Core(TM) i5-12500
GPU: NVIDIA GeForce RTX 3070
RAM: 32GB

We used the following specifications of Vera:

CPU: 'Skylake' 16 cores
GPU: NVIDIA A40 48GB * 1
RAM: 128GB

5 Results & Analysis

The following comparisons is based on the above-mentioned assemblers and solvers. Although here might be functions that are better optimized for both GPU and CPU for this specific type of FEM simulations the functions tested during this project should give an indication of the capabilities for each respective hardware.

5.1 Assembler Comparisons

In figure 3 it is clear that some assemblers perform much better than others. What's more interesting is what specific assemblers perform the best and its increase in performance compared to the others. Its clear that the most optimized GPU assembler, named *assemblyEfficientGPU* in the graph, has a significantly lower time complexity compared to its peers. The second-best performer is the same pre-compiled algorithm, with the only difference being that it is running on the CPU instead of the GPU. With both of the algorithms running at sub-second times for all tested inputs with a time complexity between $O(n)$ and $O(logn)$, where n is the number of *NDofs* in the simulation. The time measurements includes the transfer time between GPU and main memory which indicates that the possible parallelization available on the GPU outweighs the drawbacks created by data transfers.

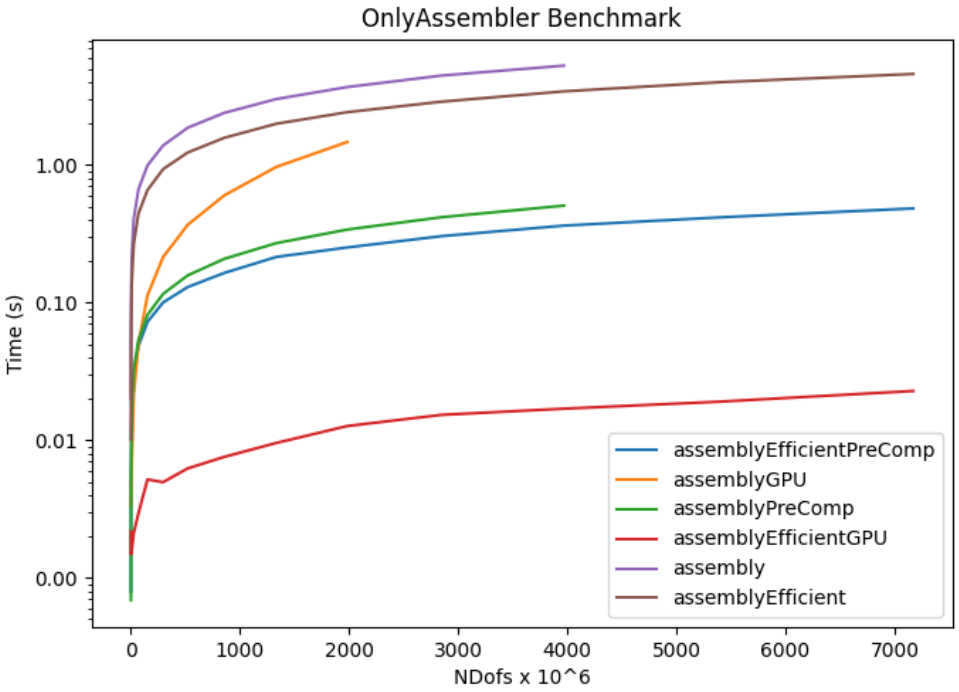


Figure 3: Assembler Execution Time

Another noteworthy conclusion that can be drawn from the test data is that all of the original assemblers crashes due to memory limitations. This is visible in figure 3 where none of *assemblyGPU*, *assemblyCPU* and *assemblyPreComp* are able to reach the same level of *NDofs* as the efficient solvers under the same set of tests. This limitation is even more severe on regular GPUs, which often have smaller on-board memory than what is available to access the CPU on the main memory. This effect can be seen in the data for *assemblyGPU* which aborts the test far earlier than all other assemblers. This problem could be avoided on larger GPU clusters which often have more on-board memory available for its GPUs, such as the Vera cluster discussed in section 2.6.

There were also indications that the transformation between sparse and dense matrices that were discussed in

the end of section 4.3 could greatly affect the assembler time when an assembler that created a sparse matrix were paired with a solver that required a dense input. In some cases this transformation could occupy above 85% of the time needed to assemble the matrix. For the comparison in the above figure, the transformation time for between matrix types are excluded. The graph therefore displays the optimal assembly time for each algorithm on the set hardware. For the assembler, it appears to be a combination of GPU acceleration, together with an as efficient assembler algorithm as possible, which also matches the desired input type of the solver that performs the best.

5.2 Solver Comparisons

The results from all the tested solvers are presented in figure 4. The original solver (`linalg.solve`), together with the GPU version of the conjugate gradient square implementation (`cgs`) shows the worst performance with an approximate time complexity similar to $O(n)$.

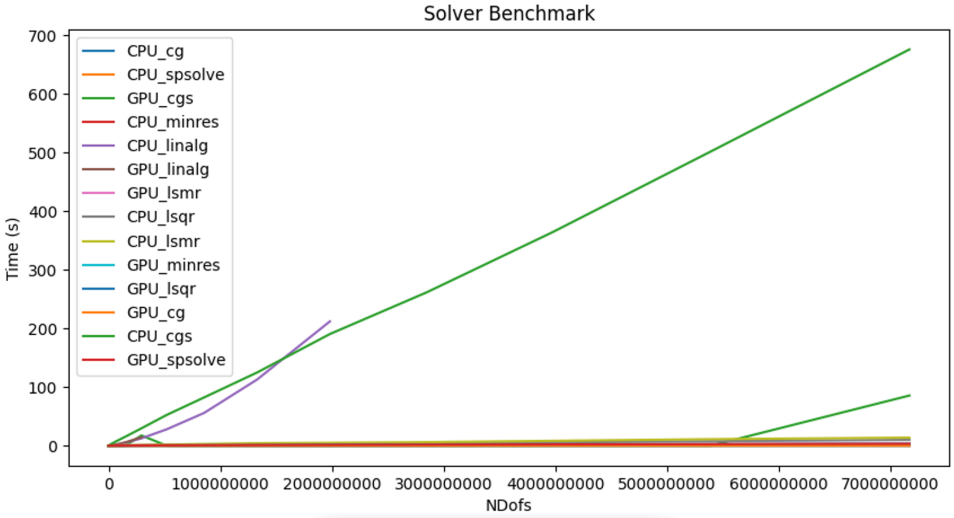


Figure 4: Solver Execution Time

While Figure 4 illustrates the scale of differences between the various solvers, a logarithmic scale graph is needed to highlight performance variations among the sparse solvers. The result of this can be found in figure 5. We note that all solvers perform better when being runned on the GPU with the expection being the *MINRES* implementation. As evident in the graphs, most of the sparse solvers perform better than the general LU-decomposition algorithm for dense matrices (`linalg.solve`), independent of CPU or GPU implementation and is as expected. Moreover, the sparse LU-decomposition method `spsolve` outperforms some of the iterative sparse solvers. The reason could be that the stiffness matrix S having a simple structure which the method could make use of. Note that a relatively high tolerance of 10^{-5} or 10^{-6} where used for the iterative solvers. Using a higher tolerance would make each iterative solver slower, but the order of the result would remain which we are more interested in. The best-performing solvers are the GPU implementations of the *cg* method (not *cgs*) and the *MINRES* method aswell as the CPU implementation of *MINRES*. Remember that the stiffness matrix S is symmetric and semi-positive definite. The reason of the *MINRES* implementation is the fastest could be that the method requires S to be symmetric but not positive definite, which are properties the system satisfies. The method makes use of these properties which may lead to quicker convergence. Other methods, such as the *cgs*, *LSQR* and *LSMR* implementations does not have the same requirements for convergence which may make them more suitable for more general sparse systems. Quite surprisingly, the conjugate gradient squared method (*cgs*) performs considerable poorly. The reason being unknown but could be that the method is not as efficiently written or more suitable for systems with different properties. Remember that the inner workings of each method is unknown and their performances depend on how they are implemented. Note also that the *cg* method assumes the stiffness matrix S is positive

definite, which the stiffness matrix satisfies.

The general matrix solvers displays a similar behavior as the general assembler by interrupting its test execution due to memory limitations. For intstance the CPU implementation of *linalg.solve* aborts its execution at 1.9×10^9 *NDofs* with a execution time of 212s. The *MINRES* solver has a execution time of 0.018s for the same number of *NDofs*. That is a $11777x$ speed-up factor compared to the general matrix solver.

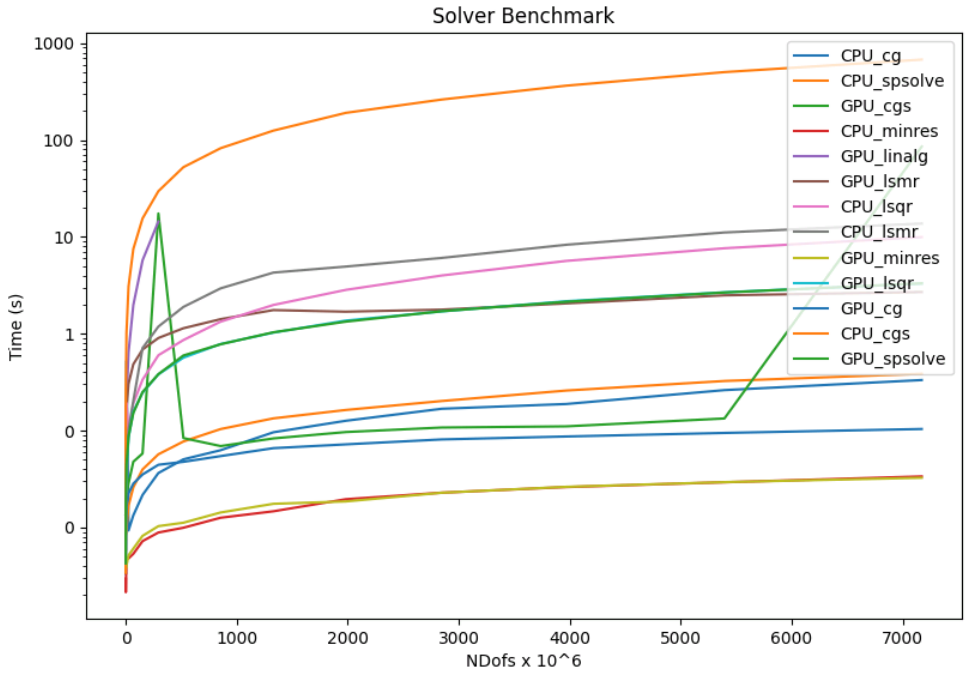


Figure 5: Solver Execution Time

The solver's behavior displays the same characteristics as the asssemblers where a combination of GPU acceleration and a carefully chosen algorithm that is tailored towards the specific matrix structure (in this case sparse, symmetric and semi-positive definite) outperforms the others by a large margin.

5.3 Combined Metrics

When comparing the best-performing assembler/solver combination against its original counterpart it is clear that the best-performing combination, the *assemblyEfficientGPU* combined with the *MINRES* solver, has a greatly improved assembly time and a close to non-existent solve time. When compared to its original counterpart which both lacks efficient assembly and solver. Notice that the CPU *MINRES* solver performs as well as the GPU version for the tested input sizes, both of these might thus be utilized. Furthermore, the best-performing combination can handle heavier loads without exhausting its memory and thus compute a more fine-grained solution. The optimal combination thus utilizes both the memory and computational resources far better than its original counterpart.

Version	NDofs $\times 10^9$	Assembler(s)	Solver(s)	Total(s)
Original	1.9	3.687	212.22	215.907
Original	7.2	-	-	-
Optimal CPU	1.9	0.251	0.019	0.270
Optimal CPU	7.2	0.483	0.033	0.516
Optimal GPU	1.9	0.012	0.018	0.030
Optimal GPU	7.2	0.022	0.032	0.054

Table 5: Best Performing Combinations

5.4 Performance on Vera cluster

Running the GPU-compatible algorithms on the Vera cluster shows a similar trend to the single GPU workstation in the original test. The original assembly method failed quite early due to memory limitations, even with the extended memory available in the cluster. The assembler optimized for sparse matrices scales well with the increasing size and successfully completes the test with a *NDof* size up to $350 * 10^9$. This is well above $200\times$ the size in the original test which clearly shows the computational capabilities of a GPU cluster with extended memory when combined with an efficiently structured algorithm.

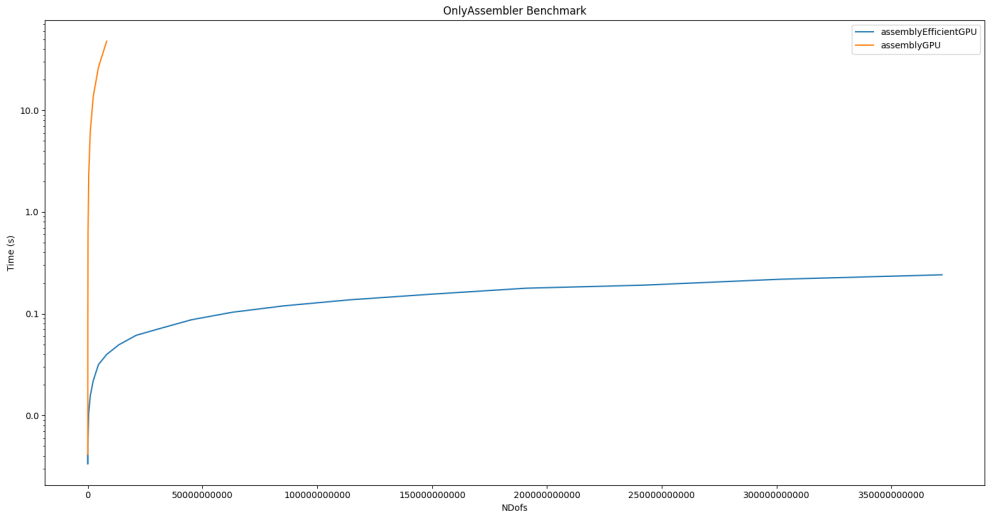


Figure 6: Assembly Time On Vera Cluster

The solvers show a similar scalability in the Vera cluster which can be observed in Figure 7. Where the *MINRES* solver scales exceptionally well with a time complexity close to $O(log(n))$. This is similar to the scalability shown in the non-cluster test seen in Figure 5 extended to far greater *NDofs* sizes.

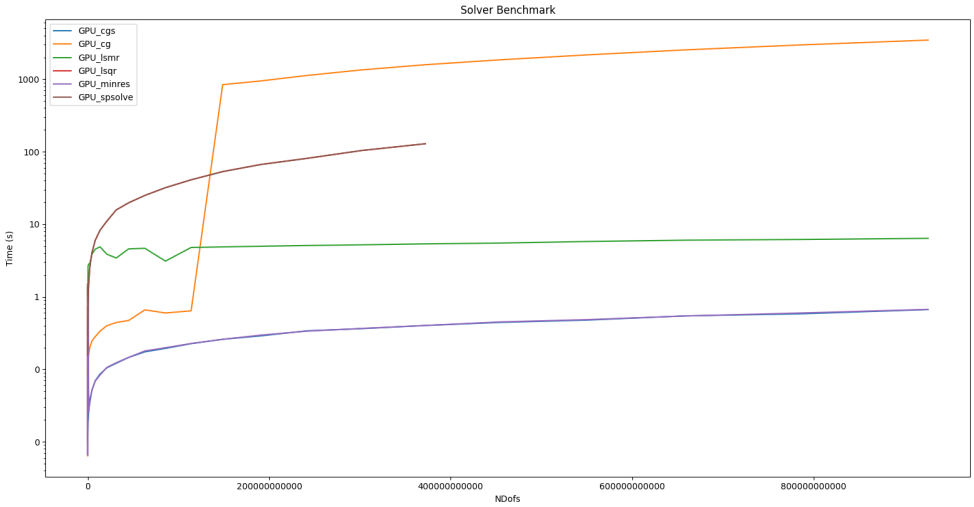


Figure 7: Solver Time On Vera Cluster

It is thus clear that the best-performing algorithms tested in this project scale well with increasing mesh sizes without exhausting too much of the underlying resources. This is in comparison to the original assembly and solver methods which require far greater memory sizes and computational resources as presented in section 5.3.

5.5 Triangular Elements

While the project's main focus was mesh nets with quadrilateral elements, there was also an implementation for triangular elements. This implementation did not have the same number of assemblers or its own solvers but rather used the solvers used in the quadrilateral calculations. This part of the project was used as a control group and a foundation for future work.

5.5.1 Triangular Element Assembler Comparisons

Three different assemblers were implemented that could handle triangular elements. These assemblers, $tri\_assembly$, $tri\_assemblyGPU$, and $tri\_assemblyEfficientGPU$, were based on the corresponding assemblers from the previous part of the project but slightly reworked to handle triangular objects.
Figure 8 shows how the three different assemblers performed and it shows that $tri\_assemblyEfficientGPU$ was the fastest and the only assembler which could compute up to at least $7.9 * 10^9$ Ndofs, ultimately making it the optimal choice of assembler. The assemblers also perform very similarly to their counterparts in the quadrilateral solution, as seen in Figure 8 and Figure 3. However, mesh nets with triangular elements have higher connectivity, and the assemblers for the triangular elements were slightly less optimized than those for the quadrilateral making these assemblers marginally slower.

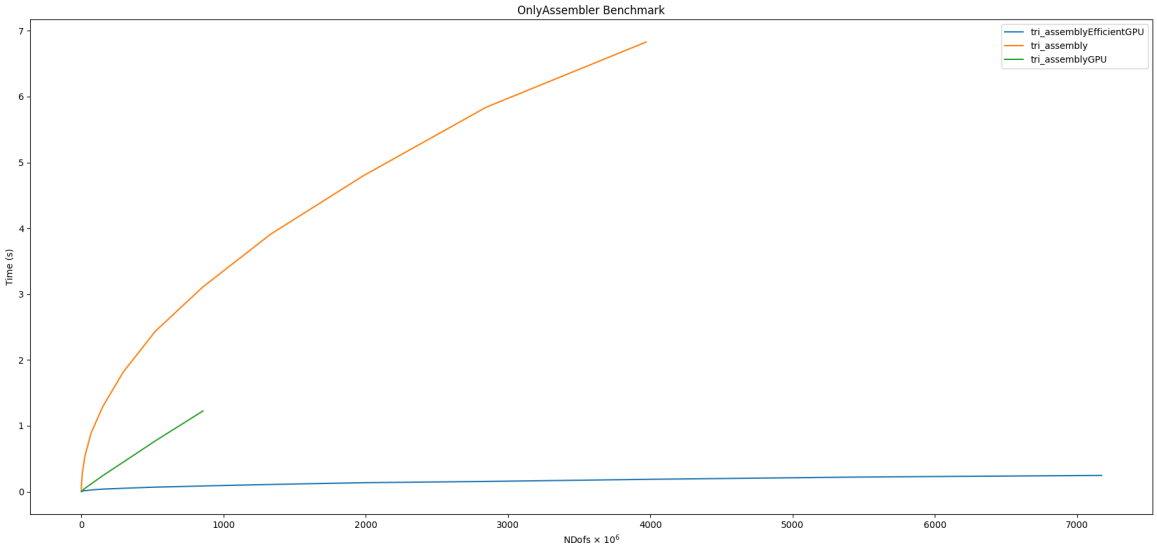


Figure 8: Logarithmic Graph of Triangular Assembler Execution Times

5.5.2 Triangular Element Solver Comparisons

Like in the quadrilateral calculations, the speed between the solvers varied greatly. Four solvers underperformed greatly. The solvers in question are, CPU_linalg, GPU_linalg, CPU_lsmr, and CPU_lsqr. The first was the original solver and this nor its GPU counterpart could handle $7.2 * 10^9$ NDofs. The latter two were significantly slower than the other solvers, around 5x and 2.5x that of the fifth slowest solver (GPU_lsmr). The complete results of every solver can be seen in Figure 9 and 10, and the result without the top 4 worst solvers can be seen in Fig 11. The results of the triangular element mesh are similar to those of the quadrilateral mesh. The main difference, however, is the overall increase in speed for all solvers. This was an expected result since a triangular mesh net should be faster to calculate [13] because there is one less node and Guass integration point per triangular element compared to the quadrilateral elements.

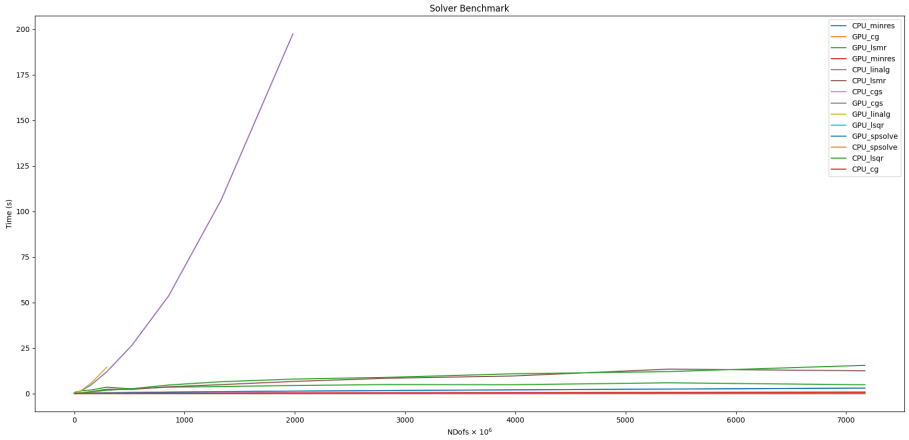


Figure 9: Linear Graph of All Triangular Solver Execution Times

The *MINRES* solver had an execution time of 0.009s at $1.9 * 10^9$ Ndofs, this is a speed increase of 22458x compared to the slowest solver, *linalg.solve* (CPU_linalg) which ran for 197.518s. It's also a speed increase of 2x compared to the same number of Ndofs with the *MINRES* solver used on a mesh net with quadrilateral elements.

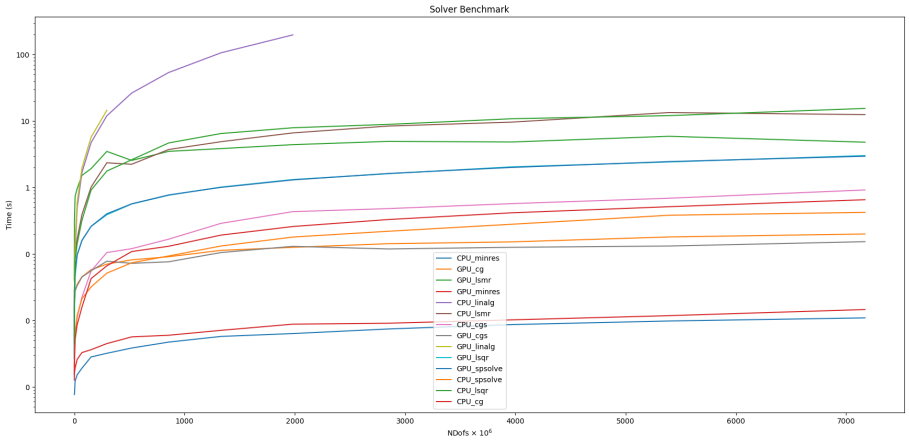


Figure 10: Logarithmic Graph of All Triangular Elements Solver Execution Times

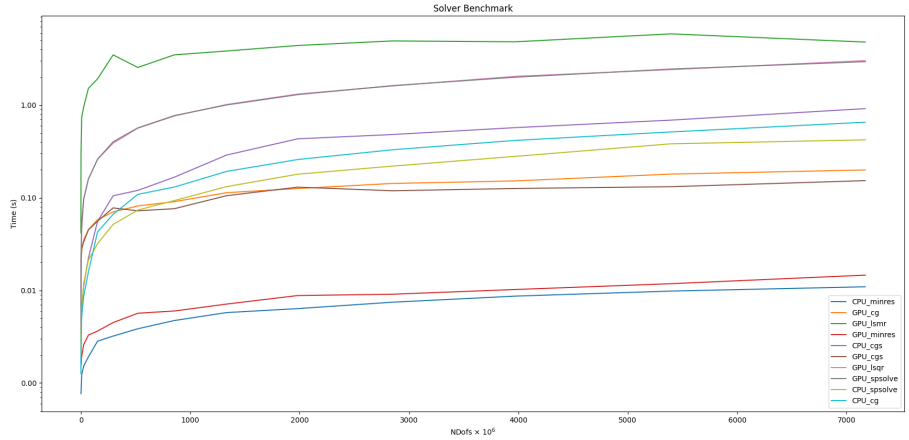


Figure 11: Logarithmic Graph of the Top 10 Triangular Elements Solver Execution Times

5.5.3 Combined Triangular Element Comparisons

The optimal combined solution was to use the *tri_ assemblyEfficientGPU* assembler along with the *MINRES* solver, which was the same result as the quadrilateral tests. The combined result for this configuration was slower than that for the quadrilateral even though the solver was twice as fast. The culprit for this result was the assembler which was slower on the triangular solution. The results can be seen in Figures 12, 13, 14, Table 6.

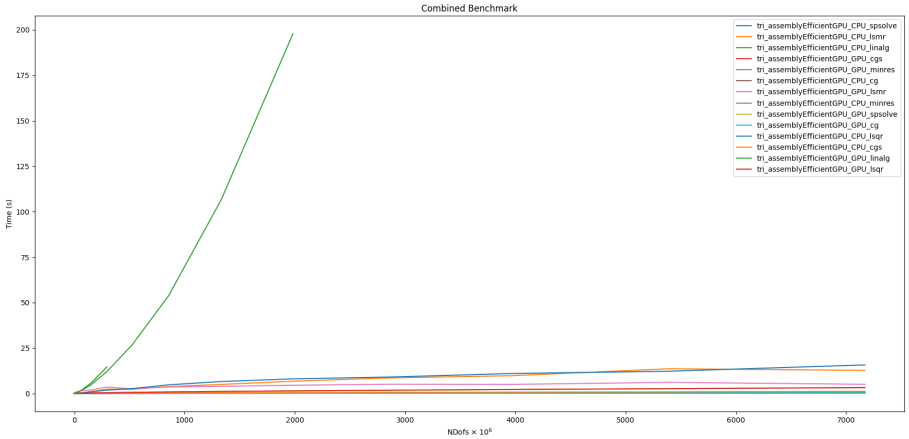


Figure 12: Linear Graph of All Triangular Elements Combined Execution Times

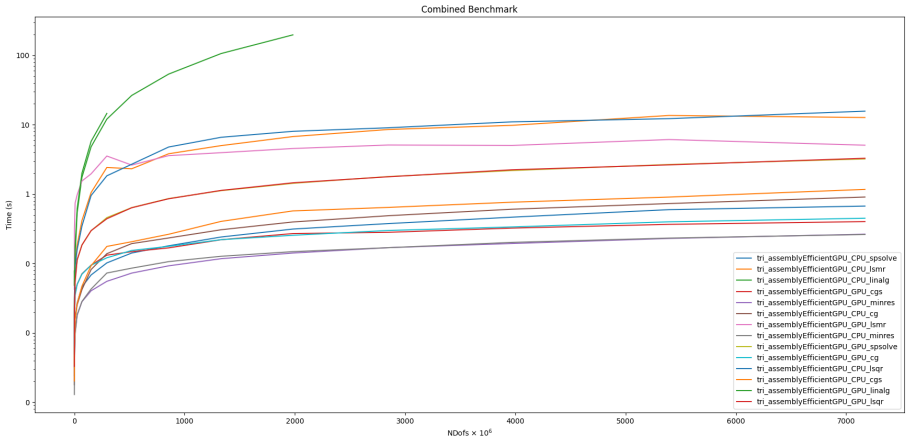


Figure 13: Logarithmic Graph of All Triangular Elements Combined Execution Times

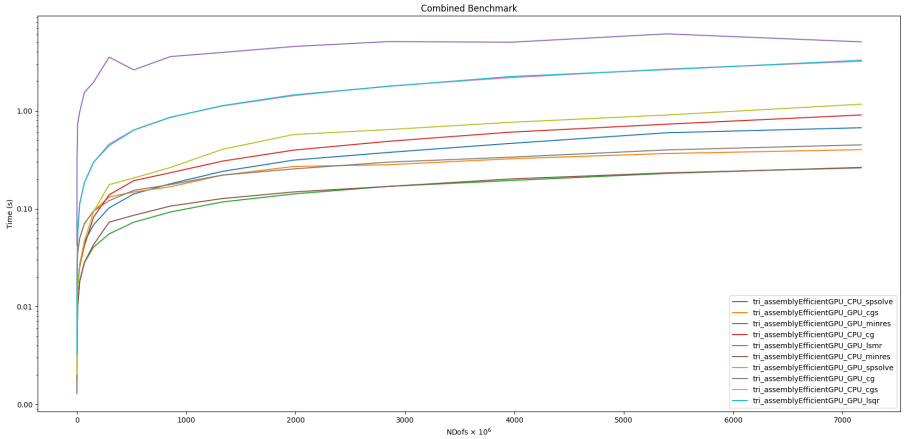


Figure 14: Logarithmic Graph of the Top 10 Triangular Combined Execution Times

Version	**NDofs** $\times 10^9$	**Assembler(s)**	**Solver(s)**	**Total(s)**
Original	1.9	4.800	197.518	202.317
Original	7.2	-	-	-
Optimal GPU	1.9	0.133	0.009	0.142
Optimal GPU	7.2	0.250	0.015	0.265

Table 6: Best Triangular Performing Combinations

5.6 Imported Mesh Nets

The imported mesh net performed significantly worse than the other solutions. This was because of how the scaling for the imported elements behaved. There was a necessary step where the new mesh net had to be saved to an intermediate temporary file. This file ended up being several million lines long, slowing the operations down and eventually forcing the process to kill itself. The results of some imported mesh nets can be seen in Figures 15, 16, and 17 which shows the assembly, solver and combined times respectively.

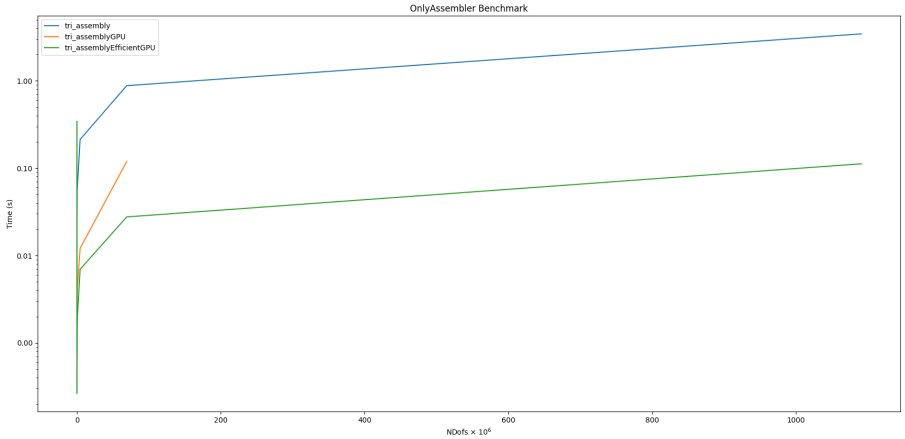


Figure 15: Logarithmic Graph of the All Imported Elements Assembler Execution Times

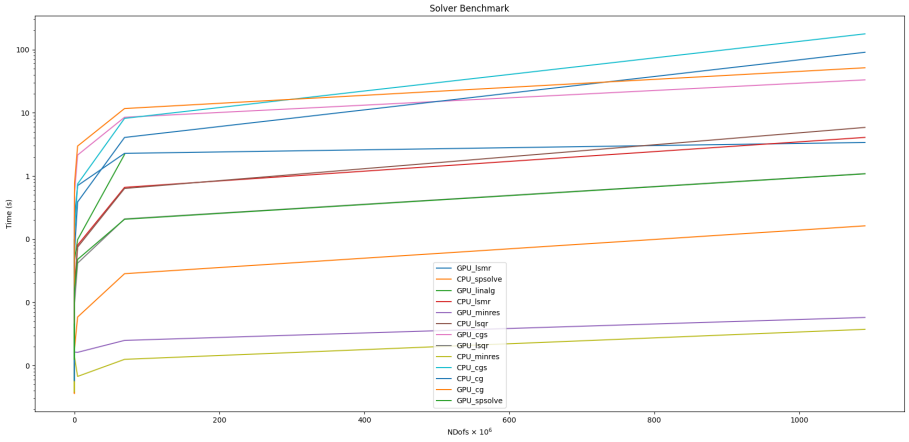


Figure 16: Logarithmic Graph of All Imported Elements Solver Execution Times

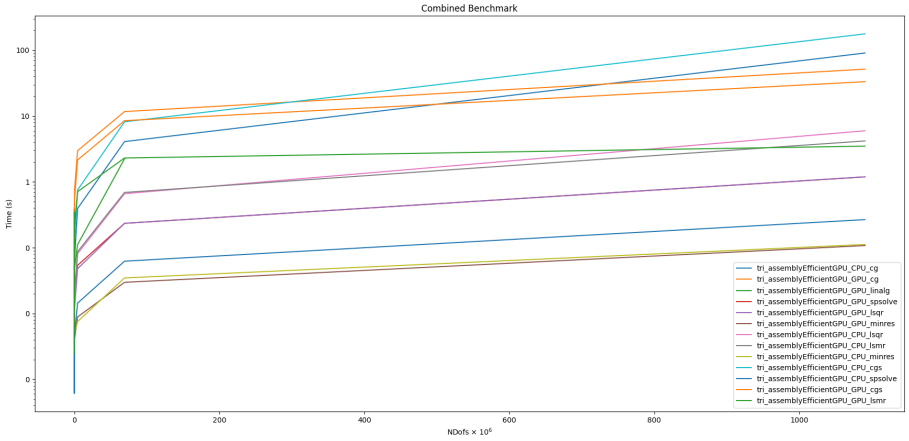


Figure 17: Logarithmic Graph of All Imported Elements Combined Execution Times

6 Conclusion

We improved the original FEM code on both CPU and GPU side, on the GPU side, we used CUDA to accelerate the assembler and the solver by using cupy and numba. Moreover, we further explored the mesh net shape from Quadrilateral to Triangular.

From the performed tests in this project, it is clear that GPU acceleration are applicable to computer-aided FEM simulations independently of the mesh size. As can be seen in figure 7 where the optimal GPU implementation outpaces its competitors. However, it is worth noting that an optimized CPU algorithm performs much closer to the optimal GPU implementation, with a one-order-of-magnitude difference. Compared to a two-order-of-magnitude difference to the original implementation.

Version	**Assembler**	**Solver**	**Performance Gain**
Original	Original	Linalg.solve	1x
Optimal CPU	Efficient PreComp	Minres CPU	800x
Optimal GPU	Efficient GPU	Minres GPU	7197x

Table 7: Comparison of versions and their performance

Even for small meshes the overhead of transferring data between main memory and GPU is a minor part of the total time for the simulation. It is also clear that the applied algorithms have a large implication for the possible optimization. It is thus the case that moving any slow/complex algorithm onto the GPU will not guarantee a speed-up in the needed computing time. In some cases, it might even make the available scope smaller due to other limitations such as memory. This was the case with the *assembly* and *assemblyGPU* algorithms presented in the project, see Figure 2.3.

A conclusion drawn from this may thus be that it requires some "algorithmic know-how" and knowledge of the GPUs resources to take full advantage of the available *CUDA* interface. However, when done correctly and applied to the correct underlying algorithms the time complexity can be reduced greatly.

6.1 Future Improvements

When profiling the best solution achieved in this project there is a somewhat even split between the time needed for the assembler and solver. With the assembler occupying 40% of the total time, and the solver needing the other 60%. Even though the solver occupies more time it is most likely the assembler that could be improved a bit more in the future. The reasoning for this is that the solver is a pre-constructed function from the *scipy* library and is most likely already optimized, while the assembler is custom-made for this specific project. It is thus more likely that we missed some possible minor optimization steps compared to the possibility that the sparse solvers available in Scipy are badly optimized. Another important optimization would be to look further into the triangular assemblers, and see where they could be optimized.

The solvers used during the project could be considered tailored towards the specific mesh and matrix structure that are solved for in this FEM problem. And it is not certain that the same structure would be used in other FEM models which could make other FEM problems considerably slower to solve compared to the results achieved in this project.

Another possible improvement is to utilize multiple GPUs of Vera to further accelerate the process. Our current tests are run on a single GPU, but the *Cupy* library has multiple GPU support which could be used in the Vera cluster. For example the **cupy.cuda.nccl** module is designed for communication between GPUs, when we use iterative solvers. Theoretically, we can use more GPUs to solve even larger problems when distributing the calculations onto multiple GPUs which not only boosts the number of cores but also the available memory. With proper partitioning of the algorithm, this could allow for faster calculations of a FEM (or similar) problem.

It would also be interesting to develop different element shapes further. For example, adding more assemblers

for the triangular elements and seeing if there is another element shape that could be incorporated, such as hexagons or 3D elements like tetrahedrons. Creating solutions for additional shapes would mean that more complex mesh nets, such as spheres or cylinders, could be imported.

7 Contributions

While all major decisions along the project were made as a group the work was somewhat divided to increase the efficiency of the group. This allowed us to research multiple problems in parallel that needed to be solved to reach our goals. These problems included underlying math of FEM simulations, converting algorithms to *CUDA* compatible counterparts, understanding the *Vera* cluster, and so on. Below are the individual contributions of each member stated.

Carl: Main responsibility of looking into the theory behind the given FEM problem and based on that find appropriate solvers to include in the benchmark. Did some very early programming in the project, where a script was made to plot the results (which was further developed by other members). Wrote on sections 2.1, 2.2, 2.3, 2.4, 3, 4.4, and a bit on 5.2.

Sebastian: Provided the foundation of the test setup described in section 4.1 with separate solver and assembler timers, and pre-compilation of Numba annotated methods. Wrote the new algorithms for the efficient solvers and its CUDA-compatible counterpart. Created a test script to automatically test all available algorithms to provide the test data needed for the report. Wrote most of the *Method* and *Result & Analysis* chapters.

Erik: I was responsible for the triangular and imported mesh net implementations, which means that I rewrote some of the code to work with different element shapes and performed all benchmark tests for them. Early on, I also looked at 3D shapes, but we decided not to continue with that. Finally, I wrote the abstract, introduction and everything related to the triangular and imported mesh nets.

Yanchen: I was mainly responsible for running the experiments on the Vera cluster: I investigated how Vera works, wrote the script to run the experiments on Vera, debugged, and logged the results. I also did preliminary investigation on the results and pointed out the anomalies in them, and they are later solved by other group members. I wrote the introductory parts that are relevant with Vera in the report, and also contributed in the limitations part.

References

[1] S. Georgescu, P. Chow, and H. Okuda, "Gpu acceleration for fem-based structural analysis," *Archives of Computational Methods in Engineering*, vol. 20, May 2013. DOI: 10.1007/s11831-013-9082-8.

[2] A. Dziekonski, A. Lamecki, and M. Mrozowski, "Gpu-accelerated finite element method," in *2016 IEEE MTT-S International Conference on Numerical Electromagnetic and Multiphysics Modeling and Optimization (NEMO)*, 2016, pp. 1–2. DOI: 10.1109/NEMO.2016.7561602.

[3] C3SE at Chalmers University of Technology in Gothenburg, *Vera computer cluster*, https://www.c3se.chalmers.se/about/Vera, Accessed: 2024-12-16.

[4] Wikipedia, *Conjugate gradient method — Wikipedia, the free encyclopedia*, http://en.wikipedia.org/w/index.php?title=Conjugate%20gradient%20method&oldid=1266914806, [Online; accessed 04-January-2025], 2025.

[5] *Compiling python code with @jit — numba*, 2024. [Online]. Available: https://numba.pydata.org/numba-doc/dev/user/jit.html.

[6] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Transactions on Mathematical Software (TOMS)*, vol. 4, no. 3, pp. 250–269, 1978.

[7] E. Anderson, Z. Bai, C. Bischof, *et al.*, *Gesv: Factor and solve*, Last accessed 4 january 2025. [Online]. Available: https://netlib.org/lapack/explore-html//d8/da6/group__gesv.html.

[8] T. A. Davis, "Algorithm 832: Umfpack v4.3—an unsymmetric-pattern multifrontal method," *ACM Transactions on Mathematical Software*, 2004.

[9] Wikipedia, *Conjugate gradient squared method — Wikipedia, the free encyclopedia*, http://en.wikipedia.org/w/index.php?title=Conjugate%20gradient%20squared%20method&oldid=1264240817, [Online; accessed 04-January-2025], 2025.

[10] C. C. Paige and M. A. Saunders, "Lsqr: An algorithm for sparse linear equations and sparse least squares," *ACM Trans. Math. Softw.*, vol. 8, no. 1, pp. 43–71, Mar. 1982, ISSN: 0098-3500. DOI: 10.1145/355984.355989. [Online]. Available: https://doi.org/10.1145/355984.355989.

[11] I. S. Duff and J. K. Reid, "The multifrontal solution of indefinite sparse symmetric linear," *ACM Trans. Math. Softw.*, vol. 9, no. 3, pp. 302–325, Sep. 1983, ISSN: 0098-3500. DOI: 10.1145/356044.356047. [Online]. Available: https://doi.org/10.1145/356044.356047.

[12] Q.-x. Huang, F. Wang, J.-h. Yan, and Y. Chi, "A two-step discrete method for reconstruction of temperature distribution in a three-dimensional participating medium," *International Journal of Heat and Mass Transfer*, vol. 55, no. 9–10, pp. 2636–2646, Apr. 2012, ISSN: 0017-9310. DOI: 10.1016/j.ijheatmasstransfer.2011.12.029. [Online]. Available: http://dx.doi.org/10.1016/j.ijheatmasstransfer.2011.12.029.

[13] A. Jabari Kh, *Quad-map mesh or tri grid mesh,which one is better?* Apr. 2021.