Finite Element for 2D Solid Mechanics
GPU Accelerated Numerical Method with Python and CUDA

Congxiao Zhang
Author

Gayana Jinde Radhakrishna
Author

Fredrik Larsson
Supervisor

Abstract

GPU has evolved beyond its initial purpose as a graphics accelerator to become a co-processor for computation-intensive tasks. A substantial growth in the speedup is evident for computationally dense tasks when implemented on GPUs. This acceleration could be achieved in two ways: Using GPU-targeted libraries for algebraic operations or replacing the entire matrix solver with one capable of GPU acceleration. Prominently, CUDA has provided developers with much broader set of software tool choices. Hence, the project aims at developing a GPU accelerated Finite Element Method solver written in python and CUDA. This paper holds a brief overview of our problem statement, background details, simple workflow of the FEM along with various implementation details. Overall, we tend to provide a fair comparison between different matrix solvers and data formats that are exploited by such solvers.

Keywords: GPU, Python, Solid Mechanics, Finite Element Method

1 Introduction

Generally, a two-dimensional elasticity problem can be expressed by a system of coupled second-order partial differential equations. The main variables are the displacements in the coordinate directions. After solving for the displacements, stresses and strains can be calculated from the derivatives of displacements. We can also use the displacements to get a new mesh structure and refine the mesh to get a more accurate solution. However, when the grid size is small enough, the problem will transfer to a large-scale problem. And the computation cost (space and time) can be huge. As the Matrix is sparse and nice, it's possible for us to use GPU to accelerate the whole computational process.

2 Basic Theory of Finite Element Method in Solid Mechanics

2.1 Preliminary

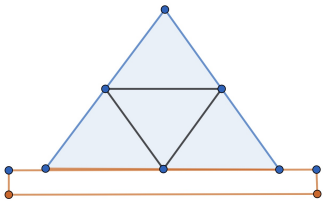


Figure 1: 2D general solid element based in continuum equilibrium

Figure 1 shows a common element structure based on continuum equilibrium. If we use a General Approximation Method to get the displacement u_{2D}, we need to focus on one cell (There are four cells in Figure 1.), which is shown in Figure 2.

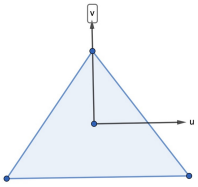
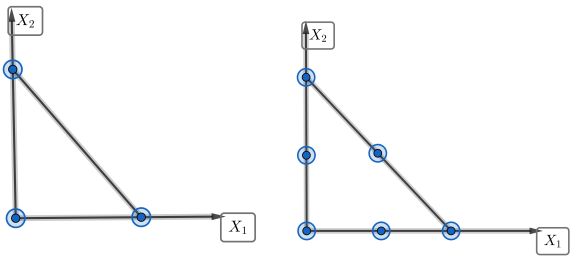


Figure 2: General Element

2.2 Element (Cell)

2.2.1 Triangular Solid Elements

In General Cases, there are usually 2 kinds of Triangular Solid Elements:



(a) Linear Triangular Element (b) Quadratic Triangular Elements

Figure 3: Triangular Element

In this project, we use the Linear Triangular Element. And it can be transferred to a Constant Strain Triangle.
Consider now a triangular element as below. The nodes of the element are numbered 1, 2, and 3 counter-clockwise

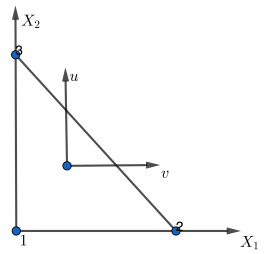


Figure 4: Constant Strain Triangle

Thus, we can assume the linear approximation function between nodes as

$$u_{2D} = \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} a_0 + a_1 x_1 + a_2 x_2 \\ b_0 + b_1 x_1 + b_2 x_2 \end{bmatrix} \tag{1}$$

For 2D solid elements, the field variable is the displacement, which has two components (u_i and v_i), and hence each node has two degrees of freedom (DOFs).
Since a linear triangular element has three nodes, the total number of DOFs of a linear triangular element is six. For the triangular element, the local coordinate of each element can be taken as the same as the global coordinate since there is no advantage in specifying a different local coordinate system for each element.

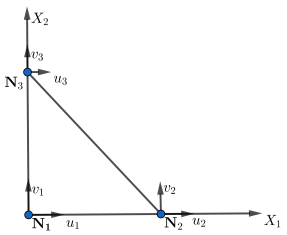


Figure 5: Nodal Displacement of a constant strain triangle

Then we can give out the linear approximation of Nodal Displacement as

$$u_{2D} = \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} u_1 N_1 + u_2 N_2 + u_3 N_3 \\ v_1 N_1 + v_2 N_2 + v_3 N_3 \end{bmatrix} \tag{2}$$

2.3 Shape Function Construction

For General Case, we start with an assumption of shape functions directly using polynomial basis functions with unknown constants.
For a linear triangular element, we assume that the shape functions are linear functions of x and y. They should, therefore, have the form of

$$N_1 = a_1 + b_1 x + c_1 y \tag{3}$$
$$N_2 = a_2 + b_2 x + c_2 y \tag{4}$$
$$N_3 = a_3 + b_3 x + c_3 y \tag{5}$$

which can be rewritten in a concised form

$$N_i = a_i + b_i x + c_i y = \begin{bmatrix} 1, x, y \end{bmatrix} \begin{bmatrix} a_i \\ b_i \\ c_i \end{bmatrix} \tag{6}$$

Note that the above equation is written for the shape functions, and not for the displacements. For this particular problem, we use up to the first order of polynomial basis. Depending upon the problem, we can use a higher order of polynomial basis functions. The complete order of polynomial basis functions in two-dimensional space up to the nth order can be given by using the so-called Pascal triangle.
This can be really complicated in some cases. Considering the matrix $\begin{bmatrix} a_i \\ b_i \\ c_i \end{bmatrix}$ in our case is full rank, we can simplify the procedure of the construction of shape function by generate N_i based on the Hamilton Principle

$$\delta \int_{t_1}^{t_2} L dt = 0 \tag{7}$$

where The Langrangian functional, L, is obtained using a set of admissible time histories of displacements, and it consists of

$$L = \Pi - T - W_f \tag{8}$$

where T is the kinetic energy, Π is the potential energy (for our purposes, it is the elastic strain energy), and W_f is the work done by the external forces.
This δ is a Kronecker delta, which states that the shape function must be a unit at its home node and zero at all the remote nodes. For a two-dimensional problem, it can be expressed as

$$N_i(x_j, y_j) = \begin{cases} 1 & if \quad i = j \\ 0 & if \quad i \neq j \end{cases} \tag{9}$$

So in our case the shape function can be generated as

$$[N] = \begin{bmatrix} N_1 & 0 & N_2 & 0 & N_3 & 0 \\ 0 & N_1 & 0 & N_2 & 0 & N_3 \end{bmatrix} \tag{10}$$

$$= \begin{bmatrix} 1 - X_1 - X_2 & 0 & X_1 & 0 & X_2 & 0 \\ 0 & 1 - X_1 - X_2 & 0 & X_1 & 0 & X_2 \end{bmatrix} \tag{11}$$

So we can seperate the nodal displacements and shape functions as

$$u_{2D} = \begin{bmatrix} N_1 & 0 & N_2 & 0 & \cdots & N_n & 0 \\ 0 & N_1 & 0 & N_2 & \cdots & 0 & N_n \end{bmatrix} \begin{bmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ . \\ . \\ . \\ u_n \\ v_n \end{bmatrix} = [N]u_e \tag{12}$$

where $[N]$ is the shape function and u_e is the nodal displacement.

2.4 Strain Vector

Set the general structure of Strain Vector as

$$\varepsilon = \begin{bmatrix} \varepsilon_{11} \\ \varepsilon_{22} \\ 2\varepsilon_{12} \end{bmatrix} = \begin{bmatrix} \frac{\partial u}{\partial X_1} \\ \frac{\partial v}{\partial X_2} \\ \frac{\partial u}{\partial X_2} + \frac{\partial v}{\partial X_1} \end{bmatrix} \tag{13}$$

Remember equation (4) only works for 2D cases.
As we have known that u_e is constant, the differentiation only applies to $[N]$.

$$\varepsilon = \begin{bmatrix} \frac{\partial N_1}{\partial X_1} & 0 & \frac{\partial N_2}{\partial X_1} & 0 & \cdots & \frac{\partial N_n}{\partial X_1} & 0 \\ 0 & \frac{\partial N_1}{\partial X_2} & 0 & \frac{\partial N_2}{\partial X_2} & \cdots & 0 & \frac{\partial N_n}{\partial X_2} \\ \frac{\partial N_1}{\partial X_1} & \frac{\partial N_1}{\partial X_2} & \frac{\partial N_2}{\partial X_1} & \frac{\partial N_2}{\partial X_2} & \cdots & \frac{\partial N_n}{\partial X_1} & \frac{\partial N_n}{\partial X_2} \end{bmatrix} \begin{bmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ . \\ . \\ . \\ u_n \\ v_n \end{bmatrix} = [B]u_e \tag{14}$$

2.5 Stress Vector

By using the Strain Vector, the Stress Vector can be generated as

$$\sigma = \begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{12} \end{bmatrix} = [C] \begin{bmatrix} \varepsilon_{11} \\ \varepsilon_{22} \\ 2\varepsilon_{12} \end{bmatrix} = [C]\varepsilon = [C][B] \begin{bmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ . \\ . \\ . \\ u_n \\ v_n \end{bmatrix} = [C][B]u_e \tag{15}$$

In 2D, all of the stress or strains act in the same plane, and in our case we are talking about the Plane-Strain Case ($\varepsilon_{33} = 0$).

$$[C] = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & 0 \\ \nu & 1-\nu & 0 \\ 0 & 0 & \frac{1-2\nu}{2} \end{bmatrix} \tag{16}$$

where $E \mapsto$ Young's Module and $\nu \mapsto$ Poisson's Ratio.

2.6 Virtual Work

Next we are going to present how to get stiffness matrix and nodal force vector.
Firstly, we must know the Principle of Virtual Work.

$$IVW_e = EVW_e \quad \mapsto \text{continuum case} \tag{17}$$

which means

$$\text{Internal Virtual Work} \mapsto \text{External Virtual Work} \tag{18}$$

Translate the equation(9) into mathematical knowledge,

$$\int_e \sum_{i,j=1}^{3} \varepsilon_{ij}^* \sigma_{ij}^* dx = \int_{\partial e} t_n \cdot u^* ds + \int_e \rho b u^* dx \tag{19}$$

The left side of the equation is the formula of Internal Virtual Work, and we can simply that by the Vectors and Matrix we have got before.

$$\int_e \sum_{i,j=1}^{3} \varepsilon_{ij}^* \sigma_{ij}^* dx = \int_e ([B]u_e^*)([C][B]u_e) dx \tag{20}$$

There is something to remark. u_e^* is not miswritten but it is the virual nodal displacements (Well we don't need to care about it, and the reason is given below.) and u_e is the actual displacements (as we have talked before.).
To throw away u_e^*, some mathematical tricks are needed.

$$\int_e ([B]u_e^*)([C][B]u_e) dx = \int_e (u_e^*[B]^T)([C][B]u_e) dx \tag{21}$$

Insert the External Virtual Work on the right side, we will get

$$u_e^* \int_e ([B]^T)([C][B]) dx u_e = \int_{\partial e} t_n \cdot u^* ds + \int_e \rho b u^* dx \tag{22}$$

Then deal with the External Virtual Work in a similar way. Replace the general displacement vectors with nodal displacement vectors.

$$u_e^* \int_e ([B]^T)([C][B]) dx u_e = \int_{\partial e} t_n \cdot u^* ds + \int_e \rho b u^* dx \tag{23}$$

$$= u_e^* \int_{\partial e} [N]^T t_n ds + \int_e \rho b (u_e^*[N]) dx \tag{24}$$

$$= u_e^* \int_{\partial e} [N]^T t_n ds + u_e^* \int_e \rho b [N]^T dx \tag{25}$$

It's obvious we can cut off the u_e^* on both sides of the equation (14) – (16).
Then we get an efficient way to solve the Stiffness Matrix and Nodal Force Vector.

$$\int_e ([B]^T)([C][B]) dx u_e = \int_{\partial e} [N]^T t_n ds + \int_e \rho b [N]^T dx \tag{26}$$

Then we have the Stiffness Matrix $[k_e]$

$$[k_e] = \int_e [B]^T [C][B] dx \tag{27}$$

$$f_e = \int_{\partial e} [N]^T t_n ds + \int_e [N] \rho b dx \tag{28}$$

In our case, we set $f_e = 0$, so we can ignore it. And we have our specific global equation as

$$\mathbf{Ka} = \mathbf{0} \tag{29}$$

3 Specific Problem for our Project

Our aim in this Project is to analyze how the structure of the plane changes when an external force is added to the plane. This requests us to find out the new coordinates of each node after the deformation happens.
We assume there is a wall on the right side, which means we can add distributed forces at the bottom, top or left side of the plane.

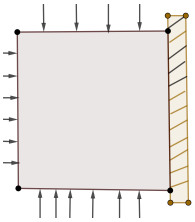


Figure 6: Problem Illustration

As we set the external forces equal to 0 in our case, we realise the initialization by setting some specific values to the displacement of the boundary nodes.
Therefore, we need to partition the Matrix used in our computational process as

$$\begin{bmatrix} \mathbf{K}_{FF} & \mathbf{K}_{FC} \\ \mathbf{K}_{CF} & \mathbf{K}_{CC} \end{bmatrix} \begin{bmatrix} \mathbf{a}_F \\ \mathbf{a}_C \end{bmatrix} = \begin{bmatrix} \mathbf{f}_F \\ \mathbf{f}_C \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix} \tag{30}$$

which is equivalent to

$$\mathbf{K}_{FF}\mathbf{a}_F = \mathbf{f}_F - \mathbf{K}_{FC}\mathbf{a}_C \tag{31}$$

$$= -\mathbf{K}_{FC}\mathbf{a}_C \tag{32}$$

where $\mathbf{a}_F$ means the free DoFs and $\mathbf{a}_C$ means the constraint DoFs.

4 GPU Acceleration for FEM

As the name suggests, Graphic Processing Units (GPUs) were initially designed to accelerate graphical computations. Over years, GPUs have taken a new role of co-processors for dense computations. With growing set of software tools, libraries and frameworks, it has become crucial to implement hardware that is capable of solving time-intense tasks with efficient memory management techniques. There is an ever increasing need for acceleration, given the trend for increasing model size and complexity. Most engineering disciplines use Finite Element Method as a common tool for analysis and thus we tend to inspect the impact of accelerators on such tools in this project, given that, FEM is the only component that could be designed and optimized to perform better in most of the cases. Toward this goal, software developers and researchers are increasingly relying on GPUs, which provide increased processing power and memory bandwidth while consuming less power. Therefore, we evaluate the need for GPUs in our problem, Stress-Strain relations of a 2D plane under the influence of external force.

There are numerous software tools, packages and libraries, that enable users to take advantage of GPU acceleration. Despite linear matrix solver being the most compute-intensive and time-consuming part of the analysis, there has been significant progress in accelerating other parts of the analysis which include computation and assembly of the global stiffness matrix. Thus brief understanding of the stages of the FEM process and analysing the speedup that could be achieved in each stage is necessary. This also helps users to understand how implementation of certain software tools can help achieve required acceleration.

4.1 WorkFlow

The CAD model generated using CAD software undergoes pre-processing, followed by the Solver implementations and finally, post-processing that gives visual results of the process. Figure 7 gives an idea of how a typical FEM workflow would look. The pre-processing step has 2 steps, CAD to 3D/ 2D model conversion and Mesh generation. Creating a computational mesh might be time consuming for industries that produce complex models and therefore could benefit from the computational power of the GPU. However, since parallel mesh generation is computationally tedious to achieve, there are not many attempts made in accelerating this process. Hence, accelerating the solver stage is given at-most priority and known to consume approximately 30 to 35 percent of the total computation.

The FEM Solver stage uses the information such as connectivity, material properties and boundary conditions from the previous mesh generation process and builds a global stiffness matrix, 'K' matrix in our problem. The stiffness matrix 'K' is the integration of all the contributions of the individual elements after being assembled together. This process in called "K-Assembly" or "Matrix-assembly". Followed by the matrix solver, which solves the linear systems of equations that are large and sparse. There are two types pf solvers, iterative solvers and direct solvers. Direct solvers work efficiently for problems of small sizes and they do not suffer convergence problems. While on the other hand, iterative solvers are scalable. The preconditioning and sparse-matrix vector multiplication operations are the most time-consuming parts of iterative solvers. These operations are inherently parallel and can usually be made more efficient by using special data formats that exploit patterns found in the matrix structure (diagonals, dense blocks, etc.). Global Stiffness matrix can be built in 2 steps. First, stiffness matrix for each individual element is computed (ke). Second, all the element stiffness matrices are summed together into global stiffness matrix. Since there is no data dependence between the elements, all

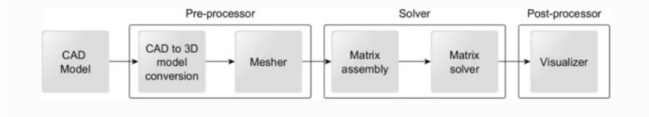


Figure 7: FEM WorkFlow

element stiffness matrices can be computed in parallel, since the type of computations involved are inherently parallel.

The next process in Solver stage is the Matrix Solvers. Iterative solvers were the first type of matrix solvers to be accelerated and most of which targeted Conjugate Gradient solver that are symmetric and positive definite. Choosing optimal sparse matrix data structure to achieve good performance and scalability is necessary. However, studies have proven that the "blocked sparse matrix" format provides faster execution on both CPU and GPU. Since this type of data format was new to us, we used CSR format that is Compressed Sparse Row matrix format which is a well-known, computationally balancing and appropriate to all solvers both in memory management and performance. Further discussion on the various solvers and results are provided in next section.

5 Results and Discussions

5.1 Hardware Information

1. CPU: Intel(R) Core(TM) i7-10870H CPU @ 2.20GHz
2. GPU: NVIDIA GeForce RTX 3060 Laptop GPU

5.2 Numba

Firstly, we used scipy and numba. The running time is shown in the figure below:

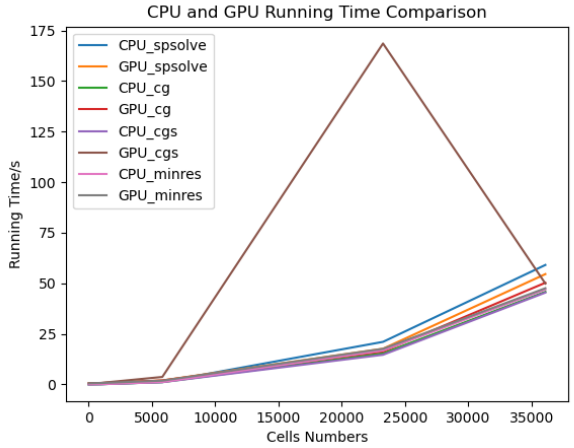


Figure 8: CPU and GPU Running Time Comparison with Numba

The difference in running time between the CPU and GPU is very slight. We tested 4 linear solvers for sparse matrix: **scipy.sparse.linalg.spsolve, scipy.sparse.linalg.cg, scipy.sparse.linalg.cgs and scipy.sparse.linalg.minres**. Except for **spsolve**, CPU always runs faster than GPU, and as cells number increases, minres has the least running time.

Particularly, strange things happened when we use cgs to solve our problem with Cell Size equal to 0.01, Cells Number equal to 23264 on GPU. We repeated the test many times, but it still costs 160+ seconds. Considering this cgs is an iterative solver, there must be some convergence issues for this special case.

Solvers	Cells Number	CPU Running Time	GPU Running Time
spsolve	5824	1.160	1.947
spsolve	23264	21.035	17.632
spsolve	36096	59.054	54.532
cg	5824	1.127	1.935
cg	23264	15.381	16.330
cg	36096	45.778	50.260
cgs	5824	1.098	3.685
cgs	23264	14.646	168.563
cgs	36096	45.393	49.858
minres	5824	1.162	1.858
minres	23264	16.789	17.606
minres	36096	46.966	47.547

Table 1: CPU and GPU Running Time Comparison with Numba

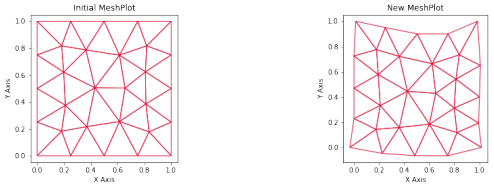
The Results doesn't look so nice, so we try to change the solver alternatives. This test is only target on spsolve. Because **spsolve** is the only one you can choose the solver alternatives, others are iterative solvers, which means you can only change the residual tolerance and set the max iterations.

Solver Alternatives	Cells Number	CPU	GPU
MMD_ATA	5824	0.091	0.404
MMD_ATA	23264	16.622	18.251
MMD_ATA	36096	48.090	52.806
MMD_AT_PLUS_A	5824	0.090	0.607
MMD_AT_PLUS_A	23264	25.078	27.346
MMD_AT_PLUS_A	36096	55.355	61.136

Table 2: CPU and GPU Running Time Comparison with Numba and spsolve: for sparse solver alternatives;

Thus, for **spsolve**, mode = 'MMD_ATA' works fastest.

5.3 An Example about the solution of this Problem



(a) Initial Mesh Structure (b) Mesh Structure of the Solution

Figure 9: EXAMPLE: Mesh Plots; Grid size = 0.3, Cells Numbers = 42

5.4 Other Details

It's really obvious that GPU didn't do a great job on acceleration for our case. So we need to check the Profiling data we get more carefully to find out the reason. There are no advantages to solve a problem with low computional

demands, so we will still focus on those 3 cases with largest cells numbers.

5.4.1 A Matrix

Sometimes the sparseness of a Matrix we are dealing with can effect the running time.

Cells Number	Nonzero elements	Size of Matrix A	Sparseness
5824	12400	966×966	0.01328824
23264	278866	23366×23366	0.0000428
36096	441149	36223×36223	0.00033621

Table 3: CPU and GPU Running Time Comparison with Numba and spsolve: for sparse solver alternatives;

The matrix is very sparse. Obviously, if we only consider the sparseness, it should work well for all sparse solvers. Thus, sparseness is not the reason we are looking for.

5.4.2 Details about Profiling Data

We asked python to show us 10 most cost functions. But there will be only two or three of them account for more than 90 percent of the whole time cost.
$< ipython - input - 541 - 58cdef79a07f >$: $39(< module >)$ is the slowest step, and it cost almost 50 percent time of the whole process. We don't think we can cut this part of the time, because it is the I/O execution at the beginning of the whole code and the only way to make it faster is to change the hardware.
We also try PyTorch on our project, but there are deadly disadvantages of this library: (1) If you want to call a function to complete the multiplication between a sparse matrix and a vector, you have to use a Linux-System Computer or install OpenBLAS (which may be not working for all computers). (2) PyTorch is really unhappy with sparse Matrix and a similar thing happens to TensorFlow. This means you have to transfer your 'Matrix.dtype' back to the 'numpy.ndarray', which will literally cost so much time.
Another thing we want to talk about is that for this particular case, I'm quite skeptical about the stability of Iterative solvers, especially those using the Conjugate Gradient Squared iteration algorithm.
The most efficient way to accelerate the computational process, of course, is to write a Matrix Solver by hand. Additionally, there are some documents talking about GPU Accelerated Linear System Solvers on NVIDIA's website, but most of them are iterative solvers based on Cuda, dealing with a heat conduction and diffusion equation or wave equation with pretty nice diagonal dominance Stiffness matrix, which is completely different from our case. So this can be a Potential Research Question in the Future.

6 Acknowledgements

We want to extend our deepest gratitude to Fredrik Larsson for his supervision and a special thanks to Lars Davidson for his great enthusiasm in managing this course. Also we want to thank Kim Luisa Auth for her Basic Code Framework.

6.1 Contributions

1. Congxiao Zhang:
   Code: Assemble the whole code, Rewrite the Grid generation and Profiling; Write the PyTorch version of code
   Report: Basic Theory; Specific Problem and Results and Discussions.

2. Gayana Jinde Radhakrishna:
   Code: Mesh Plot, Helped with profiling;
   Report: Abstract, Introduction and GPU Acceleration for FEM. Offered some good ideas about the Result Analysis.

3. We parsed the Basic Code Framework together.

6.2 Computer Code

The whole code has been uploaded on GitHub. And we will also attach it as a zip file additionally.