ISSN: 2641-2039
Global Journal of Engineering Sciences
DOI: 10.33552/GJES.2021.08.000691
Iris Publishers

Research Article

CUDA-based Accelerated FEM Calculation

Cankun Zheng, Wenhao Lin and Jiujiang Zhu*

School of Civil Engineering and Architecture, Wuyi University, China

***Corresponding author:** Jiujiang Zhu, School of Civil Engineering and Architecture, Wuyi University, China

Received Date: August 31, 2021

Published Date: September 30, 2021

Abstract

The Finite Element Method (FEM) had been widely used in engineering applications. There are two bottleneck issues in the FEM calculation. The first one is how to minimize computer memory demand so that to be able to run large-scale simulations using limited computational resources. The second one is how to carry out simulation as fast as possible to complete a large-scale simulation job. A standard finite element analysis always results in solving simultaneous linear equations. Overall, a global stiffness matrix of a finite element simulation often has sparse, symmetric, and positive definite characteristics. To overcome the first issue, we proposed integrating and storing stiffness matrix using the Compressed Sparse Row (CSR) format in this paper. We proposed to carry out parallel computing to solve simultaneous equations using the cuBlAS, cuSPARSE, and cuSOLVER libraries on the Compute Unified Device Architecture (CUDA) platform to overcome the second issue. The aim of this article is to explore the possibility to solve simultaneous equations with sparse matrix using CUDA libraries. Three algorithms of cuSOLVER libraries were evaluated through case studies. The three algorithms include the QR factorization, incomplete LU factorization, and the conjugate gradient methods. We have evaluated the efficiency, accuracy and capacity of the algorithms. Two type of case studies were investigated in the numerical experiments. First type of sparse matrix were downloaded from the Florida Sparse Matrix Collection and second type of matrix were assembled from truss structure using FEM. Numerical experiment results showed that, the accuracy of incomplete LU factorization is unacceptable, the capacity of the QR factorization is weak, since the GPU memory demand is higher. The efficiency, capacity and accuracy of the conjugate gradient methods all are satisfied. However, it may break down for some particular cases.

Keywords: FEM; cuSPARSE; cuBLAS; cuSOLVER; Compressed sparse row format matrix; QR Factorization; Incomplete LU Factorization; Conjugate gradient

Introduction

For many large-scale FEM calculations, solving sparse linear equations takes up most of the computing time and resources. Taking the static analysis of structures using the finite element method as an example, the solution of sparse linear equations usually accounts for more than 70% of the whole analysis time [1]. With the continuous expansion of engineering scale and the continuous improvement of engineering complexity, higher requirements are put forward for the scale and speed of solving sparse linear equations. Therefore, the use of high-performance computing is a critical way to solve sparse linear equations.

Nowadays, the direct method and the iterative method are the two most popular methods to solve sparse linear equations. The direct process is to transform the coefficient matrix of the equations into a triangular matrix and then use the back-substitution method or the pursuit method to get the solution of the equations. The iterative approach is a process of constructing an infinite sequence to approximate the exact solution from an approximate initial value of the solution. The rest of the paper is organized as follows: Section 2 gives a brief description of the CUDA platform. We focus on three essential libraries of CUDA Toolkit, cuBLAS, cuSOLVER, and cuSPARSE. Section 3 defines the CSR storage format, which was used in this paper. Section 4 demonstrates how cuBLAS, cuSOLVER and cuSPARSE were implemented in detail and error analysis. Section 5 shows two types of case studies to solve sparse simultaneous equations. In the first type of case study, the matrix was download from the Sparse Matrix Library [2]. The second type of case study is an example to analyze truss structure using FEM. Section 6 is the conclusion.

CUDA Platform

The CUDA platform allows developers to use the C language as a development language and also supports other programming languages or interfaces. Under the CUDA architecture, a program is divided into two parts: the Host side, which is executed serially on the CPU, and the Device side (also known as the kernel), which is implemented in parallel on the graphics card. Usually, the execution model of a complete program is that the host side prepares the data and copies it to the graphics card's memory. The GPU executes the kernel program of the device side for parallel calculation. After the calculation is completed, the result is sent back to the host side from the memory of the graphics card [3].

The cuBLAS library implements BLAS (Basic Linear Algebra Subprograms) on top of the NVIDIA ®CUDA™ runtime [4]. The cuBLAS library is mainly used for matrix operation. It contains two sets of API. One is the commonly used cuBLAS API, which requires users to allocate GPU memory space and fill in data according to the specified format. The other one is cuBLASXT API, which can assign data at the CPU end, then call function. It will automatically manage memory and perform computation [5]. In practice, the first set of APIs is usually used.

The cuSPARSE library contains a set of basic linear algebra subroutines used for handling sparse matrices. The goal of the standard library is a matrix with a certain number of (structurally) zero elements that represent >95% of the total entries. It is implemented on top of the NVIDIA ®CUDA™ runtime (part of CUDA Toolkit). And it is designed to be called from C and C++ [6]. cuSPARSE is a collection of column interfaces. And the main advantage is that it only needs to call the corresponding API when the users write the host code. So, it can save a lot of development time, and this library also supports many data formats.

cuSlOLVER is an advanced library. It is based on the cuBLAS and cuSPARES library and includes matrix factorization and solving equations. It contains three independent library files, including cuSolverDN, cuSolverSP, and cuSolverRF. The cuSolverDN can perform dense matrix factorization (LU, QR, SVD, LDLT), while the cuSolverSP is a new toolset to manipulate sparse matrices, such as to solve simultaneous equations with sparse matrix. The cuSolverRF is a sparse refactoring package [7].

Storage Format

In solving the simultaneous sparse linear equation, the coefficient matrix is sparse. Therefore, the selection of storage format is crucial to reduce the storage of unnecessary zeros, and it is also imperative to improve the performance of its operation. We mainly introduce the compressed sparse row format used in this paper.

CSR format is a mainstream general sparse matrix representation format, which stores column indices and nonzero values in array columns and array values [8]. Eq.(1) is an example of a sparse matrix, which can be stored in CSR format as shown in Eq.(2)

$$\begin{bmatrix} 1.0 & 2.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 3.0 & 4.0 & 0.0 & 0.0 \\ 5.0 & 0.0 & 0.0 & 6.0 & 7.0 \\ 0.0 & 0.0 & 8.0 & 0.0 & 9.0 \end{bmatrix} \tag{1}$$

$$\begin{aligned} csrValA &= [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0] \\ csrRowPtrA &= [0, 2, 4, 7, 9] \\ csrColIndA &= [0, 1, 1, 2, 0, 3, 4, 2, 4] \end{aligned} \tag{2}$$

For a given *n × m* sparse matrix **A** [in Eq. (1) n = 4, m = 5], assume the total number of nonzero elements in the matrix **A** is nnz. The first row of CSR format is called $csrValA$, which is a data array of all nonzero values of the matrix **A** in column-major form, the length of this array is *nnz*. The third row of CSR format is called $csrColIndA$, which is an integer array, and contains the column indices of the corresponding elements in the array $csrValA$; therefore, its length also is *nnz*. The second row of CSR format is called $csrRowPtrA$, which is an integer array in size $n+1$; the $i-\text{th}\left(i=1,2,\cdots n\right)$ element is the column index or the position in the array $csrValA$ of the first nonzero value of $i-\text{th}$ row. The last element of $csrRowPtrA$ is the total number of nonzero elements in the matrix **A**, i.e. *nnz*.

Implementation Method and Relative Error Analysis Based on cuBlAS, cuSPARSE, and cuSOLVER

The solution methods of large sparse linear equations usually include the direct method and the iterative method.

QR Factorization

QR factorization is based on matrix factorization. Based on elimination, the linear equations are transformed into several equivalent sub-problems, which are easy to be calculated and solved in turn. Theoretically, the exact solution of the system of equations can be obtained by the direct method in a fixed number of steps without considering the accumulative numerical error.

The algorithm of QR factorization can be described as follows:

$$\mathbf{A} = \mathbf{Q} \times \mathbf{R} \tag{3}$$

$$\mathbf{X} = \mathbf{R}^{-1} \times \mathbf{Q}^{*} \times \mathbf{b} \tag{4}$$

- Apply for variable memory space on the GPU and transfer *Col, row, Val* and ***b*** from the CPU to the GPU.
- Find the final **X** value by calling cusolverSpDcsrlsvqr.
- Return the **X** value from the GPU side to the CPU side.

Incomplete LU Factorization

Incomplete LU factorization is an iterative solution method, which is related to the LU factorization method. The LU factorization method is a direct solution method. When the matrix **A** is a sparse matrix, the matrix **L** and **U** lose the sparse characteristics of the matrix **A**. The storage cost of the matrix that loses the sparse property will increase, so the incomplete LU factorization method is proposed in [9], so that the iterative matrix **K** in **A** = **K** – **R** satisfies:

K = **L X U** and the matrices **L** and **U** have the sparse property.

The algorithm of incomplete LU factorization can be described as follows:

$$\mathbf{A}=\mathbf{K}-\mathbf{R} \tag{5}$$

$$\mathbf{K}=\mathbf{L}\times\mathbf{U} \tag{6}$$

$$\mathbf{b}=\mathbf{L}\times\mathbf{Z} \tag{7}$$

$$\mathbf{Z}=\mathbf{U}\times\mathbf{X} \tag{8}$$

Operations on sparse matrices stored in CSR format have been provided in the cuSPARSE library. Call the cusparseDcsrilu02 function to decompose the matrix into a unit lower triangular matrix **L** and an upper triangular matrix **U**. Call the cusparseDcsrsv2 _ solve function to find the value of the matrix **Z**, and then repeat the call to find the final value **X**. In this way, the algorithm of incomplete LU factorization is completed by calling the corresponding function through the cuSPARSE library. This article refers to this algorithm as cuSPARSE_LU. The specific algorithm flow is as follows:

- Apply for variable memory space on the GPU and transfer *Col, row, Val* and **b** from the CPU to the GPU.
- By calling cusparseCreateMatDescr, cusparseSetMatIndexBase, cusparseSetMatType, cusparSESetMatFillMode, CusparseSetMatDiagType creates descriptors of matrix **A**, matrix **L**, and matrix **U**, and opens the corresponding memory by calling csru02Info_t and csrsv2Info_t.
- Analyze the LU by calling cusparseDcsru02_analysis, cusparseDcsrsv2_analysis.
- The matrix **K** is decomposed into a lower triangular matrix **L** and an upper triangular matrix **U** by calling cusparseDcsrilu02 and cusparseXcsrilu02_zeroPivot.
- Solve the value of matrix **Z** by calling cusparseDcsrsv2_to solve.
- Find the final **X** value by calling cusparseDcsrsv2_to solve.
- Return the **X** value from the GPU side to the CPU side.

Conjugate gradient method

The conjugate gradient method is one of the most commonly used iterative methods for solving linear equations [10]. The conjugate gradient method uses successive approximation, generally using the iterative formula to get a series of approximate solutions gradually approaching the real solution, and finally get the approximate solution satisfying certain error tolerance.

The conjugate gradient algorithm is described as follows:

1) Set Initial guessed value $\mathbf{X}_0$.

2) Calculation

$$\mathbf{b_1}=\mathbf{b}_0-\mathbf{AX_0},r_1=\mathbf{b}_0^{\mathrm{T}}\mathbf{b_0} \tag{9}$$

3) While $r_1 > tol\times tol\ \&\&k <= \max\_iter$, we have

(1) when K = 1

$$\mathbf{p_1}=\mathbf{b_1};\quad dot=\mathbf{p}^{\mathrm{T}}\mathbf{Ap} \tag{10}$$

$$a=\frac{r_1}{dot} \tag{11}$$

$$\mathbf{x_1}=\mathbf{x_0}+a\times\mathbf{p} \tag{12}$$

$$na=-a;\quad \mathbf{b}=\mathbf{b}+na\times\mathbf{Ax_1} \tag{13}$$

$$r_0=r_1,r_1=\mathbf{b_1^{\mathrm{T}}b_1} \tag{14}$$

(2) when K = 2,3,4,

$$r_k=\frac{r_{k-1}}{r_{k-2}};\quad \mathbf{p_k}=r\times\mathbf{p_{k-1}};\quad \mathbf{p_k}=\mathbf{p_{k-1}}+\mathbf{b_{k-1}} \tag{15}$$

$$dot_k=\mathbf{p_k^{\mathrm{T}}Ap_k} \tag{16}$$

$$a_k=\frac{r_k}{dot_k} \tag{17}$$

$$\mathbf{x_k}=\mathbf{x_{k-1}}+a\times\mathbf{p_k} \tag{18}$$

$$na_k=-a_k;\quad \mathbf{b_k}=\mathbf{b_k}+na\times\mathbf{Ax_k};\quad r_0=r_k \tag{19}$$

$$r_k=\mathbf{b_k^{\mathrm{T}}b_k} \tag{20}$$

The solution of this conjugate gradient method calls the cuBLAS and cuSPARSE libraries. Although each step of the conjugate gradient method is serial, it mainly involves sparse matrix and vector multiplication, vector and vector dot multiplication, vector update, and scalar division in the whole algorithm process. These operations can all involve data-level parallelism, thus enabling GPU parallel computing. That is to say, GPU is responsible for the parallel calculation between matrix and vector, vector and vector before iteration and in each iteration, and CPU is responsible for the control of iteration loop and convergence condition, as well as the operation of scalar multiplication. Operations on sparse matrices stored in CSR format have been provided in the cuSPARSE library. By calling the cusparseSpMV function in the cuSPARSE library, a sparse matrix stored in CSR format can be multiplied by a vector. The operation of two-vector dot multiplication can be realized by calling the cublasDdot function in the cuBLAS library. The vector update is implemented by calling the cublasDaxpy function in the cuBLAS library. In this way, the cuSPARSE library and the cuBLAS library can be used to call the corresponding functions for the conjugate gradient algorithm. This article refers to this algorithm as cuSPARSE_CG. The specific algorithm flow path is as follows:

- Apply memory space for variables on the GPU, and transfer *Col, row, Val* and **b** from the CPU to the GPU. CPU memory variables α, β, α_1, r_0, r_1, a, r, dot, and na are also applied.

- Residual is calculated **b** = **b** – **AX** and **r=b^Tb**by calling cusparseSpMV, cublasDaxpy, and cublasDdot functions of cuSPARSE and cuBLAS libraries.

- $While\, r_1 > tol \times tol \,\&\&\, k <= \max\_iter$

① Call the cublasScopy function of the cuBLAS library to assign **b** to **p** when K = 1; When k=2,3,4, $r = \frac{r_1}{r_0}$, calculation $p \times r$ by calling the cublasDscal function of the cuBLAS library, Call the cublasDaxpy function to calculate $\mathbf{p} = \mathbf{p} + \alpha \times \mathbf{b}$.

② Calculated $\mathbf{Ax} = \mathbf{A} \times \mathbf{p}$ by calling the cusparseSpMV function of the cuSPARSE library.

③ Calculated by $dot = \mathbf{p}^{\mathrm{T}}\mathbf{Ap}, \mathrm{a} = \frac{r_1}{dot}$ calling the cublasDdot function of the cuBLAS library

④ Calculated $\mathbf{x} = \mathbf{x} + a \times \mathbf{p}, na = -a$ by calling the cublasDaxpy function of the cuBLAS library.

⑤ Calculated $\mathbf{b} = \mathbf{b} + na \times \mathbf{Ax}$ by calling the cublasDaxpy function of the cuBLAS library.

⑥ $r_0 = r_1$, At the same time, it is calculated $r = \mathbf{b}^{\mathrm{T}}\mathbf{b}$ by calling the cublasDdot function of the cuBLAS library.

⑦ Return the **X** value from the GPU to the CPU.

Relative error analysis

The relative error is the ratio of the absolute error caused by the measurement to the true value of the measurand (convention) multiplied by 100%, expressed as a percentage. Generally speaking, the relative error can better reflect the credibility of the measurement. If the true value t of the measurand is subtracted from the measurement result y, the resulting or absolute error is Δ. The relative error can be obtained by dividing the absolute error Δ by the conventional true value t. Relative error = absolute error ÷ true value. It is the ratio of the absolute error to the true value (may be expressed as a percentage, parts per thousand, parts per million, but is often described as a percentage) [11].

The algorithm for relative error is described as follows:

$$\mathbf{Ax} = \mathbf{b} \tag{21}$$

$$\boldsymbol{\varepsilon} = \mathbf{Ax} - \mathbf{b} \tag{22}$$

$$\tilde{\boldsymbol{\varepsilon}} = \frac{\|\mathbf{Ax} - \mathbf{b}\|}{\|\mathbf{Ax}\| + \|\mathbf{b}\|} \tag{23}$$

Where the modulus of a vector $a = (a_1, a_2, \cdots a_{n-1}, a_n)^T$ is defined as

$$\|\mathbf{a}\| = \sqrt{\frac{\sum_{i=1}^{n} a_i^2}{n}}, \tag{24}$$

The analysis of the relative error also calls the cuSPARSE library to speed up its calculation. This article refers to this algorithm as cuSPARSE_ Relative error. The specific algorithm flow is as follows:

- Apply memory space for variables on the GPU, and pass *Col, row, Val* and **X** from the CPU to the GPU. CPU memory variables alpha and beta are applied at the same time.
- Call the cusparseSpMV function of the cuSPARSE library to calculate **Ax** and return **Ax** from the GPU to the CPU.
- The variables *sup, sup1, sup2, sum, sum1, sum2, su(su=sum$^{1/2}$), su1* and *su2* and are declared on the CPU side and solved according to the above formula.

Experimental Analysis

Experimental platform

The computing platform of this paper is: The CPU is Intel (R) Core (TM) i9-10900K CPU @ 3.70 GHz; the GPU is NVIDIA RTX 2080 Ti; the operating system is Windows 10 64-bit, and the CUDA platform is version 11.4.

Experimental data

Table 1 lists the data used in the experiment, where Rows is the number of rows in the matrix, Cols are the number of columns in the matrix, Nonzeros is the nonzero element in the matrix, and Nonzeros/Rows is the ratio of nonzero elements to the number of rows.

Table 1: Sparse Matrices for Testing.

Matrix Name	Rows	Nonzeros	Nonzeros/Rows
bcsstk38	8032	355460	44.26
bcsstk25	15,439	252,241	16.34
bcsstk36	23052	1143140	49.59
bcsstk35	30,237	1,450,163	47.96
vanbody	47,072	2,329,056	49.48
nasasrb	54,870	2,677,324	48.79

oilpan	73,752	2,148,558	29.13
apache1	80,800	542,184	6.71
Truss1	100,000	299,998	3.00
Truss2	500,000	1,499,998	3.00
Truss3	1,000,000	2,999,998	3.00
Truss4	5,000,000	14,999,998	3.00

The first eight sparse matrices in Table 1 are downloaded from the Sparse Matrix Library [2], while the last four matrices are integrated from the global stiffness matrix of a truss structure in Figure 1. The following is a brief introduction to the derivation of the global stiffness matrix. From the known hypothesis [12], assume

$$E_1 = E_2 = E_3 = E_4 = \cdots = E_n = 1\text{pa} \tag{25}$$

$$A_1 = A_2 = A_3 = A_4 = \cdots = A_n = 1m^2 \tag{26}$$

$$L_1 = L_2 = L_3 = L_4 = \cdots = L_n = 1m \tag{27}$$

The elements and their linkage are shown in Figure 1.

First, we number the nodes and divide the elements (as shown in Figure 1), and then we calculate the element stiffness matrix equation of each element. For example, the stiffness matrix and balance equation of element ① is:

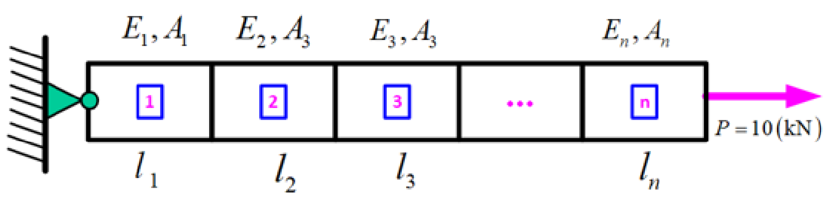

Figure 1: Sketch of connecting rod structure.

$$\begin{bmatrix} \frac{E_1A_1}{l_1} & -\frac{E_1A_1}{l_1} \\ -\frac{E_1A_1}{l_1} & \frac{E_1A_1}{l_1} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} p_1^{(1)} \\ p_2^{(1)} \end{bmatrix} \tag{28}$$

For element ②

$$\begin{bmatrix} \frac{E_2A_2}{l_2} & -\frac{E_2A_2}{l_2} \\ -\frac{E_2A_2}{l_2} & \frac{E_2A_2}{l_2} \end{bmatrix} \begin{bmatrix} u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} p_2^{(2)} \\ p_3^{(2)} \end{bmatrix} \tag{29}$$

For element ③

$$\begin{bmatrix} \frac{E_3A_3}{l_3} & -\frac{E_3A_3}{l_3} \\ -\frac{E_3A_3}{l_3} & \frac{E_3A_3}{l_3} \end{bmatrix} \begin{bmatrix} u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} p_3^{(3)} \\ p_4^{(3)} \end{bmatrix} \tag{30}$$

Then we assemble the stiffness matrix of each element to get

$$\begin{bmatrix} \frac{E_1A_1}{l_1} & -\frac{E_1A_1}{l_1} & & & \\ -\frac{E_1A_1}{l_1} & \frac{E_1A_1}{l_1}+\frac{E_2A_2}{l_2} & -\frac{E_2A_2}{l_2} & & \\ & -\frac{E_2A_2}{l_2} & \ddots & \ddots & \\ & & \ddots & \frac{E_{n-1}A_{n-1}}{l_{n-1}}+\frac{E_nA_n}{l_n} & -\frac{E_nA_n}{l_n} \\ & & & -\frac{E_nA_n}{l_n} & \frac{E_nA_n}{l_n} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \\ u_{n+1} \end{bmatrix} = \begin{bmatrix} p_1^{(1)} \\ p_2^{(1)}+p_2^{(2)} \\ \vdots \\ p_n^{(n-1)}+p_n^{(n)} \\ p_{n+1}^{(n)} \end{bmatrix} \tag{31}$$

Finally, after introducing the external boundary conditions *u₁ = 0*, the overall stiffness matrix is:

$$\begin{bmatrix} \frac{E_1A_1}{l_1}+\frac{E_2A_2}{l_2} & -\frac{E_2A_2}{l_2} & & \\ -\frac{E_2A_2}{l_2} & \ddots & \ddots & \\ & \ddots & \frac{E_{n-1}A_{n-1}}{l_{n-1}}+\frac{E_nA_n}{l_n} & -\frac{E_nA_n}{l_n} \\ & & -\frac{E_nA_n}{l_n} & \frac{E_nA_n}{l_n} \end{bmatrix} \tag{32}$$

The global stiffness matrix after introducing the corresponding boundary conditions is

$$\begin{bmatrix} 2 & -1 & & \\ -1 & \ddots & \ddots & \\ & \ddots & 2 & -1 \\ & & -1 & 1 \end{bmatrix} \quad (33)$$

Comparison and analysis of experimental results

The cuSPARSE library, cuBLAS library, and cuSOLVER library are called by the QR factorization method, incomplete LU factorization, and conjugate gradient method respectively. In order to compare the performance of the three different calculation methods, the relative run time is shown in Figure 2 and the comparison table is shown in Table 2. The relative error analysis comparison chart is shown in Figure 3, and the comparison table is shown in Table 3.

The results show that the computational efficiency of the incomplete LU factorization is two times faster than that of the conjugate gradient method, and the average relative error of the incomplete LU factorization is 4.87508×10^{-1} and that of the conjugate gradient method is 6.12337×10^{-8}. The relative error of the conjugate gradient method is much smaller than that of the incomplete LU factorization. The computational efficiency of QR factorization is similar to that of conjugate gradient, while the relative error of QR factorization 5.13303×10^{-10} is and that of conjugate gradient method is 6.12337×10^{-8}. Both QR factorization and conjugate gradient accuracies are good enough for FEM simulation [13].

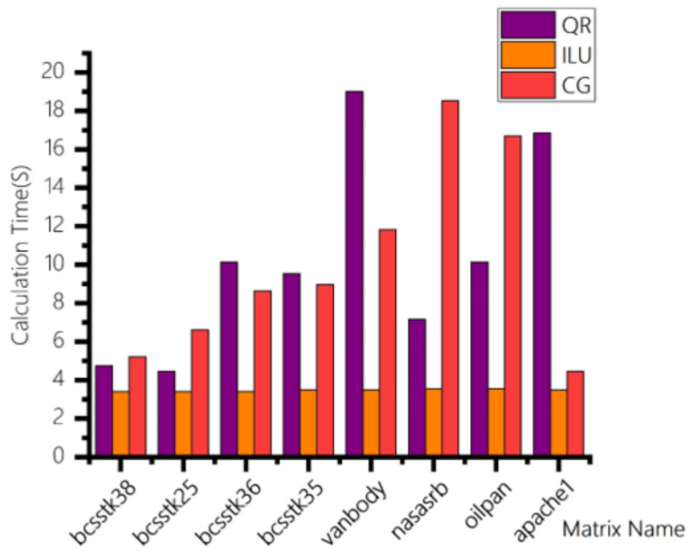

Figure 2: Comparison of calculation time of three algorithms.

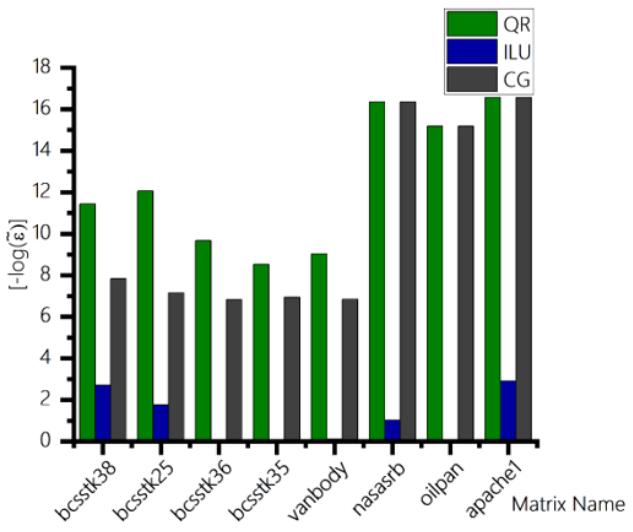

Figure 3: Relative error comparison of three algorithms.

Table 2: Comparison of calculation time of three algorithms.

Matrix Name/Calculation Time	QR Factorization	Incomplete LU Factorization	Conjugate Gradient Method
bcsstk38	4.75	3.407	5.218
bcsstk25	4.469	3.406	6.609
bcsstk36	10.156	3.409	8.641
bcsstk35	9.547	3.5	8.969
vanbody	19.015	3.5	11.844
nasasrb	7.156	3.547	18.531
oilpan	10.156	3.562	16.703
apache1	16.86	3.485	4.468

Table 3: Relative error comparison of three algorithms.

Matrix Name/Relative error	QR Factorization	Incomplete LU Factorization	Conjugate Gradient Method
bcsstk38	3.65556E-12	1.99139E-03	1.44874E-08
bcsstk25	8.69182E-13	1.73717E-02	6.99361E-08
bcsstk36	2.11603E-10	9.99999E-01	1.48439E-07
bcsstk35	2.97549E-09	9.99960E-01	1.14928E-07
vanbody	9.14804E-10	7.84756E-01	1.42079E-07
nasasrb	4.39159E-17	9.47435E-02	4.39166E-17
oilpan	6.46188E-16	1.00000E+00	6.46188E-16
apache1	2.77353E-17	1.24188E-03	2.77353E-17

Comparison of QR Factorization, Incomplete LU Factorization and Conjugate Gradient Method for Solving Truss Structure using FEM

The overall stiffness matrix of the truss elements in this test is simple, the length L, cross-sectional area A and elastic modulus E of the truss are all set as 1, and the relative residual error is set at 10^{-6}. The simple overall stiffness matrix results in that the QR factorization and incomplete LU factorization method have breakneck calculation speeds. For example, to solve 100000 orders matrix equations, the computational time of the conjugate gradient is about 4.07 times more than that of the incomplete LU factorization, while the conjugate gradient computational time is about 2.64 times more than that of the QR factorization. Table 4 compares the computation time of solving the global stiffness matrix of the truss elements using the QR factorization, incomplete LU factorization, and the conjugate gradient method. The relative error analysis and comparison are shown in Table 5. The accuracies of both QR factorization and incomplete LU factorization are satisfied. For large-scale simulation, the conjugate gradient method was not convergent, i.e., the iteration steps are less than the total freedom of the system. For this particular structure, the conjugate gradient method only convergent when iteration steps equal the total freedom of the system. If the iteration steps equal the total freedom of the system, iteration solution equals exact solution.

Table 4: Comparison of calculation time of three algorithms for truss structure.

Matrix Name/Calculation Time(S)	QR Factorization	Incomplete LU Factorization	Conjugate Gradient Method
Truss1	5.172	3.718	18.88
Truss2	10.547	5.062	45.641
Truss3	17.25	6.719	75.578
Truss4	70.922	20.125	311.25

Table 5: Relative error comparison of three algorithms for truss structure.

Matrix Name/Relative error	QR Factorization	Incomplete LU Factorization	Conjugate Gradient Method
Truss1	1.1254E-05	3.4928E-06	0.00E+00
Truss2	2.4842E-05	2.1243E-05	Non-convergence
Truss3	3.5290E-05	3.2917E-05	Non-convergence
Truss4	1.0091E-04	7.9300E-05	Non-convergence

Reason analysis: Although the computational efficiency of the conjugate gradient method and QR factorization in the Florida sparse library is slower than the incomplete LU factorization, the relative error of QR factorization and the conjugate gradient is much smaller than the incomplete LU factorization. Due to in incomplete LU factorization, some elements of the matrix **R** in Eq. (5) were dopped off; this causes incomplete LU factorization to be just an approximate method. Our testing results demonstrate that the accuracy of incomplete LU factorization is unacceptable for accurate engineering simulation. In practice, usually, incomplete LU factorization is only suitable to be used as a preprocessing. Although both efficiency and accuracy of QR factorization are satisfied, our numerical experiments show that the GPU memory consumption of QR factorization is much higher than the other two methods. It means QR factorization is not suitable for large scall simulation, at least for some cheap GPUs.

Our testing shows that, in most cases, the efficiency, accuracy, and capacity of conjugate gradient all are good enough for FEM simulation; however conjugate gradient may break down for some particular cases. For example, in our truss structure FEM simulation, if the iteration step is less than the total freedoms of the system, the residual error keeps as constant without change. If and only if when the iteration step equals the freedoms of the system, the residual error becomes zero, i.e., the iteration solution reaches the exact solution. For large-scale simulation, it is not feasible to make iteration steps equal the freedoms of the system. It needs a very long iteration time.

Conclusion and Future Work

In this paper, we have compared the computational efficiency, accuracy, and capability of three algorithms: QR factorization, incomplete LU factorization, and conjugate gradient method using cuBlAS, cuSPARSE, and cuSOLVER libraries. We have reached the following conclusions:

- In terms of computational efficiency, the incomplete LU factorization is the fastest, and the conjugate gradient is the slowest. The running speed of the incomplete LU factorization is 11.6 times faster than that of the conjugate gradient. The running speed of the QR factorization is 3.3 times faster than that of the conjugate gradient.
- In terms of computational accuracy, both the QR factorization and the conjugate gradient methods are good enough for accurate engineering simulation. The incomplete LU factorization method is unacceptable.
- In terms of computational capacity, both the incomplete LU factorization and the conjugate gradient methods are acceptable. The QR factorization method requires too much GPU memory, so it is not suitable for large-scale simulation.

To compare all three algorithms, we will find the conjugate gradient method is the best candidate for large-scale FEM simulation. However, the conjugate gradient method may break down for some particular cases. We proposed to combine the incomplete LU factorization and the conjugate gradient method in our future work. To use incomplete LU factorization as a preprocessing to carry out the precondition conjugate gradient simulation.

Acknowledgment

This research is supported by the project of “High Education funding of Guangdong Province: 2018KZDXM072”; the College Students’ Innovation and Entrepreneurship Project: 503170060206; the Natural Science Foundation of Guangdong Province: 2021A1515010704.

Conflict of Interest

No conflict of interest.

References

1. Zhang J, D Shen (2013) GPU-based preconditioned conjugate gradient method for solving sparse linear systems. Journal of Computer Applications 33(3): 825-829.
2. DAVIS T, Yifan Hu (2012) The University of Florida Sparse Matrix Collection. Acm Transactions on Mathematical Software.
3. Nvidia C (2021) NVIDIA CUDA c programming guide.
4. NVIDIA (2021) CUDA CUBLAS Library.
5. Storti D, M Yurtoglu (2015) CUDA for Engineers: An Introduction to High-Performance Parallel Computing. 2015: CUDA for Engineers: An Introduction to High-Performance Parallel Computing.
6. NVIDIA (2021) CUDA CUSPARSE Library.
7. NVIDIA (2021) Cusolver library.
8. Yang SE, JIANG Guoping, SONG Yurong, TU Xiao (2019) Research on Storage Format Optimization of Sparse Matrix Based on GPU. Computer Engineering 45(9): 23-31,39.
9. Meijerink JA, HAvdVJMo (1977) Computation, An iterative solution method for linear systems of which the coefficient matrix is a symmetric matrix. 31(137): 148-162.
10. Yang B, Tai Bao, Hong Wang (2013) FEM-Based Modeling and Deformation of Soft Tissue Accelerated by CUSPARSE and CUBLAS. Advanced Materials Research 671-674(3): 3200-3203.
11. Huijuan Y (2015) Analysis of the uncertainty of relative indication error u (e _ R) and the relative uncertainty of indication error u _ R (e). China Metrology 000(007): 88-89.
12. Pan Z (2004) Finite element analysis and application (CD attached). Finite element analysis and application (CD attached).
13. Zhu, et al. (2009) Preconditioned BiCGSTAB algorithm and its application to eddy current problem. 3: 78-82.