GPU-Based Parallel Solution of Large-Scale Finite Element Sparse Matrices: Postprint

Authors: Zhou Qinglong, Lin Wancang

Date: 2025-07-30T09:54:03+00:00

Abstract

Currently, high-performance computing for large-scale finite elements primarily relies on parallel computing using multi-CPU computer clusters, which incurs high computational costs and offers limited improvements in computational efficiency. Leveraging the advantages of advanced GPU chips in large-scale data parallel processing, this paper proposes a GPU-based parallel solution method for large-scale finite element sparse matrices, encompassing compressed storage encoding and decoding of large-scale sparse matrices, preconditioned parallel iterative solution strategies, and the development of GPU kernel functions for the main solution steps. A GPU parallel computing finite element program was developed using C++, and solution experiments were conducted on finite element sparse matrices of various scales. Numerical results demonstrate that the computational efficiency of GPU-based parallel solving is significantly enhanced, with the advantage becoming more pronounced as matrix dimensions increase. Furthermore, preconditioning the global sparse matrix during GPU solution substantially accelerates convergence, while maintaining relatively high computational accuracy.

Full Text

Parallel Computation of Large-Scale Finite Element Sparse Matrix with GPU

ZHOU Qinglong, LIN Wancang
(1. School of Resources and Safety Engineering, Central South University, Changsha, Hunan 410083, China)

Abstract: Currently, high-performance computing for large-scale finite elements primarily relies on multi-CPU computer cluster parallel computing, which entails high computational costs yet offers limited efficiency improvements. Leveraging the advantages of internationally advanced GPU chips in large-scale

data parallel processing, this paper proposes a GPU-based parallel solution method for large-scale finite element sparse matrices, encompassing compression storage encoding and decoding for large-scale sparse matrices, preconditioned parallel iterative solution strategies, and GPU kernel function implementation methods for key solution steps. A GPU parallel finite element program was developed using C++, and numerical experiments were conducted on finite element sparse matrices of varying scales. The numerical examples demonstrate significantly improved computational efficiency through GPU-side parallel solving, with greater advantages observed for larger matrix dimensions. Additionally, preconditioning the global sparse matrix during GPU solution substantially accelerates convergence while maintaining relatively high computational accuracy.

Key words: parallel computing; finite element method; GPU computing; sparse matrix

1 Introduction

Large-scale geotechnical engineering numerical calculations currently face several fundamental challenges. First, geotechnical projects operate at massive scales, with computational domains spanning thousands or even tens of thousands of meters. Due to limitations in computer memory and computational efficiency, conventional commercial software requires mesh discretization at the scale of hundreds or thousands of meters. Such coarse meshes fail to capture local deformation and failure characteristics, yet engineering disasters typically initiate from local regions [1], rendering the computational results of limited practical guidance value. Second, many major geotechnical projects in China are situated under complex geological conditions, significantly influenced by faults, fractures, topography, hydrogeology, in-situ stress conditions, and rock properties. These projects also incorporate numerous reinforcement measures such as rock bolts and anchors. However, current computational efficiency limitations force substantial model simplifications or reduced computational domains, neglecting many important influencing factors and reinforcement effects. Consequently, the obtained results struggle to objectively reflect the actual disaster initiation and evolution processes. Third, the disaster incubation and occurrence processes may represent nonlinear, complex multi-field coupling phenomena whose simulation involves ultra-long timescale iterative solutions across multiple physical fields, sometimes requiring years or decades of simulation time. Such problems demand millions to hundreds of millions of iterations, which is impossible to accomplish using current commercial software capabilities.

To address these numerical challenges in large-scale geotechnical engineering, numerous scholars have conducted pioneering research. Some researchers [2-4] proposed parallel computation methods based on domain decomposition strategies, distributing computational information across different computers for divide-and-conquer processing. Wang et al. [5] employed ParMetis software for parallel pre- and post-processing and utilized preconditioning techniques for parallel so-

lution of finite element linear equations, achieving parallel simulation of tunnel excavation processes. Ni et al. [6, 7] implemented master-slave distributed parallel improvements to optimization algorithms using computer clusters, proposing a parallel optimization inversion method for large-scale underground geotechnical engineering. References [8-10] presented finite element EBE (Element-By-Element) parallel algorithms for distributed memory parallel machines, adopting on-demand collection and exchange strategies for multi-CPU data storage and exchange to reduce data transfer and storage requirements. Xie et al. [11] ran their independently developed RFPA3D-Parallel program on computer clusters for parallel computational analysis of three-dimensional fracture processes in rock specimens containing voids. Reference [12] conducted dynamic analysis of seismic response for metro tunnel and civil air defense basement interaction based on the Abaqus software 64CPU explicit parallel computing cluster platform.

Compared with CPU parallel computing, GPU parallel computing offers substantial advantages in efficiency and cost. Currently, extensive research exists on GPU-based sparse matrix computations. Bolz et al. [13] implemented GPU-accelerated conjugate gradient algorithms. Göddeke et al. [14, 15] investigated GPU-based multigrid methods. Naumov [16] demonstrated how to implement IC/ILU preconditioned conjugate gradient and biconjugate gradient stabilized algorithms using CUSPARSE and CUBLAS libraries. Chen et al. [17] proposed using incomplete Cholesky decomposition preconditioned conjugate gradient methods for solving large-scale sparse symmetric positive definite linear systems. Zhang et al. [18] explored optimization methods for sparse matrix-vector multiplication using the CUSPARSE library.

This paper systematically addresses the large equation system solution problem in large-scale geotechnical engineering finite element parallel computing, proposing corresponding storage methods for large-scale finite element sparse matrices, GPU parallel iterative solution methods, and developing finite element GPU parallel computing programs using C++ for systematic evaluation of GPU parallel solution efficiency and accuracy through numerical experiments.

2 Preconditioned Iterative Solution of Large-Scale Sparse Matrices

Standard conjugate gradient iteration converges rapidly for well-conditioned matrices but may converge slowly for matrices with large condition numbers. Preconditioning techniques can substantially improve convergence rates.

The fundamental idea of preconditioning involves applying the standard conjugate gradient method to a transformed system equation. Assuming the transformed equation is:

$$\tilde{\mathbf{A}}\tilde{\mathbf{x}} = \tilde{\mathbf{b}} \quad (1)$$

where $\tilde{\mathbf{A}} = \mathbf{C}^{-1}\mathbf{A}\mathbf{C}^{-1}$, $\tilde{\mathbf{x}} = \mathbf{C}\mathbf{x}$, $\tilde{\mathbf{b}} = \mathbf{C}^{-1}\mathbf{b}$, with $\mathbf{C}$ being a symmetric positive-definite matrix. A proper matrix $\mathbf{C}$ must be selected to reduce the condition

number of $\tilde{\mathbf{A}}$. Applying the standard conjugate gradient method with initial solution $\tilde{\mathbf{x}}_0$ and setting $\tilde{\mathbf{p}}_0 = \tilde{\mathbf{r}}_0$, the preconditioned iterative solution process for $k = 0, 1, 2, 3, ...$ until convergence is:

$$\tilde{\mathbf{x}}_{k+1} = \tilde{\mathbf{x}}_k + \alpha_k \tilde{\mathbf{p}}_k \quad (3)$$

$$\tilde{\mathbf{r}}_{k+1} = \tilde{\mathbf{r}}_k - \alpha_k \mathbf{C}^{-1}\mathbf{A}\mathbf{C}^{-1}\tilde{\mathbf{p}}_k \quad (6)$$

$$\tilde{\mathbf{p}}_{k+1} = \tilde{\mathbf{r}}_{k+1} + \beta_k \tilde{\mathbf{p}}_k \quad (5)$$

Currently, numerous mature matrix preconditioning methods exist, including Jacobi preconditioning, block Jacobi preconditioning, and incomplete factorization preconditioning.

For the finite element equation system $\mathbf{Ax} = \mathbf{b}$, performing incomplete Cholesky decomposition on matrix $\mathbf{A}$ yields:

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T - \mathbf{R} \quad (7)$$

where $\mathbf{L}$ is a lower triangular matrix. Considering $\mathbf{C} = \mathbf{L}\mathbf{L}^T$ as the preconditioning matrix, the following relationship holds:

$$(\mathbf{L}\mathbf{L}^T)^{-1}\mathbf{A}(\mathbf{L}\mathbf{L}^T)^{-1} \cdot \mathbf{L}\mathbf{L}^T\mathbf{x} = (\mathbf{L}\mathbf{L}^T)^{-1}\mathbf{b} \quad (8)$$

According to $(\mathbf{L}\mathbf{L}^T)^{-1} = (\mathbf{L}^T)^{-1}(\mathbf{L})^{-1}$, we can further derive:

$$\mathbf{L}^{-1}\mathbf{A}\mathbf{x} = \mathbf{L}^{-1}\mathbf{b} \quad (9)$$

$$\mathbf{L}^{-1}\mathbf{A}(\mathbf{L}^T)^{-1}\mathbf{L}^T\mathbf{x} = \mathbf{L}^{-1}\mathbf{b} \quad (10)$$

$$\tilde{\mathbf{A}} = \mathbf{L}^{-1}\mathbf{A}(\mathbf{L}^T)^{-1}, \quad \tilde{\mathbf{x}} = \mathbf{L}^T\mathbf{x}, \quad \tilde{\mathbf{b}} = \mathbf{L}^{-1}\mathbf{b} \quad (11)$$

3.1 General Steps for CUDA-Based Large-Scale Sparse Matrix Parallel Solution

The general procedure for solving large-scale sparse matrices using CUDA involves the following steps:

1. Initialize the cuSPARSE and cuBLAS libraries required for computations in the CUDA system.
2. Allocate appropriate memory sizes for pointers and arrays used in the program and initialize them to zero.
3. Ensure that sparse matrix values, row indices, and column index parameters have been computed and exist in device (GPU) memory.
4. Convert data to standard cuSPARSE library formats for function interface compatibility using conversion functions `cusparseCreateCsr` (create CSR compressed matrix) and `cusparseCreateDnVec` (create vector).

5. Transfer specific sparse matrix parameters from host memory to device memory using the `cudaMemcpy` function (CPU-to-GPU data transfer function). For CSR-compressed matrices [19], transferred parameters include matrix dimensions, non-zero values, and row/column indices of non-zero elements.
6. Develop GPU kernel functions for vector multiplication ($\mathbf{z} = \mathbf{xy}$), sparse matrix-vector multiplication ($\mathbf{y} = \mathbf{Ax}$), and scalar multiplication with vector addition ($\alpha\mathbf{x} + \mathbf{y}$).
7. Compute the initial residual $\mathbf{r} = \mathbf{b} - \mathbf{Ax}$ using the developed GPU kernels (including sparse matrix-vector multiplication and scalar multiplication with vector addition).
8. Perform iterative calculations of equations (2)-(6) using the developed kernels (including vector inner products, sparse matrix-vector multiplication, and scalar multiplication with vector addition) until the residual converges to the required tolerance, then terminate and obtain the final solution.

3.2 GPU-Side Parallel Multi-Thread Kernel Function Design

The preconditioned sparse matrix iterative solution process involves three primary operations: vector multiplication ($\mathbf{z} = \mathbf{xy}$), scalar multiplication with vector addition ($\mathbf{y} = \alpha\mathbf{x} + \mathbf{y}$), and sparse matrix-vector multiplication ($\mathbf{y} = \mathbf{Ax}$). These operations consume substantial memory and time, necessitating GPU-side kernel design for parallel computation.

(1) GPU Kernel Function Design for Vector Multiplication

The GPU device-side kernel function for vector multiplication ($\mathbf{z} = \mathbf{xy}$) can be designed as follows:

Algorithm 2: GPU Kernel Function Design for Vector Multiplication

```
__{global}_ void dot(float *x, float *y, float *z) {
    int tid = threadIdx.x + blockIdx.x * numThread;
    while (tid < N) {
        z[tid] = x[tid] * y[tid];
        tid += numThread * numBlock; // Advance by total threads per grid
    }
}
cudaMemcpy(z, dev_z, N*sizeof(float), cudaMemcpyDeviceToHost);
sum = 0;
for (int i = 0; i < N; i++) {
    sum += z[i];
}
```

(2) GPU Kernel Function Design for Scalar Multiplication and Vector Addition

Assuming n is the length of vectors $\mathbf{x}$ and $\mathbf{y}$, and α is the scalar parameter in the operation, designing each thread to compute one element of vector $\mathbf{x}$ multiplied by scalar α and added to the corresponding element of vector $\mathbf{y}$ yields the following simple GPU parallel kernel function for scalar multiplication and vector addition ($\mathbf{y} = \alpha\mathbf{x} + \mathbf{y}$):

Algorithm 3: GPU Kernel Function Design for Scalar Multiplication and Vector Addition

```
__{global}_ void saxpy(int n, float alpha, float *x, float *y) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        y[i] = alpha * x[i] + y[i];
    }
}
```

(3) GPU Kernel for Sparse Matrix-Vector Multiplication Based on CSR Storage

In sparse matrix iterative solutions, matrix-vector multiplication appears repeatedly. CPU-side computation consumes substantial time and memory, making efficient GPU-side parallel computation essential for large-scale sparse matrix solving. Under the CUDA framework, three different kernel programming approaches can implement high-performance parallel computation for CSR storage format matrix-vector multiplication. The first method assigns each thread to compute one element of the output vector, i.e., each thread is responsible for computing the product of non-zero elements in each matrix row with the vector and summing them [20], as detailed in Algorithm 4. The second approach utilizes each warp to compute one output vector element, where threads within a warp compute products of non-zero elements per matrix row with the vector, then perform warp-level reduction [21]. The third method assigns each block to compute one output vector element, followed by block-level reduction of intermediate results. Assuming matrix $\mathbf{A}$ in CSR format has three vectors: `data` (non-zero values), `col` (column indices), and `rowptr` (row offsets), one possible GPU kernel implementation for $\mathbf{y} = \mathbf{A}\mathbf{x}$ is:

Algorithm 4: GPU Kernel Design for Sparse Matrix-Vector Multiplication Based on CSR Storage

```
__{global}_ void spmv_{{{scalar}}_{{kernel}}}(const int num_{rows},
                                    const int *rowptr,
                                    const int *col,
                                    const float *data,
                                    const float *x,
                                    float *y) {
    int row = blockDim.x * blockIdx.x + threadIdx.x;
    if (row < num_{rows}) {
        float dot = 0;
        int row_{start} = rowptr[row];
```

```
        int row_{end} = rowptr[row + 1];
        for (int jj = row_{start}; jj < row_{end}; jj++)
            dot += data[jj] * x[col[jj]];
        y[row] += dot;
    }
}
```

4 Numerical Experiments

4.1 Experimental Platform and Model

This study employs a conventional laboratory desktop computer for experiments. The CPU specifications are: Intel(R) Core(TM) i7-8700 @ 3.20 GHz with 8192 MB total memory. The GPU is an NVIDIA GeForce GTX 1050 Ti with DAC type Integrated RAMDAC, 8026 MB total memory, 4019 MB dedicated memory, and 4007 MB shared memory. According to NVIDIA's computational capability calculation method, the GPU capability is 6.1. The system is Windows 7, 64-bit; the experimental environment is Visual Studio 2019 with CUDA 12.4.

The numerical experiment model considers a two-dimensional soil site with height and width of 100 m. The model parameters include an elastic modulus of 10 MN/m^2, Poisson's ratio of 0.3, and a top-loading condition of 1 kN. The computational domain is discretized using four-node rectangular elements. The domain is meshed with different element sizes and node counts: 18, 800, 1800, 80000, 180000, and 8000000 nodes (see [Figure 1: see original paper]). Each node has two degrees of freedom (displacements in x and y directions). Consequently, the dimensions of the global stiffness sparse matrix are: 36 $\times$ 36, 1600 $\times$ 1600, 3600 $\times$ 3600, 160000 $\times$ 160000, 360000 $\times$ 360000, and 16000000 $\times$ 16000000. By discretizing the computational model with different mesh sizes, finite element global sparse matrices of various scales were designed.

4.2 Experimental Results and Analysis

Using the sparse matrix conjugate gradient solution method described in Section 2, we developed iterative solution programs running on both CPU and GPU sides to solve the global sparse matrices of the computational model. The CPU program is serial, while the GPU program is CUDA-based parallel.

With a convergence tolerance of 1×10^{-12}, the CPU and GPU runtimes under different mesh sizes and matrix dimensions are presented in . The results indicate that for small- to medium-scale matrices, CPU serial iterative solving is even more efficient than GPU parallel solving. At matrix orders of 36 and 1600, CPU runtime outperforms GPU, while at order 3600, CPU and GPU efficiencies are comparable. This phenomenon primarily occurs because GPU solving requires initial data transfer from CPU to GPU, incurring additional overhead. For large-scale global sparse matrices (dimensions in the hundreds of thousands

to tens of millions), GPU parallel solving demonstrates vastly superior efficiency, with advantages becoming more pronounced as matrix dimensions increase. For instance, at a matrix dimension of 160000, the GPU parallel solving achieves a speedup of 18.1×; at 360000, 23.7×; and at the ten-million scale (16000000), GPU solving efficiency reaches 176.7× that of CPU.

To verify the validity of our numerical results, we generated finite element meshes identical to those in reference [22] and conducted numerical experiments on both CPU and GPU. The detailed comparisons are shown in . Compared with reference [22], our computational efficiency shows significant improvement, partially attributable to our newer CPU and GPU versions with hardware advantages. However, the overall pattern remains consistent: for low-dimensional sparse matrix solving, CPU-side efficiency is higher, while GPU-side acceleration increases exponentially with matrix dimension.

Regarding whether preconditioning improves finite element sparse matrix solving, we conducted numerical experiments on GPU by performing incomplete Cholesky decomposition preconditioning on the global stiffness matrix before iterative solving. The results are presented in . Under the specified tolerance (1×10^{-12}), preconditioning substantially reduces convergence steps. For example, at matrix order 160000, convergence requires 1742 steps without preconditioning versus 545 steps with preconditioning.

[Figure 2: see original paper] illustrates the residual reduction during the first 200 iteration steps for matrix dimension 160000×160000 with and without preconditioning. The preconditioned residual decreases significantly faster, reaching 5.35×10^{-5} at iteration 200, compared to 1.21×10^{-1} for the standard conjugate gradient method. These results demonstrate that preconditioning finite element global sparse matrices substantially improves convergence and computational efficiency.

With a convergence tolerance of 1×10^{-4}, we analyzed the final computational errors for preconditioned matrices solved on GPU, as shown in and [Figure 3: see original paper]. The results indicate that final errors exhibit some randomness without direct correlation to matrix dimension, but overall, preconditioned solutions yield smaller errors and higher accuracy compared to non-preconditioned solutions.

Compared with open-source sparse matrix solvers provided in the CUDA platform, our optimized preconditioned parallel solving method significantly reduces matrix condition numbers, thereby decreasing numerical errors during solution. Additionally, preconditioning accelerates algorithm convergence, meaning fewer iteration steps and reduced accumulated numerical errors.

5 Conclusions

This paper proposes a GPU-based parallel solution method for large-scale finite element sparse matrices in geotechnical engineering, detailing storage methods

for large-scale sparse matrices, preconditioned optimization solution methods, and corresponding GPU kernel function implementation approaches. Numerical experiments were conducted on an NVIDIA GeForce GTX 1050 Ti graphics card in a conventional desktop computer, yielding the following conclusions:

1. For large-scale finite element sparse matrices, GPU-side parallel solving significantly improves computational efficiency compared to CPU-side serial solving, with greater advantages for larger matrix dimensions.
2. Preconditioning large-scale finite element global sparse matrices during GPU parallel solving substantially accelerates matrix solution convergence and improves computational efficiency.
3. Preconditioning large-scale finite element global sparse matrices during GPU parallel solving yields relatively smaller computational errors and higher solution accuracy.

References

[12] Mao Kunming, Zhao Kai, Zhu Liming, et al. Seismic response analysis of metro tunnel and civil air defense basement interaction [J]. Journal of Vibration and Shock, 2019, 38(05), 243-250.

[13] Bolz, J., Farmer, I., Grinspun, E. Sparse matrix solvers on the GPU: conjugate gradients and multigrid [J]. ACM transactions on graphics (TOG), 2003, 22(3), 917-924.

[14] Göddeke, D., Strzodka, R., Turek, S. Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations [J]. International Journal of Parallel, Emergent and Distributed Systems, 2007, 22(4), 221-256.

[1] Zhang Youliang, Tan Fei, Zhang Liren, et al. Scalable parallel computing for hundred-million-element finite element models in geotechnical engineering [J]. Rock and Soil Mechanics, 2016, 37(11), 3309-3317.

[2] Zhang Youliang, Feng Xiating. Parallel computation of finite element with over one million degrees of freedom for geotechnical engineering [J]. Rock and Soil Mechanics, 2007, 28(4), 684-688.

[15] Goddeke, D., Buijssen, S. H., Wobker, H. GPU acceleration of an unmodified parallel finite element Navier-Stokes solver [C]. In 2009 International Conference on High Performance Computing & Simulation, 2009; IEEE: pp 12-21.

[3] Ru Zhongliang, Feng Xiating, Zhang Youliang, et al. Parallel computation of finite element analysis of anchored rock mass in underground engineering [J]. Chinese Journal of Rock Mechanics and Engineering, 2005, 24(1), 13-13.

[16] Naumov, M. Incomplete-LU and Cholesky preconditioned iterative methods using CUSPARSE and CUBLAS. Nvidia white paper 3, 2011.

[4] Ru Zhongliang, Feng Xiating, Li Hongdong, et al. 3D elastoplastic parallel

finite element analysis of large underground engineering [J]. Chinese Journal of Rock Mechanics and Engineering, 2006, (06), 1141-1148.

[17] Chen Yao, Zhao Yonghua, Zhao Wei, et al. GPU-accelerated incomplete Cholesky decomposition preconditioned conjugate gradient method [J]. Journal of Computer Research and Development, 2015, 52(4), 843-852.

[5] Wang Xiaorui, Zhang Zhen, Jia Xiaofeng. Numerical simulation of tunnel excavation based on high-performance parallel computing [J]. Earth Science: Journal of China University of Geosciences, 2015, (12), 2119-2127.

[18] Zhang Jianfei, Shen Defei. Preconditioned conjugate gradient method for sparse linear systems based on GPU [J]. Computer Applications, 2013, 33(3), 825-829.

[6] Ni Shaohu, Xiao Ming, He Shihai, et al. Parallel optimization back analysis and case verification of underground engineering [J]. Chinese Journal of Rock Mechanics and Engineering, 2013, 32(3), 501-511.

[19] Buatois, L., Caumon, G., Levy, B. Concurrent number cruncher: a GPU implementation of a general sparse linear solver [J]. International Journal of Parallel, Emergent and Distributed Systems, 2009, 24(3), 205-223.

[7] Ni Shaohu, Xiao Ming. Displacement back analysis of underground engineering parameters based on surrounding rock loosening zone [J]. Chinese Journal of Rock Mechanics and Engineering, 2009, 28(7), 1439-1446.

[20] Bell, N., Garland, M. Implementing sparse matrix-vector multiplication on throughput-oriented processors [J]. Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, 2009, 1-11.

[8] Liu Yaoru, Zhou Weiyuan, Yang Qiang. Finite element parallel EBE method and its application [J]. Chinese Journal of Rock Mechanics and Engineering, 2005, 24(17), 3023-3028.

[21] Filippone, S., Cardellini, V., Barbieri, D. Sparse Matrix-Vector Multiplication on GPGPUs [J]. ACM Trans. Math. Softw., 2017, 43(4), Article 30.

[9] Bova, S., Carey, G. A distributed memory parallel element-by-element scheme for semiconductor device simulation [J]. Computer methods in applied mechanics and engineering, 2000, 181(4), 403-423.

[22] Zheng Jingwei, An Xuehui, Huang Miansong. Optimization of PCG algorithm for large-scale sparse matrices based on CUDA [J]. Journal of Tsinghua University: Science and Technology, 2014, (8), 1003-1010.

[10] Khan, A., Topping, B. Parallel finite element analysis using Jacobi-conditioned conjugate gradient algorithm [J]. Advances in Engineering Software, 1996, 25(2-3), 309-319.

[11] Xie Linmao, Zhu Wancheng, Wang Shuhong, et al. Parallel computational analysis of three-dimensional fracture process of rock specimens with voids [J].

Chinese Journal of Geotechnical Engineering, 2011, 33(09), 1447-1453.