Scaling the Finite Element Method to Small Problems with Dense Linear Solvers

NEW YORK UNIVERSITY

Michael Paciullo, Daniele Panozzo, Teseo Schneider

Geometric Computing Lab, Courant Institute of Mathematical Sciences, New York University

Abstract and Rationale

The finite element method (FEM) is a numerical algorithm for solving boundary-value partial differential equations by splitting the domain into basis elements represented by a mesh, solving the equation for each element, and interpolating. FEM uses linear algebra, typically accelerated by GPUs due to their parallel processing ability.

Due to the large number of elements and relatively low number of inter-element interactions, software implementing the FEM normally uses sparse matrix data structures, which use less memory and perform operations faster on large matrices containing mostly zeros. However, since GPUs are optimized for dense matrix multiplication, using a sparse matrix data structure may be slower than typical dense matrices for problems of small size due to the added overhead and non-uniform element storage.

With this experiment, I tested assembling and solving matrix equations representing Laplace's equation, which is used to calculate optimal mesh deformations with As Rigid As Possible (ARAP) energy, using dense matrix data structures. The lost time from allocating and zeroing space for a dense matrix offsets the gain in solving time.

Introduction to the finite element method

The finite element method represents the domain and solution of the equation as a linear combination of basis elements, which are usually piecewise linear functions. Solving a partial differential equation with the FEM involves finding a weak formulation for the equation, discretizing over the domain of the equation, setting boundary conditions, and solving the resulting matrix equation to find the coefficients of the linear combination representing the solution. Below is an example of using the FEM for Laplace's equation, which is used in the following experiment.

Laplace's equation is the partial differential equation:

$$-\Delta f = \nabla \cdot \nabla f = 0$$

A weak formulation of a partial differential equation finds "weak solutions", or solutions where the derivatives may not exist at all points of the domain, by multiplying by an arbitrary smooth test function v and integrating.

A weak formulation of Laplace's equation is:

$$-\int_\Omega (\nabla \cdot \nabla f)v = -\left(fv|_{\delta\Omega} - \int_\Omega \nabla f \cdot \nabla v\right) = \int_\Omega \nabla f \cdot \nabla v$$

.

The function f is approximated by a discrete basis of piecewise linear functions, in the form:

$$f(x) = \sum_{i=1}^{n} f_i \phi_i(x)$$

.

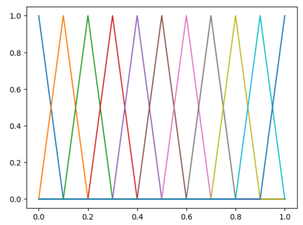

Figure 1. An example of piecewise linear basis functions.

Discretizing v with the same basis as well, the weak form can be rewritten as:

$$\int_\Omega \sum_{i=1}^{n} f_i \phi_i(x)\phi_j(x) = \sum_{i=1}^{n} f_i \int_\Omega \phi_i(x)\phi_j(x), \forall j \in \{1, \ldots, n\}$$

This is a matrix equation $Lu = 0$, which can be solved by integrating the basis functions to find the matrix values and solving. Because most of the products of the functions are 0 everywhere, L is a sparse matrix.

Discrete Laplacian operator

The Laplacian operator above is only defined for continuous functions. For a discrete mesh $M = (V, E)$ for its vertices and edges, an approximation to the Laplacian matrix $C \in \mathbb{R}^{|V|\times|V|}$ is defined as follows:

$$C_{ij} = \begin{cases} \frac{1}{2}(\cot\alpha_{ij} + \cot\beta_{ij}) & ij \text{ is an edge} \\ -\sum_{k\in\{ik \text{ is an edge}\}} C_{ik} & i = j \\ 0 & \text{otherwise} \end{cases}$$

Like the continuous Laplacian operator, this is assumed to be a sparse matrix because most vertices in a 3D mesh are not directly connected via an edge.

Dense and sparse matrix storage formats

Matrices in C++ are usually stored with row-first indexing. For the matrix

$$A = \begin{bmatrix} 1&0&0&3&0&0 \\ 0&1&0&0&0&2 \\ 0&2&1&0&0&0 \\ 0&0&0&1&0&0 \\ 0&0&0&0&1&0 \\ 0&0&0&0&0&1 \end{bmatrix}$$

the value `A[2][3]` would be accessed as $(((A+2)*)+3)* = 0$. This requires storing at least 36 integers for the above matrix. For matrices with a large number of 0s, an alternative format may be used, such as the compressed sparse row (CSR) format. In CSR format, this matrix would be represented as

value	1	3	1	2	2	1	1	1	1
col_index	0	3	1	5	1	2	3	4	5
row_index	0	2	4	6	7	8	9		

Table 1. Caption

which only requires 25 integers stored instead of 36. The benefit increases as the sparsity factor increases up to a magnitude of $O(n)$.

Matrix solving results

The area where gains were most likely to be realized is in the actual solving, which is a highly-parallizable problem. In order to benchmark the speed and memory use of dense matrices in small FEM problems, cuSolverDN, a dense solver suite by NVIDIA, has been integrated into Polysolve, a solver API written in C++. cuSolverDN was compared against PARDISO, a sparse linear solver. cuSolverDN had a clear time advantage when tested on a small problem using a 120×120 matrix.

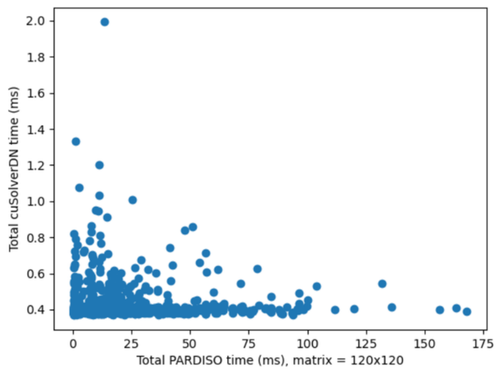

Figure 2. Speed of cuSolverDN compared against PARDISO.

Assembly Results

A subset of Thingi10K, a dataset of 10,000 models uploaded to Thingiverse, was used to test the speed of assembling the Laplacian matrix. Code from libigl was used in order to assemble the matrices. This operation is not readily parallelizable, because calculating the diagonal entries of the Laplacian matrix requires precalculating the other entries first.

Assembling a dense Laplacian matrix is significantly slower than a sparse matrix. Upon further analysis, this is due to the speed of loading the memory into cache and zeroing the space in memory where the dense matrix will be stored. Since the matrix is assembled iteratively, starting with a block of zeroed memory is necessary for the algorithm.

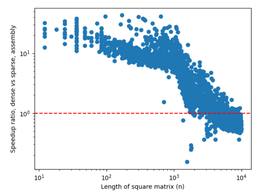

Figure 3. Assembly speedup ratio compared with the size of the matrix used. Time to set the memory to zero and load the memory into cache is not counted.

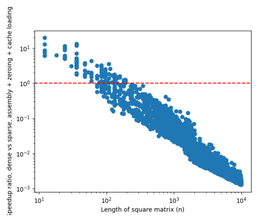

Figure 4. Assembly speedup ratio compared with the size of the matrix used under real-world computing conditions.

Conclusion and future work

There are a couple routes to be taken from here; one tweak is to rearrange the Laplacian assembly algorithm to make it more parallelizable. The cotangent values for edges can be calculated first in parallel, as they are all independent of each other. Each diagonal entry can then be computed as a vectorized sum of the row it represents.

Another test of feasibility would simply be to run PolyFEM with cuSolverDN on simple Laplacian problems. Additional work would be needed in order to benchmark this process; time may be added from the assembly or solving steps.

Though this is a negative result, I still think this project is worthwhile. Forming a dataset of small physical simulation problems could help create a machine learning model to solve similar problems. Since the application of forces is also a PDE, this methodology could theoretically be used to help embodied robots interact with real-world objects.

References

[1] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 233–244, 2009.

[2] Alec Jacobson, Daniele Panozzo, et al. libigl: A simple C++ geometry processing library, 2018. https://libigl.github.io/.

[3] Sebastian Koch, Teseo Schneider, Francis Williams, Chengchen Li, and Daniele Panozzo. Black box geometric computing with python. https://geometryprocessing.github.io/blackbox-computing-python/.

[4] Michael McCabe, Bruno Régaldo-Saint Blancard, Liam Holden Parker, Ruben Ohana, Miles Cranmer, Alberto Bietti, Michael Eickenberg, Siavash Golkar, Geraud Krawezik, Francois Lanusse, et al. Multiple physics pretraining for physical surrogate models. *arXiv preprint arXiv:2310.02994*, 2023.

[5] Teseo Schneider, Jérémie Dumas, Xifeng Gao, Denis Zorin, and Daniele Panozzo. Polyfem. `https://polyfem.github.io/`, 2019.

[6] Qingnan Zhou and Alec Jacobson. Thingi10k: A dataset of 10,000 3d-printing models. *arXiv preprint arXiv:1605.04797*, 2016.