Parallel Implementation of 3D Truss Solver using CUDA and the Direct Stiffness Method

Mark Blanco
Computer & Systems Engineering
Computer Science
Blancm3@rpi.edu

Nathan Bernard
Computer Science
Computer & Systems Engineering
bernan@rpi.edu

Drake Perrior-Small
Computer Science
perrid3@rpi.edu

ABSTRACT

We implement a parallel version of the Direct Stiffness Method for 3D trusses in static equilibrium. Finite element analysis (FEA) is used to perform stress and force analysis on a range of designed structures, from small CAD components to bridges and buildings. The calculations can be performed for statically determinate systems using the Method of Joints and for Statically Indeterminate systems using the Direct Stiffness Method. To achieve parallelism, we used cuSOLVER, a part of the CUDA programming API, to rapidly solve matrix systems representing the force-displacement characteristics of the given truss, and OpenMP, a compiler-level CPU threading framework, to parallelize data processing prior to the CUDA calls.

KEYWORDS

Direct Stiffness Method, Finite Element Analysis, Trusses, CUDA, OpenMP, Parallel Computing

1. INTRODUCTION

As a means of mechanical system analysis, finite element methods (FEA) operate by segmenting a larger structure into smaller quantized sections. These sections can generally be constructed of frames or trusses. In either, members are connected at vertices. This paper's implementation focuses on systems in static equilibrium which are defined as having net zero externally applied forces in all dimensions.

Frames support moments about their vertex connections, thereby offering greater realism in the analysis but increasing complexity in implementation and computation [1]. In a truss structure the members only support forces along their axes, and do not support moments at their connection points, transverse loads, or bending moments. In effect, truss members are joined by frictionless pins.

For implementation as a computer algorithm, truss structures may be represented by a system of joints and edges. The Method of Joints is a simple algorithm for solving for the forces within each member in a truss [2]. The sum of forces at each vertex, or node, in the direction of each coordinate axis is characterized as an equation with known applied external forces and unknown components of forces in each member. The individual equations of knowns and unknowns are compiled into a matrix of coefficients for the unknown member forces and a vector of known applied forces. The forces in each member are solved by inverting the matrix.

The method of joints applies well to statically determinate systems in static equilibrium. In statically determinate systems the number of unknowns M is equal to the number of equations N (which will be the number of nodes n times the number of coordinate directions d under consideration; $N = n * d$). The system can still be solved for consistent overdetermined systems ($N > M$), but cannot be solved for statically indeterminate systems where $M > N$.

The Method of Stiffness is a popular means of solving statically indeterminate truss systems on computers [3]. The method can solve a system for $M > N$ by considering deformation and elasticity of the members, based on Hooke's Law (shown below).

$$\sigma = \epsilon E$$

Hooke's Law relates the stress (force per unit area, σ) to the strain in a material (change in length over initial length, ϵ), by the material constant E, known as Young's Modulus or the Modulus of Elasticity. This introduces compatibility conditions: the deformation of each member must be system-consistent such that each node's displacement matches the changes in length of all members that are attached to that node. The stiffness matrices of each member are rearranged and combined into a global system matrix, which with a vector of applied external forces, can be used to find the displacements of each node. The force in each member can be calculated based on the displacement of its endpoints.

The use of matrix systems is an obvious area for parallelism in a computerized implementation of the Direct Stiffness Method. Matrix and vector operations see significant speedup when performed on SIMD architectures such as NVidia GPUs using CUDA. Data processing and compilation of the system matrix can also be parallelized on the CPU using frameworks such as OpenMP. The implementation presented here uses both of these software frameworks.

2. Algorithm for Direct Stiffness Method

The following section describes the algorithm and theory for the method of Direct Stiffness. Each element in the truss system has a local stiffness matrix that describes the required forces in each coordinate direction on the two endpoints that result in a unit displacement of that endpoint in the direction of the applied force. The upper left quadrant of the local stiffness matrix is defined as follows:

$$K_{L,UL} = \frac{AE}{L} * \begin{matrix} c_x c_x & c_x c_y & c_x c_z \\ c_y c_x & c_y c_y & c_y c_z \\ c_z c_x & c_z c_y & c_z c_z \end{matrix}$$

Where A is the cross sectional area, L is the length, and E is the modulus. c_x, c_y and c_z are proportions of the length of the member in each coordinate direction: $\boldsymbol{c} = (\frac{\Delta x}{L}, \frac{\Delta y}{L}, \frac{\Delta z}{L})$.

The stiffness matrix for the member is then composed as follows:

$$K_L = \begin{matrix} K_{L,UL} & -K_{L,UL} \\ -K_{L,UL} & K_{L,UL} \end{matrix}$$

Each entry in each quadrant of the local stiffness matrix corresponds to one or both vertices of the matrix and how they react to forces in a combination of the x,y, and z dimensions.

Compilation of the global stiffness matrix K_G is performed by summing quadrants of each K_L to sections corresponding to the nodes of the element that owns K_L. Vertices are numbered sequentially to facilitate this; the global matrix will be of size $3n$ for n vertices. The location of a local matrix quadrant is then $(3n_1{:}\,3n_1+2, 3n_2{:}\,3n_2+2)$ in K_G, assuming that vertex numbers $(n_1, start; n_2, end)$ begin at zero.

Rows and columns corresponding to constrained dimensions for each vertex can be dropped from the matrix, resulting in a reduced global matrix $K_{G,R}$ containing only the aggregated stiffness coefficients for directions to which vertices can be displaced.

The external forces in each unconstrained vertex direction are compiled into a loads vector, $\boldsymbol{f}$. The displacement for movable vertices can be solved for as $\boldsymbol{d} = \boldsymbol{K}_{G,R}^{-1}\boldsymbol{f}$. Finally, the force in each member is:

$$f_m = \boldsymbol{c}^T \cdot (\boldsymbol{d}_{n2} - \boldsymbol{d}_{n1})$$

Where $\boldsymbol{c}$ was defined on the previous page as row vector and $\boldsymbol{d}_{n1,2}$ are row vectors of the displacement in x, y, z for the start and end vertices of the member m.

3. Implementation

The truss solver was implemented in two parts – as a computational backend using CUDA and OpenMP on C++, and as a front-end visualizer with WebGL in JavaScript, CSS, and HTML. Unsolved and solved trusses are communicated between the two via a JSON (JavaScript Object Notation) file format.

3.1 Data Specification

The data is formatted as JSON with an associative array containing an array of vertices and an array of edges. Each vertex holds its location, the dimensional constraints applied to it, and the sum of all external forces applied to it. Each edge includes the indices of the two vertices it connects, its cross-sectional area, elastic modulus, internal forces, and stress.

```
{
    "Anchored": [ false, false, true ],
    "XYZAppliedForces": [-3778.1,-1493.4,0],
    "XYZPosition": [-3.0,0.85,-0.5]
},
    {
        "ElasticModulus": 25000000000,
        "Endpoints": [ 0, 1 ],
        "Force": 0.0,
        "SectionArea": 0.25,
        "Stress": 0.0
    },
```

Figure 1. JSON Format

Top: JSON Formatted Vertex

Bottom: JSON Formatted Member

This format was chosen as it can be rendered as-is before calculation, then after calculation, the backend can output a file or data stream in the same format to be rendered again using the same front end. The frontend and backend parse the same file format and ignore the information that they do not need in it.

3.2 Parallel Backend

The C++ backend uses OpenMP and cuSOLVER, part of the CUDA API, to parallelize its computations.

3.2.1 Overview

The C++ backend implements the Direct Stiffness method as described in Section 2. Three C++ classes were defined to organize information read from the input JSON data file: Node, Element, and Truss. Nodes characterize vertices in the truss structure. Elements characterize members connecting vertices and implement computation of the local stiffness matrices. The Truss class contains the Nodes and Elements corresponding to a given structure, creates the global stiffness matrix from local matrices, and calls cuSOLVER to find the displacements, and consequently the forces in each member. The host portions is this code are parallelized using OpenMP on for loops and sections that may be executed in parallel, such as data parsing from JSON and creating stiffness matrices.

3.2.2 Interaction

The main executable accepts command line arguments of the following form:

```
./main <inputFile> [-vvvvv] [-ccccc] -o
                    <outFile>
```

The input and output files are JSON, or if '-' is supplied then stdin or stdout are used instead. The verbosity and commentary flags are optional, and the number of repetitions of each character causes less or more output of intermediate results and printouts as to the status of the computations.

3.3 Visualization Interface

3.3.1 Technologies Used

The user interface is written in JavaScript and HTML, and utilizes WebGL for real-time rendering. This method was chosen as it is cross-platform and could be developed and used independently of the backend.

3.3.2 Overview

The truss currently loaded in the user interface is shown as the nodes of the truss (vertices, joints) connected by thin lines. Each line is a member of the truss (edge, beam) which is assumed to be of negligible diameter and mass. The members are colored relative to their internal stress after calculations have been completed. The nodes remain white as the net force on any node should be zero for any static system.

3.3.3 Interaction

The user can interact with the truss by clicking and dragging their mouse across the window. This rotates the truss relative to the origin of the coordinate system the truss is defined relative to. Holding shift while click-dragging pans the truss around and adjusts the center of rotation of the camera. Scrolling the mouse zooms in or out on the truss.

4. Results

The solver was run on the machine provided in class, known as parallel.ecse.rpi.edu. This machine has dual 14-core Intel Xeon E5-2660 processors clocked at 2.0 GHz and 256GB of ECC memory.

The GPU is a Pascal architecture GeForce GTX 1080 with 8GB of GDDR5 memory. Runtime results were collected for runs on automatically generated trusses of several sizes and with the maximum number of running OpenMP threads set to either 1 or 56.

4.1 Performance

Performance testing indicates that the most time spent in runtime is in solving the matrix on the GPU, and in computing a reduced matrix to send to the GPU. This is fairly intuitive, as these operations require the largest arrays and the most individual calculations in order to process. All other operations were able to complete in a combined time less than either of the two largest operations.

The implementation of parallelization has mixed results for lower numbers of edges. The two largest operations are mostly unchanged in runtime. The smaller operations, mostly element parsing and post-GPU displacement calculations, show slowdown within millisecond levels.

4.2 Time Complexity

All calculations on the CPU are at a linear time complexity. The data structures kept throughout computation are an array of Elements, or edges, and an array of Vertices. By mapping relevant values into a matrix in place of iterative comparisons, each value of the array must only be iterated over once to store back for each operation. Operations set both get and set values, but this multiplier is small compared to the number of elements in each array.

Operations on the GPU are all matrix solving operations that are part of the CUDA backend. LUD composition is a primary operation needed to solve for forces, thus the time complexity of the GPU operation is likely similar to the time complexity of LUD composition as defined by CUDA.

4.3 Memory Complexity

Memory complexity of this algorithm is primarily affected by the storage of the Element and Node objects. These classes each have member variables of a constant size in memory. This allows the memory complexity to be defined as a linear relationship to both the sum of the number of nodes and the number of elements.

4.4 Scaling Testing

For smaller trusses (less than 3500 vertices and 14,000 members), the overall time taken to complete the calculations is relatively constant and appears to be due to the overhead of using the cuSOLVER library. In the single-threaded implementation, the times for all other subtasks scaled approximately linearly with the size of the input. When OpenMP was introduced, the time for each subtask was significantly greater, but relatively constant, again, owing itself to overheads of using multiple threads and requiring atomic operations. As the input size increased though (over 500 layers), the multithreaded implementation began to outperform the single-threaded implementation. As the input size increased beyond 500 layers (at 7 vertices per layer, 29 members per layer) the time spent in the GPU became exponential and became the leading factor in the overall runtime of the program.

5. Conclusion

Based on the deformations and general consistency of results, we can imply that the program is running successfully for the algorithm implemented. The speed up seen through parallelization so far is mixed in results, but initial testing on arbitrarily large systems suggests that parallelization may have significant improvements for large systems. Further testing is required to verify this implication, but the trends shown across truss sizes would suggest this as well. Initial testing for usability, runtime, and output consistency suggests that this solution has a low barrier to entry and may be viable in scientific or industrial applications for low latency and high consistency.

6. Discussion

As the number of nodes and edges increase (in our testing, beyond 3500 nodes and 14000 edges) the largest contribution to the total runtime comes from the calculations being performed on the GPU, and the increase is exponential with respect to the number of nodes and edges. With further development, it would be interesting to see the effect of using CUDA's sparse matrix solver on the overall time as well as scaling given the sparse nature of the matrices being solved. When looking at the data generated by this system, it is obvious that the matrices generated are increasingly sparse with larger and larger systems. This is inherent to truss systems in industry, as the focus is on maximizing structural support while minimizing the materials needed. In these systems especially, sparseness would become the dominant feature in matrices.

Another practical application of this truss solving system goes into generalized finite element systems. The direct stiffness method does not require the system to be determinate, and can look at any system of nodes and edges given sufficient information of their positions and composition. The implication of this is the use of this parallelized solving algorithm into general wire mesh models, or looking at decompositions of complex systems into finite elements structured as nodes and elements. Because the solver and data specification are system agnostic, and can work on any well-formed data set, the applications can extend well beyond initial considerations as well.

7. ACKNOWLEDGMENTS

We would like to thank Professor Franklin in the ECSE Department for guidance in approaching parallelization as well as access to the machine used for compilation and testing, and to Professor K. Mills in the MANE Department for providing resources for FEA methods.

8. References

[1] V. E. Saouma, "Lecture Notes in: Finite Element I - Framed Structures," [Online]. Available: http://ceae.colorado.edu/~saouma/Lecture-Notes/lecmat.pdf. [Accessed April 2017].

[2] University of Memphis, "CIVL 3121: Trusses - Method of Joints," [Online]. Available: http://www.ce.memphis.edu/3121/notes/notes_03b.pdf. [Accessed April 2017].

[3] Purdue University, "Stiffness Methods for Systematic Analysis of Structures," [Online]. Available: https://engineering.purdue.edu/~aprakas/CE474/CE474-Ch5-StiffnessMethod.pdf. [Accessed April 2017].

[4] S. Orestis and S. Nikolaos, "The Implementation of the Direct Siffness Method in AutoCAD," in *Proc. Εφαρμογές Πληροφορικής και Μεθοδολογίες Διοίκησης*, Thessaloniki, Greece, 2010.

[5] Duke University, "Introduction to Stiffness Matrix Assembly for Trusses," [Online]. Available: http://people.duke.edu/~hpgavin/cee421/truss-intro.pdf. [Accessed April 2017].

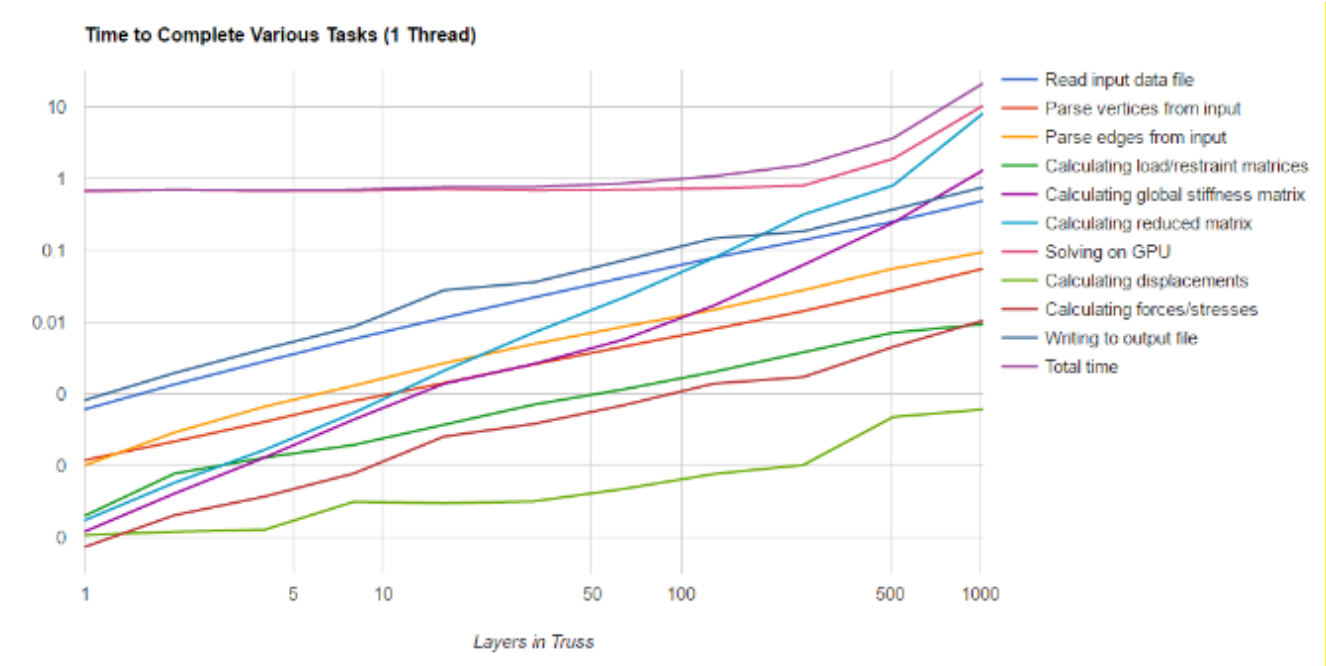

Figure 2: Times to complete various subtasks with 1 CPU thread

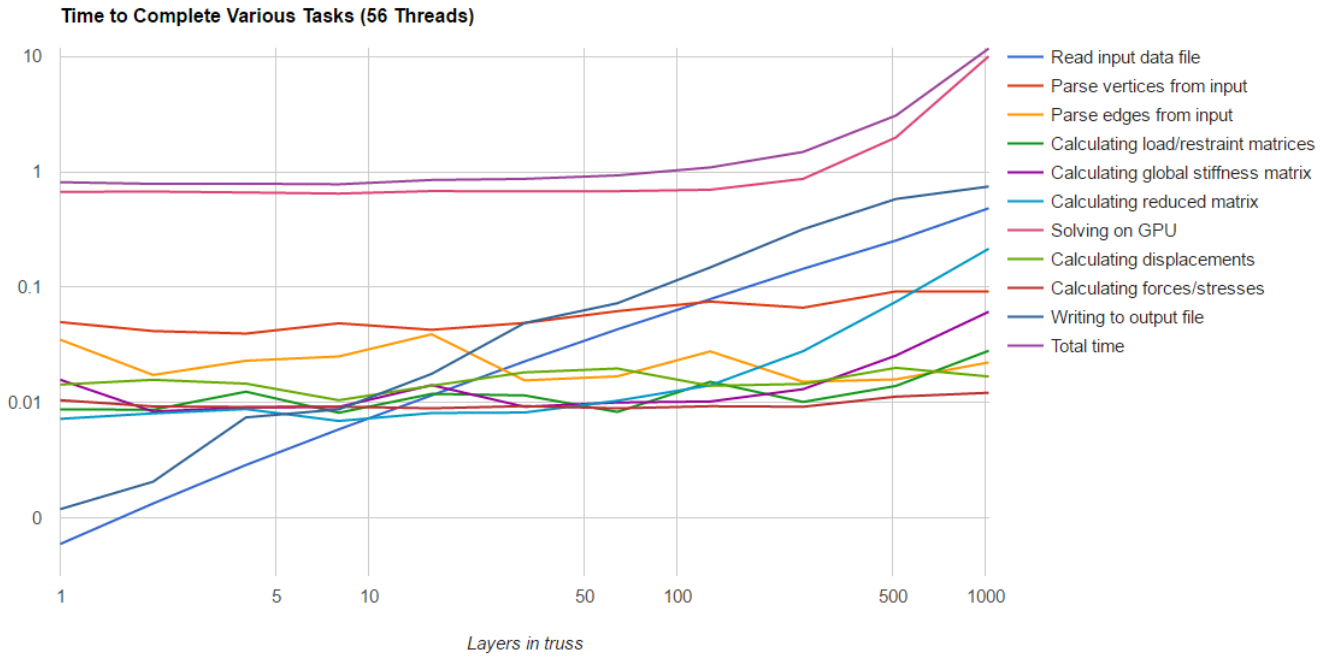

Figure 3: Times to complete various subtasks with 56 CPU threads

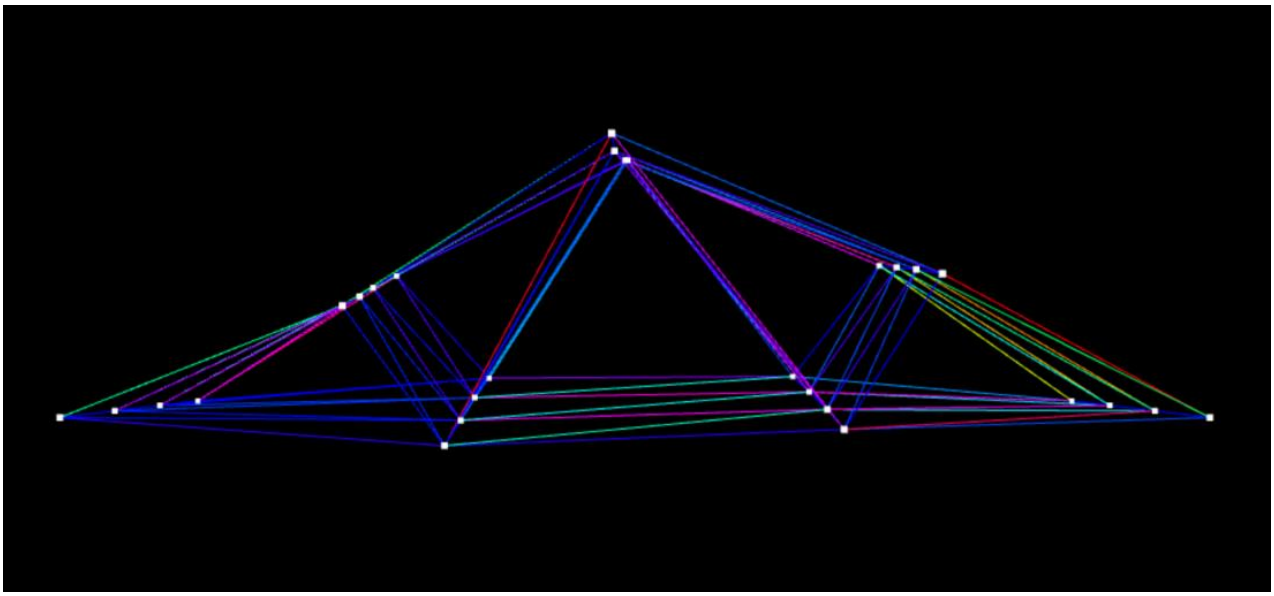

Figure 4: 4 layer truss with uniform random forces applied to vertices

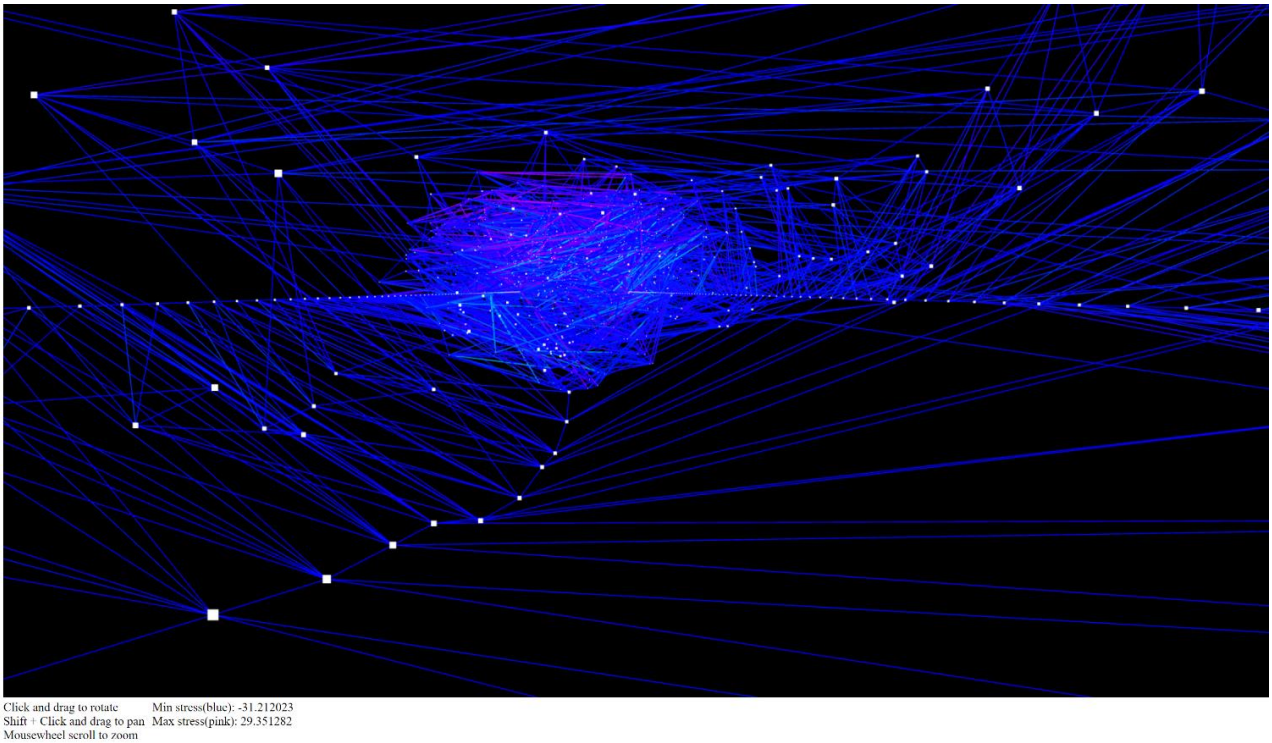

Figure 5: Very deformed truss structure with high external forces

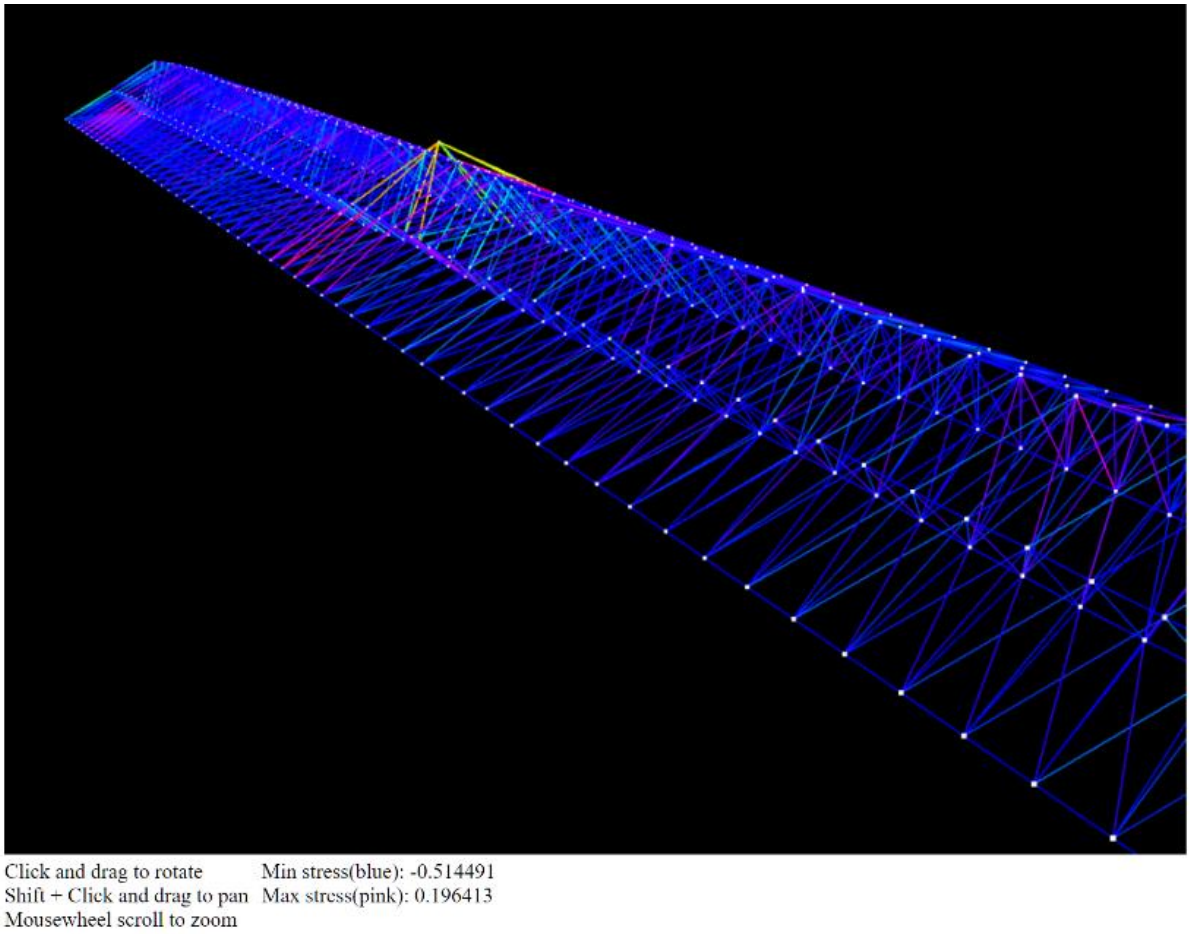

Figure 6: 64 layer truss with uniform force distribution except for center