






# Combination of GPU Programming and FEM Analysis in Structural Optimisation

Szilárd Nagy<sup>1,2</sup>(✉) , Károly Jármai<sup>2</sup> , and Attila Baksa<sup>2</sup> 

<sup>1</sup> Emerson Automation FCP Kft., Eger 3300, Hungary  
szilard.nagy@emerson.com

<sup>2</sup> University of Miskolc, Miskolc 3515, Hungary  
{karoly.jarmai, attila.baksa}@uni-miskolc.hu

**Abstract.** GPUs no longer only support graphical applications and gaming. These are becoming cheap and powerful tools for scientific and general-purpose computations. They provide a massively parallel environment with the support of a single instruction multiple data (SIMD) programming model. Making finite element calculations is also a time-consuming process in some cases due to many elements or a large degree of freedom. The FEM simulation is essential to check the analytical or measured mechanical stresses, deformations, etc. In making structural optimisation, one needs several iterations and combining the optimisation with FEM, increasing the calculation time. GPU programming is a good solution for this. In the article, we show the applicability of the combination of GPU, optimisation, and FEM simulation.

**Keywords:** Evolutionary optimisation · Finite element method · Parallel computation

## 1 Introduction

Nowadays, graphic cards (video cards, GPU) are cheap and efficient hardware for general-purpose parallel computation. They are used for scientific computations, for topology optimisation [1] or structural optimisation [2], and for manufacturing technologies. They provide a massively parallel environment with the support of a single instruction multiple data (SIMD) programming model. Nowadays, larger software vendors – such as MathWorks – are increasingly developing frameworks based on the CUDA API (application programming interface) that offer more convenient and user-friendly tools than the original CUDA Runtime API.

Nature-inspired, population-based, iterative, evolutionary algorithms – such as flower pollination algorithm [3], particle swarm optimisation [4], firefly algorithm [5], etc. – are powerful numerical optimisation methods. Their importance and effectiveness are underlined by the fact that they are used in several places in vehicle research to design optimal aerodynamics for UAVs (unmanned aerial vehicles) [6], for performance optimisation of formula vehicles [7], for optimising the manufacturing of vehicles [8] etc.

The finite element method (FEM) is a universal tool for analysing structures and determines mechanical stress and deformations inside the structure. In this paper, we connect an evolutionary algorithm – differential evolution – with FEM. The method is presented through the optimisation of the truss structure. Computational capacity is demanded by both the evolutionary method and FEM. Therefore, we present a possible parallelisation using MATLAB software and the obtained results.

## 2 Differential Evolution

Storn and Price introduced original differential evolution (DE) in [1]. DE improves the  $n_D$  dimensional  $\mathbf{x}$  individuals of  $n_p$  element population through a series of iteration steps

$$\mathbf{x} = [x_1 \ x_2 \ x_3 \ \dots \ x_{n_D}]^T \in S \subset \mathbb{R}^{n_D} \tag{1}$$

where  $S$  is searching space. Ideally, the initial population randomly covers the entire search space. Each variable in an individual is a uniformly distributed random number in the search space.

DE generates the new entity in each iteration step by performing three operations repeatedly. These are called mutation, crossover, and selection operations [9].

During the mutation operation, a  ${}^G\mathbf{v}_i$  mutant is generated for each  ${}^G\mathbf{x}_i$  individual of  $G$  generation using one of the following five strategies [10]:

- DE/rand/1:

$${}^G\mathbf{v}_i = {}^G\mathbf{x}_{r_1} + F({}^G\mathbf{x}_{r_2} - {}^G\mathbf{x}_{r_3}) \tag{2}$$

- DE/best/1:

$${}^G\mathbf{v}_i = {}^G\mathbf{x}_b + F({}^G\mathbf{x}_{r_1} - {}^G\mathbf{x}_{r_2}) \tag{3}$$

- DE/current to best/2:

$${}^G\mathbf{v}_i = {}^G\mathbf{x}_i + F({}^G\mathbf{x}_b - {}^G\mathbf{x}_i) + F({}^G\mathbf{x}_{r_1} - {}^G\mathbf{x}_{r_2}) \tag{4}$$

- DE/best/2:

$${}^G\mathbf{v}_i = {}^G\mathbf{x}_b + F({}^G\mathbf{x}_{r_1} - {}^G\mathbf{x}_{r_2}) + F({}^G\mathbf{x}_{r_3} - {}^G\mathbf{x}_{r_4}) \tag{5}$$

- DE/rand/2:

$${}^G\mathbf{v}_i = {}^G\mathbf{x}_{r_1} + F({}^G\mathbf{x}_{r_2} - {}^G\mathbf{x}_{r_3}) + F({}^G\mathbf{x}_{r_4} - {}^G\mathbf{x}_{r_5}) \tag{6}$$

where  $r_1 \neq r_2 \neq r_3 \neq r_4 \neq r_5 \in [1, n_p]$  are random indices,  $F \in [0, 2)$  is the scaling factor, and  ${}^G\mathbf{x}_b$  is individual with the best fitness value in each  $G$  generation.

The mutation is followed by a “binomial” crossover operation that combines the newly created  ${}^G\mathbf{v}_i$  mutant with the  ${}^G\mathbf{x}_i$  individual

$${}^G\mathbf{u}_{j,i} = \begin{cases} {}^G\mathbf{v}_{j,i} & U_j(0, 1) \leq C_R \text{ or } j = j_R \\ {}^G\mathbf{x}_{j,i} & \text{otherwise} \end{cases} \tag{7}$$

where  $U_j(0, 1) \in [0, 1)$  is uniformly distributed random number,  $C_R \in [0, 1)$  is the crossover rate, and  $j_R \in [1, n_D]$  is a random index.

During selection, if the fitness value of the newly generated  ${}^G\mathbf{u}_i$  is better than that of the  ${}^G\mathbf{x}_i$ , it will be included in the new generation; if not, the algorithm drops it

$${}^{G+1}\mathbf{x}_i = \begin{cases} {}^G\mathbf{u}_i & \mathcal{F}({}^G\mathbf{u}_i) \leq \mathcal{F}({}^G\mathbf{x}_i) \\ {}^G\mathbf{x}_i & \text{otherwise} \end{cases} \tag{8}$$

The operation of differential evolution, and hence the success of the optimisation, is greatly influenced by the mutation strategy chosen, the value of the scaling factor  $F$ , and the  $C_R$  crossing ratio.

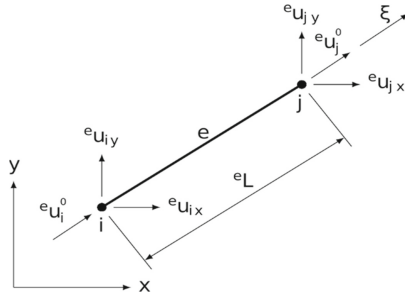
### 3 Finite Element Model of Truss Structure

The connection between members of tubular trusses is frequently modelled as pin connection inelastic analysis. The preferred value of eccentricities of the intersection of member’s center lines is [1, 12].

$$e \leq 0.25D \text{ or } e \leq 0.25H_0 \tag{9}$$

where  $e$  is eccentricity,  $D$  is the outside diameter of a circular hollow section, and  $H_0$  is a typical size of rectangular hollow section. In this case, primary bending moments are produced by these eccentricities. Excessive moments are generated in brace members when rigid connections are considered. Usage of these is not recommended also for welded joints [1, 12]. The axial force distribution in a rigid joint is like pinned joint.

The structure could be analysed with the pushed-pulled element model (shortly in the following rod or truss model) with finite element methods (FEM) if the condition of inequality (9) is met. In most cases, it is sufficient to examine the structure in a plane relevant to the load. If this was insufficient and the spatial analysis had to be performed, the presented method could be easily adapted to a spatial case. In this paper, we will only discuss planar problems.



**Fig. 1.** Truss element model.

The model of the truss element is shown in Fig. 1. An approximation of the displacement within the rod element with a kinematically admissible function [13]

$${}^e u(\xi) = \begin{bmatrix} \frac{\xi_i - \xi}{eL} & \frac{\xi_j - \xi}{eL} \end{bmatrix} \begin{bmatrix} e u'_i \\ e u'_j \end{bmatrix} = [{}^e N_i(\xi) \quad {}^e N_j(\xi)] \begin{bmatrix} e u'_i \\ e u'_j \end{bmatrix} = {}^e N {}^e \mathbf{u}' \quad (10)$$

where  $eL$  is the length of the rod element,  ${}^e N$  is the matrix of shape functions, and  ${}^e \mathbf{u}'$  is the vector of nodal displacement interpreted in element connected  $\xi$  coordinate system. In the global  $x - y$  coordinate system, nodal displacements could be described in the following form

$${}^e \mathbf{u} = [e u_{ix} \quad e u_{iy} \quad e u_{jx} \quad e u_{jy}]^T. \quad (11)$$

The transformation between the two-coordinate system could be made with the transformation matrix

$${}^e \mathbf{T} = \begin{bmatrix} {}^e T_{11} & {}^e T_{12} & 0 & 0 \\ 0 & 0 & {}^e T_{23} & {}^e T_{24} \end{bmatrix} \quad (12)$$

where

$${}^e T_{11} = {}^e T_{23} = \frac{e u_{jx} - e u_{ix}}{eL} \text{ and } {}^e T_{12} = {}^e T_{24} = \frac{e u_{jy} - e u_{iy}}{eL} \quad (13)$$

$${}^e \mathbf{u}' = {}^e \mathbf{T} {}^e \mathbf{u} \quad (14)$$

Elongation of truss element is

$${}^e \varepsilon = \frac{d^e u(\xi)}{d\xi} = \frac{1}{eL} [-1 \quad 1] {}^e \mathbf{u}' \quad (15)$$

and stress in the axial direction is

$${}^e \sigma = E {}^e \varepsilon = \frac{E}{eL} [-1 \quad 1] {}^e \mathbf{u}' \quad (16)$$

where  $E$  is elastic modulus. The strain energy of truss element with  ${}^eA$  cross-sectional area is

$${}^eU = \frac{1}{2} \int_L {}^eA \sigma^e \varepsilon d\xi = \frac{1}{2} {}^e\mathbf{u}'^T \frac{{}^eAE}{{}^eL} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} {}^e\mathbf{u}' = \frac{1}{2} {}^e\mathbf{u}'^T {}^e\mathbf{K}' {}^e\mathbf{u}' \quad (17)$$

where  ${}^e\mathbf{K}'$  is the stiffness matrix of the element. The work of external forces is

$${}^eW = \int_L {}^e u(\xi) p d\xi = {}^e\mathbf{u}'^T {}^e\mathbf{f}' \quad (18)$$

where  ${}^e\mathbf{f}'$  is the vector of external forces reduced to nodes. The total potential energy of one element could be written in the following form

$${}^e\Pi_p = {}^eU - {}^eW = \frac{1}{2} {}^e\mathbf{u}'^T {}^e\mathbf{K}' {}^e\mathbf{u}' - {}^e\mathbf{u}'^T {}^e\mathbf{f}' \quad (19)$$

It could be rewritten with quantities, which are introduced in the global coordinate system

$${}^e\Pi_p = \frac{1}{2} {}^e\mathbf{u}^T {}^e\mathbf{K}^e {}^e\mathbf{u} - {}^e\mathbf{u}^T {}^e\mathbf{f} \quad (20)$$

where

$${}^e\mathbf{K} = {}^e\mathbf{T}^T {}^e\mathbf{K}' {}^e\mathbf{T} \text{ and } {}^e\mathbf{f} = {}^e\mathbf{T}^T {}^e\mathbf{f}' \quad (21)$$

Introducing the  $\mathbf{u}$  all node displacement vectors and the  $\mathbf{f}$  all node load vectors as the total potential energy of the whole structure is

$$\Pi_p = \frac{1}{2} \mathbf{u}^T (\mathbf{K}\mathbf{u} - \mathbf{f}) \quad (22)$$

where  $\mathbf{K}$  stiffness matrix of the complete structure according to the rules of element alignment, which is detailed described in [13, 14].

Many truss structures are built from different rods with different cross-sectional properties. These rods could be grouped by  $AE$  product. From the stiffness  $\mathbf{K}$  matrix introduced initially in (22), these  $AE$  product can be extracted by cross-sectional groups

$$\mathbf{K} = A_1 E_1 \mathbf{K}_1 + A_2 E_2 \mathbf{K}_2 + \dots + A_i E_i \mathbf{K}_i + \dots + A_{n_G} E_{n_G} \mathbf{K}_{n_G} = \sum_{i=1}^{n_G} A_i E_i \mathbf{K}_i \quad (23)$$

where  $n_G$  is the number of cross-sectional groups, and  $\mathbf{K}_i$  is stiffness matrix of  $i^{th}$  group. If the unknown quantities of the optimisation are typical cross-section dimensions (for example,  $D$  outside diameter and  $t$  wall thickness for circular hollow section), pre-processing of FEM is enough to do it once before the first iteration step of optimisation.

According to the principle of minimum total potential energy [15, 16], the  $\delta\Pi$  first variation of  $\Pi$  total potential energy is zero. After applying boundary conditions, we get an algebraic equation system of FEM

$$\delta\Pi_p = \delta\mathbf{u}^T \frac{\partial\Pi_p}{\partial\mathbf{u}} = \delta\mathbf{u}^T (\mathbf{K}\mathbf{u} - \mathbf{f}) = 0 \quad (24)$$

$$\mathbf{u} = \mathbf{K}^{-1}\mathbf{f}. \tag{25}$$

Post-processing of the result of Eq. (25) is necessary for further calculations. Axial stress of elements could be determined by

$${}^e\sigma = \frac{{}^eE}{{}^eL} [-{}^eT_{11} \ -{}^eT_{12} \ {}^eT_{11} \ {}^eT_{12}] {}^e\mathbf{u} \tag{26}$$

### 4 The Optimisation Problem

Optimisation of truss structures are constrained optimisation problem

$$\begin{aligned} \min: f(\mathbf{x}) \quad \mathbf{x} &= [x_1 \ x_2 \ \dots \ x_D]^T \in \mathbb{R} \\ g_i(\mathbf{x}) &\leq 1 \quad 1 \leq i \leq q \\ h_j(\mathbf{x}) &= 0 \quad 1 \leq j \leq r \end{aligned} \tag{27}$$

where  $\mathbf{x}$  is the vector of unknowns – in this paper, vector of typical dimensions of cross-section –,  $f(\mathbf{x})$  is the objective function to be optimised,  $g_i(\mathbf{x})$  are inequality constraints,  $h_j(\mathbf{x})$  are equality constraints,  $q$  and  $r$  are the numbers of constraints.

In this paper, the target function of optimisation is the weight of the structure

$$f(x) = \rho \sum_{e=1}^{n_e} {}^eA {}^eL \tag{28}$$

where  $n_e$  is the number of truss elements, where  $\rho$  is the density of steel.

The structure must meet strength and stability requirements. In the present case, three criteria have been analysed. In the case of pulled rods, the resistance to tensile stress, and in the case of pushed rods, the buckling and finally the local buckling. The cross-sectional utilisation factor can well characterise these characteristics.

A definition of an inequality condition can interpret the tensile and compressive strength of pushed-pulled rods if the stress from the load is interpreted as a sign. Negative tension means pressure, while positive means tension.

$$g_{ii} = \begin{cases} \frac{\gamma_{M0} |{}^e\sigma|}{\chi f_y} \leq 1 & {}^e\sigma < 0 \\ \frac{\gamma_{M0} |{}^e\sigma|}{f_y} \leq 1 & {}^e\sigma \geq 0 \end{cases} \tag{29}$$

where  $f_y$  is yield strength,  $\gamma_{M0}$  is a safety factor according to [17], and  $\chi$  is buckling factor also according to [17]

$$\chi = \begin{cases} 1 & \bar{\lambda} \leq 0, 2 \\ \frac{1}{\phi + \sqrt{\phi^2 + \bar{\lambda}^2}} & \bar{\lambda} > 0, 1 \end{cases} \tag{30}$$

where  $\phi$  is a factor

$$\phi = 0, 5 \left( 1 + 0, 21(\bar{\lambda} - 0, 2) + \bar{\lambda}^2 \right) \tag{31}$$

$\lambda$  is a slenderness factor

$$\bar{\lambda} = \pi kL \sqrt{\frac{A}{I_x}} \sqrt{\frac{f_y}{E}} \tag{32}$$

where  $I_x$  is the second-order moment of the used cross-section,  $k$  is the deflection length factor, which is  $k = 1$  for intermediate bars and  $k = 0.7$  for the gripped bars.

The limit of local buckling depends on the shape of the cross-section. A different formula should be used for a different shape [11]. Currently, we use local buckling of circular hollow section

$$g_{III} = \frac{Df_y}{21150t} \leq 1 \tag{33}$$

This formula is valid only if the unit of  $f_y$  yield stress is in MPa, and the unit of  $D$  diameter and unit of  $t$  wall thickness is mm.

Using Eqs. (28), (29) and (33), the fitness function to be optimised

$$\mathcal{F}(\mathbf{x}) = \rho \sum_{e=1}^{n_e} A^e L + \sum_{i=1}^{n_e} p(g_{II}(\mathbf{x})) + \sum_{i=1}^{n_G} p(g_{III}(\mathbf{x})) \tag{34}$$

where  $p$  is the static penalty function

$$p(\mathbf{x}) = \begin{cases} 0 & g(\mathbf{x}) \leq 1 \\ 10^6 g(\mathbf{x}) & g(\mathbf{x}) > 1 \end{cases} \tag{35}$$

and  $\mathbf{x}$  is the vector of unknowns (vector of independent variables). For example, in the case of a circular tube, un-knowns are characteristic dimensions of the cross-section

$$\mathbf{x} = [d_1 \ d_2 \ \dots \ d_{n_G} \ t_1 \ t_2 \ \dots \ t_{n_G}]^T \tag{36}$$

## 5 Parallelisation with CUDA and MATLAB

Nvidia corporation offers CUDA Driver API [18] and CUDA Runtime API [19] to program their graphics cards for general-purpose computation. There are many types of graphics cards on the market, with different computation capabilities and performance. The codec containing our unique calculation must be scalable [20], and it should automatically detect the used hardware capabilities [21]. Implementing this feature is sometimes more challenging than implementing our custom calculation. As an intermediate layer between CUDA and our code, MATLAB offers much simpler possibilities for implementing our parallel computation [22]. However, this ease of use comes at a price, so the computation speed increase will never be as great as using only native CUDA.

MATLAB gives a reach toolset and many features to make operations with vectors and matrices. It offers many possible ways to rewrite original loop-based, scalar oriented operations to vector-matrix operations. This process is called “vectorisation”.

To illustrate the differences between the two types of operations, let the population be given as follows for circular hollow section tubes

$$X = \begin{bmatrix} D_{1,1} & D_{1,2} & \cdots & D_{1,j} & \cdots & D_{1,n_p} \\ D_{2,1} & D_{2,2} & \cdots & D_{2,j} & \cdots & D_{2,n_p} \\ \vdots & \vdots & & \vdots & & \vdots \\ D_{i,1} & D_{i,2} & \cdots & D_{i,j} & \cdots & D_{i,n_p} \\ \vdots & \vdots & & \vdots & & \vdots \\ D_{n_G,1} & D_{n_G,2} & \cdots & D_{n_G,j} & \cdots & D_{n_G,n_p} \\ t_{1,1} & t_{1,2} & \cdots & t_{1,j} & \cdots & t_{1,n_p} \\ t_{2,1} & t_{2,2} & \cdots & t_{2,j} & \cdots & t_{2,n_p} \\ \vdots & \vdots & & \vdots & & \vdots \\ t_{i,1} & t_{i,2} & \cdots & t_{i,j} & \cdots & t_{i,n_p} \\ \vdots & \vdots & & \vdots & & \vdots \\ t_{n_G,1} & t_{n_G,2} & \cdots & t_{n_G,j} & \cdots & t_{n_G,n_p} \end{bmatrix} = \begin{bmatrix} \mathbf{D} \\ \mathbf{t} \end{bmatrix} \quad (37)$$

where  $D_{i,j}$  is diameter of  $i^{th}$  cross-sectional group for  $j^{th}$  the individual in the population,  $t_{i,j}$  is the wall thickness of  $i^{th}$  cross-sectional group for  $j^{th}$  the individual in the population. For example, the loop-based, scalar oriented implementation of Eq. (33) could be seen in Listing 1.

```

...
f_y=235;
[n_row,n_col]=size(D); //or [n_row,n_col]=size(t);
g_II=zeros(n_row,ncol);
for i=1:n_row
    for j=1:n_col
        g_II(i,j)=(D(i,j)*f_y)/(21150*t(i,j));
    end
end
...

```

**Listing 1** Example source code for loop-based, scalar oriented operation

In contrast, the implementation in Listing 2 of the same equation covers a vectorised form.

```

...
f_y=235;
g_II=zeros(size(D)); //or g_II=zeros(size(t));
g_II=(D.*f_y)./(21150.*t)
...

```

**Listing 2** Example source code for vectorised operation



The striking difference between the two code snippets is that the latter is much shorter and more transparent. Sometimes, scalar-oriented operation vectorisation may not be formulated with element-wise operations (such as `.*`, `./`, `.^`, etc.). In such cases, `arrayfun()` could be a good tool. The point is that the scalar operation inside the loop core must be organised into a separate function (see in Listing 3). `arrayfun()` will call this function one at a time as many times as many elements in the vector or matrix passed as a parameter.

```

...
function g_II_ij=calc_g_II(D_ij,t_ij)
    f_y=235;
    g_II_ij=(D_ij*f_y)/(21150*t_ij);
end
...
//some other code
...
g_II=zeros(size(D)); //or g_II=zeros(size(t));
g_II=arrayfun(@calc_g_II,D,t);
...

```

**Listing 3** Example source code for vectorised operation with `arrayfun()`.

Provide tools for vectorising operations performed on multidimensional matrices using “page-wised” functions and operations. These detailed descriptions could be found in [22] for length reasons; these are not detailed in this paper.

MATLAB can always start the loop-based approach on only one thread, as illustrated in Listing 1. The situation is different with vectorised operations. It can automatically detect repetitive operations where only the data to be processed changes and automatically discover the capabilities of the runtime environment to perform them on multiple threads. In the simplest case, when using multi-core processors, it automatically – unless the opposite is set – takes advantage of the possibility of running on multiple cores in parallel. This automation also works for GPUs if the type of all variables in the expression is `gpuarray`. It automatically creates the required kernel functions based on the expressions and starts them on the required and possible number of threads, considering the capabilities of the GPU.

All the expressions and functions presented in previous chapters are easy to vectorise. This allows complete optimisation – evolutionary algorithm, FEM solver and fitness function calculation – to be calculated using GPU in parallel. If all steps and operations are calculated with a GPU, the host machine only manages them; it is enough to move data between the host and GPU at the beginning and end of the optimisation.

### 6 Comparison of Sequential and Parallel Optimisation

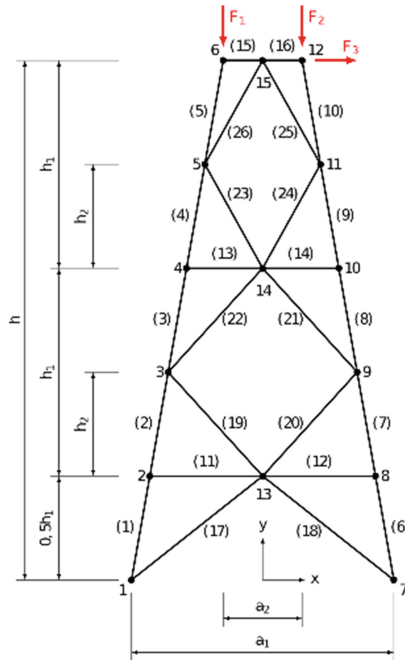
The dimensionless computation speed up between sequential and parallel processing is defined as follows

$$\nabla = \frac{t_{seq}}{t_{par}} \tag{38}$$

where  $t_{seq}$  is the average computation time of iteration steps using only sequential processing,  $t_{par}$  is the average computation time of iteration steps using only sequential processing. For measuring  $t_{seq}$  computation time, we used 1 pcs CPU thread, and for measuring  $t_{par}$  computation we used as many as possible thread on Geforce GTX 1050 Ti type graphics card.

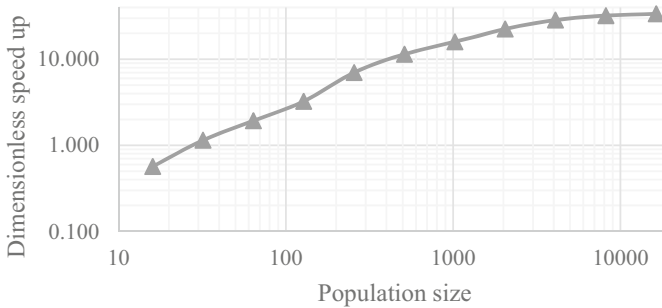
The structure shown in Fig. 2 was optimised to determine the previously defined rate increase. This is a truss structure with deltoid-shaped stiffeners. Applied loads were  $F_1 = 332.94\text{ kN}$ ,  $F_2 = 437.46\text{ kN}$  and  $F_3 = 338.08\text{ kN}$ . Node 1 and 7 were fixed, that means any displacement in these points is not allowed. Cross-section of all rods was a circular tube, where we optimised of outside diameter and wall thickness of tubes according to Eq. (36). Rods of the structure were divided into three cross-sectional groups. The first group contains rods 1–10. Horizontal rods (11–16) are in the second group. Finally, rods in the third group are rods of deltoid shape (17–26).

In our simulation, we have simulated optimisation with different numbers of individuals in the population which are used by SaDE. The dimensionless speed up achieved



**Fig. 2.** Sketch of optimised structure for comparison of sequential optimisation and parallel optimisation

is illustrated in Fig. 3, with different  $n_p$  population sizes. We did not inspect the quality of optima in this paper; we inspected only the difference in computation time.



**Fig. 3.** Dimensionless speed up with different population sizes and numbers of nodes.

## 7 Conclusion

An evolutionary algorithm is presented in this paper, the differential evolution. This algorithm relates to the finite element method for optimising truss-like structures subject to static stresses, overall buckling and local buckling. This is a powerful approach for optimising any truss structure automatically.

Evolutionary optimisation is a population-based iterative numerical method. That means the fitness function should be calculated many times; meanwhile, that could be a resource-demanding task and take a long time. One way to increase the speed of calculations is parallel computation with GPU. MATLAB offers user-friendly methods and tools for doing it. We have analysed dimensionless speed up of optimisation with tools of MATLAB.

The available speed up depends on the size of the population. Speed up increases approximately exponentially in the function of population size (see in Fig. 2). If the population size is small, there is no reason for parallelisation.

In future exploration, it could be interesting to inspect speed up in the function of the number of elements and number of nodes with fixed and varied size populations.

**Acknowledgements.** The research was supported by the Hungarian National Research, Development and Innovation Office—NKFIH under the project number K 134358.

## References

1. Xia, Z., Wang, Y., Wang, Q., Mei, C.: GPU parallel strategy for parameterized LSM-based topology optimization using isogeometric analysis. *Struct. Multidiscip. Optim.* **56**(2), 413–434 (2017). <https://doi.org/10.1007/s00158-017-1672-x>

2. Wang, J., Zhang, D., Luo, M., Zhang, Y.: A GPU-based tool parameters optimization and tool orientation control method for four-axis milling with ball-end cutter. *Int. J. Adv. Manuf. Technol.* **102**(5–8), 1107–1125 (2018). <https://doi.org/10.1007/s00170-018-2954-1>
3. Yang, X.S.: Flower pollination algorithm for global optimisation. In: Durand-Lose, J., Nataša, J. (eds.) *Unconventional Computation and Natural Computation*, pp. 240–249. Springer, Berlin, Heidelberg (2012)
4. Xie, X.F., Zhang, W.J., Yang, Z.L.: Adaptive particle swarm optimisation on individual level. In: *Proceedings of the 6th International Conference on Signal Processing, 2002*, vol. 2, pp. 1215–1218 (2002). <https://doi.org/10.1109/ICOSP.2002.1180009>
5. Yang, X.S.: *Nature-Inspired Optimization Algorithms*, 2nd (edn). Academic Press, London (2021). <https://doi.org/10.1016/C2019-0-03762-4>
6. Lee, D.S., Gonzalez, L.F., Srinivas, K., Periaux, J.: Robust evolutionary algorithms for UAV/UCAV aerodynamic and RCS design optimisation. *Comput. Fluids* **37**(5), 547–564 (2008). <https://doi.org/10.1016/j.compfluid.2007.07.008>
7. Tey, J.Y., Rahizar, R.: Handling performance optimisation for formula vehicle using multi-objectives evolutionary algorithms. *Veh. Syst. Dyn.* **58**(12), 1823–1838 (2020). <https://doi.org/10.1080/00423114.2019.1645861>
8. Galván-López, E., Curran, T., McDermott, J., Carroll, P.: Design of an autonomous intelligent demand-side management system using stochastic optimisation evolutionary algorithms. *Neurocomputing* **170**, 270–285 (2015). <https://doi.org/10.1016/j.neucom.2015.03.093>
9. Storn, R., Price, K.: Differential evolution – a simple and efficient heuristic for global optimisation over continuous spaces. *J. Global Optim.* **11**, 341–359 (1997). <https://doi.org/10.1023/A:1008202821328>
10. Qin, A.K., Suganthan, P.N.: Self-adaptive differential evolution algorithm for numerical optimisation. In: *Proceedings of the IEEE Congress on Evolutionary Computation*, vol. 2, pp. 1785–1791 (2005). <https://doi.org/10.1109/CEC.2005.1554904>
11. Wardenier, J., Kurobane, Y., Packer, J.A., van der Vegte, G.J., Zhao, X.-L.: *Design Guide for Circular Hollow Section (CHS) Joints Under Predominantly Static Loading*, 2nd edn. CIDECT, Zürich (2008)
12. Wardenier, J., Kurobane, Y., Packer, J.A., van der Vegte, G.J., Zhao, X.-L.: *Design guide for rectangular hollow section (RHS) joints under predominantly static loading*, 2nd edn. CIDECT, Zürich (2008)
13. Ferreira, A.J.M., Fantuzzi, N.: *MATLAB Codes for Finite Element Analysis*. Springer Cham, Heidelberg (2020). <https://doi.org/10.1007/978-3-030-47952-7>
14. Smith, I.M., Lee, M.: *Programming the Finite Element Method*, 5th edn. John Wiley and Sons Ltd, London (2013)
15. Páczelt, I.: *Finite element method in engineering practice (in Hungarian)*. Miskolci Egyetemi Kiadó, Miskolc (1999)
16. Páczelt, I., Baksa, A., Szabó, T.: *Fundamentals of the finite element method (in Hungarian)*. HEFOP jegyzet, Miskolc (2007)
17. EN 1993–1–1: Eurocode 3: Design of steel structures - part 1–1 General rules and rules for buildings. European Committee Standardization, Brussels (2009)
18. CUDA Drive API documentation. <https://docs.nvidia.com/cuda/cuda-driver-api/index.html>. Accessed 01 Mar 2022
19. CUDA Runtime API documentation. <https://docs.nvidia.com/cuda/cuda-runtime-api/index.html>. Accessed 01 Mar 2022
20. Cheng, J.: *Professional CUDA C Programming*. John Wiley & Sons, Hoboken (2014)
21. Sanders, J., Kandrot, E.: *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Pearson Education, Boston (2010)
22. Help center for MATLAB: Simulink and other MathWorks product. [https://uk.mathworks.com/help/index.html?s\\_tid=CRUX\\_lftnav](https://uk.mathworks.com/help/index.html?s_tid=CRUX_lftnav). Accessed 01 Mar 2022