Structures 8 (2016) 111–120

Contents lists available at ScienceDirect

Structures

journal homepage: www.elsevier.com/locate/structures

Vectorised graphics processing unit accelerated dynamic relaxation for bar and beam elements

Andrew Liew*, Tom Van Mele, Philippe Block

Institute of Technology in Architecture, Block Research Group, ETH Zurich

ARTICLE INFO

Article history:
Received 22 June 2016
Received in revised form 29 August 2016
Accepted 6 September 2016
Available online 9 September 2016

Keywords:
Dynamic relaxation
Graphics processing unit
GPU
Vectorised
Bar
Beam

ABSTRACT

Dynamic relaxation is an analysis method that can be used to find the static equilibrium state of a structure undergoing large displacements. Due to the algorithm's iterative nature, which derives from a damped dynamic oscillations approach, it can take considerable time when solving models with dense node counts. This research looks at a vectorised dynamic relaxation implementation, executed on a graphics card's processing unit (GPU) for the most computationally demanding calculations, and compared to traditional computations on the central processing unit (CPU), to determine if time savings can be found with the change in processor type. The programming language Python and module PyCUDA are utilised on a desktop graphics card with over two thousand shader cores, and compared to a quad-core processor using the same code layout. A vectorised formulation for bar elements and reduced degree-of-freedom beam elements is used, to analyse the absolute and relative run-times for three benchmark cases. It is found that for models with low node counts, the CPU is faster, while for medium to highly dense models, the GPU acceleration can reduce absolute run-times by a factor of three to just over eight. Most benefit is gained when the executed code allows the GPU cores to be overloaded, such that complex calculations involving large arrays of data can be spread over the many cores. This was evident in the GPU load demand, which increased steadily with increasing node count but remained below 100%, while the CPU was readily activated to maximum load for even simple models. The time savings are of interest to architects, engineers and researchers utilising the dynamic relaxation method with medium to complex models, as the benefits can be harnessed with minimal code alteration.

1. Introduction

Dynamic relaxation (DR) [1,2] is a dynamic analysis procedure where one is interested in the final rest state of the structure, that is, the static solution after the dynamic oscillations have been sufficiently damped. The structural system is considered to be in equilibrium after a convergence criterion is satisfied, for example, when the balance of internal and external forces is suitably close relative to some tolerance. The initial geometry of the structure need not be close to the final rest state, and so the process may be initialised from an approximate geometry and found after the external loads are instantaneously applied. The method is particularly adept at solving large displacement problems and has found application with analysing assemblies of cable, bending and membrane elements, and is used extensively for tensile structures [3]. There are various examples in the literature of using DR to analyse beam elements. Work related to bending elements stabilised by membranes can be found in [4]; in [5], six degree-of-freedom elements were used to analyse the erection and loading of double-layered timber beams for grid-shells, with an optimisation method for variable cross-section depths; tubular glass-fibre-reinforced grid-shells have been investigated by [6], and [7] looked at bending elements supported by cables and/or cable-nets. Although membrane elements are not investigated in this research, constant strain triangular elements could be used, as formulations in the literature describe the behaviour of the membrane element via the edges of the faces to give an equivalent cable-net system. Therefore by creating a relationship between membrane stresses and edge link tensile forces, the methods described here-in for net structures may be extended readily for application to membrane elements. Further literature on membrane elements in the dynamic relaxation method can be found in [2,8,9] and [10].

* Corresponding author at: Institute of Technology in Architecture, HIB E 46, Block Research Group, ETH Zurich.
E-mail addresses: liew@arch.ethz.ch (A. Liew), van.mele@arch.ethz.ch (T. Van Mele), block@arch.ethz.ch (P. Block).

http://dx.doi.org/10.1016/j.istruc.2016.09.002
2352-0124/

Dynamic relaxation can be grouped alongside other methods such as the Force Density Method [11,12], Particle–Spring systems [13] and other form-finding methods, with comparisons detailed in [14]. The use of DR has an advantage over other form-finding methods, in that engineering material properties can explicitly be included in the analysis, permitting the inclusion and extraction of structural forces, stresses and displacements. DR also possesses advantages over standard analysis methods based on Newton–Raphson solvers and stiffness matrix formulations, which can have stability problems in addition to longer run-times [15]. The DR method also has the advantage of not needing to assemble a full stiffness matrix. Difficulties with stiffness matrix methods can be particularly prevalent with the large displacements involved in form-finding, such procedures are better suited for calculating forces, stresses and displacements at working loads when the initial geometry is already known or the displacements are small.

For numerical analysis, computations may be executed on the central processing unit (CPU) or the graphics processing unit (GPU). The decision as to which is faster for a particular numerical task, depends on whether fewer cores at a higher clock-rate (CPU based) performs superior than many hundreds or thousands of cores at a lower clock rate (GPU based). Often, the former can be better suited for programming code that runs in a linear manner, making best use of the higher clock-rate of the CPU, while the latter can achieve significant speed gains when the code is formulated, or the calculations distributed, in a parallel format. This is the case when the computations involve similar repetitive operations, and the order that those operations are performed does not matter, for example, squaring the first n integers does not require pre-requisite knowledge of other calculation results to determine its own, however for calculating the first n Fibonacci numbers this is not the case. Investigations were conducted by [9] for the DR analysis of membrane structures using constant strain triangular elements, to determine the potential speed improvements from various parallel processing strategies for multiple processors. Attention was drawn to the calculation of the residual forces, which accounted for 70–80% of the time taken by the integration scheme. The greatest speed improvements were found for denser meshes, highlighting efficiencies for more complex models.

In the use of graphics cards for video games and visualisation, the near real-time processing of large data-sets is important, such as with handling the rendering of particles, meshes, lighting and physics simulations. The processing of large volumes of data in such a rapid manner, can be carried over to scientific computing with interfaces that allow the programmer to access the graphics processing unit for setting up various calculation types, not just graphics based tasks. GPU computing has been applied to a diverse range of mathematical and scientific disciplines, including molecular biology, cryptography, neural networks, computational finance, astrophysics and climate research, to name only a few. This research looks at the application of GPU programming to the dynamic relaxation method, to determine whether savings can be made in the time taken to compute the static equilibrium of structures. Improvements in the analysis time, which can be significant, has practical application in the engineering design of structural forms involving cables, membranes and beam elements.

2. GPU programming

This section presents in Section 2.1 an overview of the NVIDIA CUDA implementation for GPU computing within PyCUDA for the programming language Python; hardware differences between CPU and GPU configurations in Section 2.2, including a description of that used in this research; and finally in Section 2.3, an introduction to vectorised programming for efficient computational analysis.

2.1. CUDA and PyCUDA

CUDA is a parallel computing platform created by NVIDIA for use with their CUDA-enabled GPUs [16]. This research utilises the CUDA API access by PyCUDA [17] for coding in the Python scripting language [18]. Similar procedures to those that will be outlined can be used with other environments such as CUDA for MATLAB [19] and OpenCL [20], where the latter does not require NVIDIA chip-sets. The CUDA Toolkit version 7.5.18 is used with the GeForce 364.51 drivers, both direct from NVIDIA.

Some of the basic operations when interfacing with the GPU in PyCUDA are as follows. Data `B` may be sent from the motherboard memory to the video memory as a GPUArray `A`, with `A = pycuda.gpuarray.to_gpu(B)`, `A` can then be collected from video memory with `B = A.get()`. This gives the option to send GPUArrays and then compute the most computationally expensive routines on the graphics card, as once an operation is called to act on GPUArrays, it is automatically performed with the GPU. The programming code must utilise functions that are GPU ready. Standard element-wise arithmetic operations on GPUArrays can be processed as normal, such as `A = B/C + C*C`, as can trigonometric and other basic operations such as `pycuda.cumath.sin()`, `pycuda.cumath.acos()` and `pycuda.cumath.sqrt()`. However, for more complex tasks, not all standard technical computing functions are immediately available, and so this will involve manufacturing functions to best utilise the available simpler functions, such as `pycuda.gpuarray.sum()`, as well as controlling carefully array sizes with `pycuda.gpuarray.zeros()`, `pycuda.gpuarray.reshape()` and `pycuda.gpuarray.ravel()`. For this research, only linear algebra operations were needed, using additional routines found in SCIKIT-CUDA [21] such as `skcuda.linalg.dot()` and `skcuda.linalg.transpose()`. It was necessary to construct a custom kernel for calculating the cross-product of a matrix of vectors, as the required function was not directly available in either PyCUDA or SCIKIT-CUDA. An overview of the programming and hardware setup is shown in (Fig. 1).

It can be necessary to consider what data are sent to the limited video memory, particularly when manipulating large arrays of 64 bit floats with double-precision. The accuracy needed in the numerical analysis, particularly for engineering application, should be considered for efficient video memory management. For example, a 5000×5000 square array of 64 bit double-precision takes 200 MegaBytes (MB) of memory, as opposed to the same matrix in 32 bit single-precision taking 100 MB of memory space. If memory issues occur, it may be necessary to perform the calculations on smaller sub-arrays, and then to reconstruct the results afterwards. It is preferred to limit the number of times data are transferred over to and from the video memory, as the information is sent over the relatively slower PCI Express Bus. It is more effective to construct and send the necessary arrays to the video memory before the initiation of the main computations, and then to retrieve the results after the computations have finished, rather than having intermediate data exchanges. For the same reason it is preferable, if the necessary function is available, to construct the GPUArrays directly on the video memory rather than initialising and then sending the arrays.

2.2. Hardware

To-date, a high-performance consumer desktop CPU will have a clock-speed of between 3000–4500 MegaHertz (MHz) over four to eight physical cores, with additional virtual cores available through the enabling of hyper-threading technology. A high-end GPU equivalent will be within the range of 900–1300 MHz over 384–2048 shader cores, with a variety of professional NVIDIA workstation Quadro graphics cards currently containing up to 3072 shader cores.

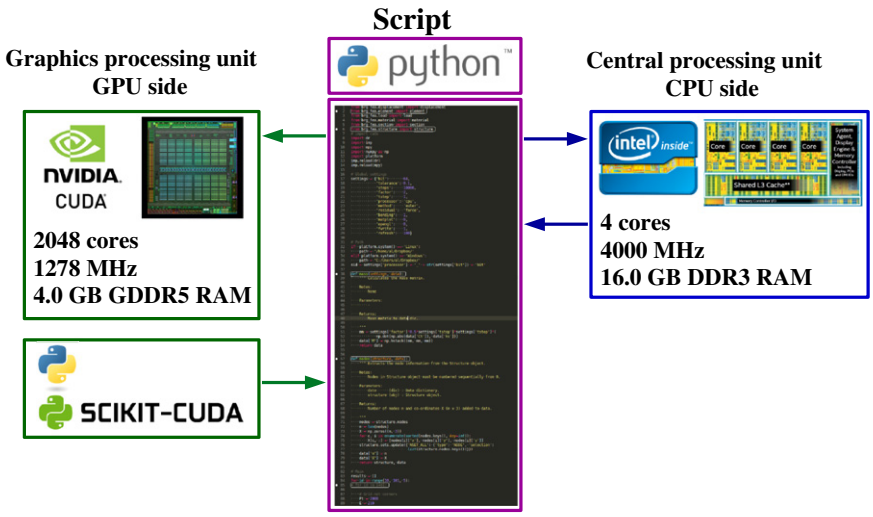

Fig. 1. Coding interface with the CPU, GPU and associated memory used in this investigation.

When working with the data arrays, the CPU will store the information on the motherboard's random-access-memory (RAM) which typically has a capacity of up to 32–64 Gigabytes (GB). When working with the GPU, data are stored on the graphics card's on-board video memory, which typically has a capacity of 2–12 GB. Consumer motherboards may also allow the chaining together of two, three or four graphics cards to utilise multiple GPU chips and video memory.

The hardware used in this investigation is as follows. The NVIDIA GPU graphics card is a Zotac GTX 980, possessing 4096 MB of DDR5 memory, with PCI Express ×16 Bus, a clock-rate of 1278 MHz over 2048 cores, and with a Compute Capability of 5.2 (a measure of the hardware architecture affecting the functions the card can perform). The CPU is an Intel Core i5-4690 K running at a clock-rate of 4000 MHz over four physical cores and with 16 GB of available DDR3 memory. The maximum blocksize for the GPU is 1024, which indicates that 340 count of 3-dimensional vectors ($340 \times 3 = 1020$) can be worked on per block. Therefore multiple blocks are arranged in a grid for the benchmarks seen later that utilise larger numbers of 3D vectors in the more complex models. Using electronics pricing at the time of writing, the CPU and memory can be purchased at a price of 290 EUR, whilst the graphics card will cost 390 EUR, highlighting little financial difference between the test components for general consumer computer parts. Higher-end professional Intel Xeon processors with 10 or 12 cores are priced at 1300–2300 EUR, compared to NVIDIA Tesla cards (K20, K40 and K80) which posses between 2496 and 4992 cores, the price ranges between 1800 EUR and 5000 EUR.

2.3. Vectorised programming

The principle of vectorised programming is to perform numerical computations on arrays representing vectors and matrices, rather than many repetitive scalar operations on individual elements. Such a programming philosophy is an effective way to make good use of multiple processing cores, as they can be overloaded (loaded to 100%), so that a large volume of data can be processed with arrays of hundreds and thousands of elements. When implemented, this involves grouping the arrays that take part in the operation, so that vectors and matrices are of the same size, and then performing calculations in one step. Such a procedure leads to code with a minimal (or sometimes zero) number of loop statements. This concept may be further explained with an example. Suppose we wish to compute $z = cos\,x \sin y$ for $x = [0, 10]$ and $y = [0, 10]$ in steps of 1 for x and y. A purposefully inefficient non-vectorised Python and NumPy programming layout could be:

```
import numpy as np

n = 11
z = np.zeros((n, n))
for x in range(n):
    for y in range(n):
        z[x, y] = np.cos(x) * np.sin(y)
```

The nested loop statement will invoke the inner `np.cos(x) * np.sin(y)` calculation 121 times, with each iteration occurring in sequence one after the other. This is highly undesirable for the further embedding of loop statements, as well as for many thousands of cycles. Instead, it is possible to use the following code as NumPy functions `np.cos()` and `np.sin()` accept vectors or matrices as inputs:

```
import numpy as np

r = range(11)
[x, y] = np.meshgrid(r, r)
z = np.cos(x) * np.sin(y)
```

This places the final result directly in an array `z`, which has size (11×11), without the use of any loop statements. This form of coding is efficient with respect to calculation time and resource use, and needs to be utilised when harnessing the benefits of GPU acceleration, where spreading the computation over many processing cores is needed. The `np.meshgrid()` function is used to create arrays `x` and `y` which are of size (11×11), so that the element by element multiplication can be performed. This is a characteristic of vectorised programming, in that in order to maintain consistent vector or matrix array sizes, some variables may need to be repeated in a larger array size through tiling or reshaping.

For the notation in this paper, the multiplication of matrices and vectors follows standard matrix product notation, such that **Ab**, where **A** is $(m \times m)$ and **b** is $(m \times n)$, will give a matrix of size $(m \times n)$. For element by element operations, the Hadamard (or Schur) product notation is used, where $(\mathbf{A} \circ \mathbf{B})_{i,j} = \mathbf{A}_{i,j}\mathbf{B}_{i,j}$ is the element by element multiplication of two matrices **A** and **B** of the same size. For division, the following notation is used: $(\mathbf{A} \oslash \mathbf{B})_{i,j} = \mathbf{A}_{i,j}/\mathbf{B}_{i,j}$.

3. Dynamic relaxation

The dynamic relaxation method involves tracing the motion of structural nodes for small discrete time steps, until the nodes come to rest, and the structure is in a state of static equilibrium. When the external loads are applied at the start of the analysis, the structure undergoes dynamic motion until eventually the displacements settle and velocities tend to zero, after energy is extracted from the system through the damping method. This section describes this process in more detail and presents the elements used in this research, and how the entire method may be organised to fit the vectorised programming framework needed for GPU processing.

3.1. Governing equations

The equation of motion for time t with a system of discrete masses at node i for n number of nodes, is given by Eq. (1). This is represented (with matrix and vector sizes in brackets) with mass matrix $\bar{\mathbf{M}}$ $(n \times n)$ with point masses along the diagonal at $\bar{\mathbf{M}}_{i,i}$, viscous damping matrix $\bar{\mathbf{C}}$ $(n \times n)$, stiffness matrix $\bar{\mathbf{K}}$ $(n \times n)$, displacement vector $\mathbf{x}$ $(n \times 1)$, velocity vector for the x direction $\mathbf{v}_x$ $(n \times 1)$ and subject to the forcing vector for the x direction $\mathbf{p}_x$ $(n \times 1)$.

$$\bar{\mathbf{M}}\frac{\mathrm{d}\mathbf{v}_x}{\mathrm{d}t} + \bar{\mathbf{C}}\mathbf{v}_x + \bar{\mathbf{K}}\mathbf{x} = \mathbf{p}_x. \tag{1}$$

If a residual vector for the x direction is defined as $\mathbf{r}_x = \mathbf{p}_x - \bar{\mathbf{K}}\mathbf{x}$ to represent the out-of-balance forces, then

$$\bar{\mathbf{M}}\frac{\mathrm{d}\mathbf{v}_x}{\mathrm{d}t} + \bar{\mathbf{C}}\mathbf{v}_x = \mathbf{r}_x. \tag{2}$$

The viscous damping term, which represents the resistance to motion, depends on the degree of viscous damping and the nodal velocities, and can be replaced by a process of kinetic damping as will be explained in Section 3.2, and so the $\bar{\mathbf{C}}\mathbf{v}_x$ term can be removed to give

$$\bar{\mathbf{M}}\frac{\mathrm{d}\mathbf{v}_x}{\mathrm{d}t} = \mathbf{r}_x. \tag{3}$$

As matrix $\bar{\mathbf{M}}$ is diagonal for discrete masses at each node, it may be written instead as a vector **m** and multiplied element by element to the acceleration vector to give

$$\mathbf{m} \circ \frac{\mathrm{d}\mathbf{v}_x}{\mathrm{d}t} = \mathbf{r}_x. \tag{4}$$

Utilising an explicit integration scheme from [2], the following central-finite-difference approximation for time t and for time-step Δt, leads to Eq. (4) being approximated by

$$\mathbf{m} \circ \frac{\mathbf{v}_x^{t+\Delta t/2} - \mathbf{v}_x^{t-\Delta t/2}}{\Delta t} = \mathbf{r}_x^t. \tag{5}$$

Rearranging this gives

$$\mathbf{v}_x^{t+\Delta t/2} = \mathbf{v}_x^{t-\Delta t/2} + \Delta t \mathbf{r}_x^t \oslash \mathbf{m}. \tag{6}$$

The new displacement vector at time $t + \Delta t$ is

$$\mathbf{x}^{t+\Delta t} = \mathbf{x}^t + \Delta t \mathbf{v}_x^{t+\Delta t/2}. \tag{7}$$

Similar equations can be constructed for the y and z directions, and can be stacked horizontally in the matrices $\mathbf{X} = [\mathbf{xyz}]$ and $\mathbf{V} = [\mathbf{v}_x\, \mathbf{v}_y\, \mathbf{v}_z]$, which are of size $(n \times 3)$, so that

$$\mathbf{X}^{t+\Delta t} = \mathbf{X}^t + \Delta t \mathbf{V}^{t+\Delta t/2}. \tag{8}$$

With the residual forces now as $\mathbf{R} = [\mathbf{r}_x\, \mathbf{r}_y\, \mathbf{r}_z]$, the external loads as $\mathbf{P} = [\mathbf{p}_x\, \mathbf{p}_y\, \mathbf{p}_z]$ and the mass vector tiled horizontally into a matrix $\mathbf{M} = [\mathbf{mmm}]$, then the updated velocities are

$$\mathbf{V}^{t+\Delta t/2} = \mathbf{V}^{t-\Delta t/2} + \Delta t \mathbf{R}^t \oslash \mathbf{M}. \tag{9}$$

3.2. Kinetic damping

In a dynamic system, passing through a local minimum potential energy state is associated with a local maximum kinetic energy, for example a swinging pendulum passing through its vertical orientation has maximum kinetic energy and minimum gravitational potential energy. For kinetic damping in the DR, the kinetic energy of the system is tracked at each time step, and the nodal velocities are set to zero on the detection of a local peak energy. The kinetic energy peaks then decline with time as energy is extracted from the system and it approaches a final rest state. The process then restarts with the kinetic energy U as zero, as all of the nodal velocities in **V** are zero. A typical plot showing the decay of a system's kinetic energy and the out-of-balance residual forces can be seen in Fig. 2, for one of the benchmarks that will be discussed later in Section 4 (specifically this is the grid-net 20 × 20 model with $E = 5$ GPa). So for the kinetic energy

$$U = \frac{1}{2}\sum \mathbf{M} \circ \mathbf{V} \circ \mathbf{V}, \tag{10}$$

when the following is satisfied

$$U^{t+\Delta t/2} < U^{t-\Delta t/2} \tag{11}$$

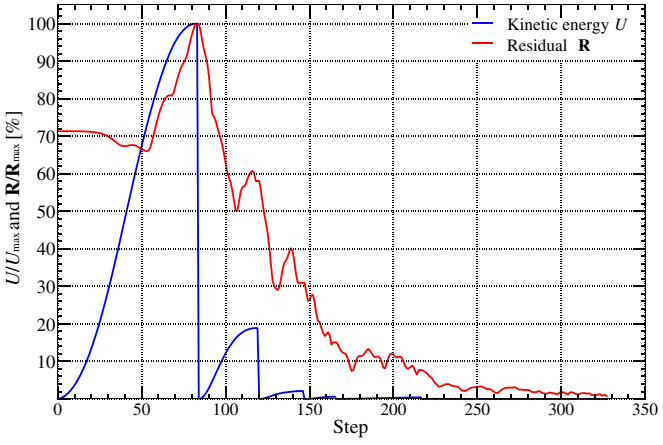

Fig. 2. Typical decay of a system's kinetic energy U and reduction of residuals **R** with increasing time-steps.

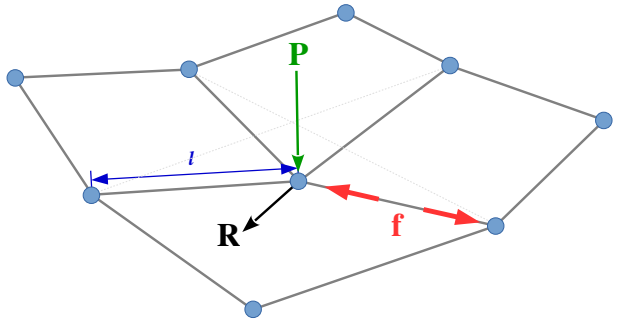

Fig. 3. Bar elements with internal forces **f**, applied nodal loads **P**, lengths **l** and out-of-balance residuals **R**.

the velocities are reset to zero. Kinetic energy damping is used in this research as it has been found to offer a faster and more stable analysis when compared to viscous damping, and is simpler to implement as it does not require intermediate steps to calculate optimal damping coefficients [15,22,23]. Further improvements to DR kinetic damping may be possible by the use of adaptive mass matrix calculations [24]. Here, the mass matrix **M** is not continuously re-calculated throughout the step cycles for computational efficiency, such a re-calculation after each kinetic energy restart may lead to a more stable analysis [22] (although with increased computation cost), but was not needed for the benchmarks described in Section 4.

3.3. Bar elements

Bar elements may transmit only axial forces, and are suited to modelling structural elements when the flexural stiffness is insignificant or not to be considered. These elements can be used to model cable structures with the dynamic relaxation method [25]. If the bar element length exceeds many times the cross-section dimensions, the compressive load carrying capacity may be neglected, as compression leads to buckling at even low loads. The following element description follows from [2], for bars that are assumed to be straight between end-nodes, with loading applied only directly at the node points (Fig. 3).

For m number of bar elements, where internal bending moments and shear forces are not considered, only the vector of axial forces $\mathbf{f}$ $(m \times 1)$ for the elements is calculated. For each new time-step this is given by

$$\mathbf{f}^{t+\Delta t} = \mathbf{f}_0 + \mathbf{E} \circ \mathbf{A} \circ \left(\mathbf{l}^{t+\Delta t} - \mathbf{l}_0\right) \oslash \mathbf{l}_0, \tag{12}$$

where the following vectors are all of size $(m \times 1)$: initial pre-load $\mathbf{f}_0$, Young's modulus $\mathbf{E}$, cross-section area $\mathbf{A}$, initial lengths $\mathbf{l}_0$ and lengths $\mathbf{l}^{t+\Delta t}$. Therefore $\mathbf{f}^{t+\Delta t}$ is based on lengths $\mathbf{l}^{t+\Delta t}$ calculated from the updated geometry $\mathbf{X}^{t+\Delta t}$. The areas $\mathbf{A}$ can be taken as constant for time t so long as the assumed engineering strains are similar to the true strains, so that the cross-section area change is negligible, which will likely be the case when the material is within its elastic range. When modelling a tie, the axial forces $\mathbf{f}$ can be set to zero for elements found to be in compression (slackness correction).

To calculate the updated residuals, we make use of a connectivity matrix $\mathbf{C}$ of size $(m \times n)$, to handle the connectivity between elements and nodes, so that row i (for element i), has two columns marked with 1 and −1 for the nodes at the two ends, where the node deemed the start node is not governing. The use of a connectivity matrix, or branch–node matrix, is familiar to the form-finding methods outlined in Section 1. From here, the distances between two ends of a bar in the x direction are the x co-ordinate differences $\mathbf{Cx}$. Once this is multiplied by the force in the bar and divided by the length to give $\mathbf{Cx} \circ \mathbf{f} \oslash \mathbf{l}$, one has the x components of the bar forces. The residual forces for the x direction are then

$$\mathbf{r}_x = \mathbf{p}_x - \mathbf{C}^{\mathrm{T}}(\mathbf{Cx} \circ \mathbf{f} \oslash \mathbf{l}). \tag{13}$$

Similar expressions can be obtained for the y and z co-ordinate directions to give the residuals $\mathbf{r}_y$ and $\mathbf{r}_z$. Alternatively, the calculation can be performed in one equation for all three directions with

$$\mathbf{R} = \mathbf{P} - \mathbf{C}^{\mathrm{T}}(\mathbf{CX} \circ \mathbf{F} \oslash \mathbf{L}), \tag{14}$$

if $\mathbf{f}$ and $\mathbf{l}$ are tiled horizontally into $(m \times 3)$ arrays such that $\mathbf{F} = [\mathbf{fff}]$ and $\mathbf{L} = [\mathbf{lll}]$.

For the mass vector $\mathbf{m}$, which makes up the mass matrix $\mathbf{M}$, the axial stiffness at a node can be taken as the sum of the maximum axial stiffness for all of the elements connected to that node, this is

$$\mathbf{m} = \frac{1}{2}\Delta t^2 \left|\mathbf{C}^{\mathrm{T}}\right| (\mathbf{E} \circ \mathbf{A} \oslash \mathbf{l_0} + \mathbf{f} \oslash \mathbf{l}), \tag{15}$$

which takes the full unresolved axial stiffness for all three co-ordinate directions $\mathbf{m} = \mathbf{m}_x = \mathbf{m}_y = \mathbf{m}_z$. This has been shown to be satisfactory when compared to evaluating different mass vectors for each co-ordinate direction. In the absence of an adaptive time-stepping regime, Δt may be taken as constant, conveniently as unity. The mass matrix $\mathbf{M}$ need not be accurately representative of the actual nodal masses, as it is only important for the dynamic portion of the analysis and may be chosen for convergence speed and stability.

3.4. Beam elements

Beam elements can be bent from an initial state (which is often flat), into a final configuration such as a grid-shell, that is subsequently subject to design loads. These elements must be flexible enough to bend into shape, yet strong enough to not fracture. Such elements in construction are typically fabricated from wood, glass/carbon fibre reinforced polymers or aluminium tubes and can be stabilised laterally by other elements such as cables or membranes, or other beam elements.

In finite element formulations for beam elements in 3D space, each node is generally characterised by six degrees-of-freedom, three translational and three rotational. Reducing the degrees-of-freedom may lead to improvements in computation times. The following beam element description uses the reduced three degree-of-freedom formulation from [26] which, as will be seen in the following equations, naturally follows a path to be vectorised when programming for the dynamic relaxation method.

For a beam represented by n number of nodes, the position vector of internal node i (where the internal nodes are $[2, n-1]$) is $\mathbf{q}_i$, along with the nodes located at position vectors $\mathbf{q}_{i-1}$ and $\mathbf{q}_{i+1}$, all lay on the same plane (Fig. 4). The vectors $\mathbf{q}_a = \mathbf{q}_i - \mathbf{q}_{i-1}$, $\mathbf{q}_b = \mathbf{q}_{i+1} - \mathbf{q}_i$ and $\mathbf{q}_c = \mathbf{q}_{i+1} - \mathbf{q}_{i-1}$ represent the local vectors between the nodes. These vectors can be created for all of the internal nodes and stored in arrays of

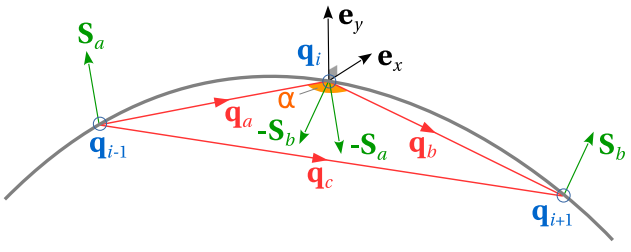

Fig. 4. Notation for beam elements: shear vectors $\mathbf{S}_a$ and $\mathbf{S}_b$, position vectors $\mathbf{q}_{i-1}$, $\mathbf{q}_i$ and $\mathbf{q}_{i+1}$, local vectors $\mathbf{q}_a$, $\mathbf{q}_b$ and $\mathbf{q}_c$, local co-ordinate vectors $\mathbf{e}_x$ and $\mathbf{e}_y$, and angle α.

size $(n-2)\times 3$, by slicing the global nodal co-ordinate matrix $\mathbf{X}$ with appropriate start and end points (these are $[1, n-2]$, $[2, n-1]$ and $[3, n]$). This gives $\mathbf{Q}_{i-1}$, $\mathbf{Q}_i$, and $\mathbf{Q}_{i+1}$, which in turn gives

$$\mathbf{Q}_a = \mathbf{Q}_i - \mathbf{Q}_{i-1} \tag{16}$$

$$\mathbf{Q}_b = \mathbf{Q}_{i+1} - \mathbf{Q}_i \tag{17}$$

$$\mathbf{Q}_c = \mathbf{Q}_{i+1} - \mathbf{Q}_{i-1}. \tag{18}$$

The array of unit vectors representing the cross-section orientations for the internal nodes are $\mathbf{e}_x$, $\mathbf{e}_y$ and $\mathbf{e}_z$ and are again of size $(n-2)\times 3$.

The curvature vector $\boldsymbol{\kappa}$ is introduced to store the curvatures at each internal node, assuming a circular arc passing through the three local nodes and contained within that plane, and is given by

$$\boldsymbol{\kappa} = 2\sin\boldsymbol{\alpha} \oslash |\mathbf{Q}_c| \qquad \text{with,} \tag{19}$$

$$\cos\boldsymbol{\alpha} = \left(|\mathbf{Q}_a|^2 + |\mathbf{Q}_b|^2 - |\mathbf{Q}_c|^2\right) \oslash \left(2\,|\mathbf{Q}_a| \circ |\mathbf{Q}_b|\right). \tag{20}$$

This is easily computed in vectorised form as all $|\mathbf{Q}|$ terms have the same array size, and the trigonometric sin, arccos and vector norm functions are commonly implemented to take vector input, as is the case with NumPy and PyCUDA.

A matrix of curvature vectors for the internal nodes, of size $(n-2)\times 3$, can be set-up such that each vector (each row in the array) is orthogonal to the plane containing the three local nodes,

$$\boldsymbol{K} = \boldsymbol{\kappa}' \circ (\mathbf{Q}_a \times \mathbf{Q}_b) \oslash |\mathbf{Q}_a \times \mathbf{Q}_b|'. \tag{21}$$

Here, the notation $\boldsymbol{\kappa}'$ indicates that $\boldsymbol{\kappa}$ of size $(n-2)\times 1$ is tiled horizontally to $\boldsymbol{\kappa}' = [\boldsymbol{\kappa}\boldsymbol{\kappa}\boldsymbol{\kappa}]$ of size $(n-2)\times 3$ for array size compatibility, similarly for $|\mathbf{Q}_a \times \mathbf{Q}_b|'$. The calculation of $\mathbf{Q}_a \times \mathbf{Q}_b$ is easily performed on arrays of vectors using function `numpy.cross(a,b,axis)` in NumPy. A custom cross-product PyCUDA kernel was used for this operation on the GPU.

The components representing the x and y curvature vectors

$$\boldsymbol{K}_x = (\boldsymbol{K} \cdot \mathbf{e}_x)' \circ \mathbf{e}_x \tag{22}$$

$$\boldsymbol{K}_y = (\boldsymbol{K} \cdot \mathbf{e}_y)' \circ \mathbf{e}_y, \tag{23}$$

where the dot products $\boldsymbol{K}\cdot\mathbf{e}_x$ and $\boldsymbol{K}\cdot\mathbf{e}_y$ are performed using arrays of vectors, i.e. each row of the curvature array represents a vector where the dot product operation is performed with the corresponding row in $\mathbf{e}_x$ or $\mathbf{e}_y$. Both $\boldsymbol{K}\cdot\mathbf{e}_x$ and $\boldsymbol{K}\cdot\mathbf{e}_y$ are tiled horizontally three times to match the sizes of $\mathbf{e}_x$ and $\mathbf{e}_y$, as the dot product operation will collapse one dimension.

The combined moment array is

$$\mathbf{M}_C = \mathbf{EI}'_x \circ (\boldsymbol{K}_x - \boldsymbol{K}_{x0}) + \mathbf{EI}'_y \circ (\boldsymbol{K}_y - \boldsymbol{K}_{y0}) + \mathbf{M}_T, \tag{24}$$

where $\mathbf{EI}_x$ and $\mathbf{EI}_y$ are the flexural stiffnesses about local co-ordinate axes $\mathbf{e}_x$ and $\mathbf{e}_y$; $\boldsymbol{K}_{x0}$ and $\boldsymbol{K}_{y0}$ are arrays of initial curvature vectors, which can be taken as zero if the beam is initially straight and unstrained, and $\mathbf{M}_T$ is the matrix of twisting vectors, which is taken as zero for this research, indicating no torsion.

To calculate the corresponding shear forces, first the following are defined,

$$\mathbf{u}_a = (\mathbf{M}_C \times \mathbf{Q}_a) \oslash |\mathbf{M}_C \times \mathbf{Q}_a|' \tag{25}$$

$$\mathbf{u}_b = (\mathbf{M}_C \times \mathbf{Q}_b) \oslash |\mathbf{M}_C \times \mathbf{Q}_b|'. \tag{26}$$

Using $\mathbf{c}_a = \mathbf{Q}_a \times \mathbf{u}_a$ and $\mathbf{c}_b = \mathbf{Q}_b \times \mathbf{u}_b$, the array of shear vectors is given by

$$\mathbf{S}_a = \mathbf{u}_a \circ \left[\left(|\mathbf{M}_C|^2 \circ |\mathbf{c}_a|\right) \oslash \left(|\mathbf{Q}_a| \circ (\mathbf{M}_C \cdot \mathbf{c}_a)\right)\right]' \tag{27}$$

$$\mathbf{S}_b = \mathbf{u}_b \circ \left[\left(|\mathbf{M}_C|^2 \circ |\mathbf{c}_b|\right) \oslash \left(|\mathbf{Q}_b| \circ (\mathbf{M}_C \cdot \mathbf{c}_b)\right)\right]'. \tag{28}$$

These shear force vectors represent the forces at the ends of each element, and must be super-imposed for the whole length of the beam. This will give the complete array of shear force vectors $\mathbf{S}$ $(n\times 3)$, which is then combined with the external point loads $\mathbf{P}$, and the resolved bar axial forces, for the definition of the out-of-balance residual forces $\mathbf{R}$. When multiple beam elements are present, the calculations are performed by using an indexing matrix specific to that beam, representing the global node numbers of the beam's nodes.

The beam element formulation has been presented for a single beam. For a structure consisting of many beams it is not desirable to introduce a loop statement to perform the calculations on each beam in turn, as shown with the example in Section 2.3. This is because it is faster to perform calculations simultaneously in one operation for as many nodes as possible, not just for the nodes of a particular beam element. This could be rectified by the expansion of the arrays with a third dimension, i.e. using arrays of vectors of size $(n-2)\times 3\times m$ where m is the number of beams. However, such a fixed array size would not make it simple to analyse arrangements of beams that do not have a consistent number of nodes n, and so would only be suitable for regular grids of beams. Alternatively, and as chosen here for adaptability, each array of vectors for a beam is instead stacked vertically under another beam, preserving a 2D array shape. This involves minimal code alteration, only that to produce indexing arrays (which are one dimensional) to represent the beams as if they were all chained together in one continuous line. Once the indexing has completed, the calculations can proceed with no node inter-dependence, as each row (each vector) does not depend on any other row or beam. The shear forces are then assigned at the final stage to the correct nodes with the indexing arrays.

4. Benchmarks

Three benchmark structures are investigated to assess how the computation speed of the described dynamic relaxation regime differs between a CPU or GPU driven analysis. The code for each can be considered identical, only differing by the relevant CPU or GPU functions that are called. A tolerance on the mean of the residual forces for all of the nodes is divided by the applied loading or pre-stress force, and used as the convergence criterion to indicate static equilibrium. A time-step of unity was kept constant throughout all benchmarks. In Section 4.1 a grid-net of ties is described to assess the performance of cable elements, in Section 4.2 a cylindrical-net subject to pre-stress and no external load, and in Section 4.3 a series of parallel simply-supported beam elements is presented.

4.1. Grid-net

An initially flat grid-net is vertically (upwards) loaded via the nodes with a distributed area load of magnitude 3.0 kN/m^2 (Fig. 5). The grid has outer dimensions of 1 m ×1 m and was discretised equally in both directions with between 10 and 100 nodes along each side, with the outer boundary nodes of the grid fixed in all three co-ordinate directions. This lead to a node count range between 100 and 10000 nodes and between 300 and 30000 degrees-of-freedom (minus the border boundary constraints). The cable elements are

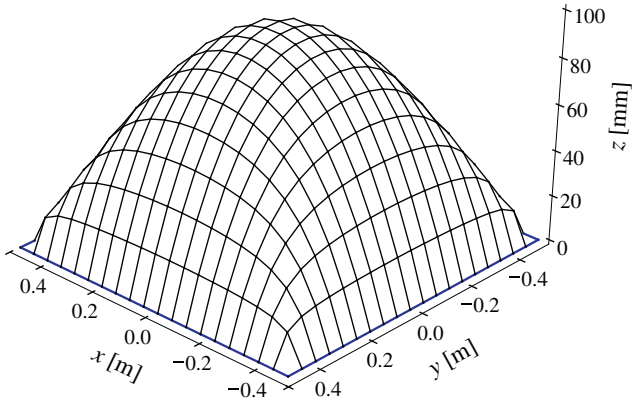

Fig. 5. Final deformed grid-net for the 20 × 20 layout with $E = 5$ GPa, translational restraints around the blue border.

rectangular solid sections, with width 1 mm and height 1 mm, and with three different Young's modulus values investigated, 5 GPa, 70 GPa and 210 GPa, to see the effect of varying the element stiffness. The six analyses are named as CPU_5, CPU_70 and CPU_210 for the CPU analyses and GPU_5, GPU_70 and GPU_210 for the GPU analyses. The cross-section flexural stiffness is not included in this benchmark, neither is there any pre-stress. A residual tolerance of 1% was used for the out-of-balance residual forces normalised by the applied nodal loads. The grid-nets are initially flat and become taut under the application of load.

4.2. Pre-stressed cylinder-net

A cylindrical-net with diameter 1.0, height 1.0 and initially straight vertical sides, was pre-stressed with three values of element stiffness (Fig. 6). The number of divisions around the circumference is twice the number of divisions in the vertical direction, of which the latter varied from 5 to 70 giving 60 to 9730 nodes or 180 to 29190 degrees-of-freedom, minus the fixed nodal degrees-of-freedom at the top and bottom edges. As there are no externally applied loads at the nodes, the tolerance chosen was 0.1% of the out-of-balance residual forces divided by the initial pre-stress force of unity. The axial stiffness for the elements was taken as 1, 10 and 100, giving CPU_1, CPU_10 and CPU_100 as the CPU analyses and GPU_1, GPU_10 and GPU_100 as the GPU analyses. No flexural stiffness is included in this benchmark.

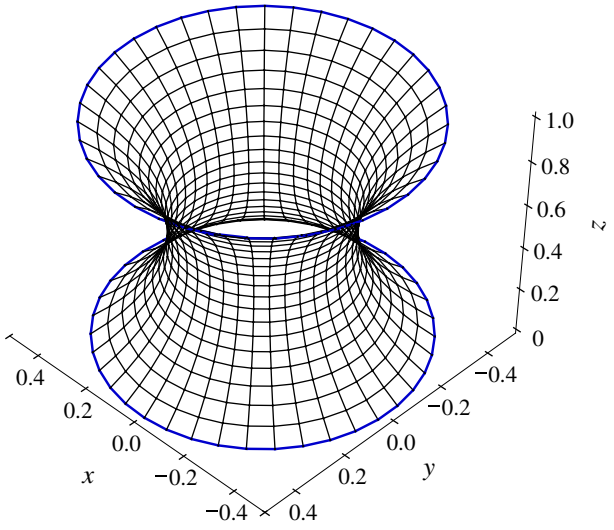

Fig. 6. Final deformed shape of cylinder-net for the 20 × 40 layout with $E = 1$, translational restraints around blue borders.

4.3. Parallel beams

A grid of equally spaced beams were arranged flat and unstressed within a 1 m × 1 m area and then vertically (upwards) loaded via the nodes with a distributed line load of magnitude 0.3 kN/m, as seen in Fig. 7 for pinned (blue) and roller (red) end conditions. The beam elements are circular hollow sections with diameter 10 mm, thickness 1 mm, and with a Young's modulus of 70 GPa. The beams are orientated along the x axis, start with an initial length of 1 m, and in the first case have a pinned connection at one end and a roller in the x direction at the other (pin-roller), and in another case pin connections at both ends (pin-pin). The number of divisions along the length of the beams is twice the number of beams spaced along the y axis, of which the latter was varied between 5 and 70, to give between 50 and 9800 nodes. This benchmark tests the flexural stiffness of the beams in the pin-roller case, and both the flexural and axial stiffness in the pin-pin case. The tolerance is satisfied when the mean of the out-of-balance residual forces, which are both the shear forces and axial forces, are less than 1% of the applied nodal loads.

5. Results

The results from the three benchmark cases are presented here in terms of absolute run-times in Section 5.1, relative run-times in Section 5.2, and with an examination on the demand on the processor and memory resources in Section 5.3.

5.1. Absolute run-times

For all three benchmarks in Fig. 8, Fig. 9 and Fig. 10, the absolute completion time in seconds is plotted against the number of nodes in each model. These figures plot time logarithmically on the y-axis to cater for the range and duration of completion times, with the CPU and GPU curves plotted in red and blue respectively. In Fig. 8, the stiffer grid-nets with $E = 210$ GPa and $E = 70$ GPa took longer to reach the residual tolerance than the 5 GPa cases. For the pre-stressed cylinder in Fig. 9, this was again the case, but with no observable difference in the run-times between the $E = 210$ GPa and $E = 70$ GPa models.

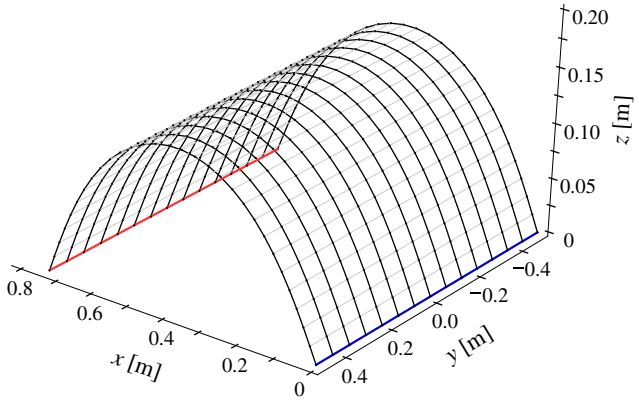

Fig. 7. Final deformed shape of the pin-roller beams benchmark for the 15 beams × 30 nodes layout.

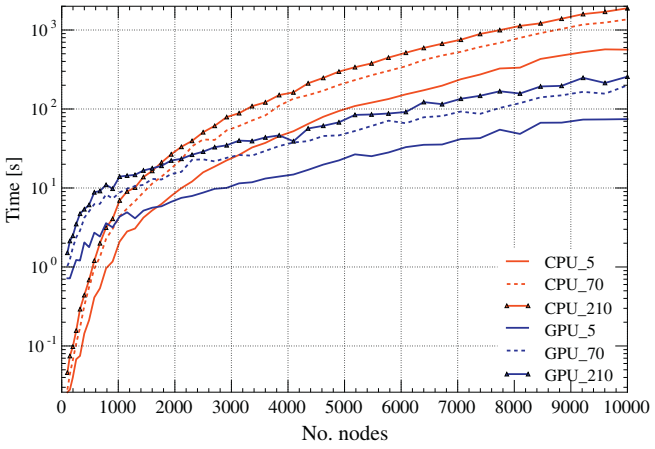

Fig. 8. Results of the grid-net benchmark, absolute completion times versus node count.

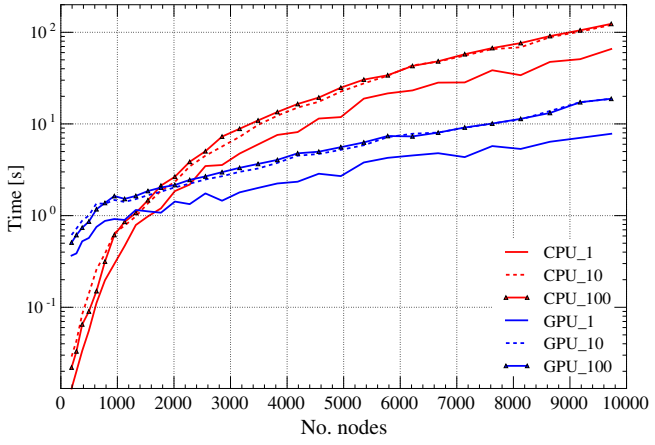

Fig. 9. Results of the pre-stressed cylinder-net benchmark, absolute completion times versus node count.

For the GPU and CPU analyses with the grid-net, for node counts less than 1750, which corresponds to a layout of around 42×42 nodes, the GPU calculations are completed overall slower than on the CPU. Thus for these calculations, which were of the order of a couple of seconds or minutes, it is disadvantageous to utilise the GPU.

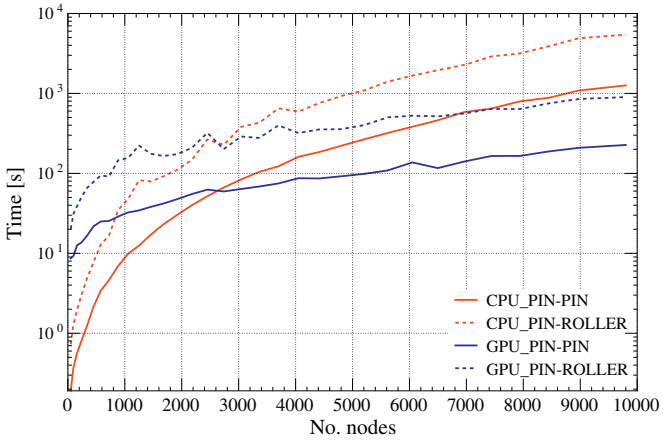

Fig. 10. Results of the parallel beams benchmark, absolute completion times versus node count.

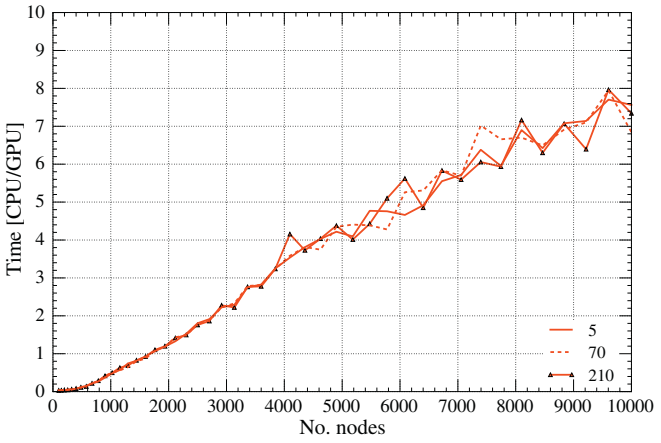

Fig. 11. Results of the grid-net benchmark, relative CPU to GPU completion time versus node count.

For the pre-stressed cylinder and parallel beams, this interface where both analysis types take the same time to compute, occurs between a node count of 1500 and 1750 for the cylinder and around 2500 for the parallel beams.

5.2. Relative run-times

The performance gains from the GPU are materialised after the cross-over transition point, which was found to be around 1500, 1750 and 2500 nodes for the three benchmark tests. Fig. 11, Fig. 12 and Fig. 13 divide the CPU time by the GPU time for the grid-net, pre-stressed cylinder-net and parallel beams respectively. In these figures, a linear increase in the relative CPU to GPU time can be seen with respect to increasing node count, across all three models. As the node count increases, the GPU analysis was up-to 8.5 times faster for the grid-net and cylinder-net and six times faster for the beam elements. There appears to be no indication that the speed gains favour either the stiffer or more flexible models, as all three curves are generally overlapping throughout the tested node range. There is also no conclusive evidence that the different boundary conditions for the beams benchmark was a significant factor, only a slight improvement for the pin-roller case at node counts above 7000.

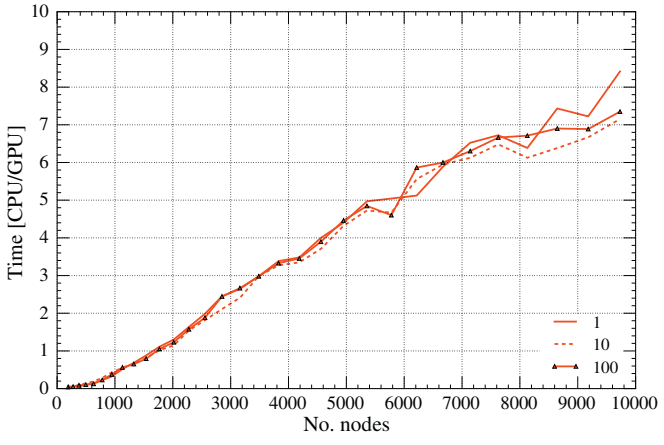

Fig. 12. Results of the pre-stressed cylinder-net benchmark, relative CPU to GPU completion time versus node count.

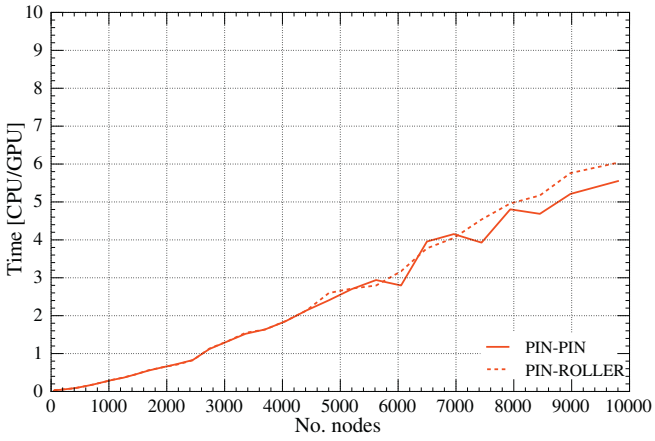

Fig. 13. Results of the parallel beams benchmark, relative CPU to GPU completion time versus node count.

5.3. Resource demand

Fig. 14 plots the CPU and GPU utilisation with respect to node count. A node count of zero defines the base-state of the test system before the start of the analysis. It is seen that the CPU is able to overload its four cores for models with a node count below 1500, and maintains a 100% working load for subsequent increases in the node count. In contrast, the most complex grid-net model with 10000 nodes and the densest cylinder-net model with 9730 nodes, managed a maximum GPU workload of 80 to 85% over the 2048 shader cores. For these two net benchmarks, the GPU was at a workload of less than 25% for when it was found to be slower than the CPU (less than 1750 nodes). There appears to be no clear distinction in the GPU utilisation between the cylinder-net and grid-net models.

To utilise the GPU to its full potential, calculations involving arrays of large size are needed to spread the load over many cores, to outweigh the lower clock-rate handicap. This condition appears to have been satisfied less so for the beam elements, as the benchmark results are slightly lower. This can be observed with the GPU utilisation staying below 60% for the beams model, and confirms why this benchmark obtained a maximum improvement over the CPU of around six times rather than just over eight. The reasons for this could be in the intermediate steps needed in the beam element analysis. Some of these steps involved array indexing and array tiling, and are less intensive on the processor when compared to the cross-product, dot-product and norm calculations, and so may lower the average GPU utilisation.

Fig. 15 plots the motherboard memory and the graphics card's on-board memory utilisation with respect to node count. The change in the memory utilisation for the CPU analyses is minimal (between 20% and 30%), given the available supply of DDR3 memory (16 GB). Sufficient memory was available for the GPU analyses, with the utilisation of the 4 GB memory falling between 10% to 70%, with the beam element benchmark taking more memory due to the additional arrays of vectors needed in intermediate calculations. However, for graphics cards with half of the available memory that was used here, the utilisation for the dense node counts, would in some instances reach 100%. It is therefore recommended not to compromise on memory capacity when many large arrays may be used. Solutions would be to break-down calculations into smaller sub-calculations, clearing memory and/or writing output at intermediate steps, or to use multiple graphics cards to stack the available memory.

6. Conclusions

Dynamic relaxation can be used to find the equilibrium state of a structure undergoing large deflections, by finding the rest state after oscillations have been dampened-out, for instance by kinetic damping. The method is iterative, with the convergence time depending on the type of damping, the applied loads and the chosen mass matrix. This research looked at executing a vectorised dynamic relaxation regime on both a central processing unit and graphics processing unit, to assess what benefits to the total analysis time could be achieved. This was carried out with the scripting language Python and modules PyCUDA and NumPy, comparing a modern desktop graphics card with just over two thousand shader cores, to a quad-core processor.

Constructing vectorised code that can execute on a GPU is not as straightforward as with running on the CPU. Consideration must be given beforehand, to the nature of the calculations, as standard numerical programming functions are not always available. Basic element-wise operations, linear algebra, standard trigonometric and mathematical functions can be used, or combined to make other more complex functions, but sometimes custom kernels may need to be constructed or alternatives found.

Three benchmarks with different sub-variables were analysed, examining axial stiffness, flexural stiffness and boundary conditions for bar and reduced degree-of-freedom beam elements. The grid-net models that were more axially flexible were quicker to analyse than their stiffer counter-parts, whilst for the beam models, the pin-roller case took significantly more time to analyse than the pin-pin case where axial tie forces acted in addition to the shear forces from flexure.

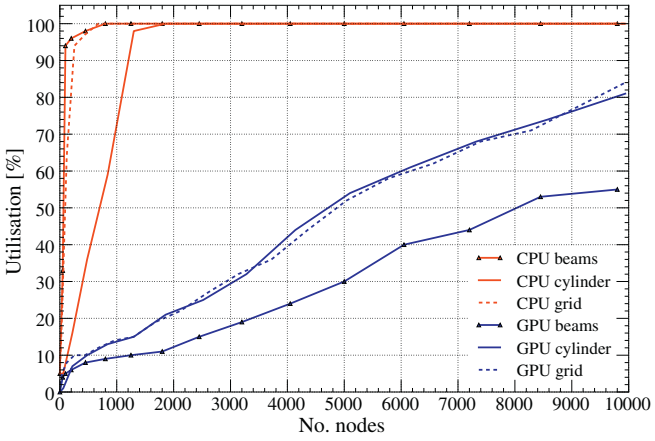

Fig. 14. CPU and GPU usage versus node count for the three benchmark tests.

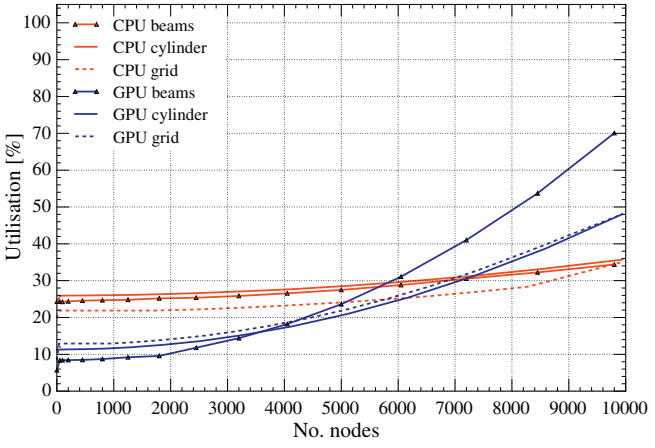

Fig. 15. CPU and GPU memory usage versus node count.

The relative run-times showed that for the cable-based models below 1750 nodes, the CPU was faster at analysing the model than the GPU, and so a GPU analysis cannot be recommended for low node density models. However for medium to high density net models, after the 1750 node count, time saving factors of over eight were achieved with approximately a linear increase in time saved with increasing node count. For the beam elements, the time savings appeared after a higher node count of about 2500. Rather than working with smaller arrays to represent beams individually, the calculations stacked all of the model's beams into one large array to use the GPU more effectively. The GPU acceleration for beams lead to run times that were up to six times faster.

The time-saving benefits are effectively gained when the executed code allows the GPU cores to be loaded as much as possible, so that complex calculations involving large arrays of data can be spread over the many cores. This factor was evident when investigating the utilisation of the GPU and CPU, as the load demand on the former increased steadily, but remained below 100% for all tests, peaking at about 85% for the densest models. While the CPU was easily activated to work at maximum load for even simple models. The time-savings witnessed in this research are of most benefit for complex models with many thousands of nodes, where a calculation that would take an hour to analyse on the CPU, has been shown to take only a few minutes on the GPU, unlocking significant efficiencies in complex dynamic relaxation models.

References

[1] Barnes MR. Form-finding and analysis of prestressed nets and membranes. Computers and Structures 1998;30: (3)685–95.
[2] Barnes MR. Form finding and analysis of tension structures by dynamic relaxation. International Journal of Space Structures 1999;14: (2)89–104.
[3] Wakefield DS. Engineering analysis of tension structures: theory and practice. Engineering Structures 1999;21:680–90.
[4] Adriaenssens SML, Barnes MR. Tensegrity spline beam and grid shell structures. Engineering Structures 2001;23:29–36.
[5] D'Amico B, Kermani A, Zhang H, Shepherd P, Williams CJK. Optimization of cross-section of actively bent grid shells with strength and geometric compatibility constraints. Computers and Structures 2015;154:163–76.
[6] Douthe C, Caron JF, Baverel O. Gridshell structures in glass fibre reinforced polymers. Construction and Building Materials 2010;24:1580–9.
[7] Van Mele T, De Laet L, Veenendaal D, Mollaert M, Block P. Shaping tension structures with actively bent linear elements. International Journal of Space Structures 2013;28: (3-4)127–35.
[8] Wood RD. A simple technique for controlling element distortion in dynamic relaxation form-finding of tension membranes. Computers and Structures 2002;80:2115–20.
[9] Topping BHV, Khan AI. Parallel computation schemes for dynamic relaxation. Engineering Computations 1994;11:513–48.
[10] Hüttner M, Máca J, Fajman P. Analysis of cable - membrane structures using the dynamic relaxation method. *Proceedings of the 9th International Conference on Structural Dynamics* 2014;1919–26.
[11] Schek HJ. The force density method for form finding and computation of general networks. Computer Methods in Applied Mechanics and Engineering 1974;3:115–34.
[12] Linkwitz K. About formfinding of double-curved structures. Engineering Structures 1999;21:709–18.
[13] Kilian A, Ochsendorf J. Particle-spring systems for structural form finding. Journal of the International Association for Shell and Spatial Structures 2005;46:77–84.
[14] Veenendaal D, Block P. An overview and comparison of structural form finding methods for general networks. International Journal of Solids and Structures 2012;49: (26)3741–53.
[15] Lewis WJ. The efficiency of numerical methods for the analysis of prestressed nets and pin-jointed frame structures. Computers and Structures 1989;33: (3)791–800.
[16] Nickolls J, Buck I, Garland M, Skadron K. Scalable parallel programming with CUDA. Queue-GPU computing 2008;6: (2)40–53.
[17] Klöckner A, Pinto N, Lee Y, Catanzaro B, Ivanov P, Fasih A. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. Parallel Computing 2012;38: (3)157–74.
[18] Foundation Python Software. Python language reference. 2016. Version 3.5
[19] MATLAB, Version 8.5 (R2015a). The MathWorks Inc.. Massachusetts: Natick; 2015.
[20] Group Khronos, Open computing language. 2016. Version 2.2
[21] Givon LE. et al. SCIKIT-CUDA 0.5.1: A Python interface to GPU-powered libraries. December 2015.
[22] Hüttner M, Máca J, Fajman P. The efficiency of dynamic relaxation methods in static analysis of cable structures. Advances in Engineering Software 2015;28–35.
[23] Rezaiee-Pajand M, Kadkhodayan M, Alamatian J, Zhang LC. A new method of fictitious viscous damping determination for the dynamic relaxation method. Computers and Structures 2011;89:783–94.
[24] Alamatian J. A new formulation for fictitious mass of the dynamic relaxation method with kinetic damping. Computers and Structures 2012;90-91:42–54.
[25] Lewis WJ, Jones MS, Rushton KR. Dynamic relaxation analysis of the non-linear static response of pretensioned cable roofs. Computers and Structures 1984;18: (6)989–97.
[26] D'Amico B, Zhang H, Kermani A. A finite-difference formulation of elastic rod for the design of actively bent structures. Engineering Structures 2016;117:518–27.