A GPU based acceleration of Finite Element and Isogeometric analysis

***Utpal Kiran, Vishal Agrawal, Deepak Sharma and †Sachin Singh Gautam**

Department of Mechanical Engineering, Indian Institute of Technology Guwahati, Assam, India.

*Presenting author: ukiran@iitg.ac.in
†Corresponding author: ssg@iitg.ac.in

Abstract

In this paper, a Graphics Processing Unit (GPU) based novel parallelization scheme is proposed to reduce the extensive computational cost associated with the finite element (FE) and isogeometric analysis (IGA) simulations of linear and non-linear problems. An innovative parallelization strategy is introduced that achieves fine-grain parallelism and is suitable for GPU. The proposed strategy is implemented within the MatLab® programming environment for the GPU based FE and IGA simulations. It, thus, avoids the need for specialized programming languages like CUDA/C++, which require in-depth knowledge for their implementation. The capabilities and performance of the proposed strategy are examined by solving both linear and non-linear problems. The results demonstrate that the proposed strategy achieves a considerable improvement in the assembly and computation of global tangent matrices over both the single core CPU and multicore CPU based computations. A maximum speedup of 41.4 times over single core CPU and 10.8 times over multicore CPU is achieved for linear problem. For non-linear problem of strip peeling from an adhesive substrate, a maximum speedup of 12.3 times is obtained in comparison to multicore CPU based computation. The proposed strategy can be easily incorporated within the existing codes with little modification.

Keywords: Parallel Programming, GPU computing, IGA, Nonlinear FEA, MatLab.

Introduction

Finite Element Analysis (FEA) is one of the most popular numerical methods used for the solution of a variety of problems governed by the partial differential equations [1]. It is known that FEA involves a large amount of computation to find the nearly exact solution of the problem. Moreover, the computational efforts increase substantially if the analysis includes a large number of finite elements. It, as a resultant, leads to a significant amount of simulation time even on fast modern computers. However, this may not be a desirable choice in the analysis community.

In 2005, Hughes et. al [2] introduced the Isogeometric analysis (IGA) technique to circumvent the above mentioned issue associated with FEA. Unlike FEA, IGA doesn't need to perform expensive mesh generation of the Computer Aided Design (CAD) model as it directly enables the transition of CAD generated model into the analysis framework. Thus, reducing the execution time of IGA based simulation significantly compared to FEA [3].

However, the computational time in case of IGA solver for large-scale problems can still be high [3]. The large simulation time in scientific applications is often reduced by using parallel computers. It involves decomposing a large-scale problem into a smaller number of parts and solving them in parallel over multiple processors. Recently, GPU based parallel computing has achieved great success in accelerating time-consuming scientific applications [4]. The

GPU is specialized hardware designed to handle parallel and independent data task in a very efficient manner. It is a massively threaded processor having thousands of simpler cores instead of few powerful cores like in CPU. A large number of cores in GPU bring the performance of a mini computer cluster to the desktop computer at very low capital cost, low cooling cost and low power consumption [5]. There are numerous applications accelerated successfully by GPU in various fields including aerospace, defense, finance & economics, oil & gas, and computer games [6]. However, writing code for a GPU requires knowledge of parallel programming strategy and specific programming language like CUDA C/C++. Since CUDA C/C++ is a lower level programming language, it demands a great amount of time and effort from the user. On the other hand, scripting languages like MatLab® have become more popular in the scientific community. MatLab® provides an integrated computing environment, which supports the effortless development of code, easier and faster debugging, visualization, and a large number of built-in functions. The ease of code development in MatLab® comes at the cost of lesser flexibility and reduced control to the programmers, which may lead to sub-optimal performance. In the current work, a novel parallelization strategy is introduced which provides a possibility to achieve accurate result at a considerably lower computational cost compared to standard sequential computation approaches. The execution time is further reduced significantly by the use of GPU.

The previous efforts to accelerate FEA on MatLab® have focused on efficient vectorization techniques or parallel computation on multicore CPU through `parfor` or `spmd` construct of parallel computing toolbox [7]. A parallel implementation for coupled electro-mechanical finite element analysis of micro-electro-mechanical (MEMS) device is found in [8]. The work shows the use of `parfor` loop to calculate element stiffness matrices in parallel over 40 MatLab® workers set up to reduce simulation time from 60 hours to 2 hours. A significant amount of reduction in FEA assembly time is achieved by vectorization of code in [9]. In another work [10], a comparative analysis of multicore parallelization and GPU parallelization is done. The implementation uses `parfor` and `spmd` construct for CPU and `arrayfun` function wrapper for GPU. The result shows GPU based algorithm performing poorly than other two. However, the authors believe that with efficient vectorization GPU based parallelization can achieve better performance.

Most of the previous work on acceleration of FEM using GPU is found to be based on CUDA C/C++. The most detailed study of GPU implementation of finite element assembly process is presented in [11]. The authors show the speedup of several folds in assembly for lower order as well as higher order elements. An efficient implementation of numerical integration on GPU is found in [12]. The authors show speedup of 7× over the efficient CPU implementation for quadrilateral element. In [13], a novel interaction-wise strategy for assembly of stiffness matrix in IGA is presented. The proposed strategy achieves speedup up to 54× over single core CPU implementation. The GPU based integration strategy of B-spline basis function in IGA is found in [14]. The above-mentioned works along with many others in literature [15] signify the effectiveness of GPU in accelerating FEM simulation. However, to the best of authors' knowledge no literature exists that discusses the capabilities of MatLab® to use GPU to accelerate FEM.

FEM consists of a sequence of computationally expensive steps like evaluation of local matrices (mass and stiffness), assembly of local matrices into global matrix and solution of assembled system of equations [1]. GPU based computing has been found to be very effective in accelerating almost every step of FEM [15]. The solution of linear system of equation often dominates the simulation time and, therefore, it must be implemented with an optimized linear solver [16][17]. However, the time consumed in element matrix creation and their

assembly to global matrix cannot be ignored, particularly, in nonlinear problems. In nonlinear problems, a large number of time steps are required to reach final solution. Within each time step, there are Newton-Raphson iterations that require revaluation of element stiffness matrices and their reassembly [1]. Thus, an optimum implementation of this step can lead to significant amount of reduction in simulation time.

The objective of the current work is to accelerate the evaluation and assembly of mass and tangent matrices by making use of GPUs through parallel computing toolbox of MatLab®. A MatLab® code can be made to run on GPUs with the minimum amount of changes requiring far less development effort than the language like CUDA C/C++. The function wrappers provided by MatLab® like `bsxfun`, `pagefun`, and `arrayfun` have been used for numerical integration. Assembly to global matrix is done by `sparse` function of the MatLab®. First, an efficient GPU parallel strategy for FEM analysis for 2D elasticity problems is proposed and compared with sequential and CPU parallel (`parfor`) strategy. The proposed strategy is further used to accelerate IGA based nonlinear analysis of a strip peeling problem. The present work aims to utilize the computational power of GPU for FEA and IGA while keeping the development effort minimum by implementing it within the MatLab® environment. The outcome of this study is expected to help people in academic and industry accelerate their FEA based simulation code in MatLab®.

The paper is organised as follows. In next section, problem formulation is presented using IGA. Thereafter, the parallel implementation of FEM is explained along with the data structure. In second last section, results of the numerical experiments done to evaluate the performance of proposed strategy are presented. The last section concludes the paper.

Problem Formulation

This section is divided into the two subsections. In the first, a continuum based formulation of adhesion model and its weak formulation is briefly overviewed. In the second, finite element and the NURBS based discretization of the continuum is presented.

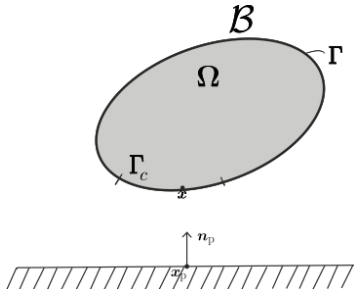

Fig. 1 Contact kinematics of a deformable body and a rigid substrate pair in their current configuration.

Model for Adhesive Contact Problem

Consider a body $\mathcal{B}$ in its current configuration having domain Ω, surface Γ, and the part of its boundary where contact takes places as Γ_c. The schematic arrangement of the body interacting with the rigid substrate is illustrated in Fig. 1. The governing equation for the quasi-static frictionless adhesive contact problem is given by the equilibrium between the work done by the internal, external, and contact forces. For the admissible variation of interaction potential

function $\delta\boldsymbol{\varphi}$, the weak form for the adhesive contact for a deformable body is given by the following statement [18]:

$$\int_{\Omega} \mathrm{grad}(\delta\boldsymbol{\varphi}) : \boldsymbol{\sigma}\, \mathrm{d}\Omega - \int_{\Gamma_c} \delta\boldsymbol{\varphi} \cdot \boldsymbol{t}_c\, \mathrm{d}\Gamma - \delta\Pi_{\mathrm{ext}} = 0, \quad \forall\, \delta\boldsymbol{\varphi} \in \mathcal{V} \qquad (1)$$

where $\mathcal{V}$ is the space of kinematically admissible variation function $\delta\boldsymbol{\varphi}$, $\boldsymbol{\sigma}$ is the Cauchy stress tensor, and $\boldsymbol{t}_c$ is the contact traction over $\boldsymbol{\Gamma}_c$. For the evaluation of contact tractions, penalty method based regularization of constitutive equation for the contact surface traction is used. First, based on the unique projection of slave points on the master surfaces, the unit normal $\mathbf{n}_{\boldsymbol{p}}$ and minimum gap r_s between the contact surfaces are determined, see Fig. 1. Using the definition of the normal gap, the contact traction can be written as:

$$\boldsymbol{T}_c(\boldsymbol{x}_s) = \begin{cases} -\varepsilon_n\, r_s\, \mathbf{n}_p, & r_s < 0 \\ 0, & r_s \geq 0 \end{cases} \qquad (2)$$

where ε_n represents the penalty parameter. For the van der Waals adhesion model and the contact traction $\boldsymbol{T}_c$ is obtained by integrating the Lennard-Jones interaction potential four time [18] and is given as

$$\boldsymbol{T}_c(\mathbf{x}_s) = \boldsymbol{T}_c(r_s)\mathbf{n}_p,$$

$$\text{where} \quad \boldsymbol{T}_c(r_s) = \frac{A_H}{2\pi {r_o}^3}\left[\frac{1}{45}\left(\frac{r_o}{r_s}\right)^9 - \frac{1}{3}\left(\frac{r_o}{r_s}\right)^3\right] \cdot \mathbf{n}_p\,. \qquad (3)$$

Here, A_H, and r_o denote the Hamaker's constant, and the equilibrium spacing of interacting particles of contacting bodies, respectively.

FE discretized weak formulation

Within the FEA, the domain of the body $\mathcal{B}$ is discretized into n^e number of elements such that $\Omega = \sum_{e=1}^{n_e} \Omega^e$ and the displacement field $\boldsymbol{u}^e$, its variation $\delta\boldsymbol{u}^e$ for a standard finite element Ω^e is given by the summation of product of Lagrange basis function and field variables as

$$\boldsymbol{u}^e = \sum_{i=1}^{n_n} N_i \boldsymbol{u}_i = \mathbf{N}\mathbf{u}\,, \; \delta\boldsymbol{u}^e = \sum_{i=1}^{n_n} N_i \delta\boldsymbol{u}_i = \mathbf{N}\delta\mathbf{u}, \qquad (4)$$

where $\boldsymbol{u}_i$ represents the displacement vector of node i, and n_n denotes the total number of nodes in an element Ω^e. $\mathbf{N}$ is the basis function matrix: $\mathbf{N} = \left[N_1\boldsymbol{I}, N_2\boldsymbol{I}, \ldots, N_{n_n}\boldsymbol{I}\right]$, where N_i represents the Lagrangian basis function associated to node i, and $\boldsymbol{I}$ is the identity tensor in $\mathbb{R}^2$. Following the Galerkin approach, the initial configuration $\boldsymbol{X}$, and current configuration $\boldsymbol{x}$ of body $\mathcal{B}$ are described in a likewise manner as in Eq. (4). In the context of IGA, NURBS basis functions used for the discretization of the geometry are employed for the determination of solution field. The displacement field $\boldsymbol{u}$, its variation $\delta\boldsymbol{u}$, and the current configuration of the geometry $\boldsymbol{x}$ is represented in terms of the NURBS basis functions $R_{i,p}(\xi,\eta)$, i.e. N_i is replaced by $R_{i,p}(\xi,\eta)$ in Eq. (4). The reader is referred to [19] for the detailed description of the implantation of IGA into the finite element code structure. The discretized weak form Eq. (1) for the adhesive contact can be cast into the following matrix form [18]:

$$\delta\mathbf{u}^{\mathrm{T}}[\mathbf{f}_{\mathrm{int}} + \mathbf{f}_c - \mathbf{f}_{\mathrm{ext}}] = \mathbf{0}, \quad \forall \;\; \delta\mathbf{u}^{\mathrm{T}} \in \mathcal{V} \qquad (5)$$

where $\mathbf{f}_{\mathrm{int}}$, $\mathbf{f}_c$, and $\mathbf{f}_{\mathrm{ext}}$, are the vectors for internal, contact, and externally applied forces, respectively. Internal force vector is described by the constitutive relation of the material

model. In the present work, a Neo-Hookean hyperelastic material model is used and the Cauchy stress is determined by the following expression [20]

$$\boldsymbol{\sigma} = \frac{\lambda}{J}\ln J\boldsymbol{I} + \frac{\mu}{J}\left(\boldsymbol{F}\boldsymbol{F}^{T} - \boldsymbol{I}\right) \tag{6}$$

where λ and μ are Lamé's constants, and J denotes the determinant of deformation gradient tensor $\boldsymbol{F}$. The contact contribution $\mathbf{f}_c$ over the contact surface $\boldsymbol{\Gamma}_c$ of an element Ω^e can be computed by the following expression

$$\mathbf{f}_c = \sum_{e=1}^{n_e} \mathbf{f}_c^e, \qquad \text{where} \quad \mathbf{f}_c^e = -\int_{\boldsymbol{\Gamma}_c^e} \mathbf{N}^{\mathrm{T}}\boldsymbol{T}_c \, d\Gamma. \tag{7}$$

Parallel implementation of FEA

The proposed strategy has been implemented entirely in MatLab® environment using the parallel computing toolbox. The parallel computing toolbox provides various ways to run code on the GPU. The simplest way is to use built-in function enhanced to work on GPU with `gpuArray` type of input data. Since the built-in functions are not always sufficient, we have written our own MatLab® functions and used them with function wrappers to implement our strategy on the GPU. The user-defined MatLab® functions can be used without any function wrappers but it may launch multiple CUDA kernels even for simpler function. The function wrapper like `arrayfun` compiles multiple operations of a function into single GPU kernel and therefore provides better performance. However, the GPU function wrappers have some restrictions. Only those user-defined functions that contain element wise operations can be used. The `arrayfun` wrapper allows the function to take arrays/matrix as input but indexing into the array is not allowed.

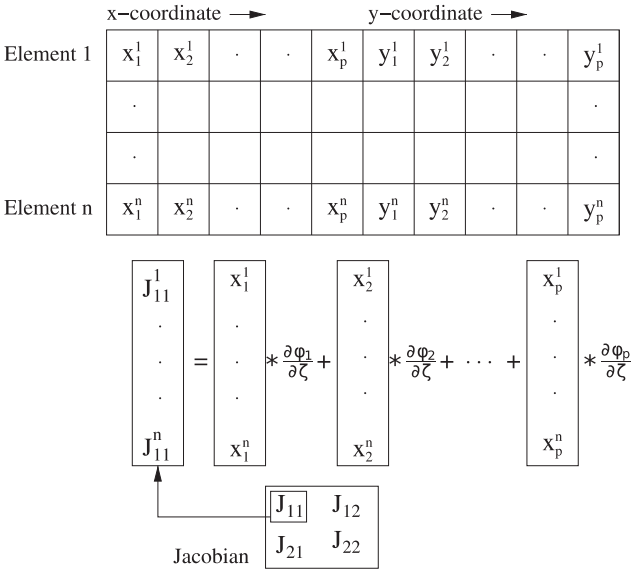

Fig. 2. Calculation of Jacobian.

Vectorization of code is critical to achieve better performance by the GPU parallelization. Our proposed strategy is based on a vectorization scheme in which computation at the elemental level with scalar variable is converted into computation at mesh level with arrays. It

enables us to easily implement our strategy by functions with element wise operations and use them with `arrayfun` function wrapper. In our implementation, the numerical integration is done by looping over the Gauss points. For each Gauss point, Jacobian is calculated for transformation to the reference coordinates. Since we are doing calculation at the mesh level, Jacobian is calculated for each element of the mesh simultaneously. The data structure and procedure for Jacobian calculation is shown in Fig 2. Here, nodal coordinates are reordered and stored in the matrix with each column containing an individual coordinate for all elements. The derivative of the basis function in reference coordinate is pre-computed for all the Gauss points and reordered to facilitate the computation of Jacobian. As shown in Fig. 2, the calculation of an entry of Jacobian is done by multiplying coordinate values with basis function derivatives for a particular Gauss point. This evaluates to an array that contains an entry of Jacobian matrix for all the elements of the mesh. The other entries of the Jacobian matrix are calculated in the similar way. The computation of determinant for all the element of the mesh can be done simply by element wise operations over array of entries of Jacobian matrix. The computation of inverse of Jacobian is done by co-factor calculation. This can again be done in parallel for all the Jacobian matrices by element wise operations.

The inverse of Jacobian is multiplied with derivative of shape function in reference coordinates to calculate shape function derivative in physical coordinates. Since we are working with arrays, the above product produces shape function derivative for all the elements of the mesh. To facilitate the computation of element stiffness matrix, the derivative of shape function is stored as shown in Fig. 3. Each column of the matrix contains derivative of particular shape function for all the elements of the mesh. The evaluation of element stiffness matrix is done by computing each individual entry of the matrix simultaneously for

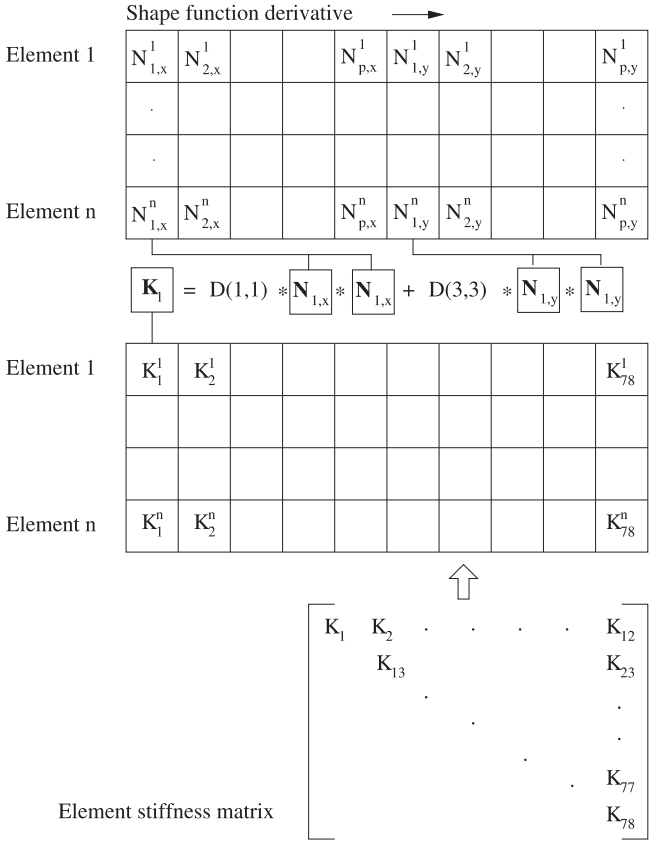

Fig. 3. Calculation of element stiffness matrix.

all the elements. The calculation and storage pattern is shown in Fig. 3. Since the elemental matrix is symmetric, we calculate only the unique entries.

All the matrices involved in the computation are stored in GPU memory. The assembly to global matrix is done by `sparse` function of the MatLab®. The off-diagonal entries are assembled first, so that it can be transposed to generate the symmetric part of the matrix. The on-diagonal entries are added later to complete the global matrix. The `sparse` function requires the row and column indices of the values to be assembled. This can be calculated beforehand using mesh connectivity. We pre-compute the row and column indices and reorder them according to the storage arrangement of element stiffness matrix. The global matrix is assembled on GPU.

Results and Discussion

To evaluate the performance of the proposed strategy two different problems are solved. The first one is linear elastic two dimensional (2D) cantilever beam problem with concentrated load at the tip and the second one is a strip peeling problem. The performance is compared among a CPU sequential, CPU parallel, CPU vectorized and GPU implementation. The CPU sequential approach uses a loop over each element of the mesh for element matrix computation and assembly. The CPU parallel implementation uses `parfor` construct of the parallel computing toolbox to utilize multiple processors on the CPU for parallel computation of element matrix computation and assembly. The CPU vectorized implementation is based on the proposed vectorization scheme but uses the CPU for computation.

The machine used for the numerical experiment consists of Intel Xeon® E5-2650 processor having 2.2GHz of clock speed and a NVIDIA Tesla K40c GPU with 2880 cores clocked at 745MHz. The proposed strategy has been implemented on MatLab® R2016a using the parallel computing toolbox.

2D Cantilever beam

A 2D cantilever beam with concentrated load at tip is taken as shown in Fig. 4. The geometric and material parameters are given as: length (L) – 10 m, breadth (B) – 1 m, Young's modulus (E) - 210 GPa, Poisson's ratio (ν) - 0.3 and end load (P) - 10^5 N. Linear quadrilateral element with two degrees of freedom (DOF) per node is used to discretize the domain.

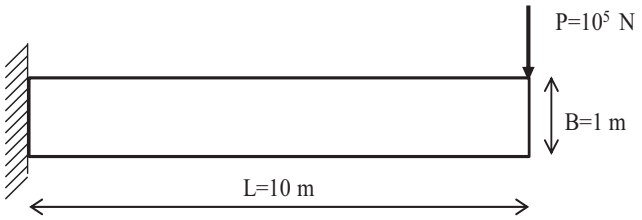

Fig. 4. A 2D cantilever beam with end load.

Table 1. Finite element mesh.

Mesh	Elements	Nodes	Degrees of freedom
Mesh 1	900 000	903 301	1 806 602
Mesh 2	1 600 000	1 604 401	3 208 802
Mesh 3	2 500 000	2 505 501	5 011 002
Mesh 4	3 600 000	3 606 601	7 213 202

The problem is solved for different level of mesh refinement keeping the aspect ratio of the elements same. The finite element mesh with varying level of refinement is shown in Table 1. Structured mesh has been used for the purpose of numerical experiment but the calculation is performed for each of the elements treating them as unstructured. Fig. 5 shows the comparison of numerical integration and assembly time as a function of mesh sizes. It can be observed that the proposed GPU implementation obtains least timings for all of the mesh sizes. The CPU vectorized implementation achieves significantly less time than CPU parallel strategy (using 12 workers), highlighting the effectiveness of the proposed vectorization scheme. The speedups of the GPU implementation over all other implementations are shown in Fig. 6. A maximum of 41.4× (Mesh 1) speedup is obtained over CPU sequential and 10.8× (Mesh 1) over CPU parallel implementations. The GPU code could achieve speedup of only 3.6× - 3.8× over CPU vectorized implementation, which shows that the proposed vectorization of FEM is able to scale very effectively also on the CPU.

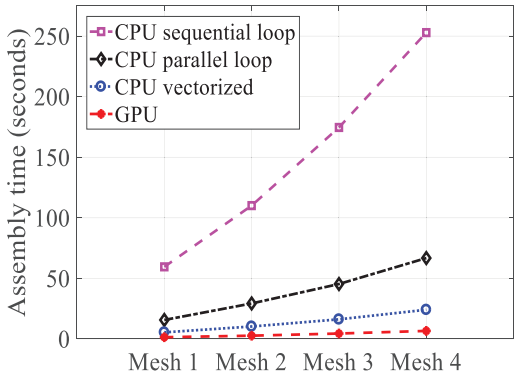

Fig. 5. Numerical integration and assembly time.

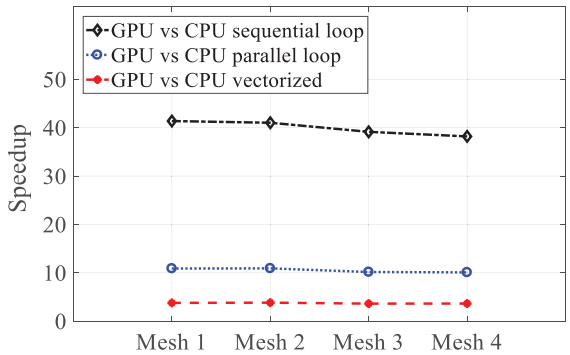

Fig. 6. Speedup in numerical integration and assembly

The linear system of equations given by FEM can be solved by either direct sparse solver or by iterative solvers. The mldivide(\) operator in MatLab® implicitly chooses the best algorithm (direct solver) depending upon the type of input matrix. On CPU, mldivide performs much better than the iterative sparse solvers. Since mldivide operator is not supported on GPU for sparse matrices, we compared mldivide on CPU with iterative solver on the GPU. For Mesh 3 in Table 1, the pcg (preconditioned conjugate gradient) function takes 295 seconds on GPU, whereas mldivide takes 25.5 seconds to solve the system of equations on CPU. This prompts us to adopt a strategy in which assembly, numerical integration is done on GPU, and solution of linear system of equations takes place on CPU by the mldivide operator. We call this as GPU + CPU strategy. Fig. 7 shows the comparison of overall execution time. The GPU + CPU strategy achieves 2.2× speedup over CPU parallel strategy, 5.6× speedup over sequential strategy and 1.4× over CPU vectorized strategy for the finest mesh.

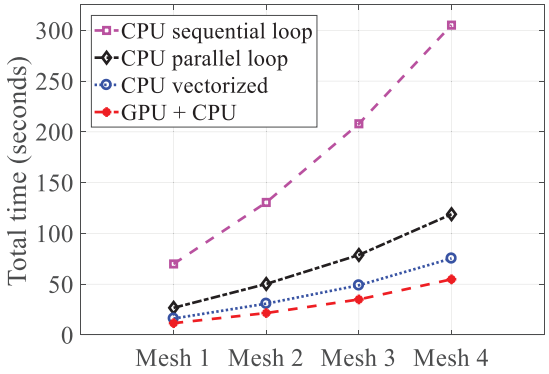

Fig. 7. Overall execution time for cantilever beam problem.

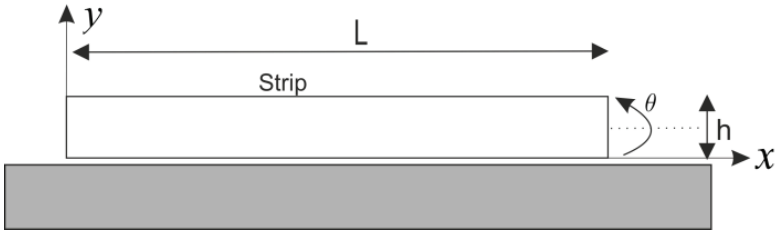

Fig. 8. The geometrical set-up of peeling problem.

Strip peeling problem

We consider the peeling of a deformable strip (having length $L = 200Lo$, height $h = 10Lo$ with $Lo = 1$ nm) adhering to a flat, rigid substrate. The schematic arrangement of this problem is shown in Fig. 8. An isotropic, nonlinear Neo-Hookean material model with $E = 1$ GPa, and $\nu = 0.2$, under plane strain conditions is used. It is considered that the adhesive contact forces are present at the 75% of the bottom length of strip (from $x = 0$ to $x = 150$ Lo) and are calculated using Eq. (3) with $r_o = 0.4$ nm and $A_H = 10^{-19}$ J. At one end, a rotation angle θ is applied in such a manner that it yields a constant moment during the peeling process and a rotation step size $\Delta\theta = 0.1^o$ is chosen for the simulation. The strip is discretized with 240 x 12, 320 x 16, 400 x 20, 480 x 24, 640 x 32, and 720 x 36 number of elements along each direction and corresponding discretization is referred as mesh 1, Mesh 2, Mesh 3, Mesh 4, Mesh 5, and Mesh 6, respectively.

In Fig. 9, the comparison of numerical integration and assembly time is done. A considerable amount of reduction in assembly timings can be observed for the proposed strategies. For Mesh 6, the integration and assembly time reduces from 21876 seconds to 1780 seconds. The CPU vectorized code achieves speedup in the range 5.1× - 4.1× over CPU parallel strategy. The GPU based strategy achieves speedup in the range 2.4× - 12.3× over CPU parallel and 0.47× - 2.9× over CPU vectorized. The device set up time and data communication time dominates in GPU +CPU strategy for smaller mesh size which results into inferior performance compared to CPU vectorized for Mesh 1 and Mesh 2. When the mesh size increases, effectiveness of GPU becomes more apparent. The comparison of total execution time is shown in Fig. 10. The GPU + CPU strategy takes the least amount of time and reduces the total simulation time from 49343.3 seconds to 28452 seconds for Mesh 6, which is remarkable.

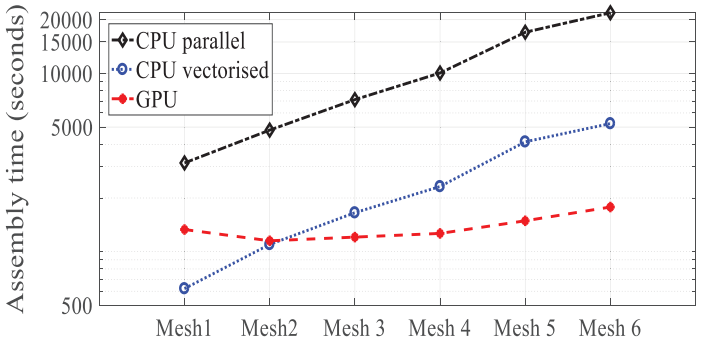

Fig. 9. Numerical integration and assembly time for strip peeling problem.

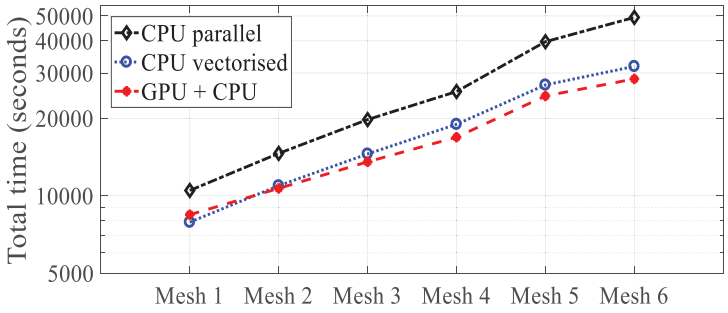

Fig. 10. Total execution time for strip peeling problem.

Conclusions

In this work, a novel vectorization strategy is presented to implement FEA and IGA on GPU using MatLab® environment. The proposed vectorization strategy accelerates elemental tangent matrix evaluation and their assembly by performing the required computation at mesh level rather than at element level. For linear elastic cantilever beam problem, the GPU based strategy is found to be 10.1× faster than CPU parallel (parfor) and 38.2× faster than sequential strategy in numerical integration and assembly for 7.2 million DOF. For strip peeling problem, a maximum speedup of 12.3× is achieved over CPU parallel strategy in numerical integration and assembly. This leads to total reduction in overall simulation time from 13.7 hours to 7.9 hours. Based on numerical simulation done in this paper, we find that a GPU is very effective in accelerating the evaluation of element tangent matrix and their assembly to global matrix in MatLab®. In future, we would like to extend this work to accelerate evaluation of contact forces on GPU.

References

[1] Bathe, K. J. (1996) *Finite element procedures*, Prentice Hall of India, New Delhi, India.

[2] Hughes, T.J.R., Cottrell, J. A. and Bazilevs, Y. (2005) Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement, *Computer methods in applied mechanics and engineering* **194,** 4135-4195.

[3] Cottrell, J. A., Hughes, T. J. R. and Bazilevs, Y. (2009) *Isogeometric Analysis: Toward Integration of CAD and FEA*, Wiley.

[4] Mittal, S. and Vetter, J. S. (2015) A survey of CPU-GPU heterogeneous computing techniques, *ACM Computing Surveys (CSUR)* **47,** 69.

[5] Huang, S., Xiao, S. and Feng, W., On the energy efficiency of graphics processing units for scientific computing, *In Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on,* IEEE 2009, 1-8.

[6] NVIDIA, *Popular GPU-accelerated Applications Catalog* 2017.

[7] Martin, J. and Sharma, G. (2009) MATLAB: A Language for parallel computing, *International Journal of Parallel Programming* **37,** 3-36.

[8] Hosagrahara, V., Tamminana, K. and Sharma, G., Accelerating Finite Element Analysis in MATLAB with Parallel Computing, *The MathWorks News and Notes*, 2010. https://www.mathworks.com/tagteam/66859_91826v00_FEM_final.pdf.

[9] Cuvelier, F. Japhet, C. and Scarella, G. (2016) An efficient way to assemble finite element matrices in vector languages, *BIT Numerical Mathematics* **56**, 833-864.

[10] Simkus, A. and Turskiene, S. (2013) Parallel computing for the finite element method in MATLAB, *Computational Science and Techniques* **1**, 214-221.

[11] Cecka, C., Lew, A. J. and Darve, E. (2011) Assembly of finite element methods on graphics processor, *International Journal of Numerical Methods in Engineering* **85,** 640-669.

[12] Zhang, J. and Shen, D., GPU based implementation of finite element method for elasticity using CUDA, *in High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC EUC), 10th International Conference on IEEE*, 2013, 1003-1008.

[13] Karatarakis, A., Karakitsios, P. and Papadrakakis, M. (2014) GPU accelerated computation of the isogeometric analysis stiffness matrix, *Computer Methods in Applied Mechanics and Engineering* **269**, 334-355.

[14] Woźniak, M. (2015) Fast GPU integration algorithm for isogeometric finite element method solvers using task dependency graphs, *Journal of Computational Science* **11**, 145-152.

[15] Georgescu, S., Chow, P. and Okuda, H. (2013) GPU acceleration for FEM-based structural analysis, *Archives of Computational Methods in Engineering* **20**, 111-121.

[16] Elman, H., Sylvester, D. and Wathen, A. (2014) *Finite elements and fast iterative solvers*, Second Ed. Oxford University Press, Oxford, UK.

[17] Jung, J. H. and Bae, D. S. (2017) An implementation of direct linear equation solver using a many-core CPU for mechanical dynamic analysis, *Journal of Mechanical Science and Technology* **31**, 4637-4645.

[18] Sauer, R. A., and Li, S. (2007) An atomic interaction-based continuum model for adhesive contact mechanics, *Finite Elements in Analysis and Design* **43**, 384-396.

[19] Agrawal, V., and Gautam, S. S., IGA: A Simplified Introduction and Implementation Details for Finite Element Users. *Journal of The Institution of Engineers (India): Series C* (https://doi.org/10.1007/s40032-018-0462-6).

[20] Bonet, J. and Wood, R. (2008) *Nonlinear Continuum Mechanics for Finite Element Analysis*, Cambridge University Press, Cambridge. (doi:10.1017/CBO9780511755446)