Parallelized Implementation of the Finite Particle Method for Explicit Dynamics in GPU

Jingzhe Tang[1], Yanfeng Zheng[1], Chao Yang[1], Wei Wang[1] and Yaozhi Luo[1, *]

Abstract: As a novel kind of particle method for explicit dynamics, the finite particle method (FPM) does not require the formation or solution of global matrices, and the evaluations of the element equivalent forces and particle displacements are decoupled in nature, thus making this method suitable for parallelization. The FPM also requires an acceleration strategy to overcome the heavy computational burden of its explicit framework for time-dependent dynamic analysis. To this end, a GPU-accelerated parallel strategy for the FPM is proposed in this paper. By taking advantage of the independence of each step of the FPM workflow, a generic parallelized computational framework for multiple types of analysis is established. Using the Compute Unified Device Architecture (CUDA), the GPU implementations of the main tasks of the FPM, such as evaluating and assembling the element equivalent forces and solving the kinematic equations for particles, are elaborated through careful thread management and memory optimization. Performance tests show that speedup ratios of 8, 25 and 48 are achieved for beams, hexahedral solids and triangular shells, respectively. For examples consisting of explicit dynamic analyses of shells and solids, comparisons with Abaqus using 1 to 8 CPU cores validate the accuracy of the results and demonstrate a maximum speed improvement of a factor of 11.2.

Keywords: Finite particle method, GPU, parallel computing, explicit dynamics.

1 Introduction

The finite element method (FEM), which is derived from variational principles and continuum mechanics, has been widely applied in the analysis of an extensive range of engineering problems over the past few decades and is also the theoretical foundation for most commercial software for numerical simulations. The numerical integration of the element stiffness matrices, the formation of the global matrices and the solution of the equilibrium equations in the form of a linear system are the main tasks of the FEM; among these tasks, the solution step often dominates the performance of the FEM pipeline. As the numbers of elements and degrees of freedom (DOFs) increase, the global matrices rapidly grow in size, causing the complexity of the linear system and the required computation time to increase exponentially, especially for large-scale dynamic simulations.

To resolve the problem of long computation time in FEM analyses, the advantages of parallel computing have commonly been exploited on central processing units (CPUs)

[1] College of Civil Engineering and Architecture, Zhejiang University, Hangzhou, China.

[*] Corresponding Author: Yaozhi Luo. Email: luoyz@zju.edu.cn.

Received: 28 July 2019; Accepted: 06 December 2019.

using the Message Passing Interface (MPI) and Open Multi-Processing (OpenMP) standards. In recent years, the use of graphics processing units (GPUs) in the field of high-performance computing has also been shown to be effective in terms of computation time and has rapidly gained in popularity [Hwu (2011)]. Considerable research on CPU- and GPU-based parallel strategies and implementations for the FEM has been carried out over the past few decades, and the main course of development of related studies has been reviewed in Georgescu et al. [Georgescu, Chow and Okuda (2013)]. To summarize, there are two main approaches. The first consists of various techniques of accelerating the mathematical procedures for solving linear algebra systems, which are often encountered in the FEM. Various parallelization strategies for direct solvers, such as the Cholesky factorization and lower-upper (LU) decomposition, and for iterative solvers using the conjugate gradient method have been proposed and implemented over the years; the performances of such parallelized solvers are compared in Cheik Ahamed et al. [Cheik Ahamed and Magoulès (2017); Pikle, Sathe and Vyavhare (2018); Rao and Kamra (2018)]. Instead of being restricted to the FEM, most of these techniques can be generically applied to other linear systems; in fact, there are already numerous corresponding parallel computing packages and toolkits available from several large software companies, which have been successfully integrated into many industrialized software packages, as listed in NVIDIA et al. [NVIDIA (2017); The Khronos Group (2013)].

The other approach consists of methods that are more specialized for the FEM theory itself, such as the domain decomposition method (DDM) and the element-by-element (EBE) method. In the DDM, the most commonly used method in parallelized FEM systems, the overall structural model is decomposed into submodels, and the calculations for each submodel are each individually performed by a different processing core; in this way, the whole model can be concurrently processed and then reassembled based on intersecting regions, as explained in Papadrakakis et al. [Papadrakakis, Stavroulakis and Karatarakis (2011)]. In the EBE method, on the other hand, the vector product of the global stiffness matrix is transformed into vector products of a set of element stiffness matrices, making the steps of formation and solution obsolete. Usually combined with the DDM, the EBE method has been successfully implemented in distributed multiprocessing systems; examples can be found in Bova et al. [Bova and Carey (2000); Gullerud and Dodds Jr (2001)]. Generally, the main focus of these methods for the parallelization of the FEM is to decompose or decouple the linear equations, and the applications of these methods all demonstrate effective improvements in speed for large-scale FEM analysis. However, equilibrium equations in the strongly coupled form still constrain further improvements in performance.

The finite particle method (FPM) is a novel numerical method for engineering-oriented applications involving complex structural behaviors. A recent review of the FPM, including its background and fundamentals, can be found in Luo et al. [Luo, Zheng, Yang et al. (2014)]. While the traditional FEM is derived from continuum mechanics, the FPM is derived from vector mechanics, as first proposed by Shih et al. [Shih, Wang and Ting (2004); Ting, Shih and Wang (2004a, 2004b)]. In this method, the subject of analysis is no longer viewed as a continuum. Instead, in the FPM, a physical body is modeled as a finite number of particles in space and a finite number of path units in time, as depicted in Fig. 1. Within each period of a given time increment, the motions of the particles are controlled by Newton's second law. The kinematic state of equilibrium is forced on each particle individually instead of the whole system, so that no global matrices need to be constructed or solved. The FPM can be characterized as a particle method since the particles carry structural variables such as mass, density, velocity, strain and stress. However, the particles in the FPM do not have physical volumes; instead, they are connected by elements. It can be said that the FPM shares the simplicity of describing topology in terms of elements with the FEM while possessing the advantage of already decoupled equations by virtue of its nature as a particle method. In recent years, the FPM has been applied to various types of complex structural behaviors with promising results. Related work in this field can be found in relation to mechanism analysis [Yu and Luo (2009a, 2009b)], contact and collision [Yu and Luo (2013)], shape analysis for tensile structures [Yang, Shen and Luo (2014)], and progressive collapse simulations [Yu, Paulino and Luo (2010); Yu and Zhu (2016)]. Similar to the FPM, the vector-form intrinsic finite element (VFIFE) method has also been developed by other scholars; this method is also based on vector mechanics and has been applied in the contexts of bridges and railways [Duan, Wang, Wang et al. (2018); Duan, Wang and Yau (2019)], smart structures [Xu, Li, Jiang et al. (2015)], mechanical joints [Yang, Cheng and Zhang (2016)] and marine risers [Li, Guo and Guo (2018)].

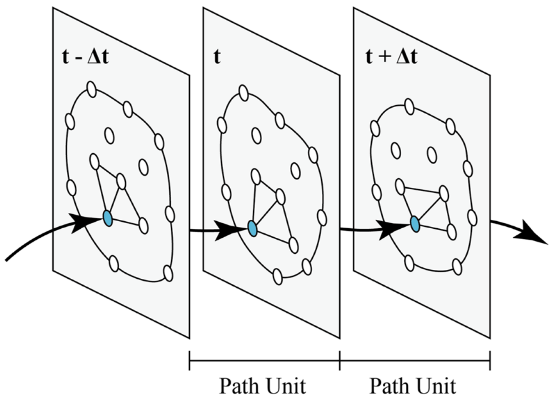


Figure 1: Discrete model of the FPM

Given the partial differential form of Newton’s second law, the kinematic equations for particles in the FPM are solved using the explicit central difference time integration algorithm, which is conditionally stable. To ensure a converged result in the FPM, the time increment must be maintained at a relatively small value; thus, more iterations are needed for a fixed amount of physical time. The FPM is in urgent need of an acceleration strategy to overcome the heavy computational burden of its explicit framework caused by this time step limitation. Moreover, unlike in the FEM procedure, no additional steps of assembling and decoupling global matrices are required; therefore, the overall complexity of the FPM continues to increase linearly as the model scale grows. As a result, the evaluations of the element equivalent forces are self-reliant between elements, and the kinematic equations can be solved individually for each particle. This feature allows the FPM to be strongly accelerated by means of parallel computing techniques, which is an advantage that many particle-based numerical methods share.

Generally, an ideal parallel implementation should allow independent procedures to be processed concurrently. In this case, parallel implementations of the FPM should be able to treat individual particles or elements simultaneously. Considering the general scale of the discrete particle model of the FPM, GPUs are much more suitable for the job than CPUs: a GPU can contain more than a thousand processing cores, while a common CPU provides only up to 24 cores. Various efforts related to GPU-accelerated implementations of other particle-based methods, such as the discrete element method (DEM) [Qi, Li, Jiang et al. (2015)] and smoothed particle hydrodynamics (SPH) [Xia and Liang (2016)], have been reported in recent years. In terms of computational efficiency, the performance of these GPU-accelerated implementations is substantially improved relative to their CPU counterparts. Inspired by these precedents for similar applications, in this study, a parallel strategy for the FPM has been implemented on a GPU architecture.

In the field of GPU-accelerated computing, the development of specialized programming models is essential to allow researchers and developers to gain access to the computing power of GPUs. In 2006, NVIDIA, one of the main producers of graphics cards, launched a development environment called Compute Unified Device Architecture (CUDA), which is available only through NVIDIA's hardware. Later, in 2008, Apple and Khronos provided a free and open language called the Open Computing Language (OpenCL), which is intended for use on all compatible graphics cards on the market [The Khronos Group (2013)]. These two general-purpose programming languages, each with its own merits and disadvantages, have made GPUs increasingly valid computing resources in engineering simulations. The greatest downside of CUDA is that a CUDA-based application can run only on NVIDIA GPUs, while OpenCL is available on graphics cards from other manufacturers, such as AMD and Intel. However, the market for graphics cards aimed at scientific computing is dominated by NVIDIA. Meanwhile, as a proprietary language, CUDA also provides higher-level application programming interfaces (APIs), better profiling tools and a richer programming ecosystem compared to OpenCL [Cook (2013)]. Therefore, CUDA has been chosen for the GPU-based acceleration of the FPM in this work.

In this paper, a GPU-accelerated parallel strategy for the FPM is proposed and implemented as the basis of an FPM platform. First, based on the fundamentals of the FPM theory, a generic parallelized computational framework for multiple types of FPM analysis is established by taking advantage of the independence of each step of the FPM workflow. Then, with the help of the CUDA programming model, GPU implementations of the main tasks of the FPM, such as evaluating and assembling the element equivalent forces and solving the kinematic equations for particles, are elaborated from the perspectives of thread management and memory optimization. Performance tests on the speedup ratios for various types of FPM elements are reported to illustrate the improvements in performance achieved with GPU parallelization. In the end, for examples consisting of explicit analyses of shells and solids, comparisons with the CPU-accelerated Abaqus are presented to validate the accuracy and efficiency of the proposed platform.

This paper is structured as follows. In Section 2, the fundamentals of the FPM are briefly introduced, and a generic parallelized computational framework for the FPM is proposed. The details of the developed FPM platform are given in Section 3, and the GPU implementations of the main tasks of the FPM are also described. Performance tests for various types of FPM elements are reported in Section 4, and the numerical results for triangular shells and hexahedral solids are compared with the results from Abaqus for validation of the proposed platform in terms of accuracy and efficiency. Finally, in Section 5, conclusions are drawn, and possible future improvements are discussed.

2 Fundamentals of the parallelized FPM

Referring to a mechanical model in applied physics, the FPM holds that every discrete particle is constantly in a state of kinematic equilibrium and that each particle's motion is governed by Newton's second law in vector form. Therefore, for an arbitrary particle α, its translational and rotational displacements follow Eqs. (1) and (2):

$$m_\alpha \ddot{\boldsymbol{d}}_\alpha = \boldsymbol{F}_\alpha^{ext} + \boldsymbol{F}_\alpha^{int} = \boldsymbol{F}_\alpha \tag{1}$$

$$\boldsymbol{I}_\alpha \ddot{\boldsymbol{\theta}}_\alpha = \boldsymbol{M}_\alpha^{ext} + \boldsymbol{M}_\alpha^{int} = \boldsymbol{M}_\alpha \tag{2}$$

where m_α and $\boldsymbol{I}_\alpha$ are the mass and mass inertia matrix, respectively, of particle α; $\ddot{\boldsymbol{d}}_\alpha$ and $\ddot{\boldsymbol{\theta}}_\alpha$ denote its acceleration vectors for translation and rotation, respectively; $\boldsymbol{F}_\alpha$, the composite force vector for particle α, consists of external loads $\boldsymbol{F}_\alpha^{ext}$ and internal equivalent forces $\boldsymbol{F}_\alpha^{int}$, which must be accumulated from every element connected to particle α; and the same applies to $\boldsymbol{M}_\alpha$, $\boldsymbol{M}_\alpha^{ext}$ and $\boldsymbol{M}_\alpha^{int}$ but for the moments of the forces. It can be seen that the kinematic equilibrium equation for each particle, and even for each DOF, is independent.

It is essential to treat Eqs. (1) and (2) numerically due to their partial differential form. Various implicit and explicit time integration schemes can be adopted. Here, the standard explicit central difference time integration algorithm is applied to the FPM. Considering the mass damping effect, given a particle's displacements at time t and $t - \Delta t$, its displacement at time $t + \Delta t$ can be explicitly obtained in an iterative manner:

$${}^{t+\Delta t}\boldsymbol{d}_\alpha = c_1 \Delta t^2 m_\alpha^{-1}\ {}^{t}\boldsymbol{F}_\alpha + 2c_1\ {}^{t}\boldsymbol{d}_\alpha - c_2\,{}^{t-\Delta t}\boldsymbol{d}_\alpha \tag{3}$$

$${}^{t+\Delta t}\boldsymbol{\theta}_\alpha = c_1 \Delta t^2 \boldsymbol{I}_\alpha^{-1}\ {}^{t}\boldsymbol{M}_\alpha + 2c_1\ {}^{t}\boldsymbol{\theta}_\alpha - c_2\,{}^{t-\Delta t}\boldsymbol{\theta}_\alpha \tag{4}$$

where Δt is the time increment for each path unit, $c_1 = (1 + \xi\Delta t/2)^{-1}$, $c_2 = c_1(1 - \xi\Delta t/2)$, and ξ is the coefficient for the mass damping effect.

As with many other explicit time integration schemes, the central difference algorithm applied here is numerically stable only conditionally. According to the Courant-Friedrichs-Lewy (CFL) stability condition, the time increment must be maintained below a critical value to keep the algorithm from diverging. The critical time increment $\Delta t_{critical}$ depends on the overall element size and the material properties; for a given dynamic structural system, it can be determined through Eq. (5):

$$\Delta t_{critical} = \min_e \{L_e / c_e\} \tag{5}$$

where L_e is the characteristic length of each element and c_e is the corresponding effective dilatational wave speed of the material. c_e is a function of the material properties, while the calculations for L_e vary depending on the element geometry; the corresponding formulations are addressed in Hallquist [Hallquist (2006)]. A time increment smaller than this critical value ensures that an acoustic sound wave has sufficient time to pass through the element between two adjacent particles in the FPM discrete model. The overall critical time increment $\Delta t_{critical}$ is, of course, the minimum value among all elements.

Generally, the iterative process for updating the translational displacement of a particle in the FPM follows the four steps listed below, and the process for rotational displacement is quite similar. The subscripts α and e in the following equations represent an arbitrary particle and an arbitrary element, respectively.

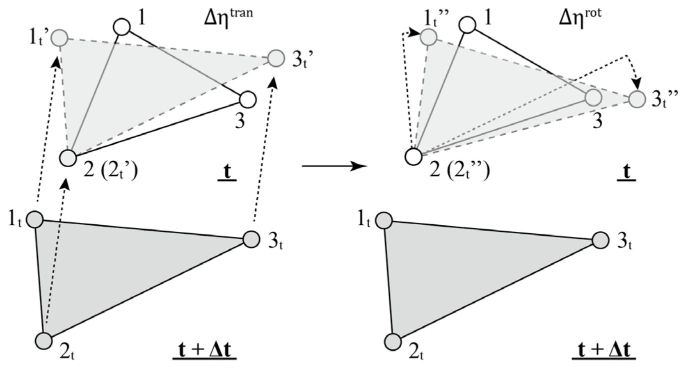


Figure 2: Illustration of the fictitious reverse motion technique

Step I: Preparation

The initial displacement ${}^{0}\boldsymbol{d}_{\alpha}$ is set, and the displacement for the next step is then calculated from the initial external loads ${}^{0}\boldsymbol{F}_{\alpha}^{ext}$ to initiate the iterative process.

Step II: Element updating

Based on the particle displacement at time t, the element equivalent internal forces ${}^{t}\boldsymbol{F}_{e}^{int}$ can be calculated. This is the most complicated and time-consuming process in the FPM, but the calculations are self-reliant between each element. This process can be divided into five substeps.

Step II.a: Pure deformation evaluation

By means of the fictitious reverse motion technique from vector-form mechanics, the effect of rigid-body motion is eliminated from the overall displacement, yielding the pure deformation of the element particles as follows:

$$ {}^{t}\Delta\boldsymbol{\eta} = {}^{t}\Delta\boldsymbol{d} - \Delta\boldsymbol{\eta}^{tran} - \Delta\boldsymbol{\eta}^{rot} \tag{6} $$

where ${}^{t}\Delta\boldsymbol{d}$ is the incremental displacement of the particles within the element and $\Delta\boldsymbol{\eta}^{tran}$ and $\Delta\boldsymbol{\eta}^{rot}$ denote the incremental displacements caused by translational and rotational rigid-body motions, respectively.

The determination of the rigid-body translations and rotations can differ greatly for different element types and geometries. The process of obtaining the rigid-body motions for a triangular element is illustrated in Fig. 2. Detailed formulations can be found in Wu [Wu (2013); Yu, Paulino and Luo (2010); Zhang, Yang and Luo (2017)] for beams, triangular shells and tetrahedral solids. A set of deformation coordinates is then introduced specifically to remove the modes related to the rigid-body motion and to reduce the total number of independent variables to the correct number. After simple coordinate manipulations, ${}^{t}\Delta\hat{\boldsymbol{\eta}}$, the pure deformation in the deformation coordinate system (DCS), is obtained.

Step II.b: Strain evaluation

Shape functions in the same form as those developed in the FEM are introduced to describe the strain distributions within each FPM element. ${}^{t}\Delta\hat{\boldsymbol{\varepsilon}}$, the strain increment at each integration point in the DCS, is evaluated as

$$ {}^{t}\Delta\hat{\boldsymbol{\varepsilon}} = \hat{\boldsymbol{B}}\,{}^{t}\Delta\hat{\boldsymbol{\eta}} \tag{7} $$

where $\hat{\boldsymbol{B}}$ denotes the matrix of the strain-displacement relations in the DCS.

Step II.c: Stress evaluation

The stress increment at each element integration point in the DCS can be evaluated as

$$ {}^{t}\Delta\hat{\boldsymbol{\sigma}} = \hat{\boldsymbol{D}}\,{}^{t}\Delta\hat{\boldsymbol{\varepsilon}}\ (\,{}^{t}\Delta\hat{\boldsymbol{\sigma}} = \hat{\boldsymbol{D}}_p\,{}^{t}\Delta\hat{\boldsymbol{\varepsilon}}) \tag{8} $$

where $\hat{\boldsymbol{D}}$ and $\hat{\boldsymbol{D}}_p$ represent the elastic and elastoplastic constitutive matrices, respectively, in the DCS. For plastic materials, the radial return-mapping algorithm [Simo and Hughes (1998)] is adopted to determine the actual form of $\hat{\boldsymbol{D}}_p$. Iterations on plastic state variables within the path unit are usually required.

Step II.d: Calculation of the element equivalent forces

Based on the principle of virtual work, one can calculate the element equivalent forces ${}^{t}\hat{\boldsymbol{f}}_e$ in the DCS as follows:

$$ {}^{t}\hat{\boldsymbol{f}}_e = \int_V \hat{\boldsymbol{B}}^T({}^{t-\Delta t}\hat{\boldsymbol{\sigma}} + {}^{t}\Delta\hat{\boldsymbol{\sigma}})\,dV \tag{9} $$

Subsequently, these forces need to be transformed back into the global coordinate system to obtain ${}^{t}\boldsymbol{F}_e^{int}$.

Step II.e: Assembling composite forces for the particles

For an arbitrary particle α at time t, the external loads (${}^{t}\boldsymbol{F}_\alpha^{ext}$) and the equivalent internal forces of the connected elements need to be accumulated to determine the final composite force ${}^{t}\boldsymbol{F}_\alpha$. This calculation is described as follows:

$$ {}^{t}\boldsymbol{F}_\alpha = {}^{t}\boldsymbol{F}_\alpha^{ext} - \sum_{i=1}^{n} {}^{t}\boldsymbol{F}_i^{int} \tag{10} $$

where ${}^{t}\boldsymbol{F}_i^{int}$ denotes the equivalent force of the *i-th* element connected to α and n is the total number of elements connected to α.

Step III: Solving the kinematic equations for the particles

From Eq. (3), the particle displacement for the next step can be evaluated. The iterative process is terminated if the predefined end of the considered time period has been reached or if the conditions for termination are met; otherwise, the process returns to step II for the next iteration.

The iterative process of the FPM as described above has a wide range of versatility. First, the kinematic equilibrium equations applied for the particles suggest that by nature, the FPM takes a dynamic approach to structural analysis. Static problems can be simply considered as special cases in which the external loads are time independent and fictitious damping effects are set only to accelerate the process of achieving the final equilibrium state. Furthermore, the only difference between elastic and elastoplastic analysis lies in whether plastic constitutive theory is used when determining the stress state. Finally, the introduction of the fictitious reverse motion and path units ensures more accurate calculation of the element equivalent forces when strong nonlinearities are involved. After minor adjustments, all of these different types of analysis can be performed in a single generic computational framework for the FPM.

In addition, the iterative process of the FPM is largely decoupled within each path unit. For example, the kinematic equations can be processed individually for each particle and even each DOF, and the element-updating process (Steps II.a-II.d) is self-reliant between elements and even between integration points. These features facilitate the parallelization of the FPM, as no extra work is needed for decoupling.

The proposed generic parallelized computational framework for the FPM is depicted in Fig. 3, which presents a descriptive summarization of the steps introduced above. The parallel implementations of each step of computation are the main focus of this paper and will be comprehensively discussed in Section 3.

3 GPU parallelization of the FPM

Based on the fundamentals of the FPM theory and the generic parallelized computational framework proposed in the previous section, a GPU-accelerated parallelized solver system for the FPM has been implemented as the numerical analysis module for the universal computational FPM platform developed in our previous work. As explained in the introduction, CUDA, a proprietary GPU-oriented programming language supported by NVIDIA, was chosen for this work. The GPU implementations developed in CUDA C for the proposed FPM solver system are discussed in this section

3.1 Parallelized solver system for the FPM

A typical GPU contains a certain number of streaming multiprocessors (SMs), and a single SM consists of multiple streaming processors (SPs). The smallest units of parallel executions in a GPU device are called threads. In groups of 32 (called warps), threads are scheduled into thread blocks, which form the overall thread grid for each parallel execution. In a manner that is transparent to the user in terms of hardware, a multithreaded program

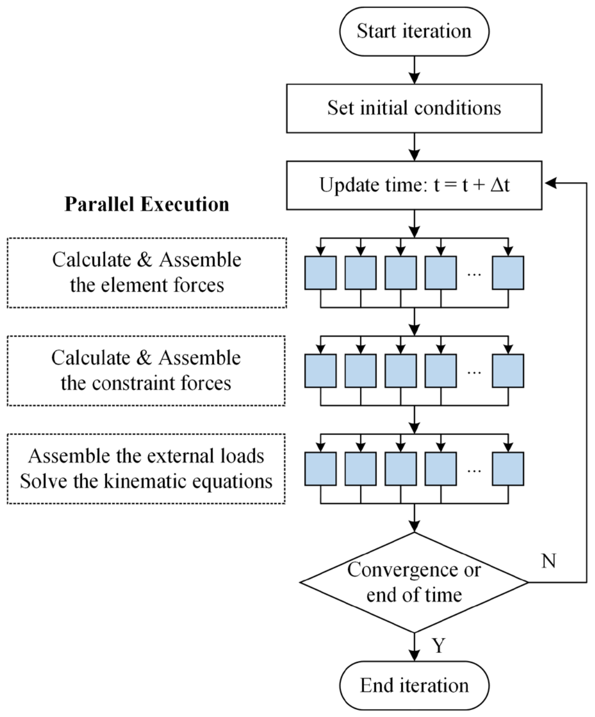

Start iteration
Set initial conditions
Update time: t = t + Δt
Parallel Execution
Calculate & Assemble the element forces
Calculate & Assemble the constraint forces
Assemble the external loads Solve the kinematic equations
Convergence or end of time
N
Y
End iteration

Figure 3: Generic parallelized computational framework for the FPM

is partitioned into blocks in each SM, and these blocks are executed independently of each other. Each thread can use the registers provided in each SM with the largest bandwidth and has its own private local memory; in addition, each block has a shared memory that is visible to all threads in the block. All threads have access to the same global memory.

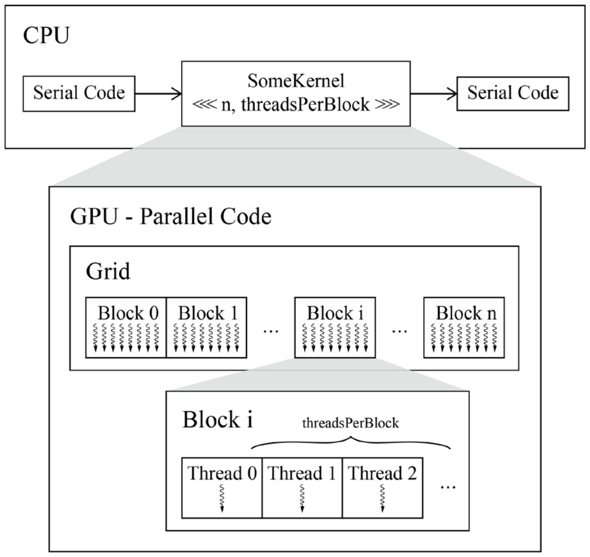


Figure 4: Generic GPU implantation of CUDA

A generic GPU implantation with CUDA is shown in Fig. 4. For a more detailed introduction to the CUDA programming model, the reader is referred to NVIDIA [NVIDIA (2019)]. For a multithreaded C++ program, CUDA threads are executed on a physically separate device that operates as a coprocessor to the host running the program. The host and the device maintain their own separate memory spaces. The host code, executed on one or more CPUs, manages the allocation and deallocation of device memory as well as the data transfer between the host and device memory. The device code mainly consists of a set of device functions, called kernels, that run in CUDA threads. The host controls when and how the device kernels are executed.

To deeply integrate the CUDA programming model, a more sophisticated parallelized computational framework has been developed for the proposed FPM platform, whose architecture is illustrated in Fig. 5. This platform is fully equipped with a preprocessing module for 3D modeling and a postprocessing module for result display; however, the GPU-accelerated solver system is the main focus of this paper.

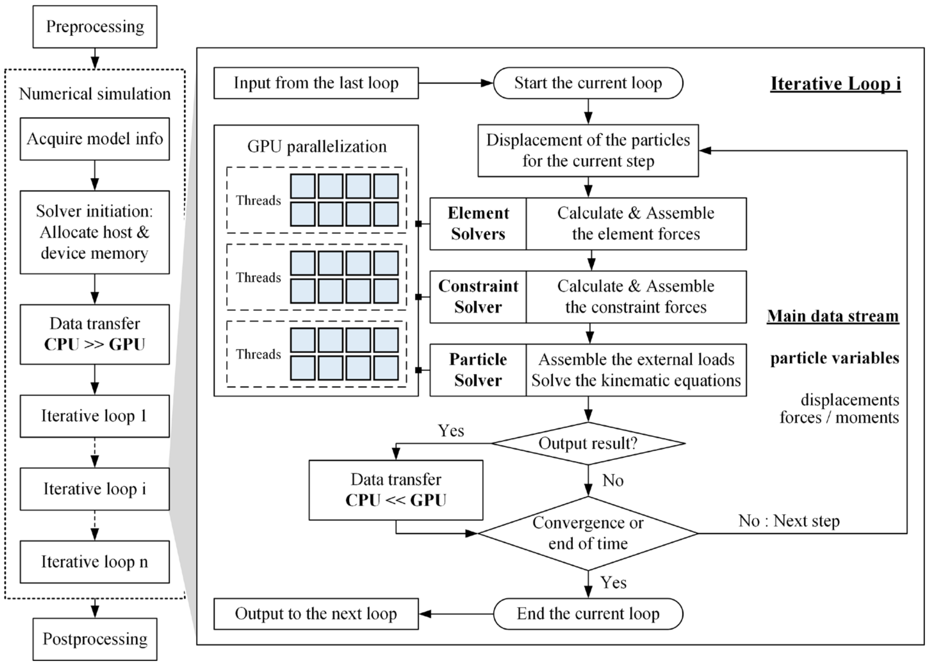


Figure 5: Architecture of the proposed FPM platform

Three types of FPM solvers have been developed: element solvers for different element types, a constraint solver, and a particle solver. When analysis begins, all model-related data are transferred to these FPM solvers for preparation and initiation, and the buffers for calculations are allocated in the host memory and transferred to the device memory. Two kinds of buffers are allocated. The first are the arrays for particle-related variables, such as the initial positions, displacements, and composite forces. Accessible by all solvers, these arrays form the main data stream that is iteratively processed throughout the whole computational process. In addition, each solver manages its own private buffers. For example, the arrays for temporary variables that are generated during the element-updating process for a specific type of FPM element are accessible only by the corresponding element solver.

As noted in Section 2, each analysis process in the FPM consists of an iterative loop of updating the main data stream of particle variables. The actual form of the iterative loop varies for different types of FPM analyses, such as static and dynamic analyses, but they share the same general structure (Fig. 4). For a specific analysis, a queue of iterative loops for different types of analysis can be scheduled and executed in order, and the main data stream of particle variables is constantly being updated or exported in this queue, as illustrated in Fig. 5.

Within each time step of an iterative loop, Steps II-III in Section 2 are executed in one or several CUDA kernels managed by corresponding solvers. As two of the most crucial steps, the element-updating process is carried out in the various element solvers, and the kinematic equations for the particles are solved in the particle solver. The GPU implementations of these two types of solvers are elaborated in Section 3.2.

3.2 GPU implementations of the FPM solvers

Conveniently, most of the computational tasks of the FPM are already decoupled to varying degrees, and the formulations for each task are rather straightforward. Following Steps II-III in Section 2, each task is implemented in several CUDA kernels. For concise description, the kernels for each task will be referred to as a single generic kernel. With this descriptive convention, for the GPU implementations of the FPM solvers, only two problems must be addressed:

- **Execution configuration:** How to launch kernels with the desirable number of blocks and threads.
- **Memory management:** How to manage memory storage to achieve the ideal throughput.

In the CUDA programming model, the host controls when and how the device kernels are executed by specifying the number of blocks and the number of threads per block. The execution configuration syntax for a given kernel is

$SomeKernel{<}{<}{<}N_{block}, N_{tpb}{>}{>}{>}(parameters\ \ldots)$

where N_{block} and N_{tpb} denote the number of blocks and the number of threads per block, respectively. When a kernel call is made with a specific execution configuration pair of N_{block} and N_{tpb}, a grid of thread blocks is automatically generated in the device. With a total of $N_{block} \times N_{tpb}$ threads, the same kernel is executed in each thread independently. It is apparent that a larger number of threads per block will result in a larger number of threads in total. However, each SM contains only a limited number of registers. If a block contains more threads or if more registers per thread are used in the kernel, then fewer blocks can be processed synchronously in a single SM since more registers are needed in each block. Therefore, all of the above factors must be considered when determining N_{tpb} to reach a satisfactory degree of concurrency. Based on numerous performance tests for most of the kernels implemented in this work, it has been found that 128-256 threads per block results in the best efficiency. Thus, for convenience, N_{tpb} is set to a constant value of 128 for all kernels.

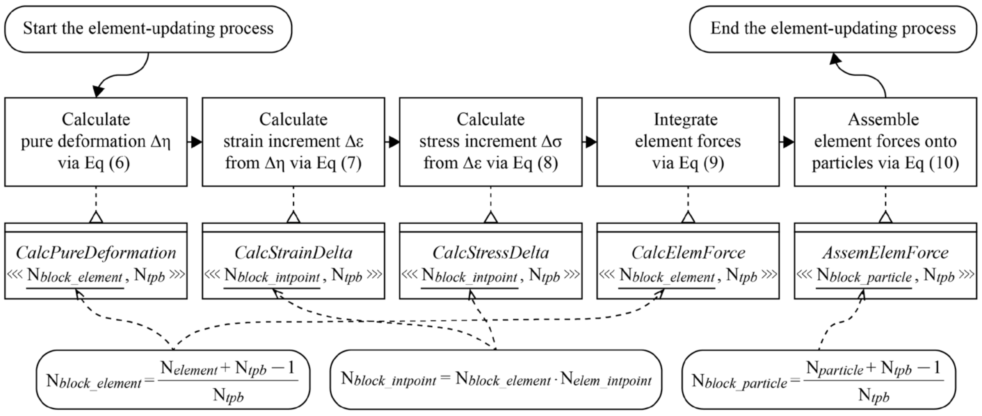
The total number of threads, denoted by N_{thread}, can be easily determined for each kernel. As noted in Section 2, each step of the iterative process of the FPM pipeline possesses its own specific type of independence, based on which the total number of threads assigned to the corresponding kernel can be determined. The types of independence for each step are listed below:

- **Pure deformation evaluation:** independence between elements.
- **Strain and stress evaluations:** independence between integration points.
- **Calculation of the element equivalent internal forces:** independence between elements.
- **Solving the kinematic equations for the particles:** independence between particles.

As a result, the numbers of particles, elements and integration points are used as N_{thread} for the different kernels to reflect their particular types of independence. Then, for a given constant value of N_{tpb}, the number of blocks N_{block} can be evaluated:

$$N_{block} = (N_{thread} + N_{tpb} - 1)/N_{tpb} \tag{11}$$

Fig. 6 shows the execution configurations for the element-updating process (steps II in Section 2), for which five kernels are implemented. For the kernels $CalcPureDeformation$ and $CalcElemForce$, each thread is responsible for all calculations for a single element, and N_{thread} in Eq. (11) is set equal to the total number of elements to obtain $N_{block\_element}$, the corresponding number of blocks. For the kernels $CalcStrainDelta$ and $CalcStressDelta$, each thread performs all calculations for a single integration point. The corresponding number of blocks, or $N_{block\_integpoint}$, is the product of $N_{block\_element}$ and the number of integration points used for the corresponding element type. The kernel for assembling element equivalent forces is slightly different and will be explained later.



Start the element-updating process
End the element-updating process
Calculate pure deformation Δη via Eq (6)
Calculate strain increment Δε from Δη via Eq (7)
Calculate stress increment Δσ from Δε via Eq (8)
Integrate element forces via Eq (9)
Assemble element forces onto particles via Eq (10)
CalcPureDeformation «« $N_{block\_element}$, N_{tpb} »»
CalcStrainDelta «« $N_{block\_intpoint}$, N_{tpb} »»
CalcStressDelta «« $N_{block\_intpoint}$, N_{tpb} »»
CalcElemForce «« $N_{block\_element}$, N_{tpb} »»
AssemElemForce «« $N_{block\_particle}$, N_{tpb} »»
$N_{block\_element} = \frac{N_{element} + N_{tpb} - 1}{N_{tpb}}$
$N_{block\_intpoint} = N_{block\_element} \cdot N_{elem\_intpoint}$
$N_{block\_particle} = \frac{N_{particle} + N_{tpb} - 1}{N_{tpb}}$

Figure 6: Parallel execution configurations for the element-updating process

Once the kernels have been executed with the appropriate configurations, each thread can determine its own global index (denoted by idx) among all scheduled threads in the kernel code as follows:

$$idx = blockDim \times blockIdx + threadIdx \tag{12}$$

where $threadIdx$ is the local thread index in the current block, $blockIdx$ is the block index, and $blockDim$ is the block size, which is equal to N_{tdp}. These values can all be acquired via CUDA APIs in the device code. For instance, if a kernel with a number of blocks equal to $N_{block\_element}$, such as $CalcPureDeformation()$, is executed, each thread corresponds to one element, and the value of idx in the kernel is equal to the index of the element processed by that thread. Similarly, for a kernel such as $CalcStressDelta$, idx is the global index of the corresponding integration point.

The global thread index idx is mainly used within the kernel to access the corresponding values from the global or private calculation-related buffers in the device's global memory. Because it has the largest storage capacity, the global device memory is used for most buffers storing the historical variables of particles and elements. However, the cost of each individual memory access is relatively high. The optimal throughput can be achieved if the memory access patterns are suitable for coalescence. In simple terms, when adjacent threads access successive memory addresses in the global memory, each warp coalesces all memory accesses within that warp into a single access. To achieve such coalescence of memory accesses, an adaptation of the structure-of-arrays (SoA) storage pattern suggested by Cook [Cook (2013)] is applied to all calculation-related data buffers in the global memory to achieve the optimal throughput. The kernels need only to read/write the corresponding historical values from the buffers using *idx* as the array index and implement the formulations given in Section 2.

The step of assembling element equivalent forces onto particles is the only step in the FPM where the formulations are not self-reliant by the standard of particles or elements. Adjacent elements must share particles, so each particle is required to gather internal forces from all the connected elements. There are two possible solutions toward implementing this step in a multithreaded way. The first is to let each thread accumulate a single element's own internal forces onto the corresponding particles, and the kernel is launched by the total number of elements concurrently. Inevitably, there must be a point where different threads (i.e., different elements) are writing at the same memory address (i.e., composite forces of the shared particle), which means that this approach is not thread-safe. Atomic operations can be used in device code to ensure safe access patterns at the cost of performance since they force concurrent write actions at the same address to be serialized. Hardware support are also required in this case.

Another approach is considered in this work. Each thread is responsible for gathering the forces for a single particle from all the neighboring elements. According to Eq. (10), the step of assembling element equivalent forces follows the steps below:

(i) Connectivity arrays are generated as private buffers in the global memory during the preparation stage. As shown in Fig. 7, these arrays mainly store (a) the number of connected elements for each particle, (b) the global indexes of all connected elements for each particle, and (c) the corresponding local index of the particle inside the connected element for each particle-element connection pair.

(ii) The kernel $AssemElemForce$ is concurrently executed for the total number of particles; thus, the thread index idx represents the corresponding particle. The index idx is used as a position index to retrieve the number of connected elements ($eCount$) to initialize the loop over all connected elements.

(iii) Within the loop for each connected element, first, the element's global index (eId) and the local index ($eParticleId$) of particle idx in element eId are retrieved. Then, the correct element internal force is acquired using eId and $eParticleId$. Finally, this internal force is incorporated into the composite force for idx.

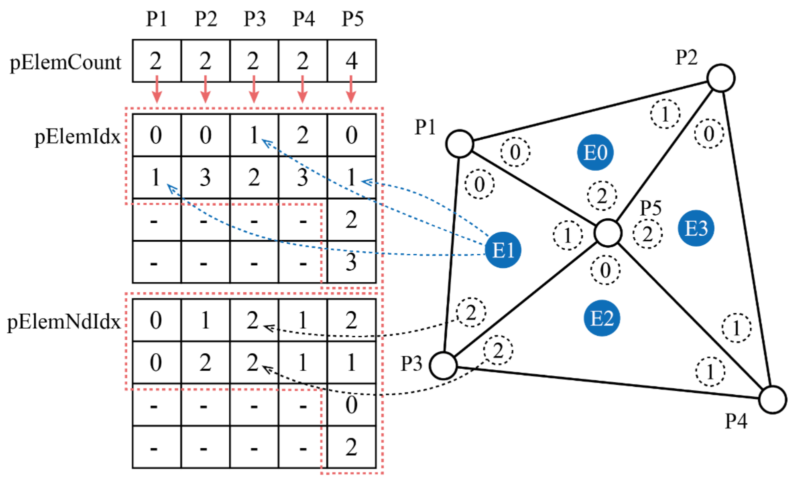
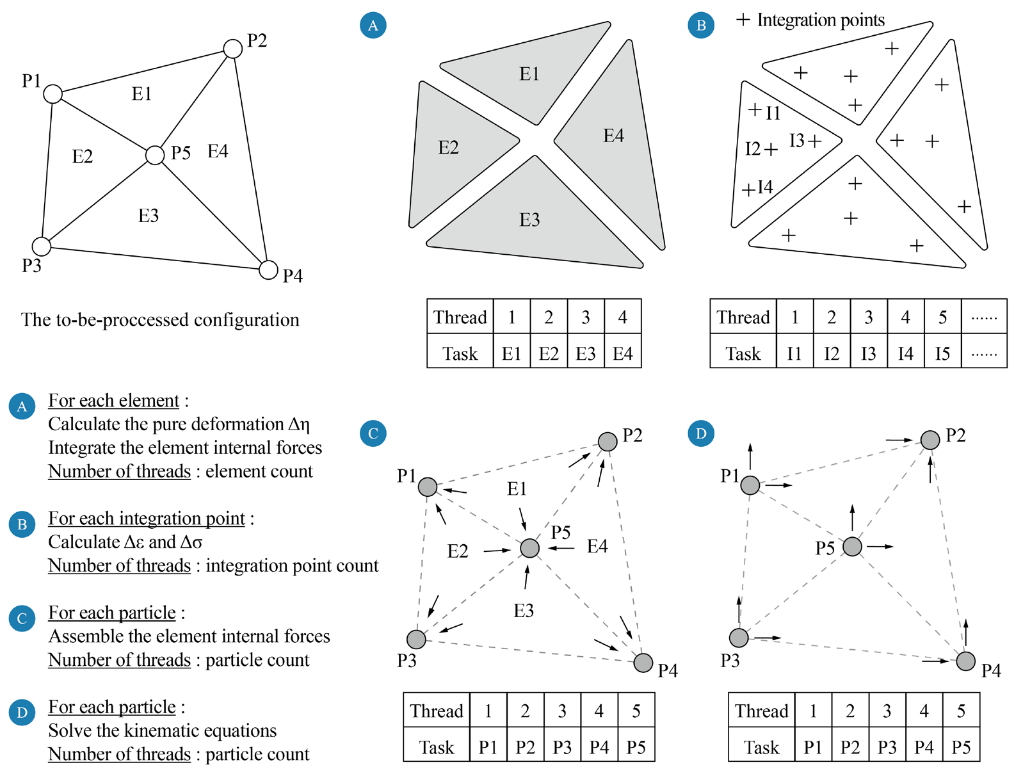


Figure 7: Data structures and memory allocations for assembling element equivalent forces

Algorithm 1: Assembling the element internal forces thread-safely

```
calculate the thread index $idx$ via Eq. (12)
read the composite force for particle $idx$ as $particleF$
read $eCount$ from the connectivity arrays
for $i = 0$ to $eCount$
    read the element index as eId using $(i, idx)$
    read the local particle index as $eParticleId$ using $(i, idx)$
    read the element force as $eInternalF$ using $(eId, eParticleId)$
    add $eInternalF$ to $particleF$
write $particleF$ back to the array of composite forces
```



Thread	1	2	3	4
Task	E1	E2	E3	E4

Thread	1	2	3	4	5	......
Task	I1	I2	I3	I4	I5	......

Thread	1	2	3	4	5
Task	P1	P2	P3	P4	P5

Thread	1	2	3	4	5
Task	P1	P2	P3	P4	P5

A For each element :
Calculate the pure deformation $\Delta\eta$
Integrate the element internal forces
Number of threads : element count

B For each integration point :
Calculate $\Delta\varepsilon$ and $\Delta\sigma$
Number of threads : integration point count

C For each particle :
Assemble the element internal forces
Number of threads : particle count

D For each particle :
Solve the kinematic equations
Number of threads : particle count

Figure 8: Schematic diagram of the GPU implementations of the FPM tasks

This approach for assembling element internal forces is described in the form of pseudocode in Algorithm 1. Since all variables of connectivity are stored separated for each particle, this is a thread-safe approach that remains effective regardless of the element type. For different element geometries, only the sizes of the connectivity arrays vary; the kernel code requires no further modification. However, this approach does consume additional memory space in the device for the connectivity arrays. In addition, since the number of connected elements is different for each particle based on the topology of the model, threads in the same warp might follow different execution paths, resulting in a certain loss of performance.

The final step of the FPM in each path unit is to solve the kinematic equations for the particles. The corresponding kernel is concurrently executed for the total number of particles, and each thread is responsible for the calculations for a single particle's movement, in accordance with Eqs. (3) and (4).

A schematic diagram of the GPU implementations of the main tasks of the FPM is presented in Fig. 8. The parallelization of the other tasks, such as evaluating and assembling the constraint forces, which is the purpose of the constraint solver, is achieved via a similar approach and will not be discussed in detail in this paper.

4 Numerical examples and efficiency tests

4.1 Speedup ratio tests

In the field of parallel computing, the speedup ratio is commonly used as an indicator of the efficiency of parallel implementation for a given algorithm. This parameter is defined as

$$S = T_s/T_p \tag{13}$$

where S denotes the speedup ratio, T_s is the time cost of the serialized version of the algorithm of interest, and T_p is the time cost of the same algorithm after parallelization.

Based on the techniques introduced in the previous section, three types of GPU-accelerated element solvers are implemented in the proposed FPM platform: a 2-particle Euler-Bernoulli (EB) beam solver, a 3-particle 20-integration-point triangular shell solver, and an 8-particle hexahedral solid solver. To test the efficiency of each solver, serialized counterparts have also been developed. Three sets of numerical models are considered, one for each type of FPM element, as shown in Fig. 9: (a) a single-layered lattice shell modeled with beam elements, (b) a thin spherical shell modeled with shell elements, and (c) a solid cantilever beam modeled with hexahedral solid elements. Under linearly increasing loads, each of several models with different meshes was analyzed using both the parallelized solvers and their serialized counterparts to obtain the corresponding elastically deformed shapes. The serialized solvers were tested on an Intel(R) Core(TM) i7-2600 CPU @3.4 GHz, while the parallelized solvers were tested on an NVIDIA GeForce GTX 760 GPU with 1152 SPs. For all models, the time costs for 2000 iterations were determined.

Fig. 10 shows the speedup ratios for models with different element types and meshes. For all element types, the speedup ratio curves show the same pattern. For models with fewer than 10^4 elements, the speedup ratio rapidly increases with an increasing number of elements. Once the number of elements exceeds the threshold at approximately 10^4, the speedup ratios for the different element types gradually stabilize at different levels. The stabilized speedup ratios are 8 for beams, 25 for hexahedral solids, and 48 for triangular shells. The differences in the stabilized speedup ratios between the different element types are fairly easy to understand. The serialized versions of the FPM solvers simply evaluate the formulations (given in Section 2) element by element, particle by particle, or integration point by integration point, while the parallelized solvers execute these calculations concurrently. Accordingly, for an element type that contains more integration points per element or requires more complicated calculations for the element-updating process, a higher speedup ratio can be achieved.

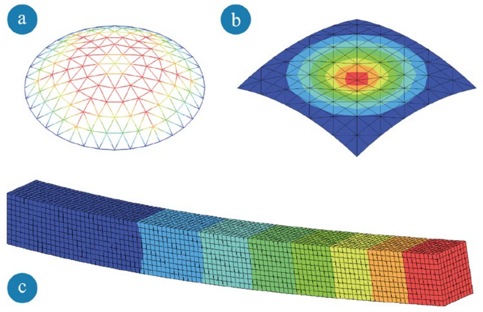


Figure 9: Models for speedup ratio tests: (a) a lattice shell, (b) a spherical shell and (c) a cantilever beam

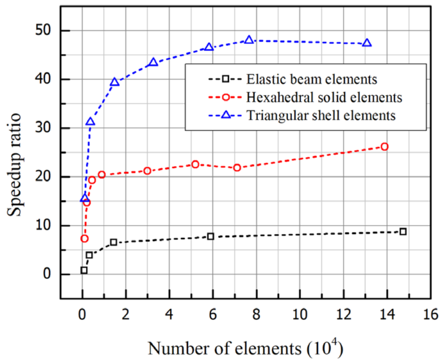


Figure 10: Speedup ratios achieved for different types of FPM elements

4.2 Performance and efficiency tests

Although the speedup ratio measures the speed improvement of the FPM after parallelization, the efficiency of the GPU-accelerated FPM still cannot be intuitively understood. In this section, the proposed FPM platform is compared with the Abaqus software for code validation and efficiency assessment. While this kind of efficiency comparison is not always fair and meaningful since equivalence cannot be fully guaranteed, it is nevertheless useful for estimating the achievable performance relative to a common and widely known software tool [Bartezzaghi, Cremonesi, Parolini et al. (2015)]. Three numerical examples are presented to test the performance of the proposed GPU-accelerated FPM solvers: two for the shell solver and one for the solid solver. Each numerical example was modeled with different meshes and analyzed using both the FPM platform and Abaqus. Due to the explicit nature of the FPM, the explicit dynamic solution step in Abaqus was selected to ensure that the results would be comparable. All controllable configuration parameters in Abaqus, including the fixed time increment (smaller than the critical time increment) and duration, the output settings (only nodal displacements were exported 100 times) and the precision setting (double precision), were set to be identical to those of the FPM platform. The test environment was a Windows PC with an Intel(R) Core(TM) i7-4790K CPU @4.00 GHz and an NVIDIA GeForce GTX 980 Titan GPU. This model has 3072 SPs and 6 GB of VRAM.

4.2.1 Spherical dome under impact pressure

A spherical dome under impact pressure is a typical benchmark for the nonlinear dynamic analysis of thin shells. The geometry and material properties of the dome are shown in Fig. 11; in this model, a bilinear isotropic hardening plastic material is considered. The dome is clamped on the edges, and a uniform pressure of 600 psi is applied to the upper surface of the dome for 1 ms. A dynamic analysis with a fixed time increment of 10^{-7} s was performed for 1 ms of physical time, resulting in 10000 iterations in total. The computation times of the FPM platform and Abaqus for different meshes are listed in Tab. 1.

Table 1: Spherical dome: computation times of the FPM platform and Abaqus

Mesh	DOFs	Computation time (s)		Unified computation time ($s/10^4$ elements/10^4 iterations)	
		FPM	Abaqus	FPM	Abaqus
1	2934	20.1	3.4	216.6	36.6
2	9270	25.1	10.7	84.8	36.1
3	22098	44.0	28.2	61.2	39.3
4	54486	66.8	74.5	37.3	41.6
5	88038	85.9	122	29.9	42.0

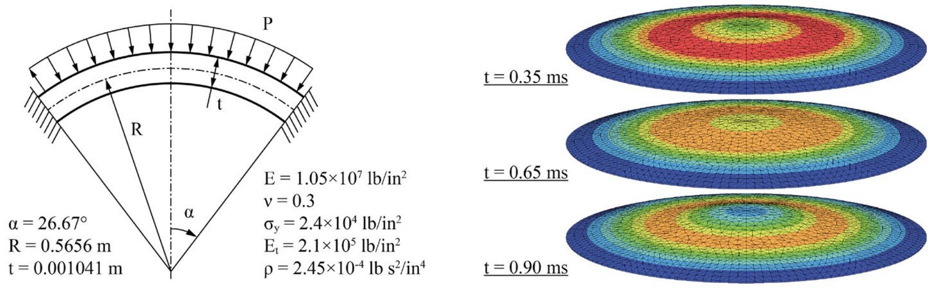
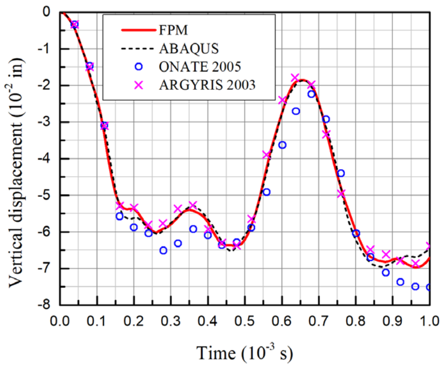


Figure 11: Spherical dome: geometry and material properties (left) and deformed contours for the Mesh 1 model (right)



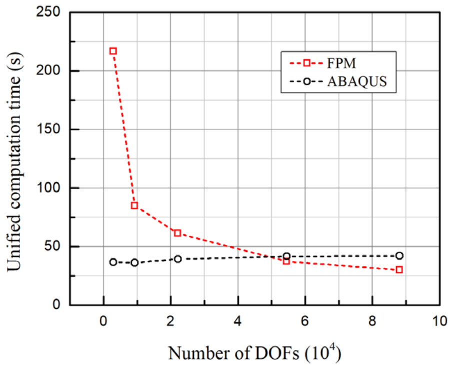


Figure 12: Spherical dome: history of central deflection for the Mesh 1 model

Figure 13: Spherical dome: unified computation times of the FPM platform and Abaqus for different meshes

Fig. 12 shows the time history of the central deflection results obtained from the FPM platform. The FPM results show perfect agreement with the Abaqus results and are also consistent with Argyris et al. [Argyris, Papadrakakis and Mouroutis (2003); Oñate and Flores (2005)], thus verifying the accuracy of the FPM solver in performing elastoplastic dynamic analysis for thin shells.

The unified computation times of the two platforms are compared in Fig. 13. The unified computation time is defined as the computation time required for processing 10^4 elements over 10^4 iterations. A smaller value of the unified computation time indicates a higher speed. As shown in Fig. 13, 50000 DOFs (approximately 10^4 elements) seems to be the turning point below which Abaqus offers the higher speed for calculations. As the speed of Abaqus decreases with an increasing number of DOFs, that of the GPU-accelerated FPM platform rapidly increases and ultimately exceeds the speed of Abaqus. This threshold is consistent with the results presented in Section 4.1. Once the scale of the model exceeds this threshold, the GPU will run at full capacity, allowing the FPM platform to provide a higher computation speed than Abaqus, as further demonstrated in the following examples.

4.2.2 Pinched cylinder

To test the performance of the FPM solver for more challenging cases, a pinched cylinder adapted from Bartezzaghi et al. [Bartezzaghi, Cremonesi, Parolini et al. (2015)] was numerically modeled to test cases with much larger numbers of elements. The cylinder, with a radius of 1.016 m, a length of 3.048 m and a thickness of 0.03 m, is clamped at one end and pinched under two opposing forces on the other end. Only the elastic case is considered in this example of geometric nonlinearity, in which the Young's modulus is 20.685 MPa and the Poisson coefficient is 0.3. Five models with different meshes (Mesh 1 to Mesh 5) were generated, with numbers of DOFs varying from 30600 to 1080000. A fixed time increment of 10^{-5} s was adopted for a physical time of 1 s.

The parallel computing capability of Abaqus was used in this example for comparison. As explained in Section 2, the FPM is, by nature, an explicit numerical method. Ideally, the performance of the GPU-accelerated FPM solver should be compared with that of the GPU-accelerated explicit solver in Abaqus. However, the GPU-based parallel computing functionality in Abaqus is available only for implicit solution steps; explicit dynamic steps can be accelerated only with the CPU. The results of two explicit methods will be more comparable than those of an explicit method and an implicit method since the computational frameworks and computational costs are vastly different for implicit and explicit methods. As a result, the comparisons presented here are between the results of the GPU-accelerated FPM solver and the CPU-accelerated explicit FEM solver in Abaqus, and the latter can be regarded only as a reference. A similar treatment has previously been presented in Bartezzaghi et al. [Bartezzaghi, Cremonesi, Parolini et al. (2015)]. In Abaqus, each model was separately analyzed with 1, 4 and 8 CPU cores. The computation times for each configuration are listed in Tab. 2.

Fig. 14 shows the deformed shapes of the Mesh 1 model under different forces as obtained from the FPM platform, and Fig. 15 compares the load-displacement curve with the results of Bartezzaghi et al. [Bartezzaghi, Cremonesi, Parolini et al. (2015); Ibrahimbegovic, Brank and Courtois (2001)], with which it shows perfect agreement. The curves of the unified computation time versus the number of DOFs are presented in Fig. 16. Since the scale of the models ensured that the GPU would be fully loaded, the FPM solver achieved a higher speed than Abaqus for all configurations. Compared with the Abaqus results for a single CPU core, the execution on the FPM platform was approximately 4.8 times faster on average, and the corresponding factor is approximately 2.6 for the cases with 4 and 8

CPU cores. It can also be observed that the computation speed of the FPM platform for this elastic case is approximately 3 times faster than that for the elastoplastic case in Section 4.2.1; this is reasonable since the elastoplastic analysis requires iterations for plastic state variables within each path unit.

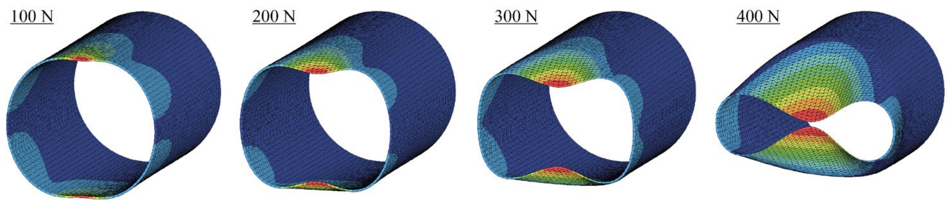


Figure 14: Pinched cylinder: deformed contours for the Mesh 1 model

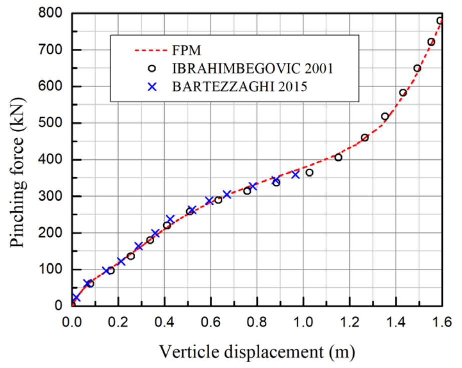


Figure 15: Pinched cylinder: load-displacement curve

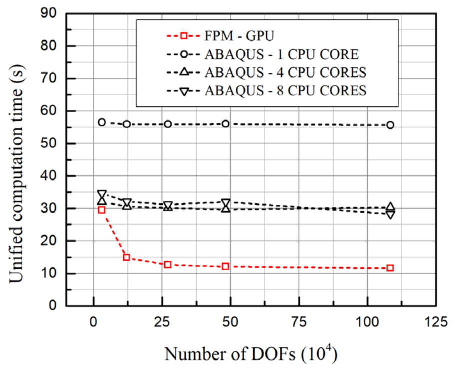


Figure 16: Pinched cylinder: unified computation times of the FPM platform and Abaqus for different meshes

Table 2: Pinched cylinder: computation times of the FPM platform and Abaqus

DOFs	Computation time (s)				Unified computation time ($s/10^4$ elements/10^4 iterations)			
	FPM GPU	Abaqus 1 core	Abaqus 4 cores	Abaqus 8 cores	FPM GPU	Abaqus 1 core	Abaqus 4 cores	Abaqus 8 cores
30600	294.4	564.8	320.7	346.6	29.4	56.5	32.1	34.7
121200	591.2	2236	1222	1284	14.8	55.9	30.5	32.1
271800	1137	5030	2714	2812	12.6	55.8	30.2	31.2
482400	1932	8960	4743	5126	12.1	56.0	29.6	32.0
1080000	4177	20036	10923	10180	11.6	55.6	30.3	28.3

4.2.3 Compressed annulus

A simple annulus was numerically analyzed to test the performance of the solid element solver in the FPM platform. The annulus, whose geometry and material properties are shown in Fig. 17, is compressed at two opposing sides by a dynamic pressure that increases linearly for 0.005 s until it reaches 5.25 MPa and then remains constant. Following the same procedures described for the previous examples, each meshed annulus model (Mesh 1 to Mesh 4) was processed using the FPM platform and Abaqus. A time increment of

2.0×10^{-6} s was set to capture the dynamic response of the annulus within 0.02 s. The effect of damping was ignored. The computation times for each mesh are listed in Tab. 3.

Table 3: Compressed annulus: computation times of the FPM platform and Abaqus

DOFs	Computation time (s)				Unified computation time ($s/10^4$ elements/10^4 iterations)			
	FPM GPU	Abaqus 1 core	Abaqus 4 cores	Abaqus 8 cores	FPM GPU	Abaqus 1 core	Abaqus 4 cores	Abaqus 8 cores
8064	56.6	27.1	12.5	26.1	295.0	141.2	65.1	136.7
54912	51.6	226.9	97.8	135.5	33.6	147.7	63.6	88.1
403200	191.9	2002.4	701.4	717.8	15.6	163.0	57.1	58.4
1747584	1296.7	14442.8	5571.2	5749.2	13.2	146.9	56.7	58.5

Fig. 17 shows the deformed shapes of the annulus at different stages of loading for the Mesh 1 model. The calculated horizontal displacement histories of the center particle in the compressed area as obtained from the FPM platform and Abaqus are compared in Fig. 18, and the results show perfect agreement.

The computation speeds are compared in Fig. 19. For the Mesh 1 model, the scale of the model is relatively small, meaning that the number of launched threads is far less than the full capacity of the hardware. For this reason, the computation time results for Mesh 1 are not included in this figure. It can be observed that the execution on the FPM platform was approximately 11.2 times faster than the Abaqus execution with a single core and 4.5 times faster than the executions for the multithreaded configurations with 4 and 8 cores. These results prove that the proposed GPU-accelerated parallel strategy for the FPM shows promising application prospects for large-scale explicit simulations.

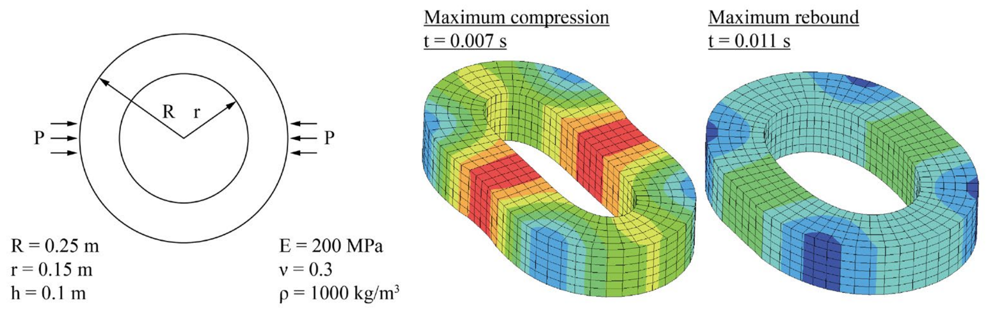


Figure 17: Compressed annulus: geometry and material properties (left) and deformed contours for the Mesh 1 model (right)

To obtain a broad understanding of the time consumption of the GPU-accelerated FPM solvers, the steps of computation described in Section 3.2 were timed separately for the pinched cylinder and the compressed annulus. Figs. 20 and 21 show the time consumption percentages for the element-updating process, for solving the kinematic equations and for data transfer between the host and device. It can be seen that on average, more than half of the computation time is consumed during the element-updating process, as expected. In

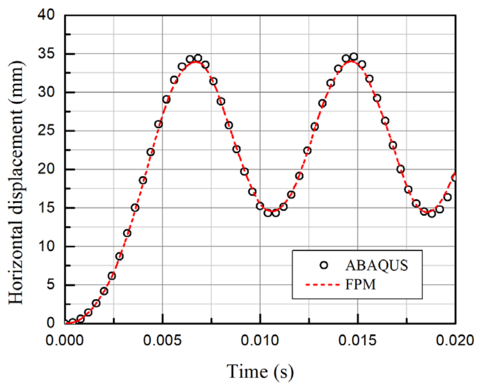


Figure 18: Compressed annulus: horizontal displacement history of the center particle in the compressed area

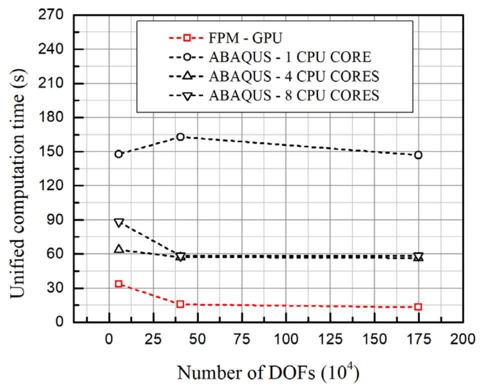


Figure 19: Compressed annulus: unified computation times of the FPM platform and Abaqus for different meshes

fact, the kernels for the element-updating process show severe register spilling. Register spilling is a common cause of performance degradation in which the available registers are not sufficient for all local variables in the kernel; thus, some of these variables are stored in the local memory, which has a much higher latency than the registers. High register usage can also lower the kernel's occupancy, which is defined as the ratio of the number of active warps on an SM to the maximum number of active warps supported by the SM. Higher occupancy corresponds to higher efficiency in most cases. The ideal number of registers for each kernel is approximately 48 for the current GPU model and block size. The actual levels of register usage and occupancy for the main kernels are listed in Tab. 4, from which we can see that most of the kernels use more than the ideal number of registers. Thus, substantial optimization is still possible for these kernels. By reusing local variables or reducing divergence in the warps, the efficiency of the GPU-accelerated FPM solvers could be further improved in future work.

Table 4: Register usage and occupancy for the main kernels

M task	Triangular shell		Hexahedral solid	
	Occupancy	Register usage	Occupancy	Register usage
Calculate pure deformation	50%	64	37.5%	78
Calculate strain and stress	50%	68	25%	110
Calculate element equivalent forces	31.25%	88	25%	128
Assemble element equivalent forces	25%	112	100%	30
Calculate particle translations	18.75%	152	18.75%	152
Calculate particle rotations	31.25%	88	-	-

The computation speed for the hexahedral solid elements is slower than that for the shell elements in these two examples. As shown in Tab. 4, the kernels for the hexahedral solid element solver require more registers, which explains the lower efficiency of this solver

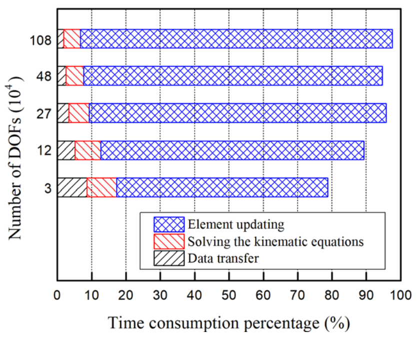


Figure 20: Pinched cylinder: time consumption for individual FPM tasks

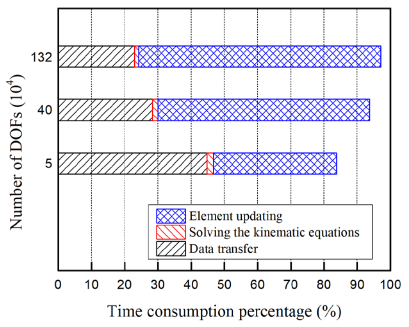


Figure 21: Compressed annulus: time consumption for individual FPM tasks

compared to that for the shell elements. Elements with more integration points require more threads to be launched for the strain/stress evaluations, while elements with simpler geometries require less effort for pure deformation extraction and fewer kernel variables. Compared to the hexahedral solid element, which has 8 integration points, the FPM shell element has more integration points (20) but a much simpler geometry (a triangle rather than a brick); thus, its faster computation speed is reasonable.

The efficiency of data transfer between the host and the device also requires attention. In fact, there is no concept of file I/O in a GPU; the results in the device memory must be transferred back to the host to be written into files or databases. This data transfer process has almost the lowest possible bandwidth, which makes it extremely time consuming. In the two tests presented above, only the displacements of the particles were exported out of the GPU memory. As shown in Fig. 20, for the example of the pinched cylinder, the cost of data transfer is quite low relative to the total time cost. The reason is that the number of particles is rather small compared to the number of elements; consequently, the exported data size is insignificant. However, for the example of the annulus, almost 40% of the time is spent on data transfer, as seen in Fig. 21. Due to the topology, the cost of transferring particle-related data is no longer insignificant. Furthermore, if the element results were to be exported, the cost of data transfer would dominate the whole process. This is a common problem faced by every GPU-accelerated system. The overall efficiency of the FPM solvers could be further improved if the latency of data transfer could be effectively reduced. In future research and development, the approach of overlapping the assignments for calculations and data transfer, as suggested by Cai et al. [Cai, Li and Liu (2018)], will be considered.

5 Conclusions

This paper has proposed a parallel strategy for explicit dynamic analysis using the FPM. Using the CUDA programming model, a GPU-accelerated computational platform for the

FPM has been developed, and its accuracy and efficiency have been validated for several numerical examples.

Through careful management of thread executions and memory access optimization, GPU implementations of the main tasks in the FPM pipeline have been developed, as elaborated in this paper, and GPU solvers for various types of FPM elements have been implemented. Performance tests show that, after parallelization, speedup ratios of 8, 25 and 48 are achieved with the proposed FPM platform for elastic beam, hexahedral solid and triangular shell elements, respectively. In general, a higher speedup ratio can be achieved for more complex elements with a larger number of integration points.

For examples involving explicit analyses of shells and solids, comparisons with Abaqus results obtained using 1 to 8 CPU cores validate the accuracy of the proposed platform and demonstrate speed improvements by factors of 2.6 to 11.2 for the GPU-accelerated FPM elements. It can be concluded that the proposed GPU-accelerated FPM platform provides faster performance than the multithreaded CPU architecture in Abaqus for large-scale explicit dynamic analysis and shows promising application prospects.

This work can serve as a starting point for future improvements and applications of the FPM. The implemented FPM platform can still be substantially improved to achieve more satisfactory performance. Optimizations with regard to device code delivery, register usage and I/O throughput will be needed in the future to achieve further improvement. Furthermore, additional types of FPM elements can be implemented using the parallel framework proposed in this paper. The FPM theories of fracture, contact and collision could also be parallelized in future studies. Simulations for such complex engineering problems, with high levels of discontinuity and nonlinearity, demand a high standard of efficiency, for which the GPU-accelerated FPM approach will be advantageous.

Acknowledgment: The authors gratefully acknowledge the financial support provided by the National Key Research and Development Program of China (Grant No. 2016YFC0800200), the National Natural Science Foundation of China (Grant Nos. 51578494 and 51778568) and the Fundamental Research Funds for the Central Universities (Grant No. 2019QNA4043).

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

Argyris, J.; Papadrakakis, M.; Mouroutis, Z. S. (2003): Nonlinear dynamic analysis of shells with the triangular element TRIC. *Computer Methods in Applied Mechanics and Engineering*, vol. 192, no. 26-27, pp. 3005-3038.

Bartezzaghi, A.; Cremonesi, M.; Parolini, N.; Perego, U. (2015): An explicit dynamics GPU structural solver for thin shell finite elements. *Computers & Structures*, vol. 154, pp. 29-40.

Bova, S. W.; Carey, G. F. (2000): A distributed memory parallel element-by-element scheme for semiconductor device simulation. *Computer Methods in Applied Mechanics and Engineering*, vol. 181, no. 4, pp. 403-423.

Cai, Y.; Li, G.; Liu, W. (2018): Parallelized implementation of an explicit finite element method in many integrated core (MIC) architecture. *Advances in Engineering Software*, vol. 116, pp. 50-59.

Cheik Ahamed, A. K.; Magoulès, F. (2017): Conjugate gradient method with graphics processing unit acceleration: CUDA *vs.* OpenCL. *Advances in Engineering Software*, vol. 111, pp. 32-42.

Cook, S. (2013): *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Elsevier, Morgan Kaufmann, Waltham, MA, USA.

Duan, Y. F.; Wang, S. M.; Wang, R. Z.; Wang, C. Y.; Shih, J. Y. et al. (2018): Vector form intrinsic finite-element analysis for train and bridge dynamic interaction. *Journal of Bridge Engineering*, vol. 23, no. 1, 04017126.

Duan, Y. F.; Wang, S. M.; Yau, J. D. (2019): Vector form intrinsic finite element method for analysis of train-bridge interaction problems considering the coach-coupler effect. *International Journal of Structural Stability and Dynamics*, vol. 19, no. 2, 1950014.

Georgescu, S.; Chow, P.; Okuda, H. (2013): GPU acceleration for FEM-based structural analysis. *Archives of Computational Methods in Engineering*, vol. 20, no. 2, pp. 111-121.

Gullerud, A. S.; Dodds Jr, R. H. (2001): MPI-based implementation of a PCG solver using an EBE architecture and preconditioner for implicit 3-D finite element analysis. *Computers and Structures*, vol. 79, no. 5, pp. 553-575.

Hallquist, J. O. (2006): *LS-DYNA Theory Manual*. Livermore Software Technology Corporation.

Hwu, W. W. (2011): *GPU Computing Gems Emerald Edition*, Elsevier, Morgan Kaufmann, Burlington, MA, USA.

Ibrahimbegovic, A.; Brank, B.; Courtois, P. (2001): Stress resultant geometrically exact form of classical shell model and vector-like parameterization of constrained finite rotations. *International Journal for Numerical Methods in Engineering*, vol. 52, no. 11, pp. 1235-1252.

Li, X.; Guo, X.; Guo, H. (2018): Vector form intrinsic finite element method for nonlinear analysis of three-dimensional marine risers. *Ocean Engineering*, vol. 161, pp. 257-267.

Luo, Y. Z.; Zheng, Y. F.; Yang, C.; Yu, Y.; Yu, F. et al. (2014): Review of the finite particle method for complex behaviors of structures. *Engineering Mechanics*, vol. 31, pp. 1-7, 23.

NVIDIA (2017): CUDA Zone: High Performance Computing.
https://developer.nvidia.com/cuda-zone.

NVIDIA (2019): CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/.

Oñate, E.; Flores, F. G. (2005): Advances in the formulation of the rotation-free basic shell triangle. *Computer Methods in Applied Mechanics and Engineering*, vol. 194, no. 21-24, pp. 2406-2443.

Papadrakakis, M.; Stavroulakis, G.; Karatarakis, A. (2011): A new era in scientific computing: domain decomposition methods in hybrid CPU-GPU architectures. *Computer Methods in Applied Mechanics and Engineering*, vol. 200, no. 13-16, pp. 1490-1508.

Pikle, N. K.; Sathe, S. R.; Vyavhare, A. Y. (2018): GPGPU-based parallel computing applied in the FEM using the conjugate gradient algorithm: a review. *Sādhanā*, vol. 43, no. 7, pp. 111.

Qi, J.; Li, K. C.; Jiang, H.; Zhou, Q.; Yang, L. (2015): GPU-accelerated DEM implementation with CUDA. *International Journal of Computational Science and Engineering*, vol. 11, no. 3, pp. 330.

Rao, S. C. S.; Kamra, R. (2018): A hybrid parallel algorithm for large sparse linear systems. *Numerical Linear Algebra with Applications*, vol. 25, no. 6, e2210.

Shih, C.; Wang, Y. K.; Ting, E. C. (2004): Fundamentals of a vector form intrinsic finite element: part III. convected material frame and examples. *Journal of Mechanics*, vol. 20, no. 2, pp. 133-143.

Simo, J. C.; Hughes, T. J. R. (1998): *Computational Inelasticity*. Springer, New York, USA.

The Khronos Group (2013): OpenCL-The open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/.

Ting, E. C.; Shih, C.; Wang, Y. K. (2004a): Fundamentals of a vector form intrinsic finite element: part I. basic procedure and a plane frame element. *Journal of Mechanics*, vol. 20, no. 2, pp. 113-122.

Ting, E. C.; Shih, C.; Wang, Y. K. (2004b): Fundamentals of a vector form intrinsic finite element: part II. plane solid elements. *Journal of Mechanics*, vol. 20, no. 2, pp. 123-132.

Wu, T. Y. (2013): Dynamic nonlinear analysis of shell structures using a vector form intrinsic finite element. *Engineering Structures*, vol. 56, pp. 2028-2040.

Xia, X.; Liang, Q. (2016): A GPU-accelerated smoothed particle hydrodynamics (SPH) model for the shallow water equations. *Environmental Modelling & Software*, vol. 75, pp. 28-43.

Xu, R.; Li, D. X.; Jiang, J. P.; Liu, W. (2015): Adaptive fuzzy vibration control of smart structure with vfife modeling. *Journal of Mechanics*, vol. 31, no. 6, pp. 671-682.

Yang, Y.; Cheng, J. J. R.; Zhang, T. (2016): Vector form intrinsic finite element method for planar multibody systems with multiple clearance joints. *Nonlinear Dynamics*, vol. 86, no. 1, pp. 421-440.

Yang, C.; Shen, Y.; Luo, Y. (2014): An efficient numerical shape analysis for light weight membrane structures. *Journal of Zhejiang University-SCIENCE A*, vol. 15, no. 4, pp. 255-271.

Yu, Y.; Luo, Y. (2009a): Finite particle method for kinematically indeterminate bar assemblies. *Journal of Zhejiang University-SCIENCE A*, vol. 10, no. 5, pp. 669-676.

Yu, Y.; Luo, Y. Z. (2009b): Motion analysis of deployable structures based on the rod hinge element by the finite particle method. *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering*, vol. 223, no. 7, pp. 955-964.

Yu, Y.; Luo, Y. Z. (2013): Impact analysis of structures based on finite particle method. *Engineering Mechanics*, vol. 30, pp. 66-72, 77.

Yu, Y.; Paulino, G. H.; Luo, Y. (2010): Finite particle method for progressive failure simulation of truss structures. *Journal of Structural Engineering*, vol. 137, no. 10, pp. 1168-1181.

Yu, Y.; Zhu, X. (2016): Nonlinear dynamic collapse analysis of semi-rigid steel frames based on the finite particle method. *Engineering Structures*, vol. 118, pp. 383-393.

Zhang, P.; Yang, C.; Luo, Y. (2017): Elastic-plastic analysis of 3D solids using the finite particle method. *Engineering Mechanics*, vol. 34, no. 4, pp. 5.