INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN ENGINEERING
Int. J. Numer. Meth. Engng 2015; **103**:859–893
Published online 21 July 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/nme.4842

Mapping Cohesive Fracture and Fragmentation Simulations to Graphics Processor Units

A. Alhadeff[1], W. Celes[1] and G. H. Paulino[2,*,†]

[1]*Tecgraf/PUC-Rio Computer Science Department, Pontifical Catholic University of Rio de Janeiro, Marquês de São Vicente 225, Rio de Janeiro, Brazil*
[2]*Department of Civil and Environmental Engineering, University of Illinois at Urbana-Champaign, 205 North Mathews Avenue, Urbana, IL, USA*

SUMMARY

A graphics processor units(GPU)-based computational framework is presented to deal with dynamic failure events simulated by means of cohesive zone elements. The work is divided into two parts. In the first part, we deal with pre-processing of the information and verify the effectiveness of dynamic insertion of cohesive elements in large meshes in parallel. To this effect, we employ a novel and simplified topological data structure specialized for meshes with triangles, designed to run efficiently and minimize memory occupancy on the GPU. In the second part, we present a parallel explicit dynamics code that implements an extrinsic cohesive zone formulation where the elements are inserted 'on-the-fly', when needed and where needed. The main challenge for implementing a GPU-based computational framework using an extrinsic cohesive zone formulation resides on being able to dynamically adapt the mesh, in a consistent way, by inserting cohesive elements on fractured facets. In order to handle that, we extend the conventional data structure used in the finite element method (based on element incidence) and store, for each element, references to the adjacent elements. This additional information suffices to consistently insert cohesive elements by duplicating nodes when needed. Currently, our data structure is specialized for triangular meshes, but an extension to tetrahedral meshes is feasible. The data structure is effective when used in conjunction with algorithms to traverse nodes and elements. Results from parallel simulations show an increase in performance when adopting strategies such as distributing different jobs among threads for the same element and launching many threads per element. To avoid concurrency on accessing shared entities, we employ graph coloring. In a pre-processing phase, each node of the dual graph (bulk elements of the mesh as graph nodes) is assigned a color different from the colors assigned to adjacent nodes. In that fashion, elements of the same color can be processed in parallel without concurrency. All the procedures needed for the insertion of cohesive elements along fracture facets and for computing nodal properties are performed by threads assigned to triangles, invoking one kernel per color. Computations on existing cohesive elements are also performed based on adjacent bulk elements. Experiments show that GPU speedup increases with the number of nodes and bulk elements. Copyright © 2015 John Wiley & Sons, Ltd.

Received 18 July 2012; Revised 30 July 2014; Accepted 13 October 2014

KEY WORDS: fragmentation simulation; many-core; CUDA; finite element method; cohesive elements

1. INTRODUCTION

Fracture, branching, and fragmentation simulations of large-scale finite element meshes have a broad range of engineering applications. To achieve realistic and more accurate results, there is a need to employ highly discretized models, thus requiring a large amount of computational resources. In order to accelerate finite element analysis, current parallel environments are based on distributed

*Correspondence to: Glaucio H. Paulino, School of Civil and Environmental Engineering, Georgia Institute of Technology, 790 Atlantic Drive, Atlanta, GA 30332-0355, USA.
†E-mail: paulino@gatech.edu

memory architectures. In this scenario, each processor (or small group of processors) of a computing node has private access to a region of the global system memory. Processors on different nodes communicate among themselves by sending messages over a network. Different parallel finite element systems with support for distributed mesh representation have been proposed [1, 2].

Fracture and fragmentation phenomena can be modeled using the cohesive zone model (CZM) [3–6], which can be simulated by enrichment functions [7] or inter-element techniques [6, 8]. Our application is based on the experiments by Sharon and Fineberg [9, 10] and addresses dynamic cracking instability; thus, it is not a simple issue as there are several papers in the literature addressing the problem. In the inter-element technique, cohesive elements are inserted between bulk finite elements (e.g. triangular elements in 2D case and tetrahedron elements in 3D case). Two types of cohesive elements are distinguished: instrinsic and extrinsic. In the intrinsic case, cohesive elements are inserted before the simulation starts. In the extrinsic case, cohesive elements are inserted adaptively during the course of the simulation, when needed and where needed. Because parallel fracture, microbranching, and fragmentation simulation using the extrinsic CZM [11] requires cohesive elements to be adaptively inserted during the simulation, challenges for parallelization emerge because mesh consistency must be ensured among partitions [12]. In such simulations that require a topological data structure (including our case), inserting cohesive elements between bulk elements during the course of the simulation requires a change in element connectivities and adjacency information, such as duplicating nodes and updating element neighbors. The convergence of the simulation could require unstructured meshes [13, 14], and therefore node perturbation is performed to partially address the problem [15].

In this paper, we focus on the use of many-core architectures, such as the one provided by modern graphics processor units (GPU), for accelerating fracture, microbranching, and fragmentation simulations based on the extrinsic CZM. To the best knowledge of the authors, this is the first proposal of a complete adaptive finite element analysis running on the GPU. During the last years, general purpose computing on graphics processor units has proved to be an efficient and powerful means to accelerate expensive numeric simulations and algorithms that require large amount of input data. GPUs are massively multithreaded many-core chips which are suited for excessive numeric computations with high arithmetic intensity, which is the case of finite element analysis. However, mapping the CPU version of an extrinsic cohesive fragmentation simulation to the GPU is not immediate or trivial. Several challenges emerge, such as algorithm parallelization, high-performance memory access, concurrency, and device architecture dependent factors.

We investigate and describe mapping and parallelization techniques for two-dimensional fracture, branching, and fragmentation simulation of finite element meshes on a GPU using NVIDIA's Compute Unified Device Architecture (CUDA) framework. We propose a simple but effective data structure for performing all data-parallel computations and algorithms for 2D models using triangular meshes (but an extension to tetrahedron elements is feasible). The previously established coloring method for FEM meshes is also used to minimize concurrency. Parallel techniques are presented for the numeric analysis code and for updating the FEM mesh when cohesive elements are inserted during the course of the simulation. As a result, we are able to speedup the simulation by a factor close to 30, when compared to the serial code running on a single CPU processor. In our experiments, we have employed a two-dimensional microbranching analysis, but the created framework for parallel simulation has the potential to support other separation phenomena simulations.

The main contributions of this paper are as follows:

- Establishment of a conceptual framework to map fragmentation simulations to GPUs.
- Creation of a novel special-purpose topological data structure tailored for adaptive finite element meshes on GPU platforms with fast global-memory access (hide latency). Because the GPU memory is limited, the data structure has to be simple and must consume little memory to store topological entities (e.g. adjacency information) so that it can provide adequate memory space for simulation attributes (e.g. nodal displacements, stiffness matrices of bulk elements, and cohesive tractions between cohesive element).

DOI: 10.1002/nme

- Establishment of a framework for dynamic adaptation of the mesh in the GPU, in a consistent fashion, by inserting cohesive elements on fractured facets (duplicating nodes and updating connectivity 'on the fly').
- Development of an effective nodal update scheme using gather and scatter techniques necessary for GPU parallelization. Gather techniques were employed to insert cohesive elements while scattering techniques, based on coloring, were employed to compute nodal stress, nodal mass, internal forces, and cohesive forces.
- Development of parallel techniques to map fragmentation algorithms to GPUs, such as splitting of kernels into simpler ones, distributing jobs among threads, taking advantage of memory coalescence (consecutive threads reading consecutive memory addresses), and using texture memory to increase kernel performance. For instance, the computation of the cohesive forces kernel uses one thread per element for n colors. It is decomposed as follows: (1) computation of cohesive separation kernel (one thread per cohesive element); (2) computation of cohesive traction kernel (three threads per cohesive element); and (3) computation of cohesive force kernel (one thread per element for n colors).

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 discusses the CUDA and GPU architecture, and how we can take advantage of many-core devices to improve performance. Section 4 briefly reviews simulations based on the extrinsic CZM, discussing the requirements for adaptive insertion of cohesive elements. Section 5 presents the proposed (simple) topological data structure to support mesh modification on the GPU. Section 6, conceptually the main section of this paper, discusses the parallelization of the fragmentation simulation itself, including the parallel algorithm for inserting cohesive elements. Kernel optimizations are also presented, bearing in mind memory and flow optimizations that lead to performance boosts. We also discuss the use of mesh coloring and its impact on the simulation's parallelization performance and concurrency issues. Section 7 presents the achieved results, and, finally, concluding remarks and directions of future work are drawn in Section 8.

2. RELATED WORK ON GRAPHICS PROCESSOR UNITS AND HIGH PERFORMANCE COMPUTING

Various researchers have investigated parallel simulations to improve the performance of finite element analysis, addressing the use of a distributed memory architecture (see References [1, 2] and citations within). Fracture, microbranching, and fragmentation simulation introduce new challenges because modeling of interface elements is required. Dooley *et al.* [11] have presented a parallel implementation of dynamic fracture simulation on extrinsic cohesive models using ParFUM, a parallel framework specifically developed for parallel finite element applications [16]. Radovitzky *et al.* [17] have opted for parallelizing extrinsic fracture and fragmentation simulations based on a combination of a discontinuous Galerkin formulation and CZMs. Like Dooley *et al.* [11], cohesive elements are pre-inserted throughout the mesh. Espinha *et al.* [12] developed ParTops, a topological distributed mesh representation for parallel dynamic simulation of fragmentation phenomena based on the extrinsic CZM.

Several references have also explored the use of GPU for numerical simulation and parallelization techniques other than fracture or fragmentation. Each reference provides a number of similar and different parallelization strategies to handle scientific simulations that use large-scale data, some of which are used in this paper. Boltz *et al.* [18] show that high-intensity numerical simulation can be performed efficiently on the GPU using sparse matrix conjugate gradient solver and a regular-grid multigrid solver, a technique widely used in scientific areas and real-time applications, such as mesh smoothing and parametrization, and fluid solvers and solid mechanics. Wu and Heng [19] present a deformation model on soft tissue, called hybrid condensed finite element model, based on the volumetric FEM accelerated by the GPU. Krakiwsky *et al.* [20] investigate acceleration of finite-difference time-domain method using the GPU. Tejada and Ert [21] use physics simulation and volume visualization of tetrahedral meshes on graphics hardware to build a physically-based

DOI: 10.1002/nme

deformation system based implicit solver. Since 2008, several works on GPU acceleration of scientific simulations have been published. Taylor *et al.* [22] present a fast GPU solution scheme for finite element equations used in nonlinear total Lagrangian explicit finite element formulation for surgical simulation. Göddeke *et al.* [23] explore parallelism for finite element simulations based on parallel multigrid solvers, and Anderson *et al.* [24] developed a general purpose molecular dynamics code running entirely on a GPU. Rodriguez-Navarro and Susin [25] have implemented cloth simulation on the GPU using FEM for trianglular meshes. Previous studies on FEM in GPUs also focused on solving large sparse linear systems [25–27] using CUDA. They have explored the strategy of mesh coloring for minimizing conflicts, thus avoiding excessive use of CUDA atomic operations, which usually degrade performance. A GPU approach for geometric multigrid solvers on finite elements for unstructured grid problems was performed by Geveler *et al*. [28], in which their GPU implementation is based on cascades of sparse matrix-vector multiplication by applying strong smoothers. Komatitsch *et al.* [29] have used CUDA to speedup numerical simulation of seismic wave propagation resulting from earthquakes. Liu *et al.* [30] use the GPU and CUDA to perform a fast finite element dynamic deformation simulation. Kindratenko *et al.* [31] present challenges with building and running GPU clusters for high-performance computing environments along with discussion on their experiments with GPU programming toolkits, and their interoperability with other parallel programming APIs. Godel *et al.* [32] use cluster of GPUs through CUDA to parallelize Maxwell's equations in the time domain using a discontinuous Galerkin finite element method for spatial discretization. Their parallel implementation tends to minimize overhead and improve efficiency through assynchronous data transfer. Kakay *et al.* [33] implement their finite element micromagnetic simulation in the GPU, demonstrating a high speed performance compared to CPU multi-core implementation. Ren *et al.* [34] model and analyze power performance of parallel 3D finite element mesh refinement on CUDA and Message Passing Interface (MPI) architecture using multi-core CPU and GPU cluster, also proposing parallelization techniques for both. In recent works on FEM, Markall *et al.* [35] use finite element advection-diffusion solver to demonstrate that FEM implementations on many-core (GPU) and multi-core (CPU) architectures differ if their performance potential is to be obtained. Different data structures are to be employed depending on the architecture and algorithms. They use coloring strategy to avoid concurrency and use of atomic operations. Zegard and Paulino [36] investigate feasibility of FEM and topology optimization for unstructured meshes in GPUs and discuss challenges in parallel implementation. This list of papers is incomplete and by no means exhaustive. The field of GPUs for scientific computing is quite vibrant, and new contributions are continuously being reported.

3. MANY-CORE DEVICES

Over the past decades, the number of parallel applications grew drastically as input data increased. Consequently, general purpose multicore CPUs became widely used to handle such amount of data. GPUs are massively parallel computers that work well on massive computation and problems. However, CPU and GPU parallelization are quite different.

Central processinng units and GPUs use multicore and many-core characteristics, respectively. The multi-core approach (CPU) seeks to maintain the execution speed of sequential programs while moving into multiple cores. In contrast, the many-core (GPU) trajectory focuses more on execution throughput of parallel applications [37]. Graphics chips can more easily achieve higher memory bandwidth than CPU chips, depending on how memory is accessed. Each multiprocessor have access to the GPU's global memory (dynamic random-access memory), which differs from CPU's motherboard dynamic random-access memory in computing in that they are the frame buffer memory that are used for graphics, such as video images and texture information. However, for general purpose computing, they work as off-chip, very-high-bandwidth memory with more latency than typical system memory. In CPUs, each core is independent and can execute several instructions for various processes (multiple instructions/multiple data). Unlike GPUs, all cores can access the same memory space. Cores are organized into warps, each one of which is assigned a warp number. All cores in warp 0 execute the same set of instructions, all cores in warp 1 execute the same instructions, and so on. In GPUs, each core in a given warp executes the same set of instructions

DOI: 10.1002/nme

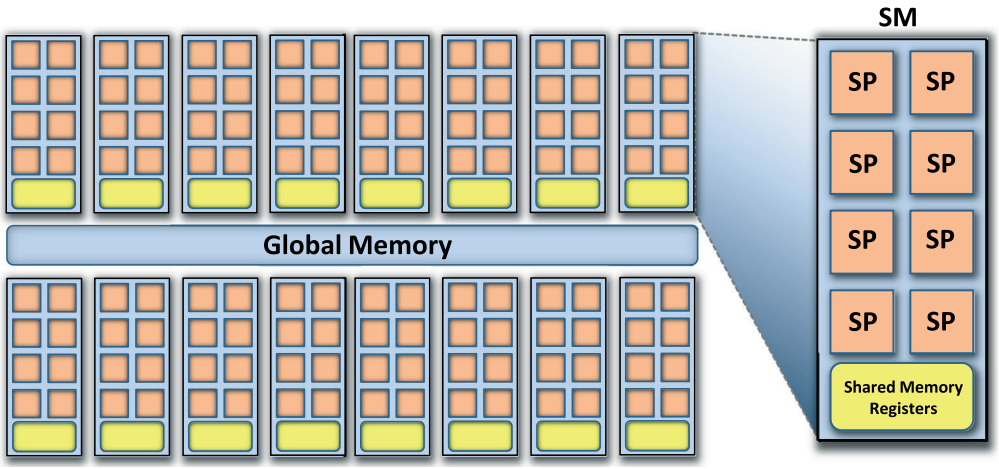


Figure 1. Diagram of a G80 architecture with 16 set of multiprocessors (SMs) and 128 streaming processors (SPs) based on the figures presented in [38].

each on different data (single instruction/multiple data). All cores have access to global memory (off-chip), which has a higher bandwidth than CPU global memory but is slower to access than on a CPU system (i.e. higher latency). Shared memory (on-chip) is extremely fast to access, but only the cores within the multiprocessor can access it. Thus, applications can be optimized by ensuring that this memory is used properly. Because each core in a given warp is executing the same set of instructions, the cores are not independent, and they communicate via their shared memory. Several aspects not encountered in CPU programming are issues that arise in GPU programming that can fatally reduce the application's performance. Such aspects include memory access (both off-chip and on-chip), branch divergence, and how to fully occupy the GPU with jobs.

3.1. GPU Architecture

We first present a few definitions that will be necessary to describe the GPU architecture. The term device means GPU, while host means CPU. Threads are light-weight processes that are managed independently by a system scheduler and executed in parallel. CUDA arranges a group of threads to make up a thread block in which they can cooperate and synchronize among themselves. Subsequently, a grid is made up of a group of blocks. A kernel is a function executed in parallel on the device (GPU) called from the host (CPU) side, while a warp is a group of threads executing synchronously within a block.

When programming in a CUDA-capable GPU, one must keep in mind its architecture and parallelism properties for they have an important role and impact on the performance of a GPU simulation. The architecture of a modern GPU is organized into a set of multiprocessors, each of which contains a number of streaming processors, as shown in Figure 1. The device memory space is organized as follows. Global memory is an off-chip memory with slow access that can be accessed by all threads. Texture access is cached, as well as the constant memory, which is also read-only and can be accessed by all threads. Shared memory is an on-chip memory space that can be accessed by all threads in a block. Threads within a thread block can use shared memory to cooperate among themselves, and this is a good alternative for optimizing a program. Finally, each thread has its own memory space known as local memory which resides on global memory. Figure 2 illustrates the CUDA memory hierarchy.

3.2. Optimization

Next, we highlight some CUDA programming issues. In a kernel execution, each thread must write on a different memory space as they are being executed concurrently to avoid writing conflicts. All threads within a warp execute the same instruction (single instruction/multiple data architecture), so

DOI: 10.1002/nme

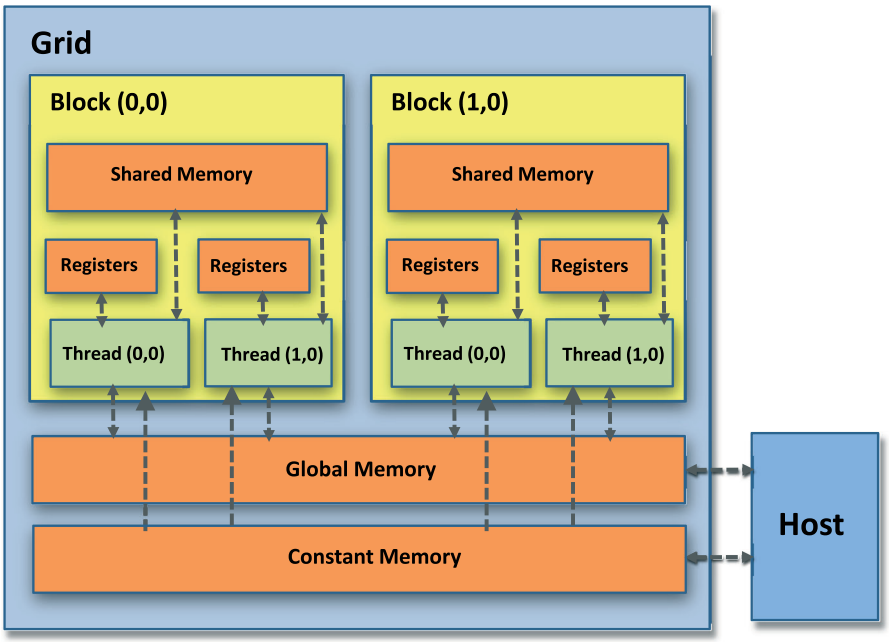


Figure 2. Compute Unified Device Architecture memory hierarchy based on the figures presented in [38].

it is suggested that there should be no conditionals and loops that lead to thread divergency within the warp. When multiple global memory accesses are coalesced into a single memory transaction by the device (i.e. proper memory access alignment and contiguity), we achieve a coalesced reading. When seeking a performance optimization in a kernel execution, it is best to minimize global memory accesses because they are slow. When access to global memory is mandatory, coalesce reading helps increase the simulation performance. To coalesce, each half warp must access contiguous 4, 8, or 16-byte words lying in the same 64 or 128-byte segment (for computing capabilities 1.0 or 1.1, respectively), which sometimes is difficult to achieve in actual simulation. Also, it is important to avoid bank conflicts when using shared memory. Shared memory is divided into banks, and if multiple threads in the same half-warp access the same bank, access must be serialized. To avoid bank conflicts, all threads of a half-warp must access different banks or all of them must read the identical address. Finally, in order to reach an optimal kernel performance, one has to maximize thread occupancy, defined as the ratio of the number of resident warps to the maximum number of resident warps, depending on the GPU architecture [38]. The ways occupancy can be maximized include minimizing the number of registers per thread and shared memory.

In this paper, we used used the following strategies to optimize our numerical simulations:

- Use of coloring and atomic operations to avoid concurrency.
- Minimizing warp divergence and avoiding different conditionals and loops by splitting one kernel into two or more .
- Reorganizing data structure pattern to properly coalesce global memory.
- Using shared memory instead of reading continuously from global memory.
- Avoiding bank conflicts in shared memory by adding padding when access is performed.
- Maximizing occupancy by minimizing register number and shared memory amount.
- Distributing jobs among threads (e.g. launching three threads per element, where each thread in the group does a different job than the other two).

Please refer to [37–39] for additional background on CUDA and many-core devices.

DOI: 10.1002/nme

4. FRAGMENTATION SIMULATION

We propose a simple but effective finite element mesh representation for fracture, microbranching, and fragmentation simulations. For a 2D simulation, we provide support for unstructured meshes of either linear (T3) or quadratic (T6) triangular elements. Our data structure was designed for unstructured meshes. Figure 3 shows a typical representation for a T6 finite element mesh. The finite element types include nodes, facets, bulk elements, and cohesive elements, which are explicitly represented as an independent element [40]. Our data structure implicitly represents facets corresponding to interfaces between two adjacent bulk elements. The intrinsic cohesive model assumes that all cohesive elements are embedded in the mesh before the simulation begins [41]. This leads to an unchanged mesh connectivity during the whole simulation process but introduces an artificial reduction of stiffness. We adopt an extrinsic cohesive model, which assumes that separation between bulk elements only occurs when the interfacial traction reaches a finite strength [6, 41]. Challenges emerge when using an extrinsic model because it requires an adaptive insertion of cohesive elements and topological changes of finite element mesh during the simulation process.

During simulation, internal, external, and cohesive forces at the nodes generate stresses along element interfaces, which may lead to fracture and fragmentation evolution. New nodes and cohesive elements are created whenever a facet fractures. Node attributes such as displacement, position, velocity, and acceleration are also updated from the internal, external, and cohesive forces. In order to obtain a precise and stable simulation, one must properly adjust parameters such as material properties and adopt small time steps, suitable for the explicit time integration scheme, together with a highly discretized model. Table I shows the algorithm for the fragmentation simulation.

4.1. Pre-processing and updating

Given an initial triangular decomposition of the domain, during the pre-processing phase, the stiffness matrix is calculated for each bulk element. We consider the stiffness matrix to remain constant

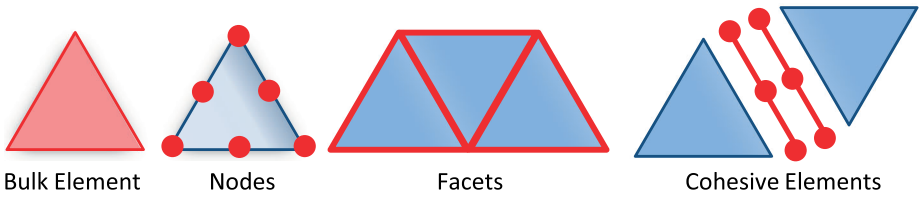


Figure 3. T6 mesh attributes belonging to the simulation.

Table I. Fragmentation algorithm.

```
Compute stiffness matrix
Update nodal mass
current step ← 0
while current step <= maximum step do
    Update displacements
    if current step == check step then
        Compute stresses
        if stresses > stress threshold then
            Insert cohesive elements
            Update nodal masses
        end if
    end if
    Compute internal forces
    Compute cohesive forces
    Update velocities and accelerations
    Update boundary conditions
    current step + = 1
end while
```


DOI: 10.1002/nme

during the whole simulation. Each lumped mass matrix is initialized before the simulation. The lumped mass matrix contains mass values relative to each bulk element node. Therefore, nodal masses are updated from the lumped mass matrix by going through the incident elements to each node. The lumped mass matrix has to be updated every time the mesh changes. This occurs when cohesive element insertion results from a fractured facet between two bulk elements.

Accelerations are computed from the cohesive and internal forces and nodal masses, which are then used to update the nodal velocities according to the following equations:

$$\begin{aligned} \mathbf{a}_{i+1} &= \frac{\mathbf{R}_{coh_i} - \mathbf{R}_{int_i}}{m_i} \\ \mathbf{v}_{i+1} &= \mathbf{v}_i + \frac{1}{2}(\mathbf{a}_i + \mathbf{a}_{i+1})\Delta t \end{aligned} \tag{1}$$

$$\mathbf{u}_{i+1} = \mathbf{u}_i + \mathbf{v}_i \Delta t + \frac{1}{2}\mathbf{a}_i \Delta t^2 \tag{2}$$

4.2. Stresses

The stresses computation is the most costly step of the simulation. After a certain number of steps (10 in this work), we compute the stress and strain at each bulk element node from their Gauss point evaluations using an extrapolation method. This whole procedure almost dominates the simulation time with excessive arithmetic operations and could be considered the bottleneck of the simulation loop if executed for all steps. To compute the stresses and strains at Gauss points of each bulk element (for each of the three Gauss points considered in this work) in 2D, we first obtain the shape functions and its derivatives, compute the Jacobian matrix and its inverse, and compute the strains and displacements relation matrix. Using the material properties of the element, the constitutive matrix is computed, followed by the stresses and strains at the Gauss points. In a two-dimensional T6 mesh case, this is a 3×4 matrix, as shown in the next equation.

$$\boldsymbol{\sigma}_{G_{element}} = \begin{pmatrix} \sigma_{1,1} & \sigma_{1,2} & \sigma_{1,3} & \sigma_{1,4} \\ \sigma_{2,1} & \sigma_{2,2} & \sigma_{2,3} & \sigma_{2,4} \\ \sigma_{3,1} & \sigma_{3,2} & \sigma_{3,3} & \sigma_{3,4} \end{pmatrix}_{3\times 4} \tag{3}$$

By means of standard extrapolation, the stress and strain matrices are obtained using the previously computed stress and strain matrices at the Gauss points and the element shape functions (N). Thus,

$$\begin{pmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{yx} & \sigma_{yy} \end{pmatrix}_{\text{node i}} = \begin{pmatrix} N_{1,1} \\ N_{1,2} \\ N_{1,3} \end{pmatrix}^T_{\text{node i}} \begin{pmatrix} \sigma_{1,1} & \sigma_{1,2} & \sigma_{1,3} & \sigma_{1,4} \\ \sigma_{2,1} & \sigma_{2,2} & \sigma_{2,3} & \sigma_{2,4} \\ \sigma_{3,1} & \sigma_{3,2} & \sigma_{3,3} & \sigma_{3,4} \end{pmatrix}_{3\times 4} \tag{4}$$

The principal stresses and their directions are calculated with respect to each nodal location. We determine if the principal stresses at each facet between two bulk elements exceed a limit for each of the nodes composing the element. Average stresses are computed to check for cohesive element insertion. We indicate that a facet is fractured if the stress exceeds a given threshold [6].

4.3. Insertion of cohesive elements

Insertion of cohesive elements imposes topological changes in the mesh [41]. After inserting the new cohesive element, each facet node is checked for duplication. Figure 4 illustrates a CPU algorithm for duplicating nodes on a triangular mesh. In 2D, the facet mid-side node must be duplicated in a T6 mesh. However, this assertion does not apply to the corner nodes, which must be checked by going through incident elements to which they belong. For each fractured facet, we verify if each of its corner nodes needs duplication. From a node, we traverse all its incident elements starting

DOI: 10.1002/nme

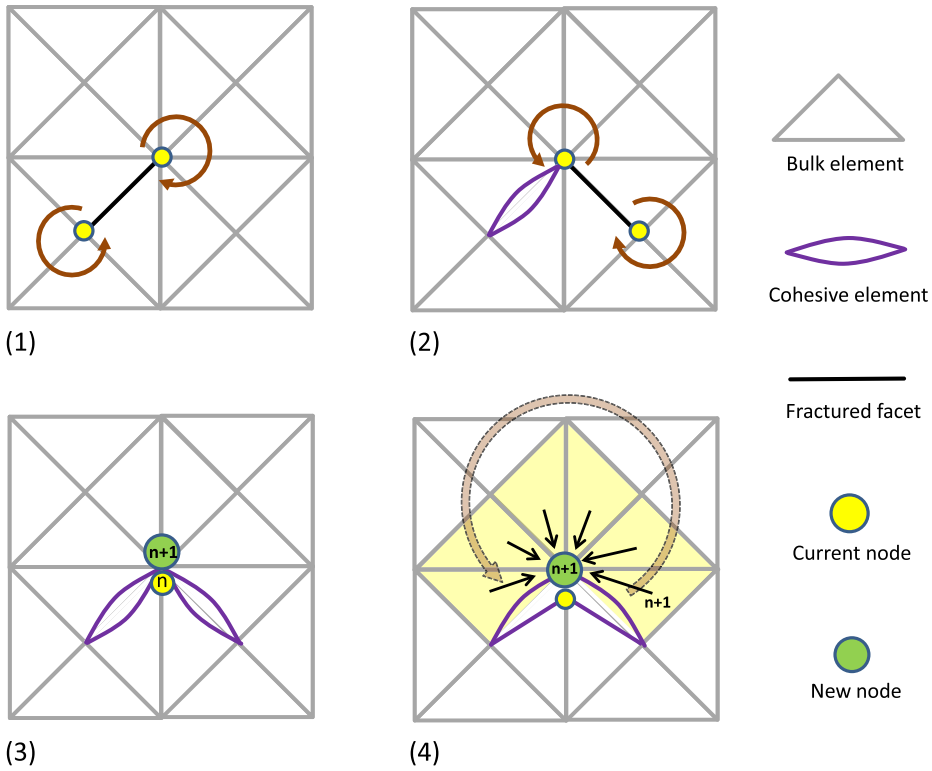


Figure 4. Cohesive element insertion algorithm on a T3 mesh. (1) Mesh with initial facets that need to be fractured. Elements belonging to each node are traversed, and cohesive element is inserted, but no node is duplicated. (2, 3) The other fractured facet is checked for node duplication; the cohesive element is inserted, and the node is marked as needing duplication. (4) Node is duplicated by traversing through the elements and updating the node index of the node belonging to them.

with one of the two adjacent elements the facet belongs to. If we reach the other adjacent element to the facet, then the node is not duplicated. However, if not reached, the node must be duplicated. The global node counter is incremented, and the new node index is retrieved from it. Once again, we must traverse the adjacent elements to update incidence with the new node index. Finally, the facet mid-side node is updated with the node index also retrieved from the node incremented global counter.

If there were new cohesive elements added to the mesh, the topological changes indicate that some nodal masses also changed because bulk elements loose adjacency relationship. Therefore, the nodal mass must be updated again like on the pre-processing phase. We then initialize the nodal internal, external, and cohesive forces for future computations.

4.4. Internal and cohesive forces

The nodal internal force computation is also computationally expensive, because it must be performed every step and requires a large number of arithmetic operations. The internal force vector results from a product of the stiffness matrix and the element displacement vector containing displacements for its six nodes in a T6 mesh, as shown in Equation (5). This means, we are multiplying a 12×12 matrix with a 12×1 vector, and it greatly reduces the performance of our parallel implementation because of its numerous global memory accesses.

$$R_{int_{12,12}} = \begin{pmatrix} k_{1,1} & k_{1,2} & \cdots & k_{1,12} \\ k_{2,1} & k_{2,2} & \cdots & k_{2,12} \\ \vdots & \vdots & \ddots & \vdots \\ k_{12,1} & k_{12,2} & \cdots & k_{12,12} \end{pmatrix} \begin{pmatrix} u_{1x} \\ u_{1y} \\ \vdots \\ u_{6y} \end{pmatrix} \tag{5}$$

DOI: 10.1002/nme

The cohesive forces are then calculated by traversing through all cohesive elements and calculating their contributions to each node attached to them. The element vector of the deformed configuration is obtained, followed by the cohesive separations in the local coordinate system. Then, the separations and tractions at each Gauss point are calculated, together with the cohesive shape functions. Finally, the nodal cohesive force vector is obtained from cohesive tractions and shape functions. Together with the internal force and stresses, calculating the cohesive forces is one of the most costly computation steps within the simulation loop.

5. DATA STRUCTURE

In order to implement efficient operations on mesh entities used in the simulation, a simple topological data structure is employed to represent the mesh. Because the GPU memory is limited, the data structure must not be complex so as to provide space for other simulation attributes that indeed require much global memory space. The proposed data structure is used for T3 or T6 mesh. Although adding support for tetrahedral elements (Tet4 or Tet10) is straightforward, inserting cohesive elements in 3D requires a graph structure to traverse a node's incident elements, which is much more expensive for the GPU. In this paper, our data structure will focus on 2D elements only.

We maintain two tables that describe the mesh. A table of nodes stores the node world-space position (x and y coordinates). A second table is used to represent the elements and adjacency relationships. The elements can be of two types: bulk elements or cohesive elements. Because the number of bulk elements remains unchanged during the entire simulation, we store the cohesive elements immediately after the bulk elements. For each bulk element, we store its nodal incidence; three node indices are used in a T3 mesh and six are used in a T6 mesh. For T6 meshes, the corner node indices are followed by the mid-side indices. The right hand rule (counter clockwise) is used to define the order of nodal incidence. Another three values represent adjacent element indices that are opposite to each of the corner nodes (three values for T3 and T6 meshes). For cohesive elements, we store six node indices for a T6 mesh and four node indices for a T3 mesh. In a T6 mesh, the first three node indices represent the three nodes of the corresponding facet of its adjacent bulk element with the smaller *id*, following the right hand rule. The next three indices belong to the adjacent facet of the second adjacent bulk element (with the greater *id*). For a T3 mesh, two indices are stored for each of the elements. Another two values are used to represent indices of both bulk elements that are adjacent to each cohesive element's facets (in both T3 and T6 meshes). Following the pattern, the first bulk element is opposite to the the first facet (and to the first three node indices) of the cohesive element. If a cohesive element is attached to a bulk element's facet, we update the opposite element of that facet to the cohesive element *id* on the element table. Nodes or elements are represented by their indices in the corresponding table. The last index in the table is not used for cohesive elements. Both node and element tables are stored in the global memory and are updated along with the adaptive numerical simulation. Figure 5 shows an example of tables used in the data structure.

With our data structure, we are able to perform the following computational patterns (Figure 6): (1) a node can be updated based on its own information; (2) a bulk element can be updated based on its own information; (3) a bulk element can be updated based on the information of its nodes; and (4) a node can be updated based on the information of its incident bulk elements. For the last pattern, it is necessary to store a bulk element identifier for each node so as to start traversing its incident elements. In our implementation, we do not store it because we use this computational pattern by sweeping the bulk elements first. This only works for global computation (e.g. applied on all nodes of the bulk elements [Section 5.2]). For each node of a bulk element, we traverse its incident elements. Not storing a bulk element *id* for each node also allows us to save GPU memory.

A finite element analysis maintains a set of simulation attributes attached to nodes and elements. We maintain such attributes in global, constant, or texture memory depending on their memory size and dynamics during the simulation. In global memory, we store the attributes that change throughout the entire simulation. The nodes have associated displacements, velocities, accelerations, and forces (internal and cohesive), which are updated at every timestep. Stresses and strains evaluated at the nodes are updated only in a number of timesteps. When facets are checked for possible

DOI: 10.1002/nme

Node Table

Id	x	y	z
0	0.0	0.0	0.0
1	1.0	0.0	0.0
2	1.0	1.0	0.0
3	-1.0	1.0	0.0
4	0.0	2.0	0.0
5	-1.0	1.0	0.0
6	1.0	1.0	0.0
⋮		⋮	

Element Table

Id	v_0	v_1	v_2	v_3	v_4	v_5	o_0	o_1	o_2
0	0	1	2	8	7	9	-1	1	-1
1	0	2	3	9	11	10	3	-1	0
2	5	6	4	12	13	14	-1	-1	3
⋮					⋮				
3	2	11	3	5	12	6	1	2	-1
⋮					⋮				

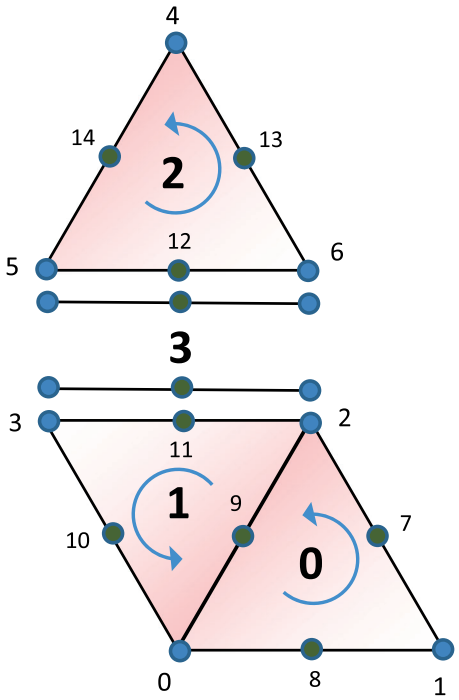


Figure 5. A special-purpose simplified data structure with mesh parameters of a T6 mesh.

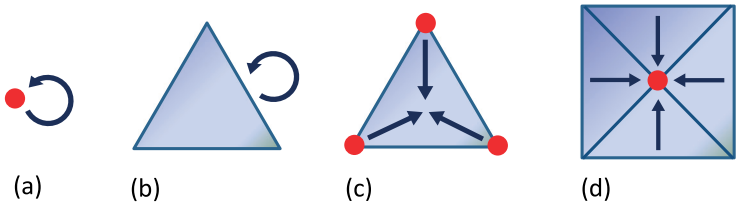


Figure 6. Computational patterns. (a) a node can be updated based on its own information; (b) a bulk element can be updated based on its own information; (c) a bulk element can be updated based on information of its nodes; and (c) a node can be updated based on information of its incident bulk elements.

fractures, the fractured facets in the element attributes store which facets have been fractured. Nodal mass and number of adjacent bulk elements are updated whenever the topology of the mesh changes. Finally, cohesive attributes such as tractions and separations are updated every step for each cohesive element. We store other node and element attributes that remain unchanged during the entire simulation, but require too much memory space, in textures. We cannot store these attributes in constant memory because of its limited memory space. Each element's stiffness and lumped mass matrix and each node's boundary conditions are stored in texture memory. Other attributes that are common to all nodes and elements, such as elastic and fracture material properties, are stored in constant memory. These attributes are stored in one table common to all elements and nodes and occupy little memory space. Because all threads in a warp access the same memory space in constant memory, there will be no bank conflicts. Figure 7 shows element and node attributes that were stored in the GPU global, texture, and constant memories.

Paulino *et al.* [41] present a topology-based framework for supporting fragmentation simulations in extrinsic CZMs for CPUs. Their topological data structure, TopS, contains all information

DOI: 10.1002/nme

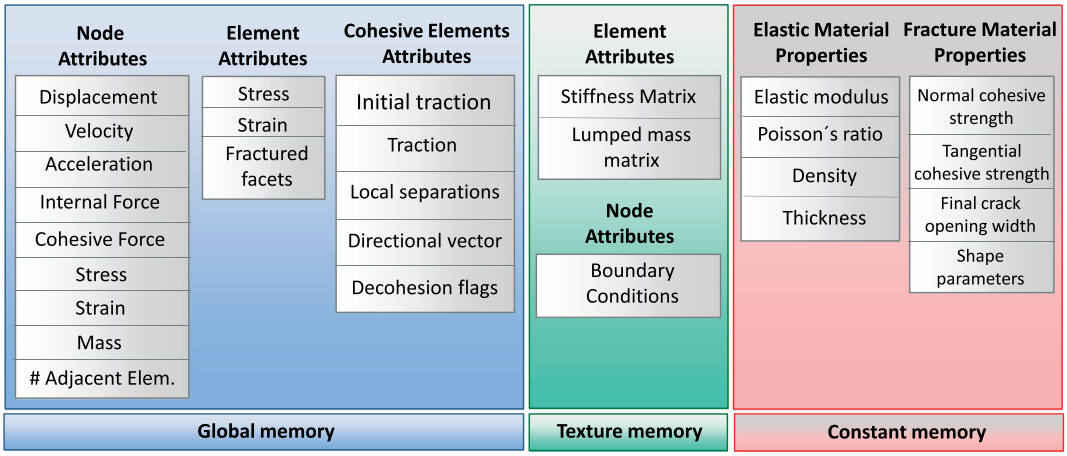


Figure 7. Simulation parameters data structure diagram of FEM model. Global memory is used for attributes that change throughout the simulation. Texture memory is used for attributes that are constant during the entire simulation but occupy too much memory space. Constant memory is used for attributes that are constant during the entire simulation but are common to all elements and node, therefore requiring few memory space.

necessary to retrieve element adjacency relationships needed for the simulation and is able to perform, for example, the previously described computational patterns (Figure 6). While their data structure is designed for general element types, our specialized data structure is simpler and has been especially tailored to operate efficiently on GPUs (i.e. data is stored in 2D arrays versus structures). To represent the mesh on the GPU, it is sufficient to store node positions, node indices for elements, and adjacent elements, which is much less information than what is stored in TopS [41]. Both the element and node tables are designed to make the minimal global memory access as possible, as well as occupying minimum device memory. Saving device memory is a key issue because memory from a single GPU is extremely limited. Currently, we lack information of storing an element *id* for each node (if we want to traverse its incident elements by sweeping the nodes first). However, while inserting cohesive elements and duplicating nodes, this information is redundant in this step because we sweep each element first, followed by each of its nodes' incident elements. We also need to perform more arithmetic operations when traversing them because we lack information of node order in each element. However, the lack of this additional information as well as additional arithmetic operations is compensated by making less global memory accesses, which is a fine tradeoff for GPU programming. It has also the advantage of saving GPU memory to run larger models.

5.1. Retrieving adjacency relationship

The node and element tables are enough to perform the previously stated algorithms and do not require too much memory usage. One key adjacency relationship for the insertion of cohesive elements is the set of adjacent elements to a given node. With the described data structure, from an element, for each of its incident node, we can easily traverse the set of adjacent elements. Given the first node, we search the other node that precedes it in the order of incidence, and then access the corresponding opposite element and find the order of incidence of the node that had the previous element as its opposite. From both the element and node order, we obtain the next node in the incidence of the element and access its opposite element. From there, we repeat the procedure until we reach the element adjacent to the first one. Figure 8 illustrates the traversal algorithm from a given node if no cohesive element is reached within the path. Blue arrows indicate which node to access in order to obtain the correct opposite element. Dashed lines indicate the node we must access to obtain the next element required to continue traversing. Figure 9 illustrates the traversal algorithm from a given node until it reaches a cohesive element. Cohesive elements can also be obtained by traversing around a node because they are also stored in the element table and can be accessed by obtaining the opposite element for a given node.

DOI: 10.1002/nme

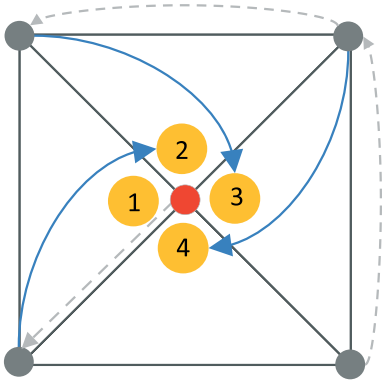


Figure 8. Traversal algorithm from a given node using the proposed data structure. The illustrated path does not contain cohesive elements.

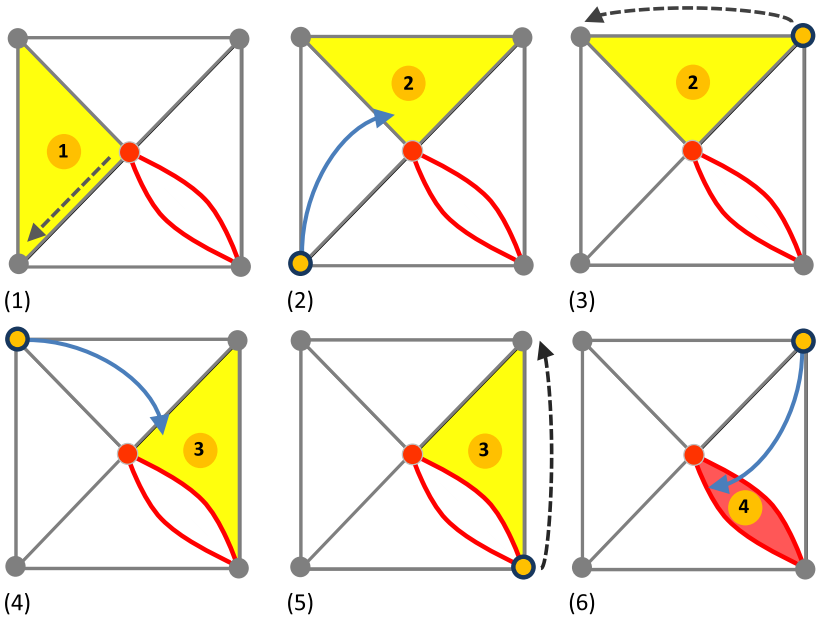


Figure 9. Traversal algorithm from a given node using the proposed data structure, with cohesive elements along the path. (1) From a bulk element, the algorithm starts by accessing a node whose opposite element is incident to the traversed node (central node). (2) The opposite element to that node is obtained (3) followed by the next node. (4) The third bulk element is accessed, (5) followed by its respective node. (6) A cohesive element opposite to a node (or adjacent to a bulk element's facet) can also be reached because it is explicitly represented in the element table (Figure 5).

5.2. Node update

The set of adjacent elements of a given node is necessary to update element incidence when a node is duplicated because of the insertion of a new cohesive element. During the simulation step, we can also identify computations where such topological relationship can be used. As an example, we can consider the mass associated to a node, which depends on contributions of all adjacent elements. However, as we discuss further in our parallel simulation, for almost all cases, accumulating contributions of adjacent elements to nodes is more efficiently handled by traversing all the elements in the model. Such cases include calculating internal and cohesive forces and stress and strain on nodes. For each element, we accumulate its contribution to all incident nodes. In the end, the contributions of all corresponding adjacent elements will be accumulated for each node. In a serial code, this algorithm is straightforward and very efficient. In a parallel environment, writing conflicts arise,

DOI: 10.1002/nme

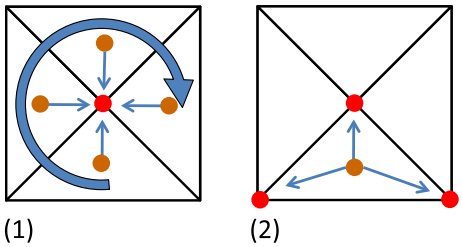


Figure 10. Node update algorithms: (1) incident elements traversal (or gather), and (2) element sweep (or scatter).

and one needs to ensure consistency, as we shall discuss. Figure 10 illustrates both strategies to compute nodal information from its adjacent elements.

The first strategy, known as Gather, traverses the node's incident elements, accumulating their information on the node. The second strategy, known as Scatter, accumulate each element's contribution to all incident nodes. We tested both algorithms on the GPU by accumulating each element's mass on their incident nodes. In our experiments, the Scatter algorithm turned to be more efficient as we increased the number of elements. While this algorithm implements a simple kernel function with few global memory accesses and no divergence, the Gather algorithm implements a more complex kernel requiring greater number of registers and adding divergence (not all nodes have the same number of incident elements). Although divergence can be minimized by sorting elements according to their incidence, this algorithm still tends to perform much more computation than the other. The Scatter approach is well suitable for problems that require many computations per elements, but when duplicating nodes and inserting cohesive elements on fractured facets, the Gather strategy is essential as the basis of the algorithm, as discussed in Section 4.4. To avoid writing conflicts in the Scatter algorithm, we adopted the commonly used mesh coloring representation, as we shall discuss later. While testing both algorithms by accumulating element mass on nodes, it was observed that a higher number of colors or unbalanced number of elements in each color lowers the efficiency of the Scatter algorithm. As we increase the number of elements, Gather's efficiency tends to decrease in relation to Scatter. For these reasons, we adopted the Scatter algorithm for all cases except when inserting cohesive elements and duplicating nodes, in which we used the Gather algorithm.

6. PARALLEL IMPLEMENTATION

The main challenge for implementing a many-core parallel fragmentation simulation, based on the extrinsic CZM, is to ensure topological consistency on mesh adaptation (insertion of new cohesive elements). However, even the mechanics code, at first straightforwardly parallelized, based on explicit integration, also imposes challenges. Memory access and usage can be a bottleneck when using the slow accessible global memory space. Concurrency is also an issue to have in mind because writing conflicts can eventually occur when updating the same memory space for different threads running concurrently. In order to maximize the performance and benefit from GPU parallelism, it is important to keep in mind programming techniques discussed in Section 4, or else the attempted GPU speedup will be negligible. Although the parallel algorithms discussed in the next dicussions refer to a T3 or T6 mesh, they can be extended to 3D meshes using a modified version of the previously discussed data structure.

In this section, the notation `<<<X>>>` denotes a kernel call, a function executed by the device (GPU) that is called by the host (CPU). The parameter X is the number of threads to be launched. As an example, if each node of the mesh is executed in parallel, then X is the number of nodes of the mesh.

6.1. *Coloring model*

In this discussion, we consider the implementation of T6 meshes. The first parallel procedure to be discussed is updating the node attributes. In our case, we are focused on updating each nodal

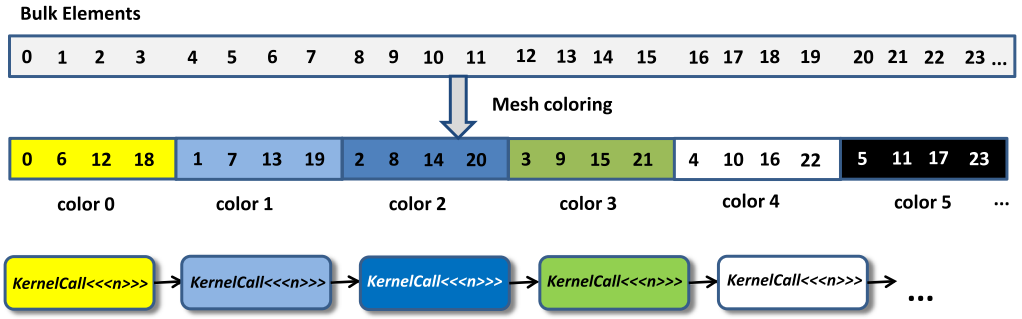
DOI: 10.1002/nme

mass with the lumped mass matrix from each adjacent bulk element. The lumped mass matrix is computed in a pre-processing phase together with the stiffness matrix. We could launch one thread per element and accumulate the element mass on its respective nodes retrieved from the incidence table. However, threads would write on the same memory space because different elements share the same node.

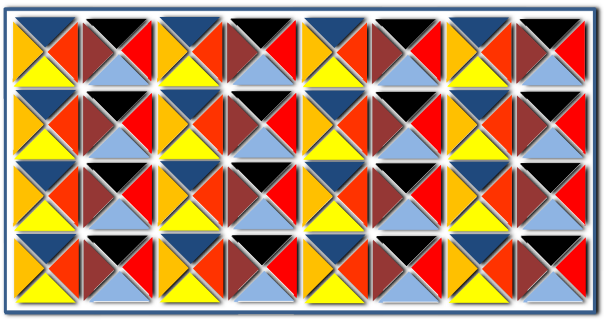
To avoid the race condition, we adopt the commonly used mesh coloring representation. The idea is that no element of the same color shares a node. Applying this technique when accumulating the nodal masses from the bulk elements means that we will launch a kernel for each color with a thread per element in that color group. With this strategy, different threads will not update the same node because there would not be elements with shared nodes being processed in parallel. Because bulk elements are neither removed nor inserted during the entire simulation (only cohesive elements and nodes are inserted), mesh coloring can be pre-processed. In graph theory, a node degree (also called valency) is the number of incident edges to that node. In our model, each element represents a graph node, and adjacent elements are connected by a graph edge. The minimum number of color groups is equal to the maximum node degree on the entire mesh. However, determining the minimal color number of a graph is known as an non-deterministic polynomial time (NP)-complete problem, although there are many heuristics for finding a reasonable solution. In our case, we will be interested in finding a reasonable and balanced solution, or else we will be wasting additional kernel computations with few threads per color containing few elements, while having other color

Table II. Kernel subroutine call algorithm using mesh coloring.

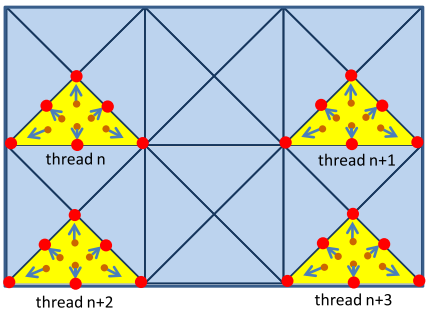
```
for c = 1 → numColors do
    numThreads ← numElements(c)
    KernelCall <<< numThreads >>> (c)
end for
```



(1)



(2)



(3)

Figure 11. (1) Bulk elements are re-arranged in color groups (preferable balanced) and the same kernel per color group is called to avoid writing conflicts. (2) Example of a colored T6 structured mesh (3) and using the colored mesh and scatter strategy to update nodal masses of the group of elements in the current color in parallel.

DOI: 10.1002/nme

with many more elements. We order the graph nodes (elements) in decreasing order of degree to obtain the closest optimal solution. Table II shows the procedure we use to perform a conceptual execution unit on the elements in parallel: we launch the same kernel multiple times, one for each group color. Figure 11 illustrates the colored mesh and its use in kernel calls and updating the nodal masses using the scatter strategy. We use the Welsh Powell algorithm [42], a greedy algorithm[‡] to color unstructured meshes. In our experiments, we use structured 'union-jack' meshes. For these meshes, we apply a coloring algorithm that takes advantage of their pattern so as to obtain the optimal color number.

6.2. *Pre-processing and update*

A pseudo-code of the parallel simulation is shown on Table III. In the pre-processing phase, also executed on the GPU, we need to compute the stiffness matrix and the lumped mass matrices

Table III. Parallel Fracture Algorithm.

```
1: ComputeMassMatrix <<< numElem >>>
2: ComputeStiffnessMatrix <<< numElem >>>
3: for c = 1 → numColors do
4:    numGroupElem ← numElem(c)
5:    UpdateNodalMass <<< numGroupElem >>>
6: end for
7: current step ← 0
8: while currentstep <= maximumstep do
9:    UpdateDisplacements <<< numNodes >>>
10:   if current step == check step then
11:      ComputeStressesAtGaussPoints <<< numElem >>>
12:      for c = 1 → numColors do
13:         numGroupElem ← numElem(c)
14:         ComputeNodeStresses <<< 12 * numGroupElem >>>
15:      end for
16:      CheckFracturedFacets <<< numNodes >>>
17:      FilterFracturedFacetElements <<< numElem >>>
18:      numFracElem ← CompactFracturedFacetElements
19:      if Current Fractured Facets > 0 then
20:         for c = 1 → numColors do
21:            numGroupElem ← numFracElem(c)
22:            InsertCohesiveElements <<< numGroupElem >>>
23:         end for
24:         for c = 1 → numColors do
25:            numGroupElem ← numElem(c)
26:            UpdateNodalMass <<< numGroupElem >>>
27:         end for
28:      end if
29:   end if
30:   for c = 1 → numColors do
31:      numGroupElem ← numElem(c)
32:      ComputeInternalForces <<< 12 * numGroupElem >>>
33:   end for
34:   ComputeCohesiveSeparations <<< numCohElem >>>
35:   ComputeCohesiveTractions <<< 3 * numCohElem >>>
36:   numElemCoh ← CompactBulkElementsWithCohesiveElements
37:   for c = 1 → numColors do
38:      numGroupElem ← numElemCoh(c)
39:      ComputeCohesiveForces <<< numGroupElem >>>
40:   end for
41:   UpdateVelocitiesandAccelerations <<< numNodes >>>
42:   UpdateBoundaryConditions <<< numNodes >>>
43:   current step + = 1
44: end while
```

[‡]Refer to: http://ghpaulino.com/educational_GreedyGraphCol.html

DOI: 10.1002/nme

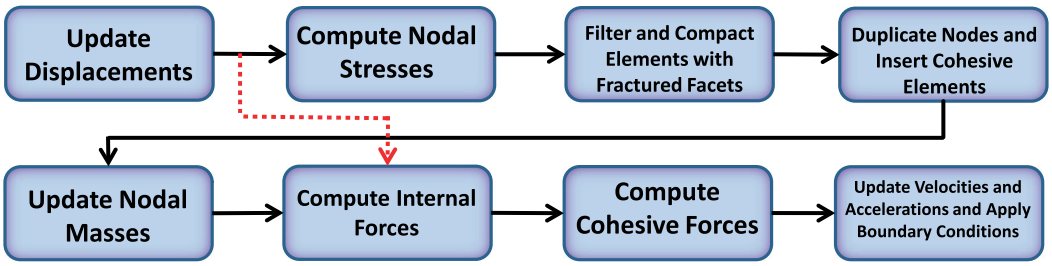


Figure 12. Fracture and fragmentation simulation loop.

associated to each element and then update the nodal masses. Building the stiffness matrix requires one thread per element but with no color subdivision scheme because we write directly in per-element memory space. The same kernel computes each element's lumped mass matrix. The last kernel in the pre-processing phase updates the nodal masses with the lumped mass matrix by using the previously discussed parallel algorithm, invoking a kernel per color group. We use constant memory for storing material attributes that are constant during the entire simulation. Cache hits when fetching these attributes during stress and other force computations will help to increase performance because threads in the same warp access the same value at the same time.

Figure 12 depicts the steps in a simulation loop. The first kernel in the simulation loop updates the nodes' displacements, launching one thread per node. Each thread fetches the velocity and acceleration of its corresponding node from global memory and updates the result back in global memory following the Equation (2). This is a simple kernel that uses few global memory accesses, and every thread in a warp follows the same path because there are no conditionals or loops.

6.3. Stresses

Before checking fractured facets, the next procedure is responsible for computing the stresses and strains on the nodes by first calculating them at the Gauss points for each element, multiplying its respective matrix with the element shape function as shown in Equation (4) in Section 5, and writing them back on the elements' nodes. Each node stress is then checked for cohesive strength over a threshold value so it can later indicate if a facet is fractured or not. To implement this whole procedure in a single thread, we would need to launch one thread per element using the color model to avoid concurrency. This single kernel would have too many loops and global memory accesses that cause a low performance. Also, the number of registers would exceed the established limit, forcing the compiler to put local variables on local memory residing on global memory. Another issue worth highlighting is that this complex kernel would be executed several times because of the color model. We have then opted for an alternative strategy to reduce effort and increase performance, dividing this complex kernel into three simpler ones. In the first kernel, we compute the elements' stresses and strains at the Gauss points by launching one thread per element and with no color model. The second kernel calculates the stresses and strains matrix for each node, launching one thread per element but this time using the color model because each element accumulates results on its nodes. Notice that this kernel's effort is reduced because it only performs read-write on global memory. The third kernel checks if each node's principal stresses exceed the cohesive strength limit by launching one thread per node. The kernel dividing technique is useful as it distributes efforts among simpler kernels by reducing global memory accesses and reducing loops, and it will be adopted on other kernels too. Looking at Equation (4), we can observe that the second kernel performs several global memory accesses because it accumulates element stresses and strains on its nodes by fetching from the element stress and strain matrix (3×4 matrix) at the Gauss points, computed on the previous kernel. An alternative strategy is to launch one thread per element node (six threads per element), and each thread is responsible for multiplying the stress and strain matrices at Gauss points with the respective nodal shape functions and writing the result in its respective node. We opt to launch 12 threads per element where each thread would fetch two columns from the four-column Gauss point element matrix line and write the result on part of the 2×2 nodal stress and strain matrix. This

DOI: 10.1002/nme

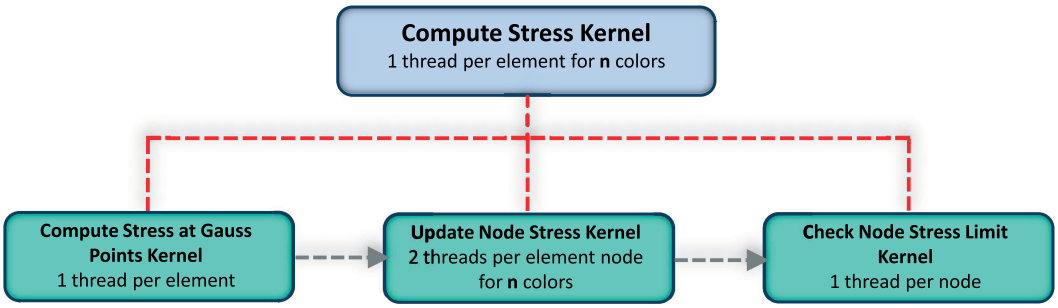


Figure 13. Splitting the kernel that computes stress and strain into simpler kernels.

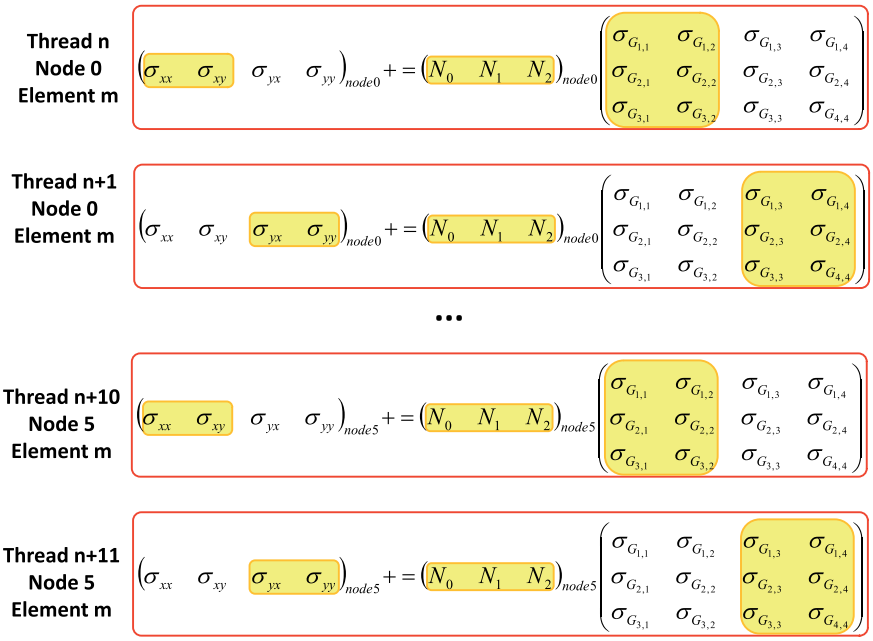


Figure 14. To accumulate the stresses and strains on the nodes, we launch 12 threads per element, where each thread will accumulate part of the stress and strain matrices by fetching from the element shape functions and from the stress and strain at the Gauss points.

strategy reduces global memory access per thread, reducing the kernel effort. Figure 13 illustrates the stress kernel division, and Figure 14 illustrates the second kernel procedure.

6.4. *Insertion of cohesive elements*

Once fractured facets are identified, new cohesive elements must be inserted in the mesh. When inserting cohesive elements, launching one thread for each element can result in idle kernels because there are few elements that contain fractured facets. In order to solve this matter, an additional kernel is used before inserting cohesive elements. This additional kernel filters only the elements that contain fractured facets by launching one thread per element and checking its three facets for possible fractures as discussed in Section 5. However, a fractured facet always belongs to two elements that are adjacent to each other, and we cannot filter both elements for the same facet otherwise the nodes will be duplicated twice. Therefore, we chose the element that has the smaller (or greater) identifier number. In our implementation, we also maintain a list of bulk elements that are adjacent to existing cohesive elements. This list is useful when later computing cohesive forces, otherwise idle kernels will be included in this simulation step as well.

DOI: 10.1002/nme

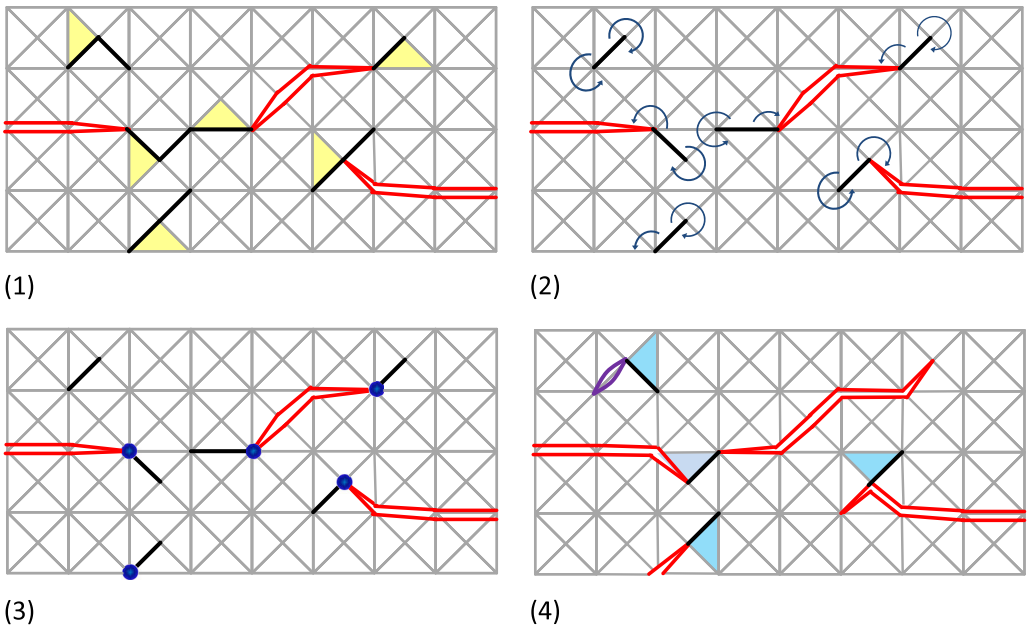


Figure 15. Cohesive elements insertion on a T6 mesh. (1) Mesh with initial cracks and facets that fractured facets. Coloring is used to avoid duplicating nodes of elements that share nodes in parallel. (2) From each facet node belonging to the element in the current color group, the algorithm traverses through its incident elements. (3) Nodes that need duplication. (4) T6 mesh with final node duplications and new cracks and cohesive elements. The fractured facets from the next color group are checked for cohesive elements insertion.

From the list of elements containing fractured facets, we now check for node duplication and insert the cohesive elements. We use mesh coloring on the filtered elements' list and launch one thread per element. Figure 15 illustrates the parallel cohesive element insertion process. During one element computation, we go through its fractured facets and check its nodes for duplication. The same traversal algorithm presented in Section 5 is used to check if the node has to be duplicated. If so, we need to update the global nodal counter and retrieve the new node index. However, because the node counter resides in one global memory address, many threads updating the same counter cause a writing conflict. To solve this matter, we use CUDA's atomic operations to perform a read-modify-write operation (in this case, a global variable increment), without the interference of other threads. The function `atomicAdd()` computes the sum on the word located in the global address and returns the previous stored word. Therefore, it returns the new node index needed to update the elements' incidence table. Node attributes are then copied to the newly appended node. The traversal algorithm is used to go through the node's adjacent elements until it reaches the cohesive element while updating their nodes with the new index value. We also need to update the opposite indices in the element table. Table IV presents the parallel cohesive element insertion algorithm.

After duplicating nodes and inserting the cohesive elements, nodal mass is changed as the sets of adjacent elements are also changed. We update the nodal mass using the previously discussed parallel algorithm. Cohesive and internal forces are then initialized as they later are calculated.

6.5. Internal Forces

Computing the internal forces is another expensive kernels and occupies a large portion of the simulation as it is executed every time step. Our first approach was launching one thread per bulk element and use the color model because the elements' nodes are updated. The stiffness matrix is multiplied by the displacement vector, resulting in the nodal internal forces. In a two-dimensional case, the stiffness matrix has dimension 12×12 and the displacement vector 12×1. With a naive

DOI: 10.1002/nme

Table IV. Parallel Node Duplication Algorithm.

```
1: e ← bulkelement
2: for each corner node n belonging to a fractured facet f of e do
3:     for each incident element of n starting with e do
4:         e ← next element
5:     end for
6:     if element adjacent to e is reached again then
7:         continue
8:     end if
9:     newNodeIndex ← atomicAdd(globalNodeCounter, 1)
10:    nodeList[newNodeIndex] = n
11:    for each incident element of n starting with e do
12:        Replace n index with newNodeIndex
13:        e ← next element
14:        if cohesive element or crack is reached then
15:            break
16:        end if
17:    end for
18:    Insert cohesive element in facet f
19: end for
```

multiplication code, we make 1728 global memory accesses. A strategy to reduce the number of global memory fetches is to load the displacement vector into shared memory once and use it for multiplying each line of the stiffness matrix. This greatly reduces the number of global accesses from 1728 to 156. The performance, however, still does not reach optimal expectations. By making each thread responsible for computing the product of one line of the stiffness matrix with the displacement vector, the number of global memory accesses is reduced to 13 (one for loading a value from the displacement vector into shared memory and 12 for fetching values from one line of the stiffness matrix), as well as the kernel's effort. Launching one thread per matrix line means we are launching 12 times the number of threads per element. Because coloring is used, the total number of blocks hardly exceeds the limit. Going further, because the stiffness matrix is constant during the entire simulation, it can be stored in a texture memory to take advantage of the texture cache and the spatial locality accessed by the warp. In order to guarantee the right memory access in each thread, we define the thread block dimension (1D) as the number of matrices per block times the number of threads per matrix (in our case, 12 threads for each matrix). To guarantee the thread block is multiple of the warp size, we use 16 matrices per block (192 threads per block) on a GeForce GTX 480 GPU (NVIDIA, 2701 San Tomas Expressway Santa Clara, CA 95050). Each thread of the block loads one value from the displacement vector into shared memory and are synchronized. Notice that each group of 12 threads will load its respective element displacement vector. One issue remains, however. At the same time thread 0 is reading address 0 (row 0, column 0), for instance, thread 1 of the same warp will be reading address 12 (row 1, column 0), thread 2 will be reading address 24 (row 2, column 0), and so forth. This means that memory is not being coalesced at all. In order to properly perform coalesced readings and achieve a higher bandwidth, consecutive threads must read consecutive memory addresses. Therefore, we transpose the matrices, and each thread of a warp will be able to read consecutive addresses. Figure 16 illustrates this strategy.

6.6. Cohesive forces and simulation outcome

Unlike the internal force kernel, computing the cohesive forces is expensive because of its numerous arithmetic operations, especially when calculating the tractions at the Gauss points. It performs few global memory access (when used registers do not exceed the limit). Launching one thread per cohesive element possibly generates writing conflicts when updating nodal cohesive forces because cohesive elements may share nodes. Therefore, one thread per bulk element would be considered. However, with many arithmetic operations, registers, and color models applied to the kernel, the previous kernel splitting technique could help increase performance. In the first kernel,

DOI: 10.1002/nme

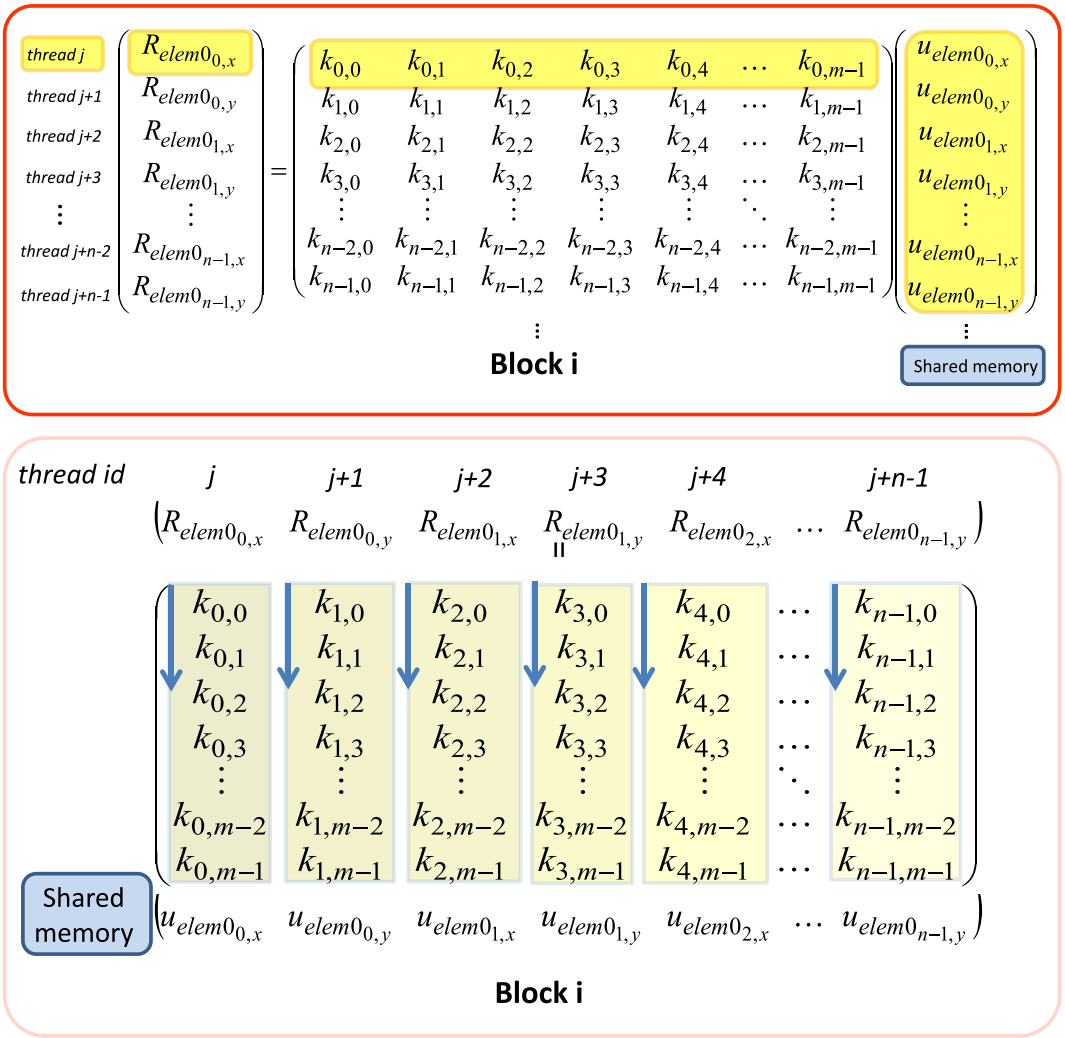


Figure 16. When computing internal forces, a thread per stiffness matrix line is launched using the color model and used to perform a dot product with the displacement vector in shared memory. In this example, the first image shows one element used per block. The second image shows the matrix transposed so memory reads can be coalesced (i.e. each consecutive thread reads consecutive memory addresses).

we calculate the cohesive separations in the local coordinate system. One thread per cohesive element is launched because we write directly on the cohesive attributes memory space. The second kernel calculates the cohesive traction by also launching one thread per cohesive element. However, this is the most expensive kernel in terms of arithmetic operations, especially when we need to calculate the cohesive tractions for each of the three Gauss points. Therefore, we adopt the previous strategy of launching more than one thread per element. In this case, we will be launching three threads per cohesive element, one for each of the three Gauss points. Each thread is responsible for calculating the tractions for its cohesive element in its respective Gauss point. Because the total number of cohesive elements in the simulation is relatively small, the number of threads will not be high. This strategy helps increase the performance of the the kernel. Finally, the third kernel consists of writing the cohesive forces on the cohesive elements' respective nodes. We then launch one thread per bulk element that contains any cohesive element (using the list previously mentioned). To avoid concurrency, the threads are separated by color group. The cohesive kernel subdivision is shown in Figure 17.

DOI: 10.1002/nme

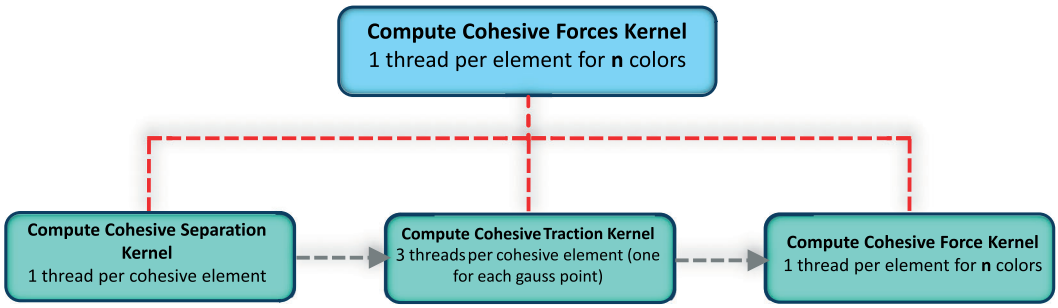


Figure 17. Splitting the kernel that computes cohesive forces into simpler kernels.

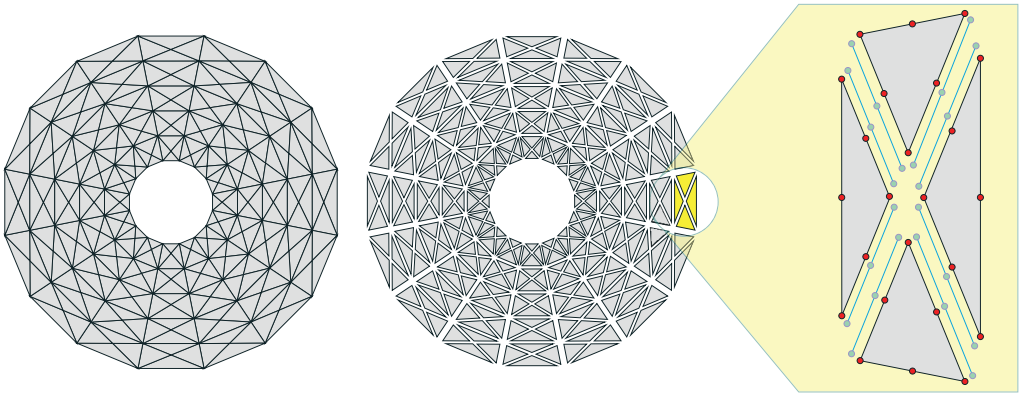


Figure 18. T6 disc mesh used to test insertion of cohesive element decoupled from analysis code.

The last two kernels of the simulation are launched with one thread per node. Updating velocities and accelerations requires only a few global memory accesses for fetching cohesive and internal forces as well as current and previous accelerations and nodal mass. They are used to write on the acceleration and velocity global memory space. Boundary conditions are then applied using a second kernel to update accelerations and velocities of boundary nodes.

6.7. Overview

Looking closely to all steps of the simulation, we can point out that there are two dominant kernels. The first is computing cohesive forces, which requires many arithmetic operations. The second is computing internal forces because it requires several global memory accesses. Splitting the kernels into simpler ones, distributing jobs among threads, and using texture memory greatly increase the kernels' performance. Non-linear simulations would need to compute the stiffness matrix at every time step instead of pre-processing it. Although it would greatly reduce the program's performance, the GPU speedup would also increase. Computing stress and strain is the most complex and expensive kernel. Although it requires a larger processing time and more numerous arithmetic operations than computing the internal and cohesive forces, it is not computed at every time step, thus not dominating the simulation time. Kernels that update displacements, velocities and accelerations, boundary conditions, and nodal masses are small kernels as they perform few and simple read-and-write operations with no warp divergence and coalesced reading. Shared memory is rarely used, working more as a cache to optimize memory access.

7. EXPERIMENTAL RESULTS

To test the performance and the correctness of our parallel code, we have run a set of computational experiments. The experiments were split in two parts: inserting cohesive elements decoupled from

DOI: 10.1002/nme

mechanics analysis and running the fracture and fragmentation simulation. The GPU simulation results are compared to CPU counterparts running on a INTEL CORE I7 (Intel Corporation, Santa Clara, CA, USA) CPU @ 2.80GHz with 12GB of memory on a 64-bit Windows 7 operating system. The GPU used device is a NVIDIA GEFORCE GTX 480 with 15 multiprocessors, each with 32 cores and a total of 480 CUDA cores, with a clock rate of 1.40 GHz and using compute capability 2.0 because we use double precision in the simulation. The total amount of GPU memory is 1.536 gigabytes.

Table V. Results for insertion of cohesive elements decoupled from analysis code.

Bulk elements	Initial nodes	Final nodes	Cohesive elements	CPU time (s)	GPU time (s)	Speedup
240,000	481,200	1,440,000	359,400	9.29	0.0407	228.3
960,000	1,922,400	5,760,000	1,438,800	36.946	0.1016	363.6
2,160,000	4,323,600	12,960,000	3,238,200	84.94	0.1935	439.0
3,840,000	7,684,800	23,040,000	5,757,600	150.04	0.3101	483.8

GPU, graphics processor unit.

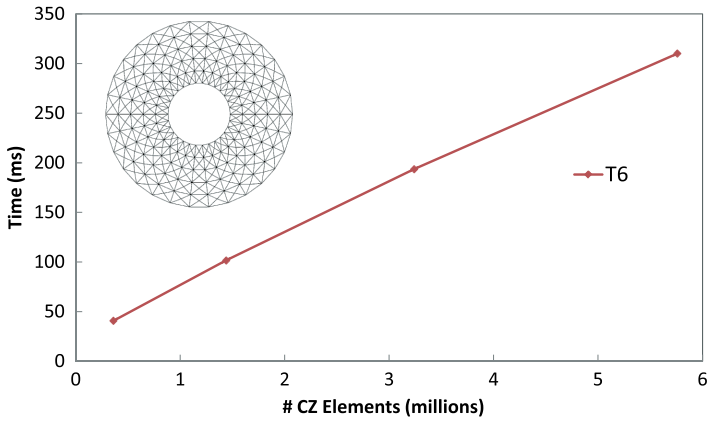


Figure 19. Time for cohesive elements insertion of a T6 mesh.

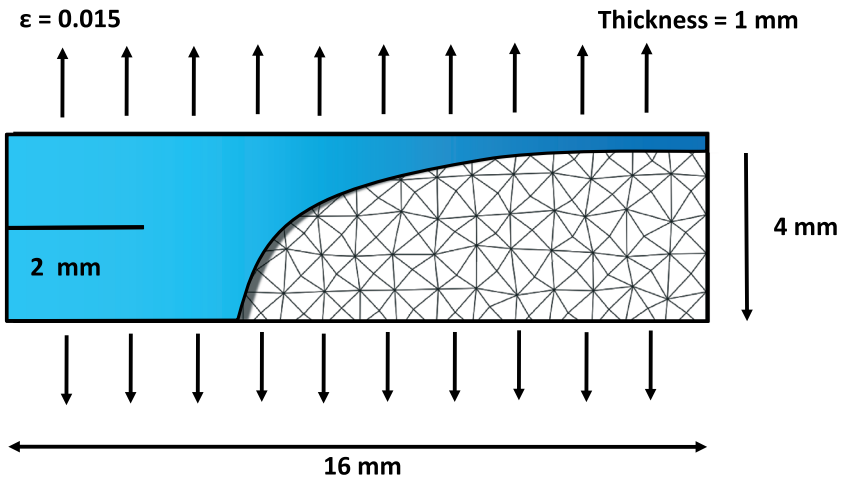


Figure 20. Two-dimensional model of a rectangular specimen with initial notch of 2 mm. Initial strain is 0.015, with node thickness of 1 mm. Model dimensions are 16mm per 4mm.

DOI: 10.1002/nme

Table VI. Simulation and mesh parameters for a T6 mesh and its refined version.

No. of bulk elements	No. of nodes	No. of new nodes	No. of cohesive elements	No. of colors
36,864	74,257	1901	979	8
147,456	295,969	5842	2976	8

Table VII. Simulation and mesh parameters and results (graphics processor unit [GPU] speedup and GPU and CPU time) for a T6 mesh and its refined version.

No. of bulk elements	Timestep	CPU time (s)	GPU time (s)	Speedup
36,864	2.0e-9	410.181	11.788	34.8
147,456	0.5e-9	6,537.839	153.809	42.5

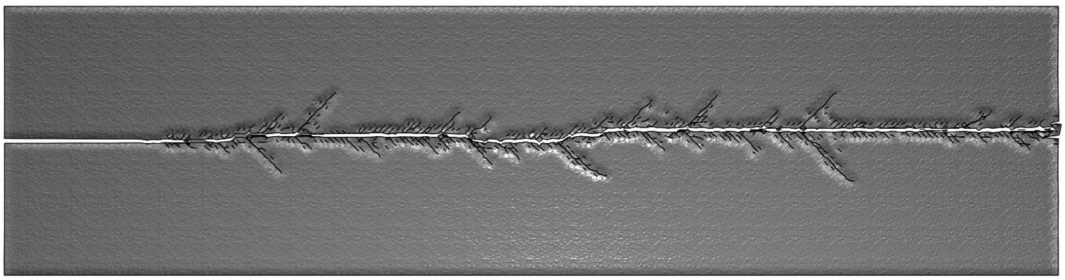


Figure 21. T6 FEM mesh with 36,864 bulk elements at the end of the fragmentation simulation.

7.1. Insertion of cohesive elements

To check the correctness of the algorithm to insert cohesive elements in parallel, we have run a computational test decoupled from any mechanics simulation (setting up experiments similar to the ones described by Pandolfi and Ortiz [43] and by Paulino *et al.* [41]). Cohesive elements were inserted, in a random order, at all the facets of the underlying meshes. The random order in which the cohesive elements are inserted results in arbitrarily complex crack patterns during the experiment. In the end, each node of the mesh is used by only one bulk element. We then have checked if the final obtained number of topological entities were the expected ones. In the experiments ran by Pandolfi and Ortiz [43] and Paulino *et al.* [41], the cohesive elements were inserted in a serial order. In our experiment, the cohesive elements are inserted in parallel. To better mimic insertion of cohesive elements in actual simulations, the facets were grouped in 20 sets, inserting 5% of cohesive elements concurrently within each group of facets, using the color model. To color the mesh, we used a greedy algorithm[§]. The number of colors achieved was 10.

We have employed a T6 disk mesh like the one in Figure 18 with different discretizations, varying the number of bulk elements from 240,000 to 3,840,000. The results are shown in Table V. Figure 19 depicts that the time to insert all cohesive elements varies linearly with the total number of inserted elements. As can be noted, the gain in performance delivered by the GPU implementation is quite significant, even though we are more interested in validating the GPU results.

When duplicating thousands to millions of nodes concurrently, as in this experiment, atomic operations can be quite slow. In order to optimize node duplications in such scenarios, we present a new algorithm that greatly speed up the kernel. The algorithm is discussed in Appendix A. However, during actual fragmentation simulations, few nodes are duplicated concurrently in a timestep, making the new strategy performance gain negligibly.

[§]Refer to: http://ghpaulino.com/educational_GreedyGraphCol.html

DOI: 10.1002/nme

7.2. *Fragmentation simulation*

The fragmentation simulation was tested using two models: a rectangular specimen and ring specimen. Serial and parallel simulation times as well as kernel times, for the parallel simulation, were measured, and simulation outputs such as number cohesive elements, number of new nodes were analyzed. During the pre-processing phase, nodal perturbation is performed on the end-nodes of a T6 element [15]. Mid-side node positions are linearly interpolated at each facet edge nodes' positions.

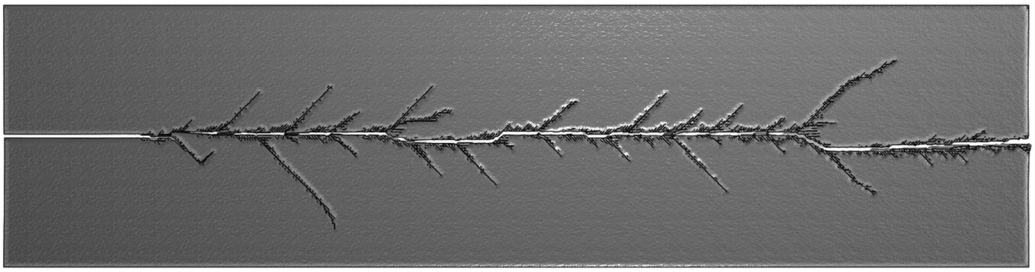


Figure 22. Refined T6 FEM mesh with 147,456 bulk elements at the end of the fragmentation simulation.

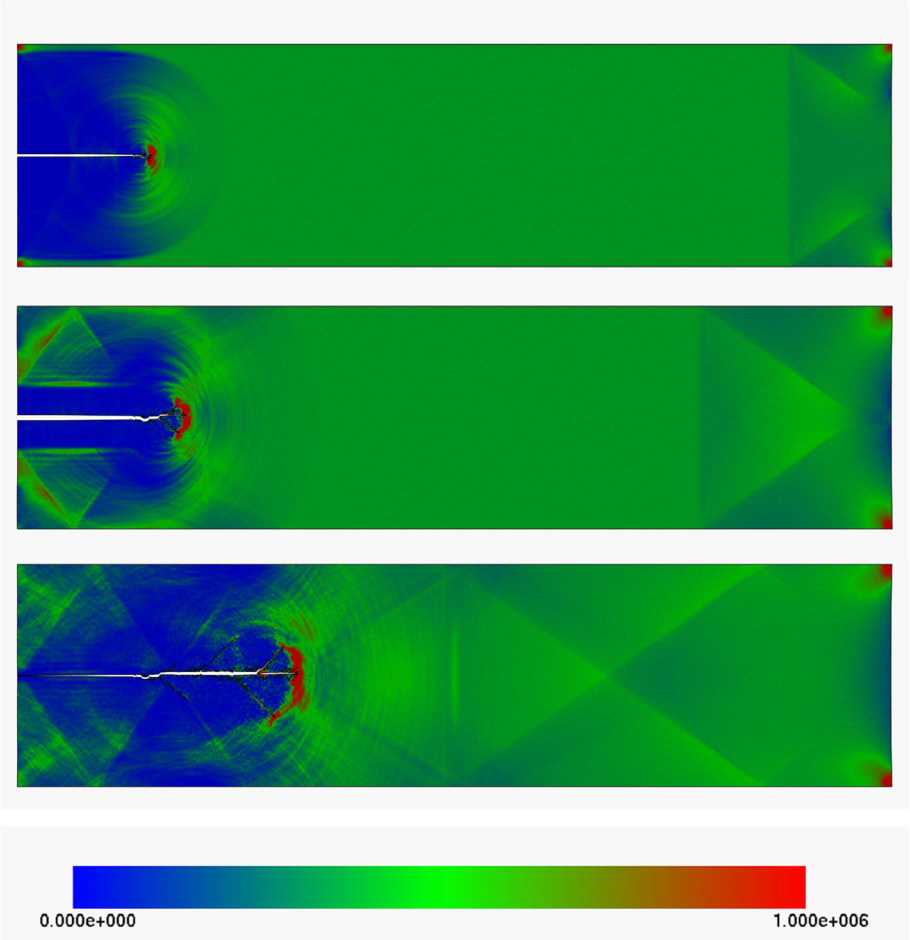


Figure 23. Strain energy evolution with crack propagation.

DOI: 10.1002/nme

7.2.1. Rectangular specimen. The first model tested was a rectangular specimen with an initial notch and refined into T6 (quadratic triangle) elements, as illustrated in Figure 20. Fracture propagation is based on mixed-mode fracture and extrinsic CZM [4, 44, 45]. Initial analysis parameters are as follows: initial strain = 0.015, elastic modulus = 3.24 GPa, Poisson coefficient = 0.35, specific mass = 1190 kg/m3, fracture energy G_I = 352 N/m, cohesive strength smax = 324 MPa, and shape parameter $\alpha = 2$.

A first version of the mesh was composed by 74,257 nodes and 36,864 bulk elements. Because of the regular mesh pattern, we employed a simple procedure to subdivide the elements into eight color groups, which is the optimal. Although we took advantage of the structured mesh to obtain an optimal number of colors, our simulation is able to handle unstructured meshes with a non-optimal number of colors (which is the case in the ring example). Total simulated time is 2 μs, in 10,000 steps of 2 ns. Figure 24 shows an extruded 2D model after the simulation and fracture propagation.

Figure 24. 3D view of fragmented 2D plate with 74,257 nodes and 36,864 bulk elements.

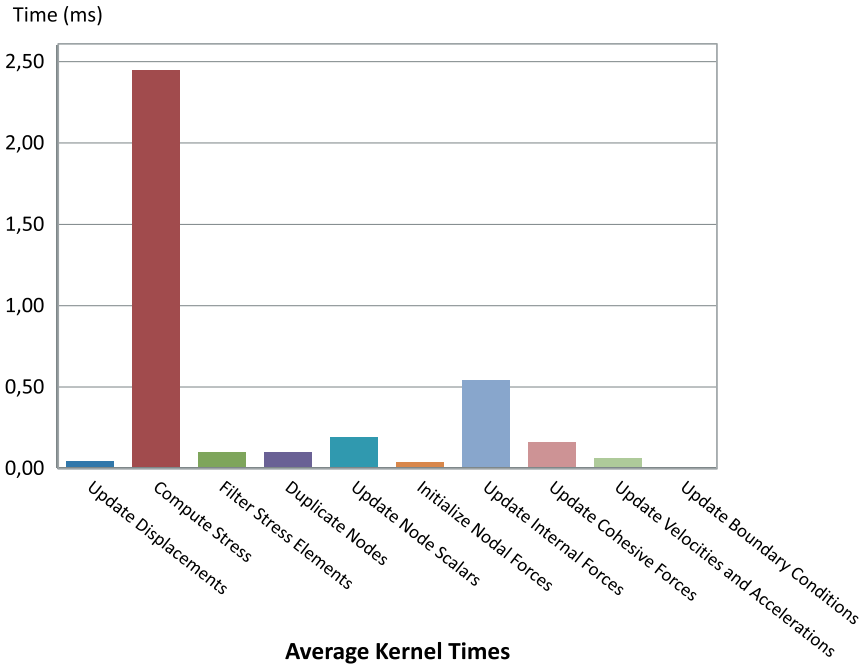


Figure 25. Average time of each kernel of the simulation for a T6 mesh with 36,864 bulk elements.

DOI: 10.1002/nme

The refined version of the same model with 295,969 nodes and 147,456 bulk elements was also tested using time steps of 0.5 ns, performing 40,000 simulation steps. In both experiments, stress computation and fractured facets are checked at every 10 simulation steps, and cohesive elements inserted 'on-the-fly', when and where needed. Tables VI and VII present results for both the mesh and its refined version. The increase in speedup with model size is expected when running the application on GPU. Figures 21 and 22 show the final plotted image of the T6 mesh and its refined version at the end of the simulation. The fracture evolved in a straight path in consequence of the initial notch of the model and the transverse strain applied on the model. Figure 23 shows the nodal strain energy wave propagation with the fracture and simulation evolution. Figure 24 shows an extruded visualization of the two-dimensional plate.

Figures 25 and 26 present results for the portion and average simulation execution times for each kernel for the first T6 mesh model. Stress computation is by far the most expensive, as shown in Figure 25. However, Figure 25 shows that the kernel responsible computing the internal forces dominates the simulation time with almost twice the time of the stress kernel because of the fact that

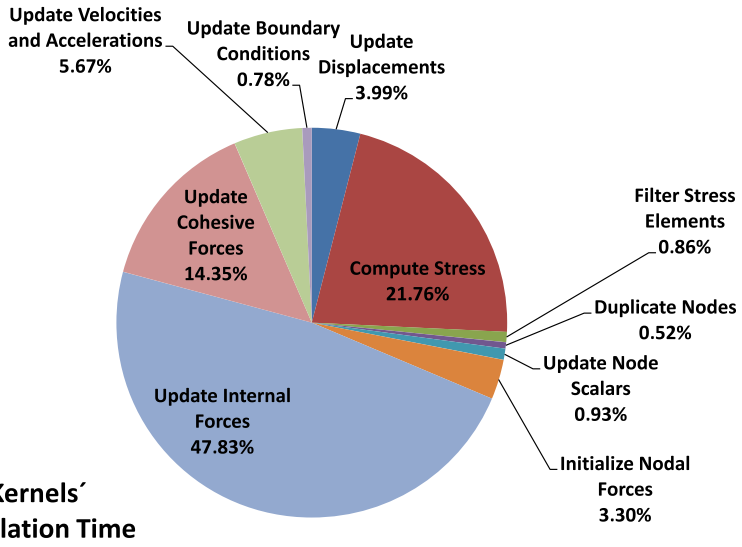


Figure 26. Total time each kernel takes in the entire simulation for a T6 mesh with 36,864 bulk elements.

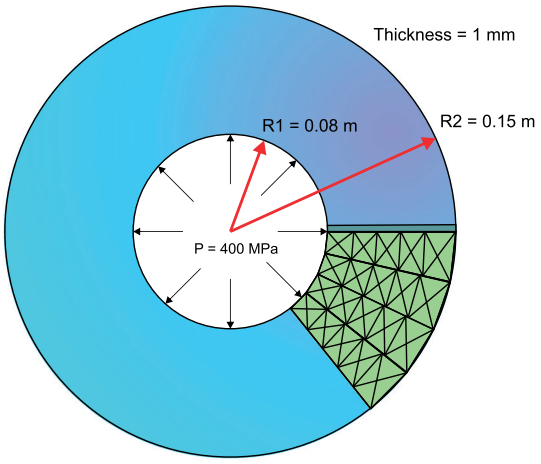


Figure 27. 2D model of a ring specimen. Initial pressure is 400 MPa, with node thickness of 1 mm. The inner radius is 0.08 m, and the outer radius is 0.15 m.

DOI: 10.1002/nme

the internal forces are computed at each time step, while stresses are computed at each ten steps. Another kernel that greatly occupies the simulation time is computing the cohesive forces because of its many numeric computations, although kernel splitting helped increase performance. The node

Table VIII. Simulation and mesh parameters for a T6 mesh and its refined version.

Mesh type	No. of bulk elements	No. of nodes	No. of new nodes	No. of cohesive elements	No. of colors
Ring 20×160	12,800	25,920	1816	955	10
Ring 30×240	28,800	58,080	4923	2506	10
Ring 40×320	51,200	103,040	9178	4686	10
Ring 96×768	294,912	591,360	59,055	29,615	10
Ring 115×787	362,020	725,614	68,842	34,245	10

Table IX. Simulation and mesh parameters and results (graphics processor unit [GPU] speedup and GPU and CPU time) for a T6 mesh and its refined version.

Meshtype	No. of bulk elements	Timestep	CPU time (s)	GPU time (s)	Speedup
Ring 20×160	12,800	3e-9	375.963	12.668	29.7
Ring 30×240	28,800	3e-9	900.095	22.632	39.8
Ring 40×320	51,200	3e-9	1601.516	36.384	44.0
Ring 96×768	294,912	3e-9	9,355.186	200.804	46.6
Ring 115×787	362,020	3e-9	11,421.568	242.802	47.0

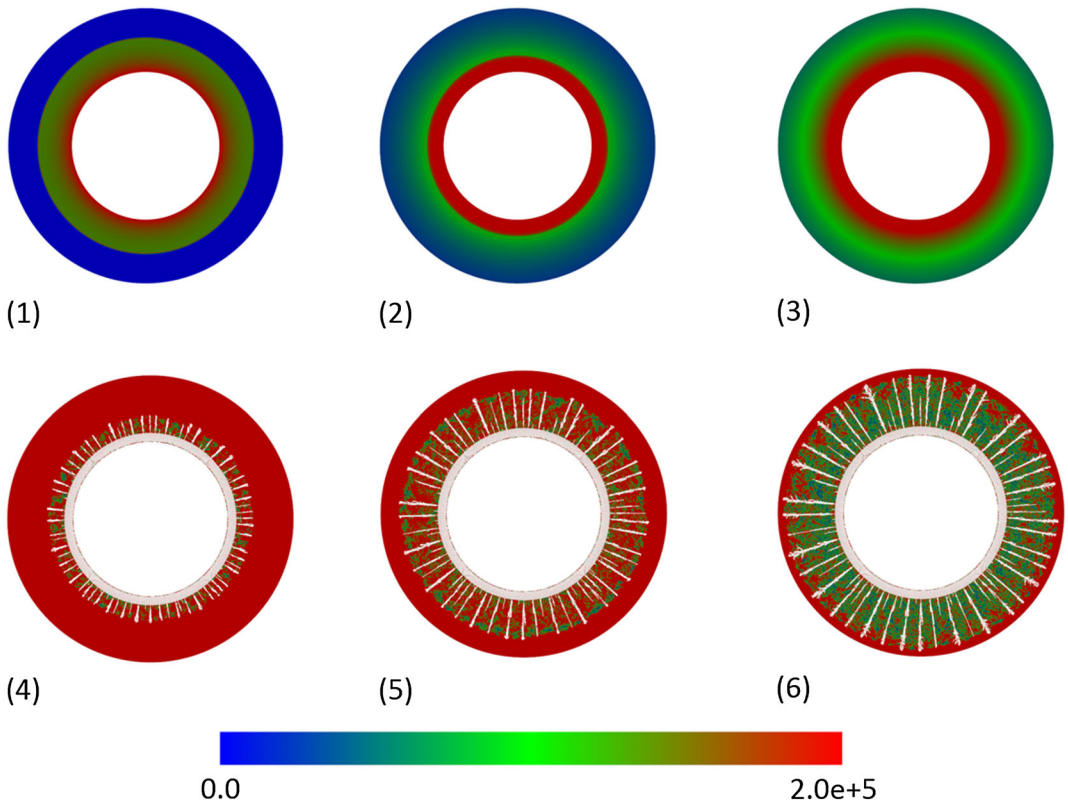


Figure 28. The figure shows a T6 FEM mesh with 362,020 bulk elements and the strain energy's evolution with the crack propagation for times (1) 5 μs, (2) 20 μs, (3) 25 μs, (4) 50 μs, (5) 60 μs, and (6) 68 μs.

DOI: 10.1002/nme

duplication kernel does not occupy a large portion of the simulation because the number of cohesive elements is relatively small. Filtering elements, updating velocities, accelerations, displacements, and nodal masses, and applying boundary conditions are small job kernels with few global memory accesses and coalesce readings that do not have high execution time.

7.2.2. Ring specimen. The second tested model was a 2D ring specimen with no initial notch, refined into T6 (quadratic triangle) elements, as illustrated in Figure 27. We performed a procedure to color the mesh using the greedy algorithm [¶]. Fracture propagation is based on mixed-mode fracture and extrinsic CZM [4, 44, 45]. Initial analysis parameters are as follows: initial pressure = 400 MPa, elastic modulus = 210 GPa, Poisson coefficient = 0.3, specific mass = 7850 kg/m3, fracture energy (GI) = 2000 N/m, and shape parameters (a) = 2.

A first version of the mesh was a 20×160 ring composed by 25,920 nodes and 12,800 bulk elements. The number of colors obtained by a greedy coloring algorithm was 10. A radial pressure was applied on the model. Total simulated time is 81 μs, in 27,000 steps of 3 ns each. The model was refined in from 25,920 nodes to 725,614 nodes. Stress computation and fractured facets are checked at every 10 simulation steps, and cohesive elements inserted as necessary. Tables VIII and IX present results for both the mesh and its refined version. Figure 28 shows the strain energy evolution throughout the simulation. As expected, the GPU speedup increases with the number of bulk elements.

8. CONCLUDING REMARKS

In this work, we propose a strategy for GPU-based parallel fracture, microbranching, and fragmentation simulations based on the extrinsic CZM. Such parallel simulations impose two main challenges. First, we need to be able to handle mesh modifications because cohesive elements are inserted adaptively along the simulation. Second, we need to efficiently perform intensive numerical computation, with numerous memory accesses, in parallel. Using a simple topological data structure, shared memory, kernel splitting, texture fetch, and minimizing global memory accesses, we could effectively map and optimize the CPU implementation of a fragmentation simulation to a GPU environment, taking advantage of CUDA benefits. Mesh coloring proved to be an effective means to avoid race conditions and simplifying algorithms that would generate complex kernels if not used.

Although our solution can be extended to three dimensions, we leave the simulation of 3D tetrahedron models on the GPU for future implementation. Fragmentation simulation could also be extended to a multi-GPU approach, where we would need to consider mesh partition subdivision and communication among them. Another future approach is the extension of the GPU implementation to adaptively perform mesh refinement and coarsening. One challenge we would have in mind is the implementation of a parallel coloring method on the mesh as the mesh structure changes at each step. Maintaining consistency of the topological changes of the mesh would also be challenging because bulk elements would also be inserted and removed from the mesh. Finally, fragmentation simulation could extend further than engineering applications. If physical accuracy is relaxed and the timestep increased, the implementation of breaking objects in computer animation could be performed using a simplified version of our model.

APPENDIX: OPTIMIZED INSERTION OF COHESIVE ELEMENTS

The following strategy is a much faster way to insert cohesive elements in the GPU. However, it is not used in our parallel implementation because it requires a large number (e.g. thousands) of cohesive elements being inserted in parallel, which is not the case in the actual simulation.

[¶]Refer to: http://ghpaulino.com/educational_GreedyGraphCol.html

DOI: 10.1002/nme

To optimize the cohesive elements insertion and node duplication, and also avoid writing on the same global memory address using the atomic operation, we tested a new strategy where each thread block will have its own node counter. Each thread within the block is responsible for updating the node counter that resides in shared memory, and only one thread in the block (not necessarily the first) will be responsible for accumulating the shared counter on the global counter residing in global memory. The advantages of this strategy are that writing in shared memory is much faster, and fewer number of threads will be updating the same global memory address simultaneously as well as few threads updating the same node counter in shared memory. Retrieving the new node index is performed by using the atomic functions' return values. For each thread, when adding one to the block counter in shared memory, we retrieve the number of block's duplicated nodes until that moment, representing the node index offset within the block. Threads are then synchronized. One thread in each block adds its block counter to the global node counter, and the atomic function returns the number of nodes immediately before the sum. This result represents the current number of nodes for all of the other blocks. Threads are synchronized, and adding the atomic intrinsic result from global counter with the shared counter index will give the current new node index for this thread. This strategy is best taken use for when many nodes are duplicated (such as test case reported in Section 7.2). In actual simulation, however, cohesive element insertion is checked at a number of steps in which few nodes are duplicated, so the increase in performance is unremarkable. Figure A.1 illustrates the algorithm for retrieving the node index inside the current block. Using the atomic function to accumulate the shared memory counter when accumulating the number of nodes inside the current block gives us the node offset inside the block. Figure A.2 illustrates retrieving the node index offset from all other blocks. Using the atomic function to add the current number of nodes in the global counter with the current number of nodes inside the block (stored in the shared node counter) gives us the current node offset for this block. Adding it with the current node offset inside the block gives the new node index. Table A.1 presents the algorithm.

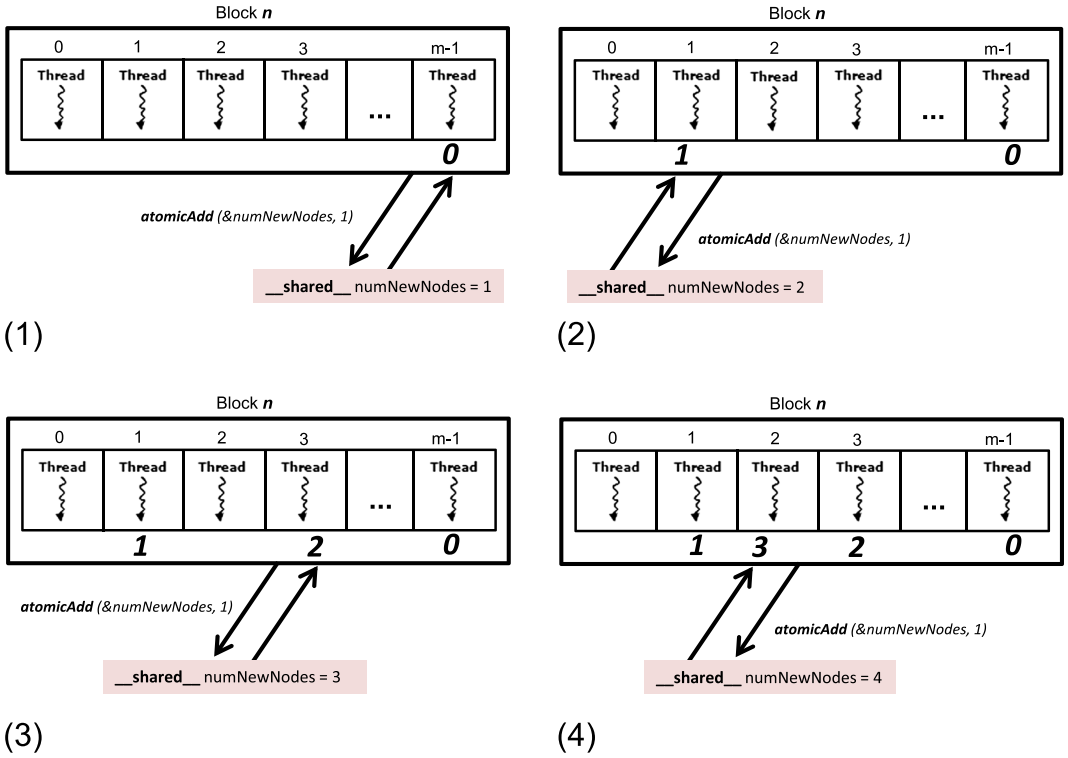


Figure A.1. Getting part of the new node index for each thread node counter offset inside the block. This value is added to the current node counters from each block.

DOI: 10.1002/nme

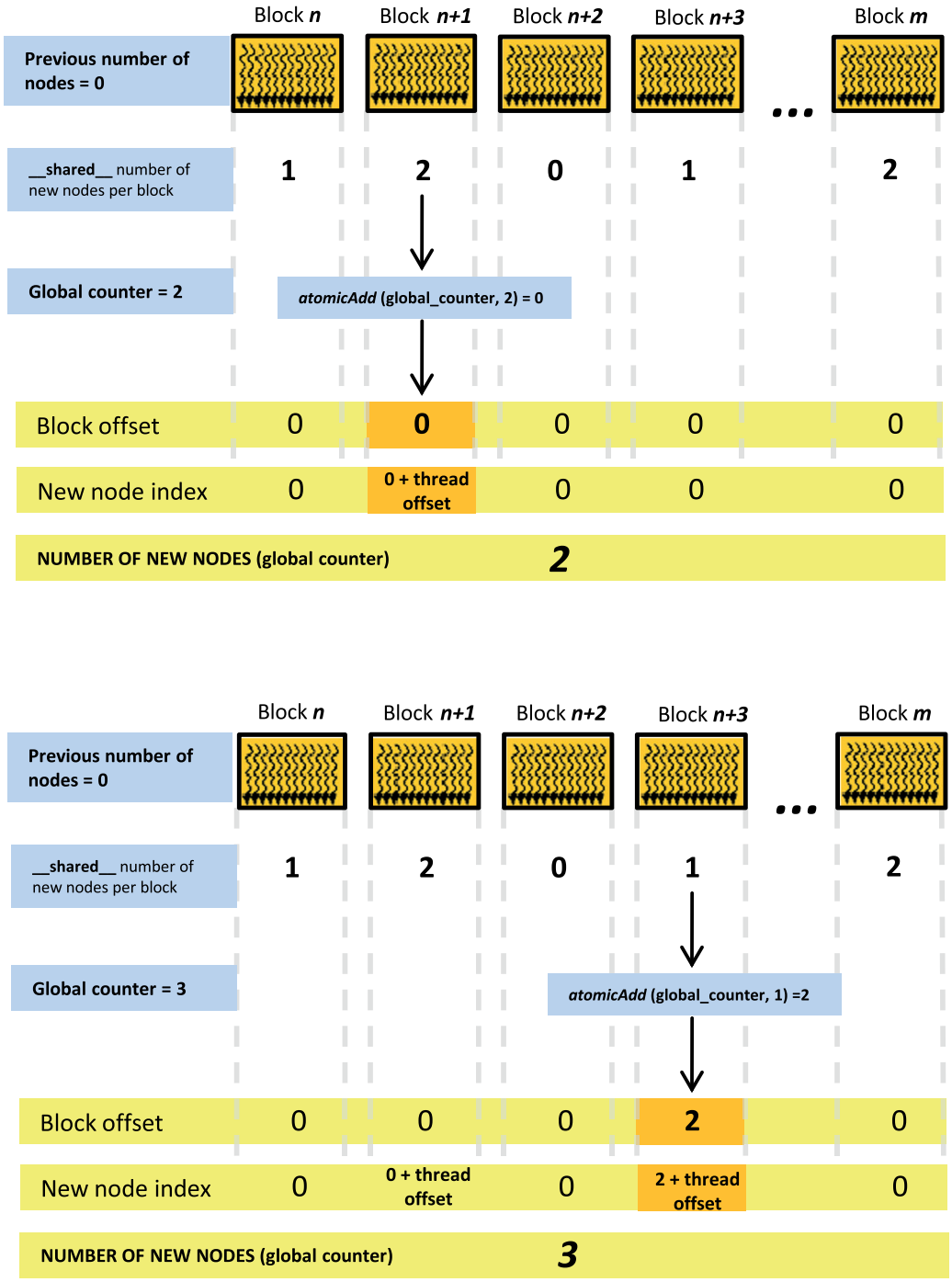


Figure A.2. Schematic demonstrating retrieval of the new node index from current block node counters.

To test the optimized insertion of cohesive elements, we fractured 5% of the facets at a time (as discussed in Section 7.2) and launched a kernel for each color. To color the mesh, we used the Welsh Powell algorithm [42]. It is important to highlight the performance boost when using a node counter in shared memory for each block. The results in the next dicussions show that the speedup rose, indicating that thousands to millions of threads updating the same memory address simultaneously is a bottleneck, and that shared memory's fast access as well as few threads updating the same

DOI: 10.1002/nme

Table A.1. Node index retrieving and appending using shared memory when inserting cohesive elements.

```
1: n ← Node to duplicate
2: if first thread then
3:     Number of New Nodes Per Block ← 0
4:     Block Offset ← 0
5: end if
6: syncthreads
7: Node Thread Offset ← atomicAdd(Number of New Nodes Per Block, 1)
8: syncthreads
9: if first thread then
10:    Block Offset ← atomicAdd(Global Node Counter, Number of New Nodes Per Block)
11: end if
12: syncthreads
13: New Node Index ← Block Offset + Node Thread Offset
14: NodeArray[New Node Index] = n
```

Table A.2. Mesh attributes performance results for T6 disc mesh [18] and its refined versions.

Bulk elements	Initial nodes	Final nodes	CZ elements	CPU Time (s)	GPU Time (s)	Speedup
240,000	481,200	1,440,000	359,400	9.29	0.0363	255.9
960,000	1,922,400	5,760,000	1,438,800	36.946	0.0778	474.9
2,160,000	4,323,600	12,960,000	3,238,200	84.94	0.1161	731.6
3,840,000	7,684,800	23,040,000	5,757,600	150.04	0.1761	852.0

GPU, graphics processor unit.

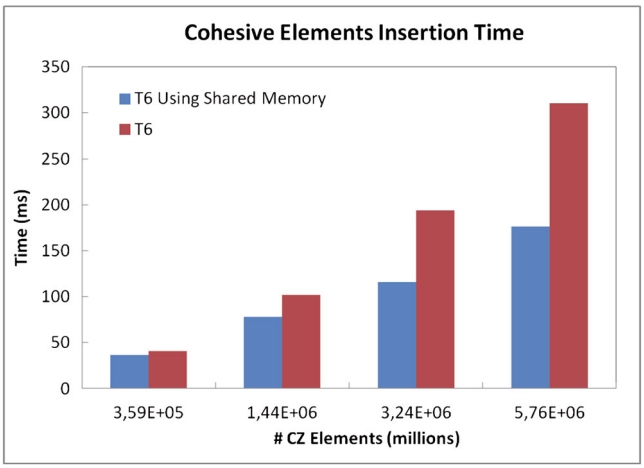


Figure A.3. Cohesive elements insertion time for T6 meshes using atomic functions in global or shared memory.

address can be a useful technique when duplicating nodes. Table A.2 shows the GPU results for T6 disc mesh and its refined versions compared to the CPU results, as well as mesh attributes before and after the simulation. Graphs A.3 and A.4 compare the GPU speedup and time of using and not using atomic functions in shared memory when inserting cohesive elements in T6 disc mesh and its refined versions.

DOI: 10.1002/nme

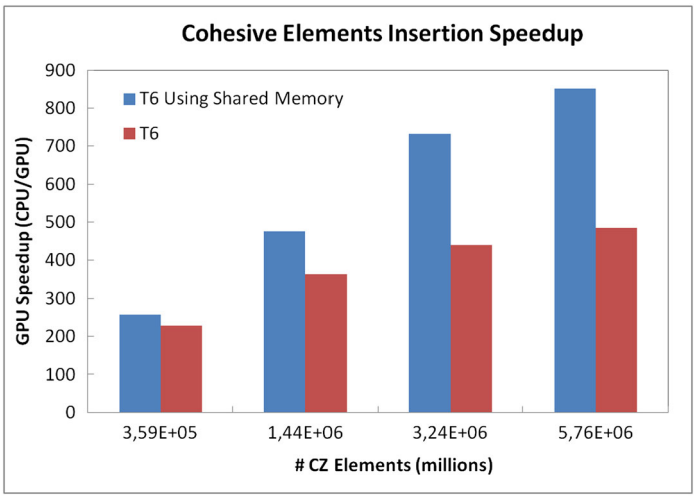


Figure A.4. Cohesive elements insertion speedup for T6 meshes using atomic functions in global or shared memory.

ACKNOWLEDGEMENTS

We acknowledge support from the US National Science Foundation under grant CMMI #1321661 and from the Donald B. and Elizabeth Willett endowment at the University of Illinois at Urbana-Champaign (UIUC). We also thank CNPq (Brazilian National Research and Development Council) for the financial support to conduct this research. We are grateful for insightful discussions with Ms. Sofie Leon during the preparation of this manuscript. Any opinion, finding, conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of the sponsors.

REFERENCES

1. Seegyoung Seol E, Shephard MS. Efficient distributed mesh data structure for parallel automated adaptive analysis. *Engineering with Computers* 2006; **22**(3):197–213.
2. Kirk BS, Peterson JW, Stogner RH, Carey GF. Libmesh: a c++ library for parallel adaptive mesh refinement/coarsening simulations. *Engineering with Computers* 2006; **22**(3):237–254.
3. Klein PA, Foulk JW, Chen EP, Wimmer SA, Gao HJ. Physics-based modeling of brittle fracture: cohesive formulations and the application of meshfree methods. *Theoretical and Applied Fracture Mechanics* 2001; **37**(1-3): 99–166.
4. Ortiz M, Pandolfi A. Finite-deformation irreversible cohesive elements for three-dimensional crack-propagation analysis. *International Journal for Numerical Methods in Engineering* 1999; **44**(9):1267–1282.
5. Cirak F, Ortiz M, Pandolfi A. A cohesive approach to thin-shell fracture and fragmentation. *Computer Methods in Applied Mechanics and Engineering* 2005; **194**(21-24):2604 –2618.
6. Zhang Z, Paulino GH, Celes W. Extrinsic cohesive modelling of dynamic fracture and microbranching instability in brittle materials. *International Journal for Numerical Methods in Engineering* 2007; **72**(8):893–923.
7. Belytschko T, Chen H, Xu J, Zi G. Dynamic crack propagation based on loss of hyperbolicity and a new discontinuous enrichment. *International Journal for Numerical Methods in Engineering* 2003; **58**(12):1873–1905.
8. Zhang ZJ, Paulino GH. Cohesive zone modeling of dynamic failure in homogeneous and functionally graded materials. *International Journal of Plasticity* 2005; **21**(6):1195–1254.
9. Sharon E, Gross P, Fineberg J. Local crack branching as a mechanism for instability in dynamic fracture. *Physical Review Letters* 1995; **74**(25):5096–5099.
10. Sharon E, Fineberg J. Microbranching instability and the dynamic fracture of brittle materials. *Physical Review B (Condensed Matter)* 1996; **54**(10):7128–7139.
11. Dooley I, Mangala S, Kale L, Geubelle P. Parallel simulations of dynamic fracture using extrinsic cohesive elements. *Journal of Scientific Computing* 2009; **39**(1):144–165.
12. Espinha R, Celes W, Rodriguez N, Paulino G. Partops: compact topological framework for parallel fragmentation simulations. *Engineering with Computers* 2009; **25**(4):345–365.
13. Arias I, Knap J, Chalivendra VB, Hong S, Ortiz M, Rosakis AJ. Numerical modelling and experimental validation of dynamic fracture events along weak planes. *Computer Methods in Applied Mechanics and Engineering* 2007; **196**:3833–3840.

DOI: 10.1002/nme

14. Molinari JF, Gazonas G, Raghupathy R, Rusinek A, Zhou F. The cohesive element approach to dynamic fragmentation: the question of energy convergence. *International Journal for Numerical Methods in Engineering* 2007; **69**: 484–503.
15. Paulino GH, Park K, Celes W, Espinha R. Adaptive dynamic cohesive fracture simulation using nodal perturbation and edge-swap operators. *International Journal for Numerical Methods in Engineering* 2010; **84**:1303–1343.
16. Lawlor O, Chakravorty S, Wilmarth T, Choudhury N, Dooley I, Zheng G, Kalé L. Parfum: a parallel framework for unstructured meshes for scalable dynamic physics applications. *Engineering with Computers* 2006; **22**(3):215–235.
17. Radovitzky R, Seagraves A, Tupek M, Noels L. A scalable 3D fracture and fragmentation algorithm based on a hybrid, discontinuous Galerkin, cohesive element method. *Computer Methods in Applied Mechanics and Engineering* 2011; **200**(1-4):326–344.
18. Bolz J, Farmer I, Grinspun E, Schrder P. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Transactions on Graphics* 2003; **22**:917–924.
19. Wu W, Heng PA. A hybrid condensed finite element model with GPU acceleration for interactive 3D soft tissue cutting: research articles. *Computer Animation and Virtual Worlds* 2004; **15**(3-4):219–227.
20. Krakiwsky SE, Turner LE, Okoniewski MM. Acceleration of finite-difference time-domain (FDTD) using graphics processor units (GPU). *Microwave Symposium Digest, 2004 IEEE MTT-S International* 2007; **2**:1033–1036.
21. Tejada E, Ertl T. Large steps in GPU-based deformable bodies simulation. *Simulation Modelling Practice and Theory* 2005; **13**(8):703–715.
22. Taylor Z, Cheng M, Ourselin S. High-speed nonlinear finite element analysis for surgical simulation using graphics processing units. *IEEE Transactions on Medical Imaging* 2008; **27**(5):650–663.
23. Göddeke D, Strzodka R, Mohd-Yusof J, McCormick P, Wobker H, Becker C, Turek S. Using GPUs to improve multigrid solver performance on a cluster. *International Journal of Computational Science and Engineering (IJCSE)* 2008; **4**(1):36–55.
24. Anderson JA, Lorenz CD, Travesset A. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics* 2008; **227**(10):5342–5359.
25. Rodriguez-Navarro J, Susin A. Non structured meshes for cloth GPU simulation using FEM 2006:1–7.
26. Göddeke D, Strzodka R, Turek S. Accelerating double precision FEM simulations with GPUs. *Proceedings of ASIM 2005 - 18th Symposium on Simulation Technique*, SCS Publishing House e.V., ASIM, 2005; 139–144.
27. Cecka C, Lew AJ, Darve E. Assembly of finite element methods on graphics processors. *International Journal for Numerical Methods in Engineering* 2011; **85**(5):640–669.
28. Geveler M, Ribbrock D, Göddeke D, Zajac P, Turek S. Towards a complete FEM-based simulation toolkit on GPUs: unstructured grid finite element geometric multigrid solvers with strong smoothers based on sparse approximate inverses. *Computers & Fluids* 2012; **80**:327–332.
29. Komatitsch D, Michéa D, Erlebacher G. Porting a high-order finite-element earthquake modeling application to Nvidia graphics cards using CUDA. *Journal of Parallel and Distributed Computing* 2009; **69**(5):451–460.
30. Liu Y, Jiao S, Wu W, De S. GPU accelerated fast FEM deformation simulation. *IEEE Asia Pacific Conference on Circuits and systems, 2008. APCCAS 2008*, Macao, 2008; 606–609.
31. Fan Z, Qiu F, Kaufman A, Yoakum-Stover S. GPU cluster for high performance computing. *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, Pittsburgh, PA, 2004; 12.
32. Godel N, Nunn N, Warburton T, Clemens M. Scalability of higher-order discontinuous Galerkin FEM computations for solving electromagnetic wave propagation problems on GPU clusters. *IEEE Transactions on Magnetics* 2010; **46**(8):3469–3472.
33. Kakay A, Westphal E, Hertel R. Speedup of FEM micromagnetic simulations with graphical processing units. *IEEE Transactions on Magnetics* 2010; **46**(6):2303–2306.
34. Ren DQ, Bracken E, Polstyanko S, Lambert N, Suda R, Giannacopulos D. Power aware parallel 3-D finite element mesh refinement performance modeling and analysis with CUDA/MPI on GPU and multi-core architecture. *IEEE Transactions on Magnetics* 2012; **48**(2):335–338.
35. Markall GR, Slemmer A, Ham DA, Kelly PHJ, Cantwell CD, Sherwin SJ. Finite element assembly strategies on multi-core and many-core architectures. *International Journal for Numerical Methods in Fluids* 2013; **71**(1):80–97.
36. Zegard T, Paulino HG. Toward GPU accelerated topology optimization on unstructured meshes. *Structural and Multidisciplinary Optimization* 2013; **48**:473–485.
37. Kirk DB, Hwu W-mW. *Programming Massively Parallel Processors: A Hands-On Approach* (1st edn). Morgan Kaufmann Publishers Inc.: San Francisco, CA, 2010.
38. Cuda c programming guide 3.2, 2010.
39. Sanders J, Kandrot E. *CUDA by Example: An Introduction to General-Purpose GPU Programming* (1st edn). Addison-Wesley Professional: Boston, MA, 2010.
40. Celes W ER, Paulino GH. A compact adjacency-based topological data structure for finite element mesh representation. *International Journal for Numerical Methods in Engineering* 2005; **64**(11):1529–1565.
41. Paulino GH, Celes W, Espinha R, Zhang ZJ. A general topology-based framework for adaptive insertion of cohesive elements in finite element meshes. *Engineering with Computers* 2008; **24**(1):59–78.
42. Welsh DJA, Powell MB. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal* 1967; **10**(1):85–86.
43. Pandolfi A, Ortiz M. Solid modeling aspects of three-dimensional fragmentation. *Engineering with Computers* 1998; **14**(4):287–308.

DOI: 10.1002/nme

44. Park K, Paulino GH, Roesler JR. A unified potential-based cohesive model of mixed-mode fracture. *Journal of the Mechanics and Physics of Solids* 2009; **57**(6):891–908.
45. Camacho GT, Ortiz M. Computational modelling of impact damage in brittle materials. *International Journal of Solids and Structures* 1996; **33**(20):2899–2938.

DOI: 10.1002/nme