Engineering with Computers (2016) 32:533–552
DOI 10.1007/s00366-015-0431-0

ORIGINAL ARTICLE

# Massively parallel adaptive mesh refinement and coarsening for dynamic fracture simulations

**Andrei Alhadeff[1] · Sofie E. Leon [2,3] · Waldemar Celes[1] · Glaucio H. Paulino[4]**

Received: 11 June 2015 / Accepted: 22 December 2015 / Published online: 18 January 2016


**Abstract** We use the graphical processing unit (GPU) to perform dynamic fracture simulation using adaptively refined and coarsened finite elements and the inter-element cohesive zone model. Due to the limited memory available on the GPU, we created a specialized data structure for efficient representation of the evolving mesh given. To achieve maximum efficiency, we perform finite element calculation on a nodal basis (i.e., by launching one thread per node and collecting contributions from neighboring elements) rather than by launching threads per element, which requires expensive graph coloring schemes to avoid concurrency issues. These developments made possible the parallel adaptive mesh refinement and coarsening schemes to systematically change the topology of the mesh. We investigate aspects of the parallel implementation through microbranching examples, which has been explored experimentally and numerically in the literature. First, we use a reduced-scale version of the experimental specimen to demonstrate the impact of variation in floating point operations on the final fracture pattern. Interestingly, the parallel approach adds some randomness into the finite element simulation on the structured mesh in a similar way as would be expected from a random mesh. Next, we take advantage of the speedup of the implementation over a similar serial implementation to simulate a specimen whose size matches that of the actual experiment. At this scale, we are able to make more direct comparisons to the original experiment and find excellent agreement with those results.

**Keywords** GPU computing · Finite elements · Adaptive mesh refinement · Adaptive mesh coarsening · Cohesive zone model · Microbranching

## 1 Introduction

We use the graphical processing unit (GPU) to perform dynamic fracture simulation using adaptively refined and coarsened finite elements and the inter-element cohesive zone model. The massively parallel nature of the GPU results in significant speedup over similar adaptive mesh refinement and coarsening strategies implemented on a CPU.

In a related work, researchers developed a framework to perform dynamic fracture simulation on a static bulk mesh [1]. While large speedup was achieved because of ability to perform floating point operations very quickly, the size of the system was limited due to the inherent memory restrictions of the GPU architecture. In the present work, we aim to solve larger problems that demand more memory due to the sizes of the finite element meshes. Thus, we develop an adaptive refinement and coarsening scheme (AMR+C) suitable for the GPU architecture to ensure only the most important information is stored during the simulation. The AMR+C will allow for analysis of

Andrei Alhadeff and Sofie E. Leon equally contributed to this work.

✉ Waldemar Celes
celes@tecgraf.puc-rio.br

[1] Tecgraf/PUC-Rio Institute, Pontifical Catholic University, Rio de Janiero, Brazil

[2] Department of Civil and Environmental Engineering, University of Illinois at Urbana-Champaign, Champaign, USA

[3] Center for Research and Interdisciplinary, Paris, France

[4] School of Civil and Environmental Engineering, Georgia Institute of Technology, Atlanta, USA

much larger systems than that of an equivalent uniform mesh as we only use the finest level of refinement where necessary. Performing adaptivity on the GPU is not a straightforward task though, as a new mesh representation data structure, finite element calculation framework, and refinement and coarsening algorithms are necessary.

Before going into the details of adaptive mesh refinement and coarsening for dynamic fracture applications, it is useful to first provide some background of the GPU architecture and comparison to a central processing unit (CPU) system. CPUs are a multi-core system, which are optimized for serial programming with sophisticated control logic and access to ample cache memory. However, neither control logic nor access to memory improves the peak calculation speed, and by about 2003 software developers were making more advances than could be supported with the existing hardware. The speed of an individual CPU is ultimately limited by energy consumption and heat-dissipation issues; leading developers moved to many-core environment of the GPU. The many-core approach focuses on execution throughput of parallel applications. Conversely, the GPU comprised a large number of small cores, so computationally intensive parts of a code can be moved to the GPU where it can be divided. Moreover, many-core systems have up to 10 times higher memory bandwidth than multi-core systems, making it possible to move data in and out of its memory much faster. Much of the GPU development has been motivated by the gaming industry in which massive numbers of floating point operations are required. In order to accommodate this demand, developers optimize codes for execution throughput of a massive number of threads; more chip area is given for floating point operations rather than to memory to increase throughput. To summarize, GPUs are well-suited for problems involving a large number of floating point calculations, thus they are not universally faster than CPU systems as certain applications are simply not suited for them [2]. In general, the developer must take great care to ensure that the benefits of the GPU outweigh the constraints for a particular application, or the performance will suffer greatly.

In the past decade, researchers have ported multi-core applications to many-core systems for increased performance. See [3] for an overview of the state of the practice and trends an GPU computing. In the field of finite elements, recent efforts have been focused on linear system assembly, storage, and solution. Researchers in [4] propose an iterative technique to generate space matrices on several GPUs in order to overcome limited storage space. Memory issues are also addressed in [5], where techniques for assembling finite element matrices are explored. Solution of linear systems is also non-trivial, especially on GPUs. To address this, the authors of [6] propose a massively parallel Cholesky factorization technique for use on GPUs.

Multi-core dynamic fracture simulations have been studied by several authors in the past. For example, researchers [7] investigate cohesive dynamic fracture in a parallel CPU environment ParFUM framework [8]. The cohesive elements require an external activation criteria and do not feature the initial elastic slope present in the intrinsic model, the interface elements are present at all facets before the start of the simulation. Nodes at all facets are duplicated from the beginning, but the traction separation relationship is not activated until the external stress-based criteria are met. In this way, the work is more similar to an intrinsic implementation because the mesh connectivity does not change as the problem evolves. In a related work, authors in [9] also pre-insert externally active cohesive elements into the domain. They use a discontinuous Galerkin approach, and because the mesh topology does not change, they are able to attain scalability of the parallel implementation. More recently, fully extrinsic dynamic fracture simulation was achieved in [10]. The work is based on the ParTopS data structure and supports insertion of extrinsic cohesive elements on-the-fly.

Fracture simulation using the many-core GPU environment and adaptive finite element mesh operations are not well documented in the literature. To the best knowledge of the author, adaptive dynamic fracture simulation on GPUs has not been investigated. Adaptivity has been explored in other fields, for example cartesian meshes are generated on the GPU for computational fluid dynamics applications resulting in speedup of up to 36 for large meshes [11]. Dynamic fracture with the extrinsic cohesive zone model on a uniform mesh is investigated and implemented in [1], and serves as the motivation and foundation on which the proposed adaptive GPU fracture is based.

In this work, we present a massively parallel adaptive finite element analysis for dynamic fracture simulations. Cohesive interface elements are adaptively inserted along bulk elements boundaries where and when needed, according to the analysis fracture criterion. Secondly, and more computationally demanding, bulk elements are adaptively refined and coarsened in order to reduce mesh size while using appropriate mesh resolution on critical regions.

The remainder of this paper is organized as follows. Our methodology for conducting adaptive mesh modification for dynamic fracture simulation, including the physical basis for the work and computational implementation, is discussed in great detail in the Sect. 2. Section 3 describes our data structure and explains mesh modification algorithms on the GPU, such as mesh refinement and coarsening and insertion of cohesive elements. Equipped with a computational framework to conduct large-scale fracture simulation quickly, we explore some numerical investigations

in Sect. 4. We simulate benchmark problems with accepted results in the literature, but investigate features of the GPU implementation that have yet to be explored. Finally, we close in Sect. 5 with a summary of the contribution and discussion of potential future research directions.

## 2 Numerical representation of dynamic fracture

The primary application of this work is dynamic crack propagation and nucleation. We simulate such problems using the cohesive zone model approach in which the fracture process zone ahead of the crack tip is approximated by a nonlinear traction-separation relation. This approach is attractive in its simplicity: the degrading and softening mechanisms where micro-cracks and voids initiate and coalesce ahead of the crack tip are not explicitly modeled, rather they are approximated by the cohesive zone [12, 13]. The concept is illustrated by the simple schematics shown in Figs. 1 and 2. The macro-crack tip contains zero tractions and complete separation, then ahead of this point the traction increases and opening decreases.

The cohesive zone model approach can be incorporated into a number of numerical frameworks. In this work we limit our attention to the inter-element approach, in which the cohesive elements are only present at the bulk finite element boundaries. We employ standard quadratic triangular elements to represent the continuum behavior. The cohesive elements are quadratic edge elements encoded with the traction-separation relationship, given by the PPR potential-based cohesive zone model. We will not address the model in detail here, instead the reader is referred to [14] for the derivation of the model and to [15] for a comparison of the model to others available in the literature. The extrinsic model is utilized, which allows for arbitrary crack growth (so long as it is along element boundaries) without restriction to predefined or preexisting fracture planes [16].

We consider temporal effects when modeling crack propagation through the explicit central difference time integration scheme with a lumped mass matrix [17]. This eliminates the need to solve the linear system when calculating the nodal displacements, which makes the continuum problem without adaptivity readily parallelizable. However, in this work, the mesh evolves in time via insertion of cohesive elements, mesh refinement, and mesh coarsening; thus parallelization is not at all trivial. The details of the GPU parallelization of this approach are described in detail in the next section.

## 3 Adaptive mesh modification

Mesh adaptivity is important in the context of finite element applications, as it enables larger simulations to be performed in less computational time. Of course, adaptivity has been widely utilized in the context of material fracture and failure modeling, which has been demonstrated throughout the course of this thesis. However, mesh adaptivity and hierarchical schemes are also utilized in other fields such as modeling electronic chip packages [18], large-eddy simulations [19], and astrophysical thermonuclear flash [20], just to name a few. We continue the focus of this work on dynamic fracture simulation and develop the data structure and algorithm for adaptive modification of the finite element mesh, namely mesh refinement and coarsening. The data structure is detailed in the following subsections.

### 3.1 Data structure for 4k adaptive finite element mesh representation

An efficient data structure is critical in adaptive fracture applications. From a data representation perspective, the implications of inserting a cohesive element or refining or coarsening a region of the mesh involves dynamically changing the size of the node/element representation and updating adjacency relations. If this is not done in an efficient way with respect to computational processing time, then the cost of solving even a small problem could

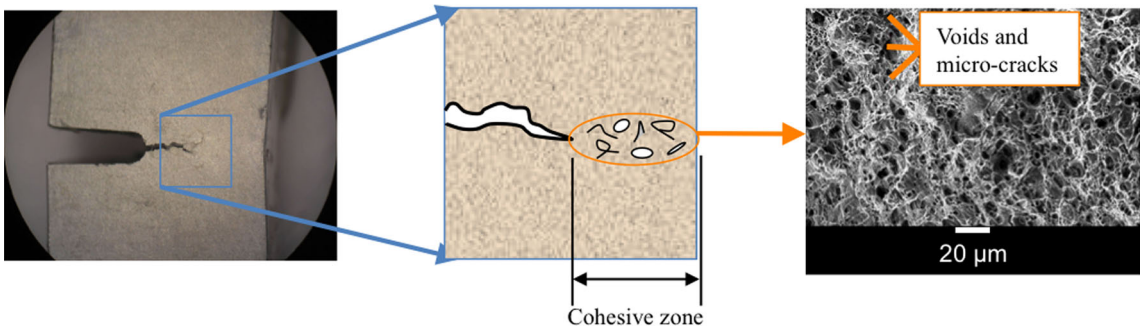



**Fig. 1** Schematic of the cohesive zone model approach. The cohesive zone ahead of the macro-crack tip consists of voids and micro-cracks

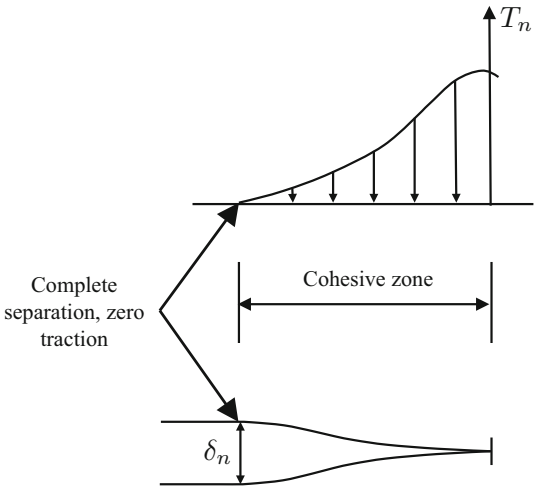



**Fig. 2** The surfaces of the macro-crack tip are completely separated and are traction free. Then separation decreases and traction increases into the cohesive zone

be dominated by upkeep of the mesh representation instead of on the structural mechanics computations.

Several data structures have been devised including those for dynamic fracture with adaptive insertion of cohesive elements and adaptive topological operators on a serial CPU platform [21], dynamic fracture with adaptive insertion of cohesive elements on a parallel CPU platform [10], and dynamic fracture with adaptive insertion of cohesive elements on a massively parallel GPU platform [1]. However, none of these previous approaches are appropriate for the present work of adaptivity on a GPU platform for a variety of reasons. First, data structures for serial platforms are not equipped to handle issues of concurrency which is critical in parallel simulations. Secondly, the CPU data structures (for serial or parallel platforms) do not have nearly the space restrictions as that of a GPU; we need to store far fewer entities explicitly on the GPU and instead derive them each time they are accessed. Finally, problems in which the bulk mesh remains constant throughout the simulation do not pose the challenges of constant insertion and deletion of variables in the data structure, as we have in the adaptive simulations proposed here.

Therefore, given the requirements of the adaptive simulation and limitations imposed by the GPU architecture, we propose a simple and inexpensive data structure for representation of the evolving 4k structured finite element mesh. The 4k mesh is composed of triangular elements that meet at vertices of 4 or 8 facets. It is constructed from a quadrilateral grid where each cell contains 4 triangular elements that form an "X" pattern in the cell (see Fig. 3a).

Since the data structure was developed for this particular type of mesh, we take advantage of its properties to reduce storage demands by avoiding extra fields that would be necessary for a generic mesh. Not only does the data structure save memory over other implementations, but it is simple and compact enough to minimize global memory access and hide latency. For example, it is organized as structures of arrays and transposed matrices (column-major matrices) in order to perform coalesced memory access. Thanks to the use of the specialized triangular mesh, we avoid many extra fields in the data structure that a generic mesh would require. Moreover, the proposed data structure is based simply on a table of nodes and a table of elements. Since we only store limited adjacency information, traversing nodes and elements requires more "on-the-fly" computations. However, this is acceptable because it is done on the GPU using registers with fast access. For example, we save space by not storing information related to the topological edges of the mesh or a table of incidence for each element. In such a table, each quadratic, triangular element would contain three values: the incidence of the node in the bulk element pointing to the next opposite element in the order of traversal. This would greatly help during element traversal, but it would consume more device memory and require more global memory access. Instead, we save the memory by only storing one incident node per element and perform extra computations on the GPU with registers.

The 4k mesh is a mesh that is well suited for refinement and coarsening, because the structure is maintained as facets are split or merged. The data structure consists of node and element tables with some basic adjacencies and information necessary for the hierarchical refinement and coarsening scheme. A schematic of a sample mesh and corresponding data structure representations is shown in Fig. 3; we will refer back to this figure several times in the next few sections to aid discussion.

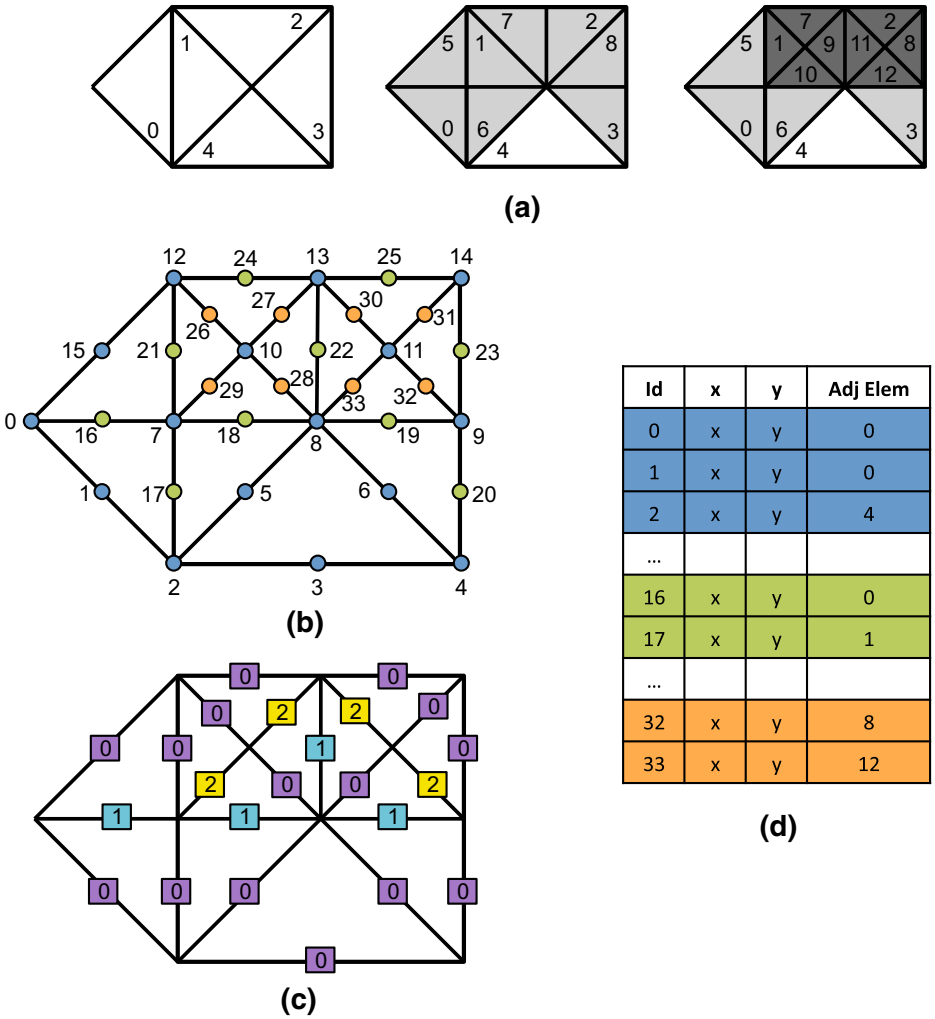
The sample coarse mesh is partially refined in three steps, as shown in Fig. 3a and the corresponding nodal number of the final mesh is shown in Fig. 3b. All of the data necessary to store this adaptive mesh are contained in the node table (Fig. 3d) and element table (Fig. 3e).

### 3.2 Nodal data

The node table contains all information related to the nodes of the mesh. The x and y values describe its 2-dimensional spatial position. The adjacent element refers to the ID of a bulk element (cross reference the element table) that is adjacent to the node. Storage of this adjacent bulk element is critical as it is frequently used to begin the process of traversing all of the adjacent elements of a given node.

### 3.3 Element data

The element table contains information associated to the bulk and cohesive elements in the mesh, each of which




| Id | x | y | Adj Elem |
|---|---|---|---|
| 0 | x | y | 0 |
| 1 | x | y | 0 |
| 2 | x | y | 4 |
| ... | | | |
| 16 | x | y | 0 |
| 17 | x | y | 1 |
| ... | | | |
| 32 | x | y | 8 |
| 33 | x | y | 12 |

| Id | v0 | v1 | v2 | v3 | v4 | v5 | $O_0$ | $O_1$ | $O_2$ | Level | Ref | Labels |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 2 | 7 | 1 | 17 | 16 | 6 | 5 | -1 | 1 | 0 | 0-0-1 |
| 1 | 12 | 7 | 10 | 21 | 29 | 26 | 10 | 7 | 5 | 2 | 1 | 0-2-0 |
| .. | | | | | | | | | | | | |
| 3 | 8 | 4 | 9 | 6 | 20 | 19 | -1 | 12 | 4 | 1 | 3 | 0-0-1 |
| 4 | 2 | 4 | 8 | 3 | 6 | 5 | 3 | 6 | -1 | 0 | 4 | 0-0-0 |
| .. | | | | | | | | | | | | |
| 11 | 13 | 8 | 11 | 22 | 33 | 30 | 12 | 2 | 9 | 2 | 2 | 1-0-2 |
| 12 | 8 | 9 | 11 | 19 | 32 | 33 | 8 | 11 | 2 | 2 | 2 | 1-2-0 |

**Fig. 3** Schematic of GPU data structure for adaptive 4k mesh: **a** progression of mesh refinement and element labeling; **b** node numbering on refined mesh; **c** facets labels indicating order of refinement on refined mesh; **d** node table showing node ids, coordinates, and adjacent element; **e** element table showing element id, nodal connectivity, adjacent elements, reference level, level of refinement, and facet labels

are referred to by their ID. The bulk elements used in this work are quadratic triangular elements (T6), so they contain six node IDs (cross reference the node table): three corner nodes followed by three mid-side nodes. Next are the IDs of three adjacent elements (bulk or cohesive) that are opposite to the corner nodes of the element. While not shown in Fig. 3e, the cohesive elements are also stored explicitly in the element table. To be consistent with the quadratic bulk element, the cohesive elements also contain six nodes (three per

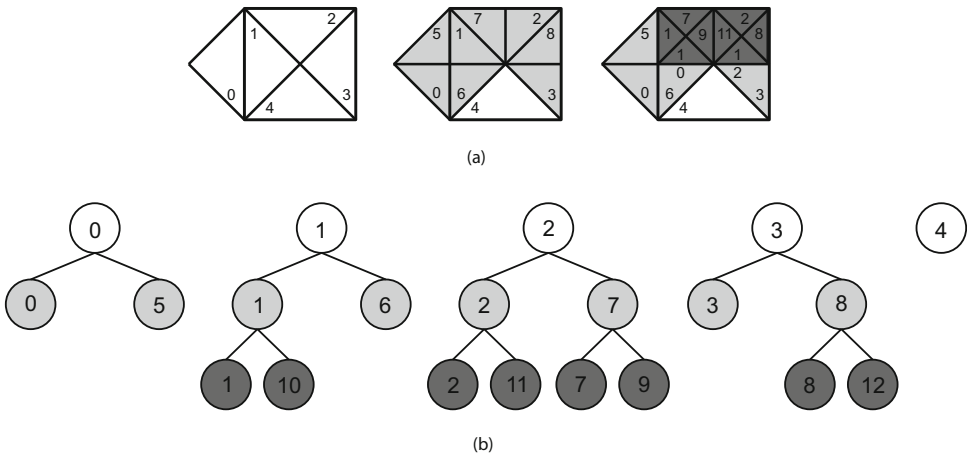



**Fig. 4** **a** Refinement of elements 0, 1, 2 and 3 from level 0 (*white*) to level 2 (*dark grey*), **b** binary tree representation of refinement

facet) but only contain two adjacent elements (one for each facet).

### 3.4 Data storage necessary for mesh refinement

As bulk elements are subdivided, they are assigned a new level of refinement; Fig. 3a illustrates the elements at level 0 shown in white, elements at level 1 shown in light grey, and elements at level 2 shown in dark grey. These levels are stored in the element table as shown in Fig. 3e. In addition to the element-level information, we adopt a edge labeling technique to assist in mesh coarsening in which new edges resulting from mesh refinement are assigned a label with a number equal to one more than its adjacent edges. For example, when the simulation begins, the edges of each element are labeled 0, then the new edges resulting from one level of refinement are labeled 1, next new edges resulting from a second level of refinement are labeled 2, and so on. The edge labels are shown on the schematic in Fig. 3c and as a column of the element table in Fig. 3e. Because edges are not represented as an explicit entity in our data structure, we store three labels per bulk element (one per edge). Fig. 4 illustrates the hierarchical representation of element subdivision in the refinement strategy.

To achieve an efficient element refinement strategy, it is necessary to be able to quickly determine if an element is inside or outside of a refinement region (read on to Sect. 3 for details on what constitutes a refinement region). Thus, we identify regions of refinement on a background grid because it is more efficient than identifying the elements directly. If the midpoint of any of the facets of an element is contained within a grid cell, then that element needs to be refined. The grid attributes are stored in constant memory and we maintain a counter per cell to indicate if the cell is inside or outside of refinement region.

The size of the element table is allocated before the simulation begins. Because the adaptive simulation grows with element refinement and insertion of cohesive elements, we must estimate of the final mesh size to ensure there is adequate storage.

### 3.5 Data storage necessary for mesh coarsening

To enable quick updates to the mesh adjacency during coarsening, we store the ID of the coarse element (parent) from which two elements (children) emerge during refinement from level n to level n + 1 as the reference element in Fig. 3e. This enables us to quickly access the coarse bulk element that should result from merging (coarsening) two fine elements.

When coarsening the mesh, elements and nodes are removed from the pre-allocated tables, which results in "holes" in the data structure. Rather than collapsing the data structure by renumbering the mesh entities (we explored this approach but concluded that it was too computationally expensive for the present application), we opt to fill the holes the next time a node or element is inserted into the mesh. Thus, we need to keep track of unused node and element IDs, which we do through a node and element stack. When new nodes/elements are added to the mesh, we first insert them at the indices stored in the respective stack. Then, once the stack is empty, we add nodes/elements by creating new entities in the node/element tables, respectively. We avoid coarsening of cohesive elements, therefore there is no need to store additional information for adaptivity. The data structure stores two stacks, one for the nodes and one for the elements, and two stack counters to reference the top of the stack.

### 3.6 Non-topological data storage

We allocate space for non-topological attributes used in the numerical simulation, e.g., displacements, velocities, accelerations, nodal forces, stresses, strains, stiffness matrices,

mass, and elastic material properties. Some of these are read and written, while others are read-only attributes.

Attributes that are read and written frequently (e.g., displacements, velocities, stress, cohesive traction) are stored in global memory. As mentioned previously, global memory is time consuming to access, so when possible we minimize use of this memory. Due to the memory access issues on the GPU, we chose to only implement a small deformation, isoparametric formulation for the numerical simulations. Using this assumption we can calculate the element stiffness matrices once without need for updates, which would be necessary in a finite deformation case. Therefore, the element stiffness matrices are stored in texture memory, which is read-only and can be accessed faster than global memory. Moreover, we utilize the central difference time integration with a lumped mass matrix and we neglect effects of damping, therefore we arrive at a fully explicit time integration scheme. In practice this means that we do not have to assemble and store the system matrices because solution of the linear system is trivial. Instead we can solve the dynamic time integration equations one element or node at a time (see the next section for details on these calculations). Finally, we store the material properties of homogenous domains in constant memory. This is similar to texture memory in that it is read only; however, the space is smaller so it is faster to access.

### 3.7 Node and element calculations

In parallel finite element simulations, it is critical to avoid concurrently writing to the same location in memory. For example, node quantities (e.g., displacements, nodal force, etc.) are composed of contributions from adjacent elements. Writing conflicts will arise if multiple elements are updating a node at the same time. Typically, this issue is handled with graph coloring schemes, such as that proposed in [22], whereby the color groups are visited in a serial fashion, and one thread is launched per element in that color group. This is usually done once at the start and absorbed in the overhead cost as it is a relatively expensive operation for arbitrary meshes. In the adaptive simulation, however, the number of bulk elements and their connectivity will change. Therefore the coloring algorithm would need to be executed every time a bulk element was removed or inserted, which would be computationally inefficient even on the GPU.

Our solution for the adaptive simulation is to sweep the nodes (one thread per node) and gather information from its adjacent elements, rather than sweeping elements (one thread per element) and updating its adjacent nodes. The data structure allows for quick access the elements adjacent to the nodes through use of the adjacent bulk elements stored in the element table. Using the node-based calculations, elements will be visited concurrently, as nodes within a close vicinity will share adjacent elements. However, since data will only be read from the element entities, the issue of concurrent writing is not present. The efficiency of this approach is similar to that of the element sweep approach. When mesh refinement and coarsening are enabled, this approach leads to some variation in final crack patterns and results from one simulation to another. While this variation is not incorrect it warrants some investigation, which is conducted via numerical experiment in Sect. 4.

### 3.8 Adaptive insertion of cohesive elements

The extrinsic cohesive model employed in this work requires an external criteria to activate (i.e., insert) the traction-separation relation associated with the cohesive element. A number of approaches have been utilized to activate the elements including strength/stress, strain, velocity, numerical instability, etc. We will assume that elements have been activated for the following discussion. When cohesive elements are inserted into the mesh, the nodes along the insertion facet are duplicated. The cohesive element is defined by the original nodes along the facet (three in this case of quadratic triangular elements) plus the additional nodes resulting from duplication.

As mentioned in Sect. 3.7, we do not utilize a graph coloring scheme in this work. One of the implications of our node-based approach is that we cannot use previous strategies, such as those adopted in [1, 10], to insert cohesive elements at bulk element facets. Instead we utilize a two-step node sweep strategy, which is analogous to the update scheme discussed in Sect. 3.7.

Figures 5, 6 illustrate the algorithm for inserting the cohesive elements on a non-colored quadratic triangular mesh (T6 elements). In this example, three cohesive elements will be inserted at the bold black facets in Fig. 5a. After detecting the facets which need a cohesive element inserted (fractured facet), we begin the procedure for inserting cohesive elements and duplicating nodes. In the first kernel, we launch one thread per bulk element that contains at least one fractured facet. Because a facet is shared by two bulk elements, we choose the element with the smaller ID in the element table to be responsible for inserting the cohesive element. The bulk element sweeps its facets and inserts the cohesive elements on the fractured facets by adding them to the element table. While inserting the cohesive elements we also duplicate the mid-side nodes by adding a new node to the node table for each cohesive

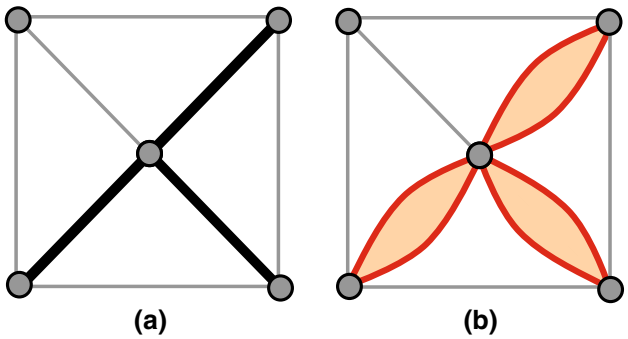



**Fig. 5** Cohesive elements are inserted on fractured facets (in *bold black*) by launching one thread per bulk element that contains at least one fractured facet. Mid-side nodes are also duplicated

element inserted. The result is shown in Fig. 5b. We mark the corner nodes that belong to the cohesive elements so we can determine which ones need to be duplicated; however, the duplication is not done in this step. Once all of the cohesive elements are inserted in parallel, then we begin the second step of duplicating corner nodes.

The nodes marked in the previous kernel will be filtered using a simple CUDA Scan and Compact operation. In the second kernel, we launch one thread per filtered node and check them for duplication, as illustrated in Fig. 6b. Starting from the node's adjacent element (stored in element table) we traverse all incident elements. The algorithm begins by accessing the adjacent element from the node in the data structure and traversing the elements in one direction, until the first cohesive element is reached. This cohesive element will serve as the reference element. Then the direction is switched and the elements adjacent to the node are traversed. The node is duplicated each time a cohesive element is passed. When we arrive back at the reference element the procedure is complete. This algorithm does not cause issues of concurrent writing because each thread is responsible for duplicating its own node.

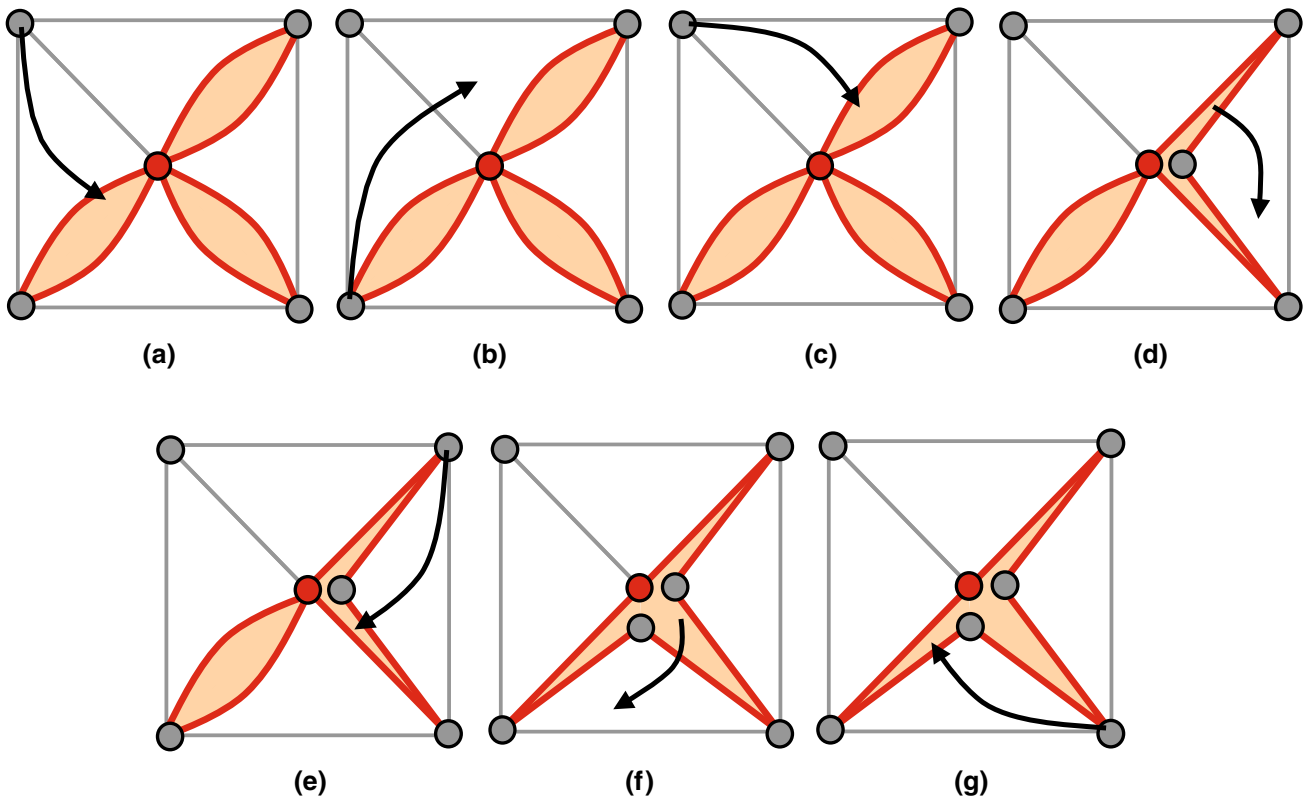



**Fig. 6** To determine if a node should be duplicated, we traverse the elements adjacent to the node. **a** Traversal begins in one direction until first cohesive element is reached, it will serve as a reference point. The traversal direction changes and **b** a bulk element is reached, **c** then the second cohesive element is reached, **d** then a bulk element is reached. When the cohesive element is passed the node is duplicated. Traversal continues and **e** the third cohesive element is reached, **f** then the last bulk element is reached. The node is duplicated again when the cohesive element is passed. **g** Traversal and node duplication stops when the first cohesive element is reached again

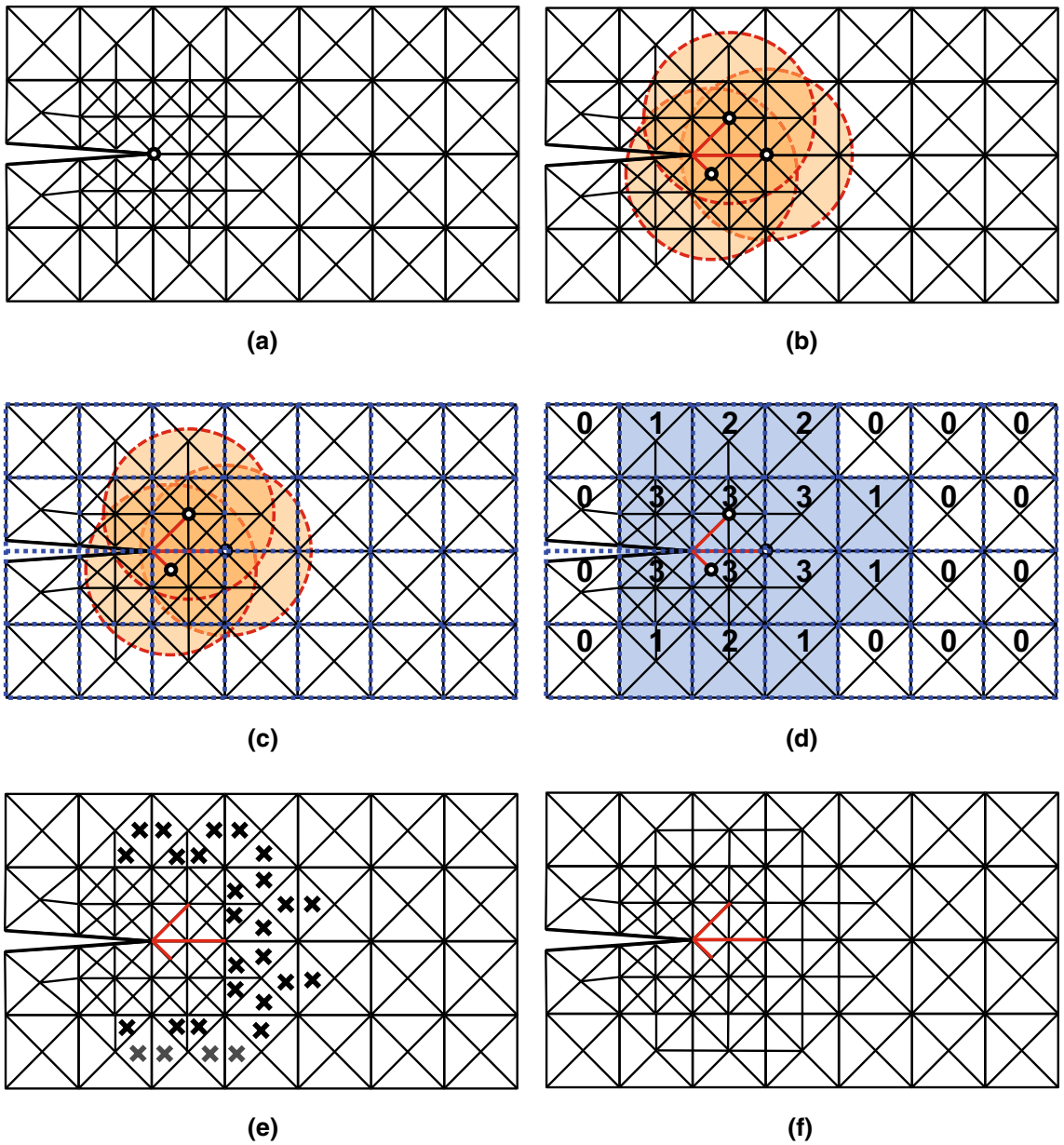



**Fig. 7** 4k refinement scheme. **a** Mesh is initially refined around the notch tip. **b** Cohesive elements are inserted along facets of fully refined elements, new crack tips are identified and new refinement regions associated with each crack tip are created. **c** A background grid, shown in dashed blue, is used to identify which elements are within the refinement region. **d** The grid cell counters indicate the number of refinement regions present in the cell. **e** The distance between the crack tips and the elements with a grid cell counter greater than 0 is calculated. If the distance is less than a user-defined value then the element is marked (*black* '*x*') and elements adjacent to the hypotenuse of a marked element are marked (*grey* |'x'). **f** Marked elements are refined to level 1, then the process repeats to refine the marked elements to level 2 and so on

### 3.9 Adaptive mesh refinement

One of the main contributions of this work is the development and implementation of the 4k refinement and coarsening scheme on the GPU. The criteria for which is similar to that of [23] and are briefly reviewed here before describing the GPU implementation. Multiple crack tips may emerge as the fracture simulation evolves in time, where we define a crack tip as an unduplicated nodes of a cohesive element. These crack tips are necessary to per-

form adaptive mesh refinement, as we use the a priori assumption that regions around crack tips (i.e., high gradient of the displacement field) must be the finest in the simulation. Given a crack tip, elements that fall within a user-specified radius are refined according to the hierarchical 4k scheme to a user-defined level. To avoid complicated transfer of internal state variables associated with the nonlinear cohesive elements, this refinement strategy prohibits cohesive elements from being refined or coarsened. Thus, cohesive elements may only be inserted at elements that are already refined to the highest level. This assumption is generally acceptable, as we expect cracks to initiate from areas that are fully refined, e.g., initial defects, notch, or crack tip.

The algorithm to refine bulk elements in a certain region of a 4k mesh is a multi-step procedure described as follows and corresponds to Algorithm 1. The notation we use in Algorithm 1, $<<<\mathrm{x}>>>$, indicates that a kernel call is being made where x indicates the number of threads launched. The scheme is demonstrated from a topological perspective in Fig. 7. The domain in the figure contains an initial notch which is refined (Fig. 7a). Next step cohesive elements are inserted (notice that they are also inserted along facets of fully refined elements) and the crack tip nodes are updated.

As new crack tips emerge during the simulation, new corresponding refinement regions must be created (Fig. 7b). An element is refined if at least one of its midside node is inside a refinement region. As discussed in Sect. 3.1, we use a regular grid to determine which elements need to be refined (Fig. 7c). Each grid cell stores a counter that indicates the number of refinement regions it belongs to. The cell size is chosen based on the size of the refinement region. If a cell is inside a refinement region, its counter is incremented by one. Cells that remain with zero counter are outside refinement regions. An element is said to be inside a refinement region if the cell it belongs to is marked with a value greater than zero. We launch one thread per cell and calculate the distance between the center of the cell and the center of each new refinement region. If the distance is less than the user-defined radius, then we increment the cell counter (Fig. 7c).

The loop in Algorithm 1 begins by refining elements to level 1, once all eligible elements are refined to level 1, then we move to level 2 and so on until the desired level of refinement is reached. The first kernel call launches one thread per bulk element and if the midpoint of at least one facet of the element has its cell counter greater than zero, then the element is marked for refinement (lines 4–5 in Algorithm 1 and the black 'x' in Fig. 7e). In the next kernel call, for each marked element we mark the element adjacent to the hypotenuse of the originally marked element (lines 7–8 in Algorithm and the grey 'x' in Fig. 7e). This is also illustrated in Fig. 8, where the facets between elements 2 and 1 and elements 2 and 3 have at least one node that falls within the radius of refinement shown as the light grey semi-circle, so all three elements are marked. Element 0 is adjacent to the hypotenuse of marked element 1, so it is also marked. This procedure is executed until there are no more marked elements (line 9 in Algorithm 1). Note that a scan operation is performed to get the total number of marked elements (line 8 in Algorithm 1).

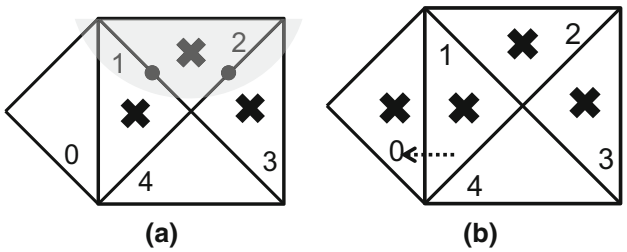



**Fig. 8** Marked opposite elements. **a** Elements with at least one node inside the refinement region are marked, **b** elements adjacent a marked element's hypotenuse are also marked

In the next kernel call we launch one thread per marked element and split it according to the 4k hierarchical refinement strategy, i.e., split the element along its longest edge [24]. It is useful to note that the first two nodes in a row of the element table are the corner nodes that define the hypotenuse of the element (see Fig. 3e), thus the longest facet of an element is directly accessible and does not require additional calculations. New nodes/elements are created in this step by either adding them to the node/element tables or by reusing node/element IDs from their respective stacks (line 10 in Algorithm 1). The last kernel updates adjacency of the newly added elements in the element table (line 11 in Algorithm 1). This procedure is continued until all elements inside the refinement region reach the level 1, then it starts again for level 2, and so on until all elements inside the refinement region reach the level prescribed by the user or there are no more marked elements. The resulting mesh after refinement of one level is shown in Fig. 7f.

**Algorithm 1** Kernel based algorithm to perform adaptive mesh refinement

```
1  RefineCUDA ()
2    MarkRefinedRegionCells <<<nCells>>>
3    do {
4      MarkElements<<<nElems>>>
5      nMarkedElems = ScanMarkedElems<<<nElems>>>
6      do {
7        MarkNeighbors <<<nMarkedElems>>>
8        nMarkedNeighbors = ScanMarkedNeighbors<<<nElems>>>
9      } while numMarkedNeighbors != 0
10     SplitFacets <<<totalNumMarkedElems>>>
11     Update adjacency <<<totalNumMarkedElems>>>
12   } while there are marked elements
13 end RefineCUDA
```

### 3.10 Adaptive mesh coarsening

For adaptive mesh coarsening, we can deduce that in regions far away from the crack tip, a coarser mesh is sufficient. However, unlike refinement, the coarsening criteria are not only based on an element's geometric position relative to the crack tip. Rather, it is also based on convergence of the norm of the strain of the coarsened mesh to that of the refined mesh. However, in regions near crack tips, coarsening does not occur. Thus, cells are marked and used also as a criteria for coarsening by verifying if they are outside existing refining regions. The error between the norm of the strain in a patch of the refined 4k element is compared to the norm of the strain in the same patch but with coarse elements. If the error is less than a certain threshold, 2 % in this study, then the patch is coarsened. Since the finite element space is becoming less rich, energy conservation is not expected; however, the loss is minimal and justified by the gain in memory and processing time.

The parallel coarsening algorithm is essentially the reverse of the refinement; however, the implementation on the GPU must be done in such a way to ensure that concurrency is avoided. First, a kernel with one thread per element is launched where the bulk elements are marked if the mid points of all of its facets are outside of an existing refinement region (lines 4–5 in Algorithm 2). This is done by verifying if the cell counter is equal to zero and if its level of refinement is greater than zero. After bulk elements are marked, the nodes are visited through a kernel call by launching one thread per node. An interior node is marked for coarsening if (1) four of its adjacent bulk elements were marked as being outside a refinement region, and (2) two of the facets emanating from it are labeled with values greater than that of any other facets on adjacent elements (lines 9–10 in Algorithm 2). Nodes of boundary elements are handled similarly.

Once the nodes are marked for coarsening, a kernel call is used to update the adjacent elements' reference element by choosing one of the adjacent elements that will merge into the coarser element (line 11 in Algorithm 2). The kernel launches one thread per node and traverses the node's incident elements The reference element is updated using atomic operations to avoid concurrency in updating the elements. Finally, the element is coarsened, which involves updating the adjacent element to the node in the node table, the nodes defining the adjacent elements, the elements opposite to the corner nodes of the adjacent elements in the element table, and the level of refinement of the adjacent elements (line 12 in Algorithm 2).

**Algorithm 2** Kernel based algorithm to perform adaptive mesh coarsening

```
1  CoarsenCUDA ()
2    MarkCoarsenRegionCells <<<nCells>>>
3    do {
4      MarkElements <<<nElems>>>
5      nMarkedElems = ScanMarkedElements <<<nElems>>>
6      if nMarkedElems == 0 then {
7        break
8      }
9      MarkNodes <<<nNodes>>>
10     nMarkedNodes = ScanMarkedNodes <<<nNodes>>>
11     UpdateReferenceTable <<<nMarkedNodes>>>
12     Coarsen <<<nMarkedNodes>>>
13   } while there are marked elements
14 end CoarsenCUDA
```

## 4 Numerical investigations

The adaptive mesh refinement and coarsening scheme implemented here is applicable for many types of fracture problems; however, the benefits of the approach are most realized for problems dominated by few cracks. Consider the contrary example of pervasive fracture problems [25]. All or most of domain needs high levels of mesh refinement to capture the fracture behavior, thus an adaptive refinement scheme would not have any effect. The following investigations are intended to examine implications of the GPU implementation on the physics of the problem, push the limits of the GPU to determine the maximum problem size that can be simulated, and to explore the response of systems through parametric studies in an efficient manner.

### 4.1 Reduced-scale micro-branching specimen

We verify the implementation of the adaptive scheme on the GPU through a series of numerical investigations on a well-known micro-branching problem. This model problem is inspired by the experimental work of [26] and has been simulated by many authors [10, 27–29]. Similar to the previous investigations, we utilize a reduced-scale model for direct comparison purposes. Later we will address the issue of the full-scale model. The problem features few major cracks, which makes it a good candidate for the adaptive scheme, and several minor cracks that results in a complex fracture pattern. The simple geometry and loading conditions are shown in Fig. 9. For the reduced-scale model, the geometry is given by $x = 16$ mm, $y = 4$ mm, the applied strain is 0.015 and we use the material parameters suggested in [27]. Due to the reduction of the model size and known issues related to representing an experimental system on a numerical model, we adopt the following: Young's Modulus of 3.24e9 Pa, density of 1190 kg/m$^3$, and a Poisson ratio of 0.3 for the bulk elements and a fracture energy of 352.3 N/m and cohesive strength of 129.6e6 Pa for the cohesive elements. The shape of the softening curve is linear, as given by the PPR shape parameter of 2 in each opening direction. Unloading is

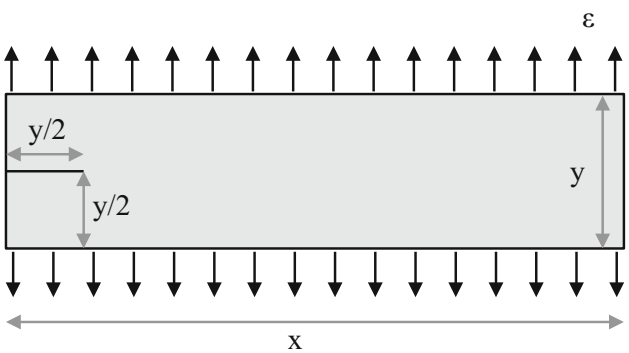



**Fig. 9** Micro-branching problem geometry and loading conditions

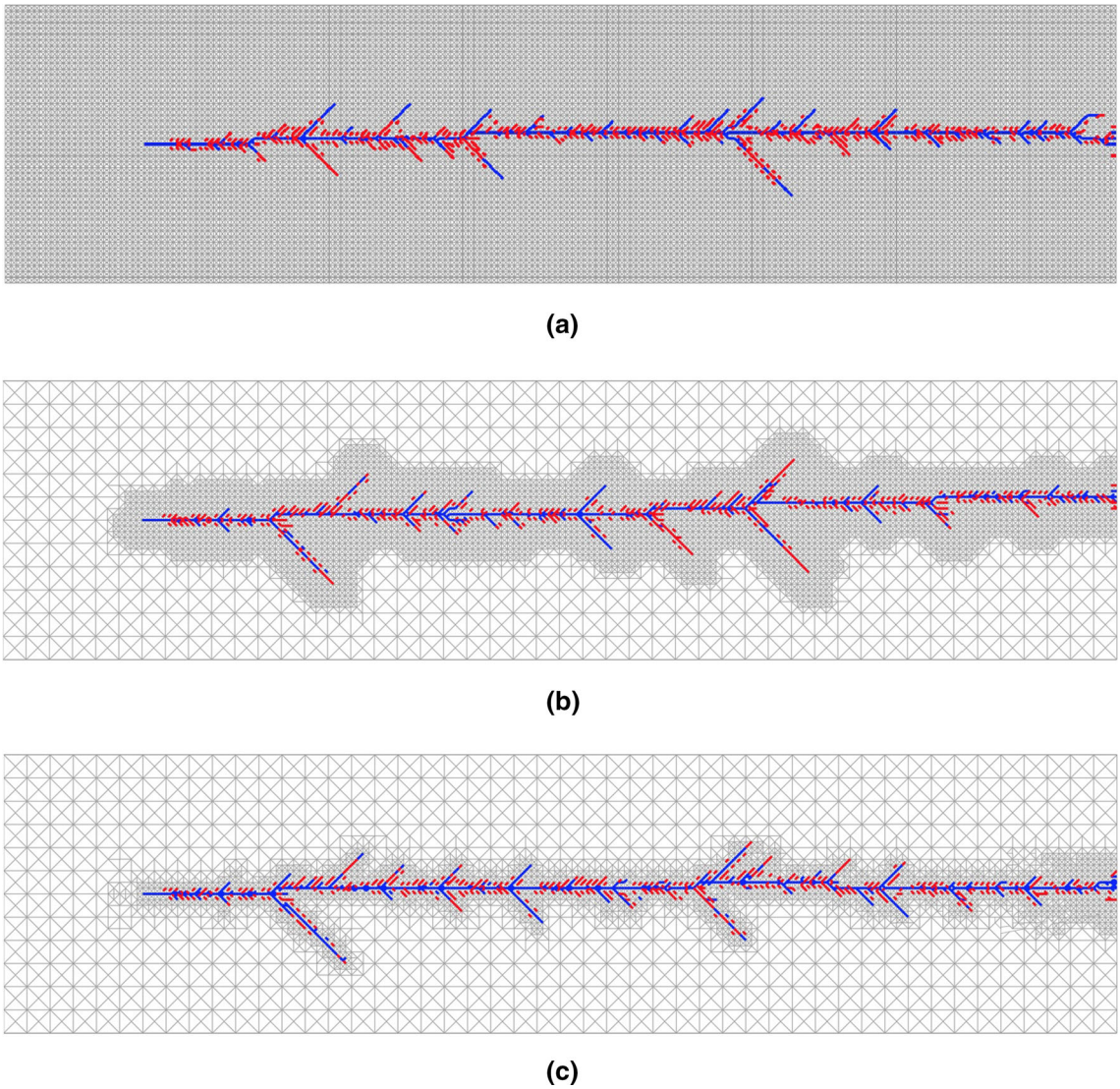



**Fig. 10** Final crack pattern for the reduced-scale micro-branching problem for **a** uniform mesh, **b** AMR-enabled mesh, **c** AMR+C-enabled mesh. Cohesive elements opened greater than 10 % of the normal or tangential critical opening distance are shown in *blue*, other cohesive elements are shown in *red* (color figure online)

assumed to occur linearly back to the origin, i.e., permanent deformation is not sustained. To prevent interpenetration of materials, a penalty stiffness is applied if cohesive tractions become negative.

First, it is useful to compare the results using AMR and AMR+C to that of an equivalent uniformly refined mesh. For the reduced-scale model, the uniform mesh comprised 192×48 4k patches, or 36,864 T6 elements. The AMR- and AMR+C-enabled meshes are initially discretized into 48×12 4k patches, or 2,304 T6 elements, then adaptively refined to a level 4 in the region of the crack tips. Elements are removed in the AMR+C case in regions far away from the crack tips when the root mean error in the strain on a patch of elements falls below the user-defined threshold of 0.01.

The final crack patterns for each case are shown in Fig. 10. The finite element meshes are visible and various levels of refinement are clear in the AMR and AMR+C cases. Cohesive elements that are open greater than a certain threshold of the critical opening distance in either the normal or tangential direction are plotted in blue. Cohesive elements that are inserted but not open greater than the threshold are plotted in red. The threshold by which a cohesive element is considered open is important when quantifying the fracture pattern. For visualization purposes, we show the fracture pattern with the relatively low threshold of 10 %, but in the quantifications reported later in this section we also examine a higher threshold.

Table 1 shows the final number of elements and nodes (after adaptivity and insertion of cohesive elements) and

**Table 1** Comparison of final quantities between Uniform, AMR and AMR+C simulations

| Tol | Mesh type | Elements initial/final | Nodes initial/final | Crack tip velocity (m/s) | Total crack length (m) | Num branches | Avg. branch length |
|---|---|---|---|---|---|---|---|
| 0.1 | Uniform | 36,864/36,864 | 74,257/76,268 | 777.5 | 0.034 | 2 | 2.2e−4 |
| 0.1 | AMR* | 2,634/13,277 | 5,411/28,506 | 754.3 | 0.036 | 1 | 5.2e−4 |
| 0.1 | AMR+C* | 2,634/8,303 | 5,411/18,296 | 755.5 | 0.039 | 2 | 5.1e−4 |
| 0.75 | Uniform | 36,864/36,864 | 74,257/76,268 | 777.5 | 0.019 | 14 | 4.6e−4 |
| 0.75 | AMR* | 2,634/13,277 | 5,411/28,506 | 754.3 | 0.021 | 19 | 4.1e−4 m |
| 0.75 | AMR+C* | 2,634/8,303 | 5,411/18,296 | 755.5 | 0.021 | 20 | 4.7e−4 m |

* The AMR and AMR+C quantities are averaged over 20 simulations

quantitative differences between the simulations, namely the crack tip velocity, total crack length, and number of branches off the main crack. The crack tip velocity is computed by performing a linear regression of the crack tip versus time, where the crack tip is defined at the right-most non-duplicated node of a cohesive element. The crack tip velocity is quite stable throughout the simulation, so the linear regression agrees well with the raw data. Notice that the crack tip velocity is the same for both tolerances, because by our definition the crack tip for the purposes of the velocity calculation is independent of the amount of element opening. The total crack length is total distance covered by all of the cohesive elements open greater than a certain fraction of the critical normal or tangential opening length (denoted Tol in Table 1). We see good quantitative agreement between the uniform, AMR and AMR+C cases in terms of the crack tip velocity and total crack length.

The number and length of branches was post-processed using a simple algorithm performed on the final fracture pattern. Starting from the notch tip node, the main branch is detected by traversing cohesive elements using the adjacency information stored in the data structure. The main branch consists of the path of full open elements that reach the right end of the specimen. Once the main crack is detected, the secondary branches are found by again traversing the main crack. Every point where the crack branches, the path is followed using adjacent information until it terminates. Primary branches are those with the longest length emanating from the main branch. A shorter branch emanating from a primary branch is denoted as a secondary branch, see Fig. 11. This algorithm excludes cohesive elements that are not connected to the main crack. We chose this approach so that the process of counting branches would be controlled and consistent between specimens. The procedure of quantifying number and length of branches is too subjective to be evaluated by a visual inspection. There is quite a difference in the number and average length of branches and between the uniform, AMR and AMR+C cases. We report this information,

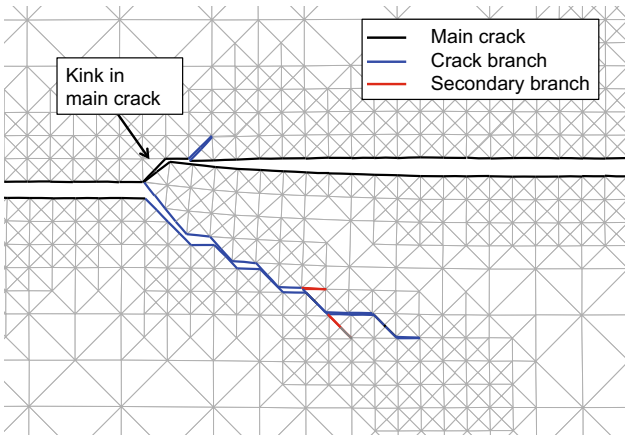



**Fig. 11** Details of crack branching including kink in the main crack, crack branches, and secondary branches

because when visualizing a fracture pattern, one often focuses on the number and length of branches; however, this data can be misleading, because it is not an accurate representation of the crack velocity or the total crack length, which includes the main crack and the many kinks it may have, as illustrated in Fig. 11. Instead, we choose total crack length as the important quantity on which to compare the cases because it is directly related to the total energy released during the fracture process, and this is quantity that should remain similar between different numerical representations of the same process.

The quantities shown in Table 1 for the AMR and AMR+C cases are average over 20 simulations. This is because the massively parallel nature of the adaptivity in GPU implementation introduces some variation into the fracture simulation. New elements resulting from mesh refinement are inserted to the mesh in a random order, so from one simulation to the next the order in which new bulk elements are inserted will be different. The impact on the simulation is realized when nodal quantities are computed. Recall that we avoid graph coloring and concurrency issues, we traverse nodes and gather necessary data from

elements as opposed to traversing elements and writing to nodes. When we gather information onto a node from a neighboring element, the random order in which the elements were inserted affects the order in which we visit the elements adjacent to a node. Since we only have a certain level of accuracy in floating point operations, we cannot, in general, guarantee $A + B \neq B + A$. So, when computing quantities on a node 1, we may pull data from elements 100, 101, 102 and 103 in one simulation, and from elements 101, 100, 103, 102 in a second simulation, which is not equal in a precise sense. These variations accumulate over all of the computations, nodes, time steps, etc., and the result is a variation in final fracture patterns.

It should be noted that we also examined an implementation in which the order of element/nodal computations is prescribed and the same from one simulation to another and verified that the results are identical. This does not imply that the implementation with no variation is correct and the one with variation is incorrect. The same randomness is present in the consistent implementation and if we chose to access the elements in a different order, we would have a similar effect as the implementation with variation. We chose to pursue the implementation that introduces randomness, because it is much more computationally efficient.

Using the reduced-scale micro-branching problem, we investigate the impact that the randomness has on the final result. We performed the simulation 20 times on each of an AMR- and AMR+C-enabled mesh, then quantified the variation in fracture patterns in Table 2.

As before, we notice a large difference in the number of crack branches especially for the low crack tolerance, which emphasizes the point that number of branches is not an ideal measure to by which to compare fracture patterns resulting from the same process, e.g., same geometry, material properties, and loading conditions. The variance on the total crack length is quite low, suggesting that the variation caused by the numerical implementation is low. The crack tip velocity also shows low variation amongst the 20 iterations, for the AMR and AMR+C cases the crack tip velocities are $754.3 \pm 9.8$ and $755.6 \pm 10.1$ m/s, respectively. Additionally, the total energy released during the fracture process is quite comparable, $75.0 \pm 2.6$ and $77.0 \pm 2.0$ N/m for the AMR and AMR+C cases, respectively. The total energy released considers all cohesive elements, regardless of their amount of opening, thus this quantity is also independent of the threshold.

We observe some other additional fracture pattern characteristics. The branch spacing is fairly regular among all simulations and the main cracks kinks about 3–6 times during the simulation. Most of the branches are 1–3 elements in length, then the frequency drops significantly, as shown in Fig. 12. Secondary branches occurred in about half of the adaptive. Thus we concluded that the variation caused by the massively parallel GPU implementation is not significant.

The variation caused by the GPU could alternatively be viewed as a way to induce randomness into the numerical model, which in other similar studies was achieved by perturbing a structured mesh [28] or by using a completely random mesh [29]. The adaptive GPU implementation allows the use of structured mesh with variability that would be expected of a random mesh.

Finally, we compare the computational time of the proposed scheme with other platforms (serial CPU, single GPU) and different types of implementation (adaptive vs. non-adaptive). The serial CPU versions were run on a 1.3 GHz Intel Core i5 processor and the GPU version implemented here was done on a GeForce GTX TITAN with 2688 CUDA cores and 6Gb memory. To the best of the authors' knowledge there has been no other implementation of an adaptive mesh refinement and coarsening scheme done on a parallel platform. Table 3 shows the run times and speedups over the serial implementation without adaptivity.

**Table 2** Variation in crack tip velocity, energy released, and occurrence of branching for 20 simulations of each the AMR- and AMR+C-enabled meshes

| | Tol = 0.1 | | Tol = 0.75 | |
|---|---|---|---|---|
| | AMR | AMR+C | AMR | AMR+C |
| Total crack length | | | | |
| Mean | 0.036 m | 0.039 m | 0.021 m | 0.021 m |
| Standard deviation | 8.8e−4 m | 6.8e−4 m | 9.2e−4 m | 9.4e−4 m |
| Number of branches | | | | |
| Mean | 17 | 19 | 1 | 2 |
| Standard deviation | 3 | 4 | 1 | 1 |
| Average branch length | | | | |
| Mean | 5.2e−4 m | 5.1e−4 m | 4.1e−4m | 4.7e−4m |
| Standard deviation | 4.2e−4 m | 4.6e−4 m | 4.2e−4 m | 6.1e−4 m |

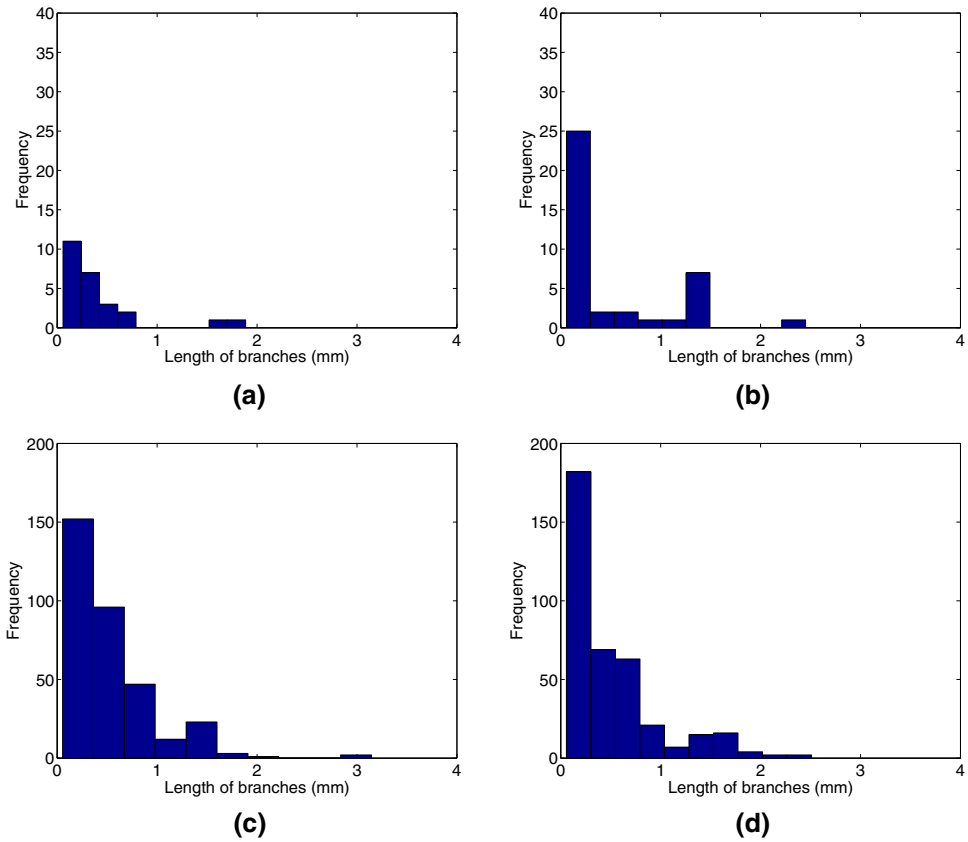



**Fig. 12** Histogram of branch lengths over 20 simulations for the **a** AMR-enabled meshes with an open crack tolerance of 75 % of critical normal opening, **b** AMR+C-enabled meshes with an open crack tolerance of 75 % of critical normal opening, **c** AMR-enabled meshes with an open crack tolerance of 10 % of critical normal opening and **d** AMR+C-enabled meshes with an open crack tolerance of 10 % of critical normal opening

**Table 3** Comparison of execution time of the reduced-scale micro-branching problem on different platforms (the speedup factor is shown with respect to the no adaptivity case on the serial CPU)

| Platform | Implementation | Execution time (s) | Speedup |
|---|---|---|---|
| Serial CPU | No adaptivity | 1196 | – |
| Serial CPU | AMR | 83 | 14 times |
| Serial CPU | AMR+C | 57 | 21 times |
| Single GPU | No adaptivity | 12 | 100 times |
| Single GPU | AMR | 18 | 66 times |
| Single GPU | AMR+C | 20 | 60 times |

Of course, the GPU is much faster than the serial CPU, thus adaptivity also performs faster on the GPU than the CPU. It is interesting to note that the cases of adaptivity on the GPU actually take longer than the uniform case. This is because for this small problem, the percentage of time spent on updates related to adaptive mesh refinement and coarsening on the GPU is greater than that spent on the finite element calculations. When the problem is larger on the GPU, then we begin to see a difference in execution time between the uniform and adaptive simulations. More important, however, is that the size of the problem is severely limited for the uniform case on the GPU; this limitation is alleviated by adaptivity, which makes large problems feasible because we store much less information than we would on a uniform mesh. So, we may not achieve a large speedup between the uniform and adaptive cases on the GPU, but adaptivity gives the ability to examine problems that we could not be able to simulate otherwise.

### 4.2 Full-scale micro-branching specimen

Next, we are interested in comparing the fracture pattern from the reduced-scale model with that of the actual experimental setup proposed in [26]. The full-scale problem size has dimensions $50 \times 200$ mm (12.5 times larger than the reduced-scale case from the previous section). Previous numerical simulations of this work using the inter-element cohesive zone model have only simulated reduced-scale problems due to limitations of computation resources and sophisticated algorithms. The adaptivity algorithm implemented on the GPU architecture makes simulation of this full-scale problem possible. We should note, that even with the GPU and adaptive mesh

refinement, computational times for the following simulations were on the order of 3 h.

In scaling up the problem, not only does the geometry of the specimen change, but also the applied load and material properties. For the full-scale model, the goal was to keep the numerical representation as close to the experiment. Thus specimen dimensions are those of the experiment, and the material properties are those of PMMA, the material used in the experiment. The Young's Modulus is 3.24e9 Pa, the density is 1190 kg/m$^3$, and the Poisson ratio is 0.3 for the bulk elements, while and a fracture energy of 352.3 N/m and cohesive strength of 62.1e6 Pa is used for the cohesive elements. As before, linear softening, linear unloading back to the origin, and a penalty stiffness to prevent interpenetration are utilized. We examined a range of externally applied loads: a low strain of 0.003, mid strain of 0.004, and a high strain of 0.005, which are similar to the loads applied in the experiment.

The difference between the full-scale and the reduced-scale model is the applied load and the cohesive strength. For the reduced-scale model, the loading was increased such that the strain energy per unit length felt by the specimen would match that of the experiment. The adjustment of the material properties for the reduced-scale model, namely the cohesive strength, is not as straightforward as has been demonstrated by other authors [27, 30]. A cohesive strength that is too large means that fracture never initiates, while a low strength results in the insertion of an excessive number of cohesive elements, which is not physically realistic. Thus, we used the value recommended in [31] and [10]. However, for the full-scale problem, we do not adjust the material properties, and use the experimentally obtained cohesive strength of PMMA.

The model is initially discretized with 300 $\times$ 75 4k mesh patches, or 90,000 elements. We use the AMR to reduce the element size at the notch tip by 4 times (e.g., the largest elements have a maximum length of 0.67 mm and the smallest elements have a maximum length of 0.167 mm). Note that a uniform mesh of comparable size would contain 1,440,000 elements, which is well beyond the size capacity of the GPU, thus adaptivity is essential.

The selection of element size was based on the results of a parametric study of element size and the limitations of the GPU storage capabilities. The full study is omitted here for brevity, but the results are summarized. Essentially, we selected the highest level of refinement to be as small as possible while not exceeding the capacity of the GPU. An overall coarser mesh where the ratio between the size of the coarsest and finest elements are the same as those chosen for this study was not sufficiently fine for the crack to initiate. We also investigated using coarser elements in the far field such that the difference between the courts and finest elements was greater than what is presented here. In this case, we found that the cracks were not initiating because the far-field discretization was too coarse to transfer the strain from the load application points to the crack tip. Finally, we investigated finer levels of discretization. We found that while it performed well initially, the simulation could not be complete because the total numbers of elements exceeded that which we could store on the GPU as cracks propagated. At this higher level of refinement, the characteristics of the initial fracture pattern were similar to that of the coarser mesh, thus giving us confidence that the level selected is adequate.

The fracture patterns for three different strains are shown in Fig. 13. Here we plot cohesive elements that have opened more the 75 % of the critical opening distance. The numerical results obtained here agree well with those shown in the original experiment [26]. At lower strains the fracture surface is smoother and features one predominate crack; however, the crack arrested before it reached the end of the domain. As the load increases, branches appear and the fractured surface becomes rougher. Finally, at the highest strain, many branches are present and are increased in length.

As a proof of concept for the computational gains of the AMR+C, we also included coarsening for the low strain case. The computation time for the AMR was 4.63 and 3.65 h for the AMR+C. Of course, we cannot compare the computational time to the uniform case because the problem would be too large to execute on the GPU; however, we clearly see that when the model is sufficiently large, the AMR+C improves computational efficiency, by 21 % in this case.

The velocities of the three cases also increase with increased applied strain. The average velocity for the low, mid, and high strain cases are 611, 625, and 706 m/s, respectively, all of which are well below the Rayleigh wave speed of the material, as expected. It is important to note that these numeric velocity values may have been impacted by the small deformation assumption. Generally, dynamic crack tip velocities in small deformation formulations are higher than in finite deformation cases because less energy is dissipated. See Sect. 3.1 for the explanation of the use of small deformation and the impacts on GPU memory availability. The velocity versus time plots for each case are shown in Fig. 14. Clearly there is overlap in the velocity versus time plots; however, the general trend is evident. There is a drop in the velocity of the high strain case around 1/3 of the way through the simulation, which corresponds to the growth of three branches simultane-

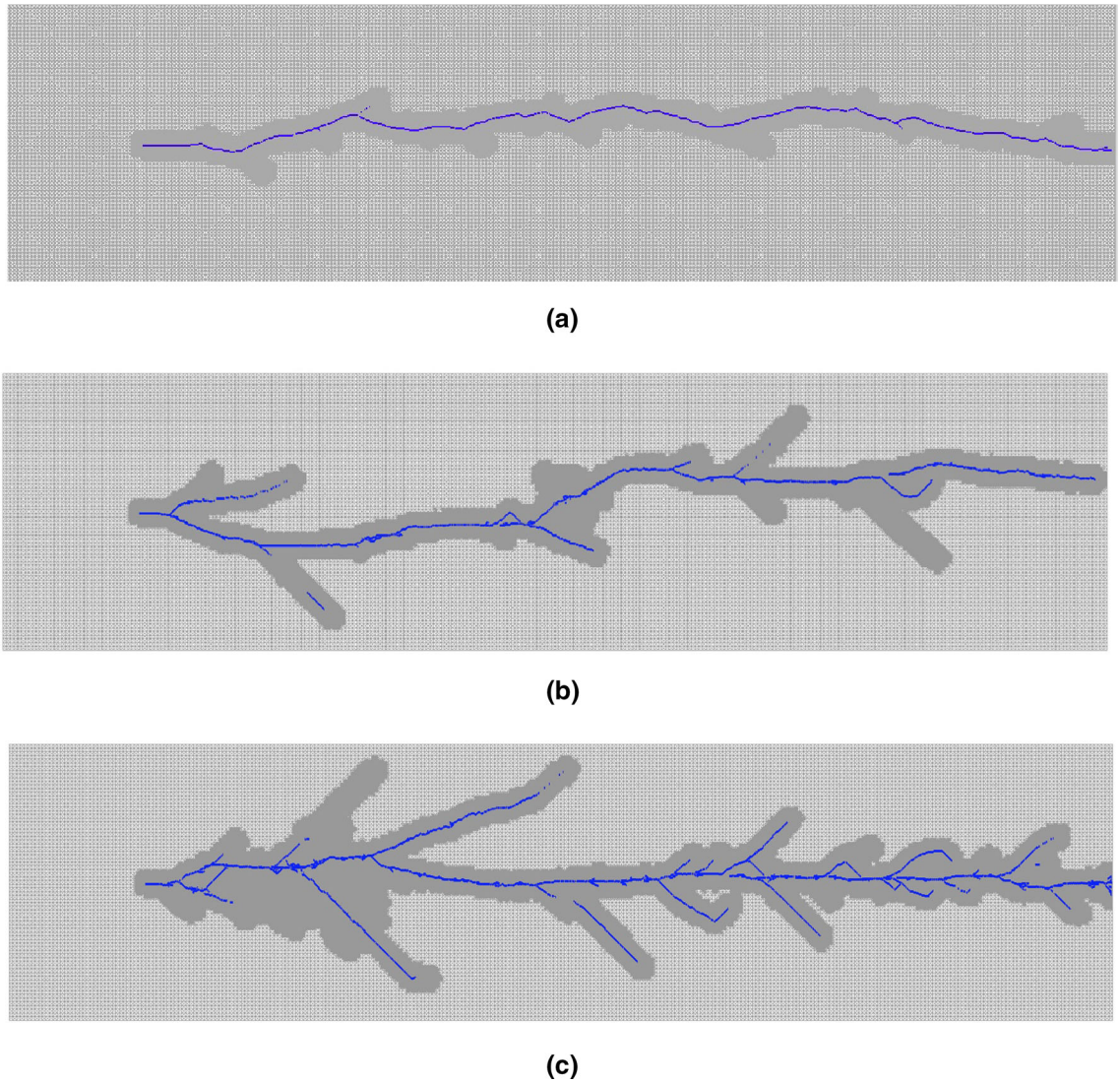



**Fig. 13** Final fracture patterns for full-scale micro-branching problem with an externally applied strain of **a** 0.003, **b** 0.004, and **c** 0.005

ously. Then, once the branches arrest and the main crack progresses again at the higher velocity.

The details of the crack pattern and the adaptive mesh refinement scheme are shown in Fig. 15 for the low strain case. Elements that are open less than 75 % of the critical opening distance are shown in the zoom-in view in red. Notice that in relation to the crack branch, the branches comprised partially open elements are quite small. The details of the refinement scheme are clear, elements within the user-defined radius of a crack tip are refined. The radius of refinement is sufficiently large such that new cohesive elements will be inserted within the bounds of the refined elements.

When comparing the reduced-scale model results and the full-scale model results, we notice some qualitative similarities, but the details are not evident in the smaller model. Thus, whenever possible, it is recommended to use a numerical model that closely resembles the actual experiment. However, in many cases, that is not entirely feasible due to lack of access to powerful and sophisticated computational resources.

## 5 Concluding remarks

Investigation of adaptive refinement and coarsening schemes on the structured 4k mesh for dynamic fracture simulation on the massively parallel GPU architecture reveals insight into intricacies of the numerical simulation. First, a specialized data structure and new approach to performing finite element calculations in parallel was detailed. The race condition and expensive graph coloring

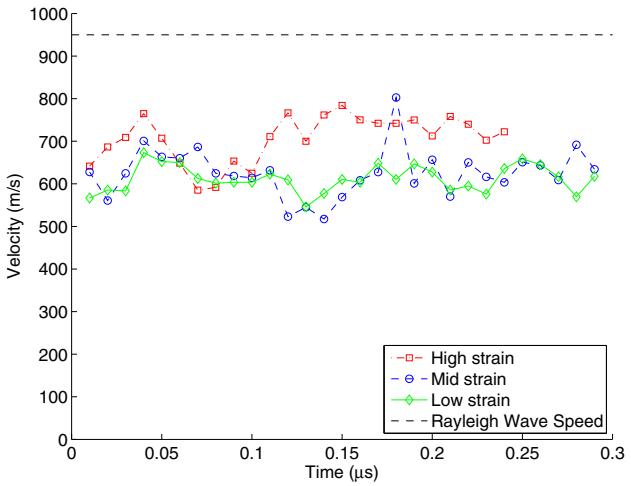



**Fig. 14** Magnitude of the crack tip velocity versus time for the high, mid, and low strain loading. The average velocity increases with increasing strain, but remains well below the Rayleigh wave speed as expected

algorithms are avoided by performing finite element calculations on a nodal basis. Nodal quantities are gathered by launching threads per nodes and accumulating element contributions rather than by launching threads per elements. Using the assumption that areas near crack tips need to be the most refined and a strain criterion to determine where elements can be coarsened, we adaptively change the mesh resolution during the simulation. We detail the parallel algorithms to systematically change the topology of the mesh.

The variations that normally occur during floating point operations are not usually apparent in serial or even parallel fracture simulations on structured meshes. This is because the order in which operations are performed, and thus the accumulation of variation, is usually the same from one simulation to the next. However, in the present implementation, new elements and nodes are inserted in a random order, meaning that the quantities added to the node are not done so in the same order from one simulation to another. While not incorrect, the result is that fracture patterns are different from one simulation to the next. To demonstrate the validity of the approach given the very different appearance of the fracture pattern, we quantified the crack patterns through several parameters and showed that those that are physically based agree well between simulations. Interestingly, the parallel approach adds some randomness into the finite element simulation on the structured mesh in a similar way as a would be expected from a random mesh.

Lastly, thanks to a data structure and adaptive mesh modification scheme developed specially for the GPU architecture, we are able to represent much larger finite element meshes than without adaptivity. With the large-scale simulation of the micro-branching problem, we are able to make more direct comparisons to the original experiment and find excellent agreement with those results.

A natural extension of this work would be include all three dimensions; however, on a single GPU the problem size would be quite limited, which is not well suited for three-dimensional finite element applications. Thus,

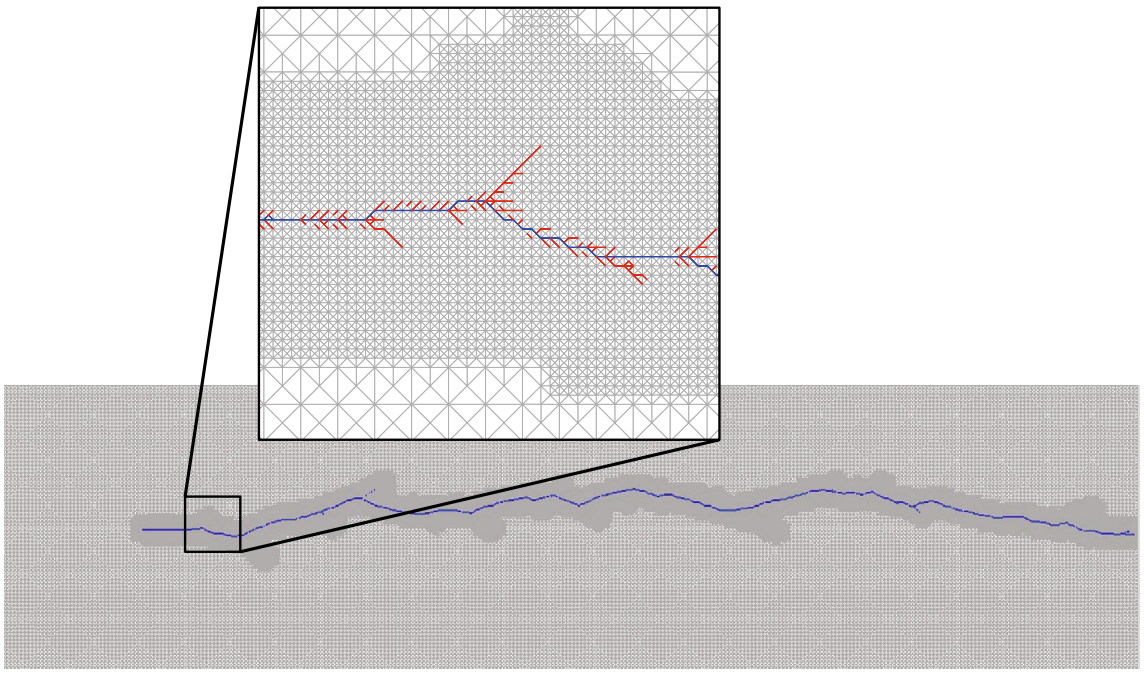


**Fig. 15** Detailed view of fracture pattern for the full-scale micro-branching problem with an externally applied strain of 0.003

current development is on distributed computing where different parts of the model would be simulated on different GPUs.

**Acknowledgments** Andrei Alhadeff and Waldemar Celes thank the support provided by the Tecgraf Institute at PUC-Rio, which is mainly funded by the Brazilian oil company, Petrobras. They also thank the Brazilian National Council for Scientific and Technological Development (CNPq) for the financial support to conduct this research. Sofie E. Leon and Glaucio H. Paulino gratefully acknowledge the support of the Philanthropic Education Organization (PEO) Scholars Award, and the Raymond Allen Jones Chair endowment at the Georgia Institute of Technology, respectively. They also acknowledge the support of the National Science Foundation (NSF) through grants CMMI #1321661 and CMMI #1437535.

## References

1. Alhadeff A, Celes W, Paulino GH (2015) Mapping cohesive fracture and fragmentation simulations to GPUs. Int J Numer Methods Eng 103:859–893. doi:10.1002/nme.4842
2. Kirk DB, Wen-mei WH (2010) Programming massively parallel processors: a hands-on approach. Morgan Kaufmann, San Francisco
3. Brodtkorb AR, Hagen TR, Sætra ML (2013) Graphics processing unit (GPU) programming strategies and trends in GPU computing. J Parallel Distrib Comput 73(1):4–13
4. Dziekonski A, Sypek P, Lamecki A, Mrozowski M (2012) Generation of large finite-element matrices on multiple graphics processors. Int J Numer Methods Eng 94(2):204–220
5. Cecka C, Lew AJ, Darve E (2010) Assembly of finite element methods on graphics processors. Int J Numer Methods Eng 85(5):640–669
6. Wang L, Zhang YS, Zhu B, Xu C, Tian XW, Wang C, Mo JH, Li J (2012) GPU accelerated parallel cholesky factorization. Appl Mech Mater 148–149:1370–1373
7. Dooley I, Mangala S, Kale L, Geubelle P (2008) Parallel simulations of dynamic fracture using extrinsic cohesive elements. J Sci Comput 39(1):144–165
8. Lawlor OS, Chakravorty S, Wilmarth TL, Choudhury N, Dooley I, Zheng G, Kalé LV (2006) ParFUM: a parallel framework for unstructured meshes for scalable dynamic physics applications. Eng Comput 22(3–4):215–235
9. Radovitzky R, Seagraves A, Tupek M, Noels L (2011) A scalable 3D fracture and fragmentation algorithm based on a hybrid, discontinuous Galerkin, cohesive element method. Comput Methods Appl Mech Eng 200(1–4):326–344
10. Espinha R, Park K, Paulino,GH, Celes W (2013) Scalable parallel dynamic fracture simulation using an extrinsic cohesive zone model. Comput Methods Appl Mech Eng 266(C):144–161
11. Park S, Shin H (2012) Efficient generation of adaptive Cartesian mesh for computational fluid dynamics using GPU. Int J Numer Methods Fluids 70(11):1393–1404
12. Dugdale D (1960) Yielding of steel sheets containing slits. J Mech Phys Solids 8(2):100–104
13. Barenblatt GI (1962) The mathematical theory of equilibrium cracks in brittle fracture. Adv Appl Mech 7(55–129):104
14. Park K, Paulino GH, Roesler JR (2009) A unified potential-based cohesive model of mixed-mode fracture. J Mech Phys Solids 57(6):891–908
15. Park K, Paulino GH (2011) Cohesive zone models: a critical review of traction-separation relationships across fracture surfaces. Appl Mech Rev 64(6):060802
16. Camacho G, Ortiz M (1996) Computational modelling of impact damage in brittle materials. Int J Solids Struct 33(20–22): 2899–2938
17. Newmark NM (1959) A method of computation for structural dynamics. J Eng Mech Div 85(7):67–94
18. Boyalakuntla DS, Murthy JY (2002) Hierarchical compact models for simulation of electronic chip packages. Compon Packag Technol IEEE Trans 25(2):192–203
19. Ducros F, Ferrand V, Nicoud F, Weber C, Darracq D, Gacherieu C, Poinsot T (1999) Large-eddy simulation of the shock/turbulence interaction. J Comput Phys 152(2):517–549
20. Fryxell B, Olson K, Ricker P, Timmes FX, Zingale M, Lamb DQ, MacNeice P, Rosner R, Truran JW, Tufo H (2000) FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. Astrophys J Suppl Ser 131(1):273
21. Celes W, Paulino GH, Espinha R (2005) A compact adjacency-based topological data structure for finite element mesh representation. Int J Numer Methods Eng 64(11):1529–1556
22. Welsh DJ, Powell MB (1967) An upper bound for the chromatic number of a graph and its application to timetabling problems. Comput J 10(1):85–86
23. Park K, Paulino GH, Celes W, Espinha R (2012) Adaptive mesh refinement and coarsening for cohesive zone modeling of dynamic fracture. Int J Numer Methods Eng 92(1):1–35
24. Velho L, Gomes J (2000) Variable Resolution 4-k Meshes: Concepts and Applications. Comput Graph Forum 19(4):195–212
25. Bishop JE (2009) Simulating the pervasive fracture of materials and structures using randomly close packed Voronoi tessellations. Comput Mech 44(4):455–471
26. Sharon E, Fineberg J (1996) Microbranching instability and the dynamic fracture of brittle materials. Phys Rev B Condens Matter Mater Phys 54(10):7128–7139
27. Zhang ZJ, Paulino GH, Celes W (2007) Extrinsic cohesive modelling of dynamic fracture and microbranching instability in brittle materials. Int J Numer Methods Eng 72(8):1017–1048
28. Paulino GH, Park K, Celes W, Espinha R (2010) Adaptive dynamic cohesive fracture simulation using nodal perturbation and edge-swap operators. Int J Numer Methods Eng 84(11):1303–1343
29. Spring DW, LeonSE, Paulino GH (2014) Unstructured polygonal meshes with adaptive refinement for the numerical simulation of dynamic cohesive fracture 189(1):33–57
30. Miller O, Freund LB, Needleman A (1999) Energy dissipation in dynamic fracture of brittle materials. Model Sim Mater Sci Eng 7(4):573
31. Zhang Z (2007) Extrinsic cohesive modeling of dynamic fracture and microbranching instability using a topological data structure, Ph.D. thesis