4
Massively Parallel Adaptive Mesh Refinement and Coarsening for 2D Dynamic Fracture Simulations

In this chapter, 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 [45]. 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 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.

In this chapter, we present a massively parallel adaptive finite element analysis for dynamic fracture simulations. Adaptiviness happens in two levels. First, cohesive interface elements are inserted along bulk elements boundaries where and when needed, according to the analysis fracture criterion. Second, and more challenging, 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 chapter is organized as follows. Section 4.1 describes our data structure and explain mesh modification algorithms on the GPU, such as mesh refinement and coarsening and insertion of cohesive elements. Section 4.2 presents an overview of the parallel simulation, much like the 2D simulation procedure, with a few differences, including mesh refinement and coarsening steps included within. Section 4.3 provides the numerical results.

4.1 Adaptive mesh modification on Graphical Processing Units

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. However, mesh adaptivity and hierarchical schemes are also utilized in other fields such as modeling electronic chip packages [64], large-eddy simulations [65], and astrophysical thermonuclear flash [66], 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. First, we present the data structure followed by the methodology for computing finite element calculations on the GPU. The next subsections detail the algorithms for adaptive insertion of cohesive elements and adaptive mesh refinement and coarsening, respectively.

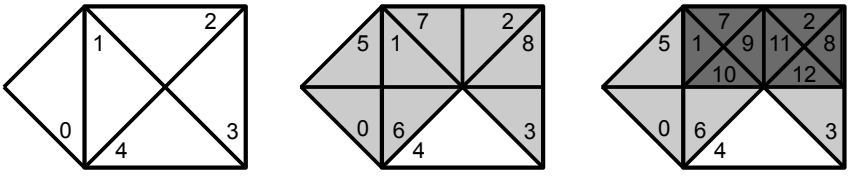
4.1.1 Data structure for 4k adaptive finite element mesh representation

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

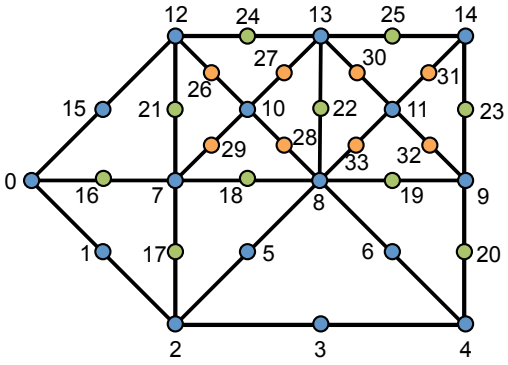
Several data structures have been devised including those for dynamic fracture with adaptive insertion of cohesive elements and adaptive topological operators on a serial CPU platform [60], dynamic fracture with adaptive insertion of cohesive elements on a parallel CPU platform [67], and dynamic fracture with adaptive insertion of cohesive elements on a massively parallel GPU platform [45]. 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

does 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 an evolving 4k structured finite element mesh. 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 Figure 4.1; we will refer back to this figure several times in the next few section to aid discussion.



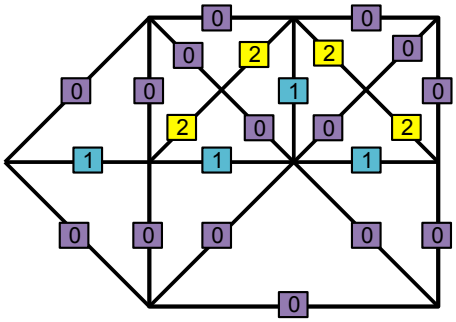
(a)



(b)

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

(d)



(c)

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

(e)

Figure 4.1 – Schematic of GPU data structure for adaptive 4k mesh (a) progression of mesh refinement and element labeling, (b) node numbering on refined mesh, (c) Edge 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 Edge labels.

The sample coarse mesh is partially refined in three steps, as shown in Figure 4.1(a) and the corresponding nodal number of the final mesh is shown in

Figure 4.1(b). All of the data necessary to store this adaptive mesh is contained in the node table (Figure 4.1(d)) and element table (4.1(e)). The node table contains all information related to the nodes of the mesh. The x and y values describe its 2-dimensional spatial position. Another value representing a bulk element identifier (which refers to the bulk element in the element table) is referenced as one of the adjacent elements of that node. Referring to an element implies in referring to its identifier in the element table. The adjacent bulk element is widely used to begin traversing all the adjacent elements from a given node. The element table contains information associated to every bulk element and cohesive element in the mesh. The cohesive elements are stored explicitly as another element in the element table. The table contains pre-allocated space for bulk elements of the initial mesh and bulk and cohesive elements that be will later be inserted along the mesh. A cohesive element is referenced by referring to its identifier in the element table. The bulk elements contain six node identifiers: three corner nodes followed by three midnodes of the quadratic triangular mesh (T6). The corner and midnodes refer to the node ids in the node table. Three adjacent elements opposite to the corner nodes refer to the element ids in the element table. An opposite element to a corner node can also be a cohesive element. A cohesive element contains two adjacent elements (onde for each facet).

As bulk elements are subdivided, they are assigned a new level of refinement; Figure 4.1(a) 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 Figure 4.1(e). 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 Figure 4.1(c) and as a column of the element table in Figure 4.1(e). Because edges are not represented as an explicit entity in our data structure, we store three labels per bulk element (one per edge).

Finally, in order to update faster the mesh adjacency during coarsening, we calculate for which bulk element two finer bulk elements will coarsen to. To accomplish this, we store the ID of the coarse element (parent) from which two elements (children) emerge during refinement from level n to level n + 1; the reference element is stored in the element table shown in Figure 4.1(e).

When coarsening the mesh, elements and nodes are removed from the 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 (node and element) and a two stack counters to reference the top of the stack.

In order to check if elements need refinement in a fast way, we use a two-dimensional grid, where we define the grid cell as a small region that needs refinement. For example, if the midpoint of at least one of the element's facet is contained within a grid cell that needs refinement, that element needs refinement also. This is a faster way to identify an element as being refined or coarsened rather than each element going through all the refinement regions. We store the grid attributes, such as cell dimension, bottom-left corner of the grid, grid size and number of cells in constant memory. We have a counter per cell indicating if the cell is inside or outside a refinement region.

Finally, we need allocate space for other non-topological attributes used in the numerical simulation, such as nodal displacements and velocities, stress, element stiffness matrix, and elastic material properties. Some of them are read and written, while others are read-only attributes. For attributes such as displacements, velocities, stress and cohesive traction that are read and written frequently, use store them in global memory. We use constant memory for attributes that are only read and do not occupy too much memory space. Constant memory is a read-only type of memory but with small space that is suitable for material properties that are common to all entities. Finally, we use texture memory for attributes such as the stiffness matrix that occupy a large memory space but are read-only.

4.1.2 Node and element calculations on GPU

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) are composed of contributions from adjacent elements. Writing

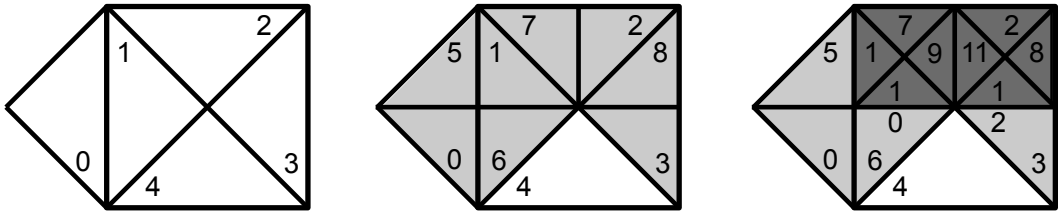


Figure 4.2 – (a) Refinement of elements 1, 2 and 3 from level 0 (white) to level 2 (dark grey)

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 [62], whereby the color groups are visited in a serial fashion, and one thread is launched per element in that color group. Mesh coloring 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 discussed in Section 4.1.1 allows for quick access the elements adjacent to the nodes with one of the bulk elements adjacent to that node. 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 Section 4.3.

4.1.3 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 may be 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 Section 4.1.2, we do not utilize a graph coloring scheme in the adaptive simulation and insertion of cohesive elements. One of the implications of our node-based approach is that we cannot use previous strategies, such as those adopted in [45, 67], 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 Section 4.1.2.

Figure 4.3 illustrates the algorithm for inserting the cohesive elements on a non-colored quadratic triangular mesh (T6). After detecting the fractured bulk element interfaces, we begin the procedure for inserting cohesive elements and duplicating nodes when needed. In the first kernel (with algorithm illustrated by Figure 4.3(a)), 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 number in the element table to be responsible for inserting the cohesive element. The bulk element sweeps its facets inserting the cohesive elements on the fractured facets and adds them to the element table. The corner nodes, however, are not yet duplicated; only the mid-side nodes are. We mark the corner nodes that belong to the cohesive elements so they can be filtered for the traversal in next kernel. Once all of the cohesive elements are inserted in parallel, we begin the second step (Figure 4.3(b)).

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 Figure 4.3(b). By accessing the node's adjacent element using the data structure, we traverse its incident elements to check if the node requires duplicating. If the adjacent element to the first bulk element is not reached, we must duplicate the node. 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. Then the path is followed in the other direction, duplicating nodes and updating incidence starting from when the second cohesive element is reached, as is illustrated in Figure 4.3(b). The procedure is done until the adjacent element to the first cohesive element is reached (Figure 4.3(6b)). The algorithm does not cause issues of concurrent writing because each thread is responsible for duplicating its own node.

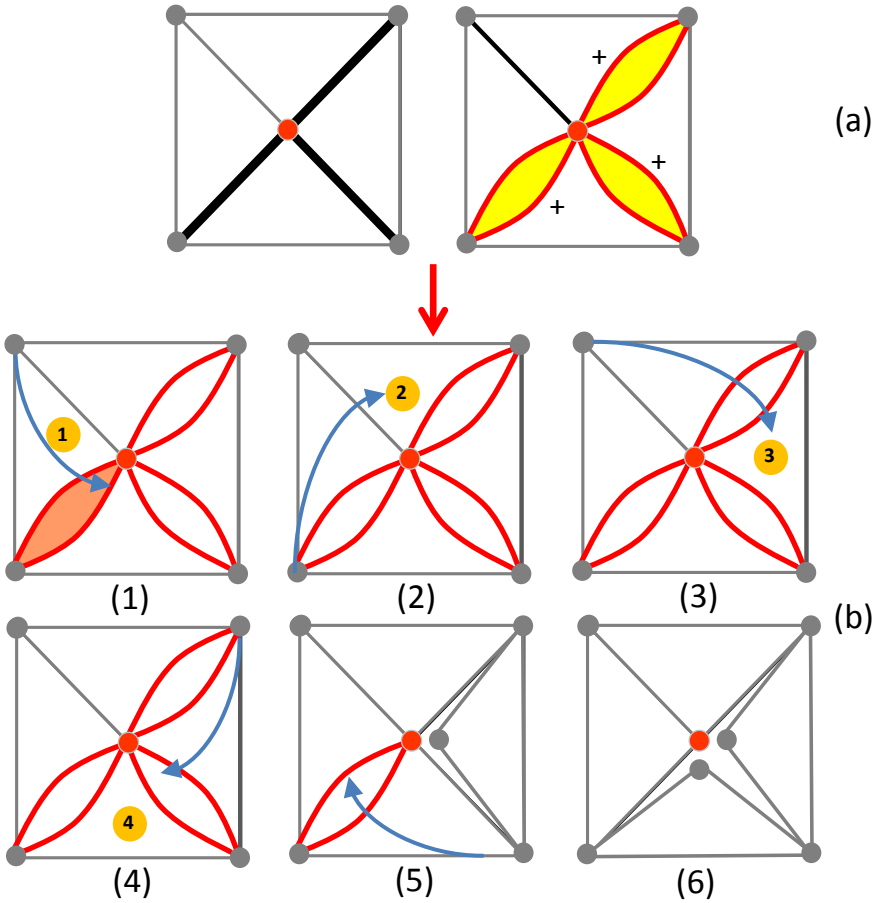


Figure 4.3 – Cohesive elements insertion on non-colored mesh: (a) Cohesive elements are inserted on fractured facets (in black) by launching one thread per bulk element that contains at least one fractured facet. (b) Launching one thread per node and by accessing one of the node's adjacent element, traversal begins in one direction until first cohesive element is reached (1b). Traversal direction changes (2b) and when second coehesive element is reached, the nodes of the bulk elements after the cohesive elements are duplicated (5b). Steps are repeated until adjacent element is reached again (6b).

4.1.4 Preset of refinement and coarsening regions

As new crack tips emerge during the simulation (Figure 4.4(a)), new corresponding refinement regions must be created (Figure 4.4(b)). An element is refined if at least one of its midside node is inside a refinement region. To verify this in an efficient way, we propose the use of a regular grid as an uderlying data structure (Figure 4.4(c)). Each grid cell stores a counter that indicates the number of refinement regions it belongs to. The cell size is chosen as R/K, where R is the refinement region radius and K is a constant. If a cell is inside a refinement region, its counter is incremented by one (Figure 4.4(d)). 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

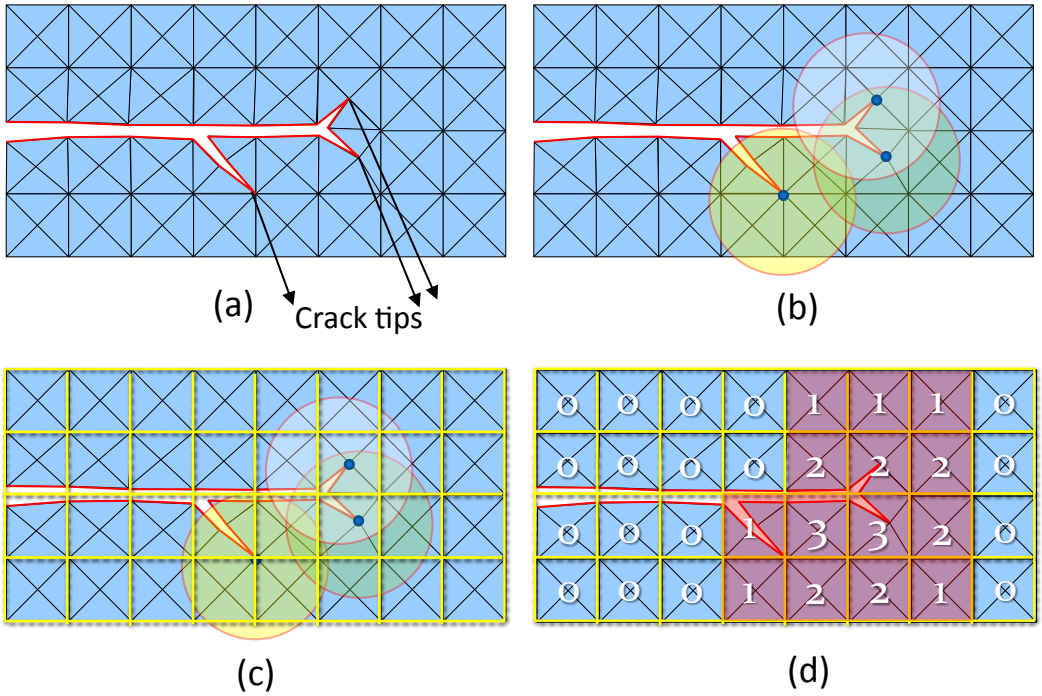


Figure 4.4 – Identifying facets of elements that need refinement given local region cells. (a) Crack tips emerge from the simulation analysis and (b) each define a refinement region radius. (c) The domain is discretized into cells, (d) in which each contain its own counter to account for overlapping regions.

value different from zero. We launch one thread per cell and for each thread we verify if the distance between the center of the cell and the center of each new refinement region (of the current step) is smaller than the refinement radius. If so, we increment the cell counter.

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 is 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.

4.1.5 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 [68] and is briefly reviewed here before describing the GPU implementation.

Multiple crack tips may emerge as the quasi-brittle fracture simulation evolves in time (recall that the definition of a crack tip is the non-duplicated node of a cohesive element). These crack tips are necessary to perform 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. Before proceeding, we will clarify the notation we use in Algorithm 1; `<<< x >>>` indicates is kernel call is being made where x indicates the number of threads launched.

The algorithm loop initiates with first-level elements refinement (line 3 in Algorithm 1). 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). 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 1). This is illustrated in Figure 4.5, 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 done to get the total number of marked elements (line 8 in Algorithm 1).

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

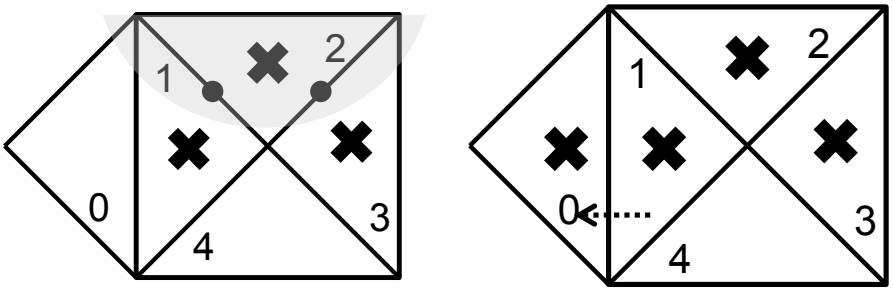


Figure 4.5 – Marked opposite elements (a) elements with at least one node inside the refinement region are marked, (b) elements adjacent a marked element's hypotenuse are also marked.

```
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
```

Algorithm 1 – Kernel based algorithm to perform adaptive mesh refinement

along its longest edge [69]. 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 Figure 4.1(e)), 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). The refinement loop continues by going back to line 3 in Algorithm 1 and moving forward to level of refinement 2 until all elements inside the refinement region reach the level prescribed by the user or there are no more marked elements.

The refinement scheme is demonstrated from a topological perspective in Figure 4.6. The domain contains an initial notch which is refined. In the 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. The new regions of refinement are shown with orange circles around the new crack

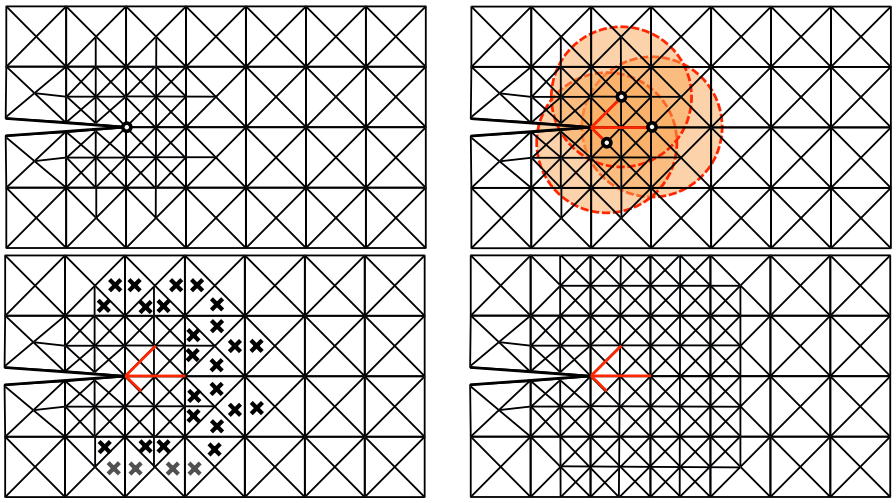


Figure 4.6 – 4k refinement scheme (a) Mesh is initially refined around the notch tip. (b) Cohesive elements are inserted along facets of fully refined elements, new crack tips are identified and new refinement regions associated with each crack tip are created. Elements to be refined for all crack tips are collected simultaneously, as opposed to one crack tip at a time. (c) Elements within the refinement region are marked (black 'x') and elements adjacent to the hypotenuse of a marked element are marked (grey 'x'). (d) Marked elements are refined to the full level and transition region refined to ensure element compatibility.

tips. The elements that fall within the refinement radius any of the crack tips is marked for refinement. Elements adjacent to the hypotenuse of a marked element are also marked. Finally marked elements are refined until the desired level of refinement is reached and elements just outside the refinement region are refined to an intermediate level to ensure compatibility with the coarser elements.

4.1.6 Adaptive mesh coarsening

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 race condition 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. The 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). Boundary conditions must be considered

```
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
```

Algorithm 2 – Kernel based algorithm to perform adaptive mesh coarsening

since two marked elements belonging to the mesh frontier may be refined also.

Once the nodes are marked for coarsening, a kernel call is used to update the adjacent elements' reference element by choosing one of both 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, updating the reference elements using the atomic operations since it may involve concurrency when updating an element. 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).

4.2 Adaptive cohesive fracture and fragmentation simulation

The adaptive fracture and fragmentation simulation procedure is similar to the 2-dimensional simulation, with a few differences. The mesh must be refined within a new crack tip region and coarsened based on the strain energy criteria. Table 2.1 shows the common simulation procedure. Table 4.1 shows the adaptive simulation procedure with a few changes. Before the simulation begins, we refine the mesh in regions where the fracture will initially evolve (i.e., where the first potential crack tip is located). The stiffness matrix and nodal masses are then updated, and within the simulation loop, displacement and stresses are calculated. After inserting cohesive elements and updating nodal masses, we coarsen the mesh based on a strain energy criteria. If the strain

energy in that is below a certain criteria in that region, we coarsen the region as explained in Section 4.1.4. Internal and cohesive forces are used to compute veolicites and accelerations, which are used to compute boundary conditions. Next, we identify crack tip regions according to Section 4.1.4 and refined the mesh. After refining the mesh, it is necessary to interpolate quantities, like displacement, velocities, and accelerations to newly created nodes. To do that, we use a T6 interpolation in boundary regions and Q9 interpolation in interior regions.

```
1:  Initially refine model in interest regions
2:  Compute Stiffness Matrix
3:  Update Nodal Mass
4:  current step ← 0
5:  while current step <= maximum step do
6:    Update Displacements
7:    if current step == check step then
8:      Compute Stresses
9:      if stresses > stress threshold then
10:       Insert Cohesive Elements
11:       Update Nodal Masses
12:       Coarsen the mesh based on strain energy
13:     end if
14:   end if
15:   Compute Internal Forces
16:   Compute Cohesive Forces
17:   Update Velocities and Accelerations
18:   Update Boundary Conditions
19:   Refine mesh in crack tip regions
20:   current step + = 1
21: end while
```

Table 4.1 – 3D distributed fracture and fragmentation algorithm

4.3 Experimental results

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 [70]. All or most of

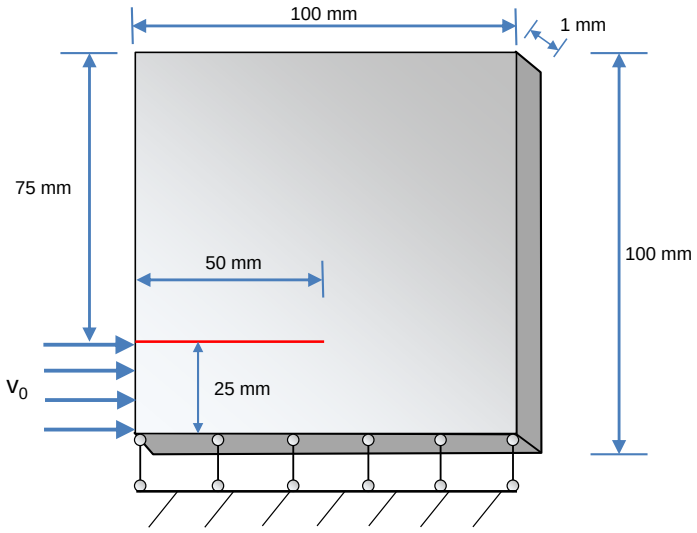


Figure 4.7 – Kalthoff-Winkler problem geometry and loading conditions

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.

4.3.1 Kalthoff-Winkler Experiment

We verify the implementation of the adaptive scheme on the GPU through a series of numerical investigations on a well-known micro-branching problem. The first problem is inspired in the experiments of Kalthoff and Winkler [9]. An illustrative design of the model using T6 elements is shown in Figure 4.7. Fracture propagation is based on mixed-mode fracture and extrinsic cohesive zone model [6, 56, 58]. The model is 100 m squared and has an initial notch at the left side at the height of 25 mm. An impact load is applied below the initial crack plane. The model was initially refined at the end of the notch with a radius of 0.01 mm until level four levels of refinement. Initial analysis parameters are as follows: initial velocity = 16.54 m/s, elastic modulus = 190 GPa, Poisson coefficient = 0.3, specific mass = 8000 kg/m3, fracture energy G_I = 22.2 kN/m, cohesive strength smax = 1.7 GPa, shape parameter $\alpha = 2$, and thickness = 1 mm.

Figure 4.8 shows the fracture propagation with the strain energy evolution throughout the simulation. The impact load is applied along the left boundary and creates compressing waves that propagate towards the right of the mesh in

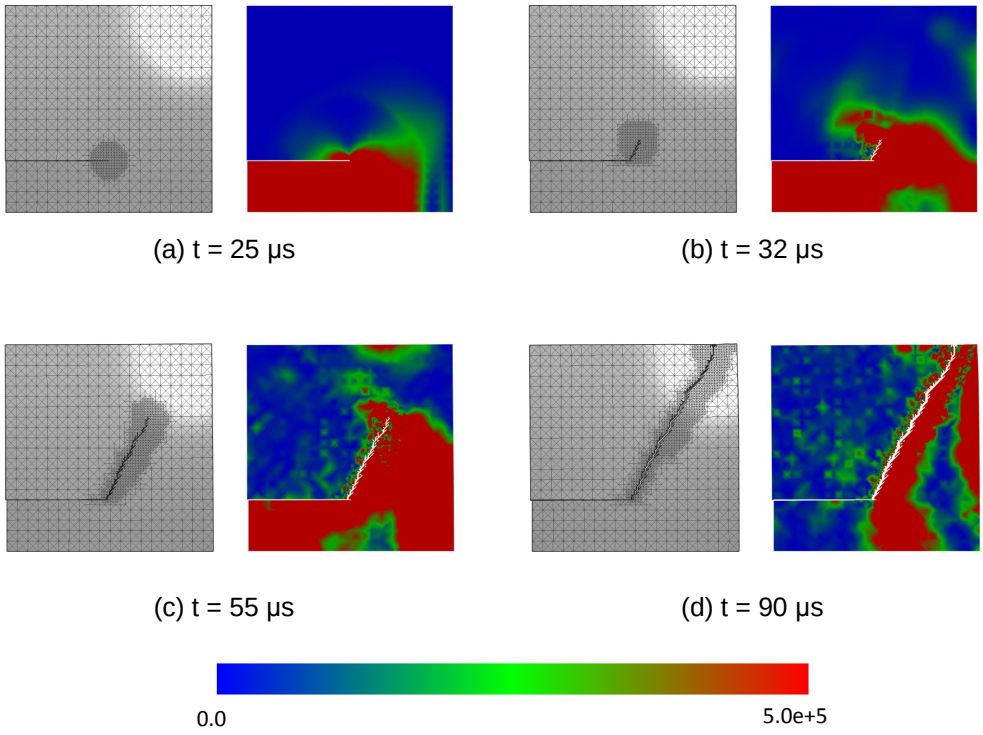


Figure 4.8 – Strain contour and crack propagation plots of the Kalthoff-Winkler experiment at different time instants: **a** 25 μs; **b** 32 μs; **c** 55 μs; **d** 90 μs

the lower section of the specimen. At t = 25 μs, a cohesive element is inserted because the local stress reaches a cohesive strength. Then, the crack propagates along the direction of about 60°. Refinement and coarsening were used in the simulation. The total time of the simulation was 27.24 s, with 18,000 timesteps and a timestep size of 5 ns. The mesh was composed of 1,600 coarse T6 bulk elements and 3,301 nodes. Nodes, bulk elements, and cohesive elements were inserted "on-the-fly" as the mesh was refined and coarsened.

4.3.2 Inclined plane

The Inclined Plane problem is based in the experiments of Dooley et. al. [13], which consists of a pre-notched specimen with length L = 456 mm, width W = 256 mm, initial crack length of $a_0 = 28$ mm, as shown in Figure 4.9. An inclined bonded interface divides the specimen domain diagonally at an angle $\theta = 60°$, changing the material interface that interferes in the crack propagation. The end of the notch has a region radius that is refined with level of refinement equals to 4. The crack initiates in the notch end, and propagates until it reaches the line. After that, it changes direction and propagates at an angle of about 60°. We change the Elastic and Fracture material properties

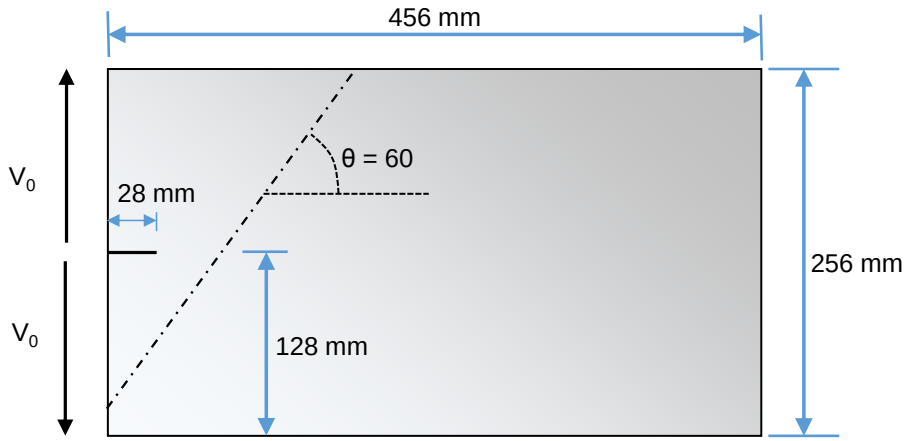


Figure 4.9 – Inclined Plane problem geometry and loading conditions

(normal and tangential cohesive strength, normal and tangential fracture energy) when the fracture reaches the bonded interface. Initial analysis parameters are: initial velocity = 1 m/s, elastic modulus = 3.45 GPa, Poisson coefficient = 0.35, specific mass = 1230 kg/m3, thickness = 1. Fracture energy materials before line are as follows: fracture energy G_I = 250 N/m, cohesive strength smax = 11 MPa, and shape parameter $\alpha = 2$. Fracture energy materials on line are as follows: fracture energy G_I = 46.4 N/m, cohesive strength smax = 7.745 MPa, and shape parameter $\alpha = 2$.

Figure 4.10 shows the fracture propagation with the strain energy evolution throughout the simulation. A velocity is applied below and above the notch. The mesh initially is composed of 29,184 bulk elements. The size of the timestep is 5 ns with a total of 100,000 simulation steps. The fracture starts propagating at t = 70 μs. At t = 195 μs, the fracture reaches the bonded interface, so it changes direction as the material interface also changes. The final number of elements in the mesh was 77,076, and the total time of the simulation was 258 s. No coarsening was applied to the mesh.

4.3.3 Reduced-scale micro-branching specimen

This model problem is inspired by the experimental work of [12] and has been simulated by many authors [17, 67, 71, 72]. 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 Figure 4.11. For the reduced

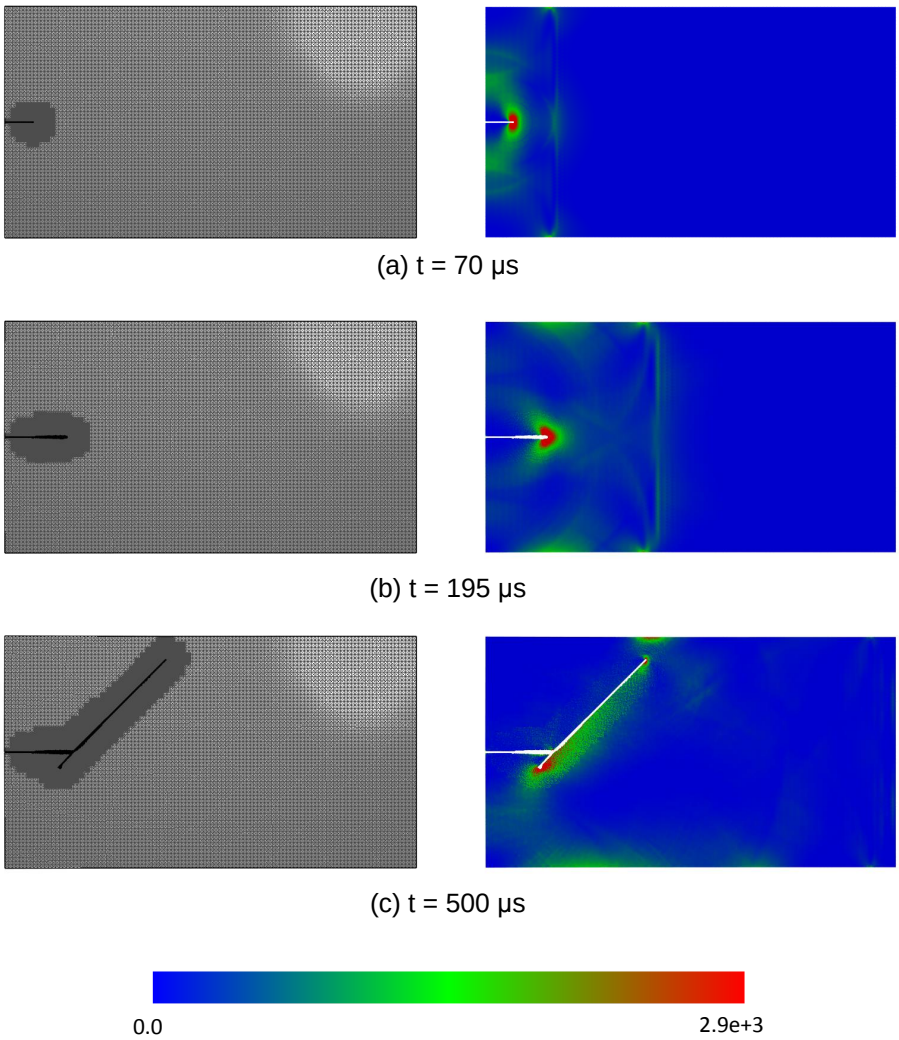


Figure 4.10 – Strain contour and crack propagation plots of the Inclined Plane experiment at different time instants: **a** 70 μs; **b** 195 μs; **c** 500 μs;

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 [71]. 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 $^{kg}/_{m^3}$, Poisson ratio of 0.3 for the bulk elements and 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 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[1] and AMR+C[2] to

[1]Adaptive Mesh Refinement.
[2]Adaptive Mesh Refinement and Coarsening.

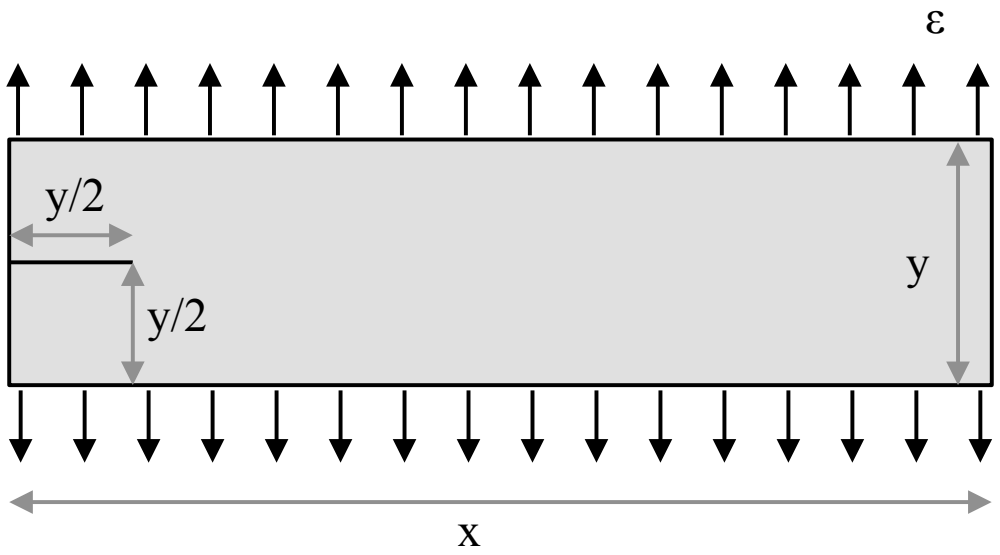


Figure 4.11 – Micro-branching problem geometry and loading conditions

that of an equivalent uniformly refined mesh. For the reduced scale model, the uniform mesh is comprised of 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 aways 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 Figure 4.12. 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 are plotted in blue, and elements that are present in the model but not open greater than the threshold in either direction 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 4.2 shows the final number of elements and nodes (after adaptivity and insertion of cohesive elements) and 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

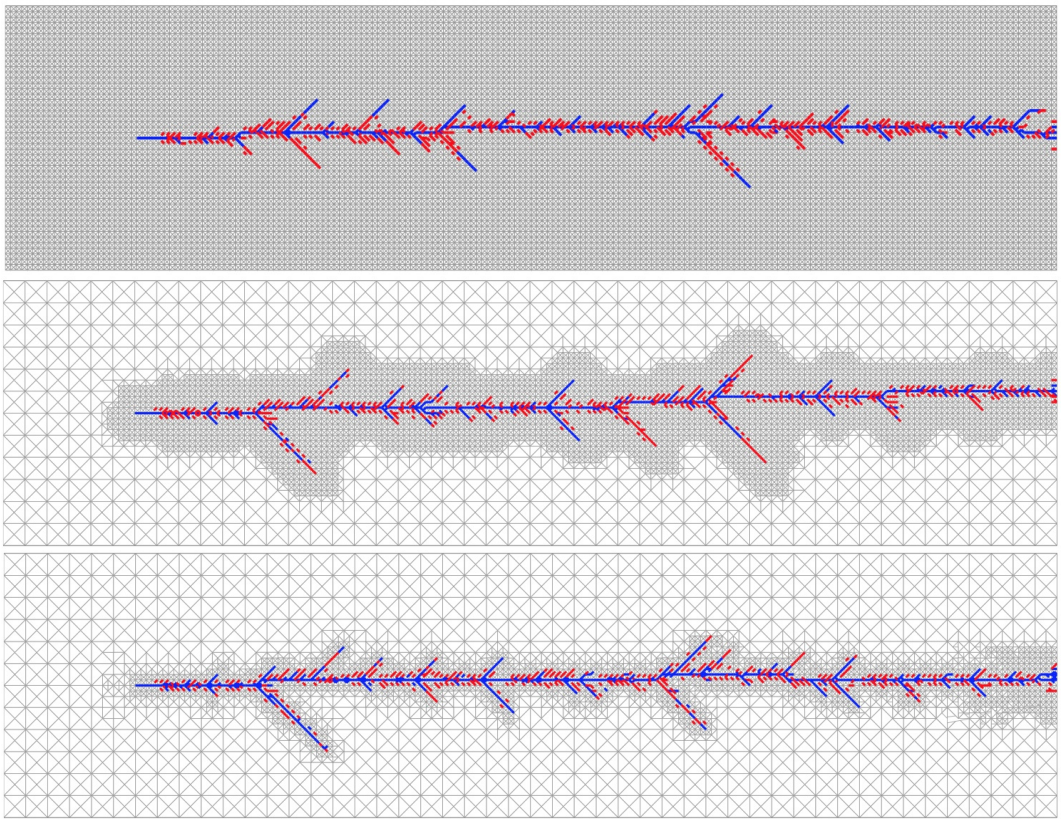


Figure 4.12 – Final crack pattern for the reduced scale micro-branching problem for (a) uniform mesh (b) AMR enabled mesh (c) AMR+C enabled mesh. Cohesive elements opened greater than 10% of the normal or tangential critical opening distance are shown in blue, other cohesive elements are shown in red.

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 4.2). 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

Table 4.2 – Comparison of final quantities between Uniform, AMR and AMR+C simulations

Tol	Mesh type	Elements	Nodes	Crack tip velocity	Total crack length	Num Branches	Avg. Brach Length
0.1	Uniform	36,864	76,268	777.5 m/s	0.034 m	2	2.2e-4 m
0.1	AMR*	13,277	28,506	754.3 m/s	0.036 m	1	5.2e-4 m
0.1	AMR+C*	8,303	18,296	755.5 m/s	0.039 m	2	5.1e-4 m
0.75	Uniform	36,864	76,268	777.5 m/s	0.019 m	14	4.6e-4 m
0.75	AMR*	13,277	28,506	754.3 m/s	0.021 m	19	4.1e-4 m
0.75	AMR+C*	8,303	18,296	755.5 m/s	0.021 m	20	4.7e-4 m

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

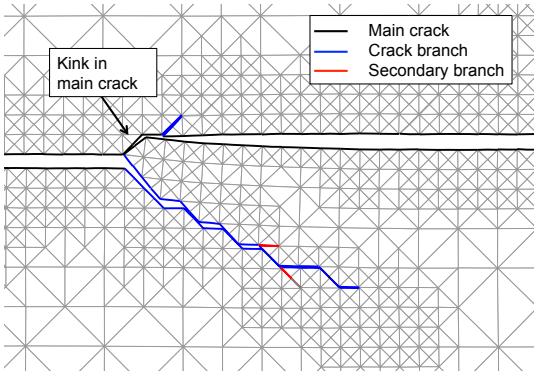


Figure 4.13 – Details of crack branching including kink in the main crack, crack branches, and secondary branches

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 Figure 4.13. 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 among 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 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 Figure 4.13. 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 the quantity that should remain similar among different numerical representations of the same process.

The quantities shown in Table 4.2 for the AMR and AMR+C cases are average over 20 simulations. This is because the massively parallel nature of the

Table 4.3 – 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	5.1e-4	4.1e-4 m	4.7e-4 m
	Standard deviation	4.2e-4	4.6e-4	4.2e-4	6.1e-4

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 to 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 $A + B = B + A$. 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 the 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 4.3.

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 among the 20 iterations, for the AMR and AMR+C cases the crack tip velocities are $754.3 \pm 9.8\,{}^{m}/_{s}$ and $755.6 \pm 10.1\,{}^{m}/_{s}$, respectively. Additionally, the total energy released during the fracture process is quite comparable, $75.0 \pm 2.6\,{}^{N}/_{m}$ 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 Figure 4.14. 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 [17] or by using a completely random mesh [72]. 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 .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. Table 4.4 shows the run times, speedups, and efficiencies over the serial implementation without adaptivity.

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 wall time between the uniform and adaptive simulations. More important, however, is that the size of the problem

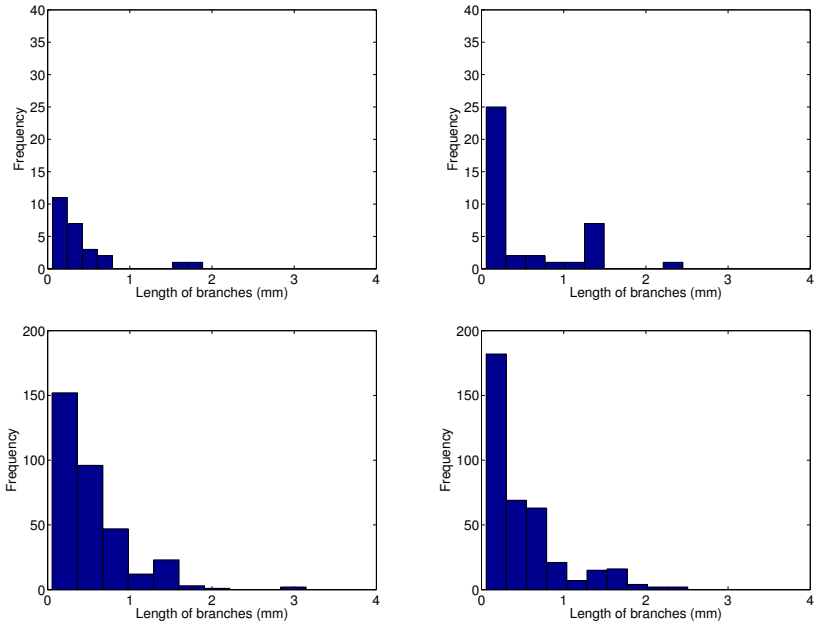


Figure 4.14 – Histogram of branch lengths over 20 simulations for the (a) AMR enabled meshes with an open crack tolerance of 75% of critical normal opening, (b) AMR+C enabled meshes with an open crack tolerance of 75% of critical normal opening, (c) AMR enabled meshes with an open crack tolerance of 10% of critical normal opening and (d) AMR+C enabled meshes with an open crack tolerance of 10% of critical normal opening

Table 4.4 – Comparison of wall time of the reduced scale micro-branching problem on different platforms (The speed up factor is shown with respect to the no adaptivity case on the serial CPU)

Platform	**Implementation**	**Wall time**	**Speed Up**	**Efficiency**
Serial CPU	No adaptivity	1,196 sec	–	–
Serial CPU	AMR	83 sec	14	–
Serial CPU	AMR+C	57 sec	21	–
Single GPU	No adaptivity	12 sec	100	3.7 %
Single GPU	AMR	18 sec	66	2.5 %
Single GPU	AMR+C	20 sec	60	2.2 %

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 would not be able to simulate otherwise.

4.3.4 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 set up proposed in [12]. The full scale problem size has dimensions 200×50 mm^2 (12.5^2 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 very high: up to 14 hours for each of the results shown here.

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 as possible. Thus specimen dimensions are those of the experiment, and the material properties are those of PMMA, the material used in the experiment. Young's Modulus is 3.24e9 Pa, density is 1190 $^{kg}/_{m^3}$, Poisson ratio is 0.3 for the bulk elements, 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 [71, 73]. A cohesive strength that is too large means that fracture never initiates, while a low

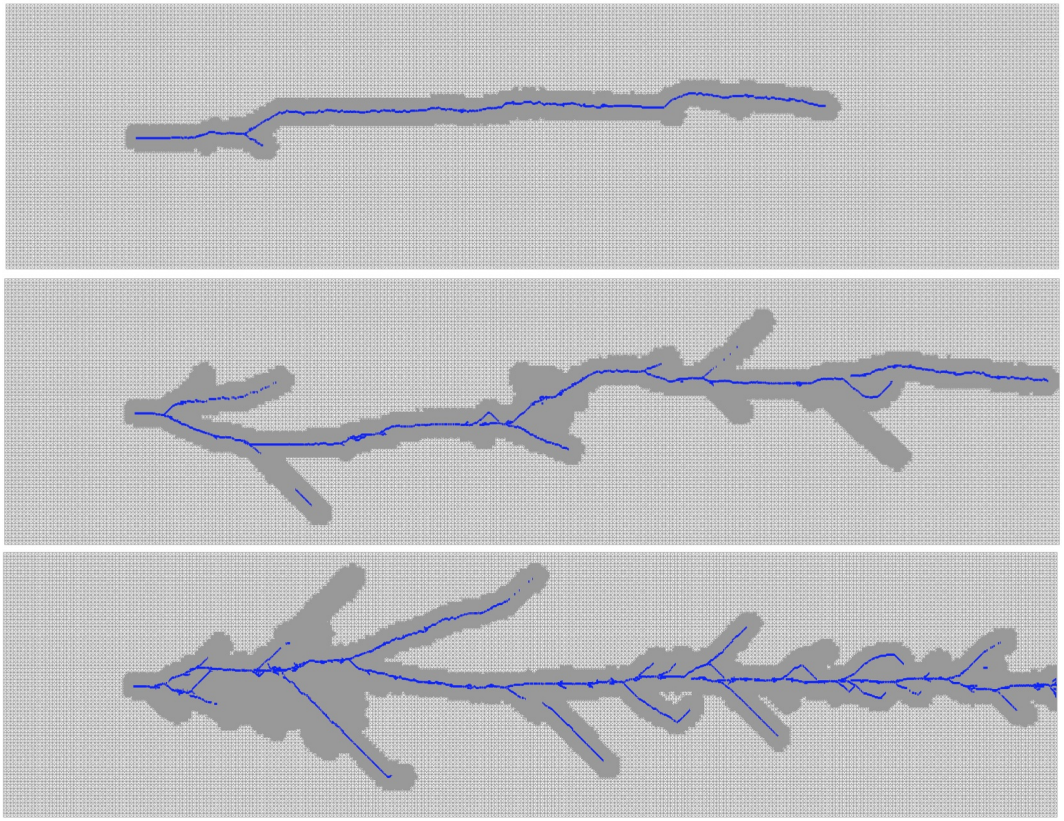


Figure 4.15 – Final fracture patterns for full scale micro-branching problem with an externally applied strain of (a) 0.003, (b) 0.004, and (c) 0.005.

strength results in the insertion of an excessive number of cohesive elements, which is not physically realistic. Thus, we used the value recommended in [8] and [67]. 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 300×75 4k mesh patches, or 90,000 elements. We use the AMR to sufficiently reduce the element size at the notch tip. 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 the adaptivity is essential. The fracture patterns for three different strains are shown in Figure 4.15. 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 [12]. At lower strains, the fracture surface is smoother and features one predominate crack. 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. The velocities of the three cases also increase with increased applied strain. In the lowest strain case, the velocity is relatively stable and lower than the higher strain cases. As the strain increases so do the velocity and the oscillation of the crack tip velocity.

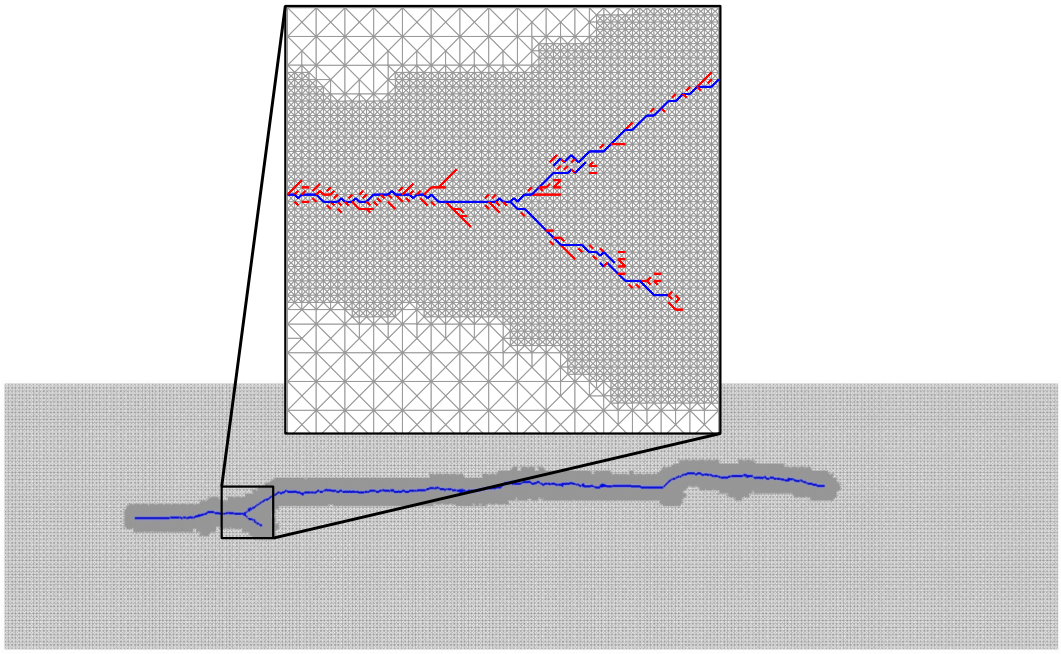


Figure 4.16 – Detailed view of fracture pattern for the full scale micro-branching problem with an externally applied strain of 0.003

The details of the crack pattern and the adaptive mesh refinement scheme are shown in Figure 4.16 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 of 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.