3
Two-dimensional Cohesive Fracture and Fragmentation Simulation

Mapping the simulation to GPUs is not a trivial task. As mentioned before, there are lots of memory and concurrency issues we have to take into account. Fortunetely, with a simple and compact data structure, we are able to minimize memory access and hide latency. Other algorithmic optimizations that can greatly help in gaining performance will be addressed in this chapter. Here, we describe the data structure used to store topological entities in the GPU and traversal algorithms we are able to perform using them. From there, we explain in details all the steps of the GPU simulation and the optimizations that can be done to gain performance. In this chapter, we focus on 2D cohesive fracture and fragmentation simulation.

3.1
Data Structure

In order to implement efficient operations on mesh entities used in the simulation, a simple topological data structure is employed to represent the mesh. Because the GPU memory is limited, the data structure must not be complex so as to provide space for other simulation attributes that indeed require much global memory space. The proposed data structure presented in this Section is used for T3 or T6 meshes. We will discuss the support for tetrahedra elements (Tet4) in Chapter 5. Although the support for tetrahedra meshes is straightforward, inserting cohesive elements in 3D requires a graph structure to traverse a node's incident elements, which is much more expensive for the GPU. In this section, our data structure will focus on 2D elements only.

We maintain two tables that describe the mesh. A table of nodes stores the node world-space position (x and y coordinates). A second table is used to represent the elements and adjacency relationships. The elements can be of two types: bulk elements or cohesive elements. Because the number of bulk elements remains unchanged during the entire simulation, we store the cohesive elements immediately after the bulk elements. For each bulk element, we store its nodal incidence; three node indices are used in a T3 mesh and six are used in a T6 mesh. For T6 meshes, the corner node indices are followed by the mid-side indices. The right hand rule (counter clockwise) is used to define the order of nodal incidence. Another three values represent adjacent element indices that are opposite to the facet that are opposite to each of the corner nodes (three

values for T3 and T6 mesh). For cohesive elements, we store six node indices for a T6 mesh and four node indices for a T3 mesh. In a T6 mesh, the first three node indices represent the three nodes of the corresponding facet of its adjacent bulk element with the smaller *id*, following the right hand rule. The next three indices belong to the adjacent facet of the second adjacent bulk element (with the greater *id*). For a T3 mesh, two indices are stored for the each cohesive element. Another two values are used to represent indices of both bulk elements that are adjacent to each cohesive element's facets (in both T3 and T6 mesh). Following the pattern, the first bulk element is opposite to the the first facet (and to the first three node indices) of the cohesive element. If a cohesive element is attached to a bulk element's facet, we update the opposite element of that facet to the cohesive element *id* on the element table. Nodes or elements are represented by their indices in the corresponding table. The last index in the table is not used for cohesive elements. Both node and element tables are stored in the global memory and are updated along the adaptive numerical simulation. Figure 3.1 shows an example of tables used in the data structure.

Node Table

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

Element Table

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

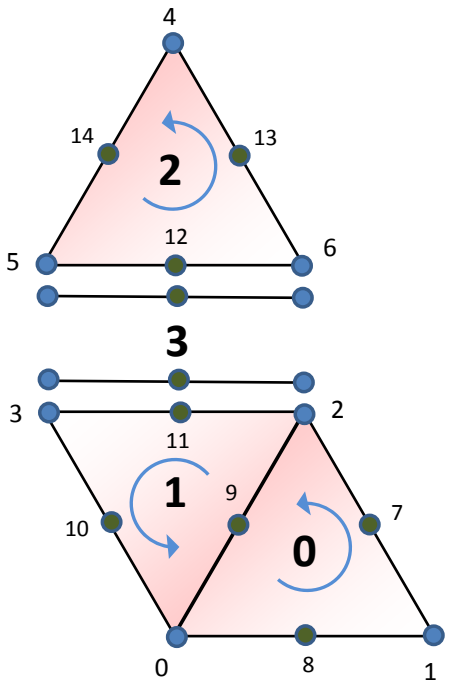

Figure 3.1 – A special-purpose simplified data structure with mesh parameters of a T6 mesh.

With our data structure, we are able to perform the following computational patterns (see Figure 3.2): a node can be updated based on its own information; a bulk element can be udpated based on its own information; a bulk element can be updated based on information of its nodes; and a node can be updated based on information of its incident bulk elements. For the last pattern, it is necessary to store a bulk element identifier for each node so as to start traversing its incident elements. In our implementation, we do not store it because we use this computational pattern by sweeping the bulk elements first. This only works for global computation (that is, applied on all nodes of the bulk elements, see Section 3.1.2). For each node of a bulk element, we traverse its incident elements. Not storing a bulk element *id* for each node also allows us to save GPU memory.

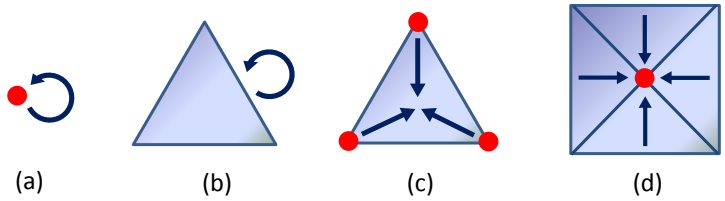

Figure 3.2 – Simulation's computational patterns. (a) a node can be updated based on its own information; (b) a bulk element can be udpated based on its own information; (c) a bulk element can be updated based on information of its nodes; and (c) a node can be updated based on information of its incident bulk elements.

A finite element analysis maintains a set of simulation attributes attached to nodes and elements. We maintain such attributes in global, constant, or texture memory depending on their memory size and dynamics during the simulation. In global memory, we store the attributes that change throughout the entire simulation. The nodes have associated displacements, velocities, accelerations, and forces (internal and cohesive), which are updated at every timestep. Stresses and strains evaluated at the nodes are updated only every few time steps. When facets are checked for possible fractures, the fractured facets in the element attributes store which facets have been fractured. Nodal mass and number of adjacent bulk elements are updated whenever the topology of the mesh changes. Finally, cohesive attributes such as tractions and separations are updated every step for each cohesive element. We store other node and element attributes that remain unchanged during the entire simulation, but require too much memory space, in textures. We cannot store these attributes in constant memory due to its limited memory space. Each element's stiffness and lumped mass matrix and each node's boundary conditions are stored in texture memory. Other attributes that are common to all nodes and elements,

such as elastic and fracture material properties, are stored in constant memory. These attributes are stored in one table common to all elements and nodes and occupy little memory space. Because all threads in a warp access the same memory space in constant memory, there will be no bank conflicts. Figure 3.3 shows element and node attributes that were stored in the GPU global, texture, and constant memories.

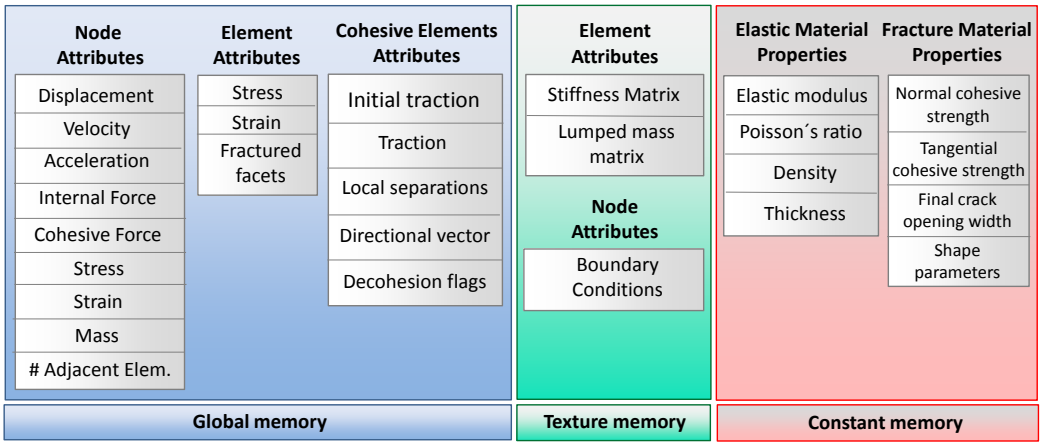

Figure 3.3 – Simulation parameters data structure diagram of FEM model. Global memory is used for attributes that change throghout the simulation. Texture memory is used for attributes that are constant during the entire simulation, but occupy too much memory space. Constant memory is used for attributes that are constant during the entire simulation, but are common to all elements and node, therefore requiring few memory space.

Paulino et al. [61] present a topology-based framework for supporting fragmentation simulations in extrinsic cohesive zone models for CPUs. Their topological data structure, TopS, contains all information necessary to retrieve element adjacency relationships needed for the simulation and is able to perform, for example, the previously described computational patterns (see Figure 3.2). While their data structure is designed for general element types, our specialized data structure is simpler and has been especially tailored to operate efficiently on GPUs (i.e., data is stored in 2D arrays vs. structures). To represent the mesh on the GPU, it is sufficient to store node positions, node indices for elements, and adjacent elements, which is much less information than is stored in TopS [61]. Both the element and node tables are designed to make as few global memory accesses as possible, as well as occupying as little device memory as possible. Saving device memory is a key issue because memory from a single GPU is extremely limited. Currently, we lack information of storing a reference element for each node (if we want to traverse its incident elements by

sweeping the nodes first, we would have to have access to one of its incident elements). However, while inserting cohesive elements and duplicating nodes, this information is redundant in this step because we sweep each element first, followed by each of its nodes' incident elements. We also need to perform more arithmetic operations when traversing them because we lack information of node order in each element. However, the lack of this additional information as well as additional arithmetic operations is compensated by making fewer global memory accesses, which is a fine tradeoff for GPU programming. It has also the advantage of saving GPU memory to run larger models.

3.1.1 Retrieving adjacency relationship

The node and element tables are enough to perform the previously stated algorithms and do not require too much memory usage. One key adjacency relationship for the insertion of cohesive elements is the set of adjacent elements to a given node. With the described data structure, from an element, for each of its incident nodes, we can easily traverse the set of adjacent elements. Given the first node, we search the other node that precedes it in the order of incidence, and then access the corresponding opposite element and find the order of incidence of the node that had the previous element as its opposite. From both the element and node order, we obtain the next node in the incidence of the element and access its opposite element. From there, we repeat the procedure until we reach the element adjacent to the first one. Figure 3.4 illustrates the traversal algorithm from a given node if no cohesive element is reached within the path. Blue arrows indicate which node to access in order to get the correct opposite element. Dashed lines indicate the node we must access to obtain the next element required to continue traversing. Figure 3.5 illustrates the traversal algorithm from a given node until it reaches a cohesive element. Cohesive elements can also be obtained by traversing around a node, since they are also stored in the element table and can be accessed by obtaining the opposite element for a given node.

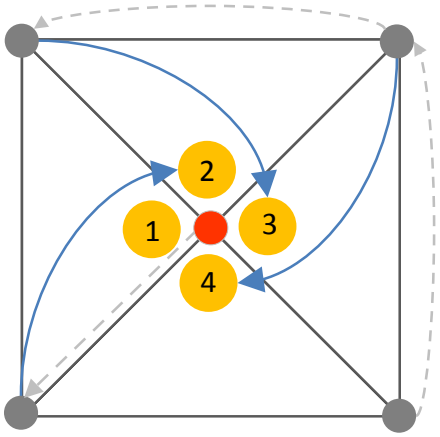

Figure 3.4 – Traversal algorithm from a given node using the proposed data structure. The illustrated path does not contain cohesive elements.

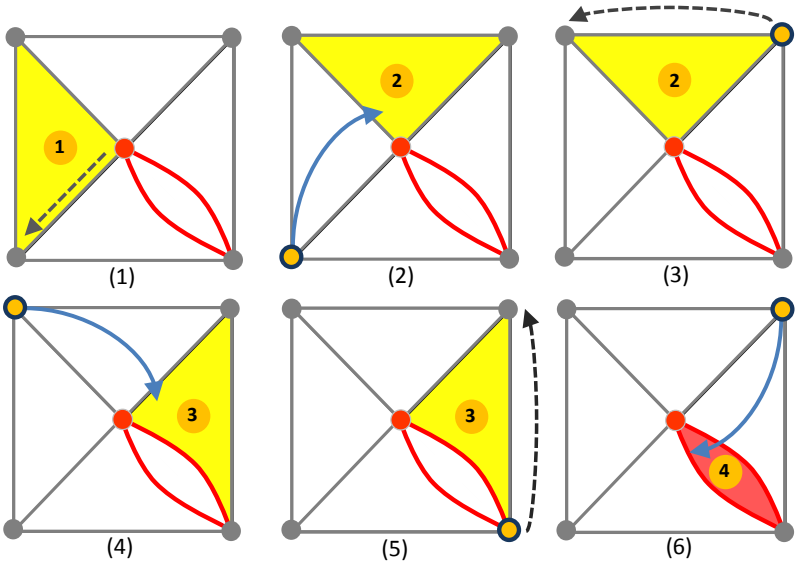

Figure 3.5 – Traversal algorithm from a given node using the proposed data structure, with cohesive elements along the path. From a bulk element, the algorithm starts by accessing a node whose opposite element is incident to the traversed node (central node) (1). The opposite element to that node is obtained (2) followed by the next node (3). The third bulk element is accessed (4), followed by its respective node (5). A cohesive element opposite to a node (or adjacent to a bulk element's facet) can also be reached (6), since it is explicitly represented in the element table (see Figure 3.1).

3.1.2 Node update

The set of adjacent elements of a given node is necessary to update element incidence when a node is duplicated due to the insertion of a new cohesive element. During the simulation step, we can also identify computations

where such topological relationship can be used. As an example, we can consider the mass associated to a node, which depends on contributions of all adjacent elements. However, as we discuss further in our parallel simulation, for almost all cases, accumulating contributions of adjacent elements to nodes is more efficiently handled by traversing all the elements in the model. Such cases include calculating internal and cohesive forces and stress and strain on nodes. For each element, we accumulate its contribution to all incident nodes. In the end, the contributions of all corresponding adjacent elements will be accumulated for each node. In a serial code, this algorithm is straightforward and very efficient. In a parallel environment, writing conflicts arise, and one needs to ensure consistency, as we shall discuss. Figure 3.6 illustrates both strategies to compute nodal information from its adjacent elements.

The first strategy, known as Gather, traverses the node's incident elements, accumulating their information on the node. The second strategy, known as Scatter, accumulate each element's contribution to all incident nodes. We tested both algorithms on the GPU by accumulating each element's mass on their incident nodes. In our experiments, the Scatter algorithm turned to be more efficient as we increased the number of elements. While this algorithm implements a simple kernel function with few global memory accesses and no divergence, the Gather algorithm implements a more complex kernel requiring greater number of registers and adding divergence (not all nodes have the same number of incident elements). Although divergence can be minimized by sorting elements according to their incidence, this algorithm still tends to perform much more computation than the other. The Scatter approach is well suitable for problems that require many computations per elements, but when duplicating nodes and inserting cohesive elements on fractured facets, the Gather strategy is essential as the basis of the algorithm, as discussed in Section 2.2.3. To avoid writing conflicts in the Scatter algorithm, we adopted the commonly used mesh coloring representation, as we shall discuss later. While testing both algorithms by accumulating element mass on nodes, it was observed that a higher number of colors or unbalanced number of elements in each color lowers the efficiency of the Scatter algorithm. As we increase the number of elements, Gather's efficiency tends to deacrease in relation to Scatter. For these reasons, we adopted the Scatter algorithm for all cases except when inserting cohesive elements and duplicating nodes, in which we used the Gather algorithm.

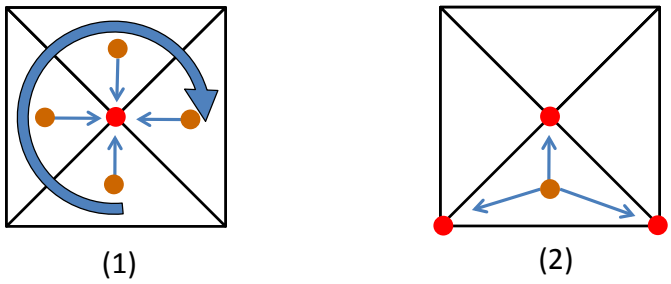

Figure 3.6 – Node update algorithms: (1) incident elements traversal (or gather), and (2) element sweep (or scatter).

3.2 Parallel Implementation

The main challenge for implementing a many-core parallel fragmentation simulation, based on the extrinsic cohesive zone model, is to ensure topological consistency on mesh adaptation (insertion of new cohesive elements). However, even the mechanics code, at first straightforwardly parallelized, based on explicit integration, also imposes challenges. Memory access and usage can be a bottleneck when using the slow accessible global memory space. Concurrency is also an issue to have in mind, since writing conflicts can eventually occur when updating the same memory space for different threads running concurrently. In order to maximize the performance and benefit from GPU parallelism, it is important to keep in mind programming techniques discussed in Section 1.2.2, or else the attempted GPU speedup will be negligible. Although the parallel algorithms discussed below refer to a T3 or T6 mesh, they can be extended to 3D meshes using a modified version of the previous discussed data structure.

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

3.2.1 Coloring model

In this discussion, we consider the implementation of T6 meshes. The first parallel procedure to be discussed is updating the node attributes. In our case, we are focused on updating each nodal mass with the lumped mass matrix from each adjacent bulk element. The lumped mass matrix is computed in a pre-processing phase together with the stiffness matrix. We could launch one thread per element and accumulate the element mass on its respective nodes

retrieved from the incidence table. However, threads would write on the same memory space since different elements share the same node.

To avoid the race condition, we adopt the commonly used mesh coloring representation. The idea is that no element of the same color shares a node. Applying this technique when accumulating the nodal masses from the bulk elements means that we will launch a kernel for each color with a thread per element in that color group. With this strategy, different threads will not update the same node since there won't be elements with shared nodes being processed in parallel. Since bulk elements are neither removed nor inserted during the entire simulation (only cohesive elements and nodes are inserted), mesh coloring can be pre-processed. In graph theory, a node degree (also called valency) is the number of incident edges to that node. In our model, each element represents a graph node and adjacent elements are connected by a graph edge. The minimum number of color groups is equal to the maximum node degree on the entire mesh. However, determining the minimal color number of a graph is known as an NP-complete problem, although there are many heuristics for finding a reasonable solution. In our case, we will be interested in finding a reasonable and balanced solution, or else we will be wasting additional kernel computations with few threads per color containing few elements, while having other color with many more elements. We order the graph nodes (elements) in decreasing order of degree to obtain the closest optimal solution. Table 3.1 shows the procedure we use to perform a conceptual execution unit on the elements in parallel: we launch the same kernel multiple times, one for each group color. Figure 3.7 illustrates the colored mesh and its use in kernel calls and updating the nodal masses using the scatter strategy. We use the Welsh Powell algorithm [62], a greedy algorithm[1] to color unstructured meshes. In our experiments, we use structured "union-jack" meshes. For these meshes, we apply a coloring algorithm that takes advantage of their pattern so as to obtain the optimal color number.

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

Table 3.1 – Kernel subroutine call algorithm using mesh coloring

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

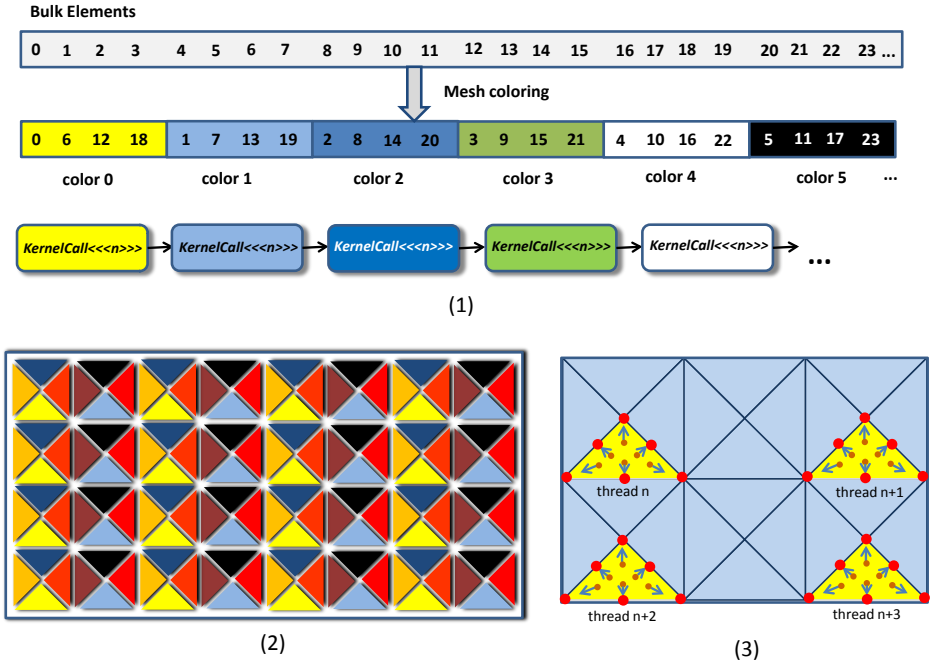

Figure 3.7 – (1) Bulk elements are re-arranged in color groups (preferable balanced) and the same kernel per color group is called to avoid writing conflicts. (2) Example of a colored T6 structured mesh (3) and using the colored mesh and scatter strategy to update nodal masses of the group of elements in the current color in parallel.

3.2.2 Pre-processing and update

A pseudo-code of the parallel simulation is shown on Table 3.2. In the pre-processing phase, also executed on the GPU, we need to compute the stiffness matrix and the lumped mass matrices associated to each element, and then update the nodal masses. Building the stiffness matrix requires one thread per element but with no color subdivision scheme since we write directly in per-element memory space. The same kernel computes each element's lumped mass matrix. The last kernel in the pre-processing phase updates the nodal masses with the lumped mass matrix by using the previously discussed parallel algorithm, invoking a kernel per color group. We use constant memory for storing material attributes that are constant during the entire simulation. Cache hits when fetching these attributes during stress and other force computations will help increase performance since threads in the same warp access the same value at the same time.

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

Table 3.2 – Parallel Fracture Algorithm

Figure 3.8 depicts the steps in a simulation loop. The first kernel in the simulation loop updates the nodes' displacements, launching one thread per node. Each thread fetches the velocity and acceleration of its corresponding node from global memory and updates the result back in global memory following the Equation 2.2. This is a simple kernel that uses few global memory accesses and every thread in a warp follows the same path since there are no conditionals or loops.

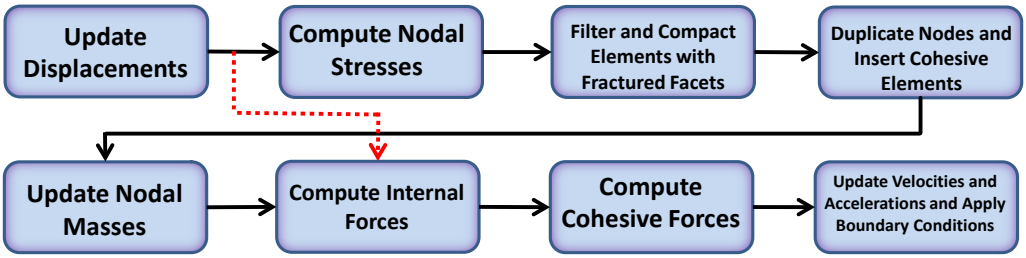

Figure 3.8 – Fracture and fragmentation simulation loop.

3.2.3 Stresses

Before checking fractured facets, the next procedure is responsible for computing the stresses and strains on the nodes by first calculating them at the Gauss points for each element, multiplying its respective matrix with the element shape function as showed in Equation 2.4 in Section 2.2.4, and writing them back on the elements' nodes. Each node stress is then checked for cohesive strength over a threshold value so it can later indicate if a facet is fractured or not. To implement this whole procedure in a single thread, we would need to launch one thread per element using the color model to avoid concurrency. This single kernel would have too many loops and global memory accesses that cause a low performance. Also, the number of registers would exceed the established limit, forcing the compiler to put local variables on local memory residing on global memory. Another issue worth highlighting is that this complex kernel would be executed several times because of the color model. We have then opted for an alternative strategy to reduce effort and increase performance, dividing this complex kernel into three simpler ones. In the first kernel, we compute the elements' stresses and strains at the Gauss points by launching one thread per element and with no color model. The second kernel calculates the stresses and strains matrix for each node, launching one thread per element but this time using the color model since each element accumulate results on its nodes. Notice that this kernel's effort is reduced since it only performs read-write on global memory. The third kernel checks if each node's principal stresses exceed the cohesive strength limit by launching one thread per node. The kernel dividing technique is useful as it distributes efforts among simpler kernels by reducing global memory accesses and reducing loops, and it will be adopted on other kernels too. Looking at Equation 2.4, we can observe that the second kernel performs several global memory accesses because it accumulates element stresses and strains on its nodes by fetching from the element stress and strain

matrix (3x4 matrix) at the Gauss points, computed on the previous kernel. An alternative strategy is to launch one thread per element node (6 threads per element), and each thread is responsible for multiplying the stress and strain matrices at Gauss points with the respective nodal shape functions and writing the result in its respective node. We opt to launch 12 threads per element where each thread would fetch two columns from the four-column Gauss point element matrix line and write the result on part of the 2x2 nodal stress and strain matrix. This strategy reduces global memory access per thread, reducing the kernel effort. Figure 3.9 illustrates the stress kernel division and Figure 3.10 illustrates the second kernel procedure.

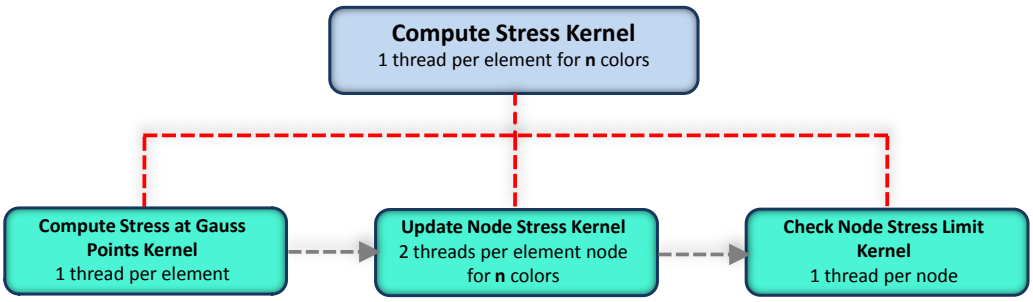

Figure 3.9 – Splitting the kernel that computes stress and strain into simpler kernels.

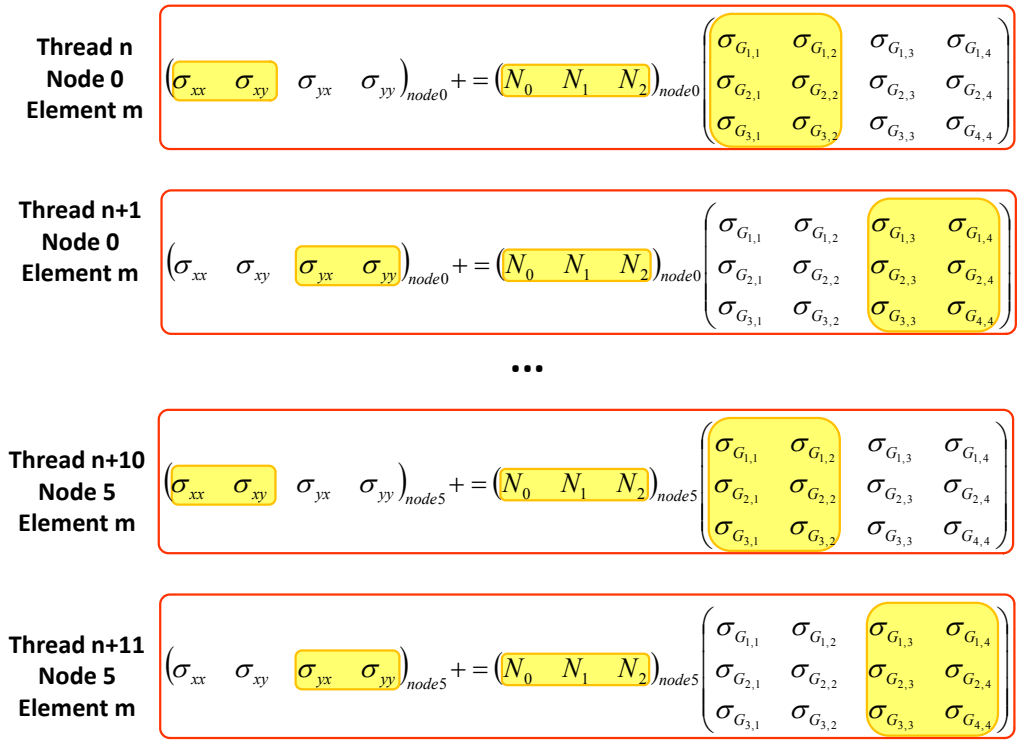

Figure 3.10 – To accumulate the stresses and strains on the nodes, we launch 12 threads per element, where each thread will accumulate part of the stress and strain matrices by fetching from the element shape functions and from the stress and strain at the Gauss points.

3.2.4
Insertion of cohesive elements

Once fractured facets are identified, new cohesive elements must be inserted in the mesh. When inserting cohesive elements, launching one thread for each element can result in idle kernels because there are few elements that contain fractured facets. In order to solve this matter, an additional kernel is used before inserting cohesive elements. This additional kernel filters only the elements that contain fractured facets by launching one thread per element and checking its 3 facets for possible fractures as discussed in Section 2.2.4. However, a fractured facet always belongs to two elements that are adjacent to each other, and we cannot filter both elements for the same facet otherwise the nodes will be duplicated twice. Therefore, we chose the element that has the smaller (or greater) identifier number. In our implementation, we also maintain a list of bulk elements that are adjacent to existing cohesive elements. This list is useful when later computing cohesive forces, otherwise idle kernels will be included in this simulation step as well.

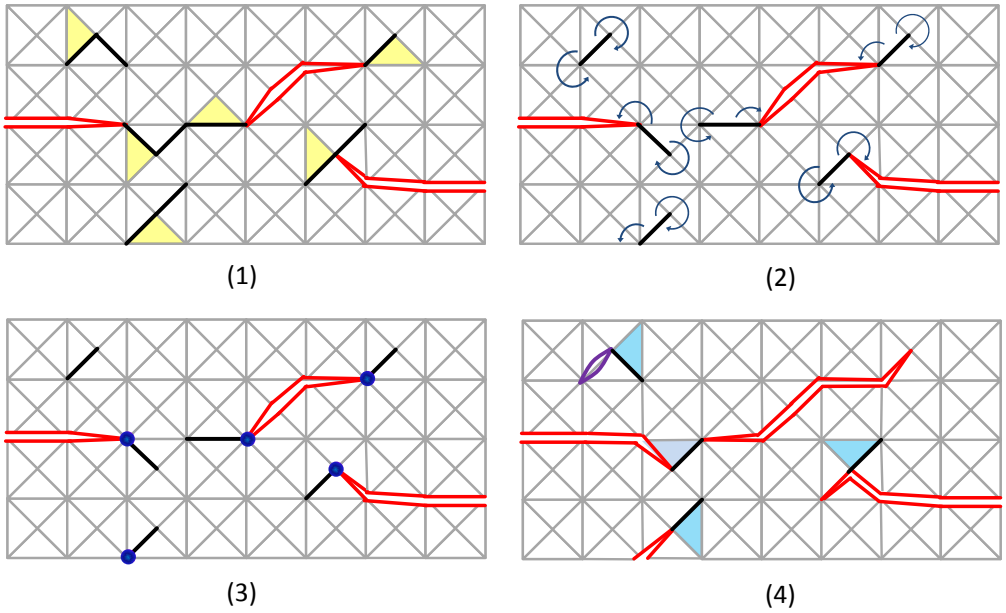

Figure 3.11 – Cohesive elements insertion on a T6 mesh. (1) Mesh with initial cracks and facets that fractured facets. Coloring is used to avoid duplicating nodes of elements that share nodes in parallel. (2) From each facet node belonging to the element in the current color group, the algorithm traverses through its incident elements. (3) Nodes that need duplication. (4) T6 mesh with final node duplications and new cracks and cohesive elements. The fractured facets from the next color group are checked for cohesive elements insertion.

From the list of elements containing fractured facets, we now check for node duplication and insert the cohesive elements. We use mesh coloring on the filtered elements' list and launch one thread per element. Figure 3.11 illustrates the parallel cohesive element insertion process. During one element computation, we go through its fractured facets and check its nodes for duplication. The same traversal algorithm presented in Section 2.2.4 is used to check if the node has to be duplicated. If so, we need to update the global nodal counter and retrieve the new node index. However, because the node counter resides in one global memory address, many threads updating the same counter cause a writing conflict. To solve this matter, we use CUDA's atomic operations to perform a read-modify-write operation (in this case, a global variable increment), without the interference of other threads. The function `atomicAdd()` computes the sum on the word located in the global address and returns the previous stored word. Therefore, it returns the new node index needed to update the elements' incidence table. Node attributes are then copied to the newly appended node. The traversal algorithm is used to go through the node's adjacent elements until it reaches the cohesive element while updating their nodes with the new

index value. We also need to update the opposite indices in the element table. Table 3.3 presents the parallel cohesive element insertion algorithm.

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

Table 3.3 – Parallel Node Duplication Algorithm

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

3.2.5 Internal Forces

Computing the internal forces is another expensive kernel and occupies a large portion of the simulation as it is executed every time step. Our first approach was launching one thread per bulk element and use the color model since the elements' nodes are updated. The stiffness matrix is multiplied by the displacement vector, resulting in the nodal internal forces. In a 2-dimensional case, the stiffness matrix has dimension 12×12 and the displacement vector 12×1. With a naive multiplication code, we make 1,728 global memory accesses. A strategy to reduce the number of global memory fetches is to load the

displacement vector into shared memory once and use it for multiplying each line of the stiffness matrix. This greatly reduces the number of global accesses from 1,728 to 156. The performance, however, still does not reach optimal expectations. By making each thread responsible for computing the product of one line of the stiffness matrix with the displacement vector, the number of global memory accesses is reduced to 13 (one for loading a value from the displacement vector into shared memory and 12 for fetching values from one line of the stiffness matrix), as well as the kernel's effort. Launching one thread per matrix line means we are launching 12 times the number of threads per element. Since coloring is used, the total number of blocks hardly exceeds the limit. Going further, since the stiffness matrix is constant during the entire simulation, it can be stored in a texture memory to take advantage of the texture cache and the spatial locality accessed by the warp. In order to guarantee the right memory access in each thread, we define the thread block dimension (1D) as the number of matrices per block times the number of threads per matrix (in our case, 12 threads for each matrix). To guarantee the thread block is multiple of the warp size, we use 16 matrices per block (192 threads per block) on a GeForce GTX 480 GPU. Each thread of the block loads one value from the displacement vector into shared memory and are synchronized. Notice that each group of 12 threads will load its respective element displacement vector. One issue remains, however. At the same time thread 0 is reading address 0 (row 0, column 0), for instance, thread 1 of the same warp will be reading address 12 (row 1, column 0), thread 2 will be reading address 24 (row 2, column 0), and so forth. This means that memory is not being coalesced at all. In order to properly perform coalesced readings and achieve a higher bandwidth, consecutive threads must read consecutive memory addresses. Therefore, we transpose the matrices and each thread of a warp will be able to read consecutive addresses. Figure 3.12 illustrates this strategy.

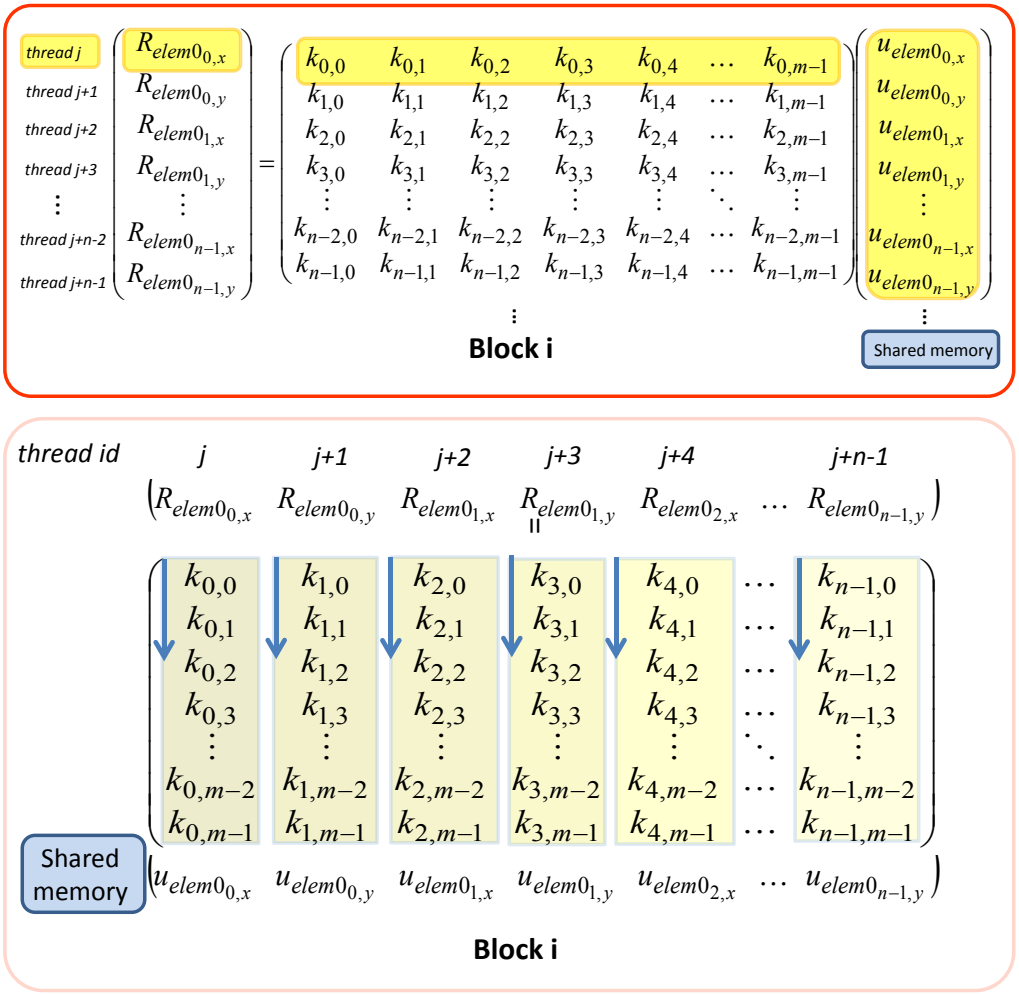

Figure 3.12 – When computing internal forces, a thread per stiffness matrix line is launched using the color model and used to perform a dot product with the displacement vector in shared memory. In this example, the first image shows two elements per block used. The second image shows the matrices transposed so memory reads can be coalesced (that is, each consecutive thread reads consecutive memory addresses).

3.2.6
Cohesive forces and simulation outcome

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

technique could help increase performance. In the *first kernel* we calculate the cohesive separations in the local coordinate system. One thread per cohesive element is launched, since we write directly on the cohesive attributes memory space. The *second kernel* calculates the cohesive traction by also launching one thread per cohesive element. However, this is the most expensive kernel in terms of arithmetic operations, especially when we need to calculate the cohesive tractions for each of the three Gauss points. Therefore, we adopt the previous strategy of launching more than one thread per element. In this case, we will be launching three threads per cohesive element, one for each of the three Gauss points. Each thread is responsible for calculating the tractions for its cohesive element in its respective Gauss point. Since the total number of cohesive elements in the simulation is relatively small, the number of threads will not be high. This strategy helps increase the performance of the the kernel. Finally, the *third kernel* consists of writing the cohesive forces on the cohesive elements' respective nodes. We then launch one thread per bulk element that contains any cohesive element (using the list previously mentioned). To avoid concurrency, the threads are separated by color group. The cohesive kernel subdivision is shown in Figure 3.13

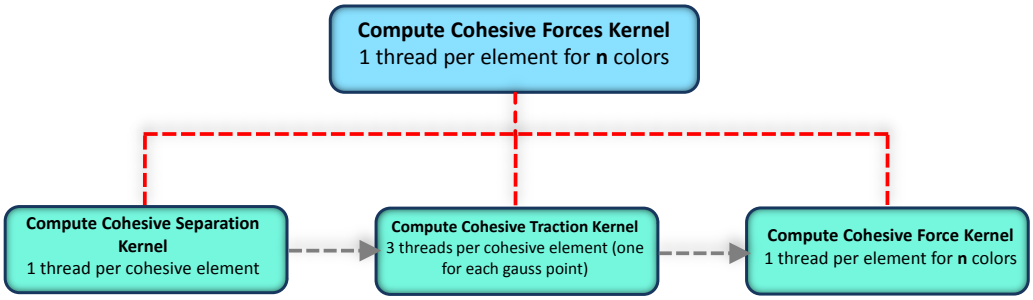

Figure 3.13 – Splitting the kernel that computes cohesive forces into simpler kernels.

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

3.2.7 Overview

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

3.3 Experimental results

To test the performance and the correctness of our two-dimensional parallel code, we have run a set of computational experiments. The experiments were split in two parts: inserting cohesive elements decoupled from mechanics analysis and running the fracture and fragmentation simulation. The GPU simulation results are compared to CPU counterparts running on a Intel Core i7 CPU @ 2.80GHz with 12GB of memory on a 64-bit Windows 7 operating system. The GPU used device is a NVIDIA GeForce GTX 480 with 15 multiprocessors, each with 32 cores and a total of 480 CUDA cores, with a clock rate of 1.40 GHz and using compute capability 2.0 because we use double precision in the simulation. The total amount of GPU memory is 1.536 Gigabytes.

3.3.1 Insertion of cohesive elements

To check the correctness of the algorithm to insert cohesive elements in parallel, we have run a computational test decoupled from any mechanics simulation (setting up experiments similar to the ones described by Pandolfi and Ortiz [63] and by Paulino et al. [61]). Cohesive elements were inserted, in a random order, at all the facets of the underlying meshes. The random order

in which the cohesive elements are inserted results in arbitrarily complex crack patterns during the experiment. In the end, each node of the mesh is used by only one bulk element. We then have checked if the final obtained number of topological entities were the expected ones. In the experiments ran by Pandolfi and Ortiz [63] and Paulino et al. [61], the cohesive elements were inserted in a serial order. In our experiment, the cohesive elements are inserted in parallel. To better mimic insertion of cohesive elements in actual simulations, the facets were grouped in 20 sets, inserting 5% of cohesive elements concurrently within each group of facets, using the color model. To color the mesh, we used a greedy algorithm[2]. The number of colors achieved was 10.

We have employed a T6 disk mesh like the one in Figure 3.14 with different discretizations, varying the number of bulk elements from 240,000 to 3,840,000. The results are shown in Table 3.4. Figure 3.15 depicts that the time to insert all cohesive elements varies linearly with the total number of inserted elements. As can be noted, the gain in performance delivered by the GPU implementation is quite significant, even though we are more interested in validating the GPU results. Here we define efficiency as the speedup over the number of CUDA cores.

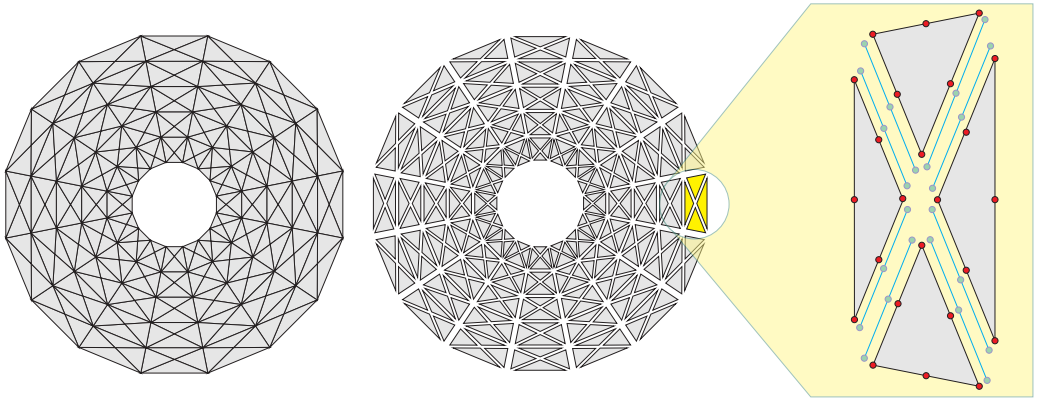

Figure 3.14 – T6 disc mesh used to test insertion of cohesive element decoupled from analysis code.

Bulk elements	Initial nodes	Final nodes	Cohesive elements	CPU Time (s)	GPU Time (s)	Speedup	Efficiency
240,000	481,200	1,440,000	359,400	9.29	0.0407	228.3	47.6 %
960,000	1,922,400	5,760,000	1,438,800	36.946	0.1016	363.6	75.8 %
2,160,000	4,323,600	12,960,000	3,238,200	84.94	0.1935	439.0	91.5 %
3,840,000	7,684,800	23,040,000	5,757,600	150.04	0.3101	483.8	100.0 %

Table 3.4 – Results for insertion of cohesive elements decoupled from analysis code.

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

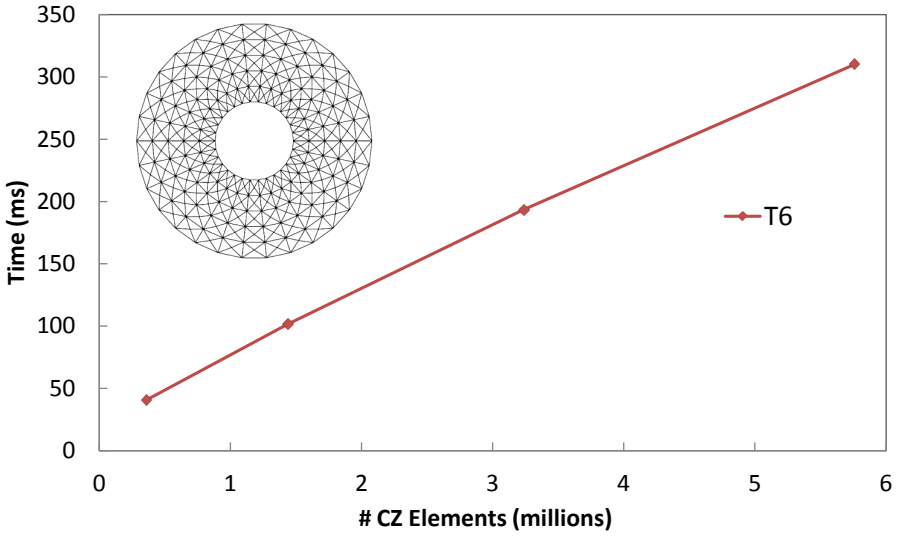

Figure 3.15 – Time for cohesive elements insertion of a T6 mesh.

When duplicating thousands to millions of nodes concurrently, as in this experiment, atomic operations can be quite slow. In order to optimize node duplications in such scenarios, we implemented a new algorithm that greatly speed up the kernel based on shared memory atomics. However, during actual fragmentation simulations, few nodes are duplicated concurrently in a timestep, making the new strategy performance gain negligible.

3.3.2 Fragmentation simulation

The fragmentation simulation was tested using two models: a rectangular specimen and ring specimen. Serial and parallel simulation times as well as kernel times, for the parallel simulation, were measured, and simulation outputs such as number cohesive elements, number of new nodes were analyzed. During the pre-processing phase, nodal perturbation is performed on the end-nodes of a T6 element [17]. Mid-side node positions are linearly interpolated at each facet edge nodes' positions.

Rectangular specimen

The first model tested was a rectangular specimen with an initial notch and refined into T6 (quadratic triangle) elements, as illustrated in Figure 3.16. Fracture propagation is based on mixed-mode fracture and extrinsic cohesive zone model [6, 56, 58]. Initial analysis parameters are as follows: initial strain = 0.015, elastic modulus = 3.24 GPa, Poisson coefficient = 0.35, specific mass

= 1190 kg/m3, fracture energy G_I = 352 N/m, cohesive strength smax = 324 MPa, and shape parameter $\alpha = 2$.

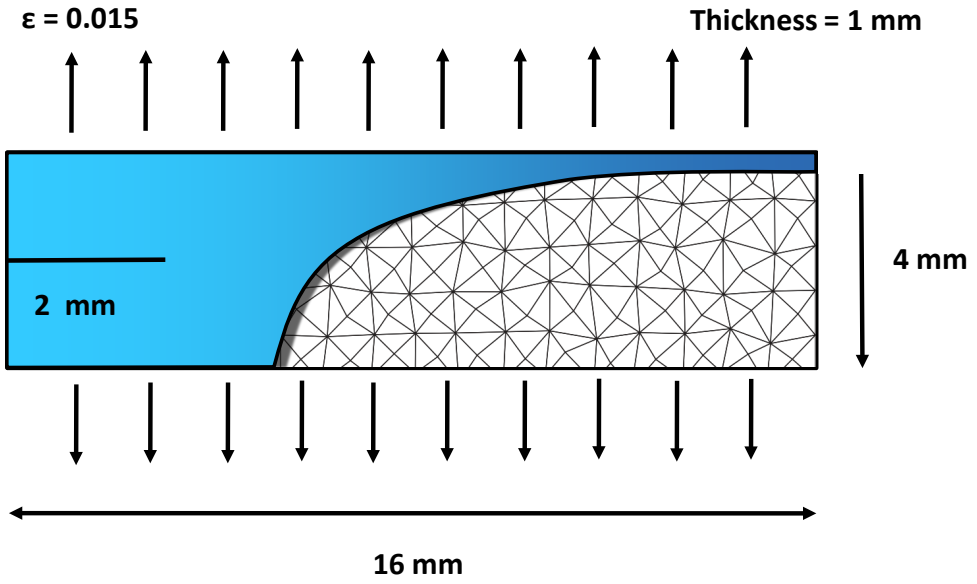

Figure 3.16 – Two-dimensional model of a rectangular specimen with initial notch of 2 mm. Initial strain is 0.015, with node thickness of 1 mm. Model dimensions are 16mm per 4mm.

A first version of the mesh was composed by 74,257 nodes and 36,864 bulk elements. Due to the regular mesh pattern, we employed a simple procedure to subdivide the elements into 8 color groups, which is the optimal. Although we took advantage of the structured mesh to obtain an optimal number of colors, our simulation is able to handle unstructured meshes with a non-optimal number of colors (which is the case in the ring example). Total simulated time is 2 μs, in 10,000 steps of 2 ns. Figure 3.20 shows an extruded 2D model after the simulation and fracture propagation. The refined version of the same model with 295,969 nodes and 147,456 bulk elements was also tested using time steps of 0.5 ns, performing 40,000 simulation steps. Efficiency, defined as the speedup over the number of CUDA cores, was also measured in comparisson with the CPU implementation. In both experiments, stress computation and fractured facets are checked at every 10 simulation steps, and cohesive elements inserted "on-the-fly", when and where needed. Tables 3.5 and 3.6 present results for both the mesh and its refined version. The increase in speedup with model size is expected when running the application on GPU. Figures 3.17 and 3.18 show the final plotted image of the T6 mesh and its refined version at the end of the simulation. The fracture evolved in a straight path in consequence of

the intial notch of the model and the transverse strain applied on the model. Figure 3.19 shows the nodal strain energy wave propagation with the fracture and simulation evolution. Figure 3.20 shows an extruded visualization of the two-dimensional plate.

No. of bulk elements	**No. of nodes**	**No. of new nodes**	**No. of CHZ elements**	**No. of colors**
36,864	74,257	1,901	979	8
147,456	295,969	5,842	2,976	8

Table 3.5 – Simulation and mesh parameters for a T6 mesh and its refined version.

No. of bulk elements	**Timestep**	**CPU time**	**GPU time**	**Speedup**	**Efficiency**
36,864	2.0e-9 s	410.181 s	11.788 s	34.8	7.6 %
147,456	0.5e-9 s	6,537.839 s	153.809 s	42.5	8.9 %

Table 3.6 – Simulation and mesh parameters and results (GPU speedup and GPU and CPU time) for a T6 mesh and its refined version.

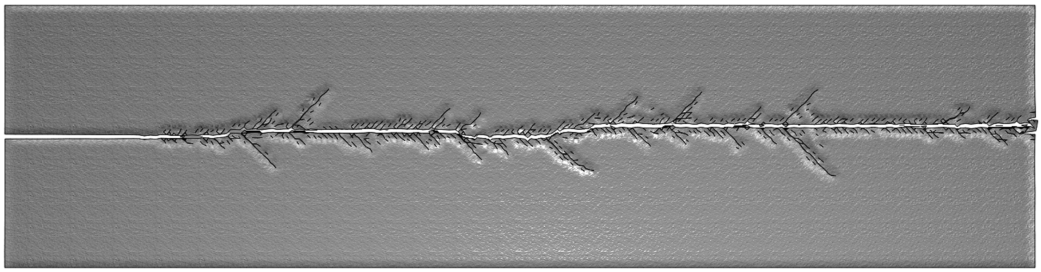

Figure 3.17 – T6 FEM mesh with 36,864 bulk elements at the end of the fragmentation simulation.

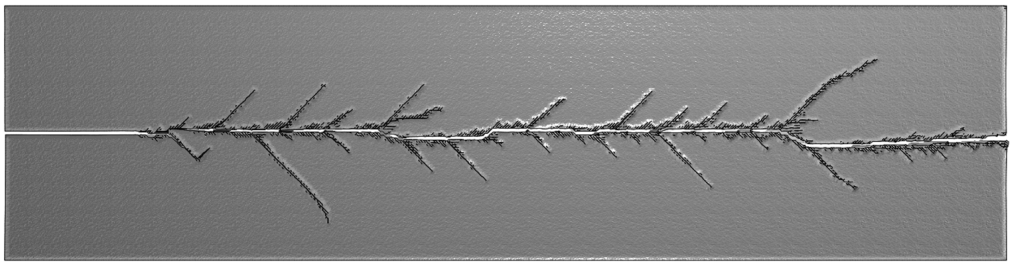

Figure 3.18 – Refined T6 FEM mesh with 147,456 bulk elements at the end of the fragmentation simulation.

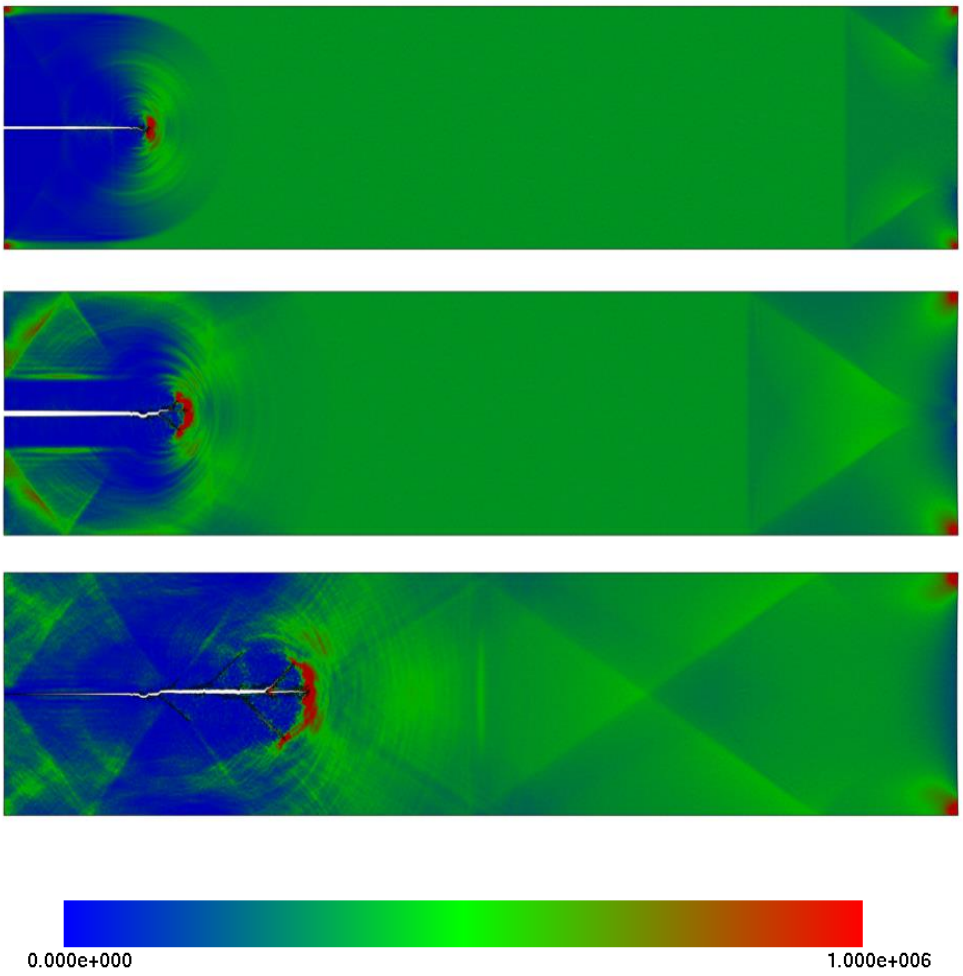

Figure 3.19 – Strain energy evolution with crack propagation.

Figure 3.20 – Extruded view of fragmented 2D plate with 74,257 nodes and 36,864 bulk elements.

Figures 3.21 and 3.22 present results for the portion and average simulation execution times for each kernel for the first T6 mesh model. Stress computation is by far the most expensive, as shown in Figure 3.21. However, Figure 3.21 shows that the kernel responsible computing the internal forces dominates the simulation time with almost twice the time of the stress kernel due to the fact that the internal forces are computed at each time step, while stresses are computed at each ten steps. Another kernel that greatly occupies the simulation time is computing the cohesive forces due to its many numeric computations, although kernel splitting helped increase performance. The node duplication kernel does not occupy a large portion of the simulation because the number of cohesive elements is relatively small. Filtering elements, updating velocities, accelerations, displacements, and nodal masses, and applying boundary conditions are small job kernels with few global memory accesses and coalesce readings that do not have high execution time.

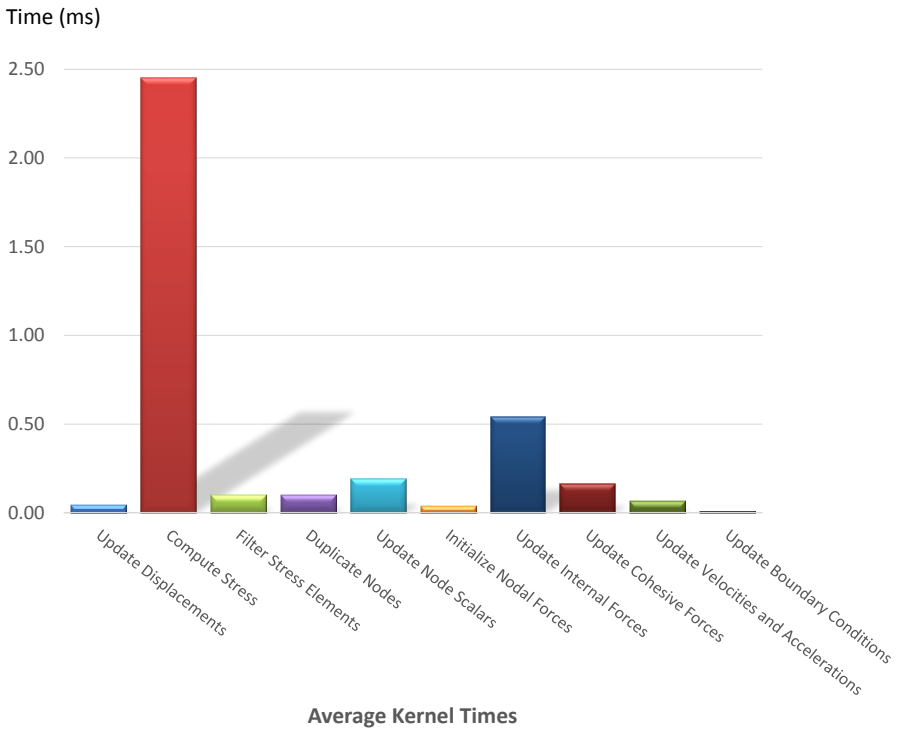

Figure 3.21 – Average time of each *kernel* of the simulation for a T6 mesh with 36,864 bulk elements.

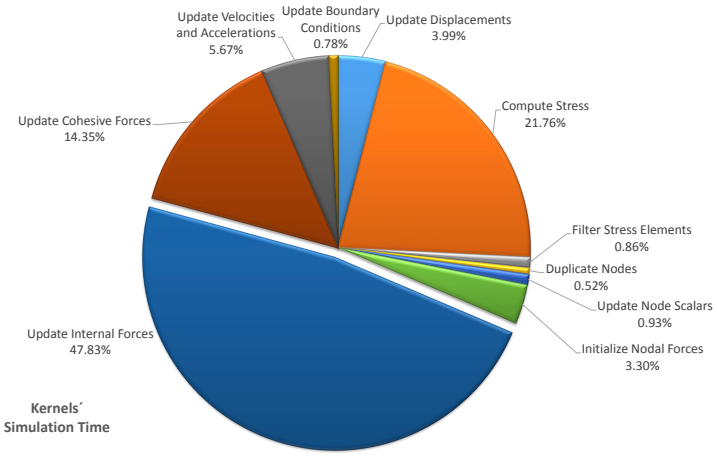

Figure 3.22 – Total time each kernel takes in the entire simulation for a T6 mesh with 36,864 bulk elements.

Ring specimen

The second tested model was a 2D ring specimen with no initial notch, refined into T6 (quadratic triangle) elements, as illustrated in Figure 3.23. We performed a procedure to color the mesh using the greedy algorithm [3]. Fracture propagation is based on mixed-mode fracture and extrinsic cohesive zone model [6, 56, 58]. Initial analysis parameters are as follows: initial pressure = 400 MPa, elastic modulus = 210 GPa, Poisson coefficient = 0.3, specific mass = 7850 kg/m3, fracture energy (GI) = 2000 N/m, and shape parameters (a) = 2.

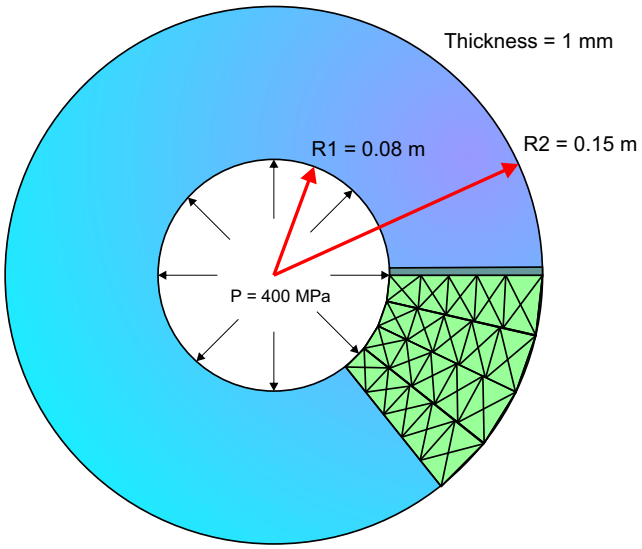

Figure 3.23 – 2D model of a ring specimen. Initial pressure is 400 MPa, with node thickness of 1 mm. The inner radius is 0.08 m and the outer radius is 0.15 m.

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

A first version of the mesh was a 20x160 ring composed by 25,920 nodes and 12,800 bulk elements. The number of colors obtained by a greedy coloring algorithm was 10. A radial pressure was applied on the model. Total simulated time is 81 μs, in 27,000 steps of 3 ns each. The model was refined in from 25,920 nodes to 725,614 nodes. Stress computation and fractured facets are checked at every 10 simulation steps, and cohesive elements inserted as necessary. Tables 3.7 and 3.8 present results for both the mesh and its refined version. Figure 3.24 shows the strain energy evolution throughout the simulation. As expected, the GPU efficiency increases with the number of bulk elements.

Mesh type	**Bulk elements**	**Nodes**	**New Nodes**	**CHZ elements**	**Colors**
Ring 20x160	12,800	25,920	1,816	955	10
Ring 30x240	28,800	58,080	4,923	2506	10
Ring 40x320	51,200	103,040	9,178	4686	10
Ring 96x768	294,912	591,360	59,055	29,615	10
Ring 115x787	362,020	725,614	68,842	34,245	10

Table 3.7 – Simulation and mesh parameters for a T6 mesh and its refined version.

Meshtype	**Bulk Elements**	**Timestep**	**CPU time**	**GPU time**	**Speedup**	**Efficiency**
Ring 20x160	12,800	3e-9 s	375.963 s	12.668 s	29.7	6.2 %
Ring 30x240	28,800	3e-9 s	900.095 s	22.632 s	39.8	8.3 %
Ring 40x320	51,200	3e-9 s	1601.516 s	36.384 s	44.0	9.2 %
Ring 96x768	294,912	3e-9 s	9,355.186 s	200.804 s	46.6	9.7 %
Ring 115x787	362,020	3e-9 s	11,421.568 s	242.802 s	47.0	9.8 %

Table 3.8 – Simulation and mesh parameters and results (GPU speedup and efficiency and GPU and CPU time) for a T6 mesh and its refined version.

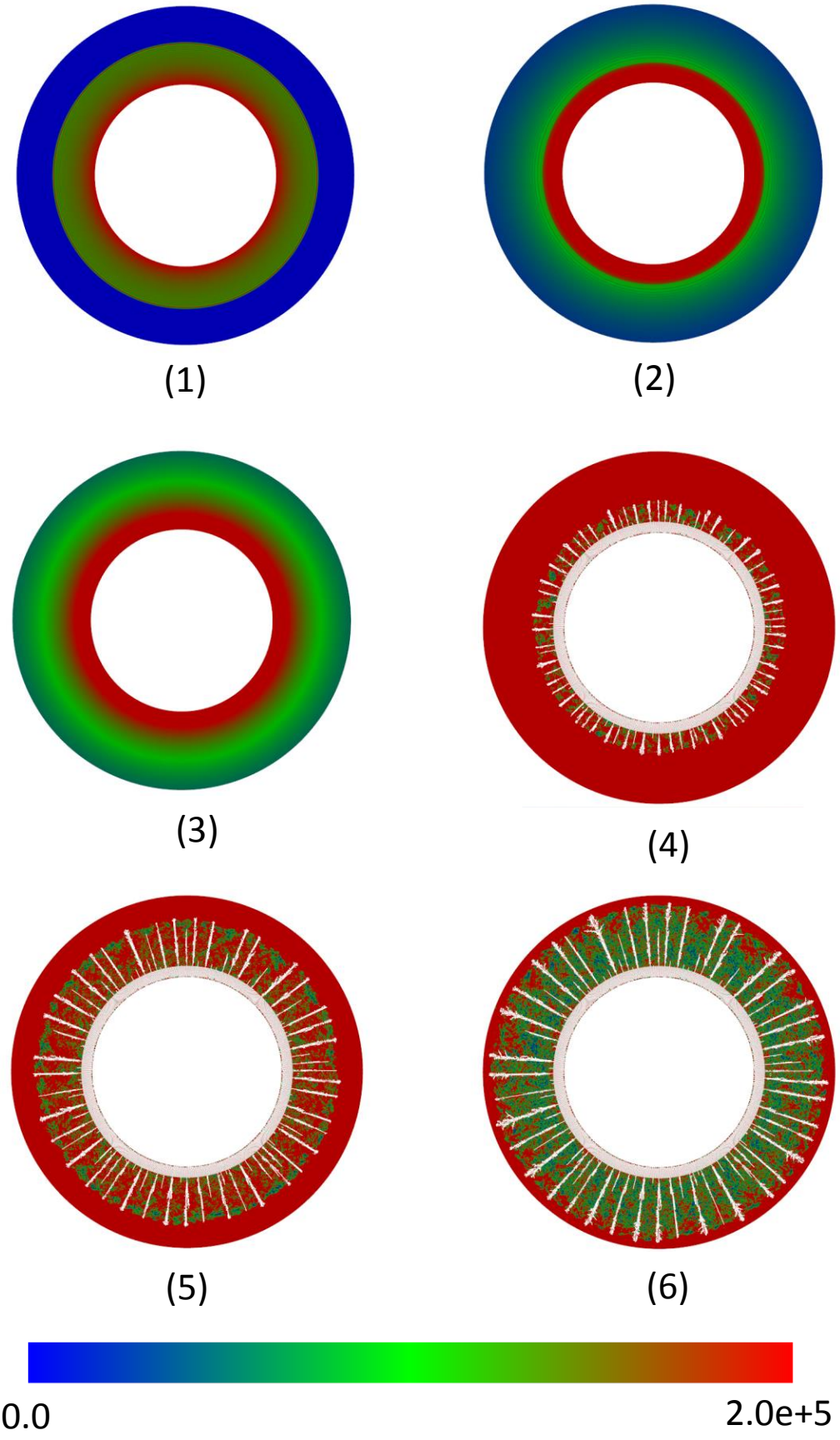

Figure 3.24 – The figure shows a T6 FEM mesh with 362,020 bulk elements and the strain energy's evolution with the crack propagation for times 5 μs (1), 20 μs (2), 25 μs (3), 50 μs (4), 60 μs (5), and 68 μs (6).