6
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 provided by CUDA, it is important to keep in mind the device architecture and programming paradigms discussed in Chapter 3.2, 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 easily extended to 3D meshes using a modified version of the previous discussed data structure. We use constant memory for storing material attributes that are constant during the entire simulation. Cache hits when fetching these attributes during stress and other force computations will help to increase performance since threads in the same warp access the same value at the same time.

6.1
Coloring model

In this discussion, we consider the implementation of T6 meshes. The first parallel procedure to be discussed is updating the node attributes. In our case, we are focused on updating each nodal 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 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. 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 use the Welsh Powell algorithm (26), a greedy algorithm to color the mesh bulk elements. Each element represents a graph node and adjacent elements are connected by a graph edge. We order the graph nodes (elements) in decreasing order of degree to obtain the closest optimal solution. Table 6.1 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 6.1 illustrates the colored mesh and its use in kernel calls and updating the nodal masses.

```
for $c = 1 \to numColors$ do
  $numThreads \leftarrow numElements(c)$
  $KernelCall <<< numThreads >>> (c)$
end for
```

Table 6.1: Kernel subroutine call algorithm using mesh coloring

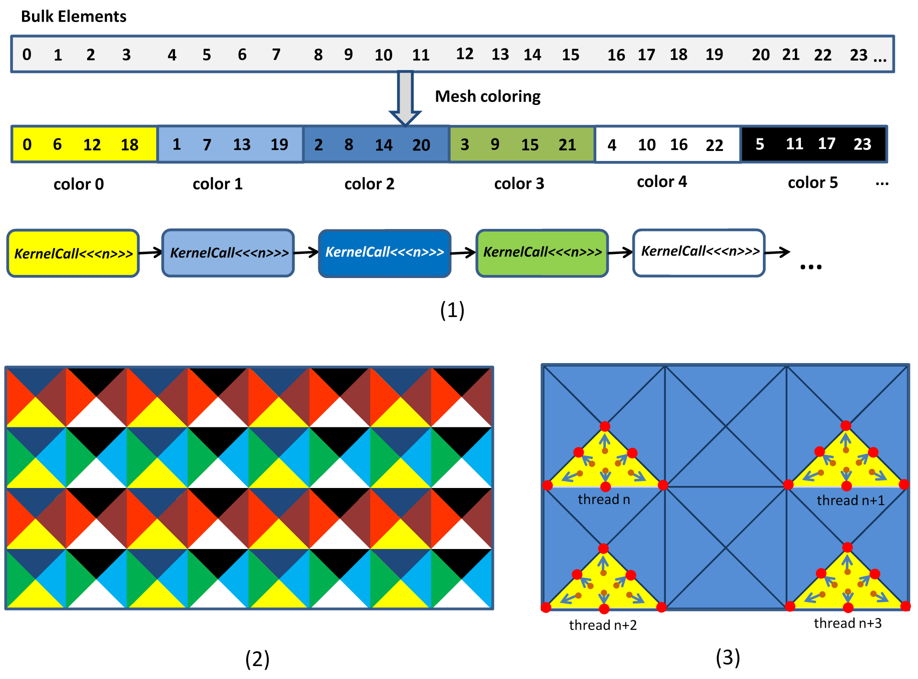

Figure 6.1: (1) Bulk elements are re-arranged in color groups (preferable balanced) and the same kernel per color group is called to avoid writing conflicts. (2) Example of a colored T6 structured mesh (3) and using the colored mesh to update nodal masses of the group of elements in the current color in parallel.

6.2 Pre-processing and update

A pseudo-code of the parallel simulation is shown on Table 6.2. 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.

```
$ComputeMassMatrix <<< numElem >>>$
$ComputeStiffnessMatrix <<< numElem >>>$
for $c = 1 \rightarrow numColors$ do
    $numGroupElem \leftarrow numElem(c)$
    $UpdateNodalMass <<< numGroupElem >>>$
end for
current step $\leftarrow 0$
while $currentstep <= maximumstep$ do
    $UpdateDisplacements <<< numNodes >>>$
    if current step == check step then
        $ComputeStressesAtGaussPoints <<< numElem >>>$
        for $c = 1 \rightarrow numColors$ do
            $numGroupElem \leftarrow numElem(c)$
            $ComputeNodeStresses <<< 12 * numGroupElem >>>$
        end for
        $CheckFracturedFacets <<< numNodes >>>$
        $FilterFracturedFacetElements <<< numElem >>>$
        $numFracElem \leftarrow CompactFracturedFacetElements$
        if Current Fractured Facets $> 0$ then
            for $c = 1 \rightarrow numColors$ do
                $numGroupElem \leftarrow numFracElem(c)$
                $InsertCohesiveElements <<< numGroupElem >>>$
            end for
            for $c = 1 \rightarrow numColors$ do
                $numGroupElem \leftarrow numElem(c)$
                $UpdateNodalMass <<< numGroupElem >>>$
            end for
        end if
    end if
    for $c = 1 \rightarrow numColors$ do
        $numGroupElem \leftarrow numElem(c)$
        $ComputeInternalForces <<< 12 * numGroupElem >>>$
    end for
    $ComputeCohesiveSeparations <<< numCohElem >>>$
    $ComputeCohesiveTractions <<< 3 * numCohElem >>>$
    $numElemCoh \leftarrow CompactBulkElementsWithCohesiveElements$
    for $c = 1 \rightarrow numColors$ do
        $numGroupElem \leftarrow numElemCoh(c)$
        $ComputeCohesiveForces <<< numGroupElem >>>$
    end for
    $UpdateVelocitiesandAccelerations <<< numNodes >>>$
    $UpdateBoundaryConditions <<< numNodes >>>$
    current step $+= 1$
end while
```

Table 6.2: Parallel Fracture Algorithm

Figure 6.2 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 4-1. 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.

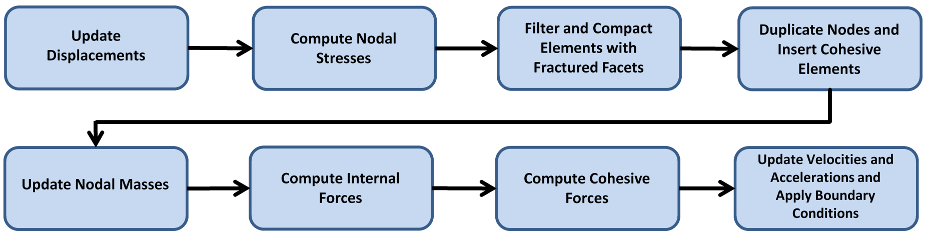
Update Displacements
Compute Nodal Stresses
Filter and Compact Elements with Fractured Facets
Duplicate Nodes and Insert Cohesive Elements
Update Nodal Masses
Compute Internal Forces
Compute Cohesive Forces
Update Velocities and Accelerations and Apply Boundary Conditions

Figure 6.2: Fracture and fragmentation simulation loop.

6.3 Stresses

Before checking fractured facets, the next procedure is responsible for computing the stresses and strains on the nodes by first calculating them at the Gauss points for each element, multiplying its respective matrix with the element shape function as showed in Equation 4-4 at Chapter 4.6, 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. 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 4-4, we can observe that the second kernel performs several global memory accesses since it accumulates element stresses and strains on its nodes by fetching from the element stress and strain matrix at the Gauss points, computed on the previous kernel. In a 2-dimensional T6 mesh case, this is a 3x4 matrix. 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 6.3 illustrates the stress kernel division and Figure 6.4 illustrates the second kernel procedure.

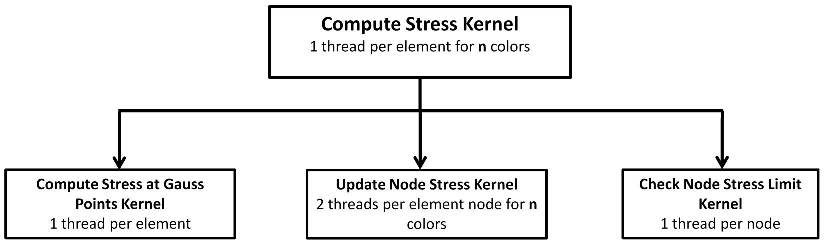
Compute Stress Kernel
1 thread per element for n colors
Compute Stress at Gauss Points Kernel
1 thread per element
Update Node Stress Kernel
2 threads per element node for n colors
Check Node Stress Limit Kernel
1 thread per node

Figure 6.3: Splitting the kernel that computes stress and strain into simpler kernels.

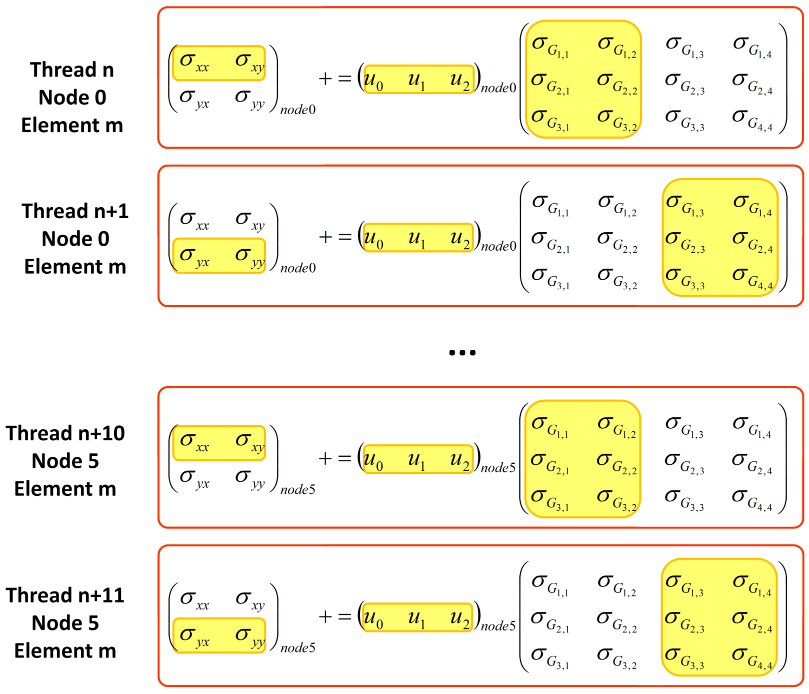

Figure 6.4: To accumulate the stresses and strains on the nodes, we launch 12 threads per element, where each thread will accumulate part of the stress and strain matrices by fetching from the element shape functions and from the stress and strain at the Gauss points.

6.4 Insertion of cohesive elements

Once fractured facets are identified, new cohesive elements must be inserted in the mesh. When inserting cohesive elements, launching one thread for each element can result in idle kernels because there are few elements that contain fractured facets. In order to solve this matter, an additional kernel is used before inserting cohesive elements. This additional kernel filters only the elements that contain fractured facets by launching one thread per element and checking its 3 facets for possible fractures as discussed in Chapter 4.6. 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.

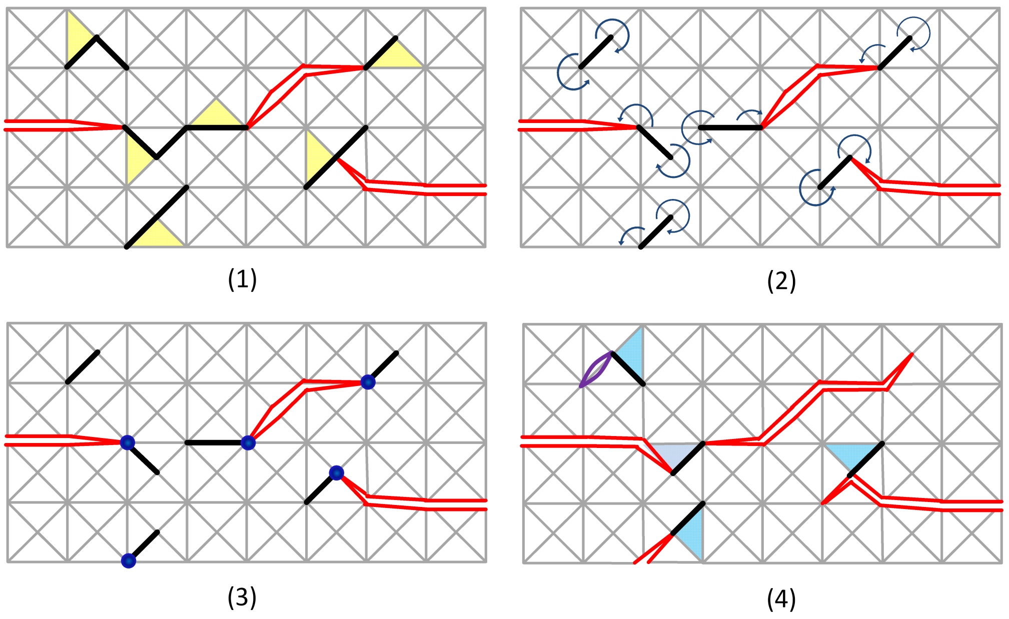

Figure 6.5: Cohesive elements insertion on a T3 mesh. (1) Mesh with initial cracks and facets that need to be fractured. Coloring is used to avoid duplicating nodes of elements that share nodes in parallel. (2) From each facet node belonging to the element in the current color group, the algorithm traverses through its incident elements. (3) Nodes that need duplication. (4) T3 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 6.5 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 4.6 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 6.3 presents the parallel cohesive element insertion algorithm.

```
1: $e \leftarrow bulkelement$
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:         $e \leftarrow$ next element
5:     end for
6:     if element adjacent to $e$ is reached again then
7:         continue
8:     end if
9:     $newNodeIndex \leftarrow atomicAdd(globalNodeCounter, 1)$
10:    $nodeList[newNodeIndex] = n$
11:    for each incident element of $n$ starting with $e$ do
12:        Replace $n$ index with $newNodeIndex$
13:        $e \leftarrow$ 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 6.3: 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.

6.5 Internal Forces

Computing the internal forces is perhaps one of the most expensive kernels and occupies a large portion of the simulation as it is executed every time step. We launch 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 naïve 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. Thus, we need to launch one thread for matrix line instead of launching one thread per bulk element matrix. This greatly reduces the number of global accesses to a number of 156. The performance, however, still does not reach optimal expectations. Each thread is now responsible for computing the product of one line of the stiffness matrix with the displacement vector. Consequently, the number of global memory accesses is reduced to 24 as well as the kernel's effort. Launching one kernel 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. Even with one thread per matrix line, we can still use the shared memory to store the displacement vector. Different threads will then use the same vector. 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). Each thread loads one value from the displacement matrix and are synchronized. Notice that each group of 12 threads will load its respective element displacement matrix. This strategy reduces the reading number of stiffness-displacement global memory accesses to a total of 13 for each thread (instead of 24). Figure 6.6 illustrates this strategy using two matrices per block.

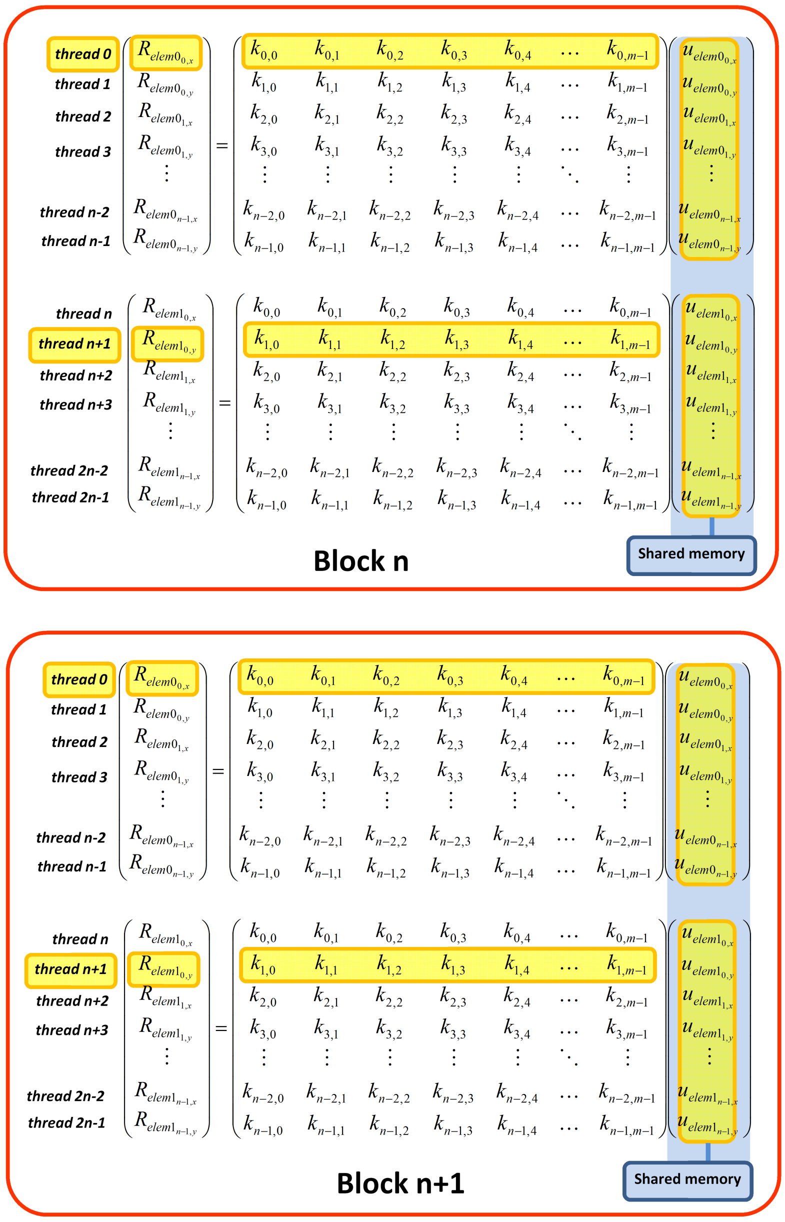

Figure 6.6: When computing internal forces, a thread per stiffness matrix line is launched using the color model. In this example, two elements per block is used.

6.6 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 does 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 element would be ideal. However, with many arithmetic operations, registers, and color model 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, we need to write 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 6.7

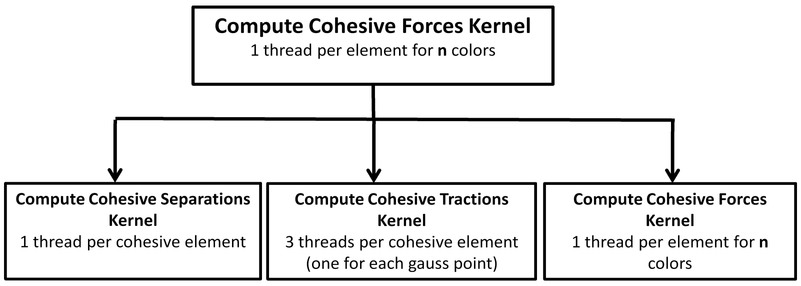
Compute Cohesive Forces Kernel
1 thread per element for **n** colors
Compute Cohesive Separations Kernel
1 thread per cohesive element
Compute Cohesive Tractions Kernel
3 threads per cohesive element (one for each gauss point)
Compute Cohesive Forces Kernel
1 thread per element for **n** colors

Figure 6.7: 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.

6.7 Overview

Analyzing the parallel simulation steps, we can conclude that there are two kernels that greatly occupy the simulation time. Computing the cohesive forces requires many arithmetic operations, while computing the internal forces requires numerous global memory accesses. Splitting the kernels into simpler ones, distributing jobs among threads, and using texture memory greatly increase the kernels' performance, although they still occupy a large portion of the simulation time. Non-linear simulations would need to compute the stiffness matrix at every simulation step instead of pre-processing it. Although it would greatly reduce the program's performance, the GPU speedup would also increase. Computing the stresses and strains is the most complex and expensive kernel, with a larger processing time and more numerous arithmetic operations than computing the internal and cohesive forces, but with the advantage of not having to launch it at every simulation step. Kernels that update displacements, velocities and accelerations, boundary conditions, and nodal masses are light kernels as they perform few and simple read-and-write operations with no warp divergence and coalesced reading for half-warps. Shared memory rarely fits the simulation, working more as a cache to optimize it.