1
Introduction

Fracture, branching, and fragmentation simulations of large-scale finite element meshes have a broad range of engineering applications. To achieve realistic and more accurate results, there is a need to employ highly discretized models, thus requiring a large amount of computational resources. In order to accelerate finite element analysis, current parallel environments are based on distributed memory architectures. In this scenario, each processor (or small group of processors) of a computing node has private access to a region of the global system memory. Processors on different nodes communicate among themselves by sending messages over a network. Different parallel finite element systems with support for distributed mesh representation have been proposed [3, 4].

Fracture and fragmentation phenomena can be modeled using the cohesive zone model (CZM) [5–8], which can be simulated by enrichment functions [9] or inter-element techniques [8, 10]. Our application is based on the experiments by Sharon and Fineberg ([11, 12]) and address dynamic cracking instability—thus it is not a simple issue as there are several papers in the literature addressing the problem. In the inter-element technique, cohesive elements are inserted between bulk (volumetric) finite elements (e.g. triangular elements in 2D case and tetrahedron elements in 3D case). Two types of cohesive elements are distinguished: instrinsic and extrinsic. In the intrinsic case, cohesive elements are inserted before the simulation starts. In the extrinsic case, cohesive elements are inserted adaptively during the course of the simulation, when needed and where needed. Because parallel fracture, microbranching, and fragmentation simulation using the extrinsic cohesive zone model [13] requires cohesive elements to be adaptively inserted during the simulation, challenges for parallelization emerge because mesh consistency must be ensured among partitions [14]. In such simulations that require a topological data structure (including our case), inserting cohesive elements between bulk elements during the course of the simulation requires a change in element connectivities and adjacency information, such as duplicating nodes and updating element neighbors. The convergence of the simulation could require unstructured meshes [15, 16], and therefore node perturbation is performed to partially address the problem [17].

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

We investigate and describe mapping and parallelization techniques for two and three-dimensional fracture, branching, and fragmentation simulation of finite element meshes on a GPU using NVIDIA's CUDA (Compute Unified Device Architecture) framework . We propose a simple but effective data structure for performing all data-parallel computations and algorithms for 2D and 3D models using triangular and tetrahedral meshes. The previously established coloring method for FEM meshes, in which the mesh is colored so that no volumetric finite element of the same color shares a vertex, is also used to minimize concurrency when non-adaptive simulation is performed. In adaptive simulation, we use the conventional gather technique, or node traversal strategy, where incident elements to a node are traversed and quantities are gathered and accumulated to their common node. Parallel techniques are presented for the numeric analysis code and for updating the FEM mesh when cohesive elements are inserted during the course of the simulation. In our experiments, we have employed a two and three-dimensional microbraching analysis, adaptive and non-adaptive simulations, but our framework for parallel simulation has the potential to support other separation phenomena simulations.

The main contributions of this work are:

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

displacements, stiffness matrices of bulk elements, and cohesive tractions between cohesive element);

- Establishment of a framework for dynamic adaptation of the mesh in the GPU, in a consistent fashion, by inserting cohesive elements on fractured facets (duplicating nodes and updating connectivity "on the fly") and refinining and coarsening the mesh by removing and adding nodes and bulk elements to the mesh;
- Improved reduced communication among nodes in clustered machines for distributed GPU simulations by duplication of *ghost* nodes;
- Analysis of randomness affected by the GPU adaptive simulation when inserting bulk elements in a random order;
- Development of an effective nodal update scheme using gather and scatter techniques necessary for GPU parallelization. Gather techniques were employed to insert cohesive elements on adaptive simulations and running adaptive simulation. While scattering techniques, based on coloring, were employed to compute nodal stress, nodal mass, internal forces and cohesive forces on non-adaptive simulations;
- Development of parallel techniques to map fragmentation algorithms to GPUs, such as splitting of *kernels* into simpler ones, distributing jobs among threads, taking advantage of memory coalescence (consecutive threads reading consecutive memory addresses), and using texture memory to increase *kernel* performance.
- Combination of our GPU fracture and fragmentation simulation with Muller et al's Position-based Dynamics framework for breaking quasi-brittle material objects, for animation purposes.

It is important to identify the nomenclature of entities, or mesh "parts", that compose our scientific models. Our mesh is discretized into volumetric finite elements in 2 (linear or quadratic triangles) or 3–dimensions (linear tetrahedra) called bulk elements. Elements can also be of interface type called cohesive elements. Cohesive elements are inserted between the volumetric element facets and impose a traction–separation relationship, as we will address later. The elements are made up of nodes that also compose the mesh. They impose displacement, velocity, acceleration, and cohesive and internal forces, for example.

The remainder of this document is organized as follows. Chapter 2 explains the dynamics of fracture and fragmentation simulations. Chapter 3 discusses the 2-dimensional fragmentation simulation and the data structure used, along with experimental results. Chapter 4 extends the simulation to an adaptive approach using refinement and coarsening to speed up the simulation and save memory consumption. It also provides a review of the proposed data structure and provides numerical results. Chapter 5 discusses the distributed 3-dimensional fragmentation simulation and the parallelization technique between different machines, as well as the data structure used and results. Chapter 6 shows that our fracture and fragmentation simulation can be combined with physics-based methods to produce interesting animation of brittle materials. Finally, concluding remarks and directions of future work are drawn in chapter 7.

1.1 Related work

Over the past decades, several works adderessed the problem of fracturing of scientific models using cohesive zone model [5–8]. Several authors incorporated the use of GPUs and computer clusters to accelerate finite element simulations and obtaining faster results [18–24]. In recent years, meshless or finite element fracture simulations are being widely used in the field of Computer Animation [25–29]. In the next two sections, we present a brief review of the scientific works in the field over the last years.

1.1.1 On GPUs and High Performance Computing

Various researchers investigated parallel simulations to improve the performance of finite element analysis, addressing the use of a distributed memory architecture (See references [3, 4] and citations within). Fracture, microbranching, and fragmentation simulation introduce new challenges since modeling of interface elements is required. Dooley et al. [13] have presented a parallel implementation of dynamic fracture simulation on extrinsic cohesive models using ParFUM, a parallel framework specifically developed for parallel finite element applications [30]. The cohesive elements require an external activation criteria and do not feature the initial elastic slope present in the intrinsic model. The interface elements are present at all facets before the start of the simulation. Nodes at all facets are duplicated from the beginning, but the traction separation relationship is not activated until the external stress-based criteria is met. In this way, their work is more similar to an intrinsic implementation because the mesh connectivity does not change as the problem evolves. Radovitzky et al. [31] have opted for parallelizing extrinsic fracture and fragmentation simulations based on a combination of a discontinuous Galerkin formulation and cohesive zone models. Like Dooley et al. [13], cohesive elements are pre-inserted throughout the mesh. Espinha et al. [14] developed ParTops, a topological distributed mesh representation for parallel dynamic simulation of fragmentation phenomena based on the extrinsic cohesive zone model.

Several references have also explored the use of GPU for numerical simulation and parallelization techniques other than fracture or fragmentation. Each reference provides a number of similar and different parallelization strategies to handle scientific simulations that use large-scale data, some of which are used in this work. Boltz et al. [32] show that high-intensity numerical simulation can be performed efficiently on the GPU using sparse matrix conjugate gradient solver and a regular-grid multigrid solver, a technique widely used in scientific areas and real-time applications, such as mesh smoothing and parametrization, and fluid solvers and solid mechanics. Wu and Heng [18] present a deformation model on soft tissue, called hybrid condensed finite element model, based on the volumetric finite element method accelerated by the GPU. Krakiwsky et al. [19] investigate acceleration of Finite-Difference Time-Domain method (FDTD) using the GPU. Tejada and Ert [33] use physics simulation and volume visualization of tetrahedral meshes on graphics hardware to build a physically-based deformation system based implicit solver. Since 2008, several works on GPU acceleration of scientific simulations have been published. Taylor et al. [20] present a fast GPU solution scheme for finite element equations used in nonlinear total Lagrangian explicit finite element formulation for surgical simulation. Göddeke et al. [34] explores parallelism for finite element simulations based on parallel multigrid solvers and Anderson et al. [35] developed a general purpose molecular dynamics code running entirely on a GPU. Rodriguez-Navarro and Susin [36] have implemented cloth simulation on the GPU using finite element method for trianglular meshes. Previous studies on FEM in GPUs also focused on solving large sparse linear systems [36–38] using CUDA. They have explored the strategy of mesh coloring for minimizing conflicts, thus avoiding excessive use of CUDA atomic operations, which usually degrade performance. A GPU approach for geometric multigrid solvers on finite elements for unstructured grid prblems was done by Geveler at al. [39], in which their GPU implementation is based on cascades of sparse matrix-vector multiplication by applying strong smoothers. Komatitsch et al. [40] have used CUDA to speedup numerical simulation of seismic wave propagation resulting from earthquakes. Liu et al. [41] use the GPU and CUDA to perform a fast Finite Element dynamic deformation simulation. Kindratenko et al. [42] present challenges with building and running GPU clusters for high-performance computing environments along with discussion on their experiments with GPU programming toolkits, and their interoperability with other parallel programming APIs. Godel et al. [21] use cluster of GPUs through CUDA to parallelize Maxwell´s equations in the time domain using a Discontinuous Galerkin Finite Element Method (DG-FEM) for spatial discretization. Their parallel implementation tends to minimize overhead and improve efficiency through assynchronous data transfer. Kakay et al. [43] implement their finite element micromagnetic simulation in the GPU, demonstrating a high speed performance compared to CPU multi-core implementation. Ren et al. [22] model and analyze power performance of parallel 3D Finite Element mesh refinement on CUDA and MPI architecture using multi-core CPU and GPU cluster, also proposing parallelization techniques for both. In recent works on FEM, Markall et al. [23] use finite element advection-diffusion solver to demonstrate that FEM implementations on many-core (GPU) and multi-core (CPU) architectures differ if their performance potential is to be obtained. Different data structures are to be employed depending on the architecture and algorithms. They use coloring strategy to avoid concurrency and use of atomic operations. Zegard and Paulino [24] investigate feasibility of finite element method and topology optimization for unstructured meshes in GPUs and discuss challenges in parallel implementation. This list of papers is incomplete and by no means exhaustive. The field of GPUs for scientific computing is quite vibrant and new contributions are continuously being reported.

Fracture simulation using the many-core GPU environment and adaptive finite element mesh operations are not well documented in the literature. To the best of our knowledge, adaptive dynamic fracture simulation on GPUs has not been investigated. Adaptivity has been explored in other fields, for example cartesian meshes are generated on the GPU for computational fluid dynamics applications resulting in speedup of up to 36 for large meshes [44]. Dynamic fracture with the extrinsic cohesive zone model on a uniform mesh is investigated and implemented in [45], and serves as the motivation and foundation on which the proposed adaptive GPU fracture is based.

1.1.2 Fracture in Computer Animation

Since 1988, there were countless publications in the field of Computer Graphics addressing the problem of fracture simulation applied in Computer Animation. Among the first ones to propose a fracture model specifically designed for computer graphics were Terzopoulos et al. [46] in 1988 and Norton et al. [47] in 1990. While Norton et al. use network of point masses connected by springs to represent physical objects that can bend and break, other methods are based on continuum mechanics to compute stresses and determine fracture propagation direction [26]. In 2001, Müller et al. [48] proposed a stable hybrid real-time method to simulate deformable and fracture of brittle materials, emplying a continuum method for the deformation model and stress-based criteria for fracture model. They have integrated the techniques into physically-based animations for objects represented by volumetric tetrahedral meshes. In 2005, Pauly et al. [49] presented a new framework for fracturing plastic and elastic materials in meshless animation. They create and maintain fracture surfaces by adding surface samples during crack propagation while initially sampling the mesh of volumetric domain. They alspo adapt dynamically the shape functions whenever new crack surfaces are created. Zheng and Doug [50] proposed, in 2010, the first physically-based approach for automatic synthesis of synchronized brittle fracture sounds for computer animation. Their model approximates the sound of brittle fracture by that of time-varying rigid-body sound moudels. The field of Computer Animation is also quite vibrant and new contributions are recently being reported.

In more recent work, finite elements are used to represent fracture of volumetric and thin sheet brittle materials in computer animation. In 2013, Busarye et at. [25] study a more realistic animation of thin plate fracture by performing a stress relaxation method to handle multiple fracture cuts in a single step. To produce fracture details, they use a fracture-aware remeshing scheme based on constrained Delaunay triangulation. Müller et al. [26] proposed a new and fast real-time method to simulate dynamic destruction of large and complex objects, representing the geometry as compounds of convex shapes. They decompose the model using their Volumetric Approximate Convex Decomposition (VACD) to apply fracture patterns locally at low cost. In 2014, many contributions in the field of Computer Animation were also presented. Pfaff et al. [27] proposed an adaptive method propagation in thin sheets, refining the triangular mesh where cracks are likely to start or advance. Refinement and coarsening allows an efficient simulation by reducing the total number of nodes and elements of the mesh. Koschier et al. [28] use a novel reversible tetrahedral mesh refinement scheme for adaptive simulation of brittle fracture of solid objects. Chen et al. [29] adaptively refine and coarsen a fracture surface into a detailed one, based on a discrete gradient descent flow.

1.2 Many-core devices

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

CPUs and GPUs use multicore and many-core characteristics, respectively. The multi-core approach (CPU) seeks to maintain the execution speed of sequential programs while moving into multiple cores. In contrast, the many-core (GPU) trajectory focuses more on execution throughput of parallel applications [51]. Graphics chips can more easily achieve higher memory bandwidth than CPU chips, depending on how memory is accessed. Each multiprocessor have access to the GPU's global memory (DRAM), which differs from CPU's motherboard DRAM in computing in that they are the frame buffer memory that are used for graphics, such as video images and texture information. However, for general purpose computing, they work as off-chip, very-high-bandwidth memory with more latency than typical system memory. In CPUs, each core is independent and can execute several instructions for various processes (Multiple instructions/multiple data - MIMD). Cores are organized into warps, each one of which is assigned a warp number. All cores in warp 0 execute the same set of instructions, all cores in warp 1 execute the same instructions, and so on. In GPUs, each core in a given warp executes the same set of instructions each on different data (Single instruction/Multiple data - SIMD). All cores have access to global memory (off-chip), which has a higher bandwidth than CPU global memory, but is slower to access than on a CPU system (i.e., higher latency). Shared memory (on-chip) is extremely fast to access, but only the cores within the multiprocessor can share it. Thus, applications can be optimized by ensuring that this memory is used properly. Since each core in a given warp is executing the same set of instructions, the cores are not independent and they communicate via their shared memory. Several aspects not encountered in CPU programming are issues that arise in GPU programming that can fatally reduce the application's performance. Such aspects include memory access (both off-chip and on-chip), branch divergence, and how to fully occupy the GPU with jobs.

1.2.1 GPU Architecture

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

When programming in a CUDA-capable GPU, one must keep in mind its architecture and parallelism properties for they have an important role and impact on the performance of a GPU simulation. The architecture of a modern GPU is organized into a set of multiprocessors (SMs), each of which contains a number of streaming processors (SPs), as shown in Figure 1.1. The device memory space is organized as follows. Global memory is an off-chip memory with slow access that can be accessed by all threads. Texture access is cached, as well as the constant memory, which is also read-only and can be accessed by all threads. Shared memory is an on-chip memory space that can be accessed by all threads in a block. Threads within a thread block can use shared memory to cooperate amongst themselves, and this is a good alternative for optimizing a program. Finally, each thread has its own memory space known as local memory which resides on global memory. Figure 1.2 illustates the CUDA memory hierarchy.

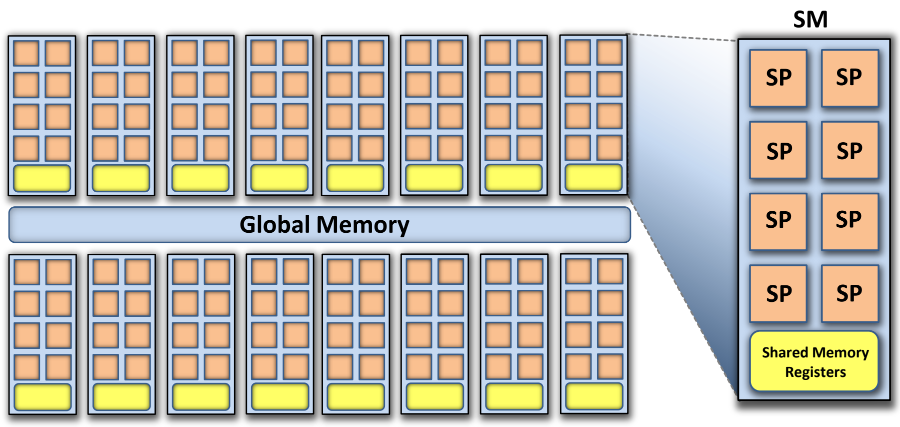

Figure 1.1 – Diagram of a G80 architecture with 16 SMs and 128 SPs, based on the figures presented in [1].

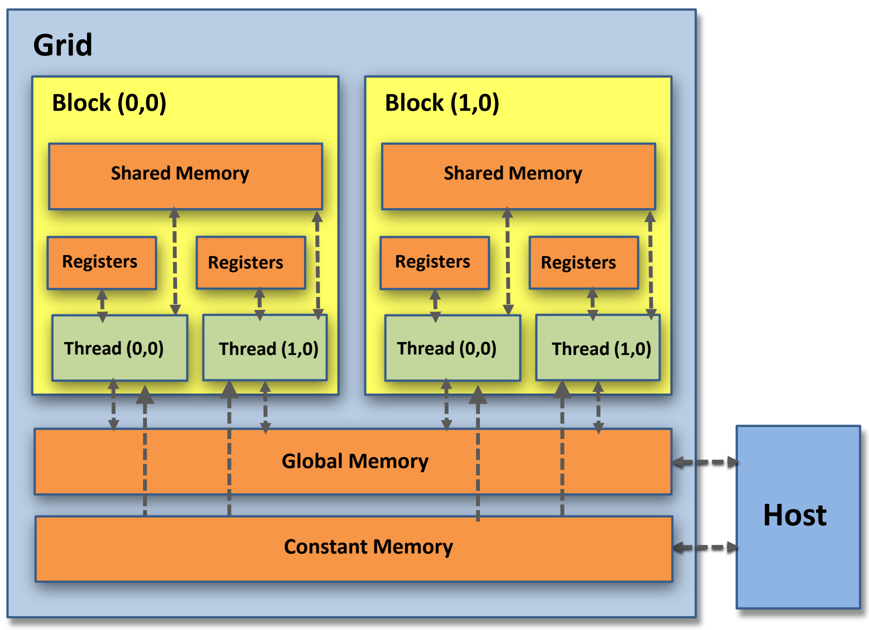

Figure 1.2 – CUDA memory hierarchy, based on the figures presented in [1].

1.2.2 Optimization

Next, we highlight some CUDA programming issues. In a kernel execution, like CPUs, each thread must write on a different memory space as they are being executed concurrently, to avoid writing conflicts. All threads within a warp execute the same instruction (SIMT architecture), so it is suggested that there should be no conditionals and loops that lead to thread divergence within the warp. When multiple global memory accesses are coalesced into a single memory transaction by the device (i.e. proper memory access alignment and contiguity), we achieve a coalesced reading. When seeking a performance optimization in a kernel execution, it is best to minimize global memory accesses, since they are slow. When access to global memory is mandatory, coalesced reading helps increase the simulation performance. Similar to blocking in CPUs, coalesce means memory accesses in GPUs should be made as coherent as possible, with all threads in a warp collectively reading a block of adjacent words, which sometimes is difficult to achieve in actual simulation. Also, it is important to avoid bank conflicts when using shared memory. Shared memory is divided into banks, and if multiple threads in the same half-warp access the same bank, access must be serialized. To avoid bank conflicts, all threads of a half-warp must access different banks or all of them must read the identical address. Finally, in order to reach an optimal kernel performance, one has to maximize thread occupancy, defined as the ratio of the number of resident warps to the maximum number of resident warps, depending on the GPU architecture [1]. The ways occupancy can be maximized include minimizing the number of registers per thread and shared memory.

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

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

Please refer to [1, 51, 52] for additional background on CUDA and many-core devices.