Original Article

International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
1–14

DOI: 10.1177/1094342015593395
hpc.sagepub.com
SAGE

High-performance simulation of fracture in idealized 'brick and mortar' composites using adaptive Monte Carlo minimization on the GPU

Rone Kwei Lim[1], J William Pro[2], Matthew R Begley[2,3], Marcel Utz[4] and Linda R Petzold[1,2]

Abstract

Simulation of the nonlinear mechanical response of materials with explicit representation of microstructural features is extremely challenging. These models typically involve a very large number of degrees of freedom, and are prone to convergence difficulties when searching for roots to nonlinear equilibrium equations. We focus on an idealized material model that is motivated by the microstructure of synthetic nacre: individual 'bricks' (representing ceramic platelets) interact through nonlinear cohesive springs (representing a small volume fraction of polymer that bonds the platelets). The model simulates composite fracture through rupture of the cohesive springs. The problem is cast in terms of energy minimization and is essentially described by 'nearest neighbor' interactions. The principal focus of this paper is to illustrate the computational gains achievable by the strategic marriage of robust, global Monte Carlo minimization algorithms to the graphics processing unit architecture, and to describe how they were realized on the Nvidia GPU. Results comparing the computation times for graphics processing unit and central processing unit implementations demonstrate that a new adaptive version of the simulated annealing algorithm yields a speedup of approximately 5 times, whereas the graphics processing unit implementation yields a speed-up of about 16 times over conventional four-core central processing unit implementations. The resulting speed enhancement for adaptive graphics processing unit minimization of a factor of 80 enables a far broader range of simulations than has previously been possible. Simulations involving as many as 300,000 bricks can be performed in hours, as compared to weeks required by central processing unit implementation. Many aspects of this approach are translatable to other physical problems involving energy minimization in systems with large numbers of degrees of freedom.

Keywords

Graphics processing unit, fracture simulation, nacre, brick and mortar structures, Monte-Carlo, Compute Unified Device Architecture

1 Introduction

Simulations of fracture that explicitly account for material microstructure can be extremely expensive, owing to the fact that large numbers of degrees of freedom are necessary to capture the effect of individual microstructure features. An excellent example of this challenge is the simulation of fracture in nacre and its synthetic analogues, which consist of small ceramic platelets bonded together with a very small volume fraction of polymer. The mechanical response of these materials is strongly influenced by the dimensions and arrangements of the platelets, and explicit representation of all features in the microstructure within the fracture process zone leads to daunting problem sizes, as a high density of discretized elements is required (Barthelat et al., 2007; Barthelat and Espinosa, 2007;

[1]Department of Computer Science, University of California Santa Barbara, USA
[2]Department of Mechanical Engineering, University of California Santa Barbara, USA
[3]Department of Materials, University of California Santa Barbara, USA
[4]School of Chemistry, University of Southampton, UK

Corresponding author:
Rone Kwei Lim, Department of Computer Science, University of California Santa Barbara, Santa Barbara, California, 93106, USA.
Email: rklim13793@cs.ucsb.edu

Rabiei et al., 2010). Further, the strong interaction between the fracture process and the brick arrangement implies that fracture pathways are not known a priori, creating the need to allow for arbitrary cracking pathways to evolve with loading (Evans et al., 2001; Meyers et al, 2008).The need to capture local material rupture (e.g. the breaking of bonds holding platelets together) further compounds the problem, as rupture represents a strong nonlinearity that produces sharp spatial gradients in stiffness (i.e. a crack has zero stiffness while the surrounding material may be intact and therefore have high stiffness). For such problems, methods that rely on gradient-based techniques to find the roots of nonlinear equilibrium equations are often prone to extreme convergence difficulties that stem from the sharp discontinuities in stiffness.

We present an idealized model for nacreous materials that represents the platelets comprising the microstructure as rectangular bricks, whose position and orientation are solution variables to be determined through simulation. The bricks interact through nonlinear springs, which represent the very small volume fraction (1–5%) of polymer mortar that hold the bricks together (Barthelat and Rabiei, 2011; Jackson et al., 1990; Kamat et al., 2000). Hence, the nonlinear spring description represents the constitutive law that describes mortar, and includes both elastic response (for small separations between bricks), plastic response (when the separation between bricks causes straining of the mortar beyond its elastic limit) and rupture (when the separation between bricks is large enough to cause material failure in the mortar). Figure 1 provides a schematic illustration of the material model. A companion paper more fully discusses the physical justification and implications of the model, using the solution techniques and algorithms described here to quantify fracture parameters controlling failure (Pro et al., 2015).

Noting the fact that many fracture problems involve limited amounts of unloading, the problem can be cast in terms of energy minimization of a nonlinear elastic system: in this case, described by the nonquadratic energy potential describing the springs, or mortar. We treat the bricks as rigid due to their extreme stiffness relative to that of the mortar, thus the problem is essentially that of a collection of nearest-neighbor particle interactions (Pro et al., 2015).

The end result is a material idealization that simply involves finding the collection of brick positions and rotations that minimize the energy in the nonlinear springs connecting the particles. From a purely mathematical point of view, this is a fairly general problem in that it involves finding global energy minima of a highly nonlinear system of nearest neighbor interactions. While the idealized material model described above serves as the motivation, the algorithms described here are applicable to other problems of this type (such as the large deformation of fiber networks). The focus of this paper is on the strategic marriage of the GPU architecture to this class of problems. We will show that the GPU architecture offers powerful advantages, particularly when combined with algorithms that exploit the nature of nearest-neighbor interactions. Results are presented quantifying computational performance. A more detailed study of the implications of the simulations for materials development is to be published elsewhere (Pro et al., 2015).

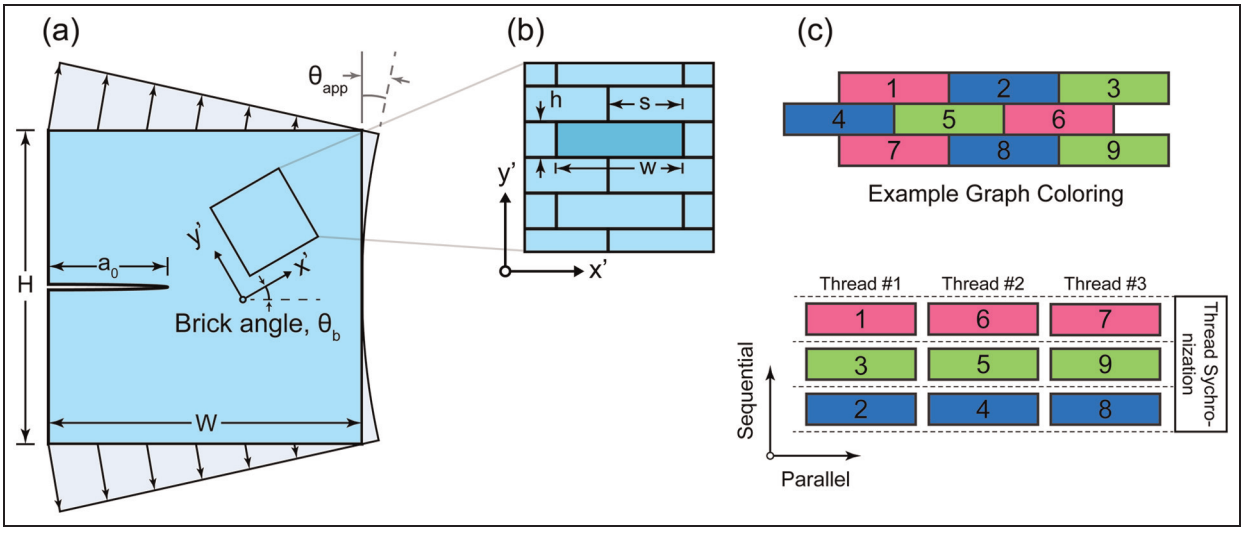

Figure 1. (a) Schematic illustration of macroscopic specimen and loading scenario (bending). There is a pre-existing crack in the middle of the specimen. (b) An example configuration of bricks. Many more bricks are used in the actual simulations. (c) Assignment of bricks to threads using graph coloring. Adjacent bricks are assigned different colors. Bricks of the same color are processed in parallel.

2 Background

2.1 Model

The central objective of the simulations presented here is to predict failure of a macroscopic specimen that has a large, pre-defined crack and is loaded in a combination of tension and bending, as shown in Figure 1(a). The macroscopic specimen is created by defining a two-dimensional 'wall' of overlapping rectangular bricks that are connected with cohesive springs; Figure 1(b) shows a close-up view of this microstructure and the dimensions that define the bricks. The bricks are treated as rigid bodies, while the mortar (implicitly represented by the cohesive springs connecting the bricks) is described as a nonlinear material. A large number of bricks are used to accurately capture the complex fracture behavior near the tip of the macroscopic crack shown in Figure 1(a). The simulation tool has been coded to allow for arbitrary combinations of brick width, height, overlap and orientation within a specimen. (Here, example results are presented for a single microstructural orientation relative to the specimen; more exhaustive study of the role of brick size and orientation is presented in a separate work (Pro et al., 2015).

The specimen is loaded by applying prescribed displacements to the bricks at the top of and bottom of the specimen shown in Figure 1(a). Bricks without prescribed displacements can undergo general rigid body motions (translation and rotation). As the bricks displace and rotate, the interface opening between bricks can change. The cohesive law describes the energy that is stored at the interfaces as the bricks change position, in terms of the relative displacements between the adjacent bricks defining the interface. The cohesive description contains three parameters: interface stiffness, critical separation and work to failure. The energy of an interface is computed as follows

$$E_{interface} = \int E_{pt} ds \tag{1}$$

where the integral is over the length of the interface. The energy at a given point is given by

$$E_{pt} = f\left(\sqrt{\Delta_n^2 + \Delta_t^2}\right) + g(\Delta_n) \tag{2}$$

where Δ_n is the displacement in the normal direction and Δ_t is the displacement in the tangential direction, and f and g are given by

$$f(x) = \begin{Bmatrix} 0.5\,k\,x^2, & x \le x_1 \\ k\,x_1\,(x - 0.5\,x_1), & x_1 < x \le x_2 \\ -0.5\,k\,\left[x_1^2 + x_2^2 + x^2 - 2\,x\,(x_1 + x_2)\right], & x_2 < x < x_1 + x_2 \\ k\,x_1\,x_2, & x \ge x_1 + x_2 \end{Bmatrix} \tag{3}$$

$$g(x) = \begin{Bmatrix} 0, & x \ge 0 \\ 0.0625\,k\,x^2\left(\frac{x}{x_1}\right)^4, & x < 0 \end{Bmatrix} \tag{4}$$

where k is the interface stiffness, x_1 is the critical separation, and $k\,x_1x_2$ is the work to failure. The traction generated between bricks is simply the derivative of the cohesive energy potential with respect to relative displacements. Figure 2 illustrates the traction–separation relationship as a function of brick separations, and the associated energy potential. At small relative displacements, the tractions are linear with separation, representing the elastic phase of mortar response: above a critical displacement, the traction remains constant,

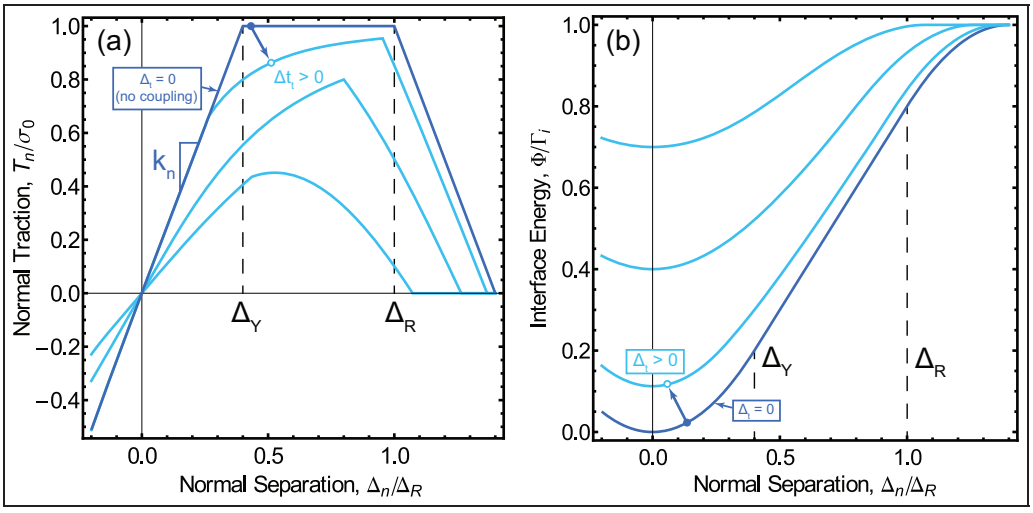

Figure 2. (a) Graph of traction-separation function. (b) Graph of the energy potential, defined as the integral of the traction-separation function.

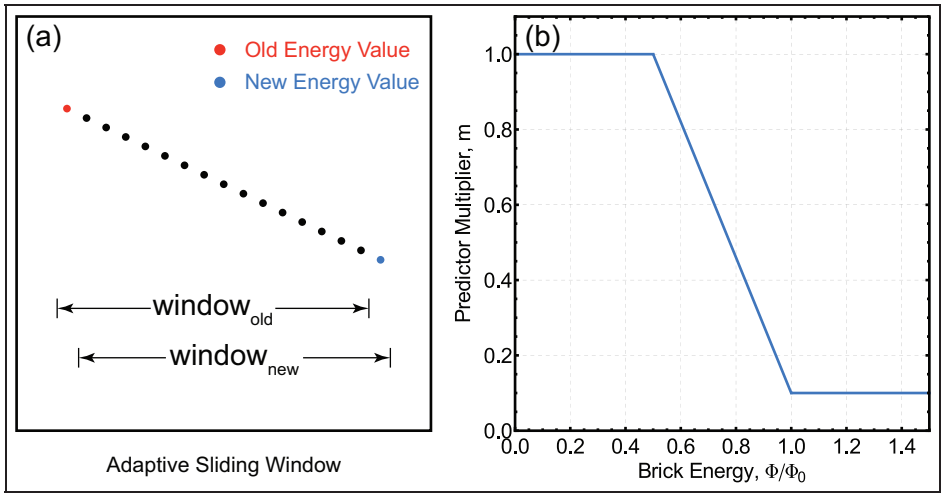

Figure 3. (a) Adaptive cycle illustration. The old sliding window is labeled $\text{window}_{\text{old}}$, while the new sliding window at the next cycle is labeled $\text{window}_{\text{new}}$. The blue circle is the new energy value, and the red circle is the old energy value. The variance and correlation coefficient are calculated each time the sliding window is updated. (b) Graph of position predictor multiplier. The predictor uses linear prediction when the energy is low, and a combination of linear and constant prediction as the energy increases.

representing the plastic yielding phase of mortar response. The rupture separation defines the point at which the mortar begins to fail: for large relative displacements, the traction between bricks is zero, and the energy potential assumes the value of the area under the traction-separation curve.

2.2 Basic numerical method

The simulation is thus an energy minimization problem, where the solution involves finding $x_1,y_1,\theta_1, x_2,y_2,\theta_2,....x_n,y_n,\theta_n$, where x_n,y_n,θ_n is the x-position, y-position and orientation of brick n, such that the energy $E(x_1,y_1,\theta_1,x_2,y_2,\theta_2,....x_n,y_n,\theta_n)$ is minimized. The energy function E depends on the brick size and orientation and the properties of interfaces between bricks. For problems involving cohesive yielding or rupture, the energy landscape is multidimensional, highly nonlinear, and may contain several solutions (local minima). For problems of this nature, typical gradient based schemes do not perform well; instead, direct heuristic search numerical optimization algorithms are preferred. We use the Monte-Carlo direct search method, also known as simulated annealing, to find the minimum (Gonzalez, 2007; Ingber, 1993; Kirkpatrick et al., 1983). In this case, the temperature parameter T is a fictitious parameter that is chosen by the user as opposed to a physically meaningful temperature used in annealing simulations involving physical quenching.

One advantage of simulated annealing is that it is highly parallelizable, which allows us to leverage the computational power of graphics processing units (GPUs). The basic parallelization concept is to move multiple bricks simultaneously, computing the associated energy change and accepting those that lower the energy. A small fraction of movements that increase the energy are accepted as well, to avoid being trapped in a local minima; an exponential function is used to describe the probability controlling the acceptance of movements that raise energy. In this approach, one must avoid moving adjacent bricks at the same time; as will be described, we address this problem by coloring the bricks using a graph coloring algorithm to identify sets of nonadjacent bricks, as shown in Figure 1(c). Different sets of bricks, each with a given color, are passed into separate threads of the GPU.

The basic algorithm is outlined as follows. While the solution for any given prescribed displacement can be found in a single minimization step, in the approach taken here, the prescribed displacements are applied incrementally. This both captures solutions at a range of loads and promotes convergence, as described in subsequent sections:

```
Apply displacement m_i to each driven brick d_k
Repeat these steps until convergence
  For each free brick b_n, perform the following steps:
    Compute the current energy E_old
    Generate standard uniform random numbers r_x, r_y, r_θ
    Perturb the position and orientation using r_x, r_y, r_θ
    Compute the new energy E_new
    If E_new–E_old < = 0
      Accept the new position and orientation
    Else
      Generate a standard uniform random number r
      If r < exp((E_old–E_new) / T)
        Accept the new position and orientation
      Else
        Reject the new position and orientation
```

3 Adaptive numerical methods

We have developed and implemented several enhancements to increase performance and usability of the basic algorithm. These methods include adaptive cycle count, adaptive step size adjustment, adaptive displacement adjustment, and a position predictor.

The adaptive cycle count allows the user to specify tolerance values. The algorithm will run as many iterations (cycles) as necessary to reach the specified tolerance values. Without this, the user would have to specify cycle count directly, but the number of cycles required to converge is not known beforehand.

The underlying idea is illustrated in Figure 3(a). We use a sliding window to monitor the variance and correlation coefficient to determine convergence. At each cycle, the new energy value is added to the sliding window (blue in Figure 3(a)), while the old value is removed (red). Then the variance and correlation coefficient of the window are calculated and compared to user-specified tolerance parameters. These tolerance parameters include WS (size of the sliding window, which corresponds to the number of Monte Carlo cycles), STOL (tolerance for variance), RTOL (tolerance for correlation coefficient), and ConvergenceRep (the number of times the convergence criteria need to be met to advance to the next displacement step). After the STOL and RTOL criteria have been met for ConvergenceRep times (which is typically set to 1/4 to 1/2 of WS), the current displacement step is considered to be finished.

There are several methods to calculate the correlation coefficient and variance. One approach is to use a two-pass algorithm that calculates the mean first and then calculates the variance. This approach does not work well for this situation since whenever a new value is added or an old value removed from the sliding window, the entire window has to be reprocessed. A different approach is to use a one-pass algorithm that dynamically updates without reprocessing the entire window. A well-known formula is the following (Chan et al., 1983)

$$\begin{aligned} SQ_{new} &= SQ_{old} + x_{new}^2 - x_{old}^2; \quad SQ_{init} = 0 \\ SN_{new} &= SN_{old} + x_{new} - x_{old}; \quad SN_{init} = 0 \\ variance &= \frac{SQ_{new} - \frac{SN_{new}^2}{WS}}{WS} \end{aligned} \tag{5}$$

where x_{new} is the new value, x_{old} is the old value, WS is the window size, and SQ and SN are intermediate variables.

However, this formula suffers from round-off errors and the computed variance can become negative rather quickly. We use a more numerically stable one-pass formula (Chan et al., 1983)

$$\begin{aligned} mean_{new} &= mean_{old} + \frac{x_{new} - x_{old}}{WS}; \quad mean_{init} = 0 \\ SS_{new} &= SS_{old} + (x_{new} - mean_{new})(x_{new} \\ &\quad - mean_{old}) - (x_{old} - mean_{new})(x_{old} - mean_{old}); \\ SS_{init} &= 0 \\ CS_{new} &= CS_{old} + (x_{new} - mean_{new})\left(\frac{WS+1}{2}\right) \\ &\quad + (x_{old} - mean_{old})\left(\frac{1-WS}{2}\right); \quad CS_{init} = 0 \\ variance &= \frac{SS_{new}}{WS} R^2 = \frac{CS_{new}^2}{WS * variance * \left(\frac{WS^2-1}{12}\right)} \end{aligned} \tag{6}$$

where variance and R^2 (correlation coefficient) are the output values, and mean, SS, CS are the intermediate variables.

This formula accumulates round-off errors more slowly than the previous formula, but it can still produce a negative variance after some time. To remedy this problem, the MPFR (Multiple Precision Floating-Point Reliably) package (Laurent et al., 2007) was used to compute the intermediate values (mean, SS, CS) at quadruple precision, whereas normal floating-point numbers are either single or double-precision. The MPFR package is a widely used library for performing extended precision calculations. After the variance and R^2 are computed, they are compared to the user-specified STOL and RTOL criteria to determine convergence.

We included an adaptive step size strategy that modifies the brick step size (the amount of perturbation to a brick's position during one step) and rotation size (the amount of perturbation to a brick's orientation) based on acceptance probability (Belegundu and Chandrupatla, 2011; Corana et al., 1987). If the acceptance probability is very low, then most of the attempted moves are rejected, which results in a loss of efficiency. If the acceptance probability is very high, then this implies that the brick step size and rotation size is small, so it would take more time to converge after an applied displacement. The equation for adjusting both the brick step size and rotation size is given by

$$s_{n+1} = \begin{cases} s_n * (0.5 + \alpha), & \alpha < 0.4, \alpha > 0.6 \\ s_n, & 0.4 \le \alpha \le 0.6 \end{cases} \tag{7}$$

where s_n is the step size at load step n, and α is the acceptance probability. The step size can be adjusted either globally or on a per-brick basis. In global adjustment, a single step size is used for all bricks and adjusted periodically. For the per-brick basis, each brick maintains its own step size.

Our strategy for adaptive displacement adjustment controls the displacement size to reach stable crack growth behavior. The displacement size is adjusted using forward control and backtracking. In forward control, the step size is adjusted proportionally based

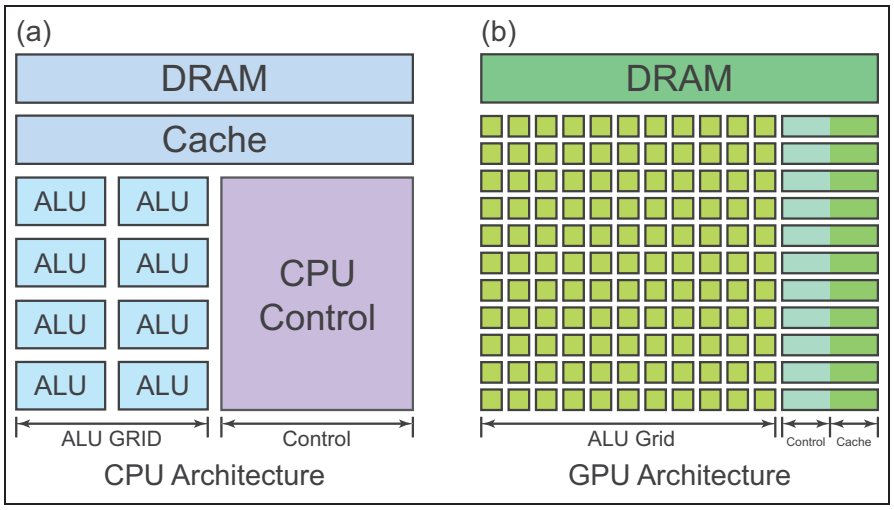

Figure 4. CPU (central processing unit) and GPU (graphics processing unit) architectures. A GPU has more transistors devoted to execution, while a CPU has more transistors for cache and control mechanisms.

on a target number of cycles for convergence. The equation for forward control is given by

$$\Delta_{n+1} = \Delta_n * \frac{c_t}{c_r} \tag{8}$$

where Δ_n is the displacement step size at load step n, c_t is the target number of cycles and c_r is the actual number of cycles required to converge the previous displacement step.

Besides forward control, we also use backtracking, which occurs when the number of cycles exceeds an upper limit. In backtracking, the displacement step is reverted and the new displacement step size is given by

$$\Delta_{n+1} = \Delta_n * \frac{2}{3} * \frac{c_t}{c_u} \tag{9}$$

where c_u is the upper limit for cycles.

A position predictor is used during the early stage of the simulation (prior to widespread rupture between bricks) to accelerate convergence, since the relationship between prescribed displacements and brick positions is approximately linear. In this case, the converged position of the bricks in one displacement step is linearly extrapolated to obtain an initial position for the next displacement step that results in a faster convergence. The predictor is given by

$$p_i^{n+1} = p_i^n + \frac{\Delta_{n+1}}{\Delta_n} * \left[p_i^n - p_i^{n-1}\right] * m \tag{10}$$

where p_i^n is the position of brick i at displacement step n, and m is the multiplier defined below.

As the simulation progresses, the interfaces between bricks approach the point of fracture, where a linear predictor is no longer accurate, and a constant predictor is more appropriate. To reduce the prediction error, the following multiplier is used

$$\boldsymbol{m} = 1 - 1.8\left(\left[\max\left(\min\left(\frac{\Phi}{\Phi_0}, 1.0\right), 0.5\right)\right] - 0.5\right) \tag{11}$$

where Φ is the current brick energy, and Φ_0 is the maximum elastic brick energy.

As shown in Figure 3(b), when the brick energy is low, the multiplier is equal to 1, so the linear predictor is used. As the brick energy increases, the multiplier value decreases, resulting in the use of a linear combination of constant predictor and linear predictor. The position predictor results in a significant performance improvement during the harmonic (nonfracture) part of the simulation, as will be discussed subsequently.

4 GPU architecture

During the past few years, GPUs have increasingly been employed for scientific and engineering computations as GPU architectures have become more flexible and powerful (Nvidia CUDA (2011a). To achieve high performance on the GPU, it is important to understand the differences between central processing unit (CPU) and GPU architectures, which are illustrated schematically in Figure 4. A general-purpose CPU, such as the Intel Core i7-2600 or the AMD Phenom II X6 1100T has several cores to run multiple threads (4 physical cores / 8 virtual cores for the Core i7-2600 and 6 physical cores for the Phenom II X6 1100T). It also has a large cache (8MB for the Core i7-2600 and 6MB for the Phenom II X6 1100T). A CPU also has sophisticated flow control mechanisms, such as branch prediction, data/instruction prefetching, out-of-order execution,

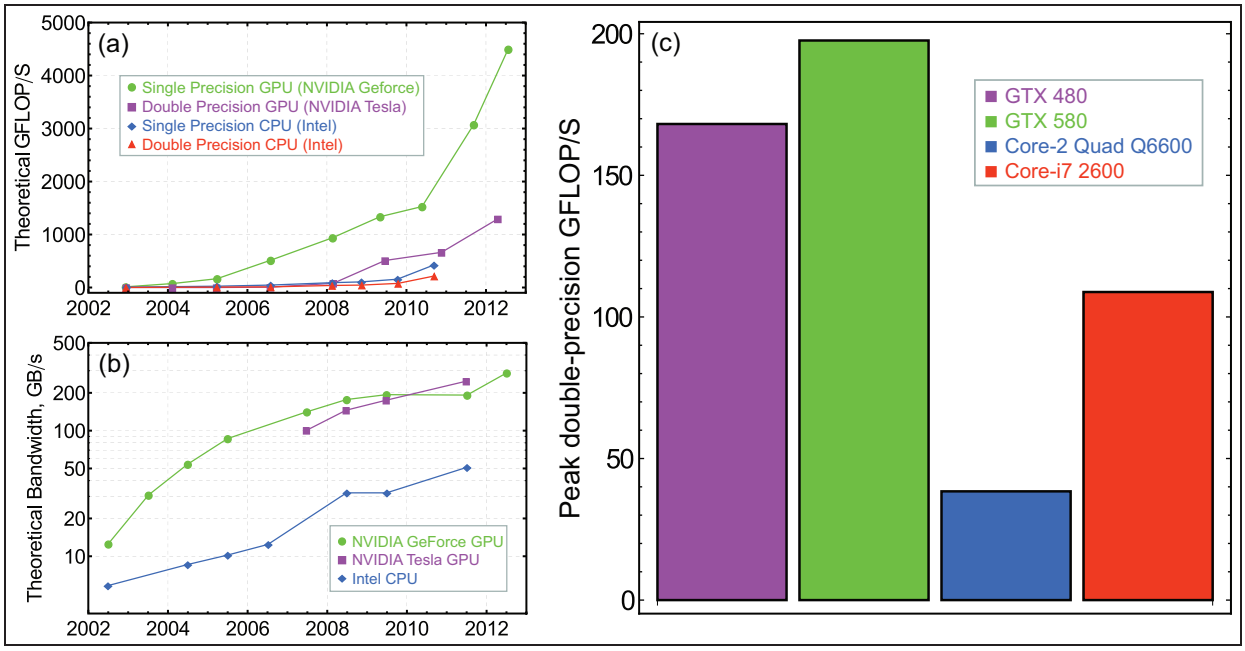

Figure 5. (a) Peak GFLOPS of CPUs (central processing units) and GPUs (graphics processing unit). The peak GFLOPS is significantly higher on a GPU. (b) Peak memory bandwidth of CPUs and GPUs. The peak bandwidth is significantly higher on a GPU (c) Peak double precision GFLOP/s on the test systems: Nvidia GTX 480 GPU/Core 2 Quad Q6600 CPU and Nvidia GTX 580 GPU/Core i7 2600 CPU.

and superscalar execution. CPUs are designed to run a few threads as fast as possible, and are well-suited for problems with a low degree of parallelism.

In contrast, a GPU, such as the Nvidia GTX 580 or the AMD Radeon HD 6970, has a large number of execution units to process a large amount of data in parallel. For example, the Nvidia GTX 580 has 16 SMs (streaming multiprocessors), where each SM has 32 SPs (shader processors). Each SM can execute independent streams of instructions, whereas the SPs within each SM execute instructions in a SIMD (Single Instruction Multiple Data) manner. The Nvidia GTX 580 has a 64K L1 cache and a 768K L2 cache. GPUs lack the sophisticated flow control mechanisms that are present on CPUs, such as branch prediction. GPUs are designed to run large numbers of threads, and are suited for problems with a high degree of parallelism (Nvidia CUDA, 2011a, 2011b).

In comparing CPU and GPU, a GPU has many more transistors devoted to execution units than a CPU, whereas a CPU has more transistors devoted to cache and flow control mechanisms compared to a GPU, as shown in Figure 4.

The result is that a GPU can reach higher peak performance, as shown in Figure 5, which compares FLOP rates and bandwidths of Nvidia GPUs and Intel CPUs. Figure 5(c) shows the double-precision FLOP rate on the two GPUs and two CPUs used for the test system.

To approach the peak performance of the GPU, the code needs to be structured in a way that takes advantage of the architecture of the GPU and avoids the limitations that reduce performance. These considerations include having enough threads running in parallel, ensuring coalesced memory access, minimizing data transfer between GPU and CPU, and minimizing warp divergence (Nvidia CUDA, 2011a, 2011c).

Having enough threads running in parallel is important because GPUs are designed for highly parallel operations, which allows latency in one thread to be hidden by performing other operations in another thread. In addition, context switching between threads on a GPU is significantly less expensive than on a CPU (Nvidia CUDA, 2011). For these reasons, it is typical to have thousands of threads running in parallel on a GPU.

Ensuring coalesced memory access improves performance since the memory controller on a GPU performs operations based on groups of threads at a time. Memory access is coalesced when the threads within a group access a contiguous region of memory. For example, the Nvidia GPU uses a warp of 32 threads (Nvidia CUDA, 2011a). If the memory access within a warp is not coalesced, then the controller must perform additional operations, which reduce performance.

Minimizing data transfer between the GPU and the CPU is another important consideration. The bandwidth of the link between CPU and GPU is much lower than the on-board memory in the GPU. For example, the Nvidia GTX 580 has a 16x PCI-Express 2.0 capable interface, which can transfer a maximum of 8 GB/s

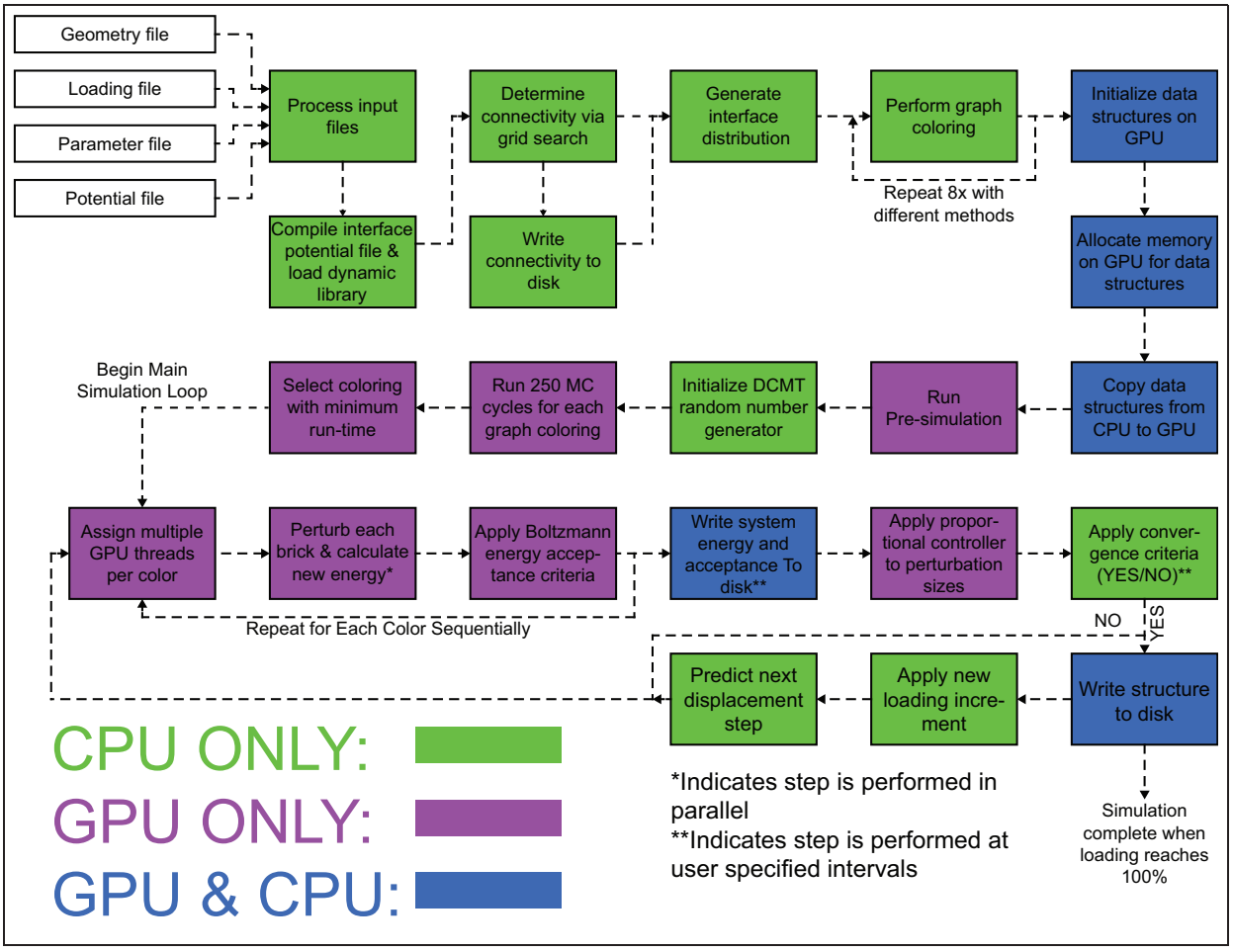

Figure 6. Flowchart of the overall simulation algorithm. CPU: central processing unit; GPU: graphics processing unit.

between CPU and GPU. In contrast, the on-board memory on the GTX 580 has a maximum transfer rate of 192.4 GB/s, which is much higher.

Minimizing warp divergence is important since GPUs operate in a SIMD manner. Warp divergence is caused by different threads taking different execution paths, such as two threads taking different branches in a conditional statement. Whenever the GPU encounters warp divergence, it serializes the execution of the warp, which leads to reduced performance (Nvidia CUDA, 2011a).

5 GPU implementation

Prior to the GPU-based simulation algorithm, input processing and initialization steps are performed on the CPU. An overview of the CPU and GPU operations in the present code is shown in Figure 6. The input processing step reads information about the bricks, interfaces, and the overall structure from input files. This step allows the simulation to handle different brick shapes and sizes, and different interface properties. In the data structure, each brick has a position, orientation, constraints, neighbor information, and interface properties associated with it.

The initialization step involves finding the connectivity between bricks and determining which bricks can be processed in parallel. We devised a grid-based algorithm to find the interface connectivity, with a time complexity of O(N). The algorithm works by setting up a rectangular grid of cells and associating each brick to a cell. Then it finds all candidate bricks that are within a certain distance of the selected brick and tests them for interface connectivity, i.e. whether two bricks have a common interface. We then use a graph coloring based algorithm to determine which bricks can be processed in parallel (Gonzalez, 2007). Neighboring bricks are colored with different colors, then all bricks with the same color are grouped together. Figure 1(c) provides a diagram that illustrates the graph coloring. Adjacent bricks cannot be processed in parallel since this would result in a data race. After these steps have been performed, the data structure containing information about bricks and interfaces is transferred to the GPU memory. The number of bricks that can be simulated is limited by the amount of GPU memory; with

the GTX 580, we can simulate up to 2,000,000 bricks. Next, the pre-simulation phase is performed. In this phase, the different candidate graph colorings from the previous phase are used to execute the simulation for a small number of cycles. The reason for performing pre-simulation is because in general, the graph coloring problem is NP-complete. As a heuristic, we generate several possible candidate colorings and determine their execution time. The execution time can vary for different colorings because different colorings generate different memory access patterns. It is difficult to know a priori which coloring will be optimal. The graph coloring that minimizes the execution time is selected for the main simulation phase.

At set intervals chosen by the user, some state information is transferred from GPU to CPU so that intermediate states can be obtained by the user. The majority of the data remains on the GPU to minimize the amount of time spent in transferring data across the bus since the bus is much slower than the on-board memory of the GPU.

The random numbers that are used in simulated annealing are generated via a two-stage process. First, parallel, independent streams of random numbers are generated using the Dynamic Creation Mersenne Twister (DCMT) algorithm (Matsumoto and Nishimura, 1998). Having independent streams ensures that there is no correlation between the different streams that could potentially skew the simulation results. Each stream of random numbers is associated with a different set of generator parameter values, which we computed on a cluster. The parameter values are computed once and can be reused multiple times with different seeds to generate different sets of independent streams of random numbers. The original DCMT algorithm was written to run on a CPU (Matsumoto and Nishimura, 1998). The DCMT algorithm was modified by NVIDIA to run on the GPU using CUDA (Compute Unified Device Architecture) (Podlozhnyuk, 2007). We have further modified the DCMT algorithm to improve its initialization and increase randomness. Rather than using one seed value for all the random number generators (RNG), we modified the process to generate multiple seed values from the initial seed, one for each RNG. We have also improved performance by increasing thread parallelism and exploiting instruction pipelining. Then, the output is processed using Advanced Encryption Standard-Electronic Codebook (AES-ECB) (United States National Institute of Standards and Technology, 2001) to improve the randomness properties. We have parallelized the AES algorithm to run on the GPU. We used TestU01, which contains a large collection of statistical tests for random number generators, to test the quality of our random number generation (L'Ecuyer and Simard, 2007). In TestU01, the Mersenne Twister passed all tests except for the linear complexity tests. Processing the output of MT using AES enables the results to pass all of the statistical tests in TestU01.

In the simulation, bricks are assigned to threads using a graph coloring based algorithm since neighboring bricks cannot be modified simultaneously. For example, in Figure 1(c), bricks 1 and 6 are nonadjacent and colored purple, while bricks 2 and 4 are colored blue, and so on. Then all of the purple bricks are processed in parallel, followed by all of the green bricks, and so on. During the simulation, thousands of threads are launched to simulate a large number of nonadjacent bricks in parallel. This takes advantage of the large number of execution units available on a GPU. However, it also leads to a less coalesced memory access pattern since the bricks are nonadjacent, which reduces performance. To overcome this issue, we rearranged the data structure on the GPU so that the memory access pattern is more coalesced. During input processing, the data is stored in array of structure format. In an array of structure format, an object's properties are grouped together. This format is used because the data structure needs to change dynamically as input files are processed. Afterward, the data is rearranged into a structure of array format for computation on the GPU since this is more efficient. In a structure of array format, properties of the same type from multiple objects are grouped together. In addition, the order of the individual elements in the data structure is rearranged to increase efficiency. The rearrangement is performed once on the CPU, prior to the main simulation phase. Figure 7 provides a diagram of the data structures used in the simulation.

6 Results

The test problem discussed below has the following configurations. The bricks are oriented with varying angles with respect to the macroscopic crack direction, and the bricks have a width to height ratio of approximately 3.5. The number of bricks in various simulations ranges from approximately 50,000 to 350,000. The bricks are displaced under a bending scenario, as shown in Figure 1(a). The parameters used in the simulations considered here are listed in Table 1 in the Appendix.

The test problem was run on two different platforms to compare performance. The first platform consists of a Core 2 Quad Q6600 CPU at 2.4 GHz, EVGA nForce 780i motherboard, 8GB DDR2-800 memory, and an Nvidia GTX 480 GPU. The second consist of a Core i7 2600 CPU at 3.4 GHz, Gigabyte Z68X motherboard, 16GB DDR3-1333 memory, and an Nvidia GTX 580 GPU. The software environment is CentOS 6, GCC 4.4, and CUDA runtime 4.0.

Figure 8 provides an illustration of the results obtained from the simulation; the energies associated

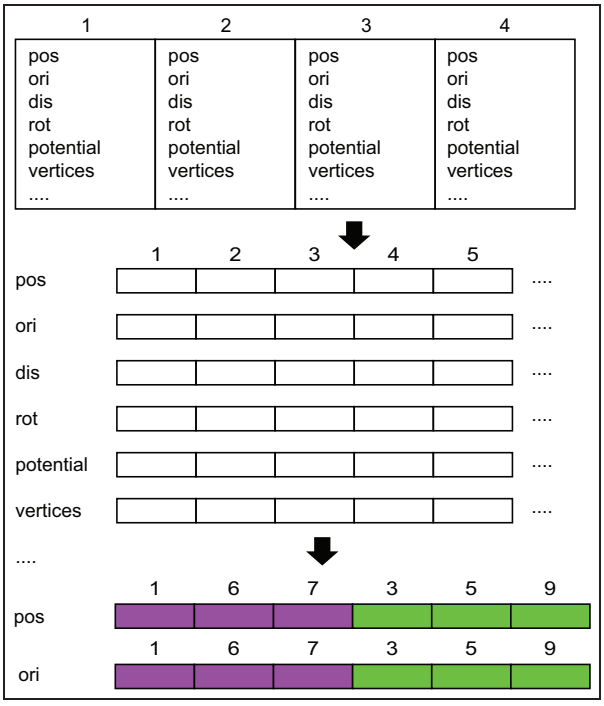

Figure 7. Diagram of the data structures used in the simulation. Initially, the data structures are in an array of structure format during input processing. It is then rearranged into a structure of array format. Additionally, the individual elements are rearranged according to the graph coloring order.

with the interfaces surrounding a given brick are shown in Figure 8(a), while the relationship between global system energy and the prescribed loading is shown in Figure 8(b). In Figure 8(a), the bricks near the crack tip have high energy due to the significant stress concentration of the crack tip, and the bricks farther away have lower energy. The region shown in red in Figure 8(a) represents the fracture process zone where the interfaces between bricks are experiencing failure; at greater loads, the macroscopic crack advances, meandering between bricks according to the microstructural orientation. Extensive details of the fracture process are provided in our companion paper (Pro et al., 2015).

Figure 8(b) illustrates that for small levels of applied displacement, the system energy is essentially quadratic, as implied by a linear (elastic) traction-separation relationship. At a critical value of the applied displacement, indicated with open circles, the quadratic nature of the energy with loading is lost, due to nonlinear rupture effects between the bricks. The solutions for displacements below the onset of rupture are obtained in a rapid fashion using the predictor method described earlier. Once fracture initiates, the linear predictor is less efficient, so the constant predictor becomes more appropriate.

A key consideration in this simulation approach is the scaling of the required computation times with the number of bricks, which effectively defines the size of problem (i.e. the level of microstructural detail) that can be simulated in a reasonable time.

As shown in Figure 9(a), the initialization time scales approximately as O(N), where N is the number of bricks. The initialization is done on a single core. Both the Core 2 and Core i7 are quad-core processors, but the Core i7 has a more modern architecture (Sandy Bridge for Core i7 vs Conroe for Core 2) and a faster

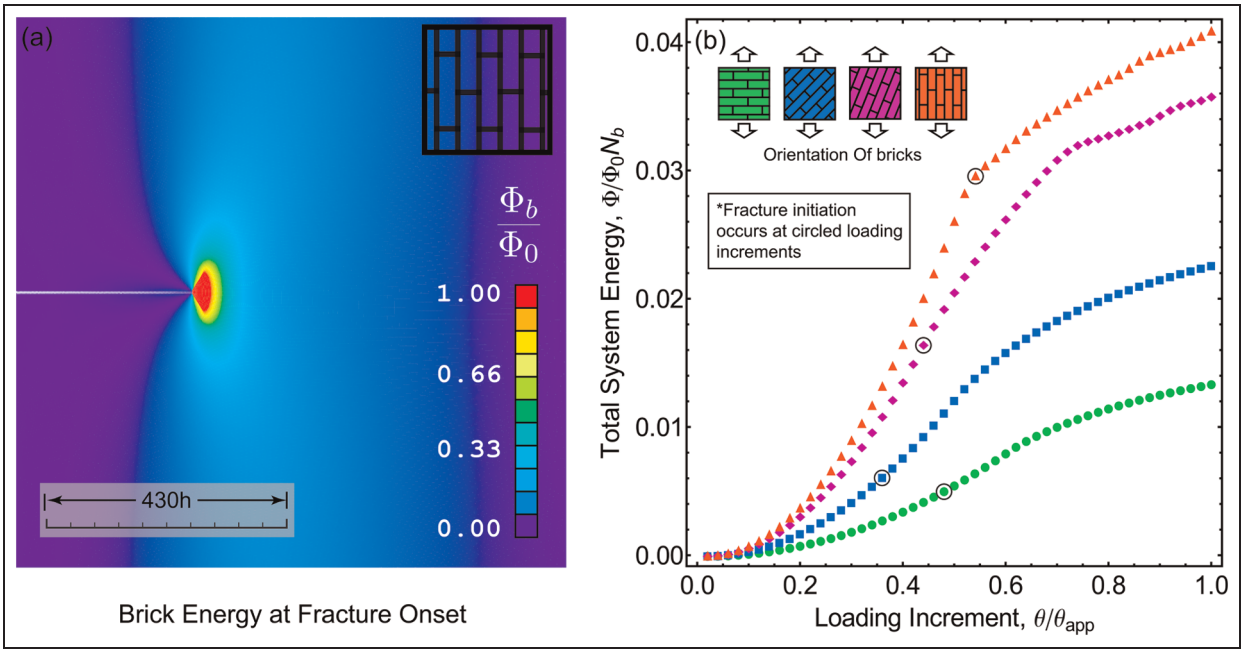

Figure 8. Simulation output. (a) The colors represent the energy of the bricks. (b) The total energy for various brick orientations.

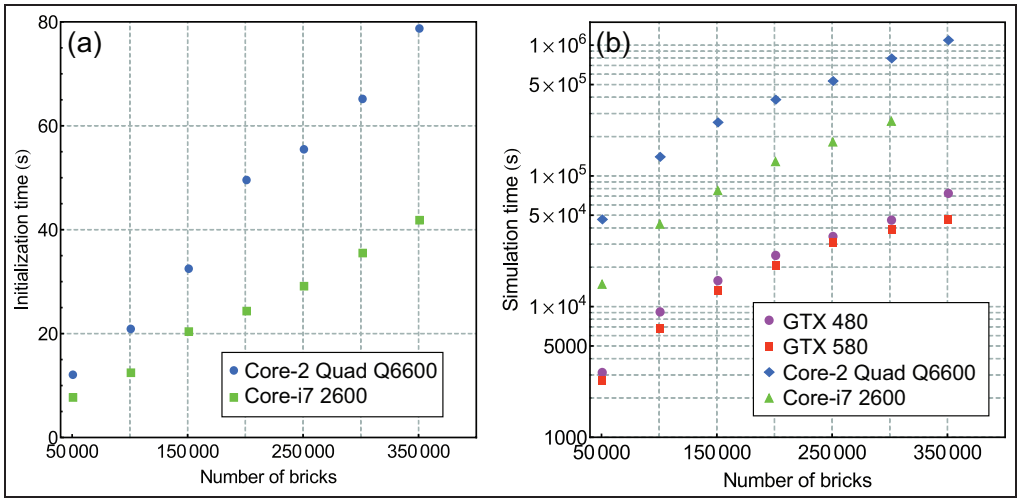

Figure 9. (a) Initialization time. (b) GPU and CPU simulation times (semilog). GPU: graphics processing unit; CPU: central processing unit.

clock speed, so the result is faster on the Core i7. In addition, the Core i7 system also has faster and larger memory.

After initialization, the main part of the simulation begins. These cases are run with the adaptive methods. The simulation times for the two different systems are shown in Figure 9(b). We can see that on the first system (GTX 480 and Core 2), the GPU version (GTX 480) is about 16 times faster than the corresponding multithreaded CPU version (quad-core Core 2), whereas on the second system (GTX 580 and Core i7), the GPU version (GTX 580) is about 6 times faster than the multithreaded CPU version (quad-core Core i7). The simulation time scales roughly as $O(N^{1.5})$, where N is the number of bricks. The simulation time for GTX 580 is approximately 15% faster than for GTX 480, which roughly matches the hardware performance difference between GTX 480 and 580. The simulation time for Core i7 is approximately 3 times faster than for Core 2. Both processors are quad-core, but the Core i7 has approximately 40% higher clock speed and a newer architecture.

In Figure 10(a), the simulation times for different RTOL and STOL are shown. The simulation times for larger tolerance values are generally lower; however, there is some variability due to the stochastic nature of Monte Carlo simulations. In Figure 10(b), the converged energy values at the last displacement step are shown for different tolerance values. Simulations with smaller tolerance values generally result in slightly lower energy values, although all three tolerance values produce energy values that are quite similar.

As seen in Figure 11, the simulation time on the CPU scales approximately in a linear manner with the number of cores. This indicates that the CPU implementation is not experiencing scalability bottlenecks that would limit performance, at least up to four cores.

To determine which tasks of the computation take the most time, Figure 12 shows the percentage of the total simulation time for the main tasks in the simulation. As seen in the figure, the Monte Carlo task takes the largest percentage of the simulation time. The Monte Carlo component involves perturbing the position and orientation of each brick, and determining whether to accept or reject based on the energy difference. The other two components, which are the energy calculation and the random number generation, take significantly less time, with the random number generation being the smallest component. The random number generation component generates random numbers which are then used in the Monte Carlo component. The relative percentages remain roughly constant with respect to the number of bricks, which is expected. The percentages are also similar across the two GPUs (GTX 480 and 580). These three components take 99% of the simulation time. The reason that the Monte Carlo component takes a significant portion of the simulation time is that it involves noncoalesced memory access. We have optimized the data structure to reduce the amount of noncoalesced memory access, but some of it still exists.

6.1 Performance improvement due to adaptive methods

In this section, the test problem is the same as in the previous section, but the simulation has been run without the adaptive methods to illustrate the difference in performance. Figure 13 provides a comparison of simulation times with and without adaptive methods for two GPUs, both in terms of raw values (Figure 13(a)) and relative performance (Figure 13(b)).

From Figure 13(a), the performance difference between running with the adaptive algorithms and

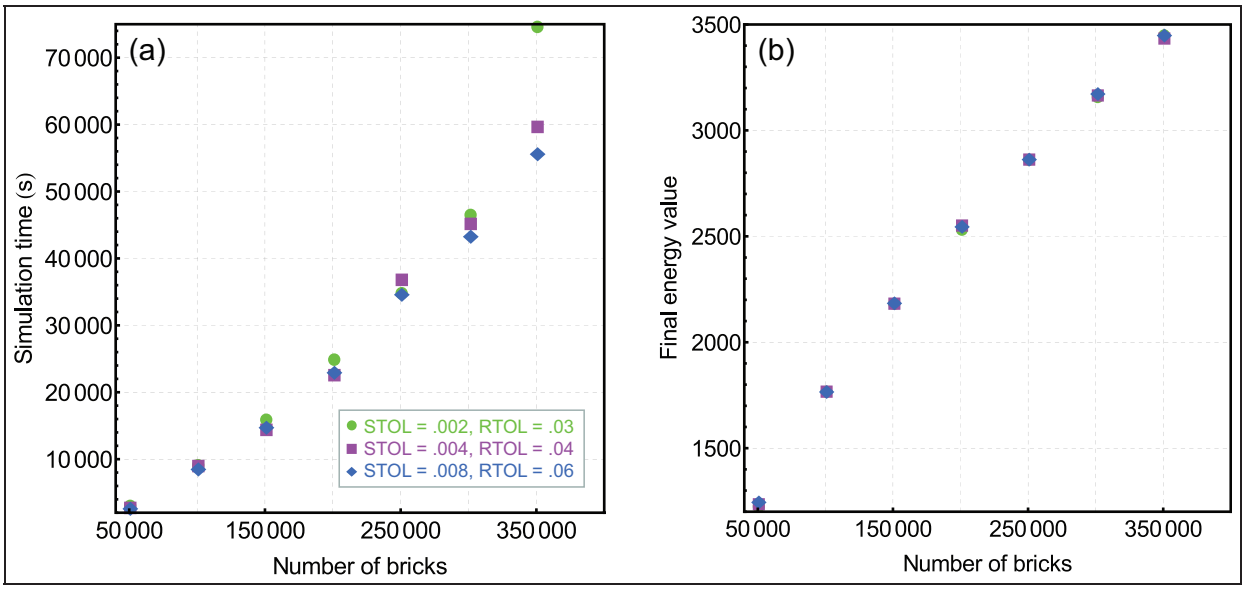

Figure 10. (a) GPU (graphics processing unit) simulation times with different tolerance values. (b) Final energy value with different tolerance values.

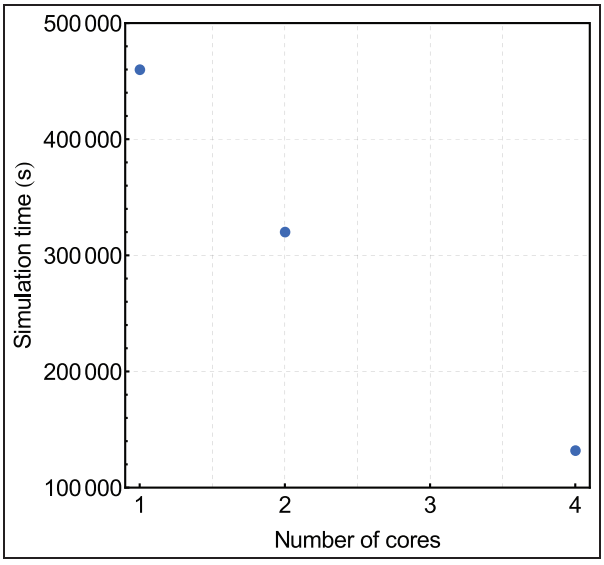

Figure 11. CPU (central processing unit) simulation time with different numbers of cores.

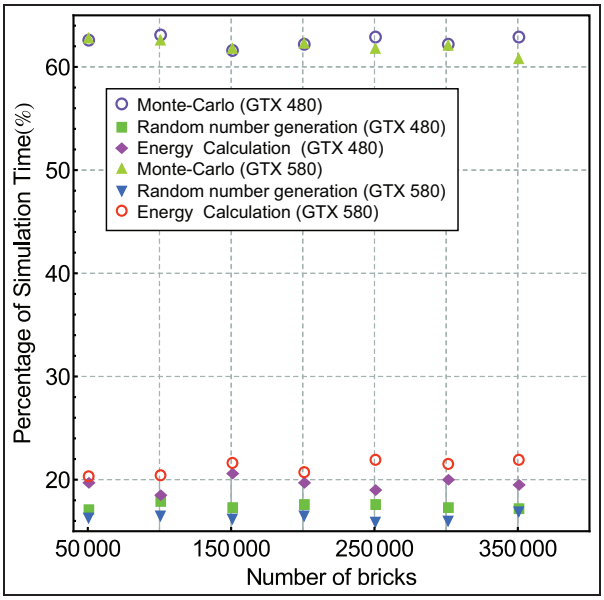

Figure 12. Percentage of simulation time for different components on GTX 480 and 580.

without the adaptive algorithms is about a factor of five. The position predictor produces the largest improvement in performance. Other adaptive methods, such as adaptive brick step size and adaptive displacement size, also produce some improvements in performance, but not as large as the position predictor. The performance gain from adaptive methods is problem dependent; in some cases, the improvement can be much higher. For example, if a large proportion of the simulation is harmonic, then the adaptive methods can result in significantly larger speedups.

Figure 13(b) summarizes the speedup we have achieved through efficient GPU implementation and adaptive strategies. As seen in the figure, our GPU implementation achieves approximately 16× speedup over the multithreaded CPU (quad-core) implementation for the test problem with 300,000 bricks, and the adaptive methods achieve an additional 5× speedup, for a total speedup of about 80x. This is a significant speed-up that will have a marked impact on studies of microstructural effects.

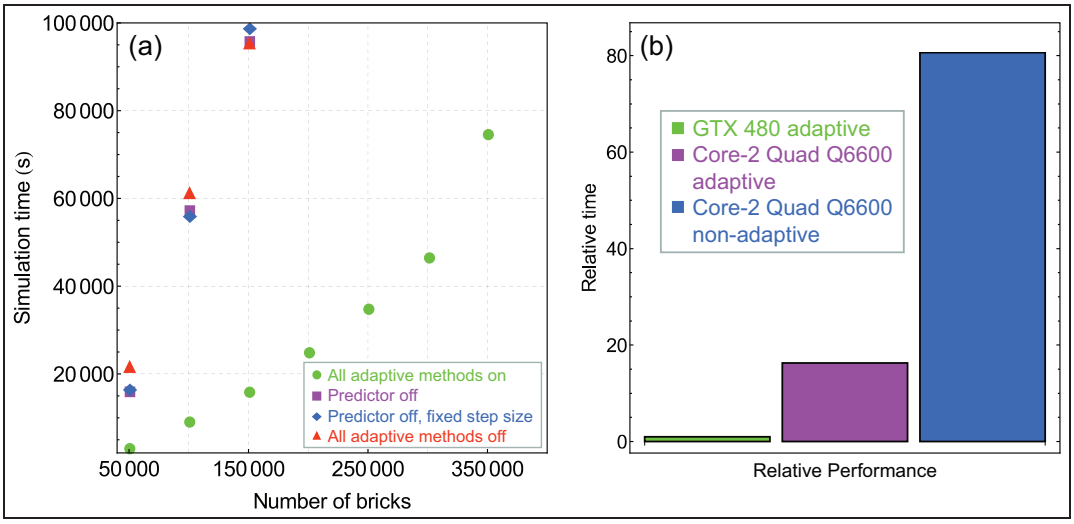

Figure 13. (a) Comparison of GPU *(graphcs processing unit)* simulation time for adaptive vs. *non*adaptive algorithms. (b) Relative speedup from using GPU and adaptive methods. The time for Core 2 Quad Q6600 *non*adaptive is an estimated runtime based on the difference between GTX 480 adaptive and GTX 480 *non*adaptive, and the difference between GTX 480 adaptive and Core 2 Quad Q6600 adaptive, since it takes an impractical amount of time to run the Core 2 nonadaptive case.

7 Conclusion

In this paper we have presented an efficient GPU-based Monte Carlo algorithm for fracture simulation of large-scale 'brick and mortar' composite materials. We have enhanced the basic algorithm with adaptive methods to increase performance and usability. Our GPU implementation processes multiple bricks in parallel using graph coloring to assign bricks to threads. Our GPU version achieved approximately 16× speedup over the corresponding multithreaded CPU (quad-core) version. An additional 5× speedup was achieved using adaptive algorithms, for a total speedup of approximately 80×. This enables a simulation to be run in hours on a GPU as opposed to weeks on a CPU. Possible future work includes generalizing the bricks to be deformable instead of rigid, incorporating more complex interface properties to allow for plastic deformation, and further enhancements to improve performance.

Acknowledgements

We would like to thank the Institute for Collaborative Biotechnologies for cluster access.

Funding

This work was supported by the National Science Foundation [grant number DMR-1233704, GOALI-8-442490-22028-3]; the National Science Foundation Graduate Research Fellowship; and the Institute for Collaborative Biotechnologies [grant number BM 4.13] through grant W911NF-09-0001 from the U.S. Army Research Office; and the Department of Energy [grant number DE-SC0008975]. The content of the information does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred.

References

Barthelat F and Espinosa H (2007) An experimental investigation of deformation and fracture of nacre-mother of pearl. *Experimental Mechanics* 47.

Barthelat F and Rabiei R (2011) Toughness amplification in natural composites. *Journal of the Mechanics and Physics of Solids* 59: 829–840.

Barthelat F, Tang H, Zavattieri P, Li CM, et al. (2007) On the mechanics of mother-of-pearl: A key feature in the material hierarchical structure. *Journal of the Mechanics and Physics of Solids* 55.

Belegundu AD and Chandrupatla TR (2011) *Optimization Concepts and Applications in Engineering*, 2nd edition. Cambridge: Cambridge University Press.

Chan TF, Golub GH and Leveque RJ (1983) Algorithms computing sample variance: analysis and recommendations. *The American Statistician* 37.

Corana A, Marchesi M, Martini C, et al. (1987) Minimizing multimodal functions of continuous variables with the simulated annealing algorithm. *ACM Transactions on Mathematical Software* 13: 262–280.

Evans AG, Suo Z, Wang RZ, et al. (2001) Model for the robust mechanical behavior of nacre. *Journal of Materials Research* 16: 2475–2484.

Gonzalez TF (editor) (2007) *Handbook of Approximation Algorithms and Metaheuristics*. Chapman & Hall/CRC.

Ingber L (1993) Simulated annealing practice versus theory. *Mathematical and Computer Modeling* 18: 29–57.

Jackson AP, Vincent JFV and Turner RM (1990) Comparison of nacre with other ceramic composites. *Journal of Material Science* 25: 3173–3178.

Kamat S, Su X, Ballarini R, et al. (2000) Structural basis for the fracture toughness of the shell of the conch Strombus gigas. *Nature* 405: 1036–1040.

Kirkpatrick S, Gelatt CD and Vecchi MP (1983) Optimization by simulated annealing. *Science* 220: 671–680.

L''Ecuyer P and Simard R (2007) TestU01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software* 33.

Laurent F, Guillaume H, Vincent L, et al. (2007) MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software* 33.

Matsumoto M and Nishimura T (1998) Dynamic creation of pseudorandom number generators. *Monte Carlo and Quasi-Monte Carlo Methods*.

Meyers MA, Chen PY, Lin AYM, et al. (2008) Biological materials: structure and mechanical properties. *Progress in Materials Science* 53: 1–206.

Nvidia CUDA (2011a) C Programming Guide, Version 4.0.

Nvidia CUDA (2011b) API Reference Manual, Version 4.0.

Nvidia CUDA (2011c) C Best Practices Guide, Version 4.0.

Podlozhnyuk V (2007). Parallel Mersenne Twister. Nvidia. Available at: http://developer.download.nvidia.com/assets/cuda/files/MersenneTwister.pdf.

Pro JW, Lim RK, Petzold LR, et al. (2015) GPU-based simulations of fracture in idealized brick and mortar composites. *Journal of the Mechanics and Physics of Solids* 80: 68–85.

Rabiei R, Bekah S and Barthelat F (2010) Failure mode transition in nacre and bone-like materials. *Acta Biomaterialia* 6: 4081–4089.

United States National Institute of Standards and Technology (2001) Federal Information Processing Standards Publication 197. Advanced Encryption Standard (AES). United States National Institute of Standards and Technology. Available at: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

Appendix

For completeness, we include the parameters used in the adaptive and nonadaptive cases described in the results section.

Table 1. Parameters used in the results section.

Adaptive	Nonadaptive
STOL = 0.002	STOL = 0.002
RTOL = 0.03	RTOL = 0.03
ConvergenceRep = 1500	ConvergenceRep = 1500
WindowSize = 3500	WindowSize = 3500
DumpInter = 10e0	DumpInter = 10e0
AdaptInter = 5e3	TEMP = 0.00075
TEMP = 0.00075	InitialStepSize = 0.001
InitialStepSize = 0.001	MinStepSize = 0.001
MinStepSize = 0.0001	MaxStepSize = 0.001
MaxStepSize = 0.02	InitialRotSize = 0.001
InitialRotSize = 0.001	MinRotSize = 0.001
MinRotSize = 0.0001	MaxRotSize = 0.001
MaxRotSize = 0.02	AmpFactor = 1
AmpFactor = 1	InitialLoadStepSize = 0.02
InitialLoadStepSize = 0.02	MinLoadStepSize = 0.02
MinLoadStepSize = 0.005	MaxLoadStepSize = 0.02
MaxLoadStepSize = 0.08	StructSteps = 1
StructSteps = 1	
WindowMul = 20	
WindowUpperMul = 50,000	

Author biographies

Rone Kwei Lim is a PhD student at University of California, Santa Barbara. He received his BS degree in Computer Science and Applied Mathematics from California State University, Los Angeles.

J William Pro is a PhD student at University of California, Santa Barbara. He received his BS degree in Mechanical Engineering from University of Kansas.

Matthew R Begley is a Professor at University of California, Santa Barbara. He received his BS and MS degree in Mechanical Engineering from Pennsylvania State University. He received his PhD degree in Mechanical Engineering from University of California, Santa Barbara.

Marcel Utz is at University of Southampton. He received his Dr Sc. Techn degree from ETH Zurich, Switzerland.

Linda R Petzold is a Professor at University of California, Santa Barbara. She received her BA degree in Mathematics and Computer Science from the University of Illinois, Urbana-Champaign, and PhD degree in Computer Science from the University of Illinois, Urbana-Champaign.