Proceedings of the 2009 IEEE International Conference on Systems, Man, and Cybernetics
San Antonio, TX, USA - October 2009

Data-Parallel Algorithms for Large-Scale Real-Time Simulation of the Cellular Potts Model on Graphics Processing Units

Jose Juan Tapia and Roshan D'Souza
Department of Mechanical Engineering-Enginering Mechanics
Michigan Technological Institute
Houghton,MI,USA
{jjtapiav,rmdsouza}@mtu.edu

***Abstract*—In the following paper we present techniques for data-parallel execution of the Cellular Potts Model (CPM) on Graphics Processing Units (GPUs). We have developed data-structures and algorithms that are optimized to use available hardware resources on the GPU. To the best of our knowledge, this is the first attempt at using data-parallel techniques for simulating the CPM. We benchmarked this implementation against other parallel CPM implementations using traditional CPU clusters. Experimental results demonstrate that this implementation solves many of the drawbacks of traditional CPU clusters, and results in a performance gain of up to 30x, without sacrificing the integrity of the original model.**

***Index Terms*—Cellular Potts Model, GPGPU, Cellular Arrays and Automata, Biophysics**

I. Introduction

Computational Biology has emerged as an area that serves as an investigatory compass for biologists [1]. Although it will not replace *in-vivo* and *in-vitro* experimentation, it enables reduction of the search space through virtual experimentation using *in-silico* models [2]. Typical techniques for simulating biological models include analytical techniques (systems of differential equations) [3] and Monte-Carlo style techniques such as the Gillespie algorithm [4], agent-based modeling [5], and Cellular Potts Models [6]. The Monte-Carlo style techniques are inherently capable of capturing the heterogeneity and stochasticity exhibited in many biological systems. However, they rely on multiple simulation runs to generate dense data-sets for statistical analysis. Moreover, simulating high-fidelity models using these techniques is often beyond the processing capabilities of a single Central Processing Unit (CPU).

The obvious solution to scale beyond the capabilities of a single CPU is to divide computation among many CPUs using parallel computing techniques. However, this solution brings its own set of problems into the mix. Due to the difference in memory bandwiths between Random Access Memory (RAM) and inter-CPU communications, it is often the case that for problems that are not embarrassingly parallel, scaling is more-often than not below par. In fact, in certain situations, adding additional CPUs actually reduces performance due to communication overheads. The second is the cost associated with acquiring and maintaining a cluster. These include costs associated with assembly, installation, and powering of the indiviudal processing nodes as well as the communication infrastructure consisting of high-speed communication networks and routers.

In addition, CPUs are optimized for von-Neuman style computation and have much real-estate on the integrated chip devoted to control. On the other hand, data-parallel architectures such as Graphics Processing Units (GPUs) are optimized for high through-put with a simplified memory architecture and most resources devoted to computing. They are increasingly becoming a powerful and economic alternative to multi-CPU parallel computing systems, particularly for scientific computing. GPUs initially had fixed functionality. However, the demand for customizable computer graphics routines led GPU vendors to introduce programmability. Computational scientists have used this programmability to develop fast algorithms for scientific computations. This technique is generally known as *General Purpose Graphics Processing Unit* (GPGPU) [7].

While GPUs have much higher through-put, this performance advantage is gained through some restrictions on the types of computations that can be performed on individual computing cores of a GPU. While the cost of launching a single execution thread is fairly small, the amount of memory resources associated with each thread are limited. Therefore GPU threads work most efficiently if the code executed is non-blocking with minimum branching. Consequently, algorithms and code developed for CPU execution cannot be directly ported to a GPU. GPU execution requires entirely new sets of algorithms that are optimized for the architecture.

In this paper we describe algorithms for executing the Cellular Potts Model on data-parallel architectures such as the GPU. Data structures have been developed to efficiently handle computation of the local and non-local effective energy terms. We have optimized memory bandwith with proper uses of different memory types such as texture, global, and shared memory. Benchmarks show a substantial performance gain when compared against results obtained from parallelization of the CPM on traditional CPU clusters. This implementation uses Compute Unified Device Architecture (CUDA), an API developed by NVIDIA specifically for non-graphics applica-

978-1-4244-2794-9/09/$25.00 ©2009 IEEE

tions.

In the following sections, we briefly describe the Cellular Potts Model. Next, we review prior work related to parallelization of the CPM. We then briefly review the CUDA programming API. In the next section we describe various data structures, algorithms, and specific optimizations that have been developed to enable CPM simulations on the GPU. In the results section, we benchmark our implementation and compare it to other parallel implementations. The discussion section summarizes the advantages and limitations of this method. In the future work section we briefly describe extensions to be published in subsequent papers.

II. Cellular Potts Model

The Cellular Potts Model (CPM) introduced by Glazier and Glaner [6], is an extension of the large-q Potts model. This lattice based model has traditionally been used for the simulation of models as diverse as those of metallic grains [8], soap, and foam; however in recent years there has been a lot of interest into its application in the simulation of the collective behavior of cellular structures.

In its most general form, the CPM is composed of the following basic elements:

1) A pixel or lattice site which is the basic element of a Cellular Potts Model. Each pixel has a variable which stores a value known as spin.
2) A lattice, or an environment made up of the totality of pixels in the simulation. The pixels may be modeled in 2D or 3D, in a square or hexagonal configuration.
3) Cells (or clusters, depending on what we are simulating) that are groups of pixels that share the same spin. As such, cells become spatially extended, yet internally structureless objects (since they are just made of pixels).
4) Cell membrane: The outermost part of a cell; or in terms of the CPM representation, a pixel which is neighboring a pixel with a different spin.
5) A set of rules that determine the energy of a particular lattice site at a particular moment in time. There are two kind of rules, local rules which are only dependant on the immediate neighborhood and non-local rules whose analysis can potentially span a large region of the lattice, and as such its parallelization is not as direct. In general, any process that can be described in terms of a potential energy can be added to the set of rules of the model.

The algorithm starts by initializing the lattice with cells distributed in space, assigning each cell a random spin value. Next, two neighboring pixels are randomly selected. An exchange of spin between the pixels is then accepted based on on a Monte Carlo probability, such that $T > 0, P(\Delta E) = \{e^{-\Delta E/T} : \Delta E > 0; 1 : \Delta E \leq 0\}$ and for $T = 0, P(\Delta E) = \{0 : \Delta E > 0; 0.5 : \Delta E = 0; 1 : \Delta E < 0\}$, where T represents the effective cytoskeletal fluctuation amplitude of cells in the simulation in units of energy [9]. ΔE refers to the difference in effective energy resulting from the potential change in the spin [6].

A. Calculating the effective energy

For a more detailed explanation of the different types of energies a CPM can include, we refer the readers to Chen et. al. [9] In this section we will only describe the energies we included in our implementation although the framework allows for the easy and inexpensive (in terms of added computational cost) inclusion of different types of energy.

B. Cell-cell adhesion energy

Adhesion energy refers to the net adhesion or repulsion between different cell membranes. It refers to the energy produced by the interaction of pixels with different spins (or boundary pixels). Adhesion is typically calculated using the formula described in Eq. 1, where J is the neighboring energy between lattice sites marked by σ and σ', and $\delta(\tau(\sigma)\tau'(\sigma')$ will be 0 if $\sigma = \sigma'$, and 1 otherwise. This is to ensure that only border lattice sites, or the cell membranes, contribute towards the overall adhesion energy.

$$\sum_{\sigma,\sigma'} J_{\tau(\sigma)\tau'(\sigma')} 1 - (\delta(\tau(\sigma)\tau'(\sigma')) \tag{1}$$

C. Cell volume constraint

A second element that can be introduced into the CPM is a constraint on the intended volume of cells [10]. The cell volume constraint energy is normally calculated using the formula shown in Eq. 2.

$$\sum_{\sigma} \lambda_{\sigma}(v(S) - v_{target}(S)) \tag{2}$$

D. Restrictions not directly related to the effective energy equation

The most important of these type of 'restrictions' is the specialization that different cells in a body perform. Even in the simplest biological processes a number of different cells participate, and those different cells have different structures, behaviors, tasks, and of course physical properties (or constraints parameters)

III. State of the art

Despite the apparent simplicity of the CPM, it is a powerful tool for simulating phenomena as diverse as tumor growth, cell separation, and organ development. The CPM's predecessor, the Large-Q Potts Model has also been used in simulations of soap froths and similar phenomenon [8]. Due to the emergent nature of CPMs, useful simulations typically have very large sizes. In biological simulations, each lattice site must represent 2-5 μm with lattice sizes reaching $10^7 - 10^9$ lattice points. This makes these simulations expensive both in memory requirements and computation time. Typically, the resources required are much beyond what is available on a computer containing a single CPU. Consequently, researchers have used parallel computing using a cluster of CPUs to scale CPM simulations.

One of the first attempts at parallelizing the CPM was that of Wright et al. [8]. They built models for (among other things) grain growth applied to microstructural morphology simulations. They only considered local rules, in particular, cell-cell adhesion energy. Their implementations reached the biggest lattice sizes to date when the article was published. They were also one of the first to demonstrate with results that parallelization does not effect model integrity. However, because of the locality of the rules, their model imposes limitations on the scalability of more complex simulations involving global rules.

Chen et al. [9] solved many of the limitations of Wright et al.'s work. The lattice space was divided into blocks to be processed on different nodes in a CPU cluster. In addition *phantom* regions were included in every block for synchronization. The *phantom* blocks allowed exchange of boundary information between adjacent regions. A checkerboard algorithm was used to limit interference between neighboring blocks and thus minimize inter-node communication. Even then, the cost of synchronization to prevent loss of accuracy at the boundaries causes performance loss. Chen et al. also showed that due to the stochastic nature of the CPM simulation, the accuracy of the model is not compromised greatly even if synchronization is not performed at every time step. This analysis is based on the assumption that lattice site updates are rare occurrences, more so at the boundaries between regions. However, for large-scale simulations, this assumption may not hold true.

The Radom Walker algorithm by Gussatto et. al. [11] is an efficient scheme for selection of update sites. In their paper, Gusatto et al. postulate that the Monte Carlo selection algorithm is highly inefficient because of its random selection nature. This type of selection means that most of the time updates will be attempted at lattice sites that are internal to cells. These sites have no chance of spin flipping. They postulate an algorithm which ensures that most of time only boundary sites are selected by only 'walking' through a cell boundary once a boundary site has been selected. However, their implementation contains several downfalls, one of the biggest ones being that they require a complete lattice copy to be in each of their computing cluster nodes, which obviously limitates scalability.

IV. Scientific computing on GPUs

GPUs were primarily developed as co-processors to handle manipulation of graphics data in computer graphics. Initially, vendors only supported fixed functionality. However, the demand for customizable shading routines led to the development of APIs that allowed computer game programmers to include specialized routines. Computational scientists used the same techniques to use GPUs for scientific computing. The extreme computing power and memory bandwidth of GPUs allowed substantial performance gains over traditional CPU-based implementations of scientific algorithms. Initially, all scientific computations had to be formulated in terms of image rendering. However, since 2007, vendors such as NVIDIA have developed direct APIs to access the computing power of GPUs for non-graphic applications. Our implementation uses NVIDIA's Computer Unified Device Architecture (CUDA) API.

V. CUDA programming model

For a throughout explanation of how the CUDA model works, we refer readers to the official guide [12]. In this section we will limit ourselves to explaining the most basic aspects we need for this article.

CUDA's basic unit of execution is called a kernel,which is the same as a single program that is executed on all processors in the GPU at a given time. Within the GPU, thousands of threads are launched, each one of which handles the execution of one instance of the kernel. Threads are grouped into execution units called blocks, which among other characteristics share a common memory space called `shared memory` that will be explained later. Threads within blocks are organized into warps. The processors in the GPU operate on warps and execute all threads in the warp in parallel. Finally, groups of blocks are organized into a grid which is the totality of the execution unit.

A. The CUDA memory model

The CUDA device memory is organized in a three level hierarchy model. A per thread local memory, a per block (a group of threads) shared memory, and a global memory available to all threads in the context. Shared memory is the fastest, although it has a much more restricted space (approximately 16K per block) and therefore has to be used wisely.

On another hand, local and global memory, while much larger, require their reading accesses to be coalesced in order to optimize reading speed. For instance, if all threads in a warp access a contiguous set of memory blocks, the CUDA architecture will transform it into a single read. Otherwise it will result in one memory read per thread in the block. Unfortunately because of the stochastic nature of the CPM simulation, we cannot ensure that reads will be coalesced. That is why we use a secondary way of accessing global memory, which is texture memory. Textures is an optimized way for either accessing non-coalesced data which has a 2D or 3D like dimensionality in its distribution, or for random access to 1D data.

VI. Data-parallel simulation of the CPM

In this section, we detail the data-parallel implementation of the CPM model. We describe data structures and algorithms that we have developed to enable efficient simulation of large-scale CPM simulation on GPUs. While our implementation is focused on GPU execution, the same ideas can be carried over to other data-parallel architectures.

In general terms, the algorithm can be understood as the pseudocode shown in 1.

Algorithm 1 General Algorithm

1: Initialize variables in the host (CPU RAM)
2: Allocate memory on the device (GPU)
3: Copy initialized variables from the host memory to the device memory space
4: Run a precalculation kernel to identify the initial values of the non-local lattice constraints.
5: **loop**
6: Run Update Kernel
7: Synchronize buffers
8: **end loop**

A. Data structures

In its most general form, the CPM model is built upon three main parts:

- A 3D lattice where the model develops. Each pixel in the lattice contains a attribute that records spin
- Information about the cells (spin, location, size) in the lattice
- The constants and parameters for calculation of the Hamiltonian energy terms (elasticity, adhesion, etc.)

We store data in a manner that maximizes the usable memory space, memory bandwidth and computational speed of the GPU. The main bottleneck in data-parallel execution is the manner in which data is accessed and the resulting latency. Careful planning of data structures and memory access patterns is of utmost importance for achieving the best performance.

The 3-D lattice data is stored in both a so called `cudaArray` for reading purposes, and in a standard array located in global memory for writing.The `cudaArray` is an structure accessed through a texture specially designed for storing data that has 2D or 3D dimensionality. Memory access to the lattice is extremely random (in terms of the sites that we will access with each iteration) because of the stochastic nature of the algorithm. Standard linear memory is made such that when it is accessed by a group of threads in a warp, it must be done in such a manner that all the threads within a block access a single continuous memory segment. In other words, reads must be *coalesced*. Otherwise reading time will be highly inefficient. If we were to directly store the reading array in global memory space, we would incur a lot of overhead because of these conditions. There is no way to make our reads to be coalesced without taking out much of the stochastic nature of the CPM. This coupled with the fact that we are storing a structure that is naturally displayed in 3D into a 1D memory space, further adds to our lack of capability to guarantee coalesced access to memory space.

In contrast, storage with texture access is better suited for non-coalesced, random access like the one we will be using. This is specially true for data structures which are logically displayed in two or three dimensions. Moreover, the CUDA engine ensures that when a memory location is accessed through a texture, the engine will store its neighboring sites on a special texture cache, so that memory bandwidth is optimized. That is, even if access to the lattice sites are random, if we ensure that our reads will be approximately close to each other, our access bottleneck will be greatly reduced.

The NVIDIA CUDA texture implementation doesn't support direct texture writing to CUDA arrays as of version 2.1. Therefore, we are using a second array directly located in global memory for writing purposes, in contrast to the texture which will only be used for reading. This double storage of the data could be avoided in future versions of CUDA when texture writing is implemented. Storing data in texture memory also facilitates visualization of results since it can be bound to a *glBuffer* object to obtain real time display using the OpenGL API. Although this approach seems redundant since we have to update two separate data structures with the same information, doing the data update from the write buffer in global memory to the read buffer in texture memory after every Monte Carlo step is almost negligible in terms of computational cost. After running our program through CUDA's profiling tool, the data update section of the program takes less than 5% of our computation time. This small penalty is eliminated by the huge performance gains achieved through avoiding non-coalesced memory reads from global memory.

The data that we store in each pixel of the lattice in these two structures is the code corresponding to the cell whose volume includes the pixel in question. The code is calculated through the formula

$$code = ID * t + type, ID = \{1..k\}, type = \{0..t-1\} \quad (3)$$

where, ID is a unique identifier assigned by the program, k is the total number of cells in the simulation, $type$ represents some kind of cell type (as described in section II-D) and t is the total of the different types of cells.

For storing the cell array we make use of two linear arrays, stored in global memory whose sizes is equal to that of the maximum number of cells in the environment. Both these arrays contain a copy of the properties every cell in the array has at a given time step (volume, surface area, etc.). It is double buffered, such that one array is used for reading, one for writing and they are synchronized at the end of each Monte Carlo step. However, in terms of access to the reading array, once again we fall into a non-coalesced memory access, because we have no way to ensure coalesced memory access pattern, which CUDA requires for optimal reading time to global memory. To solve this we will wrap our linear array with a texture, which as we mentioned is the best way to access memory when access is non-coalesced. Moreover, in this case there is no sense in storing the cell array as a `cudaArray` since there is no spacial dimensionality to the data. In summary, we map this texture to the read only array, and write directly to the secondary array. We used a double buffer scheme to avoid the race condition where multiple threads could simultaneously update pixels belonging to a single cell and corrupt input data in the process.

We make use of CUDA's constant memory for all of our constant values. Specifically, we used it for storing the lattice dimensions, the adhesion and volume restriction constants and some data inherent to our implementation.

In summary, our memory distribution is as the one shown in Fig 1 in terms of how it is internally stored, and equivalent to the one shown in Fig. 2 in terms of what each structure represents. In this example we have a 4x4 lattice with 2 types of cells, and 4 cells total. Cell 0 represents the Extra Cellular Matrix (ECM), or the medium, and as such its volume is not counted in our model. Cells 1 and 2 are of type 0, while cells 3 and 4 are of type 1. When we apply the equation 3 for the code, we obtain the ID from the type, and we can do the same in reverse through a simple division and modulo operator.

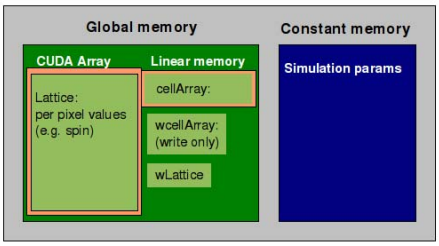


Fig. 1. Internal memory distribution of the data structures. The variables with an orange frame means that they are wrapped in a texture for reading purposes

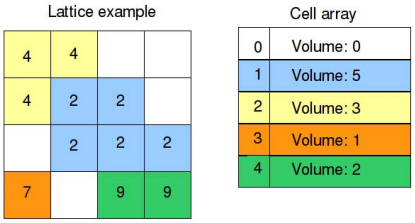


Fig. 2. Memory representation of the data structures

B. Properties initialization

The first step in the simulation is the process of creating the initial state. In this particular case, it consists of computing the volume (counting the number of pixels belonging to each cell). Initilialization is accomplished using a separate kernel that is launched before the main loop. Each thread in the initialization pass concurrently visits every pixel. If here is a cell whose volume includes that particular pixel, the volume of that particular cell is incremented by one. To avoid a race condition where several threads increment the volume of a single cell, we use an atomic add operation which serializes concurrent access to the same variable to avoid data corruption.

C. Potential problems due to spatial sub-divison and concurrent updates of lattice sites

Given the strict hierarchy of the computing resources on the GPU (threads organized in thread blocks with threads in a block able to access common shared memory), it is imperative to assign the processing of the lattice in a way that maximizes memory bandwidth by enabling coalesced memory read, taking advantage of automatic caching in texture memory, and preventing simulation artifacts due to corruption of data.

Ideally for maximum parallelization, we would assign one thread per pixel. However, it is almost always the case that the cells will encompass multiple lattice sites. Consequently, all threads operating on pixels belonging to a single cell will attempt to write to the same memory location. This will effectively cause serialization and degrade performance. Consequently, we have to carefully divide the lattice among thread blocks and threads to maximize computational efficiency.

In addition, it is a requirement for the CPM that within a Monte-Carlo step (MCS), only one lattice site belonging to a cell can change its spin based on the computation of the potential energy. Since multiple threads can potentially process lattice sites belonging to a single cell in parallel, it is quite possible that different threads could potentially change the spin of multiple lattice sites belonging to a single cell using energy calculations based on invalid data.

Because of the above mentioned scenarios we have to be careful with how we distribute the space of the lattice among the threads, and on how we update our values at the end of the each MCS. Another factor we have to take into consideration is the grid-block CUDA distribution. We have to distribute the lattice such that threads in the same block are close together so that we take advantage of the texture cache, and we have to maximize the number of threads per block, and the processor usage such that we don't waste computing power because of bad subdivisions.

D. Handling lattice updates correctly

We handle the concurrent update of the lattice sites using atomic operations. At the begining of the MCS, the input and output cell data buffers are synchronized. Subsequently, when a thread attempts to modify a lattice site, it will first request unique access (using an atomic operation) to the volume values of the cells that are involved in the change. If the values in the two buffers for each cell are different, then it is evident that some other thread has already committed a change to at least one of the cells at some other lattice site. Consequently, the current thread exits without making any changes. If not, it will change the lattice site based on the calculation of the volume and update the volume of the cell and the spin of the lattice site. This update will block other threads from updating any other lattice sites belonging to the two cells.

In Figures 3 and 4 we illustrate with an example how this method works. In Fig. 3 we see that thread (`t1`) is trying to update a lattice site shared between cells 1 and 2, thread `t2` between cell 2 and the ECM, thread `t3` between cells 2 and 3, and thread `t4` between cells 3 and 4. Suppose the lattice update changes that these threads proposed are all accepted, they all try to simultaneously change the values of the output arrays. Furthermore, they will request unique access to the memory location in the Cell array of the cells they are trying to change. So that `t1`, `t2` and `t3` have conflicting access. Let us suppose that `t1` was given access first. It would confirm that no other

thread has updated this cell before, and then proceed to update the cell values in the write buffers and update the lattice spin. When `t2` is given access, it would confirm that the volume of cell 1 has already changed and drop its update. The same will be the case with `t3`, since the value of cell 2 would have already changed it would drop its change as well. When it is `t4`'s turn (since it was separately contending access with `t3` for cell 3) it would confirm that no other thread has changed the values of cells 3 and 4, and subsequently proceed to apply its updates.

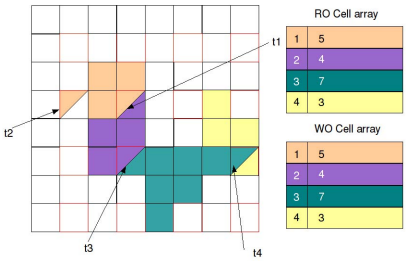


Fig. 3. Update example with more than one update per cell. Lattice change candidates are represented by a divided pixel, where the upper half represents the current state, and the lower half represents the proposed change. $t_k, k = \{1..4\}$ represents different threads.

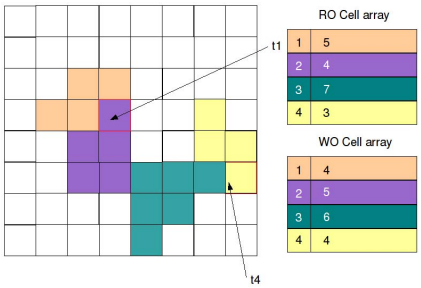


Fig. 4. Resulting lattice after the update cycle. Only the changes from threads t1 and t4 were accepted. The changes are reflected both in the lattice and in the Write Only (WO) cell array.

E. Optimal spatial sub-division of the lattice

The motivation for spatial sub-division of the lattice is two fold. The first is to take full advantage of texture caching by dividing up the lattice into sections assigned to different blocks. This ensures that threads within a thread block are reading memory that is within the vicinity of each other thus maximizing texture cache. The second is to minimize the use of atomic operations which are used to correctly handle lattice site updates. Atomic operations are expensive since they effectively serialize access to the variables involved.

The lattice is divided into subgrids each of which is assigned to a thread block. Within a thread block, the subdgrids are further divided with each division assigned to a separate thread. The region assigned to a thread is further subdivided to enable the use of a checkerboard updating scheme [9]. The checkerboard scheme avoids the scenario where multiple threads are trying to update neighboring lattice sites thus causing extensive serialization due to the atomic operations and consequently degrading performance. In our final optimized configuration, each thread is responsible for the update of a 2x2x2 subgrid of lattice sites. Each of the sites in the subgrid is numbered with a 3 tuple from (0,0,0) to (1,1,1). At the start of every MCS step we will select a 3 tuple at random and pass it to the parallel machine, so that every kernel calculates the exact same lattice for its respective subgrid.

To address the size of the lattice size handled by a thread block, we experimented with the CUDA profiler tool and the CUDA occupancy calculator in order to find out the combination that maximized resource usage while avoiding the spatial issues described before as much as possible. In general, the CUDA guidelines recommend using between 64 and 512 threads depending on the problem, but taking care that the number of threads is a multiple of 64. Experimental results conclude that keeping the block size between 2^7 and 2^8 threads for our lattices maximizes resource usage.

Based on the profiling study, the scheme that gave the best results divided the space such that one thread is responsible of a 2x2x2 subgrid, with each thread block covering a 8x8x16 space (each thread block contains 128 threads). This maximizes resource usage by optimal access to texture memory, maximizing processor usage and having concurrent calculation of as many pixels as possible, and at the same time avoids loops where one single thread has to calculate more than one lattice site per MCS, thus making the whole algorithm effectively parallel, while avoiding the simulation artifacts described earlier.

We illustrate with an example of how the lattice is divided within a 64x64x64 lattice in Fig. 5. The left side of the figure represents how the lattice is divided among all the different thread blocks. Considering that each block covers 8x8x16 pixels, we have to do a point by point division of the two vectors to obtain the total number of blocks (8x8x4 in this example). The right side figure represents a 2D projection of the 8x8x16 subgrid that each block is assigned (with the 16 pixel 3rd dimension hidden). Each color represents the 2x2x2 division (2x2 in the projection) that each thread within the block will be in charge of modifying. As shown, each color is further subdivided in 8 parts (4 in the projection) corresponding to the checkerboard division. In the example, the upper left pixel of each color is highlighted, indicating that the checkerboard algorithm decided that every thread would be dedicated to the processing of the upper left pixel. This entire scheme would be repeated for every thread block.

F. The main algorithm

Each thread starts by selecting a lattice site corresponding to the subgrid it was assigned to. It is worth noting that converting from the threadID number that is assigned by the system to every thread (in short, a 5 tuple ID vector) to the 3 coordinate system that corresponds to the lattice was

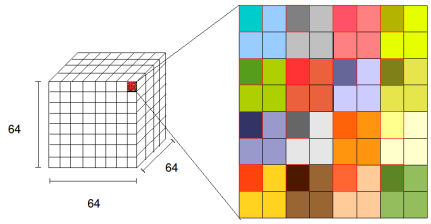
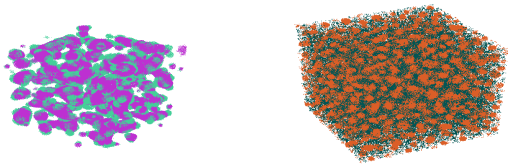


Fig. 5. Example of how the spatial subdivision works. Each color represents a section processed by a different thread.



(a) Cell 1: purple, cell 2: green

(b) Cell 1: orange, cell 2: blue

Fig. 6. Cell sorting algorithm. Addhession constants, in the first example: $J_{(1-1)}$= 15, $J_{(1-2)}$= 7, $J_{(2-2)}$=9, $J_{(\{1,2\}-ECM)}$ = 1 , in the second one $J_{(1-1)}$= 10, $J_{(1-2)}$= 1, $J_{(2-2)}$=5, $J_{(\{1,2\}-ECM)}$ = 3

done using bitwise operators such as shift and logical AND instead of multiplications, divisions and modulo operators in order to minimize computing time. The possible flip that each thread will select is randomly chosen among the immediate neighbors of the lattice site that the thread is analyzing. Once the thread has decided on a possible spin flip, it calculates the resulting differential in effective energy. Both the local and global energies are calculated. For the local energies it is a straightforward neighborhood evaluation (in our implementation we searched for neighbors up to a 2 pixel range, for a total of 125 neighbors), and for the global energy we will build upon the cell property array that we initialized previously.

Once the ΔE is obtained, the spin flip is accepted based on the direct application of the Monte Carlo probability equations described earlier. After this we do a thread synchronization step, in order for all threads to reach this point and do the next operation concurrently. Those threads that rejected a change will return execution control here. The rest that accepted the spin change will request unique access to the cell properties variable in cellArray corresponding to the cells which will be affected by its lattice change. In case the change is acceptable (no other thread has modified the volume of this cell) they will proceed with the rest of the update, and return control. Otherwise they just return control.

Algorithm 2 CUDA kernel

```
    Select a lattice site, and a random neighbor spin site
 2: if the two selected sites have the same spin then
        Kernel return // the lattice sites is not a cell boundary
 4: end if
    Calculate the ΔE
 6: Accept spin flip based on Monte Carlo probability
    Synchronize threads
 8: if update accepted then
        Atomic access volume
10:     if volume not modified then
            Update values
12:     end if
    end if
14: Kernel return
```

VII. Results

Our experiments were run on a NVIDIA GTX 280. The CPU part of the code ran on an Ubuntu 8.04 system, with an Intel Q6600 processor and 2GB RAM.

For display purposes we enabled a real time OpenGL display of the data. The results of some of the simulations are shown in Figure 6. In general, the results show that the algorithm scales linearly, as illustrated in Figure VII, which shows that there is no significant overhead in terms of managing larger quantities of data. Our algorithm computation time on a GTX 280 can be calculated as $time = (5.976 * 10^{-6}) * size$, where $size$ is the lattice size in pixels, and $time$ is the number of seconds required to compute 2000 MCS.

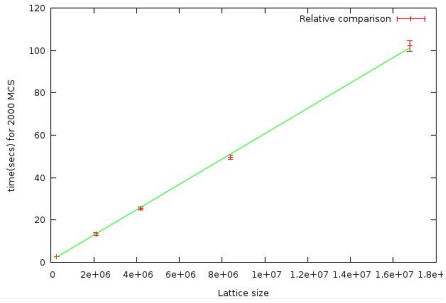


Fig. 7. Performance scaling with different matrix sizes. Steps vs time

A. Benchmarking

We compared our algorithm against two different implementations of the parallel Cellular Potts model. These implementations are characterized, one for being run on a remarkably powerful computer cluster, and the other for introducing a noticeable algorithmic improvement over the classical Monte Carlo random algorithm. The implementation by Chen et. al. was run on the Biocomplexity Cluster at the University of Notre Dame. The cluster consists of 64 dual nodes with two AMD 64 bit Opteron, with 4 GB RAM on each node. The Random Walker algorithm ran on a 4 CPU cluster. The results are shown in Table I, between parenthesis we indicate the number of reported CPU's actually used for a particular simulation (with RW meaning Random Walker and MC Monte Carlo). For our report, we ran each configuration a total 40

TABLE I
COMPARISON BENCHMARK

Model	Grid Size	Cells(10^6)	Iterations	Time(secs)
RW Cluster(4)	1E6		10000	2145±6
MC Cluster(25)	2.25E6	0.09	2000	370
MC GPGPU	2.097E6	0.1	2000	12.08±0.21
MC GPGPU	4.19E6	0.2	2000	25.47±0.61
MC Cluster(9)	9E6	0.36	200	254
MC GPGPU	8.38E6	0.4	200	4.73±0.35
MC GPGPU	8.38E6	0.4	2000	49.63±1.11
MC Cluster(16)	1.6E7	0.64	200	274.5
MC GPGPU	1.677E7	0.8	200	10.02±0.28
MC GPGPU	1.677E7	0.8	2000	98.03±2.06

times with random simulation parameters (e.g. cell adhesion constants, target volume) within a [1,31] range. The reported value is the mean, and the tolerance is the standard error of the mean.

VIII. DISCUSSION

The results show that our methods for data-parallel simulation of the Cellular Potts Model on GPUs outperform CPU cluster based implementations by nearly 30x. There are two main reasons for this. The first is that GPUs are built primarily for high through-put computation with most resources on the chip devoted to computation rather than control. The GPU computing power has shown an exponential growth in recent years because it is achieved by simply adding additional computing cores to the chip. Consequently, they are ideal for a situation where fine-grained parallelization is possible.

The second one has to do with the very nature of CPU clusters, and how they are interconnected. It is very often the case that the main bottleneck in these clusters is the communication bus between each node. Most often they will be connected by a network type bus, and in the best case scenario where they are all located in the same rack, they still have separate memory spaces, so that every bit of communication between the CPU nodes requires a memory synchronization step. The processors on the GPU while not as powerful as the CPU are well suited for CPM type problems because they all share a common memory space that allows for almost free synchronization. Furthermore, proper design of the algorithm regarding the disrtibuting lattice subdivisions among threads will mean that information sharing will be at cache level with all of the memory speed and bandwidth advantages that that it implies.

The combination of the two makes for parallel computing power that is greater than what one would expect when directly comparing the floating operations that a cluster and a GPU can achieve. An AMD Opteron 248 like the one used by Chan et al. in their implementation is approximately 4.4GFlops per CPU, giving a 25 CPU cluster a peak theoretical performance of 110 GFlops. Comparing this to the NVIDIA 280GTX's reported 933 Gflops, simple extrapolation would tell us that the our algorithm should at most perform 9 times faster than the CPU cluster implementation. However our algorithm was shown to run up to 30x faster when compared to the 25 CPU implementation.

Furthermore, the GPU is a far more economical platform compared to CPU clusters. For example, a PC equipped with an Opteron 248 of the type used in one of our reference benchmarks costs around $1000. An array of 25 nodes will approximately cost $25,000 without factoring in maintenance cost, electricity bills, etc. In comparison, a single PC equipped with a cutting edge NVIDIA 280GTX GPU will cost under $1500. Moreover, real-time display is impossible with clusters while it is virtually free on the GPU.

It is our belief that the numbers we have presented in this article will not be limited solely to the Cellular Potts Model, but to many biophysics simulation models that allow for certain level of parallelization.

IX. FUTURE WORK

The framework we have built can be further extended both in terms of the complexity of the simulation, and increasing the number of GPUs in order to demonstrate the direct scalability of these type of algorithms. We would also like to introduce some of the principles of the Random Walker algorithm into our algorithm. Our idea would be to run a preliminary border detection filter through the lattice which would return a lattice containing only boundary pixels. With this information, it would be trivial for a kernel to identify whether a lattice update should be evaluated.

ACKNOWLEDGMENTS

This work was funded by grants CCF-845284 and IIS-840666 from the National Science Foundation (NSF). Any views, opinions and conclusions are those of the authors alone and do not reflect those of the NSF.

REFERENCES

[1] J Southern and et al. Multi-scale computational modelling in biology and physiology. *Prog. Biophys. Mol. Biol.*, 96(1-3):60–89, 2008.
[2] G An. Concepts for developing a collaborative in silico model of the acute inflammatory response using agent-based modeling. *J. Crit. Care*, 21(1):105–111, 2006.
[3] B Oksendahl. *Stochastic differential equations: an introduction with applications*. Springer - Verlag, 2002.
[4] D T Gillespie. Exact stochastic simulation of coupled chemical reactions. *J. Phy. Chem.*, 81:2340–2361, 1977.
[5] G An. Agent-based computer simulation and sirs: building a bridge between basic science and clinical trials. *Shock*, 16(4):266–273, 2001.
[6] F. Graner and J.A. Glazier. Simulation of biological cell sorting using a two-dimensional extended Potts model. *Physical Review Letters*, 69(13):2013–2016, 1992.
[7] J D Owens, D Luebke, N Govindaraju, M Harris, J Kruger, A E Lefohn, and T Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics, State of the Art Reports*, pages 21–51, 2005.
[8] S.A. Wright, S.J. Plimpton, T.P. Swiler, R.M. Fye, M.F. Young, and E.A. Holm. Potts-model grain growth simulations: Parallel algorithms and applications. *Sandia Report SAND97-1925*, 1997.
[9] N. Chen, J.A. Glazier, J.A. Izaguirre, and M.S. Alber. A parallel implementation of the Cellular Potts Model for simulation of cell-based morphogenesis. *Computer Physics Communications*, 176(11-12):670–681, 2007.
[10] N.J. Poplawski. Volume and surface constraints in the Cellular Potts Model. *Arxiv preprint physics/0512129*, 2005.
[11] E. Gusatto, J.C.M. Mombach, F.P. Cercato, and G.H. Cavalheiro. An efficient parallel algorithm to evolve simulations of the cellular Potts model. *Parallel Processing Letters*, 15(1):199–208, 2005.
[12] NVIDIA. *NVIDIA CUDA Programming Guide 2.0*. 2008.