SIMULATING LARGE VOLUMES OF GRANULAR MATTER

by

Nicholas Boen

B.S., Kansas State University, 2014

A THESIS

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computer Science
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2016

Approved by:

Major Professor
Daniel Andresen

Abstract

Modern techniques for simulating granular matter can produce excellent quality simulations, but usually involve a great enough performance cost to render them ineffective for real time applications. This leaves something to be desired for low-cost systems and interactive simulations which are more forgiving to inaccurate simulations, but much more strict in regards to the performance of the simulation itself. What follows is a proposal for a method of simulating granular matter that could potentially support millions of particles and several types for each particle while maintaining acceptable frame rates on consumer level hardware. By leveraging the power of consumer level graphics cards, effective data representation, and a model built around Cellular Automata a simulation can be run in real time.

Table of Contents

List of Figures

Acknowledgments

I would like to express my gratitude toward those that have been a tremendous help to me thusfar. Not only do I have my committee of Dr. Daniel Andresen, Dr. Mitchell Neilsen, and Dr. William Hsu to thank for helping me out along the way, but also the rest of the KSU CS department faculty that have shared their knowledge and wisdom with me over the years. I also have Jake Ehrlich to thank as a good friend and sounding board for the ideas and methods that went into this paper. And of course I have my family whose relentless support and desire to see me succeed are a large part of why I have made it this far. Finally, a special thanks to all of my friends who have made these last few years so enjoyable.

Chapter 1

Introduction

Granular matter such as sand, grain, pollen, and gravel fill the world around us. They are particles that are massive enough to be seen with the naked eye and hold many interesting properties that make computer simulation a challenge. Dense granular matter like sand and dirt can be used to form heaps, mounds, hills, dunes, and mountains. On the other hand pollen is light enough to be cast in the air and be carried by air currents while behaving aesthetically similar to a gas. Despite this visual similarity, granular matter behaves very differently from gases where temperature and pressure play a much bigger role. While many of these materials will act as solids at rest, they become more like liquid once set into motion. Liquid-like behavior often occurs when the maximum angle of stability of a mound of material is exceeded and an avalanche occurs. During this event everything between the maximum angle of stability and the angle of repose starts moving rapidly downhill. Avalanche behavior differs from true liquid behavior as underneath the rapidly moving upper layers remains the quasi-static particles that form the rest of the heap [1].

On top of these pseudo-state behaviors granular matter also expresses some properties of its own. Arching is a phenomenon exclusive to granular matter where flow through a hopper will cease as the particles form a stable archway and can cause problems for many automated systems [2]. Granular materials with particles of irregular size will often undergo a sort of segregation, similar to the Brazil nut effect. Larger objects or more dense objects will tend to be nearer to each other than they were when the motion began during violent and prolonged shaking [3]. While modeling and simulating various states of matter from liquids to gases can be done quite effectively, granular material still largely stands out as a challenge to balance quality and computational performance.

Simulation of granular matter is a problem that has been approached using several different methods over the last few decades. Most of these approaches start with a goal of achieving as realistic of a simulation as possible [4; 5]. These simulations do well with approaching the goal of realism and limiting computational complexity, but have yet to fully utilize some options for faster computing. Other methods do focus on performance and even manage to maintain a representation that is realistic, interactive, and similar to experimental observations [6]. It is possible that the scope of simulations that can be run by those methods may be limited, but what can be done is impressive for being efficiently streamlined and interactive.

Analyzed below is an implementation of a method that uses a Cellular Automata (CA) model that is updated in parallel using a CUDA compatible GPU. The major benefit of such a system is that it vastly increases performance with a moderate cost to simulation accuracy. This accuracy may not hold up to numerical simulations like in [6], but will suffice in being visually appealing and extensible. This sort of solution is useful for systems that might require many prototype simulations to be done that need a high degree of variability or some capacity for interaction.

Approaching the problem in this way allowed me to simulate upwards of 1 million simulation particles at approximately 105 frames per second (FPS). The speeds reported throughout this paper will be based on the simulation system alone; the rendering system was never a focus and so it utilizes few optimization techniques and was used simply to visually verify the simulation itself. That being said, the rendering portion in its current state constitutes approximately 80% of the per frame run time and the above simulation of 1 million cubes can be run at a about 13 FPS with rendering enabled. While the simulation itself appears more rudimentary the granular matter involved still responds to gravity and will still flow around objects and itself. Furthermore the implementation outperforms many other solutions due in part to its simplicity and its leverage of the underlying hardware, its own data representation, and its use of precomputation.

The rest of this paper first discusses related work in Chapter 2, and then describes my implementation in Chapter 3. Chapter 4 describes how I evaluated my system and presents the results. Chapter 5 presents my conclusions and describes future work.

Chapter 2

Related Work

2.1 Interactive Cellular Automata

Pla-Castells, García-Fernández, and Martínez have already shown that models of Cellular Automata for simulating granular matter are viable for real time applications. Their methods give them phenomenal performance and produce realistic heaps and sand piles. In addition their numerical simulations hold up to other simulations and do not vary significantly from other experimental results. The methods outlined by Pla-Castells et al. in [6] are developments upon their previous work done in [7] which also expands further on the BCRM method for simulating realistic sand piles [8]. Models described in [6] expand on their previous simulations by including two modes of interactivity. One mode involves how a model of their system responds to an external force being applied while the other concerns a model for pressure distribution at the base of the system. These methods of interactivity cover a sizable spectrum of simulations such as those concerning avalanches and sandpile stability, but do not address simulations where the positions of discrete particles matter and may be difficult to predict with continuum mechanics. Simulations that involve problems such as buried explosions, hopper flows (hour glass), integration of different granular types, or some of the other simulations seen in [5] and [4] may benefit from a different approach. I make no claims that my implementation can hold up to experimental observation and is notably simplistic in comparison to other simulations. What it does show is how dry, incompressible sand (and potentially other material) can be simulated in a general fashion while maintaining high frame rates that CA models have been shown to have.

2.2 The Discrete Element Method

One common numerical method is to use discrete particles as a model and rely on collision resolution and the summation of acting forces on each particle to support the simulation. The Discrete Element Method (DEM) is a great way to model oddly shaped or irregular granular materials and even lends itself well to parallelization. The DEM is typically implemented in two dimensions to keep collision resolution simpler. Expanding it to three dimensions increases the complexity and the quantity of computation needed for resolving collisions. While this can be done it will usually come with the cost of longer simulation times [9].

2.3 Granular Material as a Fluid

Another method that achieves good results includes the use of re-purposed or restructured fluid solvers that add cohesion to liquid models. This sort of approach attempts to use continuum mechanics to model a mass of granular materials as a whole. A notable use of this method is from Zhu and Bridson where they modeled a body of sand as a cloud of particles while accounting for internal friction and boundary friction. This coupled with their incompressibility assumption makes for an excellent model of cohesive granular materials like mud or wet sand. With this method Zhu and Bridson were able to get a performance of approximately 12 seconds per frame while simulating 433,479 particles on a 2 GHz G5 processor [4].

2.4 Continuum and DEM

A more recent approach to a granular material solver implemented a combination of characteristics of the previous two methods. Narain, Golas, and Lin constructed a solution that treats a granular material as a continuous fluid and couples it with large and compressible simulation particles to act as “clumps” of particles. These clumps can then be split or merged with surrounding particles as needed [5]. The result of this method is an excellent representation of dry granular material that can be blown around, scattered, and mixed while behaving in a realistic fashion. Performance of the work of Narain et al. falls between 6 and 33 seconds per frame. The fastest simulation ran with 403,000 simulation particles (5.2 million render particles) computing at 6.1 seconds per frame. This speed was achieved using a 3.33 GHz Intel Core i7 processor with 5.8 GB of RAM [5].

Chapter 3

Implementation

A natural progression was made toward a CA model because maximizing performance was the main goal, but that still leaves the number of operations that need to be done as a big problem. Modern CPUs are good at complex calculations and even perform well at doing a lot of them. However, this problem of simulating particles (or 'cells' in this case) is an N-body problem at its core. While the calculations that each cell must perform tend to be no more complex than contextual comparisons with neighbors they are so numerous that the performance seen from Zhu and Bridson and from Narain et al. is quite impressive. Trying to solve the problem with faster CPUs becomes less realistic as the cost of better compute performance rapidly grows for diminishing returns.

A great solution in many respects is to tackle the problem with more CPUs. By using cheaper CPUs that perform most calculations at an acceptable rate and then having them all work together on different parts of the problem the problem space itself can be reduced many fold. This approach is analogous to what a GPU does and while a single CPU might be able to handle up to 8 threads, a GPU can typically handle thousands. Breaking the problem up, distributing it to separate cores, and doing other forms of maintenance for making sure things run smoothly is not free, but if the problem is defined appropriately then this latency can be masked with a higher throughput. The general idea of this is occupancy which is the ratio of active threads and maximum threads on each multiprocessor. This value is expected to fluctuate during runtime, but it is ideal to maintain as high of an occupancy as possible during peak load on the GPU. The overall effect is that even though the individual CPUs may not be particularly fast the throughput of large groups of threads finishing simultaneously results in a more effective system.

The benefits from GPU computing can be great, but it can complicate the problem. A solution for a single-threaded CPU, is slow, but is easy to grasp and to implement. Parallelizing a solution, whether it is over a few threads in a CPU or thousands of them in a GPU, can still introduce classic parallelization pitfalls like resource management and race conditions. It would be best to avoid these where possible, but to mitigate potential problems I needed a simplistic algorithm that would still lend itself well to parallelization. Additional benefits to a basic CA model include being relatively easy to reason about and that simple rules that govern singular cells or even groups of cells can lead to extraordinarily complex behavior. A typical example of this is the “Game of Life” which is a two dimensional CA that only uses a few rules to determine if a cell lives or dies in the next generation based on the living cells around it. The “Game of Life” CA (and even other, simpler automata) have been proven to be capable of producing a universal Turing machine [10].

The core of this implementation requires that the CA model is updated and stepped as quickly as possible; each ‘generation’ of the automata will represent a frame. To do this I used Nvidia’s CUDA platform for mass parallelization and scalability. This was a preference based on experience; other implementations using different platforms or frameworks like OpenCL should perform comparably. I leveraged the generous allowance of memory available on many of Nvidia’s GTX line of graphics cards for precomputation. Bit-packing methods are used repeatedly to maintain as much performance as possible. The end result is an implementation that operates on large numbers of particles at speeds acceptable for interactive simulation while still having room to be further optimized and expanded to fit a broader spectrum of simulations. Many of the expansions discussed were not implemented and officially analyzed because this research stands as a “worst-case” analysis to make evaluation simpler and more straightforward.

3.1 CUDA and Cellular Automata

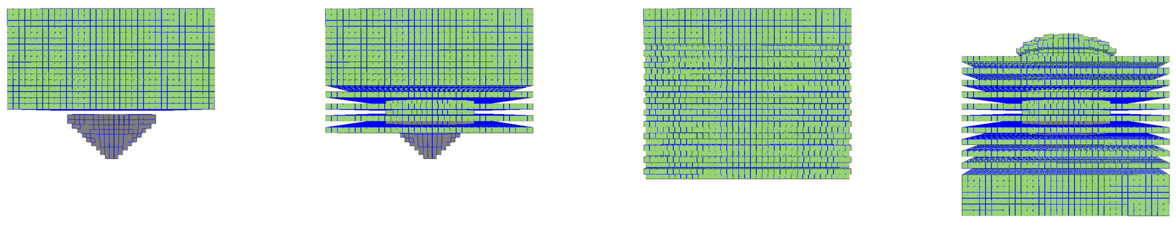
The decision to use CUDA was made from the beginning on the grounds that compatible methods and models would benefit greatly from the large scale parallelization that it offers. The main obstacle is that classical implementations for algorithms on a CPU tend to differ from their GPU counterparts in a few substantial ways. GPU's are incredibly efficient at doing similar operations on large scales, but experience a sharp decrease in performance when many different operations need to be done by several threads. The reason for this performance loss is that GPU's step groups of threads (a warp in CUDA) through the operations together. When a thread diverges then the rest of the warp is stalled until that thread is stepped to a point of convergence and then the rest of the warp will follow. Branching is typically fine if it can be guaranteed that an entire warp will follow the same path, but the more that the threads in a warp diverge relative to each other the worse the overall performance will be. This means that any sort of branching in an implementation is best avoided or applied in cases that fit as many threads as possible to mitigate the overall effect.

Another issue that can severely impact CPU based implementations is that GPU's are well separated from the rest of the hardware in a system. Data and parameters that individual kernels might need from the CPU side must be transferred at what is the slowest transfer rate involved with GPU computing: from the host side to the device side [11]. The issue with data-transfer rates is less of an issue overall than thread divergence, but can pose a real problem if CA results need to be saved or retrieved for rendering. In many cases this can be mitigated while rendering by using CUDA interoperability with graphics libraries like OpenGL. CUDA can work in this way by sharing memory space on the graphics card so that buffer data rarely, if ever, needs to transferred to the host computer. Saving a simulation may require a more involved approach and may required the compression of data on the graphics side before being transferred in appropriately sized batches. Granularity of the simulation could also be adjusted as each frame is generated as a function of the previous frame, but may not require every frame when revisiting the simulation once it has been recorded.

The goal of the research outlined in this paper was to implement a model of simulating granular material that at least had potential to be interacted with at a reasonable frame rate. A reasonable frame rate being based on current interactive media where 60 frames per second (FPS) is ideal but as low as 24 FPS is choppy yet tolerable. Pla-Castells et al. already showed with their work that CA models had the capability of simulating granular material in real time [6]. What I set out to do was use a CA model to create a more general simulation of granular material that could still be viable for interactive simulations while being willing to sacrifice some degree of accuracy in the simulation itself.

The basic CA model is easy to reason about and fits nicely into the lock step process that GPU threads use. Zhu & Bridsons approach using continuum mechanics would also work well across a GPU while avoiding the difficulties that accompany discrete particles in this type of simulation. The main reasons that I went with the CA model instead are that it would be faster for me to implement and the options for optimizing it were more clear to me. It may not always be possible to completely remove the issue of thread divergence completely from the computation of a cells resultant state. For most cases the CA can function normally and only the rules that govern updates of specific cell states need to be re-factored and optimized. As will be discussed further into this section the overall computational cost of computing cell rules is removed via precomputation and will actually play little role in the frame-to-frame updates.

One of the many challenges to this work was actually designing rules for a single cell CA (a ‘single cell CA’ being an automata that updates a single cell based on its direct neighbors) that simulates sand. The rules for simulating gravity are straightforward enough to simply be moving cells down each frame if a cell was not already occupying that space. Granular flow, the motion that creates heaps would cause problems with race conditions as now cells could be trying to move on top of each other. To solve this issue I used a block cell automata known as a Margolus neighborhood as described by Toffoli and Margolus [12]. Essentially the cellular automata is broken down into *blocks* or *neighborhoods* and each block is updated as an atomic unit. Cells can then be represented as individuals and stored as such, but when they are updated they are partitioned into grids containing *2*x*2*x*2* cells. Individual regions or neighborhoods can then be denoted as having a specific state which encapsulates the cell states and positions in that neighborhood. Using Margolus neighborhoods in this way I can isolate cells that affect each other and update them as a collective group rather than individuals.



(a) Start of simulation **(b)** 20 steps in **(c)** 50 steps in **(d)** 100 steps in

Figure 3.1: Simulation in 100 Steps
*35*x*35*x*17* granular cells as they fall onto 863 solid cells that form a bowl. (a) Generation 0 - takes place in a *35*x*35*x*35* grid. (b) Generation 20 - The bowl is still empty but granular cells are piling up around the edges of the bowl and falling into it. Other granular states fall as if in a vacuum. The gaps here are caused by the Margolus neighborhood updating, only the neighborhoods that consist of a granular cell above a void cell will display any form of gravity. (c) Generation 50 - bowl is mostly filled. Gaps are closed because the size of the grid does not line up with the Margolus neighborhood at this specific offset on this frame causing cells on the interior to be updated but some cells on the edge have to wait until next frame to fall or otherwise update. (d) Bowl is filled and now sand will continue to pile up into a mound that extends over the bowl. The excess sand in this case will slough off onto the rest that will eventually bury the bowl itself.

For Margolus neighborhoods to work there needs to be an offset of the partition on each frame. If the partition was left to divide the same cells into the same neighborhood then the entire system will simply be divided into several static subsystems. In order to be effective the neighborhoods must also share cells between each other. To prevent any anisotropy during simulation the partitions will need to be offset by a single cell along each potential axis. For a 2 dimensional simulation this would mean 4 different positions (vertical, horizontal, and both diagonals), but for a 3 dimensional simulation the number of offsets increases to 8. The overall effect of this is that cells are allowed to move over adjacent boundaries so that static subsystem division is avoided. The partitions now follow an 8 step cycle which could be called a super frame that consists of 8 individual frames computed all in the same way, but with different partition offsets. A ‘frame’ as it is used in the implementation is one step in this 8 step cycle. Artifacts of this process can be seen in Figure 3.1, specifically Figure 3.1b and Figure 3.1c. As the kernel scans it breaks the buffer of cells into a partition of Margolus neighborhoods which often do not line up perfectly with edge boundaries at some point in the cycle of 8 offsets.

One potential benefit of this Margolus neighborhood implementation is that it has removed the need for a double buffer that many CA models use. Double buffering is a technique typically used to prevent the update process from overwriting data from the previous generation that is still needed to compute the next generation and is common for single cell CA. The Margolus neighborhood breaks the buffer into atomic partitions that can fit into faster cached memory and updated as a whole. That is not to say that double buffering would not be beneficial as it could allow the rendering and updating processes to be decoupled more and multiple buffers can be added for expanded functionality.

The workload of updating the buffer is spread across the GPU itself. Each horizontal layer of the partitioned buffer is assigned a block via the CUDA platform. For each of those layers 256 threads are used per block to compute the updates to the neighborhoods. The number of threads was chosen based on attempts to keep the multiprocessor thread occupancy as high as possible. CUDA operates by running active warps on a Streaming Multiprocessor (SM) and graphics cards will usually have a few of these. Occupancy is the ratio of active warps on an SM and the maximum number of warps that an SM can support. CUDA attempts to interleave warps to hide latency wherever it can and Occupancy is an indirect way of measuring how effective it is. As such it is desirable to maximize this Occupancy wherever possible. Fortunately CUDA provides a tool specifically for calculating the Theoretical Occupancy that a kernel can maintain based on how many resources are used. Resources are finite and refer to the threads per block, shared memory, and number of registers that each multiprocessor can supply to the blocks they are running.

3.2 Data Representation

How data is represented in this system plays a large part in how it can be expanded upon while minimizing the costs of doing so. Individual cells are represented by a single enumerated state. The current implementation can support a total of 8 cell states. Only the 5 states ‘void’, ‘solid’, ‘liquid’, ‘gas’, and ‘granular’ are included in the system and only the 3 states void, solid, and granular are actually being simulated. The void state is simply a representation of blank space and is a default state. Solid states are immovable rigid bodies that do not update under any rules so that they can act as barriers or form useful structures like bowls and hoppers. Liquid, gas, and granular states are meant to represent their indicated states of matter, but only granular states are currently implemented.

The main buffer itself is an array of 32-bit integers that represent each cell. A neighborhood as previously defined is a collection of 8 cells in a *2*x*2*x*2* configuration and is used as an intermediate data structure with several convenience methods. Neighborhoods are constructed in threads as a kernel scans across the buffer and is represented by an array of 8 integers. The neighborhood is capable of converting to and from a 32-bit integer representation where the least significant 24 bits are used as an array of 8 values with each value being represented with 3 bits and corresponding to a single cell in the neighborhood.

In order to perform an actual update a ‘transition’ is computed. Transitions are represented by a single integer where the 24 least significant bits are used to store the transition itself, allowing for a compact set of 8 3-bit values. They are similar to how neighborhoods are stored, but actually represent something else. These values are used in a lookup table where each index of the 8 cells in a neighborhood are used as indices to specific cells and the resulting values represent the index to move the neighborhoods value at the original index to. Basically transitions encode which cells need to be swapped from one generation to the next and their values can be retrieved using only a few shifts and bitwise operations. When a transition is applied to a neighborhood it will move values around the *2*x*2*x*2* structure of the neighborhood according to how the transition was generated.

This method may seem a bit convoluted so why not just generate the actual result and update it that way? While it is true that using transitions like this results in about 16 more accesses (half for reading, half for writing) into the temporary neighborhood array it allows the program to expand. For example, the system could have multiple concurrent buffers that map properties to specific cells allowing several different types of granular matter to exist that behave slightly differently in certain situations.

The global lookup table is a critical component of the entire system. As previously mentioned neighborhoods represent a collection of 8 cells and can be converted to an integer that effectively represents the *state* of the entire neighborhood. By making the global lookup table a mapping between neighborhood states and their appropriate transitions an update can technically occur as quickly as it would take to extract a value at an index in a large array followed by a few more reads and writes.

3.3 Precomputation and Memory

The first operation that occurs when the program is run is that the entire lookup table for neighborhood transitions is precomputed. Only 3 states are actually simulated, but an effort was made to have the lookup table allocate enough space to support 8 states. While generation time of the permutations of this larger lookup table will vary based on the rules that are implemented, the efficiency of using this will not differ greatly from a fully implemented rule set for 8 states including complex interactions.

With 8 possible states for 8 cells there are 8^8 possible permutations of the neighborhood so the lookup table will contain 16,777,216 elements. Using 32-bit integers as elements the result is a lookup table that will consume 67.12 MB. Though realistically only 5 states are used at the moment so only a 1.5 MB lookup table is necessary. Graphics cards with 1 GB to 2 GB of VRAM are fairly common and Nvidia cards with this amount of VRAM can be cheap relative to the typical cost of a computer. This 67.12 MB chunk of memory should not be cause for much concern as performance here will be much more limited by the size of the buffer rather than the size of the lookup table.

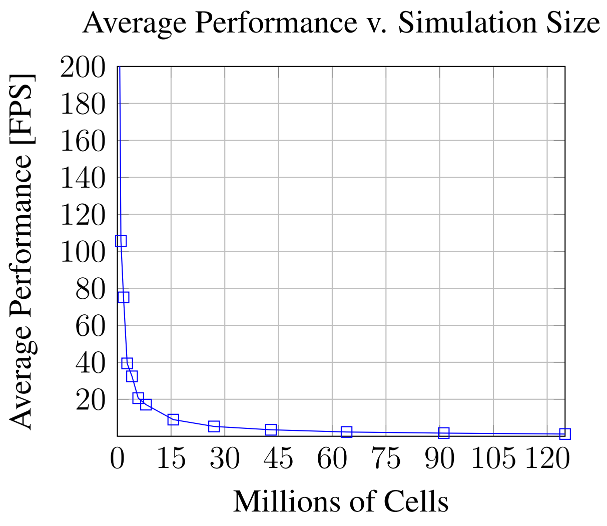
One caveat might be that CUDA has a built in termination time on kernels that is enabled by default. If a kernel runs for longer than 2 seconds it will be killed and the application will seize. This may cause issues on slower systems if the kernels cannot complete the generation of the lookup table in time. While there is currently no contingency for this in the implementation, the lookup table generation can simply be spread across multiple kernel calls to avoid this. Alternatively the time limit can be removed completely, but depending on how the program is used that may not be a realistic expected action for users to perform. Rather than precomputation these values could instead be memoized during run time, but precomputation prevents unexpected slow downs when several values need to be computed at once. Memoization would also mean that more involved update rules would directly impact performance. Alternatively the lookup table generation could be offloaded onto a different program entirely. Such a system could allow for much larger and more complex rules that get compiled and saved to system memory. When the simulation starts it could then load the saved system memory from system memory to GPU memory and exchanging the cost of generating the lookup table with copying it and only generating it once.

Part of the reason only 24 bits of a 32 bit integer are used to represent a neighborhood is because of the explosion in necessary resources that comes from storing more information. If only 1 bit were added for each cell then it would fit perfectly in an integer so space could be maximized and then up to 16 states could be supported. Unfortunately even adding one more state makes it clear that this will not be possible. At the moment the lookup table requires every possible permutation of the 8 cells in a neighborhood to guarantee full coverage. When another state is added then this sort of lookup table will consume over 170 MB (9^8 permutations) of memory to simulate 9 states and 400 MB (10^8 permutations) for 10 states. This could potentially be optimized for space by rotating the *2*x*2*x*2* cells around a specific axis in $\frac{\pi}{2}$ increments and using the smallest state representation as the result. Using rotations to remove duplicate entries in that fashion would only result in a $\frac{1}{4}$ reduction to the overall size of the array; this optimization might make simulating up to 10 states feasible, but returns diminish quickly.

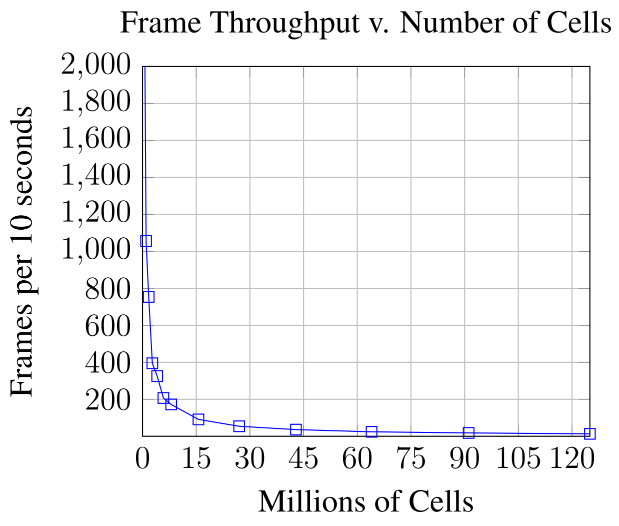
To actually generate the lookup table itself CUDA is used to iterate over all 16,777,216 states. The kernel constructs a grid of 4,096 blocks each encapsulating 256 threads. Each thread will compute the transition of 16 states by starting with the state that the current index represents and applying gravity and then flow rules before storing that transition in the lookup table. This way the index of the lookup table is as much a part of the computation itself as the transition that is stored there and is integral to the rapid retrieval of successive neighborhood states.

Chapter 4

Evaluation



(a) Values range between [1.2, 1665] FPS



(b) Values range between [13, 16636] frames

Figure 4.1: FPS Performance Graphs Related to Simulation Size
Update performance and frame throughput. Successive simulations were set up to be similar using an NxNx$\frac{N}{2}$ granular blocks, 863 solid blocks, and the rest were left void. Updates occur without consideration of the state of the individual cells. Graphs are scaled and not all data points are visible, but the trend is clearly preserved. (a) The average frame rate of simulations of varying sizes over an unspecified amount of time. Relatively inaccurate, but shows the $\frac{1}{x}$ trend. (b) Frame throughput of the first 10 seconds of running the simulation after the lookup table has been generatied. Shows the same $\frac{1}{x}$ trend.

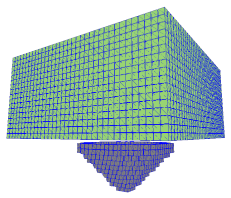
4.1 Environment

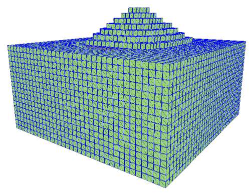
Performance here is typically a measure of frame rate using frames per second (FPS). Seconds per frame, the inverse, may be used at times to draw comparisons between the work done by Zhu and Bridson and Narain et al. as well as demonstrate worst-case complexity. A frame here represents a single update of the entire cell buffer at a single offset of the Margolus neighborhood partition.

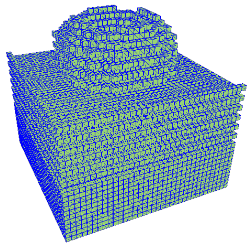
Testing was done on a GTX GeForce 960M which has 4 GB of VRAM and 1096 MHz base clock rate. The mobile graphics card itself also supports CUDA compute capability of 5.0 even though the solution this paper uses was compiled with compute capability of 2.0. While this maintains the backwards compatibility of the solution it is possible that further performance increase could be gained by utilizing a higher compute capability. I am not entirely familiar with the potential gains of doing this, but predict that it would be much less significant than tackling some of the bigger issues of this solution.

Frame rate was computed in a few different ways, but for all values rendering was disabled. First by averaging the time to update the previous 5 frames and letting a simulation run for a lengthy but approximate amount of time. Next was a more rigid accounting of frame throughput where the number of frames computed is counted and output after 10 seconds after the lookup table is generated. This accounting was averaged across 5 trials for each of the 10 values used and produces a graph very similar to the first method.

The actual program that I developed runs in the worst case on every update by updating every neighborhood every frame which makes efficiency testing of the update method a bit easier. The implementation makes no effort to reduce the number of updates that occur each frame. Intentionally avoiding this optimization allows me to establish a base line of performance for the system itself that later optimizations can compare to and it makes the computational complexity of the update a little more obvious. Avoiding update reduction is also convenient because the configuration of the simulation itself will have less of an impact, if any, on performance. The simulation configuration that can be seen in Figure 4.2a and Figure 3.1a was used for N cells involves a large block of NxNx$\frac{N}{2}$ elements with a granular state falling around a floating rigid 'bowl' consisting of 863 solid cells. There is a slight increase in performance when the entire system 'settles' or


(a) Start of simulation


(b) End of simulation


(c) Midway through the simulation from above

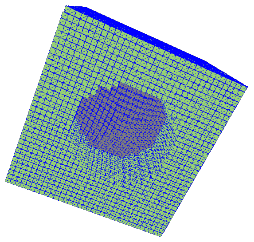

(d) Midway through the simulation from below

Figure 4.2: Sample Simulation Environment
Shows a standard simulation that was used to benchmark the implementation. (a) The start of the simulation is displayed after the lookup table has been generated. The NxNx$\frac{N}{2}$ block of granular cells can be seen floating above the 863 solid cells that form a bowl all in a NxNxN grid. For this simulation $N = 35$. (b) This is the end state. The solid bowl is completely buried and has been filled to the brim and more with granular cells. The granular cells continue to form a mound above the bowl while the rest rolls off into the body of granular cells below that are contained by the simulation bounds. (c) A look from above midway through the simulation. Here artifacts of the Margolus neighborhood can be seen around the edges as the offset does not fall on neat boundaries for this grid size. It can also be seen how the granular cells stack up around the edge of the bowl forming a sort of hopper. While this is not quite realistic, it does display hopper-like behavior similar to dry and fine sand as particles in the middle of an hourglass fall quickly and particles that can cling to the edges stay behind longer. (d) Same generation as (c) that shows the view from below. Granular cells 'fall' as if they were in a total vacuum and are only affected by gravity and an arbitrary flow rule. Here they display that they fall around the bowl, but over time will flow to swallow the space underneath the bowl as would be expected of falling sand.

becomes static which I suspect is a slight benefit to caching that occurs.

4.2 Results

Figure 4.1a and Figure 4.1b show the frame rate and frame throughput that were gathered from testing respectively. It can be seen that the graphing of frames per second express a clear $\frac{1}{x}$ relationship. When this graph is inverted to show seconds per frame, as seen in Figure 4.3, a linear relationship emerges between the time it takes to compute a frame and the number of cells being updated.

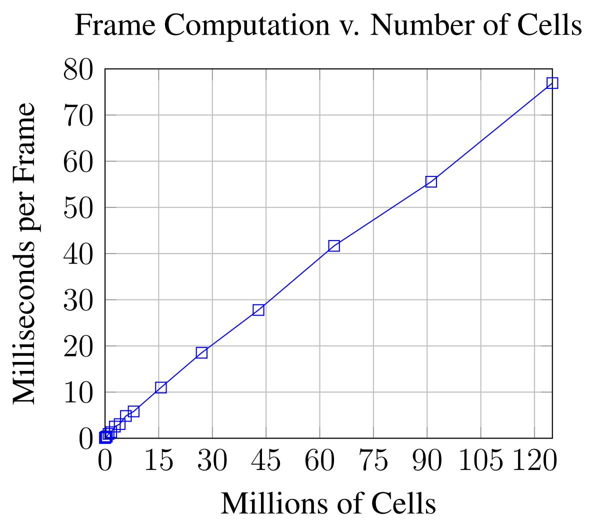


Figure 4.3: Frame Generation Time Related to Simulation Size

This linear complexity of the update is expected; once the lookup table is generated an update to single cell becomes fairly straightforward. A *2*x*2*x*2* Margolus neighborhood of 8 32-bit integers is constructed by retrieving the appropriate cells from the main buffer. The neighborhood is then converted into an integer format by encoding each value as a 3-bit integer so that the state of the entire neighborhood is represented as the least significant 24 bits in a 32-bit binary integer. The encoding process entails a small series of bitwise operations and the indexing of an array of length 8 and should be a constant time operation. This encoded integer is then used as an index into the lookup table where all of the possible state transitions have been precomputed. The result is a transition that is then applied to the original *2*x*2*x*2* neighborhood and written back to the main buffer to complete the update. For *N* neighborhoods this update maintains a linear time complexity.

The solution described here was constructed around the heavy use of a consumer level graphics card to simulate a CA model of granular matter. A simple to implement and cheaply scaled update method is used to simulate large numbers of simulated particles.

The actual rules used for progressing the simulation are precomputed in the lookup table. While this makes rules harder to be reactive to contexts that are bigger than the Margolus neighborhood it does mean that there are few consequences to more complicated update rules aside from additional loading times at the start of a simulation. Additionally the system can be expanded to support up to 8 states for the basic rules or even sacrificing a few states to support a larger neighborhood.

The renderer used takes advantage of a few optimization techniques such as hardware instancing and CUDA interoperability so that data is moved as little as possible, but does little in the way of any culling. With only these basic optimizations about 1 million cells can be simulated at approximately 105 FPS.

This implementation was meant to be a simplistic and scalable simulation of granular matter. As such it performs well in those respects, but finds itself lacking in the accuracy of the simulation itself. Overall the implementation performed better than expected and shows promise for being useful as a potential boilerplate for expanding upon as an interactive granular simulator. Also of note, Zhu and Bridson mention that their simulation costs are linearly proportional to the quantity of simulated particles [4]. Should their method or derivative methods (like the work of Narain et al.) be implemented in conjunction with a GPU the increase to performance may be on the order of what is seen here. The work done here may be indicative of the sorts of benefits that their method or derivative methods (like the work of Narain et al.) may see should they be implemented in conjunction with a GPU.

At the moment the implementation may appear a bit crude, but visual techniques combining shaders and 3D textures as well as marching cubes to smooth voxels could create a more appealing visual. For large enough sizes this system may also start to struggle to find enough consecutive memory for the main buffer. Simulating at sizes large enough would be quite slow as 125 million cubes consumes over 550 MB of memory with the lookup table included and can still barely maintain over 1 FPS.

The results here show that the performance benefits from using GPU's to simulate granular matter using a model of Cellular Automata can be large. Performance of this system is outlined in terms of the update function alone. Rendering the results in real time or transferring the rendered frames from GPU memory and saving them to a drive will have a large impact on overall performance. This sort of approach may also be more appropriate for budget conscious systems. A high level of performance can be obtained with a low cost to memory on a consumer level system.

The program itself is written specifically in CUDA and is therefore only compatible with CUDA enabled Nvidia graphics cards. However, the methods described here could be implemented for another framework that is a bit more robust such as OpenCL. There may be a bit of a performance loss in edge cases and where data transfer is concerned, but otherwise an OpenCL program should perform about as well as a CUDA program [13; 14].

Chapter 5

Conclusion

Simulating granular matter is costly in terms of computing power and can take a reasonable amount of time to generate a complete simulation. This cost is a barrier to cheaper systems and interactive simulations. By precomputing as much as possible and focusing on representing the data efficiently it is possible to implement a solution that is fast, cheap to scale, and conceptually simple. Coupling this simplicity with the brute power of a GPU results in an effective solution for interactive simulation in granular materials. Not only is the method proposed here effective on weaker hardware, but it should scale linearly along with memory access times and cache sizes of more expensive hardware with minimal modification. With the ability to simulate over 8 million cubes this solution effectively handles the size of the problem and only stands to gain from optimization and expansion.

5.1 Future Work

There are a few ways that this implementation could be further improved. At the moment the system blindly updates all cells as it scans across the grid. This action is quick because of the mass parallelization of the GPU, but could potentially be made faster with the use of some sort of spatial data structure. For example an octree could be used to denote large regions of neighborhoods with completely void states or states that do not change when updated so that updates occur only on the apparent moving parts of the simulation. Alternatively an octree could also be used to denote large regions of similar or the same states in order to make updates of larger portions of the grid. Leveraging the potential benefits of more recent compute capabilities for graphics cards and using more effective GPU optimization techniques like shared memory and simplifying kernels may be an effective method of optimization. It also stands to be shown how gases and liquids could also be added and simulated. The lookup table for all 8 possible states is constructed and populated so the implementation of rules for these states would have little to no impact on the performance of the simulation overall.

An extension of this problem could be implemented by utilizing more of the remaining memory space for auxiliary buffers. Only 8 states can be emulated with the current system, but it would be possible to have multiple buffers that support different functions. In this way a vector map could be used for combining continuum mechanics with Cellular Automata to produce a more realistic simulation. Environmental effects like wind or heat could be implemented or a buffer could support more specific elements of materials. An example of that might be boiling water to steam or introducing liquid gallium to solid aluminum to produce a powdered or granulate aluminum. While that may not be a perfect analogy of liquid metal embrittlement, other effects like this could potentially be simulated. Alternatively the number of states that the lookup table considers could be reduced in favor of expanding the Margolus neighborhood representation itself. By doing this it may be possible to use a taller neighborhood (i.e. *2*x*2*x*3*) in order to simulate granular materials with either a greater angle of repose or more cohesive particles like wet sand, mud, and snow.

Bibliography

[1] Heinrich M Jaeger and Sidney R Nagel. Physics of the granular state. *Science*, 255(5051): 1523, 1992.

[2] Kiwing To, Pik-Yin Lai, and HK Pak. Jamming of granular flow in a two-dimensional hopper. *Physical review letters*, 86(1):71, 2001.

[3] Azadeh Samadani, A Pradhan, and A Kudrolli. Size segregation of granular matter in silo discharges. *Physical Review E*, 60(6):7203, 1999.

[4] Yongning Zhu and Robert Bridson. Animating sand as a fluid. In *ACM Transactions on Graphics (TOG)*, volume 24.3, pages 965–972. ACM, 2005.

[5] Rahul Narain, Abhinav Golas, and Ming C Lin. Free-flowing granular materials with two-way solid coupling. *ACM Transactions on Graphics (TOG)*, 29(6):173, 2010.

[6] Marta Pla-Castells, Ignacio García-Fernández, and Rafael J Martínez. Interactive terrain simulation and force distribution models in sand piles. In *International Conference on Cellular Automata*, pages 392–401. Springer, 2006.

[7] Marta Pla-Castells, I García, and Rafael J Martínez. Approximation of continuous media models for granular systems using cellular automata. In *International Conference on Cellular Automata*, pages 230–237. Springer, 2004.

[8] J-P Bouchaud, ME Cates, J Ravi Prakash, and SF Edwards. A model for the dynamics of sandpile surfaces. *Journal de Physique I*, 4(10):1383–1410, 1994.

[9] Richard P Jensen, Peter J Bosscher, Michael E Plesha, and Tuncer B Edil. Dem simulation of granular meditructure interface: effects of surface roughness and particle shape. *International Journal for Numerical and Analytical Methods in Geomechanics*, 23(6):531–547, 1999.

[10] Stephen Wolfram. Computation theory of cellular automata. *Communications in mathematical physics*, 96(1):15–57, 1984.

[11] Mark Harris. How to optimize data transfers in cuda c/c++. *NVIDIA Developer Zone*, 2012.

[12] Tommaso Toffoli and Norman Margolus. *Cellular automata machines: a new environment for modeling*. MIT press, 1987.

[13] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of cuda and opencl. In *2011 International Conference on Parallel Processing*, pages 216–225. IEEE, 2011.

[14] Kamran Karimi, Neil G Dickson, and Firas Hamze. A performance comparison of cuda and opencl. *arXiv preprint arXiv:1005.2581*, 2010.