ECE1724 Project Final Report: *HashLife on GPU*

Charles Eric LaForest

April 14, 2010

Contents

List of Figures

List of Tables

1 Introduction

Cellular automata (CAs) are interesting parallel distributed systems with only local interactions. They can express great complexity emerging from very simple rules [11] and have been posited as a discrete model of physics [12]. I hope that further improvements in computing CAs can transfer to other parallel distributed systems such as neural networks.

Conway's Game of Life [2] is a well-know 2D cellular automaton system. It's algorithm is simple, well-understood, and the structures which emerge from it have been extensively studied and cataloged [8], up to and including implementations of Turing-complete systems.

Although the direct computation of the next state of a Life CA is straightforward, the computation time increases linearly with the size of the CA being computed. To improve its performance, Bill Gosper created the HashLife algorithm [3][7], which accelerates computation by several orders of magnitude. Using HashLife, it becomes possible to rapidly calculate quintillions of iterations over quadrillions of cells on commodity uniprocessors.

The goal of this project is to see if a GPU-based implementation of HashLife is feasible and practical. Although the algorithm seems ill-suited to a GPU, the speedup obtained on a single CPU is such that even within the limitations of a GPU, the additional parallelism and greater memory bandwidth might improve the performance of the algorithm and grant a further speedup over a direct evaluation of a Life CA.

2 Related Work

There have been several implementations of Life on GPUs [4][6][9][13][5], but none using anything but the original algorithm. It is possible to use SIMD tricks to represent several CA cells within a single memory word and thus evaluate them all at once [1], but they fundamentally do not alter the underlying algorithm. The best-known implementation of CAs is the freely available Golly platform [10], which implements both Life and HashLife, as well as its own variant called QuickLife.

3 Life Algorithm

The basic algorithm of Conway's Game of Life is simple:

1. Assume a 2D rectangular grid of cells, with each cell having eight neighbours.
2. Cells may be initially either Alive or Dead.
3. If a live cell has fewer than two or more than three live neighbours, it dies.
4. If a dead cell has more than two live neighbours, it comes to life.
5. Else, the cell remains in its current state.

This algorithm is usually implemented using two grids, the state of one updating the state of the other in alternation. The edges of the grid can either be considered as dead cells, or may loop back to the opposite side. Although simple, the basic Life algorithm does an amount of work and uses an amount of memory proportional to the number of cells that must be updated. Some Life patterns take a large number of cycles and cells to evolve into interesting states.

3.1 HashLife

Bill Gosper invented the HashLife algorithm [3][7] which replaces the direct 2D representation and evaluation of the entire CA with a quad-tree to compress the spatial representation, evaluates the tree nodes recursively, and uses a hash table to memoize the result of the recursive evaluation function at each level of the quad-tree. As the quad-tree and the memoized evaluation find the common patterns in the CA, they accelerate its evaluation by several orders of magnitude.

3.1.1 Quad-Tree Representation

In HashLife, a quad-tree node represents a $N \times N$ space. It contains four child node pointers (clockwise: nw, ne, se, sw) to nodes representing its $\frac{N}{2} \times \frac{N}{2}$ sub-spaces, and one result pointer to the $N \times N$ node which represents the next state of the whole current node, if known. Note that for a given node, the result node is only valid for the $(N-1) \times (N-1)$ nodes inside, as the next state of the edgemost cells cannot be computed without more neighbours. This detail is what makes HashLife difficult to implement, as I'll explain shortly.

This recursive description bottoms-out when N is equal to four, as it is the smallest space where the central cells have enough neighbours to have their next state evaluated. There are only 2^{16} such 4×4 leaf nodes, and thus all possible cases can be enumerated practically as 2D arrays and their result pointers pre-computed using the basic Life algorithm.

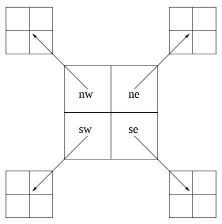

Figure 1: Quad-tree representation of cell space

3.1.2 Recursive Memoized Evaluation

To evaluate a given quad-tree node, the HashLife algorithm first checks if the result pointer is set. If so, then the corresponding node pointer of the parent node is replaced with the child's result pointer. If the result pointer is not set, then the algorithm recurses down into the child nodes to see if they have been pre-evaluated.

Unfortunately, a simple recursion into each child node of the current quad-tree node would not yield the correct results: an $N \times N$ space can only update its inner $(N-1) \times (N-1)$ cells since the cells on the edge do not have enough neighbours to evaluate their next state, thus there would be gaps of cells that are never updated.

This is where the HashLife algorithm gets complicated: Instead of recursing down the actual child nodes at a given level, we must temporarily create a new set of nine overlapping $\frac{N}{2} \times \frac{N}{2}$ child nodes with overlapping $(\frac{N}{2}-1) \times (\frac{N}{2}-1)$ updated cell areas which then compose the updated $(N-1) \times (N-1)$ cells of the starting node at the current quad-tree level. This new node is entered into a hash table, and both the appropriate child pointer of the parent node and the result node of the original current node are updated with the address of the new node. For even greater speedup, the algorithm can then recurse into the updated $(N-1) \times (N-1)$ space of cells to calculate several iterations ahead.

Despite this complicated recursion, it is easy to see that all occurrences of a pattern reduce to a single quad-tree node in a hash table, and that the computation of any cyclic sequence of patterns (and there are many of those) eventually reduces to a series of pointer dereferences. The end result is a tremendous spatial and temporal compression of CAs.

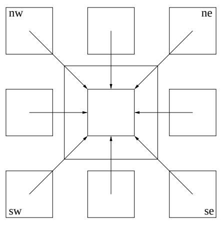

Figure 2: Exploded view of 9-tree of overlapping nodes used for computation of the parent node (largest square in centre)

4 Implementation

In the interest of simplicity and to keep the data representation algorithm-independent, I represented the cells with unsigned bytes, despite each cell having only two states. Packing 32 cells into one unsigned integer would have been much more efficient, but might have posed problems and irregularities later which would undermine performance comparisons.

4.1 Basic Life on CPU and GPU

The CPU version of Life simply iterates through the cells, reads the status of its neighbours, and updates the state of the same cell in another grid. The position of the old and the updated grids is exchanged and the process repeated. For simplicity, the edge cells are not updated and considered fixed. The only optimization is that the algorithm traverses the grid in a cache-friendly, column-major order: updating a cell preloads the cache with most of the neighbours of the next cell.

The GPU version is a direct translation of the CPU version. Each cell is assigned one thread, and its new state is written to another grid. The host then calls the same kernel again with the location of the grids switched. Shared thread memory is not used, but since each thread will access cells in the same pattern (the three neighbouring cells above and below their own cell, plus the east and west neighbours) the global memory accesses are coalesced.

4.2 'HashLife' on the CPU

I attempted to implement HashLife on the CPU with an eye to its future GPU implementation: without recursion and without dynamic memory allocation. While this design decision would have resulted in a more even comparison of the CPU and GPU implementations, it made the already complicated recursion even more difficult to understand and implement. There were too many simultaneous constraints.

I had to give up and settle on implementing only the bottom-most layer of HashLife: a pre-computed 64k-entry array of all possible 4×4 cell grids, each with a result index pointing to another array entry which gives the next state of the 2×2 inner cells.

The CPU implementation iterates as before on the whole grid, but skips every other cell in both the X and Y dimension, making the base unit a 2×2 array of cells. The neighbours of these cells are used to compute the index of the corresponding 4×4 precomputed array entry, which then points to the entry with the updated value of the 2×2. Thus the loop iterations perform lookups of overlapping 4×4 cell spaces instead of directly applying the rules of Life to each individual cell.

4.3 'HashLife' on the GPU.

The GPU implementation of the bottom layer of HashLife is a direct translation from the CPU. However, the CPU pre-computes the array of 4×4 spaces and copies it to the GPU global memory. Since this array is read only, this memory is then mapped to a texture to benefit from hardware caching.

Instead of each cell having a thread which updates its state, only every other cell in the X and Y axis has a thread assigned to it. This reduces the number of threads by 75%, which eliminates redundant traffic and wasted work running threads which would do nothing. As the threads still read the north, south, east, and west neighbours of each 2×2 space in the same sequential pattern, the memory accesses remain coalesced.

5 Methodology

Evaluating the performance and correctness of Life and HashLife is easy: compare the wall-clock times and ensure that identical starting patterns yield identical final patterns after the same number of iterations. Even a single error in the state of a cell can permanently alter the evolution of the overall grid, so the initial test of correctness is to visually compare the evolution of different implementations over a small grid (32×32 for example), using a pattern with a known progression. The algorithms are applied uniformly over the entire grid, therefore a correct progression on a small grid implies a correct progression on any larger grid. Finally, since both algorithms are purely discrete and integer-based, there are no issues with floating-point precision on the GPU.

6 Evaluation

Experimental Setup All implementations of Life and HashLife were tested on a 2.83GHz Core 2 quad-core CPU with 4GB of RAM (unknown type and speed), although there is only one thread of execution. The GPU used is an NVIDIA GeForce GTX 280 with 1GB of RAM. All the grids were square with 22,000 cells on a side. Given one byte per cell, this translates to about 484MB, enabling two such grids to fit in the video card memory at one time, allowing alternating updates.

All cells are initially set to ’dead’ in both grids, and a small test pattern is then inserted at the centre of the first grid. This pattern is the R-Pentomino, which is a simple pattern of 6 cells which is known to evolve for 1103 iterations, at which point it settles into a number of stable, cyclical patterns.

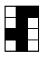

Figure 3: R-Pentomino (black cells are Live)

The grids are then copied to the GPU (when used), and no further data transfers occur during measurement. Only the kernel is restarted at each iteration. The number of generations is set to 100 for CPU implementations and 4000 for the GPU, to achieve similar benchmark times of 300 to 500 seconds. Table 1 shows the number of milliseconds for each iteration of each algorithm, and the resulting speedups are worked out in Table 2.

Speedup and Memory Bandwidth The raw speedup granted by moving Life to the GPU is 38.4x. Relative to that Life baseline, the speedup granted by the partial implementation of HashLife is 1.53x on the CPU, and only 1.18x on the GPU. The GPU implementation was already memory bound but benefited nonetheless from caching of the texture memory, since the total number of cells accessed remains otherwise the same. The benefit is relatively small as the texture caches do not reduce latency, but merely free up some global memory bandwidth. In contrast, the CPU caches do reduce latency and thus achieve a greater speedup.

Thread Dispatching It was expected that reducing the number of threads by 75% (prior to using texture memory) would have a positive performance impact since otherwise most of the threads would do nothing at all, reducing the utilization of the processors. However, there was no significant difference in performance, further supporting the idea that access to global memory is the limiting factor. This lack of change when going from 484 million threads to only 121 million also suggests that the dispatching of threads is extremely efficient.

Milliseconds per Iteration		
	CPU	GPU
Life	4612	120
HashLife	3006	102

Table 1: Milliseconds per iteration of Life and HashLife implementations.

Speedups				
	Life (CPU)	Life (GPU)	HashLife (CPU)	HashLife (GPU)
Life (CPU)	1	0.0260	0.652	0.0221
Life (GPU)	38.4	1	25.1	0.850
HashLife (CPU)	1.53	0.0399	1	0.0339
HashLife (GPU)	45.2	1.18	29.5	1

Table 2: Speedup comparisons between all implementations of Life and HashLife. The speedups are for the rows, relative to the columns.

7 Conclusions

My inability to really implement HashLife on the GPU is disheartening, and the resulting speedup not much better than a direct translation of Life to the GPU. It was however a salvageable educational experience as I learnt a little more about the CUDA memory hierarchy and corrected my assumption that dispatching threads costs time as it does on operating systems running on CPUs.

In retrospect, I made one major mistake: I tackled both the CPU and GPU implementations of HashLife at the same time. I would have been better off to implement a plain HashLife on the CPU with recursion and dynamic memory, and then find out how to transform it to operate without them.

In fact, I can now see that the solution is to collapse the quad-tree spatial representation into the temporary 9-tree used for computation. It may appear inefficient, but the overhead of five extra pointers per node would have been rapidly absorbed as the grid size increased, and the static allocation would have benefited performance also. Any remaining memory on the GPU would then be used for a large static hash table which would use linear probing to resolve collisions (e.g.: finds the next empty spot), and eject old entries in a Least-Recently-Used manner.

Also, the recursion could have been converted into iteration with some help from the host CPU:

1. Run the kernel on one thread at the root of the tree to check if it needs updating or if the next state is in the hash table, return the status back to the CPU.
2. Based on that status, run the kernel on up-to four threads on the next level down of the quad-tree ’subset’ and check if updating is needed.
3. If part of the tree needs updating, run the kernel on nine threads on the ’temporary’ overlapping 9-tree child nodes used for computation. Return their status to the host CPU.
4. Repeat until all ’quad-tree’ nodes are visited, and all 9-tree nodes are updated, then start over at the root node.

Despite the overhead of repeatedly launching kernels and copying data back to the CPU, the algorithm would still behave as it should and try to reduce the computations to hash table lookups.

References

[1] FINCH, T. LIAR: Life in a register. `http://fanf.livejournal.com/81169.html`, January 2008. Accessed March 11th 2010.

[2] GARDNER, M. Mathematical Games. *Scientific American 223* (1970), 120–123.

[3] GOSPER, R. W. Exploiting regularities in large cellular spaces. *Physica D: Nonlinear Phenomena 10* (January 1984), 75–80. Online at `http://dx.doi.org/10.1016/0167-2789(84)90251-3`.

[4] PERUMALLA, K. S., AND AABY, B. G. Data parallel execution challenges and runtime performance of agent simulations on GPUs. In *SpringSim '08: Proceedings of the 2008 Spring simulation multiconference* (San Diego, CA, USA, 2008), Society for Computer Simulation International, pp. 116–123.

[5] PETRICEK, T. Accelerator and F# (II.): The Game of Life on GPU. `http://tomasp.net/blog/accelerator-life-game.aspx`, Dec 2009. Accessed March 15th 2010.

[6] RACARR. GPU Life. `http://blogs.gnome.org/racarr/2009/01/26/gpu-life/`, January 2009. Author pseudonym only, Accessed March 15th 2010.

[7] ROKICKI, T. G. An Algorithm for Compressing Space and Time. *Dr. Dobb's* (April 2006). `http://www.drdobbs.com/java/184406478`, Accessed March 15th 2010.

[8] SILVER, S. A., AND MARTIN, E. Life Lexicon. `http://www.bitstorm.org/gameoflife/lexicon/`, December 2006. Accessed March 15th 2010.

[9] TARDITI, D., PURI, S., AND OGLESBY, J. Accelerator: using data parallelism to program GPUs for general-purpose uses. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2006), ACM, pp. 325–335.

[10] TREVORROW, A., AND ROKICKI, T. Golly. `http://golly.sourceforge.net/`. An open source, cross-platform application for exploring Conway's Game of Life and other cellular automata. Accessed March 15th 2010.

[11] WOLFRAM, S. *A new kind of science*. Wolfram Media Inc., Champaign, Ilinois, US, United States, 2002.

[12] ZUSE, K. *Calculating Space*. MIT (Proj. MAC), Cambridge, Mass., 1970. MIT Technical Translation AZT-70-164-GEMIT.

[13] ZVOLD. Conway's "Life" and "Brian's Brain" cellular automata using GPU. `http://zvold.blogspot.com/2010/01/conways-life-and-brians-brain-cellular.html`, January 2010. Accessed March 15th 2010, Author's name is unknown, pseudonym given.