CUDA and OpenMP Implementations of Conway's Game of Life

Eric Biggers

December 16, 2011

1 Introduction to the Game of Life

Conway's Game of Life is a well-known example of a cellular automaton. A cellular automaton consists of a regular, uniform lattice of sites (or cells), where each site takes on one value of a discrete variable.[8] This lattice can potentially be any number of dimensions. The state of the automaton changes at discrete time steps. For each step of time, the state of each cell is changed by following a set of rules that determine the state of that cell in next generation, based on its current state and the current state of some set of cells defined to be its neighbors. This is done for every cell in the grid, conceptually at the same time.

The Game of Life was designed by British mathematician John Conway in 1970.[1] It is a two-dimensional cellular automaton with very simple rules. The cells are square, and the grid is considered to be infinite. Each cell has 8 neighbors surrounding it. For each generation, the following table of rules is considered to detmine the state of each cell in the next generation:

Table 1: Rules for Conway's Game of Life

		Number of living neighbors			
		< 2	2	3	> 3
Cell state	*Alive*	Die	Continue to live	Continue to live	Die
	Dead	Remain dead	Remain dead	Come to life	Remain dead

This might be expressed in pseudocode as follows:

Algorithm 1 ALGORITHM NEXTGENERATION(*cell*)

```
num_neighbors ← cell.COUNT_NUM_NEIGHBORS()
if cell.ISALIVE() then
  if num_neighbors < 2 or num_neighbors > 3 then
    cell.setState(DEAD)
  end if
else
  if num_neighbors = 3 then
    cell.setState(ALIVE)
  end if
end if
```

The rules of Life are simple, yet they were chosen carefully. While some cellular automata are not very exciting because their cells all die out very quickly, or their alive cells grow explosively forever, Conway's Game of Life is interesting because its rules allow for a wide range of behavior. There can be steady-state sets of cells (still lifes), oscillating sets of cells (oscillators), and moving sets of cells (spaceships), as well as complex changes that occur when these patterns interact.[1]

The Game of Life has seen much interest over the years because despite its simple rules, it is possible to find very complex patterns. In fact, Life is Turing complete, so given enough time and a Life grid set up in an appropriate initial state, it is theoretically possible to compute anything that can be computed. In 2000, a Turing Machine was implemented in Life,[5] and in 2010, a Universal Turing Machine was implemented.[6] Also in 2010, a pattern that constructs a copy of itself was invented. However, it destroys itself in the process, so "real" artificial life that can replicate itself multiple times has yet to be created.[7]

2 Algorithm

The most straightforward algorithm for Life on today's real computers is to store the grid in memory with alive cells as bytes set to 1 and dead cells as bytes set to 0. Two copies of the grid would be required so that the next generation could be calculated from the previous generation without disrupting the values needed for the calculation.

Since the Life grid is supposed to be infinite, this method of representing the grid has to restrict the grid size. Ideally, these limits would be set to an appropriate value for the pattern so that its development is not disrupted by reaching the boundaries.

A C implementation of this algorithm might look like this:

Source Code:

```
1 void compute_next_generation(const unsigned char** grid_in,
     unsigned char** grid_out, int width, int height) {
2     for (int y = 1; y < height - 1; y++) {
3         for (int x = 1; x < width - 1; x++) {
4             unsigned char num_neighbors =
5                 grid_in[y+1][x-1] + grid_in[y+1][x] + grid_in[y+1][x
                      +1] + grid_in[y][x-1] + grid_in[y][x+1] + grid_in
                      [y-1][x-1] + grid_in[y-1][x] + grid_in[y-1][x+1];
6             grid_out[y][x] =  ((num_neighbors == 3) || (cell(x, y)
                 && num_neighbors == 2));
7         }
8     }
9 }
```

The code takes in a parameter `grid_in`, which is a 2D byte array that contains the current generation. `width` and `height` are the dimensions of this array. The next generation is written into the array `grid_out`. For simplicity, the code simply does not compute the cells on the boundaries. (Another way of handling the boundaries would be to have them wrap around so that cells going off one boundary would appear on the other side.)

3 Parallelization

It is easy to see how the computation of Life using the above algorithm can be parallelized. In an m by n grid, $m \times n$ calculations need to be performed, but none of them are dependent on each other. Therefore, theoretically there could be up to $m \times n$ processors used to perform the calculation, completely in parallel.

3.1 OpenMP

If coding in C, C++, or Fortran, it is easy to parallelize the above code listing by adding an OpenMP directive that directs the compiler to parallelize the loop. In C, it would look like this:

Source Code:

```
1 void compute_next_generation(const unsigned char** grid_in,
     unsigned char** grid_out, int width, int height) {
```

```
2     #pragma omp parallel for
3     for (int y = 1; y < height - 1; y++) {
4         for (int x = 1; x < width - 1; x++) {
5             unsigned char num_neighbors =
6                 grid_in[y+1][x-1] + grid_in[y+1][x] + grid_in[y+1][x
                      +1] + grid_in[y][x-1] + grid_in[y][x+1] + grid_in
                      [y-1][x-1] + grid_in[y-1][x] + grid_in[y-1][x+1];
7             grid_out[y][x] =  ((num_neighbors == 3) || (cell(x, y)
                  && num_neighbors == 2));
8         }
9     }
10 }
```

At the point in the program where the pragma is placed, a default number of threads (usually the number of processors available on the machine) is created, and the iterations of the loop are divided up evenly among them. No synchronization is needed except for the implicit barrier at the end of the `for` loop.

If multiple generations are to be computed, it would be more optimal to create threads at a level before the `compute_next_generation()` function, so that the threads could work on multiple generations with lightweight synchronization between each generation before being destroyed. This is what I did in my implementation.

3.2 GPU

As a highly data parallel problem, the computation of the next generation of a Life grid is quite appropriate for implementation on a graphics processing unit (GPU).

Performing general-purpose computations on GPUs is a fairly new, developing area, and not many programs take advantage of it yet. One framework that shows some promise is OpenCL[2], which is an open specification that allows programs to execute on GPUs. It currently works on both the latest ATI/AMD devices and Nvidia devices.

However, the more developed framework at this point is Nvidia's CUDA (Compute Unified Device Architecture). CUDA is the name of a parallel computing architecture for Nvidia's GPUs, and its software development kit comes with an API (libraries and headers) and a compiler for a language very similar to C in which functions to be executed on the GPU can be written.

The Life grids can be created in the regular system memory, and then copied to a CUDA-capable device using the CUDA API. Then, the implementation of the

`compute_next_generation()` function would run code not on the CPU, but instead launch a "kernel" to be executed on the GPU.

The following is a kernel for computing the next generation of a Life grid:

Source Code:

```
1  __global__ void kernel(const unsigned char** grid_in, unsigned
       char** grid_out, int width, int height) {
2      int x = (blockIdx.x * blockDim.x) + threadIdx.x;
3      int y = (blockIdx.y * blockDim.y) + threadIdx.y;
4      if (x > 0 && x < width - 1 && y > 0 && y < height - 1) {
5          unsigned char num_neighbors = grid_in[y+1][x-1] + grid_in[
               y+1][x] + grid_in[y+1][x+1] + grid_in[y][x-1] + grid_in
               [y][x+1] + grid_in[y-1][x-1] + grid_in[y-1][x] +
               grid_in[y-1][x+1];
6
7          grid_out[y][x] = ((num_neighbors == 3) || (cell(x, y) &&
               num_neighbors == 2));
8      }
9  }
```

It it fairly similar to the C code, as the CUDA language is based on C. The `__global__` keyword indicates an entry point to a kernel.

The following code is the actual `compute_next_generations()` function that is equivalent to the serial and OpenMP versions given earlier. It is a wrapper around the kernel.

Source Code:

```
1  void compute_next_generation(const unsigned char** grid_in,
       unsigned char** grid_out, int width, int height)
2  {
3      dim3 block(16, 16, 1);
4      dim3 grid(width / block.x, height / block.y, 1);
5      kernel<<< grid, block>>>(grid_in, grid_out, width, height);
6  }
```

Perhaps the strangest thing about the CUDA code is that there is no `for` loop. This is because if the kernel is called with the correct number of thread blocks and threads per block, each data element will be assigned to 1 thread. The calculation is expressed merely for 1 element, but this computation is being done in a massively parallel manner by the launching of the kernel with the appropriate number of thread blocks and threads per block. This would be a very poor way to write the

program on a CPU. But on the GPU, writing the program in this way is possible, and actually preferable, because the the CUDA architecture uses very lightweight threads, implemented in hardware. Large numbers of these threads can be executed at the same time, as the GPU consists of a number of multiprocessors, each of which can command 32 threads simultaneously.[3]

In the above code, grid of Life cells is divided into thread blocks of size 16 x 16 x 1. Therefore, each thread block contains 256 threads. As the code is written, the grid dimension must be evenly divisible by 16 for all grid cells to be calculated. The first lines in the kernel have the thread figure out which grid element it is responsible for; then, the remainder of the kernel does that calculation.

4 Implementation

4.1 The Platform

The LittleFe platform provided a way for me to try implementing Conway's Game of Life using CUDA and OpenMP. The LittleFe is a "portable cluster" that has 6 motherboards connected by an ethernet switch. Each node has a CUDA-capable GPU and a dual-core "Intel Atom" processor. However, despite the availability of 6 nodes, I only used one node because I was not doing any distributed parallelism (e.g. with MPI). The operating system is a 32-bit version of the Bootable Cluster CD, which is based on Debian GNU/Linux.

4.2 The Code

The implementation of Life that I wrote is interactive. The generations are rendered using OpenGL, and the GUI uses the GLUT library, which is a common library used for cross-platform OpenGL programs of low to medium complexity. The arrow keys can be used to move around, the plus and minus keys zoom in and out, respectively, and Space and 'b' speed up and slow down the simulation, respectively. By default, it will start up with the R Pentonimo pattern, which is a small but interesting pattern. Give the `--help` argument to see all command line options. (A name of a file containing a Life grid can be given, but there are only 2 Life file formats supported, and not all files of those formats will open correctly yet.) See the README in the code directory for a little more information.

I designed the program so that, to some extent, different implementations of the grid representation and computation could be used. The files `cuda.c` and `openmp.c` contain the same externally visible functions, such as a function to advance a number

of generations in the simulation, but their implementations are different. A CUDA-powered executable and an OpenMP powered executable are built by linking the rest of the object files with these different implementations.

The source code files are the following:

- `life.c`, which contains `main()`; Creates the window and starts the event loop, or runs a timed test.
- `callbacks.c`, which contains the callbacks for the GUI (display, key pressed, etc.).
- `cuda.c`, which is the CUDA implementation of the computation.
- `cuda-kernel.cu`, which goes along with `cuda.c` and consists of the code that launches the kernel, and the kernel itself.
- `openmp.c`, which is the OpenMP implementation of the computation.
- `fileio.c`, which currently can read in LIFE grids in the `.rle` (run length encoding) and `.lif` formats, which are two file formats for life patterns. I include a directory `patterns` that contains some patterns in these formats. (Note: I put together the file input functions quickly and they will not open every file correctly. All the ones in `patterns/lifep` that I have tried have worked, however).

As mentioned, the rendering is done using OpenGL. This is unlike the Life program that was used in the example that came with the Bootable Cluster CD, which used the Xlib `XDrawRectagle()` function to send the data through the X server. I did not really use any of the code from the BCCD version. In my version, I did something somewhat unusual by treating the grid data as pixel data and passing it directly to OpenGL using `GLDrawPixels()`, indicating that the data is in the `GL_UNSIGNED_BYTE` format and contains only one color component. This allows the pattern of dead and alive cells to be send directly to the GPU for rendering without any extra processing.

In the case of the CUDA implementation, the data does not even leave the GPU. I created an OpenGL buffer object with `GLGenBuffers()` and `GLBufferData()`. This a region of memory allocated on the GPU, which normally would be filled with static pixel or vertex data to be repeatedly rendered without sending the data to the GPU over and over. In this case, however, it is used for dynamic pixel data, as it doubles as the actual representation of the grid. CUDA provides a way to register

OpenGL buffer objects and retrieve pointers to the memory address of them in device memory, which can then be passed to a CUDA kernel. Therefore, other than the picture that comes out of the graphics card, the data is simply kept on the GPU the entire time and never reaches the CPU. [1] This avoids the overhead of sending large quantities of data to the GPU over and over.

My program does not yet provide a way to calculate an arbitrary Life generation and dump the output to a file. If it was to do this, it would be nice to have an option to do so without initializing the GUI. The program does, however, provide a way to do timing, as described in the next section.

5 Performance

The program provides the `--time` command line option to time how long it takes to calculate an arbitrary Life generation without any intervening OpenGL commands.

The actual code that is being timed is nearly the same as the code samples I gave earlier in the paper, but I moved the OpenMP parallelization out of the `compute_next_generation()` function to avoid some thread creation overhead, as suggested earlier. Both the OpenMP and CUDA codes also have the grids be of type `unsigned char*` instead of `unsigned char**`, since this optimizes out memory accesses to an auxiliary array of pointers.

To see the raw power of the two approaches to parallelization, I tried them both out on a fairly large pattern, the Turing Machine, which is located in `patterns/tm.rle`. (Note that this is not the *Universal* Turing Machine, which is also located in that directory, but is too big to run using my program). The grid created to hold the pattern was 1760 by 1696 cells. Counting all the arithmetic operations in the code, over 30 million operations are needed per generation. I ran both programs, `openmp-life` and `bccd-life`, for varying numbers of generations. The results are shown below.

[1] When I was testing the program, I used the LittleFe remotely. Normally it is not possible to run OpenGL programs in such a case, but a program called VirtualGL is able to intercept OpenGL calls, make them render into a special buffer, and make the rendered result available to be sent to the remote computer, possibly via an intermediate VNC server. This process, of course, means that the data leaves the GPU.

Table 2: Running time for the Turing Machine (1760x1696 grid)

	Number of Generations			
	1	10	100	1000
`openmp-life` (1 thread)	35 ms	355 ms	3557 ms	35621 ms
`openmp-life` (2 threads)	18 ms	181 ms	1807 ms	18163 ms
`cuda-life`	10 ms	100 ms	996 ms	9963 ms

The results were quite consistent. The OpenMP version achieved approximately 1.96x speedup by using 2 threads instead of 1, which is nearly full efficiency. Perhaps more interesting though, was that the graphics processing unit outperformed the CPU. We can see that CUDA code completed in about 55% of the time the OpenMP code took with 2 threads.

I next experimented to see how the two approaches did on a much smaller grid, only 50 by 50 cells. The pattern run was the R Pentonimo. However, the specific pattern should be irrelevant because the algorithm performs the same number of operations regardless of what the pattern is. The only significant factor is the grid size.

Table 3: Running time for the 50x50 grid

	Number of Generations			
	1	100	10000	1000000
`openmp-life` (1 thread)	0 ms	3 ms	295 ms	29503 ms
`openmp-life` (2 threads)	0 ms	2 ms	345 ms	36831 ms
`cuda-life`	0 ms	2 ms	280 ms	28286 ms

The CUDA implementation didn't do quite as well as with the bigger grid; it completed in about about the same time as the single-threaded OpenMP implementation. This can be accounted for by the software overhead associated with the kernel launches. Although launching the kernels is designed to be very fast, each launch still has to involve running code inside the CUDA library on the CPU, then sending the command to the GPU, so it must take some time.

It also is apparent that for the small grid, unlike on the large grid, the OpenMP implementation was slower with two threads than with one thread. This can be accounted for by the large number of synchronizations required.

The OpenMP implementation with 2 threads likely would have performed even worse if I had not included the optimization where the threads synchronize between each generation rather than being recreated. Such a lightweight synchronization

is not possible in the CUDA architecture, however. There is no way to synchronize threads running in different thread blocks, and the recommended way to do calculations that would require this is to use separate kernel launches.

It is worth noting that I tried adding some other optimizations to the code in both the OpenMP and CUDA versions. For the CUDA version I tried loading data into the per-thread-block shared memory before accessing it, as most bytes of this memory are accessed by multiple threads as they count up their neighbors. I thought that this possibly could result in better performance, as presumably global memory would be accessed less. However, this actually resulted in much worse performance. I would guess that the hardware is already able to optimize out the redundant global memory accesses, and trying to do it explicitly just resulted in time wasted due to the unnecessary operations and thread synchronization. The use of shared memory was shown to be useful in some of Nvidia's examples such as matrix multiplication, however.[3]

For the OpenMP version I tried reducing the number of additions by keeping a running total going across the row, and subtracting the left neighbors whenever they left the neighborhood. This resulted in much worse performance for some reason, however. If I had time to examine the assembly code it might have explained why.

6 Other Algorithms

While the Life implementation that has been described so for can certainly be very fast with the help of GPU acceleration, there are issues with the approach that has been taken so far. It is primarily a brute-force approach, as it spends the same time considering every single cell of the grid, even though most of the cells are empty, static, or oscillating. Another problem is the necessity to set some fixed size of the grid. Unless we know exactly how big the pattern is going to become, it is possible that the grid size selected is too small, making the result useless, or too big, making the computation take too long.

Many of the Life patterns that recently are being worked on and discovered, such as the Universal Turing Machine, are too big to fit into memory using a fully represented grid. Or, even if one of these huge patterns does fit into memory, then the computations on the enormous grid would be too slow to see what happens to the pattern in a reasonable amount of time.

The main algorithm that is used in this case is called HashLife. It represents the grid using a quadtree, which recursively divides the grid up into regions. This allows memory to be saved by duplicating references to identical regions. Additionally, time can be saved by caching the result of computations on regions, and these results can

be reused whenever necessary. The results can also be cached for a state many generations in advance, so large numbers of generations can be skipped over.

The HashLife algorithm is complicated and I am not yet completely sure how it works. It sounds like it would be harder to parallelize than the "brute-force" algorithm, especially on a GPU, but some degree of parallelization may be possible. Also, it sounds like it does not explicitly find the solution for any generation other than the goal, but rather is constantly advancing different number of generations on different regions of the grid. This would make it difficult to visualize how the pattern changes over time.

7 Conclusion

Overall, it was interesting to see that the performance of the Life simulation was indeed accelerated by nearly 2x by CUDA on the larger grids, when compared to OpenMP. However, I was hoping there would be a larger improvement than this because the GPU uses a highly parallel architecture, which is very appropriate for this data-parallel problem. I think the mediocre performance can be accounted for partly by the fact that the CUDA device on the LittleFe is fairly low end and only includes 2 multiprocessors. This is not very many compared to some of the higher-end cards that provide dozens of multiprocessors. Also, even though there were many more execution units on the GPU available than on the CPU (each multiprocessor has 32 execution units), there is still the overhead for the kernel launch and for accessing the global device memory. Accessing the global GPU memory is very slow compared to accessing CPU memory when the latter has been cached. Nvidia has implemented hardware managed caches on some of their more recent GPUs, however.[4]

The CPUs on the LittleFe weren't very high performance either, as the "Atom" processor is optimized for power consumption. I found that the OpenMP code ran 4 times faster on my laptop, which has a dual-core processor that uses the AMD K10 architecture.

It would be interesting to time the code on some other computing platforms and see how it performs. It also would be interesting to implement HashLife and see if it can be parallelized with OpenMP and/or CUDA.

References

[1] Conway's Game of Life. http://www.conwaylife.com/wiki/Conway%27s_Game_of_Life.

[2] Khronos Group. OpenCL - the open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/.

[3] NVidia Corporation. *NVIDIA CUDA C Programming Guide*, 3.2 edition, October 2010.

[4] NVidia Corporation. *Tuning CUDA Applications for Fermi*, 1.0 edition, February 2010.

[5] Paul Rendell. This is a Turing Machine implemented in Conway's Game of Life. http://rendell-attic.org/gol/tm.htm, 2000.

[6] Paul Rendell. This is a Universal Turing Machine (UTM) implemented in Conway's Game of Life. http://rendell-attic.org/gol/utm/index.htm, 2010.

[7] Andrew J. Wade. Universal constructor based spaceship. http://conwaylife.com/forums/viewtopic.php?f=2&t=399&p=2327#p2327, 2010.

[8] Stephen Wolfram. Statistical mechanics of cellular automata. *Reviews of Modern Physics*, 55(3):601–644, July 1983.