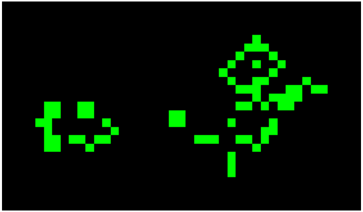
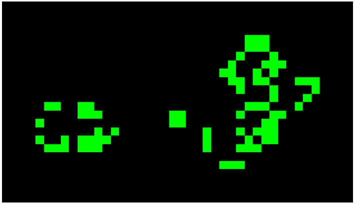
Parallelization of Conway's Game of Life

Eric Biggers

December 16, 2011

What is the Game of Life?





- A cellular automaton
- Cells can be alive or dead.
- Dead cells come to life if bordered by 3 living cells. Living cells continue living only if bordered by 2 or 3 living cells.

Why is the Game of Life Interesting?

- Rules were carefully chosen to allow complex, hard-to-predict patterns to emerge.
- Turing Complete: Life patterns can be used to simulate logical operations, memory, etc.
- Shows how complex structures can emerge from very simple rules
- 2010: Self-replicating pattern discovered (it destroys the original pattern though)
- Fun to implement and watch

Implementing the Game of Life

"Brute Force" Algorithm

- Represent the Life grid, which is infinite, as a 2D array in memory of limited size. 2 copies of it are needed
- Set living cells to 1 and dead cells to 0.
- For each generation, iterate through all the cells and compute their fate.

HashLife Algorithm

- A dynamic programming approach that can hugely reduce the number of calculations needed
- This approach is more complicated and may be difficult to parallelize

This presentation focuses on the brute force algorithm.

Sample Serial Implementation (“Brute Force” Algorithm)

Source Code:

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

Parallelization of the Game of Life

- Highly data parallel problem
- No iteration of the loop depends on any prior iteration

Shared memory

- In class we learned how we can easily parallelize a loop by placing an OpenMP directive. This is done in the next slide.

GPU

- Data parallel problems like this can be effectively solved on a graphics processing unit with the help of a framework such as CUDA.

Sample OpenMP Implementation

Source Code:

```
void compute_next_generation(const unsigned char** grid_in, unsigned char** grid_out, int width, int height) {
  #pragma omp parallel for
  for (int y = 1; y < height - 1; y++) {
    for (int x = 1; x < width - 1; x++) {
      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];
      grid_out[y][x] = ((num_neighbors == 3) || (cell(x, y) && num_neighbors == 2));
    }
  }
}
```

CUDA (Compute Unified Device Architecture)

- A framework developed by Nvidia for doing general-purpose computing on their GPUs.
- Components
  - Device: A supported Nvidia GPU
  - Language: A C-like language for writing kernels (code that runs on the device)
  - Compiler: `nvcc`
  - Libraries: `libCuda`, `libcudart`
  - Driver (Linux kernel module): `nvidia`
- Closed source and does not work on ATI/AMD GPUs. However we do have it available to use on the LittleFe.
- We must rethink how the code is to be written because the CUDA architecture is very different from what we are used to.

Conway's Game of Life in CUDA

- Grid data must be transferred to the GPU using the CUDA API. If needed back on CPU, it must explicitly be sent back.
- The CUDA architecture uses a large number of extremely lightweight threads. We should assign each element of the grid to 1 thread.
- The `compute_next_generation()` function will launch a CUDA kernel with appropriate numbers of thread blocks and threads per thread block.

Sample CUDA Implementation (kernel only)

Source Code:

```
__global__ void kernel(const unsigned char** grid_in, unsigned char** grid_out, int width, int height) {
    int x = (blockIdx.x * blockDim.x) + threadIdx.x;
    int y = (blockIdx.y * blockDim.y) + threadIdx.y;
    if (x > 0 && x < width - 1 && y > 0 && y < height - 1) {
        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];

        grid_out[y][x] = ((num_neighbors == 3) || (cell(x, y) && num_neighbors == 2));
    }
}
```

Sample CUDA Implementation (kernel launch)

Source Code:

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

My Implementation

- 2 executables: `openmp-life` and `cuda-life`
- Renders grid using OpenGL (`glDrawPixels()`)
- In `cuda-life`, OpenGL pixel buffer object is shared with CUDA kernel
- Some support for opening files containing Life patterns.
- OpenMP implementation optimized by moving `pragma` up one function so that when multiple generations are advanced, only lightweight synchronization is needed among the threads.
- The above is not possible for CUDA, which requires new kernel launch each generation.

Performance Comparison: Big Grid

LittleFe hardware: Intel Atom Processor (2 cores), Nvidia GPU (2 multiprocessors)

Table: 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

- OpenMP implementation achieves 1.96x speedup with 2 threads compared to 1.
- CUDA implementation completes in 55% of the time of 2-thread OpenMP implementation!

Performance Comparison: Small Grid

Table: Running time for a 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

2 threads was slower than 1! Too many synchronizations were required.
CUDA was still faster, but not by much. Launching a CUDA kernel requires some overhead.

Conclusion

Main Points

- The LittleFe platform provided a way to test how Conway's Game of Life can be accelerated using OpenMP or CUDA.
- On large grids, both methods accelerate the computation compared to the sequential version, but CUDA did better.
- On very small grids, neither is much better than the serial, CPU version. Most interesting patterns need large grids, though.

Further Research

- Can the HashLife algorithm be parallelized? With OpenMP? With CUDA? With MPI?

References

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

- NVidia Corporation.
  NVIDIA CUDA C Programming Guide, 3.2 edition, October 2010.

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

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