Life Algorithms

Tomas Rokicki

June 28, 2018

Abstract

We describe some ideas on how to implement Conway's Game of Life.

1 Introduction

Conway's Game of Life serves as the "Hello World" of recreational computing. Many of us who cut our teeth on minicomputers or microcomputers will remember writing our own implementations, and then struggling to make them faster [5]. In this paper we briefly describe some of the more interesting ways to implement Life, without taking the fun out of getting all the details of implementation exactly correct.

We will focus on calculating subsequent generations from current generations, ignoring the impact on other operations such as loading and saving patterns, setting and clearing cells, calculating the current population of the universe, and displaying the pattern. We will also ignore the impact of the borders (if any) of the universe. In a real Life program, all of these are important aspects that should be considered before adopting a too-complex algorithm.

All the code for this paper is available at https://github.com/rokicki/lifealg along with more information and performance comparisons. We include the algorithms in the Golly program as well as high-performance algorithms created by Adam Goucher as part of his lifelib project.

2 Elementary Approaches: Arrays

2.1 Basic Algorithm

The most fundamental implementation of the Game of Life uses two two-dimensional arrays representing the current universe and the universe in the next generation. Computing the next generation array from the current generation array involves iterating over all the cell positions, calculating the number of live neighbors to that cell from the current generation array, and then determining if the next generation cell is alive or dead based on this neighbor count and the life status of the current generation cell. After this calculation, we either swap the current and next generation arrays, or copy the next generation array into the current generation array.

```
void nextgen(unsigned char u0[][W], unsigned char u1[][W]) {
   for (int i=1; i+1<H; i++)
      for (int j=1; j+1<W; j++) {
         int n = u0[i-1][j-1] + u0[i-1][j] + u0[i-1][j+1] + u0[i][j-1] + u0[i][j+1] +
                 u0[i+1][j-1] + u0[i+1][j] + u0[i+1][j+1] ;
         u1[i][j] = (n == 3 || (n == 2 && u0[i][j])) ;
      }
}
```

In some sense, this is the best we can do; for a completely random pattern that fills our array, we can only make constant-time improvements over this basic algorithm when calculating a single next-generation step. But these constant-time improvements can be significant. Furthermore, we can exploit the fact that subsequent generations are not random, but show certain behaviors; for instance, when starting from a totally random pattern, the average population density decreases quickly, and regions of the space achieve some degree of stability over limited time intervals.

Also, once we have played with many random universes, our attention may shift to specific constructions that exhibit specific behavior, such as puffer trains or patterns that allow you to play Tetris. These patterns are frequently sparse and exhibit a high degree of regularity that can be exploited for speed.

2.2 Simple Improvements

The previously given simple algorithm runs at about 270 million cell generations per second on a laptop, so it is easily fast enough to animate a large 2,000 by 2,000 universe at more than 60 generations per second. If all you want is a pretty display, there's likely to be no reason to do anything more complicated. But let's assume you are interested in much larger patterns, or in running them far more quickly.

Some of the changes we will make will have much larger impacts than we might expect. For instance, replacing the line

```
u1[i][j] = (n == 3 || (n == 2 && u0[i][j])) ;
```

with a precalculated table given the value for the next generation and neighbor count, as in

```
u1[i][j] = nextval[u0[i][j]][n]
```

can nearly double the speed to 470M cell generations per second, even though the total number of instructions executed per generation in the two versions is almost identical. Modern CPUs predict conditional expressions in order to evaluate multiple instructions per cycle, and they pay a significant performance penalty when such branches are not easily predictable. Replacing conditionals with lookups into small tables and straight line code can help the processor run faster.

2.3 Integration

Instead of checking all eight neighbors for each cell, we can instead accumulate cell accounts for each row and then across rows, with code like the following:

```
void nextgen(unsigned char u0[][W], unsigned char u1[][W+1]) {
   for (int i=0; i<=H; i++)
      u1[i][0] = 0 ;
   for (int j=0; j<=W; j++)
      u1[0][j] = 0 ;
   for (int i=0; i<H; i++)
      for (int j=0; j<W; j++)
         u1[i+1][j+1] = u0[i][j] + u1[i][j+1] + u1[i+1][j] - u1[i][j] ;
   for (int i=1; i+1<H; i++)
      for (int j=1; j+1<W; j++) {
         unsigned char n = u1[i+2][j+2] - u1[i+2][j-1] - u1[i-1][j+2] + u1[i-1][j-1] ;
         u0[i][j] = (n == 3 || (n == 4 && u0[i][j]))
      }
}
```

This algorithm uses `u1` to integrate the original universe in two dimensions (using an unsigned data type to avoid undefined behavior), and then calculates the new result back into the `u0` array. This is just about as fast as the original algorithm for the normal Conway game of life, but it is significantly faster for a variant called "Larger than Life", where we use a larger neighborhood. With this algorithm we can compute "Larger than Life" for a large neighborhood (such as 201x201) about as fast as we can calculate the normal Game of Life.

2.4 Single Instruction Multiple Data

In the simple algorithm presented, we use two arrays, each with a full byte (eight bits) per cell; this wastes memory, since each cell has only two potential values. But it also wastes computation. Our neighbor count is a full integer value, with 32 bits, but the largest value it will ever need to store is 8, so can get away with four bits. Modern CPUs operate on 64-bit wide words (or even wider; 128-bit and 256-bit one-cycle operations are now common, and 512-bit one-cycle operations are starting to appear). We can use this to speed up our calculations by using a wider type for our arrays and packing more than one cell state in this wider type.

Let us say we use 64-bit unsigned words, and we allocate four bits for each cell; this lets us put 16 cell states in a single 64-bit word. We will assign 16 horizontally-adjacent cells to each 64-bit word. We are now wasting 3 bits for each cell—but we do this so we can accumulate neighbor counts for all 16 cells in one sequence of instructions by using carryless addition.

Consider adding the decimal values 314 and 271; as long as the sum of digits in each position is never greater than 9, we will never generate a carry. For hexadecimal, we can parallelize small additions up to a maximum value of 15 as long as we ensure no small addition ever exceeds 15. Since our maximum neighbor count is only 8, this is guaranteed.

Once we have the 16 neighbor sums packed into a single 64-bit word, we need to calculate the 16 resulting cells. Using a lookup table to do this would require 16 separate shifts, masks, lookups, and adds, but some clever bit manipulation and masks can do the work for us. In addition, it is easier to calculate the sum of all nine cells and then modify our "stay alive" counts slightly. Getting the shifts and masks exactly correct is a bit tedious but not extremely so; the resulting code looks like this:

```
void nextgen(unsigned long long u0[][W], unsigned long long u1[][W]) {
   for (int i=1; i+1<N; i++)
      for (int j=0; j<W; j++) {
         unsigned long long pw = u0[i-1][j] ;
         unsigned long long cw = u0[i][j] ;
         unsigned long long nw = u0[i+1][j] ;
         unsigned long long n = (pw << 4) + pw + (pw >> 4) + (cw << 4) +
            (cw >> 4) + (nw << 4) + nw + (nw >> 4) ;
         if (j > 0)
            n += (u0[i-1][j-1] + u0[i][j-1] + u0[i+1][j-1]) >> 60 ;
         if (j+1 < W)
            n += (u0[i-1][j+1] + u0[i][j+1] + u0[i+1][j+1]) << 60 ;
         unsigned long long ng = n | cw ;
         u1[i][j] = ng & (ng >> 1) & (~((ng >> 2) | (ng >> 3)))
                                               & 0x1111111111111111LL ;
      }
}
```

This code runs at about 6.1 billion cell generations per second, about thirteen times faster than the previous code. An excellent book on such bit tricks is [7].

But that is not nearly as fast as we can make it.

2.5 Bitwise Parallel Addition

In order to do carryless addition, we packed 16 cell values into a 64-bit integer, because we needed four bits in the neighbor sum to represent values up to 9. Instead of storing those four bits in a single register, we can store the bits in separate registers. We can use a single register to store the low-order bit for 64 values, a second register to store the next higher-order bit for 64, values, and so on. This requiring four registers to represent 64 4-bit neighbor counts. With this representation, we can store 64 Life cells in a single 64-bit integer.

Internally the CPU performs arithmetic addition using logical operations. To calculate the low-order bit of the sum of two bits (known as a half adder), we calculate the exclusive-or of the two bits;

to calculate the high-order bit, we calculate the ‘and’ function. To do this for three bits, we cascade two half-adders and add an additional ‘or’ function to merge together the carries from both sums. Processors contain bitwise logic operations that allow us to perform these sorts of calculations on the entire width of a register in a single instruction. So to calculate the low and high order bits of the bitwise sums of each bit in a 64-bit register, we can use the following code:

```
void halfadder(unsigned long long a, unsigned long long b,
               unsigned long long &s0, unsigned long long &s1) {
   s0 = a ^ b ;
   s1 = a & b ;
}
void fulladder(unsigned long long a, unsigned long long b, unsigned long long c,
               unsigned long long &s0, unsigned long long &s1) {
   unsigned long long t0, t1, t2 ;
   halfadder(a, b, t0, t1) ;
   halfadder(t0, c, s0, t2) ;
   s1 = t1 | t2 ;
}
```

To sum more than three bits, we cascade full adders as needed.

There are a few small optimizations we can perform. We don’t need the highest-order bit, since total cell counts of 8 and 9 have the same effect as total cell counts of 0 and 1. Overall, using 64-bit registers and standard C++ code, we can achieve 13.6 billion cell generations per second using this approach.

In some environments, it may be useful to work on full bitplanes instead of registers. On modern CPUs this won’t give maximum performance because it requires too much memory bandwidth. But on the HP48 calculator, for instance, where bitwise logical operations on bitplanes are supported, the following code executes Life quickly even in interpreted RPL:

```
    GEN1 << {#0 #1} DUP2 SH OVER LX ROT REVLIST
      SWAP OVER SH 5 ROLLD 4 ROLLD SH 4 PICK LX
      ROT 3 PICK + NEG + 4 ROLLD NEG + LX + NEG >>
    SH << DUP2 OVER DUP ROT {#FFFh #FFFh} SUB
      LX 3 DUPN 7 ROLLD GXOR 5 ROLLD GXOR + >>
    LX << {#0 #0} SWAP GXOR >>
```

2.6 Minimum Logical Operation Count

There have been some published papers that incorporate parallel bit-wise addition for calculating Life, often focusing on performance. The ones I have seen use suboptimal sets of logical operations. To provide a basis for comparison, we will imagine that we have an 8x8 section of the universe in row-major order in a single 64-bit word, and we wish to compute the inner 6x6 section in as few CPU instructions as possible. The principles we use apply to many conventional universe layouts but focusing on a square in a single word and ignoring the edges allows us to simplify our presentation without losing generality.

Just cascading half-adders to generate the low-order three bits of the sum of neighbors would look something like this:

```
lifeword gen4(lifeword a) {
   lifeword a0, a1, b0, b1, c0, c1, d1, d2, e1 ;
   add3(a>>9, a>>8, a>>7, a0, a1) ;
   add3(a<<9, a<<8, a<<7, b0, b1) ;
   add3(a<<1, a>>1, a0, c0, c1) ;
   add3(a1, b1, c1, d1, d2) ;
   e1 = c0 & b0 ;
   c0 ^= b0 ;
   d2 ^= e1 & d1 ;
   d1 ^= e1 ;
   return d1 & (~d2) & (c0 | a) ;
}
```

This takes 29 Boolean operations and 8 shifts. But since the operations are bitwise, the first two full adds are computing the same result, just shifted. We can do a horizontal sum of three bits and then use that result three times with the following code (which sums all nine cells, not just the eight neighbors). When applied to dense bitmaps, this is equivalent to retaining the horizontal sum for each row across the three rows that require it.

```
lifeword gen4(lifeword a) {
   lifeword a0, a1, b0, b1, c1, c2 ;
   add3(a>>1, a, a<<1, a0, a1) ;
   add3(a0>>8, a0, a0<<8, b0, b1) ;
   add3(a1>>8, a1, a1<<8, c1, c2) ;
   c2 ^= b1 & c1 ;
   b1 ^= c1 ;
   return (b0 ^ c2) & (b1 ^ c2) & (a | b0) ;
}
```

This takes only 23 Boolean operations and six shifts. But we can improve this further. The first full-add is composed of two half-adds, and instead of throwing away the sum of the first two elements, we can preserve those and use this to compute a proper sum of neighbors rather than a sum of all elements. Further, rather than compute three binary bits of the sum, we can use a simpler expression that determines if precisely one of four bits is set. The final code looks like this (after expanding all full adds):

```
lifeword gen3(lifeword a) {
   lifeword aw = a << 1, ae = a >> 1,
       s0 = aw ^ ae, s1 = aw & ae,
       hs0 = s0 ^ a,
       hs1 = (s0 & a) | s1,
       hs0w8 = hs0 >> 8, hs0e8 = hs0 << 8,
       hs1w8 = hs1 >> 8, hs1e8 = hs1 << 8,
       ts0 = hs0w8 ^ hs0e8,
       ts1 = (hs0w8 & hs0e8) | (ts0 & s0);
   return (hs1w8 ^ hs1e8 ^ ts1 ^ s1) &
          ((hs1w8 | hs1e8) ^ (ts1 | s1)) & ((ts0 ^ s0) | a);
}
```

This code requires only 19 Boolean operations and 6 shifts. I have not proved this to be optimal but I know of no better solution.

2.7 SSE and AVX2

Processor designers have been incorporating a simple vector processor in the form of wider registers and explicit SIMD instructions for many years. Using these facilities allows us to extend our techniques above from the default register width of 64 bits to 128 bits (using SSE), 256 bits (using AVX2), and even 512 bits (using AVX-512); in each case we potentially gain another doubling of performance. For SSE on my laptop, I was able to achieve 23.9 billion cell generations per second, nearly 100 times faster than our original simple algorithm. While in some cases carefully written C++ code can be automatically translated into these vector operations by the advanced compilers of today, typically to get maximum performance the programmer must resort to the very careful use of low-level "intrinsic" functions. The principles are the same as described previously, but instead of using basic C++ operations on 64-bit words, special intrinsic functions are called on special SSE datatypes. Describing how to write this code is beyond the scope of this paper, but simple examples are given in the code repository referenced above.

2.8 Multithreading

More performance is still available to us through the use of multithreading. Modern CPUs have multiple cores, with two and four being common on laptops, four and eight common on desktops, and many more frequently available on even small server computers. It is simple to assign each core a region of the life universe to work on. On my two-core laptop using four threads allowed me to nearly double the performance to 45.7 billion cell generations per second. On a 4-core desktop CPU, performance is 130 billion cell generations per second; on an 8-core server CPU, performance is 230 billion cell generations per second.

At this calculation speed we are nearing the memory bandwidth of modern CPUs, and additional tricks to exploit caching and memory locality may need to be used. For instance, instead of advancing the whole field one generation, we can advance smaller subregions that fit in the cache multiple generations at once. As AVX instructions get wider, this may be of increasing importance.

2.9 Graphics Processing Units

The performance of current CPUs is dwarfed by the performance of modern graphics processing units (GPUs), which frequently contain thousands of smaller, slower processing units that all work simultaneously. On these devices, the bit tricks that worked so well on CPUs become even more effective, but transferring the field data to and from the different cores of the GPU can be a challenge. In addition, exploiting GPUs can be extremely tedious and challenging, requiring knowledge of APIs such as CUDA or OpenGL as well as architectural knowledge of the specific graphics card to be used. Nonetheless, performance gains of between one and two orders of magnitude can be obtained using common gaming cards [1].

3 Work Smarter, Not Harder

So far we have focused on speeding up the low-level calculation of the next generation, without any consideration for exploiting empty space or regularity in the pattern and its evolution. While these low-level tricks are useful, most performance improvements in simulating Life on interesting patterns are obtained by using smarter algorithms rather than making the bit banging faster.

Our algorithms so far have been for small universes limited by the size of a 2D array we can fit in memory. Our algorithms from here on out will support unbounded universes, limited only by the data type used for coordinate arithmetic. These algorithms are all designed around a data type or types used as a container of a different data type used as a node. For containers, we may use lists, hash tables, or trees; the nodes can be individual life cells, or they can be smaller subfields within which algorithms from the previous section are used.

3.1 Associative Containers

One of the simplest implementations of Life is to just use a container of the coordinates of live cells; we can use a dictionary in Python, or a hash in Lisp, but we will describe using a set in C++. The code is just:

```
void nextgen(const univ &src, univ &dst) {
   map<pair<int, int>, int> ncnt ;
   for (auto i=src.begin(); i!=src.end(); i++) {
      ncnt[make_pair(i->first-1, i->second-1)]++ ;
      ncnt[make_pair(i->first-1, i->second)]++ ;
      ncnt[make_pair(i->first-1, i->second+1)]++ ;
      ncnt[make_pair(i->first, i->second-1)]++ ;
      ncnt[make_pair(i->first, i->second+1)]++ ;
      ncnt[make_pair(i->first+1, i->second-1)]++ ;
      ncnt[make_pair(i->first+1, i->second)]++ ;
      ncnt[make_pair(i->first+1, i->second+1)]++ ;
   }
   dst.clear() ;
   for (auto i=ncnt.begin(); i!=ncnt.end(); i++)
      if (i->second==3 || (i->second == 2 && src.find(i->first) != src.end()))
         dst.insert(i->first) ;
}
```

In this code we use a `map` to calculate a neighbor count for each cell, and then use that to calculate the new generation counts. This code only calculates at about 500,000 cell generations per second, but—unlike all previous code—it executes in time roughly linear in the number of live cells, rather than the total size of the universe. (There's a log factor due to the use of `map` and `set` that I am ignoring.) In addition, there are no bounds on the size of the universe except for the maximum coordinate that fits in an int. Despite all the advanced bit-level machinery we put in place previously to push the cell generations per second as high as possible, for many uses, the simple algorithm just given is more useful. Realistic interesting patterns tend to be sparse and non-rectangular; doing fast bit computations on, or even just examining, empty space is extremely wasteful. For example, using our fastest AVX2 algorithm on a 4K x 4K universe on the `lidka` Methuselah takes 0.223 seconds to get to generation 512; using the sample algorithm above, with all the map overhead, accomplishes this in 0.0014 seconds.

Actual performance of these algorithms is harder to pin down than for the brute force algorithms given previously because they are highly pattern-dependent. Some optimizations work well for some patterns but fail for others.

3.2 Fat Nodes

The algorithm just given was slow in terms of cell generations per second, but still performs well for many uses because it focuses only on the actual live cells of the universe. Many interesting patterns do not fit perfectly into a nice bounding rectangle or are extremely sparse. For instance, the recently found Gemini oblique spaceship fits in a 217,807 by 4,220,191 bounding box but has a population of only 846,278; fewer than one cell in a million is alive. Even at 230 billion cell generations per second using eight cores in a server with the required 230GB of memory with a flat bitplane representation, it would take at least 4 seconds to calculate each generation; the simple algorithm shown above on a single core easily achieves nine generations a second using only 60MB of memory.

In practice, we want to strike a balance between time spent managing the shape of the active region, managing knowledge of what is changing and what is stable, and computing the low-level next generations for some cells. The easiest way we do this is to use fatter nodes—nodes that contain more than a single cell of the universe. We store these nodes in some sort of container structure, and calculate the next generation by scanning this container, finding the relevant fat nodes and their neighboring nodes, and computing the next generation of each node. If the result has any cells set, we create a new node in the next generation containing those cells.

Within a fat node, to compute the next generation quickly, we can use all the fancy bit manipulation we described in the previous section of the paper. This can give us a speedup of more than 100. We will want to carefully set the size of the fat nodes, trading off efficiency within a fat node against the wasted computation on areas within the fat nodes that do not contain any active cells.

In some cases it may be simpler to have the fat nodes overlap, so some cells may be represented by bits in more than one fat node. This eliminates the necessity to consider the neighbors when calculating the next generation, but requires us to update more than one fat node with the results.

3.3 Shifting Coordinate Systems

A complication for fat nodes is that they have nine neighbors; that is a lot of neighbors to manage. A common trick that is used to improve performance is to use a shifting coordinate system. For instance, if I'm keeping two generations, I may have a node maintain the rectangle $(0, 0) - (15, 15)$ for the even generation but $(1, 1) - (16, 16)$ for the odd generation. If I do this then I only need to examine three neighbors for each calculation; I can calculate the state of $(1, 1) - (16, 16)$ from the prior states of

$(0,0)-(15,15)$ from the current cell, $(16,0)-(17,15)$ from the node to the right, $(0,16)-(15,17)$ from the node down, and $(16,17)-(16,17)$ from the node down and to the right. This approach can simplify calculation code significantly.

In some cases it may be simpler to shift the coordinate system shift continuously, so a cell keeps information about a moving rectangle of space over time; for instance, a given cell might track the region $(g,g)-(g+15,g+15)$ in generation g.

Another way to reduce the count of neighbors to examine is to arrange the fat nodes in a bricklaying pattern rather than a simple lattice; this reduces the neighbor count from eight other fat nodes to only six.

3.4 List Containers

The simplest container to consider is a simple list (possibly implemented with an array). Each list element could contain the two coordinate values of the fat node and the cells of the fat node itself. Other state information can be added if appropriate. If the list is organized in raster order, it can be scanned by multiple simultaneous pointers that track corresponding neighbor values in the rows above and below, eliminating the need for a hash table or other mechanism to look up the relevant fat nodes.

In our shootout below we compare list containers containing single cells, 8x8 fat nodes, and 16x16 fat nodes.

3.5 Tree Containers

Trees work well as containers because there's no need to store coordinate, neighbor, or parent information; all of that can be passed as parameters down the call stack. Furthermore, we can maintain a hierarchy of population and status information. With this organization we don't even need to examine large regions of stable nodes to see if we can skip them; we can skip an entire region of any size once we reach the appropriate tree level.

By passing neighbor information down the call stack, we do not need an associative container to randomly locate neighbors; we are always directly passing the appropriate neighbors from the appropriate tree calls. Alternatively, we do not need to maintain neighbor pointers.

Finally, trees can be very efficient in memory consumption because only the leaves have actual data, and we can size the leaves so the bulk of memory is consumed by the leaves, while still supporting appropriate status information higher up in the tree.

In our shootout we find that simple list containers tend to outperform tree containers, but if we add support for skipping stable regions, tree containers often regain the lead.

3.6 Stable Regions

In the game of life, large random regions eventually settle down, usually to a constellation of stable lifeforms and lifeforms of period 2, with the occasional larger-period oscillator and some gliders escaping the region. By storing two consecutive generations in each fat node, along with a flag indicating whether that region is stable (i.e., either not changing, or only changing with a period of 2), we can quickly skip over sections of space that are not changing.

Some care must be given to neighboring regions. If a neighboring region is changing, we may need to recompute an otherwise stable region. We can optimize this by keeping flags on stability for the border of the node as well as the node as a whole. Only if the adjacent border is changing do we need to recompute a neighboring node.

When using list containers, skipping large stable or period-2 regions can become a bottleneck. Using a tree container instead can resolve this. Another alternative is to separate active and idle fat nodes into their own lists.

3.7 Hashlife

A simple breeder defeats all the algorithms we've described so far. For this pattern, space increases at $O(n^2)$ and computation time increases at $O(n^3)$, so computing the pattern at generation $2n$ takes eight times as much time as computing the pattern at generation n. The majority of the pattern becomes waves of gliders, which our algorithms so far do not optimize.

William Gosper introduced the amazing hashlife algorithm in 1983 [3], though it was not widely used until Golly was released in 2005. Hashlife represents the universe using a quadtree representation where identical subtrees are coalesced or canonicalized; this provides dramatic compression of space for many patterns. The algorithm caches the result of computing Life for any particular node in the quadtree to reuse this computation across generations. For more information see [6].

The real magic of hashlife, however, is that it does not just calculate the next generation for each tree node; instead, it jumps further ahead in the future than that. For a quadtree node at level n, representing a 2^n x 2^n region of space, we calculate the result at 2^{n-2} generations forward. This simplifies our recursive calls and allows hashlife to compute patterns many billions of generations in the future.

For most Life patterns that people run, hashlife is astonishingly faster than anything we have presented so far. It is not uncommon for a pattern to run somewhat normally for a second or two but suddenly explode in speed, generating trillions of generations in seconds, and often unexpectedly. But for patterns that stay essentially random for long periods of time, or have significant amounts of entropy, hashlife can be much slower, although it is highly unusual for hashlife to be significantly slower than any non-hashlife algorithm over several minutes of runtime.

Hashlife is a fascinating algorithm. During its execution the only thing it is doing is hashtable lookups and hashtable insertions (and the occasional garbage collection); almost no time is spent doing a primitive cell computation (at least not in the standard game of life). The performance of hashlife is entirely dependent on the efficiency of the hashtable implementation used.

Hashlife is reputed to take a tremendous amount of memory, and this can certainly happen. But a "tremendous" amount of memory is a changing quantity. That 32MB that Bill Gosper had to work with used to be a tremendous amount of memory, but it is the cache size of modern CPUs, which are typically paired with over 2,000 times that much RAM memory.

The reason hashlife uses so much memory is because there's no free lunch; if you ask it to generate 1,000 generations of a 1,000 by 1,000 highly random pattern, and the generation rule is such that the results are highly unpredictable, you will be consuming enough space to store a substantial fraction of all of those generations. Hashlife is amazing, but it is not magic. If you use a smaller step size, the memory consumption of hashlife is significantly mitigated (but so is its potential for galloping ahead.)

The fact that hashlife usually spends little time doing the primitive cell computation, and the fact that it essentially caches this computation across many uses, makes it possible to simulate significantly more complex rules effectively. This is the basis of Golly's extension to 256-state automata and the reason such perform so well in Golly. Again, highly random patterns can defeat this by causing generation of more leaves than fit in memory, but for most interesting patterns this does not happen. Golly uses smaller leaves for 256-state automata than it does for 2-state automata to help prevent this from occurring.

Normal life algorithms are generally highly parallel, so they are easily sped up as CPUs gain more cores and as highly parallel GPUs take over more and more the computation. So far, hashlife has been resistant to parallelization. A big challenge for the community is to write an effective, efficient, parallel hashlife, perhaps based on lock-free hashtables.

4 Comparisons

In this section we present a shootout between different algorithms for computing the game of life. This shootout is far from the last word, as we've selected only a handful of interesting patterns, and the algorithms themselves have different capabilities. We present this only to provide insight in algorithm performance.

4.1 Algorithm Implementations

The algorithms we compare are given here, in roughly increasing order of sophistication and performance. The first seven only support bounded universes; the last eight support unbounded universes (or at least universes bounded only by the datatype used to represent coordinates). All implementations except the last four were written specifically for this paper with straightforward code; they might all benefit from various tuning or compiler optimizations, so the results should only be considered in the large. The last four algorithms are well-tuned implementations.

Bounded universe algorithms:

- `array4`: The naive schoolboy algorithm, implemented with static arrays and one byte per cell.
- `lookup`: The schoolboy algorithm but uses a single-cell lookup table and thus supports other outer totalistic rules.
- `lookup4`: Stores data as dense bitplanes, and uses a 2^{16}-element lookup table to compute the 2x2 result of a 4x4 square at once. This provides speed, while still permitting arbitrary rules to be used. This is most like what qlife and hlife use as their inner loop.
- `nybble`: Stores data as 16 cells per 64-bit long, and uses bit tricks to compute the next generation so is specific to GOL.
- `bitpar3`: Stores data as dense bitplanes, and does parallel bitplane addition. Specific to GOL.
- `sse3`: Similar to bitpar3 but uses SSE instructions to compute 128 bits at a time.
- `avx23`: Similar to bitpar3 but uses AVX2 instructions to compute 256 bits at a time.

Unbounded universe algorithms:

- `list3`: Simple list-based algorithm that lists the coordinates of cells that are alive. Supports unbounded universes. The actual computation is specific to the game of life but can be easily modified to be more general without much performance impact.
- `tree`: Uses 8x8 fat nodes with a tree container. Each fat node is computed with an algorithm like bitpar3.
- `list8x8`: Uses 8x8 fat nodes with a list container similar to list3. Fat node computation is the same as bitpar3.
- `list16x16`: Uses 16x16 fat nodes with a list container similar to list3. Fat node computation is similar to avx23.
- `qlife`: The non-hashing algorithm in Golly. Uses a tree structure (but not a quadtree. Each node splits space eight ways in one dimension). Performs period-two optimization. Computation is generic across algorithms without recompilation [2].
- `ulifelib`: Tree-based algorithm that uses AVX2 or SSE (depending on platform). Supports additional rules but only with recompilation. Performs period-two optimization [4].
- `hlife`: The hashing algorithm in Golly. Uses 8x8 leaves and supports general rules. We run it with 1GB RAM [2].
- `lifelib`: Hashlife algorithm that uses AVX2 or SSE (depending on platform). Supports additional rules but only with recompilation. We run it with 1GB RAM [4].

4.2 Patterns

- `r4kx4k`: A random 4000 by 4000 universe at 30% density. Stabilizes (with respect to a period of 2) after 12,000 generations (except for escaped gliders).
- `bp4kx4k`: A 4000 x 4000 universe filled with period-3 pulsars. Intended to evaluate the baseline performance of different algorithms that may have period-2 recognition. The hashlifes will run away quickly with this pattern.

	r4kx4k	r4kx4k	r4kx4k	bp4kx4k
	2K	16K	4M	2K
array4	99.444	-	-	93.963
lookup	129.182	-	-	131.504
lookup4	10.683	85.827	-	10.706
nybble	7.513	60.151	-	7.443
bitpar3	2.377	19.073	-	2.379
sse3	1.195	9.456	-	1.238
avx23	0.782	6.372	-	0.827
list3	107.285	-	-	199.282
tree	10.945	99.114	-	10.488
list8x8	9.596	78.011	-	6.899
list16x16	2.196	18.486	-	2.070
qlife	3.804	5.696	140.768	9.205
ulifelib	2.491	3.429	82.889	6.842
hlife	53.533	68.816	68.943	0.083
lifelib	9.496	14.991	14.992	0.033

Table 1: Bounded vs. unbounded algorithms at 4K x 4K size. We include results at 4M for the random case to compare how the different sophisticated algorithms manage patterns that are primarily stable.

- `QGC1`: A difficult pattern that contains mostly closely-spaced glider streams that are going different directions; this gives hashlife a tough time.
- `breeder`: Gosper's original breeder. Fills one eighth of the universe with a wave of gliders. Population is quadratic in generations.
- `caterpillar`: A huge spaceship. The bounding box is about 4,195 x 330,721 (and thus requires 173MB as a single bitmap). There is a lot of regularity but the pattern is so large that the hashlifes don't easily run away.
- `jagged`: A pattern that grows linearly with generations and generates interesting curves as it runs.
- `lidka`: A small Methuselah. It expands rapidly kicking out many gliders and eventually settles down about generation 30,000.
- `mcc`: Metacatacryst, one of the early small quadratic-growth patterns.
- `spiral`: Similar to but smaller than QGC1.
- `unlim`: A pattern intended to produce unlimited novelty by positioning two rakes at right angles that emit gliders into the debris of each other.

4.3 Results

The first set of results we present is for large rectangular patterns that mostly stay rectangular, so we can compare the bounded and unbounded algorithms. For the bounded algorithms we allocate a universe just large enough to hold the pattern, and we ignore edge effects (mostly gliders that hit the edge and become blocks). The results are in Table 1.

With the exception of the four sophisticated algorithms, performance is very similar between the random initial state and the field of pulsars; this is because the simple algorithms are data-independent. They do not recognize pattern behavior. The first two algorithms that only compute a single cell at a time are very slow. The next five bounded algorithms are increasingly fast as they do more and more bits at once. The lookup4 algorithm, which does four output results at a time from a 4x4 input field,

	QGC1	breeder	caterpillar	jagged	lidka	mcc	spiral	unlim
	128K	32K	1K	128K	1M	128K	512K	128K
list3	-	-	-	-	99.789	-	-	-
tree	231.231	263.220	147.303	166.304	38.217	130.305	238.966	557.311
list8x8	107.790	119.485	74.010	37.655	18.072	58.601	117.828	223.520
list16x16	65.802	43.734	25.808	29.624	6.245	26.492	67.860	87.257
qlife	112.341	170.113	95.010	76.559	3.474	10.779	104.610	22.847
ulifelib	62.401	141.140	41.081	79.610	1.860	5.822	60.214	13.066
hlife	57.239	0.018	25.654	0.100	0.320	0.215	68.581	2.974
lifelib	105.515	0.004	26.153	0.133	0.076	0.039	60.685	1.147

Table 2: Unbounded algorithm comparison.

but is easily generalized to other rules because of that lookup table, is almost as fast as the nybble algorithm but loses increasingly badly as we use 64-bit, 128-bit, and 256-bit logical operations.

For the unbounded algorithms, a fraction of the CPU time must be spent managing the information about what parts of the universe are alive and active (as well as handling the gliders that are shot off from the central section). For these dense patterns, tree and list8x8 are very similar, but list16x16 is much faster because of the AVX2 instructions it uses. The qlife algorithm, despite only doing four bits at a time in the leaves, eventually outperforms these simpler algorithms for the random case because it recognizes period 2 stability. Ulifelib beats qlife with its highly efficient leaf-level computations. (It is possible to run ulifelib in a way that it recognizes period-3 periodicity as well.)

Both hashlife algorithms are significantly slower than the sophisticated non-hashlife algorithms for the random universe while much random activity is still taking place. Lifelib soundly trounces hlife due to the much more efficient leaf-level computations. Note how the hashlife algorithms both take almost no time to advance from 16K to 4M generations, while the other algorithms either don't get there in our time limit or take more than a minute to do so. For the pulsar field, both hashlife algorithms easily and quickly run away.

Our second set of results are for more complex patterns. We chose patterns that are not trivial since most trivial patterns run too fast for meaningful comparison.

The simplest algorithm, list3, that uses individual cells rather than fat nodes, was not able to compete in any meaningful way with algorithms that used fat nodes. We include the lidka result just because the pattern had few enough live cells even at high generation counts where it was able to not lag too far behind.

For almost all patterns, the overall performance increased from the top of the table to the bottom as the algorithms increased in sophistication. For two of the patterns, QGC1 and spiral, the fast ulifelib algorithm was very close to the fastest hashlife algorithms (in this case hlife). The two sophisticated unhashed algorithms, ulifelib and qlife, were fairly comparable, although ulifelib generally won with its AVX2-powered routines specific to the game of life. There were a few patterns (breeder, caterpillar, jagged) where the simple list16x16 algorithm outperformed qlife and ulifelib. Where this happened, however, hashlife algorithms were usually even better.

For the hashing algorithms, lifelib beat hlife on breeder, lidka, mcc, and unlim; hlife won on QGC1, and they were close to tied on caterpillar, jagged, and spiral. Lifelib also makes more efficient use of memory, using 32-bit node indexes rather than 64-bit node pointers. On the other hand, hlife uses prefetching to overlap some memory references for hashtable lookups.

5 Discussion

It is clear from our results that there is no best Life algorithm; as you increase the sophistication of algorithms to take advantage of specific behaviors, you generally decrease the performance on the class of patterns that do not exhibit those behaviors. The attraction of hashlife is that it provides significant acceleration for a wide class of behaviors, even though for large random patterns it is typically outperformed by simpler algorithms.

Equally clear is that the use of low-level bit tricks, and especially the wide bit operations available through SSE and AVX, can significantly improve performance. This remains true for hashlife algorithms; use of large leaves and efficient leaf calculations can help mitigate the overhead of creating nodes and canonicalizing them with a hashtable. Unfortunately the use of such bit tricks usually requires recompilation for different rules. On some platforms it may be possible to support just-in-time compilation of appropriate low-level code to provide a combination of performance and flexibility.

We might expect performance to increase even more as AVX is extended to 512 bits. Further increases are possible as newer instructions are introduced.

As GPUs take over from CPUs in terms of overall operations per second, effective use of GPUs for Life should be considered. Existing research focuses on computation on large bounded arrays without taking advantage of any periodicity or regularity. Can we exploit GPUs with more sophisticated algorithms and gain both of these advantages at once?

Even exploiting multiple cores in a CPU can be a challenge as the algorithms get more complex. In particular, no parallel versions of hashlife have been seen that outperform a single-threaded version.

These challenges will provide great entertainment and challenges for hackers in the decades to come, much as the generation of the algorithms described in this paper have provided so much fun for this author and many others in the past few decades.

References

[1] Fujita, Toru, Daigo Nishikori, Koji Nakano, and Yasuaki Ito, "Efficient GPU Implementations for the Conway's Game of Life", 2015 Third International Symposium on Computing and Networking.

[2] The Golly Team, Golly, `https://golly.sf.net`.

[3] Gosper, R. William, "Exploiting regularities in large cellular spaces", Physica D: Nonlinear Phenomena, v.10 pp. 75–80.

[4] Goucher, Adam P., LifeLib, `https://gitlab.com/apgoucher/lifelib`.

[5] Niemiec, Mark B., "Life Algorithms." Byte Magazine, January 1979.

[6] Rokicki, Tomas G., "An Algorithm for Compressing Space and Time", Dr. Dobbs, April 2006.

[7] Warren, Hank S. Jr., "Hacker's Delight", Addison-Wesley, 2012.