# Efficient Game of Life Implementation via Embedding into an Asymmetric Neighbours 16 State Cellular Automata

Qi-rui Chen and Raef Coles

## Introduction

Through embedding the rules of game of life into an asymmetric neighbours 16 state cellular automata we can achieve unparalleled performance on a xCORE-200 eXplorerKIT. The method is not new and was originally devised by Paul Callahan in 1997[1]. Unfortunately given the increase of memory in modern systems is a rarely used implementation over faster algorithms such as hashlife. By exploiting the nature of the game of life and the greater than 16-bit word length in xCore systems[2] we demonstrate 70,000,000 cells per second (cps) implementation for a board size of 1024x1024. Of course, it also performs process control and user feedback.

## Initial Design and Previous Models

It is well known that the temporal behaviour for Game of Life tends towards a less than 5% density[3] but the Maximum Still Life Density for finite grids can be over 50%[4]. Thus, at first it may seem like a good idea to implement something smarter such as keeping a list of alive positions. It was only after seeing the memory per tile[5] that we decided to stick with a naiver implementation that stores the entire board as a matrix of cells, but possibly perform more complex channel synchronization such as recursive double reduction.

An early naive sequential model could achieve 68,000 cps on the example.pgm. Unfortunately, it was quickly found that passing the entire world as a function argument hurts the performance for larger board sizes, with it only achieving 15,000 cps on 64x64.pgm and 4,400 cps on 128x128.pgm. This model also stored the entire board state twice, and with the addition of 2 more worlds existing in the function stack, the maximum world state was a measly 128x128.pgm.

## Parallelisation

The theoretical board size for one tile is 1448x1448[5], not including any bytes for code. To achieve close to this, we turned to pointers. With a better unsafe pointer implementation, the process of implementing parallelism via shared memory was relatively easy, achieving 120,000 cps. Further optimisations using xTIMEcomposer timing analysis slowly increased the cps to 500,000, then to 800,000 (fig. 1). Being a shared memory implementation, the structure of the program is simple - every worker is allocated a portion of the world and checks a shared memory location to see if they should do work, making sure to stick to their portion.

To attempt to make it even faster, every row of the world was given a flag for whether it or any adjacent rows have alive cells in them. Then the distributor can dynamically allocate worker a set range of rows to work on, attempting to effectively ignore rows that do not need to be calculated (fig. 2). This method of partitioning attempt to make it more likely that a worker does useful work instead of iterating through a row of 0s with no neighbours. With this in place the parallel program raised to up to 120,000 cps.

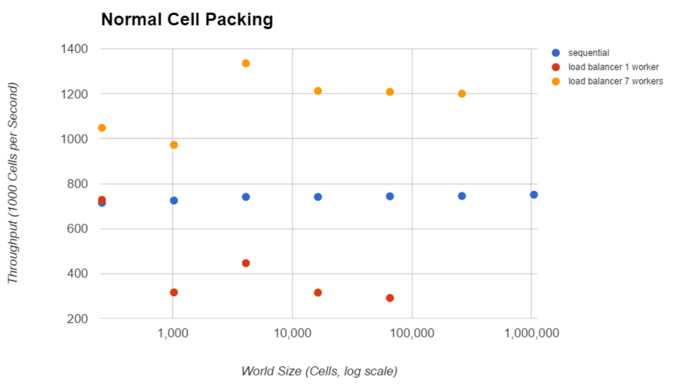


fig. 1 throughput against world size tested at 1024 iterations with random inputs

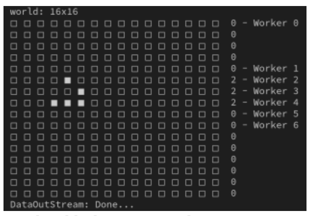
It was later found that while the speedup from the load balancer was significant, the time taken to mark adjacent rows and then calculate where each worker should start was also very large, accounting for over 50% of total runtime. This was due to the number of iterations through the whole world required by these operations. Overall the load balancer produced almost no speed increase.

## Further Optimisation

The first optimisation to be performed was to reduce the branching inside of the main loop. Most optimisations were performed on the original sequential code to achieve where it is today. The two most notable optimisations here is the introduction of a world wrap buffer and for each worker to keep a buffer of 3 rows to perform their updates. This increases the maximum world size up to 1024.

The buffer wrap simplifies the code used to lookup neighbouring cells from what used to be a positive modulo or equivalent to a simple lookup. Since this was calculated for every cell, even a minor time shaving will lead to a major performance increase.

Further optimizations such as defining macros instead of functions, performing bitwise operations and using pointers slowly increased the speed on the sequential program to 700,000 cps. To achieve speeds that others were achieving, we needed to do better.



```
world: 16x16
□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □  0 - Worker 0
□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □  0
□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □  0
□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □  0
□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □  0 - Worker 1
□ □ □ □ ■ □ □ □ □ □ □ □ □ □ □ □  2 - Worker 2
□ □ □ □ □ ■ □ □ □ □ □ □ □ □ □ □  2 - Worker 3
□ □ □ ■ ■ ■ □ □ □ □ □ □ □ □ □ □  2 - Worker 4
□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □  0 - Worker 5
□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □  0 - Worker 6
□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □  0
□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □  0
□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □  0
□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □  0
□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □  0
□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □  0
DataOutStream: Done...
```

fig. 2 load balancing workers on test.pgm

[1] Paul Callahan, 8 Nov 1997, on an efficient implementation for game of life (http://conwaylife.com/forums/viewtopic.php?f=9&t=448)
[2] XMOS Application Note: AN10000, 2016, "The natural word length on the xCORE is 32 bits and it is usually fastest to operate with 32 bit integer types (e.g. int or long)."
[3] Franco Bagnoli, 1 Oct 1998, Cellular Automata
[4] Geoffrey Chu, Peter J. Stuckey, 10 February 2012, A complete solution to the Maximum Density Still Life Problem
[5] xCORE-200 XE Product Brief, 2015, "XEF216: 256/512 RAM (KB)" (http://www.xmos.com/download/private/xCORE-200-XE-Product-Brief%281.2%29.pdf)

|  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
| 1 | 3 | 5 | 7 | 9 | 11 | 13 | 15 |
| 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
| 3 | 5 | 7 | 9 | 11 | 13 | 15 | 17 |
| 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 |
| 5 | 7 | 9 | 11 | 13 | 15 | 17 | 19 |
| 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 |
| 7 | 9 | 11 | 13 | 15 | 17 | 19 | 21 |
| 8 | 10 | 12 | 14 | 16 | 18 | 20 | 22 |

fig. 3 order of update for cells

## Functionality and Design

The key idea for our implementation is to store each block of 2x2 cells together as a single cell and perform updates using only the east, south and south east neighbours[1]. This is the same as embedding the game of life into a different 16 bit cellular automata with asymmetrical neighbours.

The next step of any given 2x2 square is the 4x4 square surrounding that contains it. Instead of calculating this every iteration update itself can pre-calculated once into an array of uint8_t containing 65536 elements. Even without much optimisation this pre-calculation only takes 0.37 seconds on the xCore-200. The next iteration is then stored into the north west 2x2 of the 4x4 square.

Not only does this mean an entire 2x2 is updated in one instruction, the worst-case memory lookups per update is 4 different indexes, down from 6. Since it is known that all cells are stored in powers of 2, it also requires less bit manipulation to update each block of neighbours.

By accounting for the north-west drift of cells using an offset, this technique easily beats out all sequential implementations and many parallel implementations in terms of speed. Our current sequential implementation can perform over 5,000,000cps which equates to 20,000 iterations per second for a 16x16 board, with a maximum board size of 1060x1060.

The parallel implementation employs a farmer with 7 workers on tile 1 with process control and user feedback threads on tile 0. Each worker is linked to the farmer via a channel and to the next worker like a list. Once again, the board is passed as shared memory to prevent expensive memcpy operations. Performing an update from the top left to bottom right (fig. 3) corner takes advantage of the north-west drift and to ensure the correct update without a buffer to prevent results interfering with future calculations. This determined our work pattern, in which each worker scans from left to right, with lower workers waiting until the cells above have been calculated before starting work.

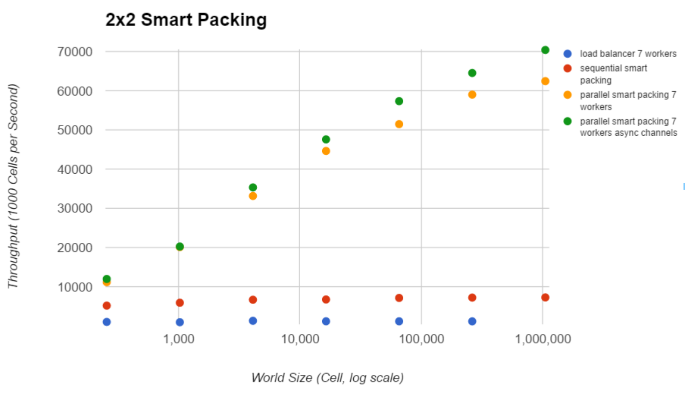


fig. 4 throughputs against world size tested at 1024 iterations with random inputs

## Tests and Experiments

### Streaming Channels

As shown in fig 4, a minor optimisation occurs using enabling asynchronous channels. We hypothesise that this is due to less synchronization being required between the threads leaving them more time perform calculations.

### The Price of Abstraction

As mentioned at the beginning, it was found that passing a large array as a function arguments will force the compiler to generate a memcpy operation. This is clear to see when analysing function timing in xTimecomposer. This was a major bottleneck in the original sequential implementation since the entire board was passed around to calculate the Moore neighbours. The solution was to either use pointers or to use macro definitions.

It was also found that wrapping data types in structs will also cause a performance penalty on the xCore-200 boards. The removal of a struct simply containing two integers saw a 120,000 cps increase and the removal of a struct containing a list saw a further 120,000 cps increase (fig. 4).

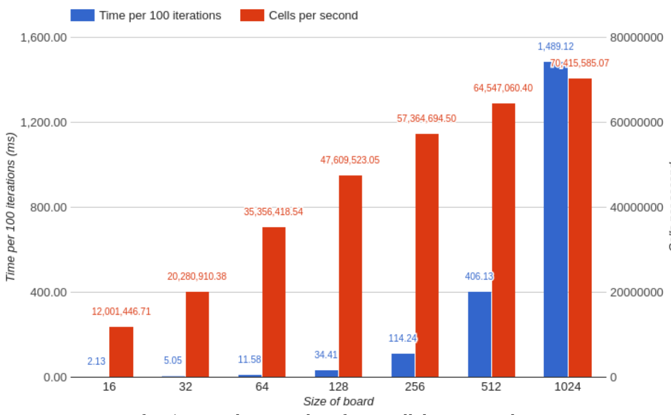


fig. 5 general timing data for parallel smart packing

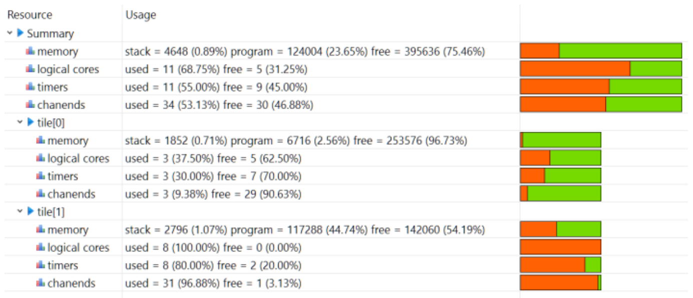
## Critical Analysis

Our programs were compiled with the -O3 flag and has a maximum world size of 1052x1052. This is due to a large amount of memory being used by the hash function. Unfortunately, this only 50% of the theoretical maximum for the core and it is very difficult to use the other core since the implementation uses unsafe pointers.

The implementation also uses 31 channels on tile 1 even though we only explicitly declare 17 due to a quirk of the XC language, which we encounter due to the requirement to declare the unsafe pointers to be passed to the workers. Unfortunately, since we are almost at the channel limit and we have hit the core limit, it would be very difficult to add new workers or enforce a graceful shutdown. In theory, less channels can be used at the expense of speed.

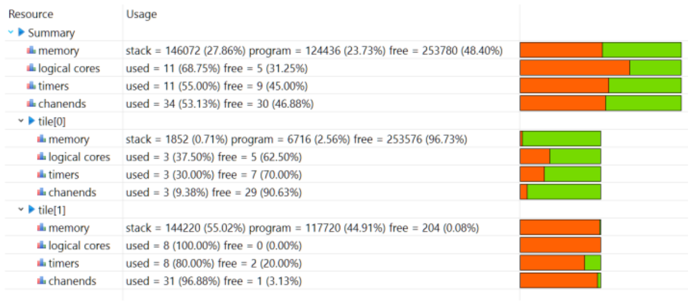
The maximum throughput of our program was recorded at 70,427,647cps when it was calculating a random board of size 1024x1024 (fig. 4). As shown by fig 2, the speed of the program increases with number of cells showing that the current bottleneck is due to channel communication, which decreases in relative frequency as the size of the board increases.

The current implementation does not fix the timer overflow after 43 seconds. Nor does it implement any dynamic load balancing techniques although it is very fast already. A very simple implementation would be to keep track of the number of alive cells each iteration. If this number stays the same, check to see if the board is a still life (no new births or deaths). If this is the case, then the board will remain a still life forever thus all calculation can stop.



| Resource | Usage |
|---|---|
| Summary | |
| memory | stack = 4648 (0.89%) program = 124004 (23.65%) free = 395636 (75.46%) |
| logical cores | used = 11 (68.75%) free = 5 (31.25%) |
| timers | used = 11 (55.00%) free = 9 (45.00%) |
| chanends | used = 34 (53.13%) free = 30 (46.88%) |
| tile[0] | |
| memory | stack = 1852 (0.71%) program = 6716 (2.56%) free = 253576 (96.73%) |
| logical cores | used = 3 (37.50%) free = 5 (62.50%) |
| timers | used = 3 (30.00%) free = 7 (70.00%) |
| chanends | used = 3 (9.38%) free = 29 (90.63%) |
| tile[1] | |
| memory | stack = 2796 (1.07%) program = 117288 (44.74%) free = 142060 (54.19%) |
| logical cores | used = 8 (100.00%) free = 0 (0.00%) |
| timers | used = 8 (80.00%) free = 2 (20.00%) |
| chanends | used = 31 (96.88%) free = 1 (3.13%) |

fig. 6 xta resources output for 16x16



| Resource | Usage |
|---|---|
| Summary | |
| memory | stack = 146072 (27.86%) program = 124436 (23.73%) free = 253780 (48.40%) |
| logical cores | used = 11 (68.75%) free = 5 (31.25%) |
| timers | used = 11 (55.00%) free = 9 (45.00%) |
| chanends | used = 34 (53.13%) free = 30 (46.88%) |
| tile[0] | |
| memory | stack = 1852 (0.71%) program = 6716 (2.56%) free = 253576 (96.73%) |
| logical cores | used = 3 (37.50%) free = 5 (62.50%) |
| timers | used = 3 (30.00%) free = 7 (70.00%) |
| chanends | used = 3 (9.38%) free = 29 (90.63%) |
| tile[1] | |
| memory | stack = 144220 (55.02%) program = 117720 (44.91%) free = 204 (0.08%) |
| logical cores | used = 8 (100.00%) free = 0 (0.00%) |
| timers | used = 8 (80.00%) free = 2 (20.00%) |
| chanends | used = 31 (96.88%) free = 1 (3.13%) |

fig. 7 xta resources output for 1052x1052

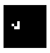


fig. 8 test.pgm after 2 generations

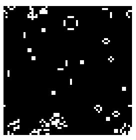


fig. 9 64x64.pgm after 1024 generations

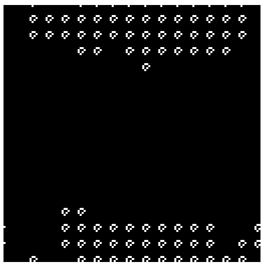


fig. 10 128x128.pgm after 1024 generations

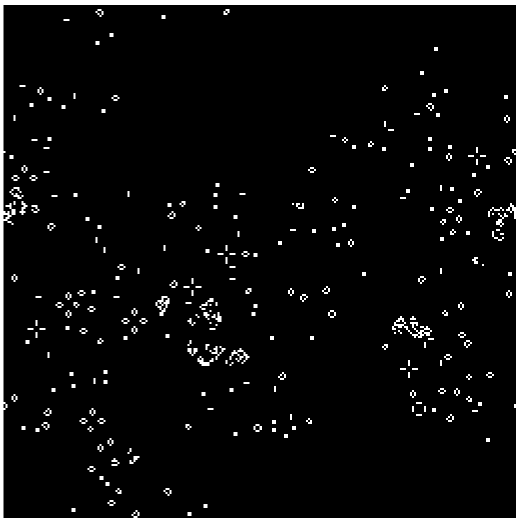


fig. 11 256x256.pgm after 1024 generations

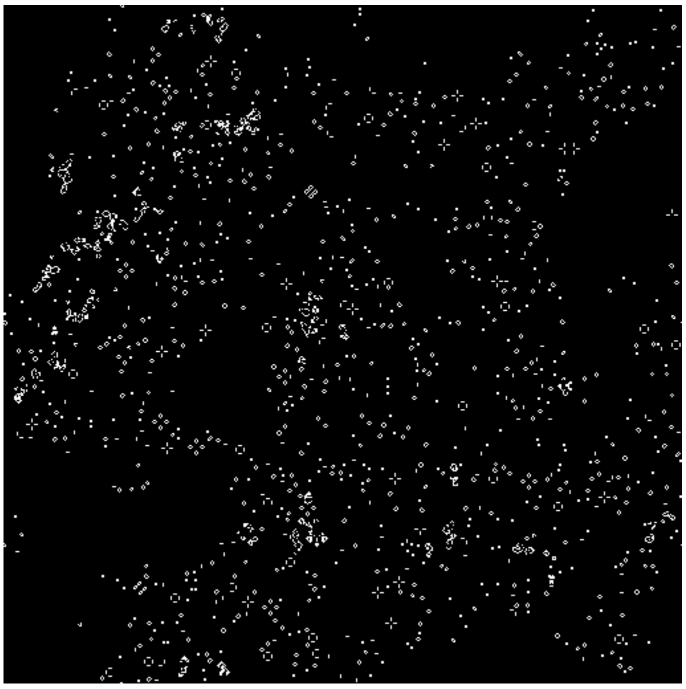


fig. 12 512x512.pgm after 1024 generations

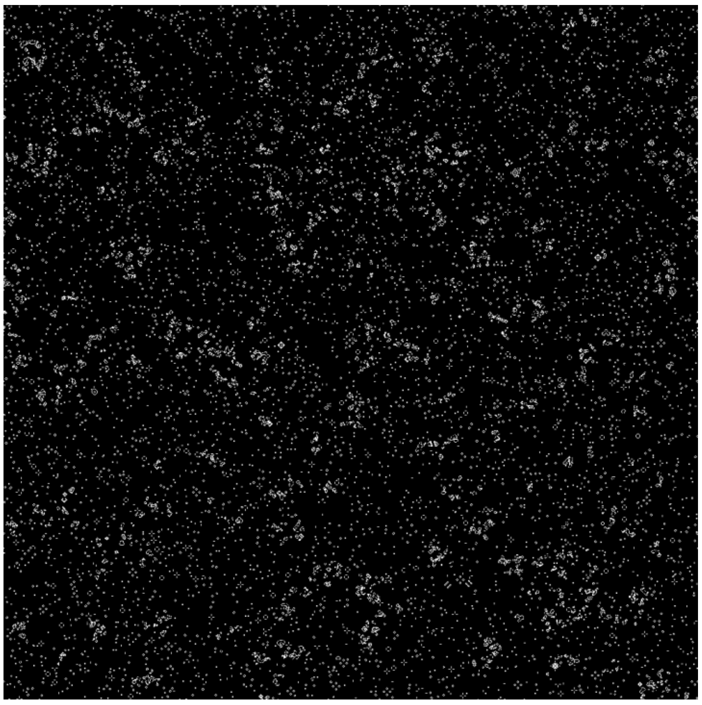


fig. 13 1024x1024 random seed 0 after 1024 generations