Report: xCore-200 Cellular Automaton Farm

Callum Pearce and Jacob Montgomery

G403 2nd Year

cp15571@my.bristol.ac.uk jm15969@my.bristol.ac.uk

Functionality Implemented

We have implemented the game of life logic for as many turns as is specified for a given image. The program can apply the rules to images with a size well above 512x512 pixels quickly and efficiently. This is done by using a combination of geometric and processor farming, where multiple worker processors apply the game logic to specific parts of the image and report back their results to a farmer processor. This farmer processor builds the next turn of the world by combining the workers' results together. Many variables can be changed within the program for experiments to show changes in run time. These changes are things such as number of workers, synchronous vs. asynchronous communication and variable type used to store the world and communicate it. The program also allows the processing to be paused at any time by tilting the board beyond 30 degrees. This causes output to the screen of the the time spent processing, number of live cells on the current turn and turn number. The button SW1 on the board begins the image processing, whilst SW2 begins the output of the image on the current turn.

Image Processing

Geometric and processor farming were used to parallelise the processing of the image and in turn reduce the time needed to apply the game rules to a given image. This has been implemented by running the workers on multiple processors, with each worker (findNextGenLine function) performing an identical task of looping through the partial array sent to them from the farmer (sendToWorkers function). They apply the game logic to each individual cell in their stored array to compute what the specific part of the image converts to; one turn later. The partial array is then sent back to the farmer (sendToWorkers, createNextGenArray functions) which stitches all the data from the workers correctly back together to produce the next generation of the world. In the main function it has been specified what tiles specific functions should be located on. Workers have been spread across multiple tiles which comes with the cost of reducing performance speed as it takes longer for the workers to communicate with the farmer if they are not on the same tile. However, it allows for larger images to be processed as multiple tiles' memory are being used to store the partial arrays which the workers are processing. The image is split up based on the height divided by the number of workers. This means each worker is sent rows of the image based on the split size of the image. Special attention is required to the cells on the edge of the images; we had to implement it so that the top of the image wraps to the bottom and the same with left to right. Without having done this, live cells would disappear from the edge of the image. Below, the strategy of storing and sending these rows is highlighted in more detail.

Memory Management

To allow the program to manipulate a large world concurrently, a variety of strategies has been implemented to make sure memory is used efficiently. Bit packing has been used which largely reduces the stored array size of the world. Instead of storing 1 cell inside each unsigned character (which is 8 bits long), 8 cells are instead stored, where a 1 represents a live cell and a 0 represents a dead cell. This makes each row in the stored world array 8 times smaller than the original, which uses far less memory. Implementing bit packing means the game logic function had to be adjusted to access the correct cell from a single unsigned char. Bit shifting was required to carry this out. This further complicated the computation of finding the next generation of cells for each worker, however the trade off is very small as bit shifting operations are a very simple computation. In return the program can handle much larger image sizes. The program also allows wrapping for 32 bit integers and other data types specified, and the game rules are still applied correctly. Applying the game logic to the wrapped bits is far quicker then unwrapping the bits then apply the game logic before finally wrapping again. This is because it requires multiple nested for loops to wrap the world array.

Worker Communication Strategy

A simple yet fast communication strategy was implemented to allow the workers to communicate with the farmer processor in order to send and receive the partial arrays. Due to the simplistic communication required, channels were used for communication. The communication process works as follows: loop through every value in the array and send it on the specified workers' channel (sentArraySend function). Then on the receiving end, loop through the newly constructed array and fill in the values using the received data. We have also implemented asynchronous channels in order to send multiple array values before one is received by the receiver thanks to the usage of a buffer. However the board only allows for a certain number of asynchronous channels so this could not be implemented for 8 worker channels. However for 4 workers it was possible, the data on this is displayed in the tests and experiments section. Bit wrapping was also an essential part of the communication strategy as the number of variables needed to communicate a row in the world array is 8 times smaller than previously. This was experimented with further by wrapping the rows of the world in integer arrays instead of unsigned characters, this meant 32 cells were located in one variable. This seemingly should speed up the total processing as there are less steps in the communication of the partial arrays, however the time required to wrap the rows in 32 bits is far more than what is required for 8 bits; causing the run time to increase. Also 32 bits wrapping does not work 16 bits images at the minimum row size of the world array required is 32, providing yet another reason for the advantage of using unsigned chars.

Timing

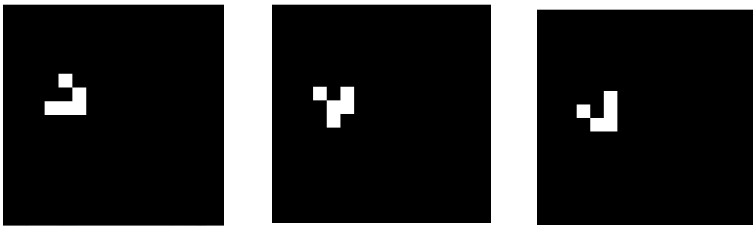
To understand whether an implemented feature actually improves run time, a timer was added to the distributor, this provided accurate information on the programs performance. However xC's timer resets after 42 seconds, therefore when processing larger images the timer had to be modified in a way so that it can accurately record times above 42 seconds. A select case with a 'when' statement allowed us to record the time of each round accurately and reset the timer every two seconds and add the time for a round to be completed to the total time recorded.

I/O

To implement the button and LED functionality the distributor has been supplied with two channels which it can send data to the showLEDs function to change the LED display pattern on the board, as well as receive data from the buttonListener function. The two functions run in parallel to the distributor on a different tile, until the program is stopped. They both work in a similar way by constantly monitoring the channels within a select statement in an endless while loop. As soon as a value is sent on a channel either function is ready to receive it and interact with the I/O accordingly. The orientation function works in the same way however only sends a value to the distributor if the board is tilted beyond 30 degrees to remove unnecessary communication.

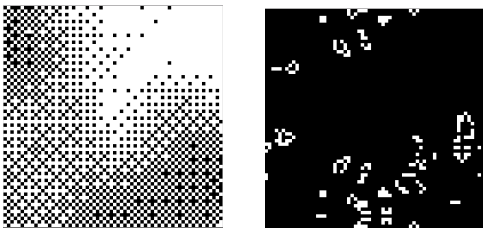
Tests and Experiments

Below is the initial 16x16 test image followed by the resultant image after the 1st and 2nd round of our game of life program:

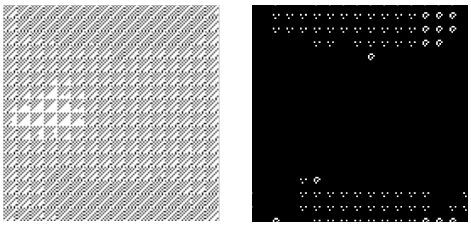


We carried out experiments of testing the logic over multiple different rounds. Below are the results for all the other provided test images, each result image is the game of life after 100 rounds:

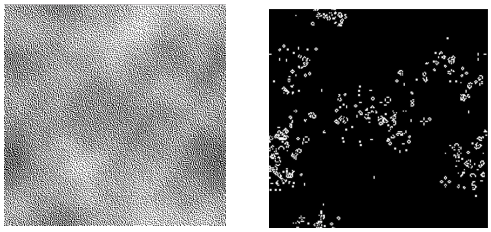
64x64:



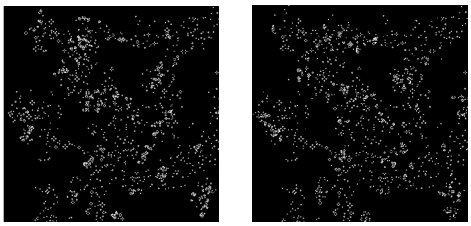
128x128:



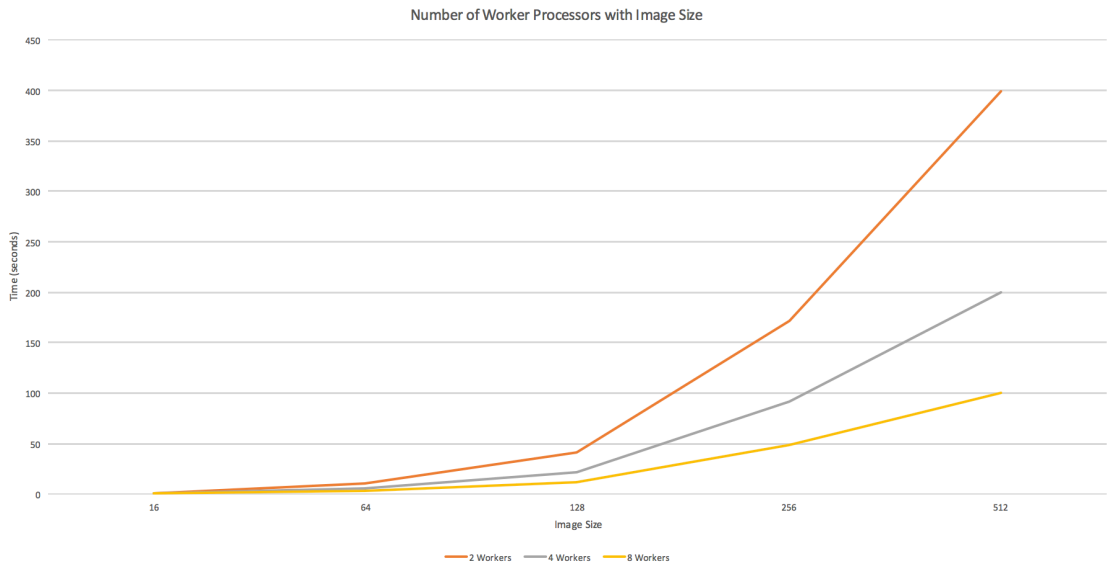
256x256:

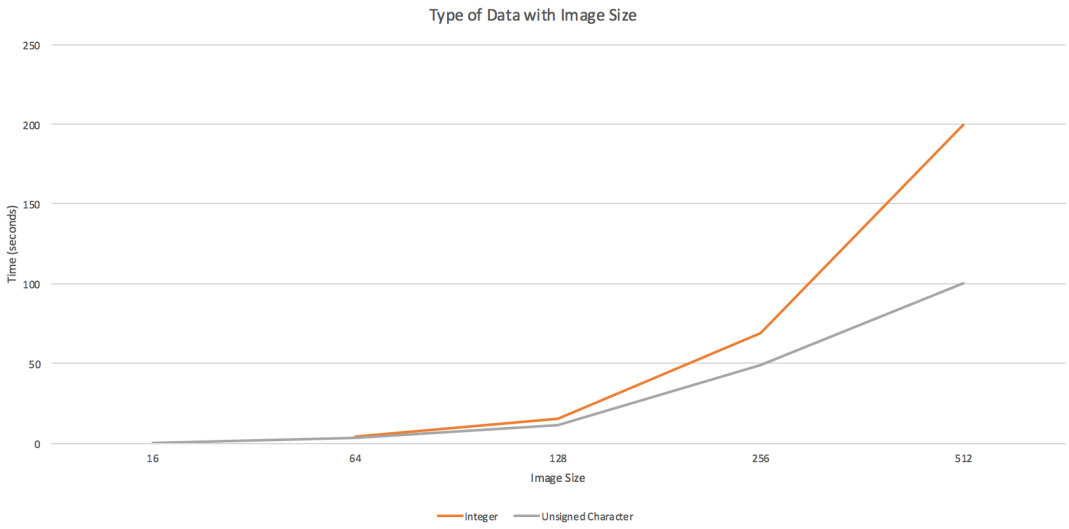


512x512:



We carried out many tests on our program to show increases in performance. We tested the changes in performance when using 2, 4 and 8 worker processors. The change in performance when using asynchronous communication instead of synchronous communication was also thoroughly tested. The last form of test was of different forms of data exchange and storage; unsigned char and integer. Below are the results of these tests.





Asynchronous vs synchronous communication (4 workers and unsigned characters)

Image Size	Asynchronous Communication	Synchronous Communication
16x16	0.4	0.4
64x64	5.7	5.8
128x128	21.7	21.7
256x256	90.9	91.4
512x512	200.8	200.0

From the results above it is clear that the most important factors in reducing processing time for the images is the number of worker processors and the the type of data used to store the wrapped world array. As for the number of workers, as we double the number of workers, the processing time halves. This is due to the fact that the amount work done by each processor is essentially halved making the average time taken for each worker processor to process each partial array half. As we experimented with only a small number of cores, we did not encounter bottleneck; the communication required between the farmer and all the workers being too large and slowing processing time. This, combined with the fact that our communication was kept relatively simple, made increasing the number of workers lead to a large increase in the performance of the program. As for the type of data used, unsigned characters provide the quickest processing time due to less bit packing needed, as discussed in the memory management section. Due to the simplicity of the communication required, changing to asynchronous communication did not provide a significant improvement in run time, making this variable not very relevant for the project. We can evaluate overall that to get the fastest performance for any given image inputted to the program the following variables should be set; number of workers = 8, communication and data storage type = unsigned char and enable streaming however this only works for 4 worker processors.

Critical Analysis

From our results we found that processing time compared to image reading and writing time was very large, we could have taken advantage of the time spent idling by the majority of the processors by beginning the processing of the image on the current rows of the image read in that had already been read in. For example if 100 rows had already been read in then get 8 workers to start processing these rows. Then begin to process next 100 rows which are read in for 100 turns and so on, then stitch the final image together. Another area for improvement in our code would be to only stitch the image together after 100 rounds of processing or when button SW2 was pressed. This is instead of stitching the image together every round which requires 100 times more bit packing as functions like findSentArray have to be called 100 times more. These were the two major areas of processing improvements which if we had more time we would attempt to implement. We ran our program on an image of size 720x720 and the processing time was 200 seconds. We researched the size of memory available on the board and this was very close to the maximum amount of memory available when we stored the entire wrapped array corresponding to the image on the board.