Ngoc Khanh Nguyen (id: nn14160) *Silviu Tugulan (id: at14873)*

xCore-200 Cellular Automaton Farm - Report

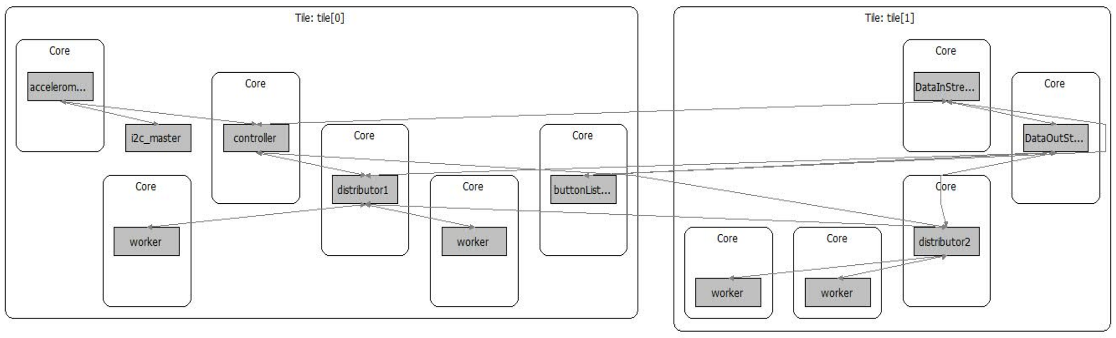
1. Introduction

Processing images is becoming a challenge which we can encounter in many areas of applied computer science. Nowadays we are able to store large images and consequently we try to find faster and faster algorithms to process them. One of the solutions might be to use standard concurrent programming techniques such as dynamic data division. Thus many threads can work on specific areas of the image at the same time independently, combining all their results at the end. Eventually the whole image is processed. In this report we show how we implemented the *Game of Life* using farming technique.

2. Functionality and design

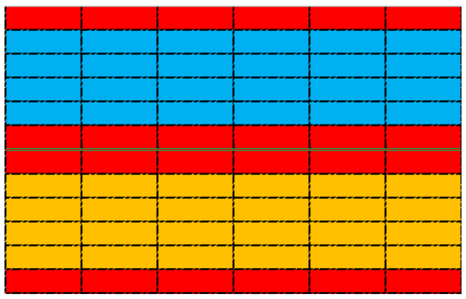
2.1 Overview of the system

Our program consists of the following processes which interact with each other in order to evolve the *Game of Life*: DataInSream waits to receive the start signal from the buttons and then reads the image and sends the values to Controller. Controller splits the image to the two distributors and also sends LEDs patterns and listens to accelerometer. Distributors give areas to workers and receive the results from them. DataOutStream listens for button press and tells the distributors when to enter the sequence where they send data to DataOutStream. The buttonListener listens for any button press.



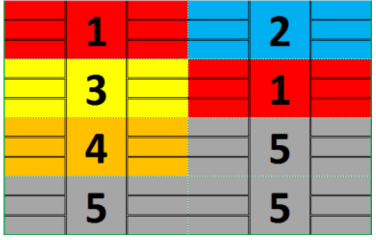
2.2 Storing images in the memory

In order to store large images on the XMOS board, we use memory on both tiles. We split the image (horizontally, suppose) into two halves and then send them to distributors. Note that *Distributor1* does not have enough information to process all of its cells. Indeed, in order to compute the number of live neighbours of a cell in the very first row of the image, we have to look at some cells in the last row. Hence, we send not only one half of the image but also the first and the last row of the second half. Then we are sure we have all the needed information for *Distributor1*. Analogously, we send the first and the last rows of the first half of the image to *Distributor 2*.



Example: In this case we have a $6x12$ image and we split it horizontally (the green line). We send all blue and red cells to *Distributor1* and all yellow and red cells to *Distributor2*.

2.3 One round of processing image



In every round both distributors get half of the new image and assign areas, which have not been calculated yet, to free workers. Then, after the workers finish processing, distributor collects information from them and combines the results. It finishes when the half of image it is responsible for is completely processed and all of its workers are free. Consequently, during one round every worker has always an area to work on until all areas have been assigned.

Example: We are given a $6x12$ image. Workers 1, 2, 3 are assigned to *Distributor1* and workers 4, 5 are assigned to *Distributor2*. The first worker gets the first area to start working on it. Then, the second one gets the next area to farm. Since worker nr. 3 is free, it gets the next field. Then, worker nr. 1 finishes and gets a new area to work on. Since all fields have been already assigned, workers 1, 2, 3 will not get any more work. *Distributor2*, however, sends the first area to worker nr. 4. Then it sends the next field to the fifth worker. While worker nr. 4 still farms, worker nr. 5 finishes and gets another field to process. Worker nr. 4 works slowly and hence worker 5 finishes earlier and gets all the areas to farm. Since all areas have been already assigned, workers 4 and 5 will not get any more fields to farm. One round is completed.

After one round is over, we set the output image as the input image and we are ready for the next round of processing.

2.4 Optimisation techniques

a) Image compression

In order to work with large image (for example $1024x1024$) we might want to use compression techniques. Let us consider a matrix corresponding to the input image with an entry 0 if the corresponding cell is black and 1 if it is white. Then we can treat a row of the matrix as a binary representation of an integer (or sequence of integers). Thus our array T_i has only size$\left(\frac{n}{2}+\ 2\right)\lceil\frac{m}{31}\rceil$.

b) Skipping *less interesting* areas of the image

First of all, we check if the image is all white or all black, then after one round it becomes all black. This leads to the following simple observation:

Claim: *Let $[(a,b),(c,d)]$ be a rectangle where upper-left corner has coordinates (a,b) and lower-right corner has coordinates(c,d). If $[(x_1-1,y_1-1),(x_2+1,y_2+1)]$ is all black or white, then after one round $[(x_1,y_1),(x_2,y_2]$ is all black.*

Hence, before sending the area to the workers, a distributor first checks if it is all black or white and skips it, if it is the case.

c) Considering transpose of an image

In subsection 2.2 we mentioned that we split the image into two halves. Now, consider any distributor and the size of its array T , where T is the compressed version of the image matrix. Then, if we split it horizontally and use compression, T will have size $\left(\frac{n}{2}+\ 2\right)\lceil\frac{m}{31}\rceil$. On the other hand, if we split it vertically then T has size $\left(\frac{m}{2}+\ 2\right)\lceil\frac{n}{31}\rceil$. Therefore, if $n\leq m$ then we get the smallest size of T by splitting the image horizontally. So the way we should split depends on the size of input image. But implementing two ways of splitting is quite complicated. The following lemma turns out to be helpful:

Lemma: *Let M be a matrix corresponding to input image X and f be a function so that $f(M)$ is a matrix after processing X according to the rules of Game of Life . Then $f(M^T)\ =\ f(M)^T$.*

As a consequence, if $n \leq m$ then we split horizontally and proceed as described earlier. Otherwise, we do the transpose operation on the image, then split it horizontally and continue processing. Before writing it to the file we remember to transpose the output image (using the fact $(M^T)^T = M$).

As a result, we reduce the amount of used memory and also the communication between the two distributors.

2.5 Deadlock-freedom

Looking at the layout of the code, it is easy to see that the only moment when the deadlock might appear is when the distributor shuts down but workers still want to communicate. Note that distributor listens to commands until there are no more areas to assign and also when all workers have finished farming. In other words, if distributor stops listening to workers then we are sure they will not send any data (unless new round begins) and thus our program is deadlock-free.

3. Tests and experiments

3.1 Experiment 1

We have designed our system so that it would be very dynamic. As a result, we have the size of our workers as one of the parameters. We wanted to see how the processing time is influenced by this parameter in order to determine the optimal worker size. So we carried out some experiments for different images (small and large), while changing the size of the workers (see Figure 1 and 2). As we can see from the results, the processing time drops from worker size 4 to worker size 8 and then the time increases gradually with the worker-size. Explanation: Let $n\ x\ m$ be the size of the matrix and x the worker size. Every time a worker gets an area, it gets $(x+2)^2$ numbers. Since the total number of areas to assign is $\frac{n}{x}\frac{m}{x}$, we know that distributors send $\frac{n}{x}\frac{m}{x}\ (x+2)^2$ numbers in total. Note that $\frac{n}{x}\frac{m}{x}\ (x+2)^2$ is decreasing on interval $[1,\ \infty)$, so the bigger work-size we choose, the less communication we have between distributors and workers, hence the drop in time from worker size 4 to worker size 8. Now, keep in mind that we also skip areas that are all black(i.e. don't send them to workers). So the smaller worker size we choose, the bigger chance we have to skip some black areas, hence reducing the communication and the computations. So if we increase the worker size, we have less chances to skip an area, so the communication and the number of computation will increase. According to our experiments, an optimal worker size with the best trade – off would be 8.

Figure1: y-axis: seconds; x-axis: worker size

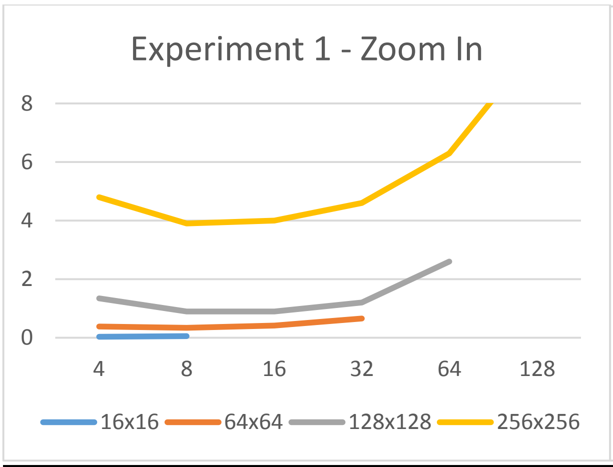
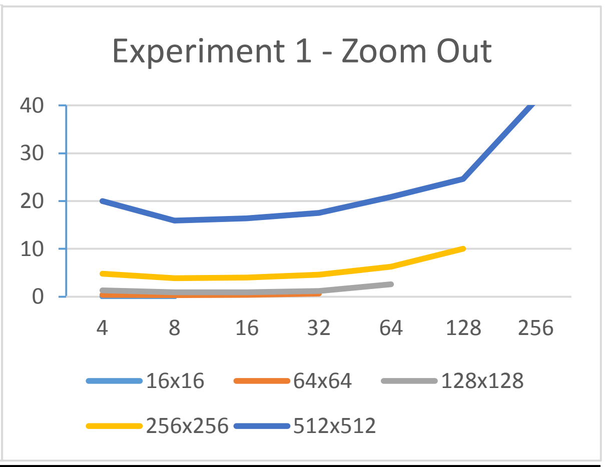


Figure2: y-axis: seconds; x-axis: worker size



3.2 Experiment 2

For this experiments we have chosen the optimal worker size – 8 – and changed the number of worker threads for some small and large images (Table 1). We can clearly see from the table a decrease in time with the number of worker threads (please note that we can have a maximum of 9 worker threads). Explanation: the decrease in time happens because we have more worker threads that can work in parallel, hence more areas processed at the same time. Also, the decrease is very big from 2 workers to 4 workers and then smaller and smaller. That happens because with more workers, there will also be more communication, so the difference that an extra worker can make will be smaller and smaller. For Table 2 we have chosen two images that can show the benefit from using transpose for images where there is a big difference between their height and width. We can see that the amount of reduced time can be very big for large images (32 x 1024). The difference is less noticeable for smaller images.

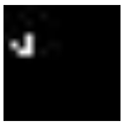
Table 1

wks/IMG	16x16	64x64	128x128	256x256	512x512
2 wks	0.06s	1 s	2.6s	9.3s	40.3s
4 wks	0.05s	0.5s	1.5s	5.4s	22.7s
6 wks	0.05s	0.4s	1.1s	4.2s	17.6s
9 wks	0.05s	0.34s	0.9s	3.9s	16.3s

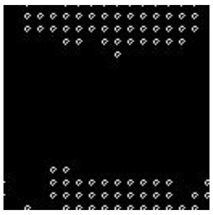
Table 2

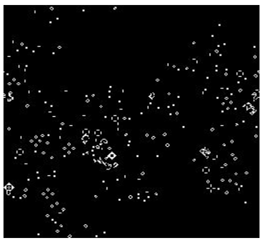
	33x64	32x1024
Without Transpose	0.129s	2.2s
With Transpose	0.116	1.4s

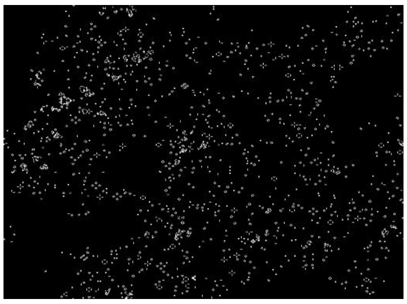
The results over 1000 rounds for the provided images:


16x16

64x64


128x128


256x256


512x512

After experimenting with different image sizes, we discovered that our system can store and process an image with a maximum size of 1220x1220.

4. Conclusions

To conclude, we have managed to implement a system that evolves the *Game of Life* by applying concurrent programming techniques, while also trying to efficiently use the resources of the provided hardware by applying optimization strategies. Other optimizations that can be included in our system would be using just one array instead of two for holding the current state and the next state of the game (more available memory), sending compressed data to workers instead of bits (less communication) and also, when one distributor has finished processing its half of the image, send workers to "help" the other distributor to finish its half.