An investigation of the efficient implementation of Cellular Automata on multi-core CPU and GPU hardware.

Mike Gibson (Corresponding author), Ed Keedwell and Dragan Savić

College of Engineering, Mathematics and Physical Sciences

University of Exeter, Harrison Building, North Park Road

Exeter, Devon

EX4 4QF

{mg278, E.C.Keedwell, D.Savic}@ex.ac.uk

(+44)1392 {724075, 724014, 723637}

Keywords - Cellular Automata (CA), General Purpose Graphic Processing Unit (GPGPU), OpenCL, Single Instruction Multiple Data (SIMD), Single Instruction Multiple Thread (SIMT), OpenMP.

Abstract - Cellular automata (CA) have proven to be excellent tools for the simulation of a wide variety of phenomena in the natural world. They are ideal candidates for acceleration with modern general purpose-graphical processing units (GPU/GPGPU) hardware that consists of large numbers of small, tightly-coupled processors. In this study the potential for speeding up CA execution using multi-core CPUs and GPUs is investigated and the scalability of doing so with respect to standard CA parameters such as lattice and neighbourhood sizes, numbers of states and generations is determined. Additionally the impact of 'Activity' (the number of 'alive' cells) within a given CA simulation is investigated in terms of both varying the random initial distribution levels of 'alive' cells, and via the use of novel state transition rules; Where a change in the dynamics of these rules (i.e. the number of states) allows for the investigation of the variable complexity within.

1. Introduction

Modern hardware is becoming increasingly parallel in nature with modern commercially available single CPUs equipped with up to 8 cores, and Graphical Processing Units (GPUs) having many hundreds or even thousands of cores (e.g. the latest Nvidia card has 3072 cores [1]). This increase in parallelism provides challenges and opportunities in that many software packages will need to be re-engineered to exploit the new hardware and techniques such as cellular automata become more attractive as their structure lends itself to the multi-core/many-core nature of hardware. In recent years the development of the GPU into a processor capable of General Purpose processing has received particular attention, due to fact that GPUs have needed even greater parallelism than their CPU counterparts. The literature shows that although methods for parallelisation on the CPU are fairly well established and understood that some of the unique architectural differences between the General Purpose-GPU (GPGPU) and CPU are not so well known. GPGPU computing is still very much an expert field, which means that there are few comprehensive studies of the performance and scalability of the performance gains possible through GPGPU computing, particularly for cellular automata. Cellular automata (CA) are excellent techniques for the efficient simulation of a wide variety of systems and natural phenomena, in addition to being interesting from a theoretical perspective. In this paper, the new open standard OpenCL is used to perform benchmark tests using the well-studied 'game of life' cellular automaton [2] along with some novel variants. Experimentation is conducted on a variety of different parameterisations of cellular automata that impact performance, notably the lattice size, the number of states, the neighbourhood size, the complexity of the state transition rule sets, and population levels within the random initial configuration by assigning the chances of a cell being alive in the initial configuration (initial configuration distribution probability). It is found that each of these key parameters affects the ability of multi-core CPU and GPU architecture to speed-up CA execution. In addition, these key CA parameters cover the majority of variations in CA that might be implemented to simulate a variety of natural systems. Through the intensive study of the multi-core/many-core speed-up available for a wide variety of parameter settings, it is possible to infer some general properties of CA efficiency operating on multi-core CPU and GPGPU hardware.

1.1. Cellular Automata

Cellular Automata (CA), were first conceived by John von Neumann in the 1950s [3] as a model of a simple system capable of abstracting the concept of self-reproduction of living organisms; since this time, the basic framework of CA has been widely applied to many natural systems which are governed by local laws. The automata consist of a lattice of cells commonly in a 2D regular grid, with each cell adopting one of a number of states. The dimensionality of the lattice, its shape, cell shape (e.g. irregular, square, hexagonal) and the number of states are all problem-specific parameters. Each cell is updated using state transition rules (a rule set), that uses the current state of that cell and its adjacent neighbours in a particular pattern (e.g. Moore, Von Neumann neighbourhoods) to determine the state of that cell in the next iteration of the algorithm. Each cell is updated in an idealised parallel sense to form a new lattice of cells states, and once a completely new grid is formed the process is repeated for a number of iterations, referred to as generations. The describe model of CA described above is the most conventional, but variants exist, such as asynchronous CA, CA with memory, indeterminate CA, etc.

1.2. Applications of Cellular Automata

There are a wide variety of scientific papers using CA models, for a large number of modelling uses and disciplines. These applications include but are not limited to:

- **Physics** (fluid/turbulent fluid flow [4] - lattice Boltzmann methods [5] [6] [7] [8], reaction-diffusion [2] [9] [10], laser dynamics [11], magnetization [12], collision detection, fracture modelling [13]);
- **Chemistry/Biophysics** (reaction-diffusion [14])
- **Biology** (systems biology [15], cell simulation internal-chemical/reaction diffusing within an e-coli cell [14], groups of cells-Keratinocyte skin cells [16], cardiac tissue [17], artificial life/systems [18] [19], and tumour cell growth, bacteria swarming, epidemiology [20], viral infection/epidemic spreading [21])
- **Computer science** (image processing/visualisation [22] [18] [19] [20] [23] [24] [25], algorithmic study/benchmarking [8] [4] [21] [16] [20] [26] [27] [28] [29] [30] [31] [32], cellular programming/GA/GP-classification [27] [33], cognitive science [34], cryptography [21], computer graphics and animation/virtual reality [10])
- **Geography/Environmental Sciences** (population movements/dynamics [21] [35], land uses/deforestation [36] [30] [37], forest-fires [21], wildfires [38]);

- **Engineering** (wet chemical etching [24], designing hardware(FPGA) to run CA, communication [21])
- **Mathematics** [39]
- **Hydroinfomatics** (fluid dynamics [8] [4] [33], sewer optimisation [40], pluvial flooding modelling [41])
- **Economics** (stock markets) [21]

This variety of applications demonstrates the wide applicability of CA models and in many cases illustrates that the discretisation of time and space for use with a CA model is able to provide results of acceptable accuracy with greater efficiency than traditional models. Although the application level details for these CA will be different, they each implement the core principles of CA which is that time and space are discrete, cells will be updated in parallel and interactions between cells will be local. Traditional CA are also required to have discrete sets of possible states, but continuous cell states are now used with some frequency. From this variety of applications it is also clear that any improvement in the speed of the basic CA system will be through efficient use of modern hardware which therefore has the potential to deliver large-scale impact in a wide variety of scientific fields.

1.3. Multi-core CPU and Many-core GPU computing

With this wide variety of disciplines and applications, a growing number of modellers are harnessing the inherent parallelism of the CA algorithm in modern hardware, i.e. multi-core CPUs and GPUs. This is motivated by the idea that the multi-core nature of most modern CPUs which is allowing Moore's law to continue to predict processing speed increase. [16]. Also several sources suggest that co-processors like GPGPUs, with their inherently many-core nature, may be increasing in performance at a quicker rate than their CPU counter parts [13], with Fan et al. stating that: "*Driven by the game industry, GPU performance has approximately doubled every 6 months since the mid-1990s, which is much faster than the growth rate of CPU performance that doubles every 18 months on average (Moore's law), and this trend is expected to continue.*" [7]. Although this publication dates from some eight years ago and it is now an established fact that significant increases in computational power in many areas will need to be achieved through the use of parallelism rather than the increase in performance of a single core. Therefore the scientific community needs algorithms which will scale to take advantage of this emerging parallelism, such as CA, and to understand how these algorithms will scale to the emerging hardware available.

2. Related Work

There are few attempts in the literature to develop parallel CA algorithms and to investigate how exactly they will interact with many-core technologies. There are a number of examples of implementations which are discussed below; however few of these investigate the spread of possible speed-ups, or how the variation of the CA's base parameters (e.g. lattice size, number of generation, number of states/data types, neighbourhood sizes, or rule complexity) affects these speed ups.

Recent approaches to the use of CPU and GPU computing to speed up CA execution include Lopez-Torres et al. (2012) who used CA to simulate laser dynamics, and noted in summary of recent CA-GPGPU works that "*Depending on application, they are offering a 10 to 100 times speed up at price points extremely affordable*" [21]. Rybacki et al. (2009) examine and benchmark CA algorithms and investigate different levels of multi-threading with either a "brute force" or sparse ("discrete" which only applies the rule set to those cells that might change) method of implementation, on both the CPU and GPGPU of several machines. They use five different rules sets: the game of life, parity, majority, wireworld, and a benchmark case. They find that "*there is no perfect algorithm for everything*" [26], which is largely due to the discrete algorithms being outperformed on the GPGPU, but they note that this is due to the small size of the grid and/or the small number of living cells after the first few generations. This work highlights the issue of sufficient parallelism, if CA with a low number of cells (and therefore low number of parallel elements) is applied to hardware with a large number of cores there is a high likelihood that computational resources will be wasted due to the lack of sufficient algorithmic parallelism. The algorithmic representation must match the many-core nature of the GPGPU and sparse representations either don't work well or are difficult to code on the GPGPU. Also, if a rule set is somehow known to produce little to no *activity* (the number of cells alive and/or changing value over the whole simulation) within a given grid and initial configuration then it may still be more efficient to use a sparse representation on the CPU, as there is relatively little work to be done. For these reasons, only the brute force algorithm is investigated here, i.e. the full neighbourhood look-up and rule set is applied to each and every cell in the grid, as opposed to only applying such updates to cells which are either currently alive and those within a single neighbourhood radius range, for each generation.

There are circumstances where a sparse implementation has been implemented on a GPGPU, for example Ferrando et al. (2011) [24] have employed an Oct-tree representation which subdivides a 3 dimensional cube of space into 8 smaller cubes at each branch of the tree. Although this does mean that the tree structure must be stored and manipulated using the CPU, a lot of processing can still be carried out on the GPGPU. This is done by means of the CPU organising the tree structure, which then issues commands to the GPGPU to order the respective array of 'memory clusters'. These memory clusters are organised linearly upon the GPGPU, and each contains all the information needed to process a single cell (i.e. the cell and its neighbour's states), these can then be processed in bulk by the GPGPU. The optimised use of further GPGPU data structures are used to minimise the amount of traffic between the CPU and GPGPU, which is known to be a bottleneck. However, Ferrando et al. [24] are more keenly interested in carrying out the high resolution of simulation in feasible amounts of time, and so do not directly claim that this approach provides speed ups because their system works as a co-operation of the CPU and GPGPU. In experiments conducted later in this paper it is found that with the GPGPU as a co-processor, a certain amount of instruction passing from the CPU is necessary.

Of particular interest is work by Zaloudek et al. (2010) [27], in which they examine the evolution of 1D CA systems, using Nvidia's Compute Uniform Device Architecture (CUDA). CUDA describes both Nvidia's architecture and high level language for its manipulation. Zaloudek et al. examine the possibility of parallelisation at the level of cells, but also an evolutionary systems CA system which requires a population of solutions be evaluated, often with each possible solution (state transition rule set) needing to be run under a number of initial conditions in order to reach an average fitness. They have examined the possibility of parallelising their algorithm in terms of 'training vectors' and 'individual solutions', as well as by cells. The results are encouraging in favour of using parallelisation at the 'training vectors' and 'individual solutions' levels. However this is due to the fact that they confine themselves to the very closest, fastest and conversely smallest forms of memory on the GPGPU (known as 'local memory', analogous with cache memory on the CPU). This severely limits the size of CA grids which they can simulate mainly due to the way that synchronisation works differently on a GPGPU with current limits at 1024 threads/cells. They show that this local memory can allow for a huge processing speed increases where they show a CA simulation (without any evolution) for 50,000 generations/fitness evaluation has a speed up of 489.75 times on one machine and 621.68 on another [27]. These speed-ups are exceptional and are at the high-end of the findings here. One possible source of disparity between their results and those shown here is the use of local memory, although experiments by the authors of this paper, tailored towards these hardware specific parameters [32] explains in greater detail how this local memory may benefit some machines, and the limitations of using the GPGPUs specific memory types. This work showed that local memory is indeed faster in all machines than the GPGPUs main (global) memory, but due to limitation on the number of threads/cells that speed-up factors of 2.5x-5x are obtained with these local memory implementations. However by using the global memory to allow for synchronisation of much larger grids, greater speed-ups of up to nearly 50x are obtained. The final significant finding of this study is they show that the workgroup (OpenCL) or block size (CUDA), is vitally important to the speed up of GPGPU processing, and should be selected from the small spectrum of possible sizes though empirical testing. In the work below, this limiting factor is investigated along with the effects of these models on the GPGPU. Lastly Brodtkorb et al. [31] perform a good review of current trends in GPGPU computing, and say "reporting a speedup of hundreds of times or more holds no scientific value without further explanation supported by detailed benchmarks and profiling results." [31]. Thusly the work below seeks to understand the relations between standard CA parameters and GPU speed-up and efforts are made to explain these effects.

Collectively, the literature demonstrates that there is considerable interest in the use of multi-core and GPU computing to parallelise cellular automata for specific applications. Papers such as [27] have investigated a more general-purpose approach to the parallelisation of the technique but this experimentation is conducted with a hybrid 1D CA/EA algorithm and with variety of vectors but single lattice size. The research shown in the remainder of this paper is designed to demonstrate the ability of multi-core CPUs and many-core GPUs to speed-up the performance of CA under a wide variety of conditions that might be found in the application of CA to problems in the real-world. CA parameters such as lattice size, neighbourhood size, number of states and iterations (generations) are all considered for implementation on a GPGPU. As part of this investigation, it is demonstrated how a key element for the full utilisation of the GPGPUs many-core architecture, is the balance between decreasing the memory requests latency through using appropriate memory models, and increasing the computational workload to harness the hardware processing parallelism. To the best of the authors' knowledge, this is the first comprehensive set of trials for cellular automata simulation parallelisation on GPGPU hardware.

3. Method

3.1. Rule sets

In the majority of tests the well-studied 'game of life' rule set is used, which has 2 states and a Moore neighbourhood. This is often instantiated by means of some form of look up from the possible previous states of a cell and its neighbours, mapped to the corresponding next state of the main cell (current cell under evaluation). However to allow for more complex systems with variable number of states and neighbourhood sizes, a programmatic function is used here (a series of C if-statements) which forms the basis of a decision tree. The basic definition of the game of life states there are two states known as dead (zero), and alive (one), and that if a cell is currently dead and has 3 live neighbours then it becomes alive, and if it is currently alive and has 3 or 2 live neighbours then it remains alive otherwise becoming dead. This is interpreted in pseudo-code as follows in section 3.2.
Since the 'game of life', is confined specifically to two states and a Moore neighbourhood, a number of new rule sets have been created based on the decision tree which demonstrates the compactness and simulation variety possible. Also this enables the testing of the effect of variable numbers of states and neighbourhood sizes without excessively large look-up tables. Two of the most interesting rule sets which demonstrate the relationship between activity and speed increases are shown in rule sets MSGOL (Multi-State Game Of Life) and MSGOL4 (Multi-State Game Of Life version 4) in

section 3.3., and 3.4., respectively. In order to test the effects of different neighbourhood sizes, the adaptive nature of such decision tree based rule sets is demonstrated further, by using the same ‘game of life’ rule set (section 3.2.) and an extended Moore neighbourhood where the size of the radius of the neighbourhood is defined by a user-specified parameter.

3.2. Pseudo code for the game of life rule set function

Below is the code interpretation of the game of life rule set. Where ‘mainCell’ variable contains the current value (state) of the central main cell, and should finish with the next state of the main cell. The ‘NH_Count’ variable holds the number of live cells in the neighbourhood excluding the main cell.

```
if(mainCell == ALIVE)
      if(!(NH_Count == 3 OR NH_Count == 2))
            mainCell = DEAD;
else
      if(NH_Count == 3)
            mainCell = ALIVE;
```

3.3. Pseudo code for Multi-State Game of Life (MSGOL) rule set function

This rule set implements a simple multistate conversion of the Game of Life where the state represents the energy of the organism rather than a simple alive/dead delineation. The rules are constructed so that the rule that would usually lead to a cell becoming alive, increases energy and the rule leading to death decreases it. Death only occurs when the cell reaches the lowest possible state.

```
if(NH_Count == 3)
      if(mainCell != states-1)
            ++mainCell;
else
      if(NH_Count != 2)
            if(mainCell != DEAD)
                  --mainCell;
```

3.4. Pseudo code for Multi-State Game of Life (MSGOL4) rule set function

MSGOL4 is a modified version of MSGOL and has a more stringent requirement for life in addition to a distinction between having too many or too few neighbours which results in immediate death or loss of a single energy level respectively. Also once a cell is at its maximum energy level another increase will cause immediate death , except when there are only two states, which maintains the rule sets ability to mimic the ‘Game of life’. This rule set has been developed to gain a better understanding of the effect of activity levels on potential GPU speed-up.

```
if(NH_Count == 3)
      if(mainCell != states-1)
            ++mainCell;
      else
            if(mainCell != 1)
                  mainCell = DEAD;
else
      if(NH_Count != 2)
            if(NH_Count < 2)
                  if(mainCell != DEAD)
                        --mainCell;
            if(NH_Count > 3)
                  mainCell = DEAD;
```

4. Experimental Set up

The C/C++ language and MSVC 2010 SDK compiler were used, and an application profiler was used to ensure that the program did not make excessive memory allocations, which were found to cause large slowdowns in processing in the CPU implementation that could therefore skew comparisons. The '/O2' level of compiler optimisation was also used. The state value of each cell was stored in a single byte (C style 'char' or 'unsigned char'), and used a single array to store the lattice. The experiments below are limited to square grids, and only use a static border condition (of dead cells). Although other border conditions exist, such as wrap-around or reflect inwards, these would require slightly more work at each generation. It was determined that the best way to deal with border conditions is to pad the grid with a border apron of cell values as large as the neighbourhood radius (one in the case of the classic Moore neighbourhood). A second grid (also padded with this border apron) is created and these two memory spaces are used alternately as the current grid and the new grid, for each generation. Importantly the implementation increments a counter for each live cell in each cell's neighbourhood, as opposed to adding the value of every neighbouring cell to a counter. This becomes very important in demonstrating how variable arithmetic (computational work) can cause variable speed-ups (I.e. conditional branching around the arithmetic work dependant on the automaton's current state and neighbouring states causes far greater variation in processing speed with the CPU compared that of the GPGPU).

A simple testing framework is developed where the initial grid and the parameters (e.g. grid size, number of generations) are passed into a function which processes the whole CA simulation, and the system clock is used to record how long it takes for the resulting final grid to be returned. Since it is expected to be difficult to time the processing on the GPU at intervals within the simulation fairly, the simulations are repeated for the each generational experiment. Each experiment is repeated 15 times in order to gain an average timing result.

In order to achieve parallelisation on the CPU the common shared memory model called OpenMP [42] is used, which generates worker threads at each generation for each cell in the lattice. This is achieved by simply applying #pragma labelling to the C/C++ for-loops which would otherwise sequentially process each cell of the lattice. The compiler then generates code which composes a single master thread, and at each generation launches worker threads for each cell, these are then distributed and time-sliced by the operating system between the CPU cores.

Finally the new open-standard language/API called OpenCL is used, designed specifically for parallel hardware such as the GPGPUs, and multi-core CPUs. For detailed information the reader is referred to the OpenCL specification [43]. The experiments include a small amount of compilation time for the kernel in every test (although this is from an intermediate form), and the transfer time to and from the GPGPU. Special hardware in the GPGPU time-slices these threads, and load balances between hardware sub-groups of each workgroup which need to access memory and those which need processing, and in this way can hide a lot of processing time behind memory latency. Work by Zaloudek et al. [27] shows that the workgroup size affects the processing time by affecting how the hardware time-slices threads (referred to a SIMT – single instruction multiple threads). Therefore initially 2-3 different workgroup sizes are used, to determine the fastest, and the workgroup size is set to this for the remainder of the tests; however it has been shown that each different machine may require auto-tuning in order to determine the optimum workgroup size.

A key aspect of a study such as this is the hardware used to determine the level of speed-up obtained by the algorithms. The comparison between a single core benchmark and the multi-core implementation will depend to a great degree on the hardware involved. Therefore, during testing two very different machine set ups have been used, firstly Machine A is a Dell XPS M1530 laptop, and is approximately 4-5 years old at the time of testing. Machine B is a recently constructed PC tower unit which contains a modern Core-i7 processor and latest-generation 'Fermi' Nvidia graphics card. The full specification of each machine is shown below.

Full specifications of machines used for testing. [44] [45]

Machine	Machine A	Machine B
Type	Dell XPS M1530 laptop	PC workstation
Age	4-5 years	Recent
CPU	Intel Core2 Duo T8100 @ 2.1GHz	Intel Core-I7-2600 @3.4Ghz
CPU cores	2	4 (8 with hyperThreading)
GPGPU	Nvidia GeForce 8600M	Nvidia GTX 560 Ti
GPU Processing elements (CUDA cores)	32	384
GPU Compute cores	4	8
GPU speed (Core, Shader, Memory MHz)	475, 950, 700	1645, 822, 1000/4000
GPGPU bus type	PCI-E x 16	PCI-E 2.0 x 16

Visualisation of the rule sets was achieved through a basic OpenGL interface; also the outputs from the GPGPU algorithms were found to match the CPU implementations exactly. OpenCL possesses interoperability with OpenGL which opens up the possibility to be able to accelerate visualisation as well as processing.

A key difference between the CPU and GPGPU is that the GPGPU is firstly a co-processor which has its own independent RAM memory, meaning that data must be transferred along the bottleneck of the PCI connection. Secondly the GPGPU has distinct architectural differences to the CPU, for example whereas CPU cores may run independently from each other (i.e. operate on different sections of code at the same time, by virtue of each possessing its own program counter) the modern GPGPU has a hierarchy of processing cores. OpenCL calls each core capable of operating independently a 'compute core' (Nvidia/CUDA call this a 'Streaming-MultiProcessor'); each compute core may possess one to many 'processing elements' (Nvidia/CUDA call this a CUDA core), where each processing element may run a thread in an SIMD (Single Instruction Multiple Data) fashion within each compute core. A single workgroup is only ever processed on a single compute core, which allows the GPGPU hardware further parallelism by allowing it to distribute workgroups to compute cores as it sees fit. It attempts to best use the hardware (number of compute cores) available, much in the same way that the operating system and CPU distribute threads amongst its cores.

However the modern GPGPU uses yet another level of parallelism within each workgroup and compute core, known as SIMT (Single Instruction Multiple Thread). Where a workgroup possesses more threads than processing elements with the compute core, the hardware may swap between many groups of threads with each group at different stages within the code. This allows the GPGPU to put its processing elements to best use, i.e. if one group of threads is waiting on a memory request, then another group which isn't may be used for processing. This allows the GPGPU hardware to maintain far more simultaneous thread processing compared to the CPU. The majority of this is abstracted away from the programmer, apart from the workgroup size. OpenCL and the underlying GPGPU hardware stipulate both an upper limit on the number of threads within a workgroup (size), and that the lattice of cell/threads must be a multiple of the workgroup size in each dimension.

5. Experimentation

5.1. Lattice size and workgroup tests

The wide variety of application domains for cellular automata means that a commensurate range of lattice sizes are possible. The lattice size is therefore the first variable to be investigated here.

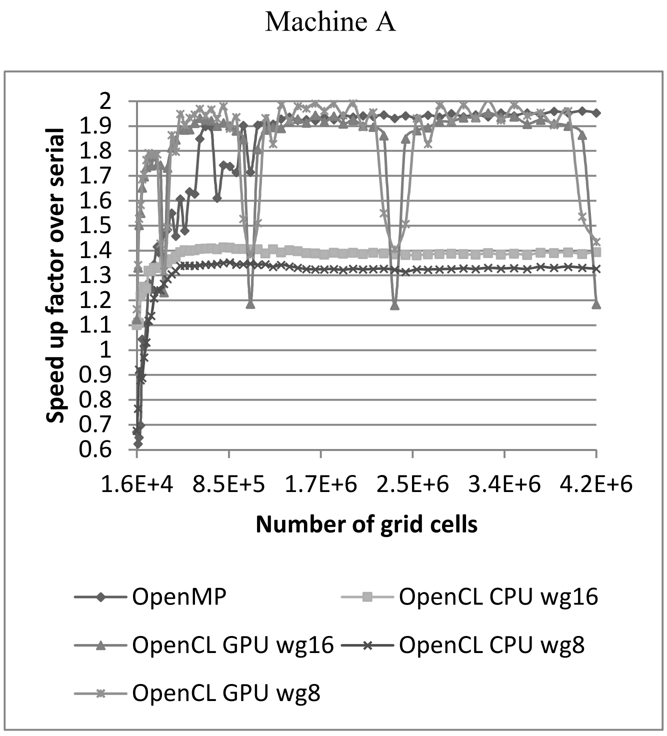
5.1.1. Method

In order to allow the processing of any size of lattice, any size which is not a multiple of the workgroup size is padded up to the nearest with threads/cells which do nothing. For these reasons two sets of lattices size tests are conducted with the first testing a wider spectrum of lattice sizes, and second testing a smaller spectrum but at much finer granularity along with testing 3 different workgroup sizes (8x8, 16x16, and 32x32).

5.1.2. Experimental set up

A random initial lattice configuration of live and dead cells is created, using a seed value and a 50% chance (initial configuration distribution probability) of each cell being made alive or dead. Tests are run for 1,000 generations on Machine A, and 10,000 generations on Machine B because Machine B is much faster making such longer runs more feasible. Lattice sizes begin at 128x128 and proceed at increments of 32x32 up to 2048x2048, also a spread of lattice sizes from 500x500 to 600x600 at increments of 1x1 are presented. Experiments are conducted at workgroups sizes of 8x8 and 16x16 on both machines, but as Machine A is limited to a maximum of 512 threads per workgroup and Machine B is limited to a maximum of 1024, a workgroup size of 32x32 could only be used on Machine B. With these experiments OpenCL is utilised on the both the CPU and GPGPU.

5.1.3. Experimental results



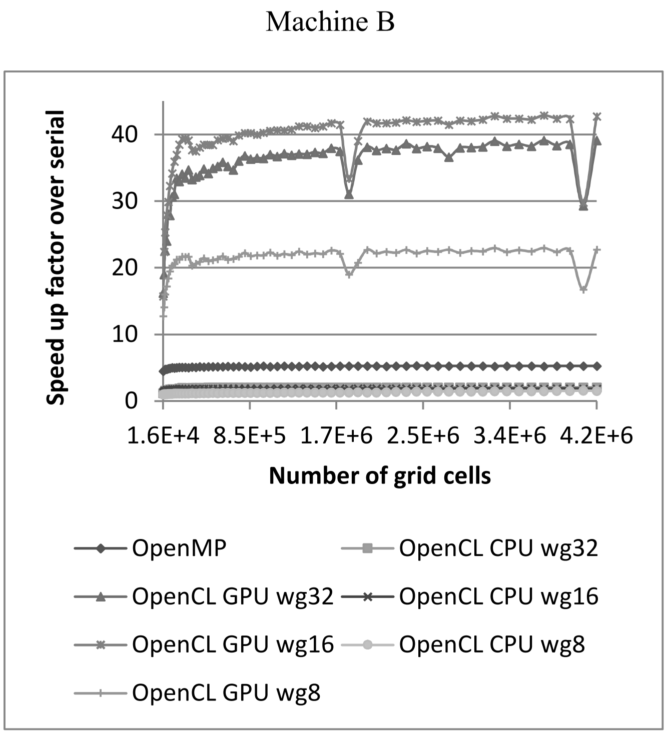
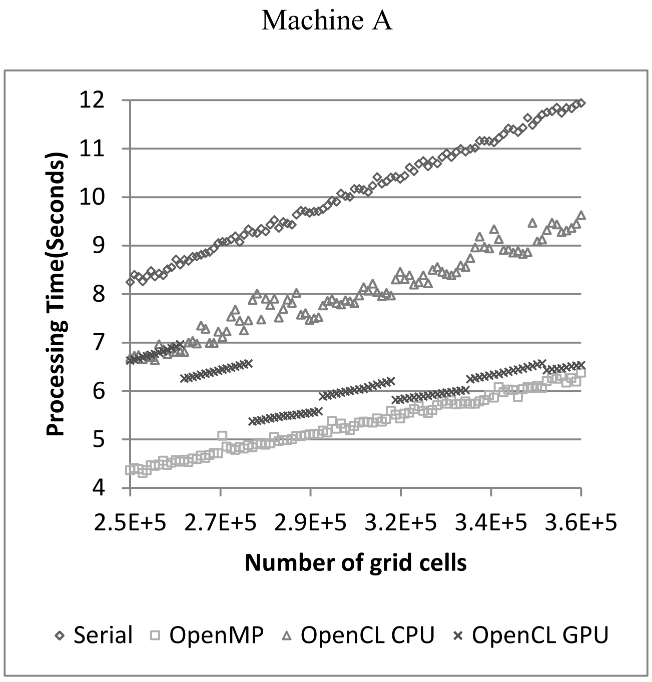


Figure 1 - speed ups over the serial implementation for OpenMP and OpenCL on the GPU and CPU, at 1,000 generations on Machine A, and 10,000 generations on Machine B, for lattice sizes of 128x128 to 2048x2048, at increments of 32x32.

Figure 1 demonstrates several key factors involved with lattice and workgroup sizes; firstly OpenCL running on the CPU is not very competitive, whereas OpenMP appears to scale fairly well to the number of CPU cores. Although OpenCL on the GPGPU performs up to 40x that of the serial CPU implementation on Machine B (the newer of the two machines), this is not in scale with the number of processing cores, although later in this paper the question of what factors lead to this level of performance increase are addressed. Secondly it is clear to see that small lattice sizes are affected by the overheads of parallelisation which on the GPGPU include the transfer and as such gain lesser performance increases than larger lattices, with this stabilising at approximately 800x800 sized lattices. A workgroup size of 16x16 appears to show the best performance increases across both machines, and is therefore utilised in the experiments in the rest of this paper. Finally it can be seen in

Figure 1 that for particular grid sizes, performance decreases abruptly. On Machine A this occurs at lattice sizes of 512x512, 1024x1024, 1536x1536, and 2048x2048; whereas on Machine B this occurs at lattice sizes of 1344x1344 and 2016x2016.



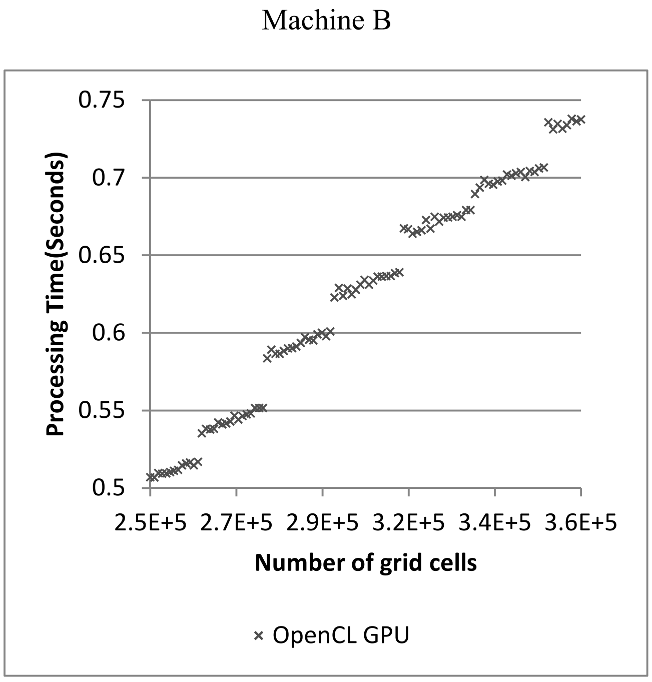


Figure 2 - processing time for OpenCL on the GPGPU, and for Machine A only on the OpenCL CPU, OpenMP and Serial implementations, for 500x500 to 600x600 lattice sizes in increments of 1x1, with a workgroup size of 16x16.

Figure 2 shows that both the CPU (sequential and parallel), and GPGPU approaches all scale linearly with the number of cells being processed, which demonstrates how the CA's complexity is based around the work in each cell. Where the GPGPU shows abrupt performance decreases (shown in Figure 1), this is associated with 'load balancing' of the workgroups to available compute cores of the GPGPU. Figure 2 indicates that the load balancing should be associated with the number of workgroups within the given lattice sizes. This is illustrated by the way that the performance on the GPGPU steps abruptly after each 16 successive grid size, where the number of workgroups changes.

5.1.4. Conclusion

Both CPU and GPGPU show linear increases in processing time with the number of cells/threads, however due to transfer and other overheads there is an offset, and due to the greater computational power of the GPGPU its processing times increase at a lesser gradient; therefore a sufficiently large grid is required in order to gain the most efficient use of GPGPU hardware and thus the greatest speed-up factor. Where this result is somewhat expected, it is informative to see the scale of this threshold number of cells/threads is relatively large compared to the number of hardware cores upon the GPGPU. In addition, there are exceptions where particular lattice sizes or rather the number of workgroups within, give lesser performance than lattice sizes with similar numbers of workgroups; this is attributed to the way GPGPU hardware distributes workgroups to be processed between available compute cores.

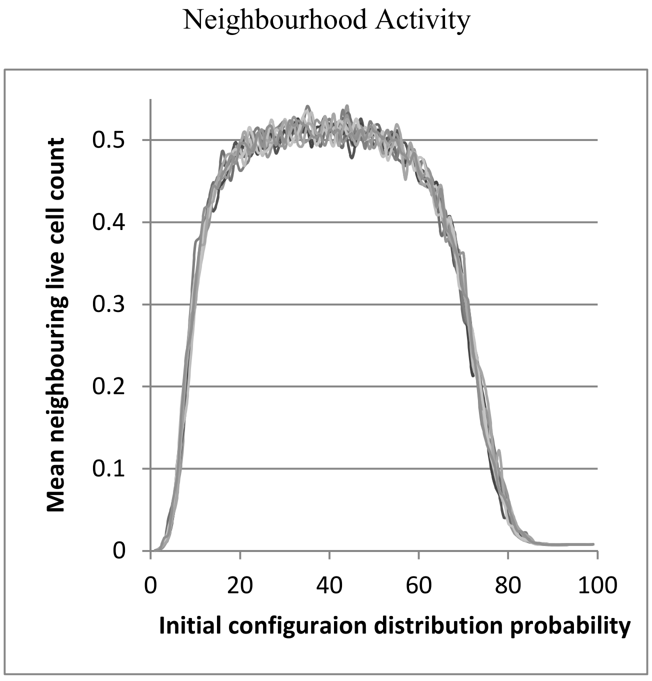
5.2. Initial configuration distribution probability and Activity tests

A further variation in cellular automata is the extent to which their formulation in terms of starting conditions and rule sets leads to activity (i.e. the number of 'alive' cells) over the life of the CA. The standard 2-state game of life from random conditions for instance is known to produce a set of short-lived static and mobile structures (e.g. gliders) and will eventually converge on a stable state that will include some oscillating structures. Clearly, the number of alive cells in the CA will change over time, and in the case of the game of life, will start high and converge to a stable minimum. The following work investigates the impact of the ratio of 'alive' cells in the initial CA and records the level of activity in the CA to determine their effect on the potential speed-up of the CA on GPU hardware.

5.2.1. Experimental set up

A separate implementation is used to count the number of live cells (those with a state of 1) and the number of live neighbouring cells for each cell to ensure that the timing results are not biased. Having counted the number of live cells and neighbours for every cell in every generation, an average proportion is calculated by dividing by the number of cells and generations. Tests are performed over a spectrum of lattice sizes, and initial configuration distribution probabilities which are used in the creation of each cell being alive or dead in the initial configuration. Ten different seed values for the random number generator are used in these experiments. The timing tests are repeated for 15 trials, but this is not necessary for the counts, as they are deterministic. Tests were conducted on a 512x512 lattice size for 1,000 generations on both machines with a workgroup of 16x16 are used.

5.2.2. Experimental results



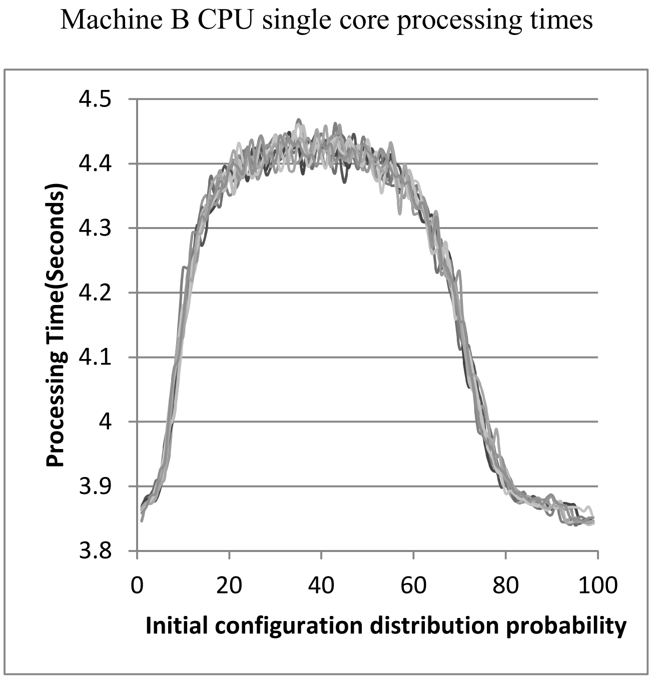
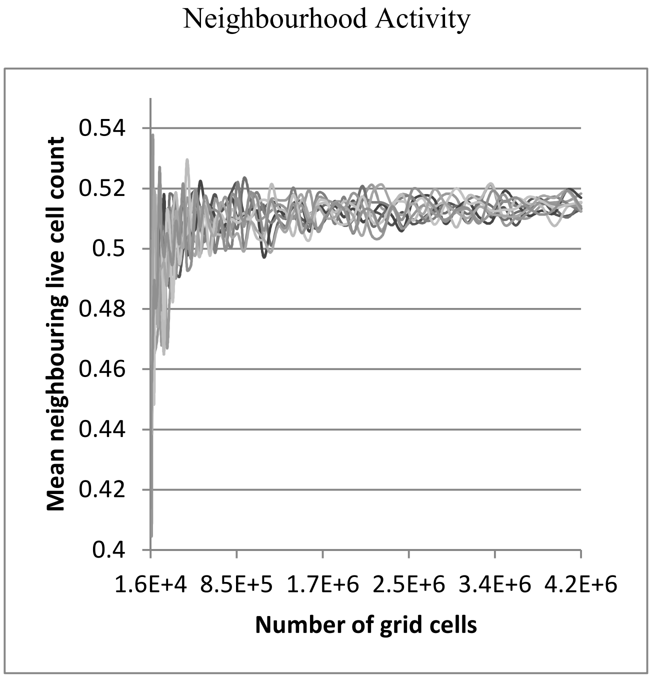


Figure 3 - average neighbourhood live cell counts per cell over the entire simulation for a range of initial configuration distribution probability/chances of live cell creation in the initial configuration (left), and the processing time on a single CPU core for the same ranges (right), processed at a lattice size of 512x512 for 1,000 generations.

Figure 3 shows firstly that as the neighbourhood counts are averaged over the entire simulation, this restricts the variation due to the difference in the underlying patterns formed through the differently seeded simulations. Therefore these averages are a measure of *Activity* over the entire simulation. Secondly figure 3 shows that there is little to no activity below initial configuration distribution probability levels of approximately 5% and above 80%, and between these, the level rises to a plateau. This plateau level is surprisingly low with an average live neighbourhood cell count of approximately 0.5 live neighbours out of a possible 8. Although with this variation from approximately 0 to 0.5 activity levels within the average neighbourhood the processing time of the serial implementation varies accordingly in a highly correlated fashion (shown in figure 3, right). Although it can also be seen in Figure 3 (right) that with no activity levels, the processing time is dominated by other work within each cell; i.e. a baseline non-varying amount of arithmetic computational work and the memory look up of the neighbourhood within each cell.



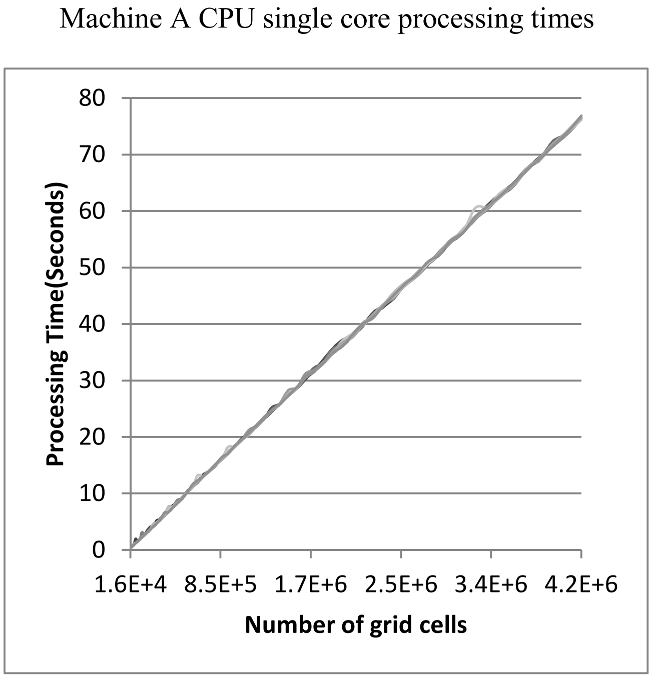


Figure 4 - Average neighbourhood live cell counts per cell over the entire simulation, when using a initial configuration distribution probability of 50% for a range of lattice sizes of 128x128 to 2048x2048, at increments of 32. (left) Note the difference in the scale of the y-axis, and the processing time on a single CPU core for the same ranges (right), for 1,000 generations.

Figure 4 shows activity variation when using the same 50% initial configuration distribution probability for different lattice sizes. This variation is present due to the difference in the underlying patterns formed, but is small and decreases as the lattice size increases. The processing times accordingly show very little variation and are thus dominated by the symmetrical work within each cell of the lattice.

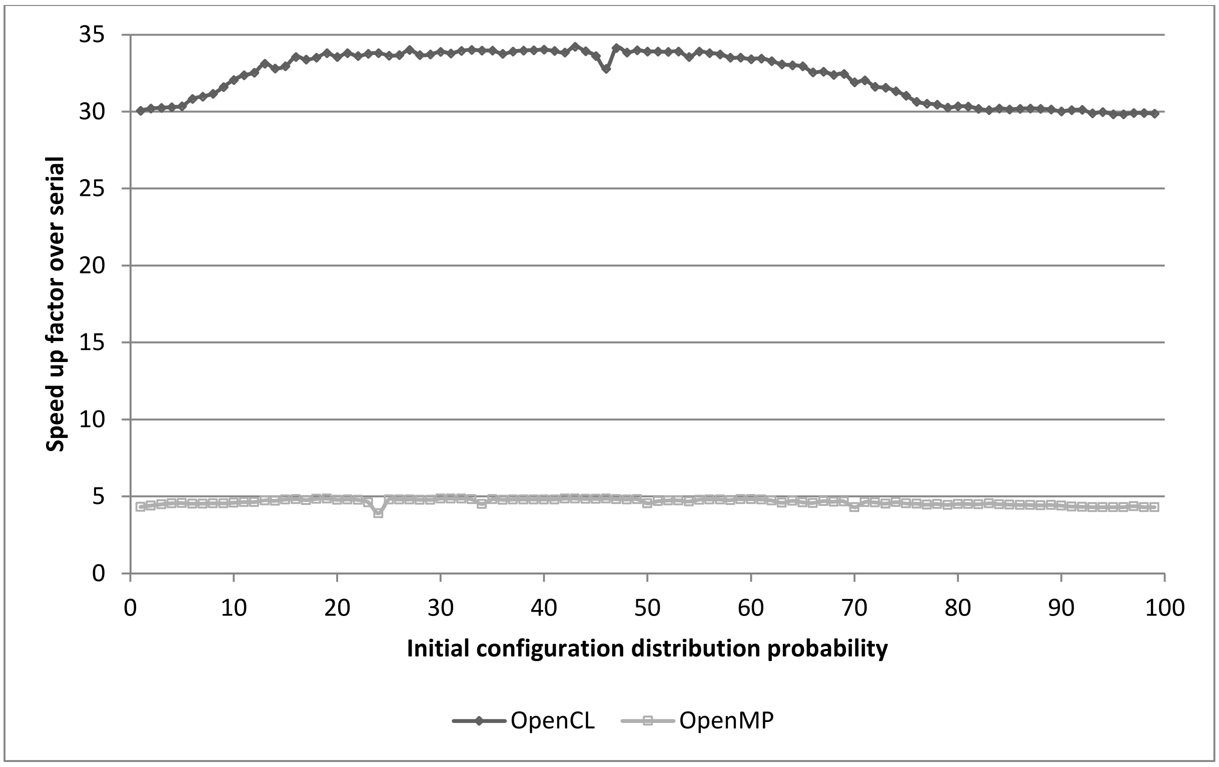


Figure 5 - speed-ups relative to the serial implementation for OpenMP, and OpenCL on the GPGPU (workgroup size of 16x16) on a 512x512 lattice size at 1,000 generations, over a range of initial configuration distribution probability values from 1% to 99% at intervals of 1%; results shown for Machine B.

Figure 5 shows that it is this increase in arithmetic from the counting of 'alive' cells, due to the correlation with the activity levels shown in Figure 3 which, when parallelised, leads to proportional increases in relative performance between the parallel approaches and the sequential approach. This effect is more greatly noticed in the GPGPU due to the greater level of hardware ALU (Arithmetic Logic Unit) parallelism. Here, results from only Machine B and are only shown for a single seed; however these are representative of results recorded from all initial configurations and clearly show that the GPU, and to a lesser extent, the parallel CPU are able to increase speed-up when activity levels are high.

5.3. Number of states tests

5.3.1. Method

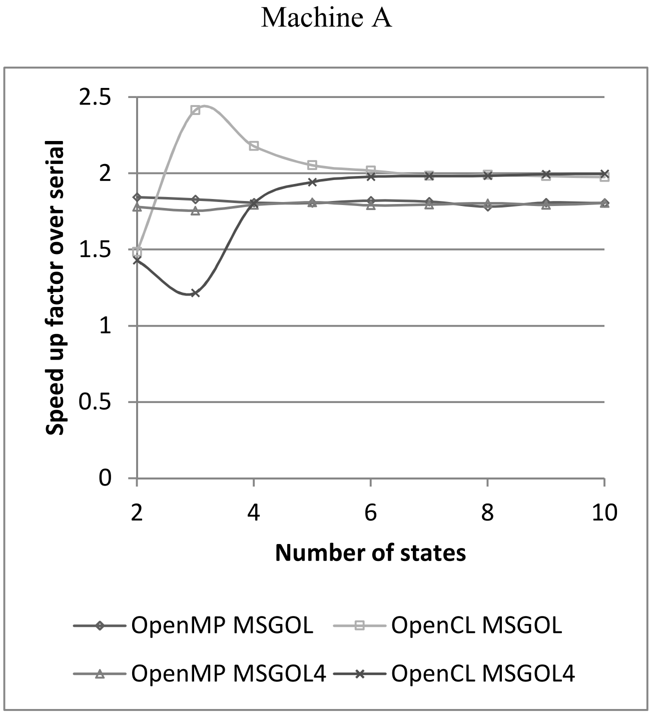
Since the game of life rule set specifically has only two states, it has been adapted here to a multi-state interpretation in order to test the effects of a variable number of states. Many such interpretations have been created and two interesting rule sets are presented which are called MSGOL (Multi-State Game

Of Life, section 3.3.) and MSGOL4 (version 4, section 3.4.). Both of these at two states reproduce the game of life exactly, but at higher numbers of states produce very different behaviours. In both rule sets, cells states are counted as dead if their value is zero, but from 1 to maxstates-1 are counted as alive. MSGOL at 3 or 4 states produces large areas of what appears as chaotic behaviour, where small snake like collections of cells are born, move around and die in between the chaos. At larger numbers of states, this forms maze-like patterns over the whole grid, with fluctuations which move over the grid as if searching for a stable global pattern. MSGOL4 at 3 and 4 states look more like the game of life, so much so that new and distinct gliders are detected at both of these numbers of states. However as the number of states is increased in MSGOL4, larger areas of what appears to be chaotic behaviour consume the simulation. In the timing tests, the lattices are populated with the same initial configuration as before. The number of states is modified by using a parameter within the decision tree rule sets. The decision tree is able to represent an increasing number of state transitions with the same decision tree because of the way it programmatically maps the relation between each state, as opposed to using an increasingly large look-up table.

5.3.2. Experimental set up

Experiments are conducted with a range of states for simulations from 2 states up to 256 as limited by the single byte data type used, at intervals of 1 state, although only experiments from 2 to 10 states are shown. In these experiments a lattice size of 512x512, with a workgroup size of 16x16 are again used (which notably is a badly load balanced size on Machine A), and run for 1,000 generations on Machine A, and 10,000 on Machine B.

5.3.3. Experimental Results



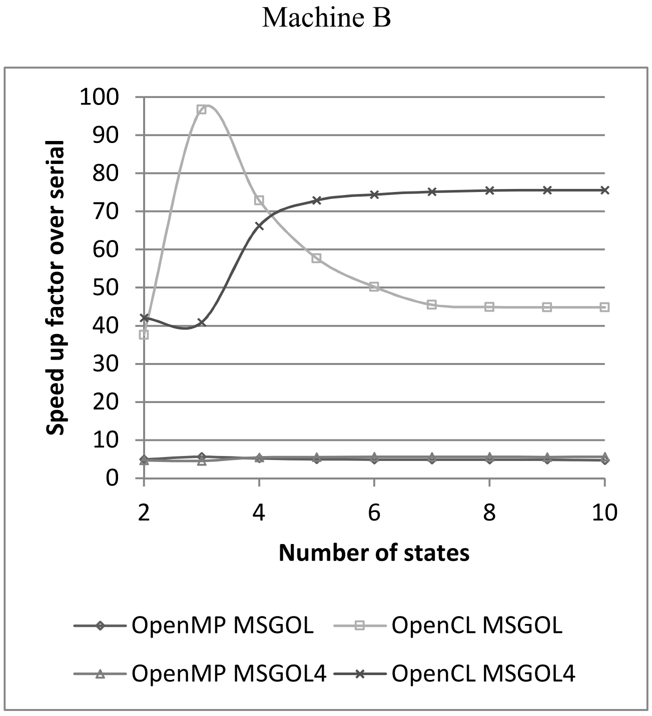


Figure 6 - speed-ups over the serial implementation for OpenMP, and OpenCL at a lattice size of 16x16, for 1,000 generations on Machine A, and 10,000 on Machine B. Showing resulting for the MSGOL and MSGOL4 rule sets with 2 to 10 states.

Figure 6 shows the relationship between the number of states (from 2-10) and the speed-up on the GPU. The graphs for the GPU and MSGOL show a peak around 3 states which then drops down to a converged rate of speed-up later for higher numbers of states. For MSGOL4, this situation is reversed with a dip in speed-up at 3 states. It is therefore clear that for the modified game of life rule sets, at least, the number of states does have an effect on the speed-up possible from GPUs but that the hardware has a very much larger effect (note the difference in axes ranges for Machine A and Machine B). However, the discrepancy between 3 states and the others was not expected and required further experimentation (shown in Appendix A). This work showed that again, it is the level of arithmetic that is conducted by the rule set that is the main driver of speed-up. The specifics of the rule set and the decision tree implementation mean that the (relatively fast on GPU) addition and subtraction operations only occur at specific leaves of the tree. Simply put, the more often these leaves are used, the greater the speed-up on the GPU. Of course, this depends on the specifics of the rule set and the decision tree implementation, but this does mean that the optimisation of the rule set to maximise arithmetic and minimise memory operations is an important element of parallelising CA with GPUs.

5.4. Neighbourhood size tests

A further parameter that varies among applications in cellular automata is the neighbourhood size. An in-depth investigation is conducted here into the effect of modifying neighbourhood size, in conjunction with activity levels to determine possible speed-up on a GPU.

5.4.1. Method

The Moore neighbourhood is extended and defined by the size of the radius (r), where the standard Moore neighbourhood has a radius of 1 as each (central, main) cell is surrounded by 1 cell in either direction, forming a square neighbourhood where the number of cells is defined as $(2r+1)^2$. The 'game of life' decision tree rule set is used (it should be noted that the GOL rule set only uses assignment for its state changes, and therefore the only variable arithmetic is within the counting of live neighbouring cells); and the collection of neighbouring live cell counts is altered to a set of two loops which takes the radius parameter, and finally each neighbouring cell is counted as it is visited, as opposed to storing the entire neighbourhood which is more difficult for the GPGPU. Since the 'game of life' rule set looks for specifically 2 or 3 live cells in order to trigger activity, as the neighbourhood size is increased, the range of possible live neighbouring cells also increases; thus the chance of finding 2 or 3 live neighbours decreases when the initial configuration is seeded with the same 50% initial configuration distribution probability in the creation of live cells. However it is found to be possible to generate long lasting patterns in all radius sizes tested from 1 to 5 for the decision tree game of life. It was consequently found that the initial configuration needed to be seeded with fewer live cells as the neighbourhood size was increased. So, similar activity tests as in section 5.2 were repeated for each neighbourhood size by using a separate implementation to ascertain the cell and neighbourhood counts over a range of initial configuration distribution probability for initial live cell creation.

It is determined that there are ranges of values within the initial configuration distribution probabilities which favour activity (shown in figures 3 and 7). Within these ranges the initial population levels are neither too few nor too many to generate widespread amounts of live cells both spatially and temporally, and as such are termed as ‘habitable spectrum’ of initial configuration distribution probabilities. Unfortunately it can be seen in Figure 7, that the ‘habitable spectrum’ for each radius of the extended Moore neighbourhood shifts dramatically towards the lower end of the initial configuration distribution probabilities, so much so that using a 50% initial configuration distribution probability with a radius greater than 2 would not likely yield high activity levels. Therefore a simple estimation of the centre of these ‘habitable spectrum’ is utilised, which also coincides with using a 50% initial configuration distribution probability as before with the tests using a neighbourhood radius of 1 (as in sections 5.1-5.3). Equation 1 shows the initial configuration distribution probabilities relative to the neighbourhood radius used in the preceding time experiments to ensure that high activity levels are generated for all neighbourhood radius sizes. Interestingly, the centres of these habitable spectrums can be approximately calculated using the golden ratio of 1.618, a ubiquitous constant in natural systems. This leads to the initial configuration distribution probabilities as shown in Table 1.

$$Initial\ Configuration\ Distribution\ Probability = \frac{1}{radius^{1.618} + 1} \qquad (1).$$

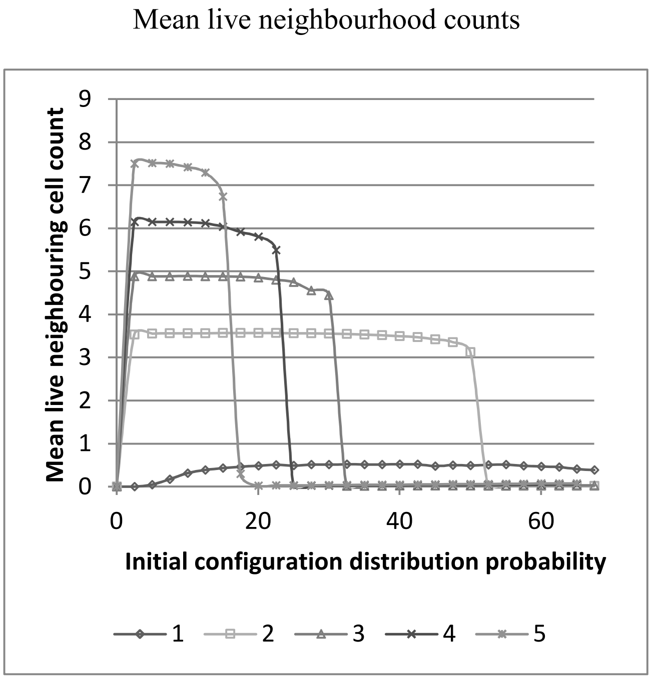
Radius	1/(radius$^{1.618}$+1)
1	50%
2	24.57%
3	14.46%
4	9.59%
5	6.89%

Table 1 – The estimations of biases for the first 5 neighbourhood radius sizes, which correspond roughly to the centre of the discovered habitable zones.

5.4.2. Experimental set up

Both the initial configuration distribution probability estimates produced by Equation 1 that yield high activity levels and a zero initial configuration distribution probability, which gives all dead cells in the initial configuration and thus the rest of the simulation, are tested. Again a workgroup size of 16x16, on a 512x512 grid is used, at 1,000 generations on Machine A, and 10,000 on Machine B.

5.4.3. Results



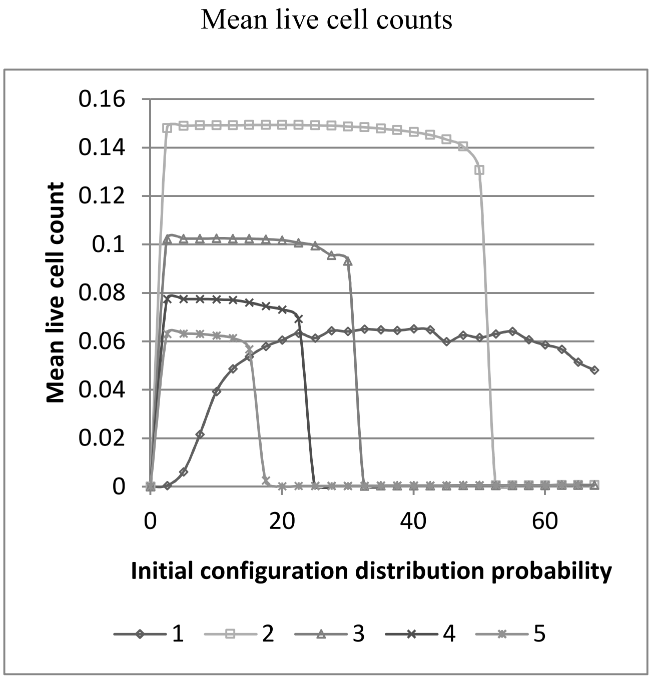


Figure 7 - Average live neighbours and live cell counts for initial configuration distribution probability of 0% to 67.5% at intervals of 2.5%, for a 512 lattice size and 1,000 generations, for the neighbour radius sizes 1 to 5.

Figure 7 shows the average amount of activity and the described plateaux or habitable spectra. Figure 7 also shows that there is a reasonably large jump in neighbourhood count activity between neighbourhood radius sizes 1 and 2, but after that appears to follow a fairly linear increase in activity as the radius of the neighbourhood is increased. Interestingly the live cell counts follow a very different pattern.

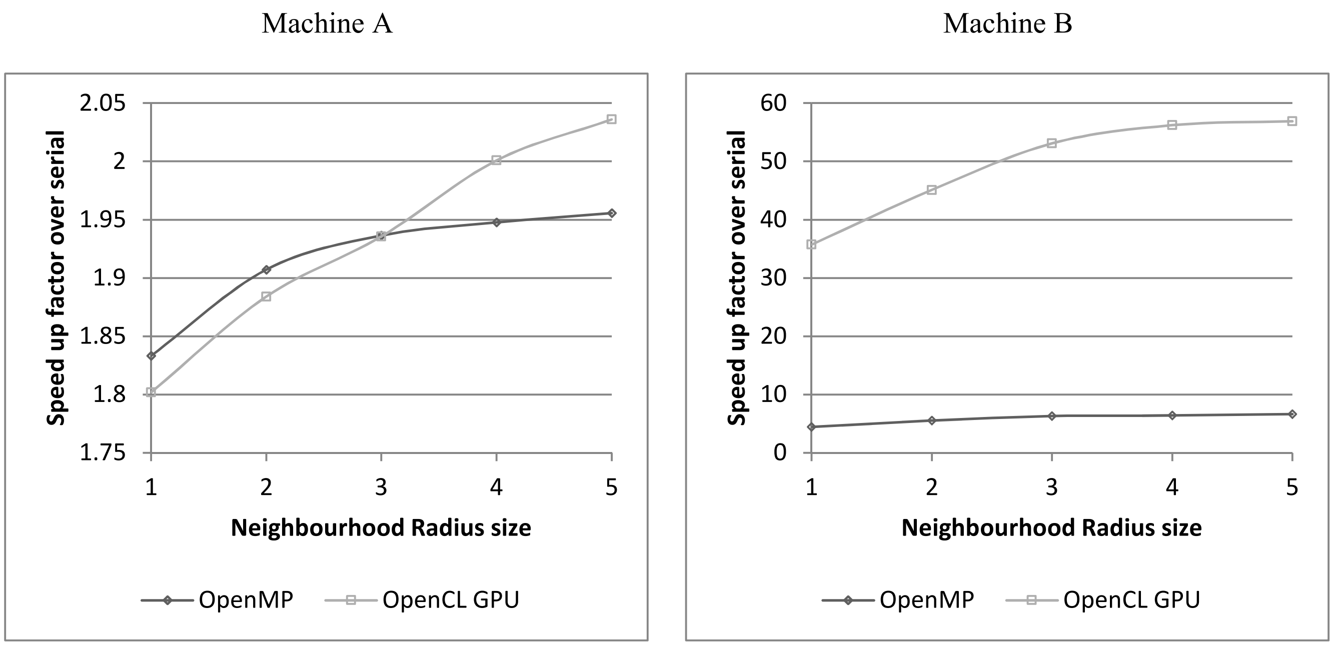


Figure 8 - The relative speed improvements of the GPGPU and the OpenMP implementation, over the serial implementation with a variable neighbourhood size. Results shown for a 512x512 sized lattice, for 1,000 generations on Machine A, and 10,000 generations on machine B. Seeding with a zero initial configuration distribution probability and therefore no activity.

Figure 8 shows that where there is no activity, only a slight increase in performance for larger neighbourhood sizes is observed, with the exception of the GPGPU on Machine B which shows larger increases. This is attributed to a more efficient use of the cache of the GPGPU in Machine B allowing for a greater use of the extra hardware parallelism with the additional memory work in each cell.

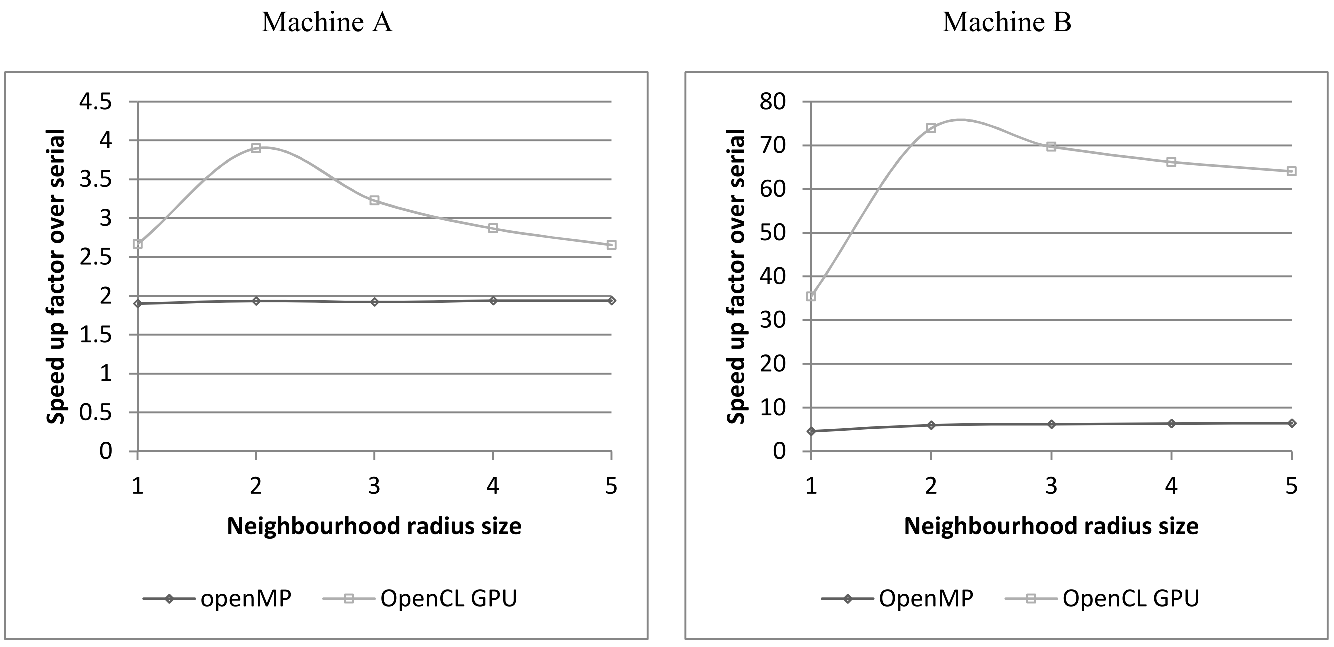


Figure 9 - The relative speed improvements of the GPGPU and the OpenMP implementation, over the serial implementation with a variable neighbourhood size. Results shown for a 512x512 sized lattice, for 1,000 generations on machine A, and 10,000 generations on machine B. Seeding with the 'initial configuration distribution probability relative to the radius' as shown in Equation 1, to produce activity in all simulations.

Figure 9 shows that where there is activity, a very different pattern in the speed ups of the GPGPU compared to the CPU (parallel) approach can be observed, whereby there is a spike in the performance at a radius of 2. It is proposed that not only is there a link between the amount of arithmetic and speed up, but there is there is also a relation between the proportions of arithmetic to memory accesses, shown in equation 2. I.e. as the number of memory accesses is increased, caused by the larger neighbourhoods, this proportion is reduced along with the speed ups.

$$Speedup\ Factor \propto \frac{Arithmetic\ per\ cell}{Memory\ requests\ per\ cell} \qquad (2).$$

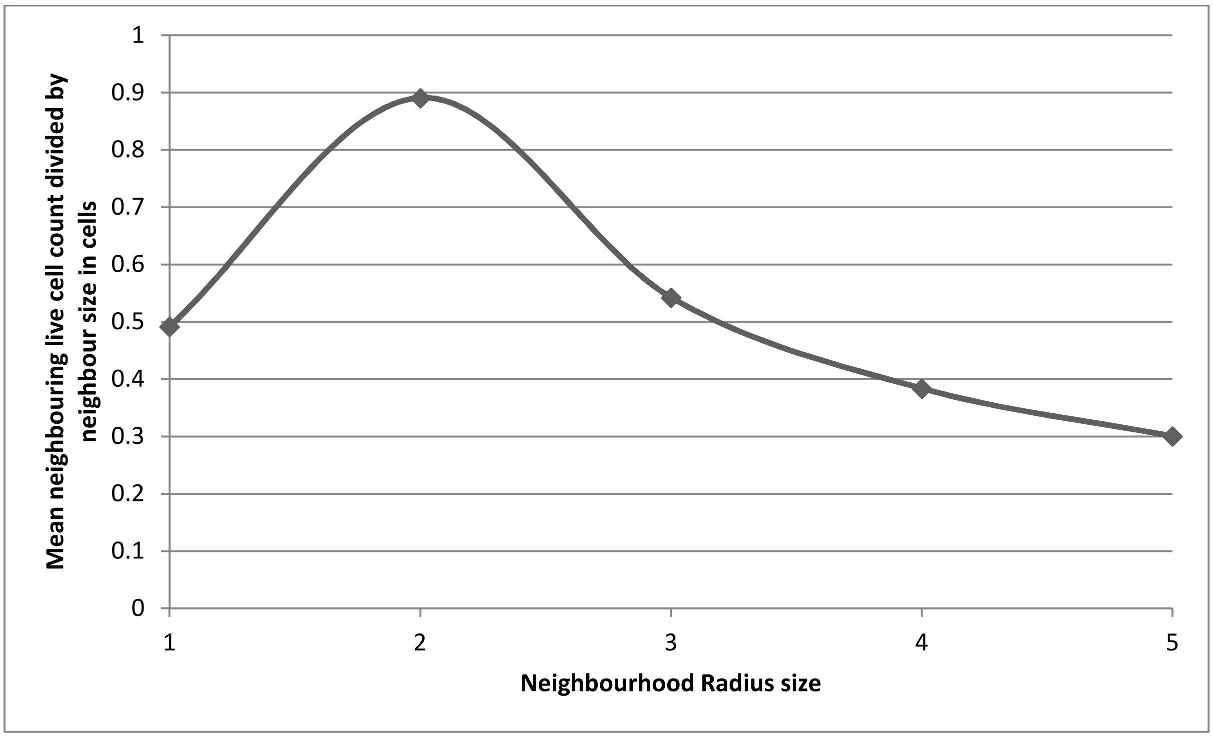


Figure 10 - The ratio of the average live neighbouring cell count (activity) for each radius, against the neighbourhood size in cells, for a 512 sized lattice and 1,000 generations (I.e. the predicted speed-up level from Equation 2).

This relation (Equation 2.) is shown in Figure 10, where the average live neighbouring cell counts from each radius divided by the number of cells in each neighbourhood are plotted. This measure of the level of activity within the neighbourhood relative to the neighbourhood size, for each neighbourhood radius clearly mimics the shape of the GPGPU speedup curves for both machines in Figure 9. For machine B, there is a slight trend to increase in performance with larger radius sizes, which is attributed to the increase in speedup seen in Figure 8 where the performance increases with neighbourhood size irrespective of activity level.

5.5. Generational size tests

Clearly, longer CA runs will benefit more greatly from any speedup that the GPU can provide. However, there are overheads associated with the implementation of a CA on the GPU and so these experiments attempt to characterise the length of run under which speedup on the GPU will be maximised.

5.5.1. Method

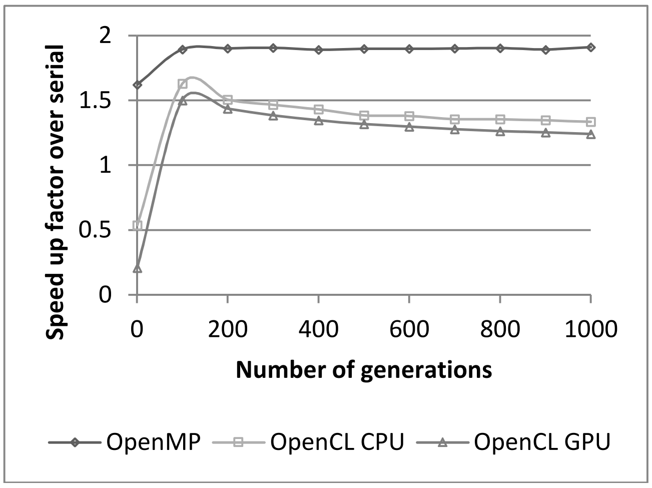
In order to avoid excessive transfer to and from the GPGPU when testing over a range of CA generations, for each number of generations tested, a full simulation is repeated up to the required number of generations, as opposed to running a single long simulation and timing it at sample intervals. Earlier sections (5.2-5.4) have shown how activity affects the entire simulation; experiments are now conducted to see how this effect correlates with the number of CA generations by again counting the live average cell and neighbourhood activity as well as a separate implementation purely for timing results.

5.5.2. Experimental set up

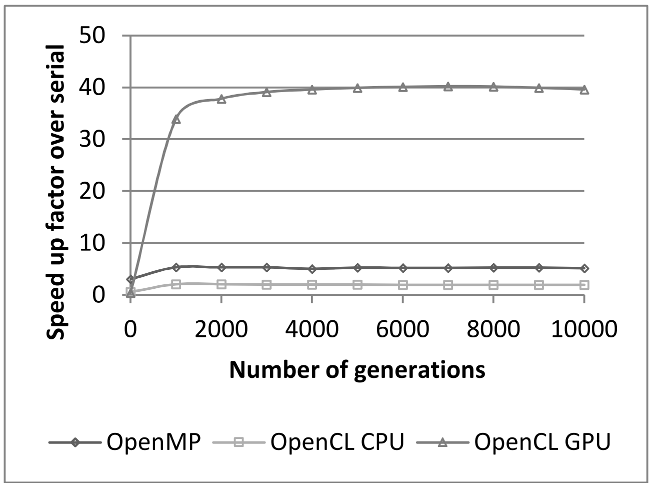
In a similar fashion to the lattice size tests in section 5.1, tests were run on a spectrum of generation sizes and fully independent runs (repeated 15 times for an average) were conducted for each total count of generations. Tests were run at a single generation, and then at increments of 100 on Machine A up to 1,000, and at increments of 1,000 on Machine B up to 10,000. These tests are performed at lattice sizes of 512x512, 1024x1024, and 2048x2048, which are noted to be the particular lattice sizes where Machine A suffers from load balancing issues on the GPGPU, with a workgroup size of 16x16. Tests are also repeated on Machine B, at smaller lattice sizes of 480 and 448, in order to confirm the theory for the difference in the Machine B's response at a 512 lattice size. Machine B's GPGPU has more cores and therefore processes each generation quicker than Machine A. However the overheads of parallelisation do not change as greatly for Machine B and this therefore means it takes more generations to overcome them at these smaller sized lattices.

5.5.3. Results

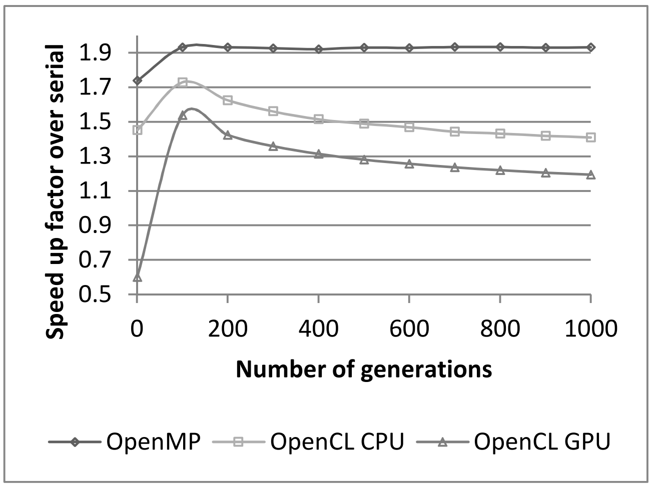
512 lattice – Machine A



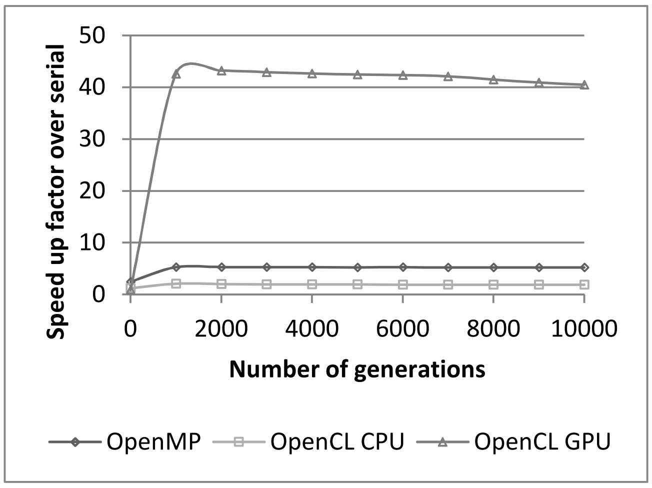
512 lattice – Machine B



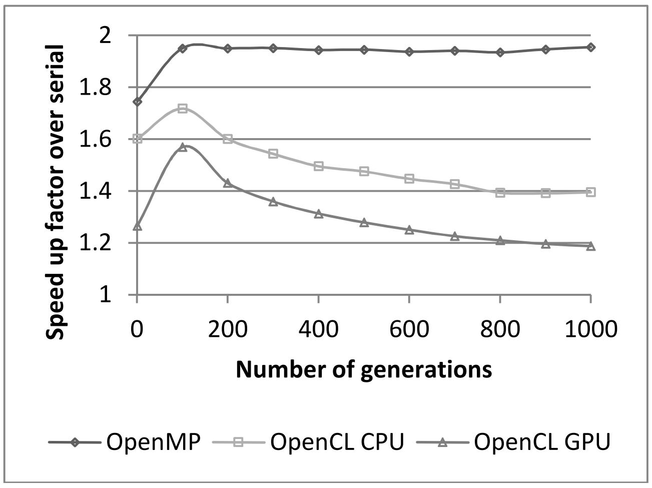
1024 lattice – Machine A



1024 lattice – Machine B



2048 lattice – Machine A



2048 lattice – Machine B

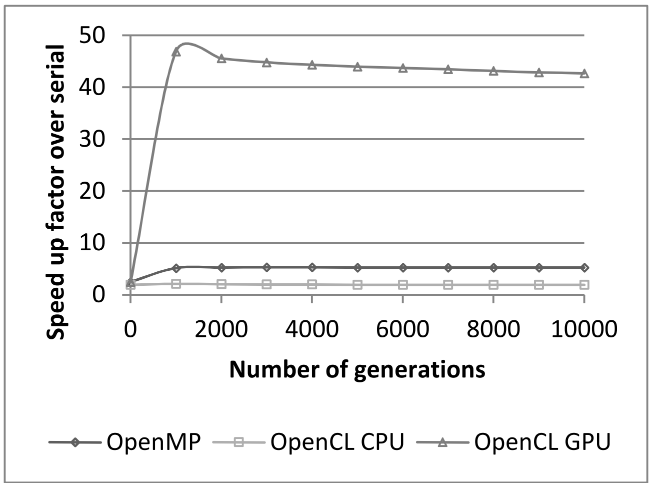


Figure 11 - speed up factor over the serial implementation, for OpenMP and OpenCL on the CPU and GPU, for a spread of generations, at lattice sizes of 512x512, 1024x1024, and 2048x2048.

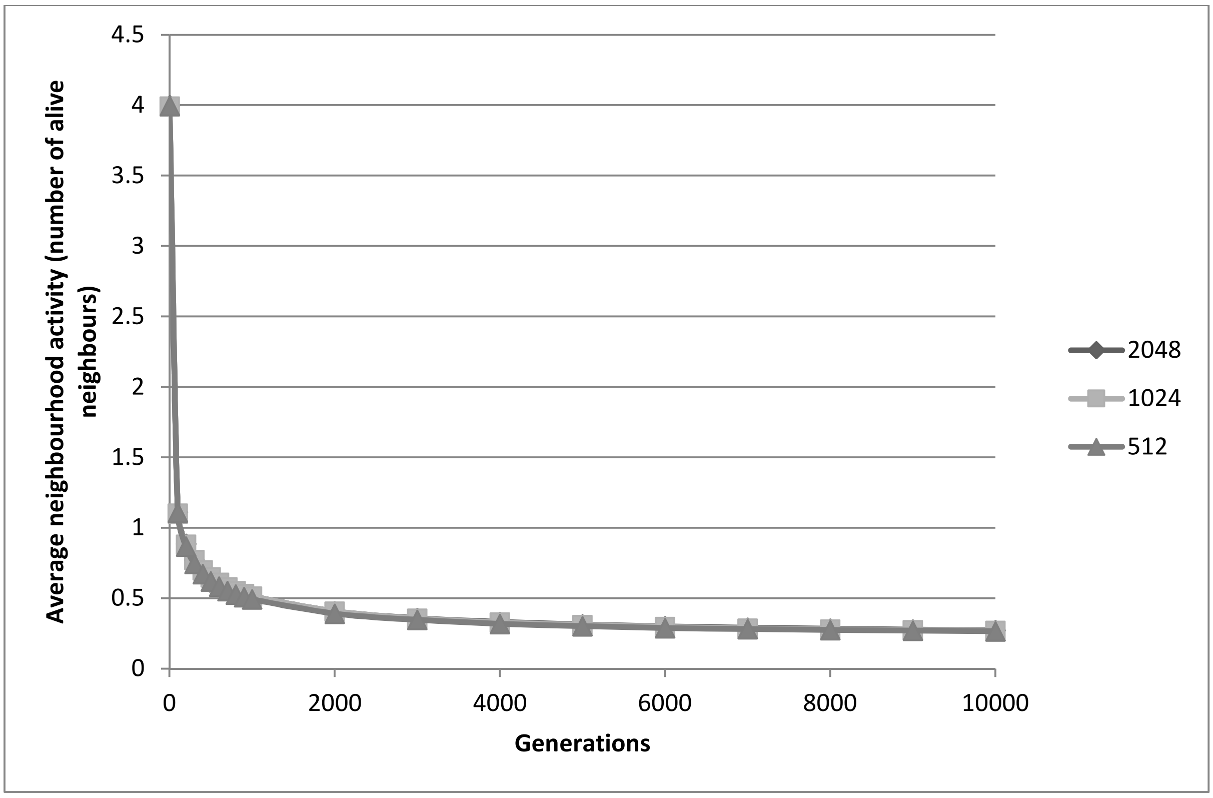


Figure 12 –Average neighbourhood counts (Mean number of live neighbouring cells), for a spread of generations, at lattice sizes of 512x512, 1024x1024, and 2048x2048.

It was found that once again, the timing results indicate a fairly linear relation between the number of generations and the processing time. However overheads of parallelisation which, on the GPGPU, include the transfer time to and from the device and small amount of compilation time for the kernel, mean that a curvature up to a plateau can be observed in the relative performance shown in Figure 11. In many of the relative performances of the parallel approaches, as the number of generations increases the parallelisation overheads are overcome as a larger proportion of the whole algorithm is parallelised. However it appears that, if the number of generations is increased further, a slower drop in relative performance of the parallel approaches can be observed. Sections 5.2-5.3 have shown that over the course of the entire simulation the activity is directly proportional to the relative performance of the parallel approaches and can be partially attributed to the slower drop in activity, in the later stages of the simulations (Shown in Figure 12). However experimental results appear to show a greater drop at the higher end of the tested spectrum of generations, especially on the older/slower machine A, which may be due to the continued bottleneck of load balancing, which occurs at that grid size. After a single generation the highest amount of activity are observed in Figure 12, and in Figure 11 the lowest relative performance (even a marginal decrease in some cases) is observed. This demonstrates the extent to which simulations with very few generations are dominated by these parallelisation overheads, specifically for the GPGPU the transfer time bottleneck. Figure 11, for Machine B at a 512 lattice size shows that it takes more generations to overcome the initial bottleneck. This pattern is due to the use of relatively small lattice in relation to the level of hardware parallelism and the transfer bottleneck; i.e. as Machine B has a greater level of hardware parallelism compared to Machine A, it can process each generation of the smaller lattice relatively faster compared to the transfer bottleneck, and therefore takes more generations to overcome this effect.

6. Discussion

The experimentation here has shown that in order to gain the greatest performance from the GPGPU the lattice size, or rather the number of threads used, must exceed the number of hardware cores by more than an order of magnitude in order to best utilise all layers of hardware parallelism. Particular difficult lattice sizes are shown to be caused by the lack of fit between the numbers of workgroup threads and the number of hardware's independent sub-groups of processors. Therefore caution is suggested when using particular lattice sizes, and claiming acceleration rates for the GPGPU. It is also shown that these load balancing effects are caused by the number of workgroups, and one solution to this problem might be to pad the number of threads and therefore workgroups. An advantage to this approach is that these extra threads need only check if they are within the actual lattice or of the padded threads area, and thus need not transfer additional data.

It is noted that part of these findings are due to many of the intricacies of way the implementation has been formulated, even though only a brute force approach is investigated. For example, a programmatic function/decision tree is used, which doesn't require a look-up. Also, apart from the final tests with neighbourhood sizes, unrolled loops have been used for the collection of neighbouring cell values. Also, in the 'game of life' with two states, it would be possible to implement this by means of summing the values of all neighbouring cells in order to gain a count of living cells, although this would be sub optimal for serial implementations, it would yield a very high speed up factor. The experimentation shows how important variable amounts of arithmetic are to the speed improvement shown by the GPGPU, and how this causes variations in GPGPU accelerations over the CPU. It is demonstrated through the implementation of state transition rules with variable amounts of arithmetic, that where the conditions of the CA simulation result in large variations in the arithmetic levels across the grid then the GPGPU's performance varies proportionately. This may go some way to explain the wide variation of reported speed increases from the literature. All CA will need to follow a similar memory pattern, as all CA need each cell to access the neighbouring cells states; therefore any variation in speed-up between different CA will come from both the different neighbourhood size, in addition to the amount of certain arithmetic and variable arithmetic within the given state-transition rule of the CA. It is shown for the investigated rule sets that the majority of processing time can be attributed to the number of memory look-ups for each cell however the variation in the arithmetic work of rule set can still cause relatively large difference in speed-up. Therefore, for a given size of neighbourhood used (the amount of memory requests per cell), one should if possible increase the amount of arithmetic within the rule set on a per cell basis, in order to gain greatest speed-ups. Conversely if the rule set uses minimal arithmetic compared to memory access, it is expected that speed-ups will be smaller.

Multi-state interpretations of the game of life have been created and experimental results are shown. The variation in the number of live cells over the course of the simulation, and the average specific course through the decision tree are shown to affect the processing speed of the CPU implementation due mainly to the varied amount of arithmetic necessary. Whereas the GPGPU may hide this variation in arithmetic complexity behind the memory latency and within the hardware parallelism, and as a result produce much more predictable processing times in the presence of such variation. The variation is a consequence of the behaviour produced, and therefore very hard, if not impossible, to predict without prior knowledge of the given rule set. Activity levels effect the processing time by different amounts at each generation and, with the game of life rule set, the activity was found the be greatest in the early stages; after an initially large drop tends to a slow decrease. In these early generations/short simulations the bottleneck of transferring to and back from the GPGPU for processing is found to by far outweigh the acceleration in processing. Therefore suggesting that this bottleneck is especially critical to short lived simulation, which would certainly have consequences for distributed/cluster systems; i.e. it is even more important to run long CA simulations (large number of generations) than it is to use larger lattices, in order to overcome the overheads of parallelisation upon co-processors such as the GPGPU.

The cache in the newer Fermi generation of GPGPUs facilitates better use of larger neighbourhoods. This is because the additional hardware parallelism makes good automatic uses of this fast memory. A relation between the proportions of arithmetic activity to the amount of memory re-use is proposed (shown in equation 2), again suggesting the need for a fine balancing act between the amount of arithmetic and memory complexity within a CA system for best performance increases. The decision tree interpretation of the game of life rule set when applied to larger neighbourhoods is found to produce very interesting patterns:- ‘habitable spectra’ within the initial configuration distribution probabilities (chance for each cell creation as ‘alive’) exist for the incitement of such patterns. A relation between the radius of the neighbourhood and these habitable spectra are proposed which appears to follow an interesting mathematical pattern found from nature, i.e. the golden ratio.

7. Conclusions

As a lattice-based, parallel method of computation, cellular automata lend themselves to parallelisation on GPUs very well. This work has thoroughly investigated the performance increases that can be expected from this parallelisation for a wide range of expected cellular automata parameters. The results have provided some expected results; that CA run for longer generations provide increasing speed-up and that the machine type (and in particular relative speeds of GPU and CPU) have a large bearing on the level of speed-up possible on these machines. The results have also provided some less obvious insights into GPU parallelisation though. Firstly, that the maximum speed-ups are found when maximising the arithmetic use of the GPU, while minimising the amount of memory look-ups per cell of the CA also increases the speed-up factor. Secondly, that the amount of activity in the CA has a large effect on performance. With a high dependency on the specific implementation, CAs with more cells that carry out lengthy processing during their evolution are more amenable to parallelisation than those which have low cellular computational complexity and are therefore able to exploit the parallel GPU more effectively. Thirdly, that the choice of lattice size is important in the speed-ups possible on the GPU and care should be taken to ensure that the lattice size fits with the underlying hardware where possible. Fourthly, that there exists a complex relationship between the number of states, neighbourhood size, state-transition rules and the level of activity (and therefore effective parallelism) within a CA. This relationship will ultimately determine the specific level of speed-up available to a CA implementation and is, for obvious reasons, problem specific. However, using the results from this study the likely speedup for a CA implementation on a GPU can be estimated based on the lattice size, activity levels observed within the CA and the number of generations required. This estimation can aid decision making when considering whether the degree of speedup is sufficient to warrant a GPGPU implementation on an application by application basis.

Finally, the work above has shown that for a simple CA such as the Game of Life, speed-ups of between 50 and 100 times are possible on modern hardware. It should be noted that this is likely to be a conservative estimate as this figure is a comparison with one of the fastest modern CPUs available and the Game of Life is comparatively simple. More complex rule sets will make better use of the GPUs native capability for performing fast parallel calculations.   The best speed-ups will occur when the CA is sufficiently complex and run for a large number of generations, which is beneficial to the field in that the most complex CA will benefit the most from parallelisation.

8. Acknowledgement

The work contained in this paper was supported by the Engineering and Physical Sciences Research Council (EPSRC),grant number EP/H015736/1.

9. Appendix A

In the majority of cases the number of states simulations take approximately the same amount of processing time irrelevant of the number of states, however in the area of the most variation in activity a large spike in performance for both rules can be observed in figures A2, A3 and A4, as they change from one type of behaviour to another. As shown in Section 5.2, it is the variable amount of arithmetic carried out in each cell which directly relates to the processing time and consequent increases in performance. Therefore it is necessary to account for the variable amount of arithmetic from the decision tree. The MSGOL and MSGOL4 rule sets both have leaf nodes which carry out a simple plus/minus-one calculation for the next state, therefore counting implementations have been created which, as well as counting the live cells and live neighbours per cell, also count the number of cells taking each leaf node of the decision tree rule sets. Shown in figure A1 is the binary decision tree represented by the MSGOL rule set (section 3.3.), with leaf nodes labelled A-E, where it is node B that carries out a one-plus to the current state, and node E carries out a one-minus to the current state in order to find the next state, and all other leaf nodes represent leaving the current state of main cell as it was in the previous generation. A similar decomposition of the MSGOL4 rule set is performed, with leaf nodes A-G, where nodes C and F are responsible for arithmetic operations. With both rules sets, as with the game of life rule, the operation time also depends on the number of live neighbours for each cell. The average live neighbourhood counts, and proportion of cells over the entire simulation for each leaf node of the decision tree for MSGOL and MSGOL4 rule sets are shown in Figure A2 and Figure A3 respectively. In Figure A4, first the timing results from the MSGOL and MSGOL4 rule sets for the parallel CPU approach on Machine A are shown; this is compared to the combination of the variable amount of arithmetic (i.e. the average live neighbour counts, plus the leaf nodes, which carry out arithmetic), in order to demonstrate how it is again the variable amount of activity, which causes the difference in processing time on the CPU. The GPGPU is found to have much smaller variations in processing time over the same area, which leads to huge computational speed up in the area of high activity, shown in figure A4 and then figure 6.

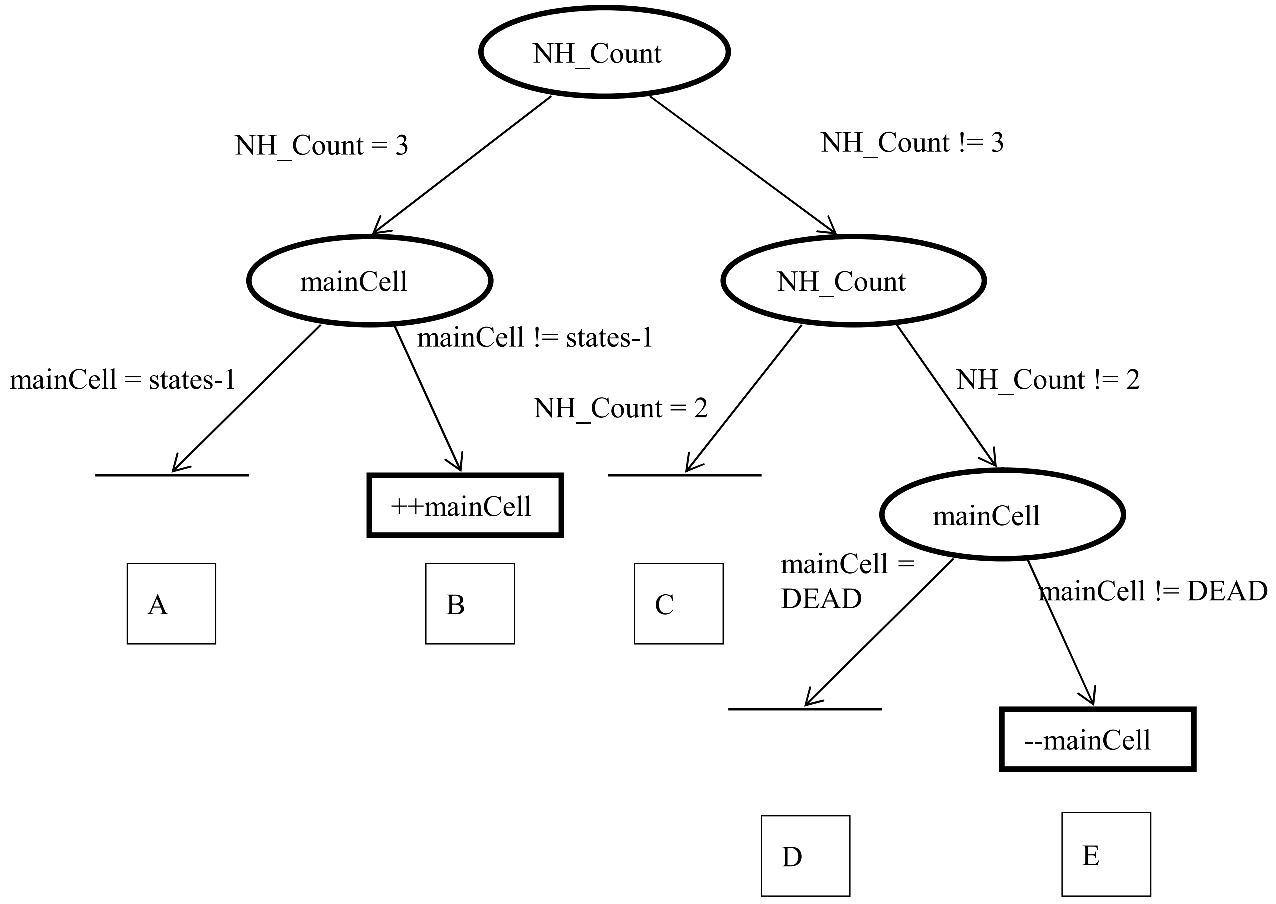


Figure A1 - the binary decision tree version of the MSGOL rule set, with leaf nodes labelled A-E. With the variables 'NH_Count' which represents the number of live neighbouring cells, and 'mainCell' to represent the central main cell's value, and finally 'states' to represent the number of states variable.

9.1. Experimental results

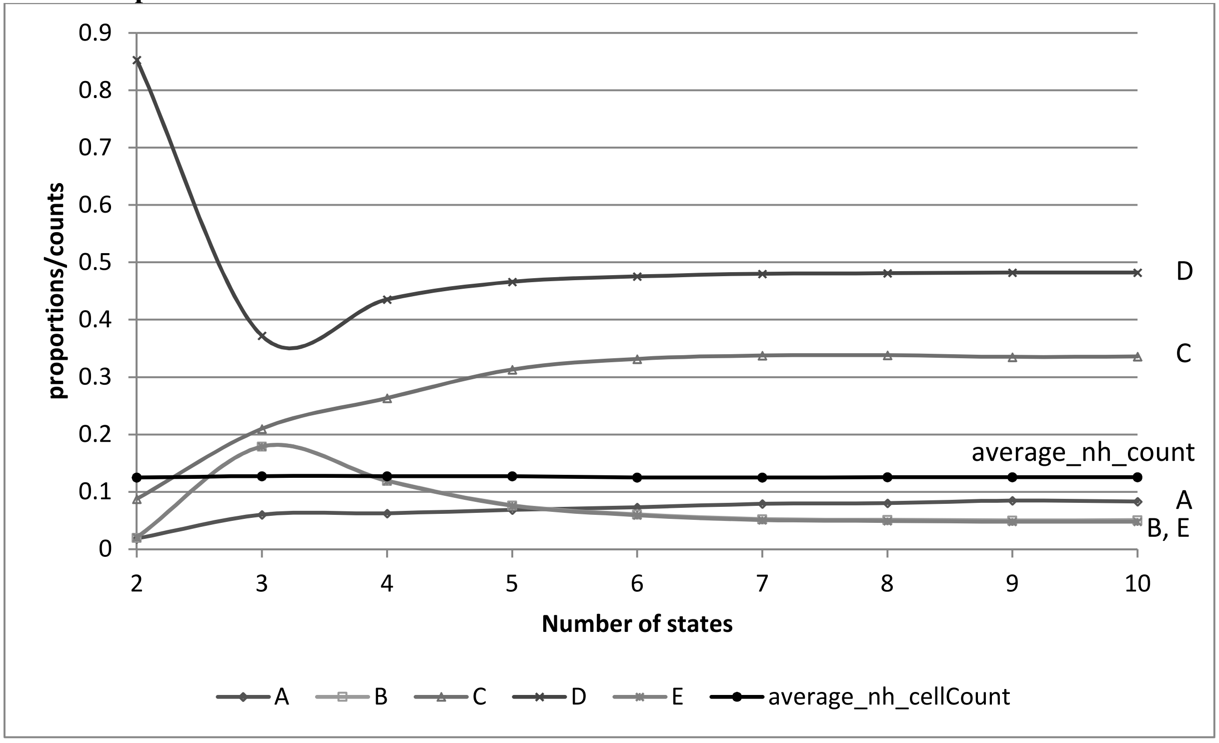


Figure A2 - The average neighbouring cell counts for each cell and the proportion of cells over the entire simulation taking each possible leaf node through the rule sets MSGOL (which has leaf nodes A-E as shown in Figure A1.

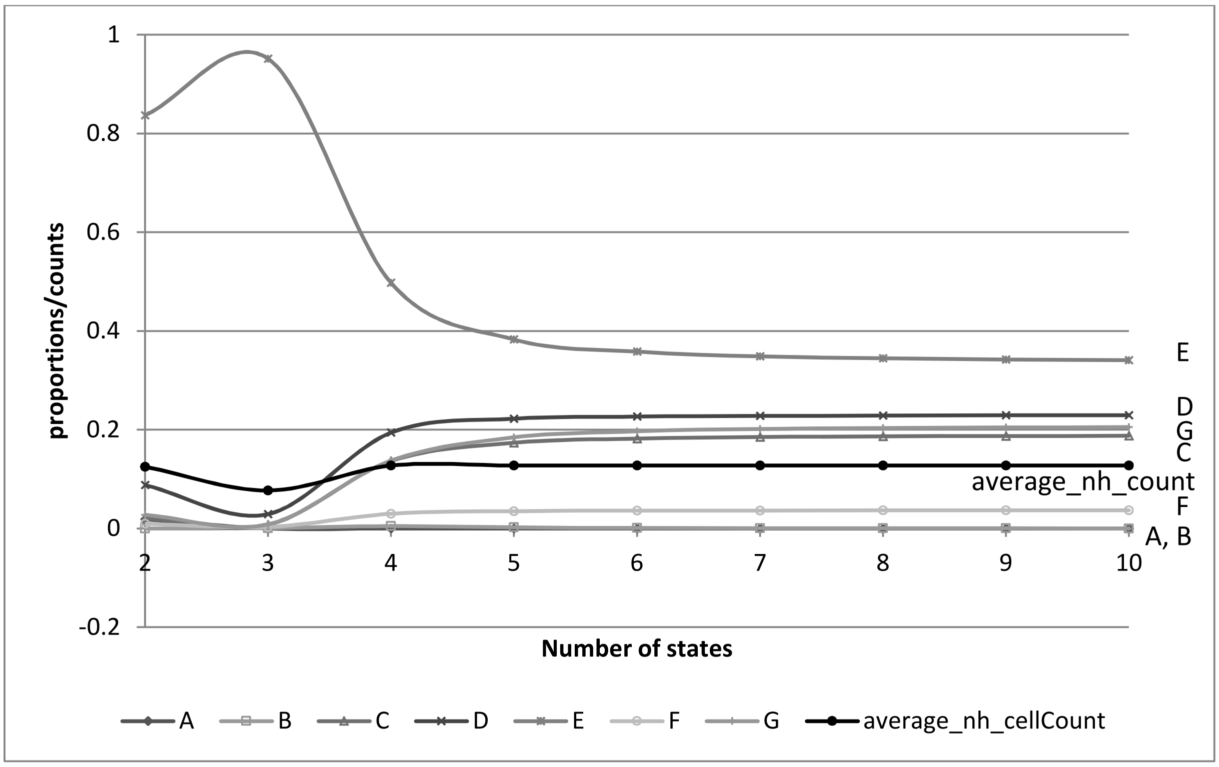
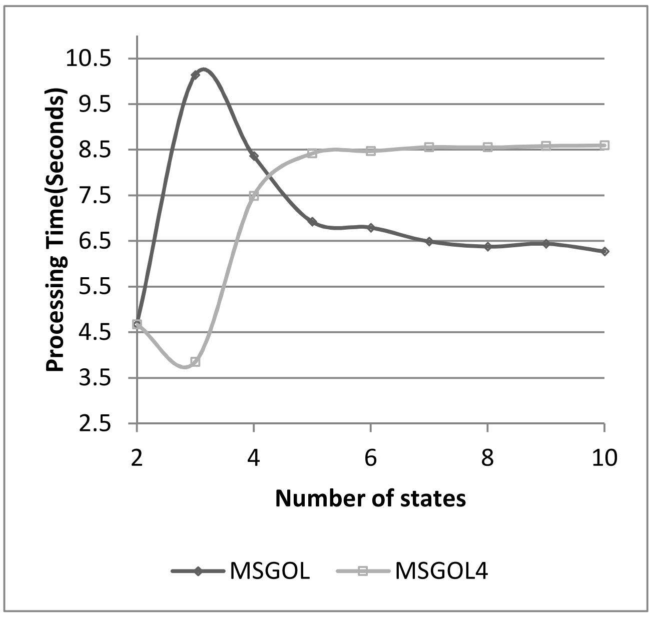


Figure A3 – The average neighbourhood live cell counts, and proportion of cells over the simulation taking each leaf node for MSGOL4 rule set, on Machine A.

Processing times Machine A



Count results

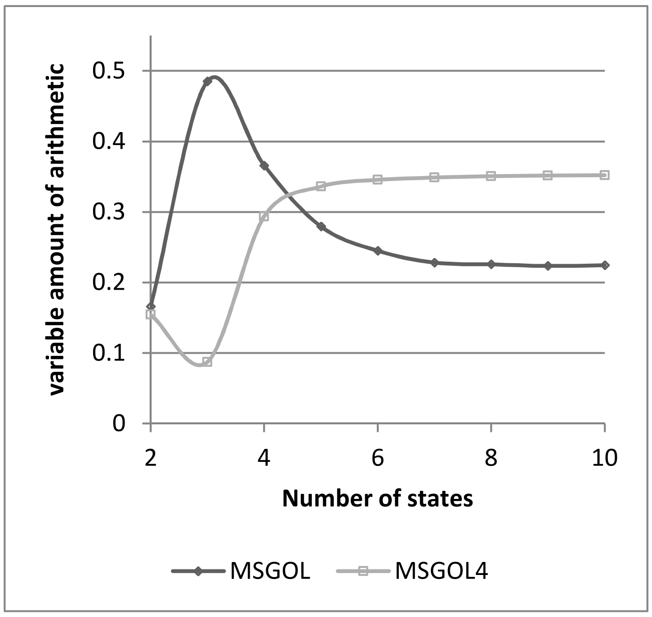


Figure A4 – processing times of the OpenMP implementations of MSGOL and MSGOL4 in comparison to each other for 2-10 states (left), shown (right) the theory of the arithmetic complexity by showing the average neighbourhood count (shows the number of increments of a counter, on average), plus the proportions of cells on average over the whole simulation which perform an arithmetic operation. In the case of MSGOL this is leaf nodes B and E, and for MSGOL4 leaf node C and F.

Figures A2 and A3 demonstrate that the rule sets generate most of the extra arithmetic complexity compared to the neighbourhood counting. Figure A4 shows how it is indeed this arithmetic complexity caused by the resulting behaviour which causes the relative slowdown in the CPU processing, and figure 6 shows how this also causes a relative speed performance increase from the GPGPU over the CPU in the same area.

10.Bibliography

[1] N. Corporation, "K40 specification," Nvidia Corporation, 2014. [Online]. Available: http://www.nvidia.com/object/tesla-servers.html. [Accessed May 2014].

[2] B. Chopard and M. Droz, Cellular Automata Modeling of Physical Systems, Cambrige University Press UK, 1998.

[3] J. Von Neumann, Theory of Self-Repoducing Automata, University of Illinois press, Urbana, 1966.

[4] S. Szkoda, Z. Koza and M. Tykierko, "Accelerating cellular automata simulations using AVX and CUDA," ARXIV, 2012.

[5] R. Denzi, S. Succi and M. Vergassola, "The Lattice Boltzmann equation: Theory and application," *Physics reports,* vol. 222, no. 3, pp. 145-197, 1992.

[6] J. A. Somers, "Direct simulation of fluid flow with cellular automata and the lattice-Boltzmann equation," *Applied Scientific Research,* vol. 51, no. 1, pp. 127-133, 1993.

[7] Z. Fan, F. Qiu, A. Kaufman and S. Yoakum-Stover, "GPU Cluster for High Performance Computing," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, 2004.

[8] M. G. B. Johnson, D. P. Playne and K. A. Hawick, "Data-Parallelism and GPUs for Lattice Gas

Fluid Simulation," Massey University, New Zealand, Auckland, 2010.

[9] A. Kolnoochenko, P. Gurikov and N. Menshutina, "General-purpose graphics processing units application for diffusion simulation using cellular automata," in *21st European Symposium on Computer Aided Process Engineering – ESCAPE 21*, 2011.

[10] M. J. Harris, G. Coombe, T. Scheuermann and A. Lastra, "Physically-Based Visual Simulation on Graphics Hardware," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, Saarbrucken, Germany, 2002.

[11] J. L. Guisado, F. Jiménez-Morales and F. Fernández De Vega, "Cellular automata and cluster computing: An application to the simulation of laser dynamics," *Advances in Complex Systems,* vol. 10, no. supp01, pp. 167-190, 2007.

[12] G. Y. Vichniac, "Simulating physics with cellular automata," *Physica D: Nonlinear Phenomena,* vol. 10, no. 1-2, pp. 96-116, 1984.

[13] J. Ning, H. Xu, B. Wu, L. Zeng, S. Li and Y. Xiong, "MCA-based Animation of Fracturing Heterogeneous Objects," in *Computer-Aided Design and Computer Graphics (CAD/Graphics), 2011 12th International Conference on*, Jinan, 2012.

[14] E. Roberts, J. E. Stone, L. Sep´ulveda, W.-M. W. Hwu and Z. Luthey-Schulten, "Long time-scale simulations of in vivo diffusion using GPU hardware," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, Rome, 2009.

[15] L. Dematte and D. Prandi, "GPU computing for systems biology," *Briefings in bioinformatics,* vol. 11, no. 3, 2010.

[16] P. Richmond, D. Walker, S. Coakley and D. Romano, "High performance cellular level agent-based simulation with FLAME for the GPU.," *Briefings in Bioinformatics,* vol. 11, no. 3, pp. 334 - 347, 2010.

[17] J. Tran, D. Jordan and D. Luebke, "New Challenges for Cellular Automata Simulation on the GPU," in *SIGGRAPH*, Los Angeles, 2004.

[18] S. Gobron, F. Devillard and B. Heit, "Retina Simulation using Cellular Automata and GPU Programming," *Machine Vision and Applications,* vol. 18, no. 6, pp. 331-342, 2007.

[19] R. Dolan and D. Guilherme, "GPU-Based Simulation of Cellular Neural Networks for Image Processing," in *Proceedings of International Joint Conference on Neural Networks*, Atlanta, Georgia, USA, 2009.

[20] S. Gobron and D. Mestre, "Information Visualization of Multi-dimensional Cellular Automata using GPU Programming," in *Information Visualization, 2007. IV '07. 11th International Conference*, Zurich, 2007.

[21] M. R. L´opez-Torres, J. L. Guisado, F. Jim´enez-Morales and F. Diaz-del-Rio, "GPU-based cellular automata simulations of laser dynamics," jornadassarteco.org, Seville, Spain, 2012.

[22] S. Wolfram, "Universality and complexity in cellular automata," *Physica D: Nonlinear Phenomena,* vol. 10, no. 1-2, pp. 1-35, 1984.

[23] S. Gobron, H. Bonafos and D. Mestre, "GPU accelerated computation and visualisation of hexagonal cellular automata," in *Cellular Automata - Lecture Notes in Computer Science*, vol. 5191, U. Hiroshi, M. Shin, N. Katsuhiro, K. Toshihiko and B. Stefania, Eds., Springer Berlin / Heidelberg, 2008, pp. 512-521.

[24] N. Ferrando, M. A. GosÃ¡lvez, J. CerdÃ¡, R. Gadea and K. Sato, "Octree-based, GPU implementation of a continuous cellular automaton for the simulation of complex, evolving surfaces," *Computer Physics Communications,* vol. 182, no. 3, pp. 628 - 640, 2011.

[25] C. Kauffmann and N. Piche, "Cellular automaton for ultra-fast watershed transform on GPU," in *Pattern Recognition, 2008. ICPR 2008. 19th International Conference on*, Tampa, FL, 2008.

[26] S. Rybacki, J. Himmelspach and A. M. Uhrmacher, "Experiments with Single Core, Multi-core, and GPU Based Computation of Cellular Automata," in *Advances in System Simulation, 2009. SIMUL '09. First International Conference on*, Porto, 2009.

[27] L. Žaloudek, L. Sekanina and V. Šimek, "Accelerating cellular automata evolution on graphics

processing units," *International Journal on Advances in Software,,* vol. 3, no. 1-2, p. 294–303, 2010.

[28] C. Grelck and F. Penczek, "Design and Implementation of CAOS: An Implicitly Parallel Language for the High-Performance Simulation of Cellular Automata," in *Cellular Automata - Simplicity Behind Complexity*, A. Salcido, Ed., 2011, pp. 545 - 566.

[29] J. Singler, "Implementation of Cellular Automata using a Graphics Processing Unit," in *Proceedings of ACM Workshop on General Purpose Computing on Graphics Processors*, Los Angeles, 2004.

[30] J. Drieseberg and C. Siemers, "C to Cellular Automata and Execution on CPU, GPU and FPGA," in *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, Madrid, Spain, 2012.

[31] A. R. Brodtkorb, T. R. Hagen and M. L. Sætra, "Graphics processing unit (GPU) programming strategies and trends in GPU computing," *Journal of Parallel and Distributed Computing,* vol. 73, no. 1, pp. 4-13, 2013.

[32] M. J. Gibson, E. Keedwell and D. Savić, "Understanding the efficient parallelisation of Cellular Automata on CPU and GPGPU hardware," in *Genetic and Evolutionary Conference(GECCO)*, Amsterdam, 2013.

[33] S. Harding and W. Banzhaf, "Fast Genetic Programming and Artificial Developmental Systems on GPUs," in *21st International Symposium on High Performance Computing Systems and Applications(HPCS'07)*, Saskatoon, SK, 2007.

[34] L. M. Rocha and W. Hordijk, "Material Representations: From the Genetic Code to the Evolution of Cellular Automata," *Artificial Life,* vol. 11, no. 1-2, pp. 189 - 214, 2005.

[35] R. Koenig and M. Daniela, "Cellular-Automata-Based Simulation of the Settlement Development in Vienna," in *Cellular Automata - Simplicity Behind Complexity*, A. Salcido, Ed., 2011, pp. 23-46.

[36] J. G. Hasbani, N. Wijesekara and D. J. Marceau, "An Interactive Method to Dynamically Create Transition Rules in a Land-use Cellular Automata Model," in *Cellular Automata - Simplicity Behind Complexity*, A. Salcido, Ed., 2011, pp. 3-22.

[37] A. C. Ximenes, C. M. Almeida, S. Amaral, M. I. S. Escada and A. P. D. Aguiar, "Spatial dynamic Modelling of deforestation in the Amazon," in *Cellular Automata - Simplicity Behind Complexity*, A. Salcido, Ed., 2011, pp. 47-66.

[38] S. D. Gregorio, G. Filippone, W. Spataro and G. A. Trunfio, "Accelerating wildfire susceptaibility mapping through GPGPU," *Journal of Parallel and Distributed Computing,* vol. 73, no. 8, pp. 1183-1194, 2013.

[39] S. Wolfram, "Cellular automata as models of complexity," *Nature,* vol. 311, pp. 419-424, 1984.

[40] Y. Guo, G. Walters, S.-T. Khu and E. Keedwell, "Optimal Design of Sewer Networks using hybrid cellular automata and genetic algorithm," in *Proc., 5th IWA WorldWater congress*, Beijing, China, 2006.

[41] M. Guidolin, A. Duncan, B. Ghimire, M. Gibson, E. Keedwell, A. S. Chen, S. Djordjević and D. Savić, "CADDIES: A New Framework for Rapid Development of Parallel Cellular Automata Algorithms for Flood Simulation," in *10th International Conference on hydroinformatics, HIC*, Hamburg, Germany, 2012.

[42] "The OpenMP Architecture Review Board Home Page," The OpenMP Architecture Review Board, 2012. [Online]. Available: http://openmp.org/. [Accessed August 2012].

[43] "OpenCL 1.2 Specification," Khronos group, 2012. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf. [Accessed March 2012].

[44] Nvidia, "GeForce 8600M specification," Nvidia, 2012. [Online]. Available: http://www.nvidia.com/object/geforce_8600M.html. [Accessed 2012].

[45] Nvidia, "GeForce GTX 560 Ti," Nvidia, 2012. [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-560ti/specifications. [Accessed

2012].

[46] M. J. Gibson, E. C. Keedwell and D. Savić, "Understanding the Efficient Parallelisation of Cellular Automata on CPU and GPGPU Hardware," in *GECCO 2013*, Amsterdam, 2013.