Regular Lattice and Small-World Spin Model Simulations using CUDA and GPUs

K. A. Hawick and A. Leist and D. P. Playne

2011

Data-parallel accelerator devices such as Graphical Processing Units (GPUs) are providing dramatic performance improvements over even multicore CPUs for lattice-oriented applications in computational physics. Models such as the Ising and Potts models continue to play a role in investigating phase transitions on small-world and scale-free graph structures. These models are particularly well-suited to the performance gains possible using GPUs and relatively high-level device programming languages such as NVIDIA's Compute Unified Device Architecture (CUDA). We report on algorithms and CUDA data-parallel programming techniques for implementing Metropolis Monte Carlo updates for the Ising using bit-packing storage, and adjacency neighbour lists for various graph structures in addition to regular hypercubic lattices. We report on parallel performance gains and also memory and performance tradeoffs using GPU/CPU and algorithmic combinations.

Keywords: Ising model; GPU; CUDA; data-parallel; cluster update

BiBTeX reference:

```
@ARTICLE{CSTN-093,
        author = {K. A. Hawick and A. Leist and D. P. Playne},
        title = {{Regular Lattice and Small-World Spin Model Simulations using CUDA
                and GPUs}},
        journal = {Int. J. Parallel Prog.},
        year = {2011},
        volume = {39},
        pages = {183-201},
        number = {CSTN-093},
        doi = {10.1007/s10766-010-143-4},
        institution = {Computer Science, Massey University},
        keywords = {Ising model; GPU; CUDA; data-parallel; cluster update},
        owner = {kahawick},
        timestamp = {2009.09.06}
}
```

Regular Lattice and Small-World Spin Model Simulations using CUDA and GPUs

K. A. Hawick and A. Leist and D. P. Playne

Institute of Information and Mathematical Sciences,
Massey University – Albany, North Shore 102-904, Auckland, New Zealand.
Email: { k.a.hawick, a.leist, d.p.playne }@massey.ac.nz
Tel: +64 9 414 0800   Fax: +64 9 441 8181

Abstract   Data-parallel accelerator devices such as Graphical Processing Units (GPUs) are providing dramatic performance improvements over even multicore CPUs for lattice-oriented applications in computational physics. Models such as the Ising and Potts models continue to play a role in investigating phase transitions on small-world and scale-free graph structures. These models are particularly well-suited to the performance gains possible using GPUs and relatively high-level device programming languages such as NVIDIA's Compute Unified Device Architecture (CUDA). We report on algorithms and CUDA data-parallel programming techniques for implementing Metropolis Monte Carlo updates for the Ising using bit-packing storage, and adjacency neighbour lists for various graph structures in addition to regular hypercubic lattices. We report on parallel performance gains and also memory and performance tradeoffs using GPU/CPU and algorithmic combinations.

Keywords: Ising model; GPU; CUDA; data-parallel; bit-packing.

1 Introduction

The Ising model [1–3] of a system of physical spins is a simple but very useful model in the study of phase transitions. In addition to its interpretation as a model of magnetic physical systems it forms a basis for comparison with phase transitions and critical phenomena occurring in many other systems including social networks, physical computer networks and overlay networks such as web page relationships on the world wide web. A phase transition is essentially when some property of a system is seen to display a sudden or unexpected change at a particular parameter value vicinity when that

parameter is varied. The Ising model shows a drastic change in the magneti-
sation of a system of magnetic spins at and around a critical temperature.
Simulations of the Ising model are often posed in terms of a quench whereby
a random "hot" system is quenched to a specific temperature. If that tem-
perature is below the inherent critical temperature ("cold"), the system
orders or separates out into distinct phases. In the case of the Ising model
these are visible as clumps of like spin values. A quench that keeps the sys-
tem above its critical temperature will not show this ordering phenomena.
A quench to around the critical temperature shows ordering and typically
clusters of like-like spins on all possible length scales.

The Ising model is often studied as a square planar lattice of spins or
as a cubic lattice in three dimensions. In principle however the model can
be set up on any graph or network whereby spin nodes interact with their
nearest neighbouring nodes according to a Hamiltonian or energy functional
of the form:

$$H = -\sum_{i,j} J_{ij} S_i S_j \tag{1}$$

where $S_i = \pm 1$, $i = 1, 2, ...N$ sites, and J_{ij} is $|J| = 1/k_B T$ is the ferromag-
netic coupling over neighbouring sites i,j on the network.

Ising model spins have just two values - "up" and "down" and can be
represented by a single bit. The Ising model can be extended to a system
with "spins" of more than just two values and the Q-State Potts model [4]
uses a similar Hamiltonian but with integer-valued spins up to some value
Q. The Ising and Potts model and a continuous vector field valued model
such as the Heisenberg [5] model have been well studied both analytically
as well as numerically and the location of their critical temperatures and
various other scaling properties such as critical exponents are known. The
models are therefore good bases of comparison for related systems. Recent
interest has been in systems that exhibit critical phenomena on arbitrarily
structured graphs or networks rather than regular lattices. Of particular
interest have been small-world systems which have a mix of localised nearest
neighbour node-node links as well as some long distance shortcut links that
drastically reduce the effective shortest path properties of the network as a
whole.

Irregular models such as these can only be studied numerically rather
than analytically and indeed even the regular three dimensional Ising model
has required numerical treatment. A great deal of supercomputer resources
were expended in the 1980s on the regular lattice three-dimensional Ising
model [6–9] and its numerical properties. The data-parallel programming
paradigm was found then to be very suited to executing fast simulations
of the model. In the present era, Graphical Processing Units (GPUs) have
a similar data-parallel programming structure to SIMD supercomputers of
yester-year [10].

Some work has already been done on the behaviour of the Ising and
related models on small-world systems, albeit mostly on 1-dimensional lat-

tices [11–16]. The major work to date on higher dimensional small-world rewired Ising systems is [17]. An obstacle to studying small-world effects is that the phenomena shows inherent scale free behaviour. This means that the controlling parameter - in this case the rewiring probability P - must be studied over many length scales to obtain sound quantitative conclusions. In [18] it was found necessary to scan in P logarithmically over several decades of scale.

In this present paper we report on how we have used GPUs to dramatically speed up simulations of the Ising model on both regular and irregular networks. We describe some of the memory management and optimal programming techniques to make best use of the very many cores on modern GPUs. We have improved upon performance reported in recent work for the regular lattice Ising model [10]. We have also obtained encouraging GPU performance results for Ising model simulations on *arbitrary* network structures.

Studying large and arbitrary network structures such as small-world rewired lattice, or damaged lattice systems with missing links, requires considerable computational power. Phase transitions are by their intrinsic nature sensitive to the control parameters and to locate a phase transition accurately - so as to make a comparison with other models - requires many independent statistically ample runs to be carried out and usually also a carefully managed scan in parameter space. GPUs therefore open up possibilities for study of realistically sized systems in applied problems areas that hitherto would have been infeasible.

In Section 2 we summarise the Metropolis Monte Carlo approach to simulating problems like the Ising model. We summarise key ideas for programming modern GPUs in Section 3 before describing our implementations for regular Ising systems in Section 4 and for irregular rewired network structures in Section 5. We give specific algorithmic and some GPU programming details in those sections. We discuss the performance of our implementations in Section 6 and offer some conclusions for use of these techniques in general numerical studies of complex systems and some areas for future work in Section 7.

2 Monte Carlo Cluster Updates

Considering a system of N spins each carrying a spin variable S_i and interacting according to 1 we initialise the system randomly - whereby it is effectively at an initially infinite hot temperature. We then quench the system by applying Monte Carlo updates to the individual spins so that they are conditionally flipped according to a probability which is derived from the quench temperature T using a Boltzmann weighting factor.

The Ising model has no intrinsic temporal dynamics of its own so the Monte Carlo update time is an artificial one. Empirically however, although it is purely an algorithmic artifice to progress the system through its internal

microstates, this artificial Monte Carlo time makes the system exhibit quantifiably similar equilibration behaviour as appears to occur in real physical systems [19, 20].

In summary the Metropolis algorithm is: **1)** Pick a spin to "hit"; **2)** Count how many like-like bonds the spin has with its nearest neighbours and hence the change in energy ΔE that would result if the hit spin were flipped. **3)** If the hit would result in a lowering of energy then do the flip; otherwise only do the flip with probability given by the Boltzmann weight factor $\exp(-\Delta E/k_B T)$. **4)** Repeat from 1 above.

Generally we express the Ising model coupling J in units of $k_B T$ (where k_B is Boltzmann's constant). The coupling is essentially just the inverse of temperature T and not an independent parameter. The probabilistic comparison is performed by using a numerical random number generator to "throw a probabilistic dice" to compare with the Boltzmann factor. There are various ways to choose spins to hit and these are discussed in the sections below since they can considerably affect the computational efficiency of a parallel algorithm. A simple completely ordered sweep through the spins is likely to introduce correlation artifacts; a completely random order is overly cautious and adds to the overhead of the computation; but various partially random sweeps through interleaved lattices has been found to be adequate and can aid considerably in parallelisation.

3 GPU Programming Model Summary

Graphics Processing Units or GPUs are highly parallel architectures containing a scalable array of multiprocessors, which provide a number of scalar processing units each. These processors can be programmed using CUDA kernels, which are executed by many threads in parallel. CUDA or Compute Unified Device Architecture has been developed by NVIDIA to allow kernels to be written in a C-style language and executed on NVIDIA GPUs [21]. Under this model, many threads are created which execute the same program or kernel on the multiprocessors of the GPU.

Thread execution management is performed by the hardware of the GPU and presents little overhead. The best performance is achieved when the problem is decomposed into many threads, thousands or even millions of threads are not uncommon. These threads are managed as blocks, which can be scheduled to execute on the multiprocessors. Every thread block is further subdivided into warps. A warp in CUDA's SIMT (single instruction, multiple thread) terminology is a group of 32 threads that are created, managed, scheduled and executed together by the multiprocessor's SIMT unit.

In most applications, memory access rather than computation is the limiting performance factor. To improve performance, GPUs contain several optimised memory types that can be used explicitly by the developer. For most applications, the memory types of interest are global, shared, texture and constant:

Global memory is the largest and slowest type of memory. This is the only type of memory that can be accessed by the host and is typically used for output from the GPU. Sequential memory transactions from a half-warp that read/write to global memory can be combined into a single coalesced transaction (see [21, 22] for the details of memory coalescing).

Shared memory allows threads within the same block to share information. This can be used effectively to reduce the number of global memory transactions required when threads within the same block must access the same information.

Texture memory is a cached method for reading from global memory. The cache automatically loads values spatially (in 1D, 2D or 3D) surrounding an accessed value. This is useful when the threads of a block all access addresses in the same spatial locality.

Constant memory is another read-only, cached method of accessing global memory. This cache is designed for when all the threads in a block access the same value from memory.

The platform used to do the performance measurements that we report, runs the linux distribution Kubuntu 9.04 64-bit. It uses an Intel® Core™2 Quad CPU running at 2.66GHz with 8GB of DDR2-800 system memory and an NVIDIA® GeForce® GTX 295 graphics card, which has 2 GPUs with 896MB of global memory each on board. Only one of the GPUs present was used for the measurements as each GPU must be controlled by separate host threads which would then incur explicit data communication and synchronisation overheads within the host program.

4 Regular Lattice Simulation

The data for a regular lattice Ising simulation is generally stored as an array of boolean values where each boolean value represents a single spin. The address of a cell's neighbours can be calculated from its (x,y) location and applying a suitable boundary condition method (in this paper we use periodic boundaries). This storage method is simple, memory efficient and with additional data can be suitable for simulations with decayed or rewired meshes.

Bit packing can be used to improve performance when simulating the Ising model on GPUs. Rather than an array of boolean values, the mesh is stored as an array of unsigned integers where each bit in the integer represents a spin. The advantage of using this memory storage method is that when the GPU threads load an element from memory, instead of one spin they load 32 spins. This equates to less costly memory transactions.

4.1 CUDA Checkerboard Implementation

This implementation is designed to be a highly optimised GPU simulation of the Ising model for a simple non-rewired, non-decayed lattice. These

optimisations are valid for the simple lattice case as the number and address of each spin's neighbours can be calculated explicitly.

This implementation stores spins as bit packed unsigned integers to minimise memory transactions on the GPU. As this implementation uses the checkerboard update method, alternating bits in these integers will have to be updated separately as they belong to different groups. To overcome this issue we use a method known as crinkling to re-order the bits in the mesh. Crinkling was pioneered for distributed array processors like the DAP by [23] and the same process is described for GPU architectures in [24].

The process of **crinkling** a mesh involves relocating the red and black spins of the mesh into separate parts of the array or into separate arrays. This allows them to be processed separately and is an extremely important part of this implementation. By crinkling the mesh (bit-wise crinkle), all the bits in each unsigned integer will belong to the same group. The application of a crinkle operation creating two separate meshes can be seen in Figure 1.
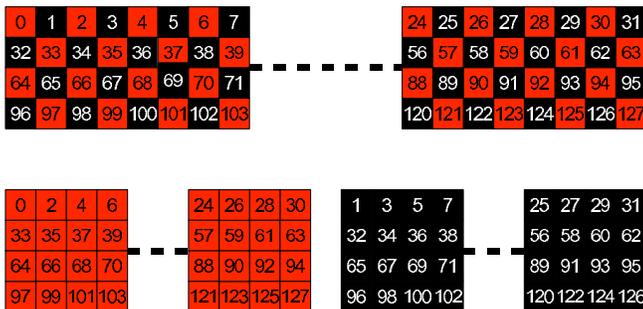

Fig. 1 A crinkle operation applied to a mesh resulting in two separate arrays.

Once these individual meshes have been created the checkerboard method can be implemented quite simply by updating them one at a time. To update an element in one mesh, the values of the neighbouring spins must be read from the opposite mesh, ie. to update one element in the red array (32 spins), four elements from the black array must be loaded. The calculation of the neighbouring elements (with periodic boundary conditions) can be seen in Algorithm 1 (Note that in this paper we use % to denote the modulo operator).

The four neighbours of the element (x,y) are: (xm1,y), (xp1,y), (x,ym1) and (x,yp1); however, these are the element-wise neighbours, what is needed is the spin-wise neighbours. The (x, ym1) and (x,yp1) neighbours are correct as the array is crinkled in the x direction. However to produce the correct (xm1,y) and(xp1,y) neighbours a SHIFT and OR operation is necessary as seen in Algorithm 2 (Note that << denotes a left shift operator and >> denotes a right shift).

Algorithm 1 Calculating the positions of the neighbours of an element (each red/black mesh has dimensions (X, Y)).

$W \leftarrow X/\text{spins\_per\_element}$
$ym1 \leftarrow (y = 0) ? (Y - 1) : (y - 1)$
$yp1 \leftarrow (y = Y - 1) ? (0) : (y + 1)$
if $((black) \text{ XOR } (y\%2) = 0)$ **then**
$\quad xm1 \leftarrow (x = 0) ? (W - 1) : (x - 1)$
$\quad xp1 \leftarrow x$
else
$\quad xm1 \leftarrow x$
$\quad xp1 \leftarrow (x = W - 1) ? (0) : (x + 1)$

Algorithm 2 Calculating the positions of the neighbours of an element (each red/black mesh has dimensions (X, Y)).

if $((black) \text{ XOR } (y\%2) = 0)$ **then**
$\quad (\text{xm1,y}) \leftarrow ((\text{xp1,y}) >> 1) \text{ OR } ((\text{xm1,y}) << 31)$
else
$\quad (\text{xp1,y}) \leftarrow ((\text{xm1,y}) << 1) \text{ OR } ((\text{xp1,y}) >> 31)$

This provides four elements containing the spinwise neighbours on the element at (x,y). Using these values we can calculate a new value for the element.

4.2 Bit-Packing and Associated Logic

As each element in the lattice contains 32 spins, we cannot use the Ising calculation in its standard form. Instead we construct a series of bit logic equations to compute the change in 32 spins simultaneously. The reason for these somewhat complicated bit logic expressions is they can compute the simulation significantly faster than extracting the 32 spin and computing their change separately.

To update an element containing multiple spins we want to create a 'flip' mask where each bit is 1 if the corresponding spin should be flipped and 0 if it should remain the same. There are two conditions for flipping a spin - one if there are $>= 2$ neighbours with different spins or if there are $<= 1$ neighbours with different spins and some random condition is fulfilled (the probability is defined by the number of neighbours and the temperature of the system).

First of all we generate a flip mask (n_{mask}) for the first condition using the following equations (the element in question is named (x,y) and its four bit wise neighbours are: (xm1,y), (xp1,y), (x,ym1) and (x,yp1)).

$$n_1 = (x, ym1) \text{ XOR } (x, y) \tag{2}$$
$$n_2 = (xm1, y) \text{ XOR } (x, y) \tag{3}$$
$$n_3 = (xp1, y) \text{ XOR } (x, y) \tag{4}$$
$$n_4 = (x, yp1) \text{ XOR } (x, y) \tag{5}$$
$$n\_mask = (n_1 \text{ AND } n_2) \text{ OR } (n_1 \text{ AND } n_3) \text{ OR } (n_1 \text{ AND } n_4) \text{ OR }$$
$$(n_2 \text{ AND } n_3) \text{ OR } (n_2 \text{ AND } n_4) \text{ OR } (n_3 \text{ AND } n_4) \tag{6}$$

4.3 Random Number Generation

The next consideration for this implementation is the generation of random numbers. As each element in the mesh contains 32 different spin values, 32 separate random numbers are required. However, loading in 32 separate random numbers will require an undesirable number of memory transactions. In fact 32 random numbers are not required. The random number values are used when there are 0 or 1 neighbours with different spins with a probability dependent on the temperature.

As the temperature does not change (at least not within a time-step) this comparison can be performed during the random number generation process. Instead of generating random numbers and storing them in an array, the random number generator can generate sets of 32 random numbers, perform the comparison with the probability values and store the results in two unsigned integers (one each for the probability values of 0 or 1 unlike neighbours).

Each bit of the integers is 1 if the random value was less than the probability value and 0 if it was not. Thus the spin should be flipped if it has 0 or 1 neighbours and the bit in the appropriate random number int is 1. To use these random number integers we must create a mask for the two conditions of 0 dislike neighbours and 1 dislike neighbour. These masks can be computed as follows:

$$n_0\_mask = (\text{NOT } n_1) \text{ AND } (\text{NOT } n_2) \text{ AND} \tag{7}$$
$$(\text{NOT } n_3) \text{ AND } (\text{NOT } n_4)$$
$$n_1\_mask = ((n_1 \text{ AND } n_2 \text{ AND } n_3) \text{ OR} \tag{8}$$
$$(n_1 \text{ AND } n_2 \text{ AND } n_4) \text{ OR}$$
$$(n_1 \text{ AND } n_3 \text{ AND } n_4) \text{ OR}$$
$$(n_2 \text{ AND } n_3 \text{ AND } n_4)) \text{ AND}$$
$$(\text{NOT } n_0\_mask)$$

With these masks n_mask, n_0_mask and n_1_mask and the random number elements rnd_0 and rnd_1 we can formulate a final flip mask:

$$flip\_mask = n\_mask \text{ OR}$$
$$(n_0\_mask \text{ AND } rng_0) \text{ OR}$$
$$(n_1\_mask \text{ AND } rng_1) \tag{9}$$

With this flip mask we can calculate the new value for the element syx.

$$syx = syx \text{ XOR } flip\_mask \tag{10}$$

This bit logic is applied to each element in the mesh (alternating meshes) such that each mesh is updated once per time-step. This process can be repeated as for the necessary number of time-steps after which the meshes must be uncrinkled (the reverse of the crinkling process) to provide the final state of the system.

4.4 CUDA Implementation Of The Regular Lattice Ising Model

This implementation of the Ising model is specifically designed for CUDA. The implementation can either take an existing Ising system or randomly initialise one. The Crinkle and Uncrinkle operators are performed on the GPU and described in [24]. One thread is created for each element in the crinkled arrays, these threads will perform the algorithm described above for a single element from one array and write the results to the same array. This process is performed twice per time-step (once for each crinkled array red/black). The high level description of this algorithm is shown in Algorithm 3.

Algorithm 3 Calculating the positions of the neighbours of an element (each red/black mesh has dimensions (X, Y)).

COMPUTE x, y
LOAD rnd_0, rnd_1 from global rnd arrays
LOAD (x, y) from update array
COMPUTE $ym1, xm1, xp1, yp1$
LOAD $(x, ym1), (x, yp1), (xm1, y), (xp1, y)$ from alternate array
COMPUTE spinwise elements $(xm1, y), (xp1, y)$
COMPUTE $n\_mask, n_0\_mask, n_1\_mask$
COMPUTE flip_mask
COMPUTE updated (x, y) value
WRITE (x, y) to update array

This algorithm can compute an Ising simulation time-step very quickly and this Ising kernel is not the limiting performance factor of the simulation. The cost of computing random numbers quickly becomes more expensive. The quality of random numbers is very important to the accuracy of the results that the Ising simulation produces. For our implementation (the implementation used to produce the performance results presented in this paper) we have used the Lagged-Fibonacci Generator implemented in CUDA described in [25]. Computing random numbers is inextricably linked to the Ising simulation yet should be considered a separate challenge. Thus in all our performance data we present both the time taken for the Ising simulation and the time taken to generate the random numbers.

4.5 Regular Lattice Performance Results

The GPU implementation has been compared to a standard CPU checkerboard implementation to give an idea of the speed up it provides. The CPU

implementation performs the checkerboard update on a boolean mesh and does not use the bit-packing method. In both two- and three-dimensions the GPU provides significant performance improvements, the speedup factor for both implementations is just over 125 times faster than the CPU implementations. The performance results for the total time, as well as the individual RNG and Ising kernel times are shown in Figure 2.
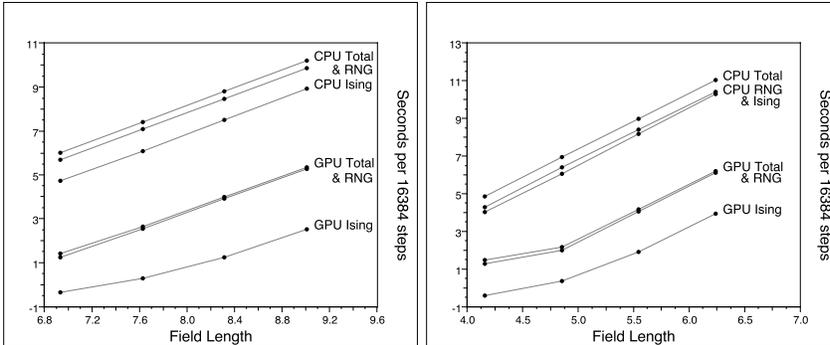

Fig. 2 Performance results comparing CPU and GPU implementations of the Ising simulation. In two-dimensions on the left (1024^2 to 8192^2) and in three-dimensions on the right (64^3 to 512^3). Results are shown in ln-ln scale. Error bars - calculated from several independent runs - are present but are smaller than the plot symbols.

One point of interest is the optimal GPU memory type to employ. In previous discussions of lattice-based simulations (mainly the Cahn-Hilliard simulation [22]), texture memory has provided the best performance for CUDA implementations. However, in our experiments we found that the kernel completed faster when global memory was used. The global memory access methods of the 200 series graphics cards reduces the negative impact of misaligned global memory accesses but spatial caching functionality of texture memory normally provides a performance benefit.

We believe that the number of neighbours that the Ising model accesses is the reason for the performance results. The Cahn-Hilliard Equation has a much larger memory halo (12 neighbours) than the Ising model (4 neighbours) and can thus make better use of the values in the texture cache. However, for the Ising model the increased copy time (copying data into texture bound arrays) had a larger impact than the improved memory access the cache provides.

5 Small-World Rewired Lattice Simulation

In this section we describe an alternative implementation for the case of a system on an arbitrary graph network rather than a regular lattice. The

particular system of interest is however constructed starting with a regular lattice, and a number of edges are rewired randomly to create the shortcuts common to small-world networks.

Either end of an edge, which connects two cells, is independently considered for rewiring with probability $\frac{1}{2}P$, thus rewiring a fraction P of all edges. A new neighbour is selected randomly from all cells of the same colour as the previous neighbour, thus preserving the condition that no cells of the same colour are connected to each other, which is the case when using the checkerboard pattern on the regular lattice. This limitation makes it possible to update cells of the same colour in parallel without the risk of running into race conditions. Figure 3 illustrates such a rewired lattice. In this example, the edge connecting cells 0 and 1 was rewired and now connects cells 0 and 19, decreasing the degree of cell 1 to 3 and increasing the degree of cell 19 to 5.
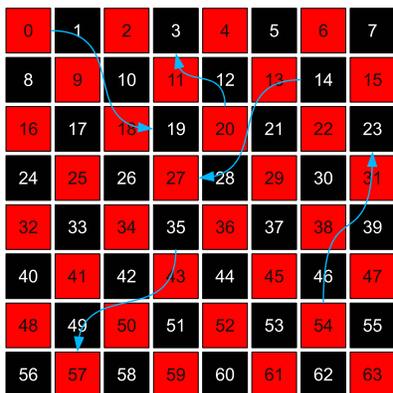

Fig. 3 A 2-dimensional lattice where every cell is initially connected to its 4 direct neighbours using periodic boundaries, with a small number of rewired edges. The numbers represent the cell IDs.
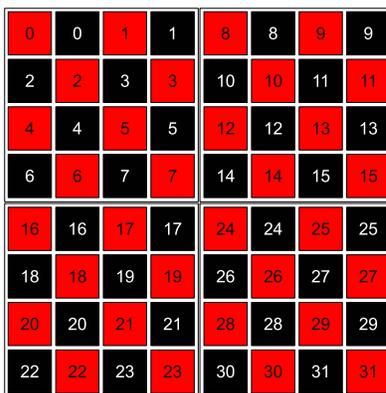
Fig. 4 The ID of the thread that processes a particular cell for thread blocks of size 2×4. The actual implementation uses thread blocks of size 8×16 for 2D and $4 \times 8 \times 8$ for 3D simulations.

The CUDA implementation uses thread blocks of size 8×16 in 2D and $4 \times 8 \times 8$ in 3D when processing either the red or black cells of a 16×16 and $8 \times 8 \times 8$ block of cells respectively. Figure 4 illustrates which thread processes a particular cell.

5.1 Rewired Irregular Data Structure

The neighbour coordinates of cells that are not affected by rewiring can be calculated and do not need to be stored explicitly, thus conserving memory and memory bandwidth. However, for all other cells, the neighbours can

Fig. 5 The two vertex-arrays for red and black cells respectively. The figures illustrate the IDs of the cells whose data is stored at the respective positions. The actual data stored are the indices into the arc-array (Figure 6) at which the adjacency-list information begins.

not be deferred and need to be stored and looked up. Two vertex arrays of length $\frac{1}{2}N$ each, one for the red cells and one for the black cells, are used to store the index into the arc-array at which the neighbour information for the cells affected by rewiring is stored. N is the system size (i.e. the total number of cells). As only cells of the same colour are processed in one iteration, two different vertex-arrays are used to enable threads to access sequential elements of the arrays. But this is not enough to achieve coalesced memory accesses, as the threads of a half-warp must access sequential memory addresses which adhere to strict alignment requirements for coalescing to work. However, the x-dimensions of the thread blocks are 8 and 4 for 2D and 3D respectively as mentioned before, which means that the threads of a half-warp process values from 2 different rows of cells in 2D and 4 different rows in 3D. To accommodate for this fact, the vertex-arrays store the elements in the order of the thread IDs instead of the cell IDs. This is illustrated in Figure 5.
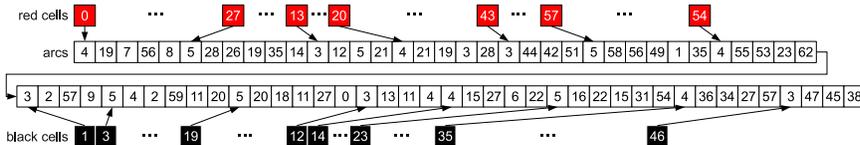

Fig. 6 While the vertex-arrays have an element for every cell, the arc-array only stores the adjacency-lists of cells that were affected by rewiring. The neighbours of all other cells can be deferred from their own coordindates, thus reducing the memory storage and bandwidth requirements. The arrows between the vertex and arc-arrays represent the index into the arc-array stored in the vertex-arrays. The contents of vertex-array elements that were not affected by rewiring are irrelevant. The first element of an adjacency-list is its length.

The arc-array, illustrated in Figure 6, contains all explicitly stored adjacency-lists. The first element of every such list is its length, followed by the IDs of the cells that are adjacent to the respective source cell.

The spin states are stored as 2D or 3D arrays in global memory using **unsigned char** values. Only the least significant bit (LSB) is used to record the spin state in the Ising model. Additional bits can be used to store more

spin states, which makes it easy to extend the algorithm to the Potts model. Bit 6 is used to mark cells that are affected by rewiring and therefore need to explicitly look their neighbour information up using the vertex and arc-arrays.

5.2 CUDA Implementation Of The Rewired Ising Model

The small-world, rewired lattice Ising simulation running on the GPU updates either the red or the black cells of the checkerboard pattern in parallel. This allows it to avoid race conditions, as all the neighbours of a red cell are black and vice-versa. This means that it takes two kernel iterations to perform a full simulation step. Algorithm 4 describes the host code that prepares and manages the CUDA kernel execution. The spin-, vertex- and arc-arrays have been described before. The random number array is used to temporarily store the random numbers required by the Ising kernel and is filled by a dedicated random number kernel.

Algorithm 4 Evolve the model by $STEPS$ simulation steps. This is the host code that manages the graphics processing unit. N is the system size.

---

allocate device memory for the spin-array S, vertex-array V, arc-array A, random number array R and the arrays needed for the random number generators
copy metropolis table T to constant device memory
set the marker bit $mask\_modified$ on the spin value of cells affected by rewiring
copy initial spin values to the device
copy RNG seeds to the device
do in parallel on the device using 32768 threads: initialise the RNGs
for $i \leftarrow 1$ to $STEPS$ **do**
  do in parallel on the device using 32768 threads:
        generate $N/2$ random numbers and store them in R
  do in parallel on the device using $N/2$ threads: process all red cells
  do in parallel on the device using 32768 threads:
        generate $N/2$ random numbers and store them in R
  do in parallel on the device using $N/2$ threads: process all black cells
copy final spins back to the host
remove all marker bits from the spin values

The random number kernel implements Marsaglia's lagged-Fibonacci random number generator (RNG) [26] as described in [27]. Every CUDA thread executing this kernel uses its own lag-table to produce an independent stream of random numbers. With a table length of 97, every RNG requires 400-bytes of device memory in total to store the lag-table as well as auxiliary data. Because every thread generates an independent stream of random numbers, there are no issues with read-write race conditions. However, for the implementation to perform well, it is necessary that all threads in a half-warp collectively generate new random deviates, even if some of them do not actually need a new random number. This enables global memory reads and writes to be coalesced.

The Ising kernel uses one CUDA thread per cell and updates $\frac{1}{2}N$ cells per iteration. If the RNG function was called directly by this kernel, then it

would require 3200MB of global memory just for the RNGs when simulating a system with 256^3 cells. Therefore, a separate kernel which is executed by 32768 CUDA threads is used. Every thread generates x random numbers per kernel call, where x is the next multiple of 32768 that is equal to or greater than $\frac{1}{2}N$.

Algorithm 5 Calculating the coordinates (ix, iy, iz) of the cell processed by the current thread. (bix, biy, biz) are the coordinates of the upper left front corner of the thread block. id is the cell's index into the vertex-array. D is the length of all dimensions. Every simulation step consists of two kernel calls, one to process all red cells (offset1 $\leftarrow$ 1, offset2 $\leftarrow$ 0) and one to process all black cells (offset1 $\leftarrow$ 0, offset2 $\leftarrow$ 1).

---

declare bix, biy, biz, bid in shared memory
if first thread in thread block **then**
    $bix \leftarrow (blockIdx.y * gridDim.x + blockIdx.x) * (blockDim.x * 2)$
    $biy \leftarrow ((int)(bix/D)) * blockDim.y$
    $biz \leftarrow ((int)(biy/D)) * blockDim.z$
    $bix \leftarrow bix\%D$
    $biy \leftarrow biy\%D$
    $bid \leftarrow blockIdx.y * gridDim.x + blockIdx.x$
    $bid \leftarrow bid * blockDim.z * blockDim.y * blockDim.x$
synchronise threads in thread block
$id \leftarrow bid + threadIdx.z * blockDim.x * blockDim.y$
$id \leftarrow id + threadIdx.y * blockDim.x + threadIdx.x$
$ix \leftarrow bix + threadIdx.x * 2$ //process every second cell
$iy \leftarrow biy + threadIdx.y$
$iz \leftarrow biz + threadIdx.z$
$ix \leftarrow ix + ((iz \text{ AND } 1) \text{ XOR } (iy \text{ AND } 1)?\text{offset1}:\text{offset2})$ //checkerboard

---

Algorithm 5 describe the first part of the Ising kernel. The first thread in every thread block calculates the cell with the lowest ID processed by this block. Following this, all threads can use their thread indices as offsets to calculate the coordinates of the cell that they process. The id is used to read from the vertex and random number arrays as described in Section 5.1.

The pseudo-code for the second part of the Ising kernel is given in Algorithm 6. Every thread loads the spin of its cell. The spin value carries an additional marker bit which is used to indicate if the cell was affected by rewiring. If this marker bit is set, then the thread needs to load the cell's adjacency-list from global memory. Otherwise, it calculates the coordinates of neighbouring cells. Then the spin of the cell is compared to the spins of its neighbours, and the like-like bond counter is incremented if flipping its own spin value would make it equal to the spin of a neighbour and decremented if the spins are already equal. Eventually, the spin is flipped if this either increases the like-like bonds or with a random probability which depends on the temperature.

The spin-array is bound to a texture reference to read the spin values. Texture fetches are optimised for spatial locality, which is exactly what is needed, as every cell requires the spins of its neighbours and most of its neighbours are stored spatially close to it unless the value of P is rather

Algorithm 6 Every CUDA thread compares the spin of a cell (x, y, z) to the spins of its neighbours and flips it if this either increases the like-like bonds or with a random probability which becomes smaller the more like-like bonds would be undone by flipping the spin. S is the spin-array, V the vertex-array, A the arc-array, D the length of all dimensions, R the random number array and T the Metropolis table.

```
v ← S(ix, iy, iz) //load spin value (including marker bits)
s ← v AND mask_spin //remove non-spin bits
b ← 0 //change in like-like bonds if spin gets flipped
if v AND mask_modified then
    //the cell was affected by rewiring, load neighbour data from arc-array
    idx ← V[id] //load the index into the arc-array
    c ← A[idx] //load the adjacency-list length
    for all n_id ∈ {A[idx + 1], A[idx + 2], ..., A[idx + c]} do
        //calculate the neighbour coordinates from the neighbour's cell ID n_id
        n_ix ← n_id%D
        n_iy ← ((int)(n_id/D))%D
        n_iz ← (int)(n_id/(D * D))
        n_s ← S(n_ix, n_iy, n_iz) AND mask_spin //load the neighbour's spin
        b ← b + (s = n_s? − 1 : 1) //compare spins
else
    //calculate the neighbour coordinates
    n_ix ← ix = 0?D − 1 : ix − 1 //cell on left
    n_s ← S(n_ix, iy, iz) AND mask_spin
    b ← b + (s = n_s? − 1 : 1)
    do the same for all other neighbours (right, above, below, front, behind)
r ← R[id] //load the random number generated for this cell
if b ≥ 0||r < T[−b] then
    //store the flipped spin including marker bits to global memory
    S(ix, iy, iz) ← (s XOR 1) OR (v AND mask_modified)
```

large. Typically, texture references are bound to CUDA arrays, which are "opaque memory layouts optimized for texture fetching" [21]. However, they are also read-only and we need to be able to update the spin value. This can be solved by writing to a second array, which is stored in global memory and copied to the CUDA array before every iteration. This copy, however, turns out to be too expensive even though it is a fast device-to-device copy. Simply reading from global memory is faster than using texture fetches that require this copy operation. For the 2D simulation exists another solution though. CUDA allows 1D and 2D texture references to be bound directly to linear memory, which means that the texture can be bound to the same array that is used to write to. No copy operation is needed in this case. The written values are only guaranteed to be visible after the kernel call returns, but this is not an issue, as the checkerboard pattern ensures that a spin is never read and modified in the same call. Unfortunately, 3D texture references can not be bound to linear memory (as of CUDA 2.3), which means that the 3D implementation has to read directly from global memory. While this comes at a performance deficit compared to the 2D implementation, it frees up the texture cache, which can be useful in a different way. Binding a 1D texture reference to the arc-array allows the neighbours of a cell to be read using texture fetches, which provides a moderate performance boost.

5.3 Irregular Rewired Kernel Performance Results

This section shows how the execution times of the small-world Ising CUDA implementation are affected by the system size N and rewiring probability P. It also compares the measurements to a checkerboard update based CPU implementation, which also uses an implementation of Marsaglia's random number generator. The timings are for 16384 simulation steps. However, the CPU algorithm was only executed for 2048 steps and the timing results were scaled up by a factor of 8 for comparison. Every data point is the mean value of 10 CUDA or 5 CPU simulation runs using different random number seeds for every run.
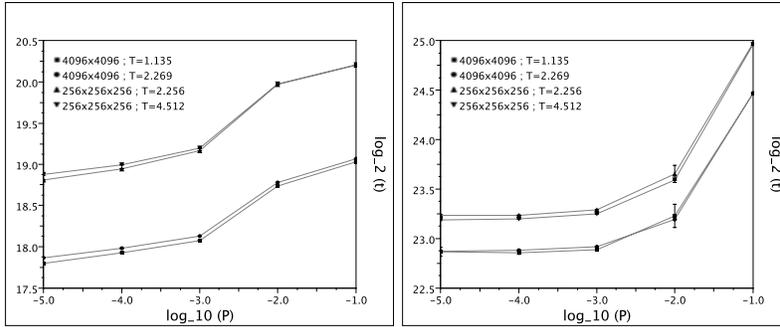

Fig. 7 These plots illustrate how the rewiring probability P affects the timing results t (in milliseconds). The results for the CUDA kernel are shown on the left and the results for the CPU on the right. Error bars are smaller than the symbol sizes except for the CPU results with $P = 10^{-2}$.

Figure 7 shows how both the GPU and CPU implementations cope with increasing rewiring probabilities. As expected, the performance degrades with increasing randomness. An increase in the randomness of the graph means more explicit look-ups of adjacency-list data from global memory for the CUDA implementation and less predictable memory accesses for the CPU. Every rewired edge changes the neighbour structure of 3 cells and there are 2 times (2D simulation) or 3 times (3D simulation) as many edges as cells. Therefore, the number of modified cells increases quickly with the value of P. Both implementations show a significant jump in the timings between $P = 10^{-3}$ and $P = 10^{-2}$. For the CPU, this is even more significant between $P = 10^{-2}$ and $P = 10^{-1}$.

Figure 8 compares the performance measurements for the CUDA and CPU implementations. Depending on the system size and value of P, the GPU outperforms the CPU by $\approx 5 - 27$ times in the 3D simulations and by $\approx 9 - 42$ times in the 2D simulations. The smaller performance gain of the 3D CUDA simulations compared to the 2D simulations is due to the texture caching issue explained in section 5.2 and the higher average degree.
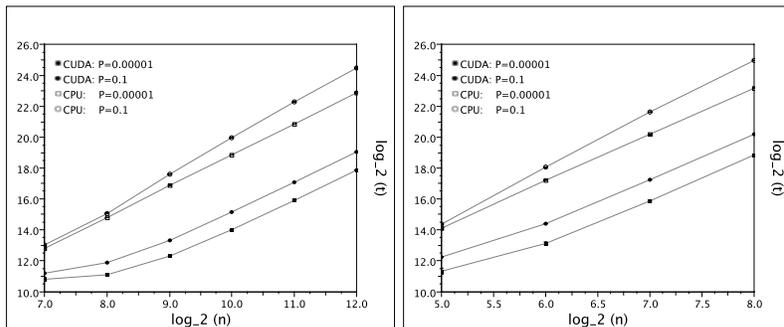
Fig. 8 These plots show the timing results t (in milliseconds) compared to the system size N, where $n^2 = N$ for 2D (left) and $n^3 = N$ for 3D (right). The least square linear fits for the 2D/3D simulations are 1.87/2.92 (filled squares), 1.96/2.88 (filled circles), 2.03/3.05 (empty squares) and 2.29/3.49 (empty circles). Error bars are smaller than the symbol sizes.

The least square linear fits quoted in the caption show that the CUDA implementation scales better than the CPU implementation. The comparably small difference in the CUDA measurements between 2D systems of size $(2^7)^2$ and $(2^8)^2$ is due to the fact that random numbers are generated in multiples of 32768, but $(2^7)^2 = 16384$, which means that half of the random numbers are not actually used. This also highlights the fact that the random number generation accounts for a significant portion of the overall execution time.

5.4 Alternative Implementation Approaches for Irregular Networks

The previous section described our overall best algorithm for rewired Ising simulations. Here we briefly describe a number of approaches that, in certain configurations, slightly outperform this implementation.

The first alternative approach only uses up to 32768 threads for the Ising kernel, updating multiple cells per thread. This way the random number generation can be merged into the Ising kernel and the random numbers do not have to be written to global memory. Every thread block updates x consecutive blocks of cells, where x is the next multiple of 32768 that is equal to or greater than $\frac{1}{2}N$. By updating consecutive blocks of cells, it gets the most benefit from the texture cache. This kernel performs slightly better for $P \lesssim 0.01$, but suffers considerably more from higher values of P.

The second alternative implementation uses a bit-packing approach for random numbers similar to the one described for the regular lattice implementation. However, there are not only two cases in the 2D rewired lattice Ising simulation that require a random number (i.e. 3 or 4 like neighbours), but k cases, where k is the degree of the cell. This is because the degree of a rewired cell is not always $k = 4$. Thus it is necessary to bit-pack the result

for every possible case in the given graph, which is $k = \{1, 2, 3, \ldots, k_{max}\}$, where k_{max} is the maximum degree of any cell in the graph. This means that only $ceil(32/k_{max})$ random number results can be packed into a 32-bit unsigned integer, thus having to call the random number kernel more frequently the larger k_{max}. This kernel is slightly faster for small systems with small values of P, but larger systems and/or higher values of P make it more likely that k_{max} is larger and thus perform worse.

The third approach attempts to optimise the neighbour lookup from the arc-array. This is not very optimal in the main implementation, as even consecutive threads access the arc-array with a stride of $k + 1$, which means the reads can not be coalesced well if at all. Unfortunately, the degree of every cell is arbitrary, which makes it complicated to optimise the memory layout. The 3D implementation somewhat improves this sequential reading from global memory by using a 1D texture reference. However, even this is not optimal and does not give a performance benefit at all if the texture cache is already used for the spins. Our approach to improving this access is to let the threads in the first warp of a thread-block look up their indices into the arc-array and then use all n threads in the thread block to load $n, n \times 2$ or $n \times 3$ elements from the arc-array, more the larger the value of P, starting from the index determined by the first thread in the thread-block. These global memory reads are mostly coalesced. The values are then written to shared memory. Subsequently, if a thread needs to load the neighbour information, it can look it up from the cached data stored in the fast on-chip shared memory, and only needs to read from global memory in case of a cache-miss. However, this kernel introduces a significant computational overhead and only outperforms the main implementation for very large P.

6 Discussion

Table 1 summarises our performace findings for the cases of various regular lattice and irregular small-world network structured Ising model simulations of GPUs with comparison figures on a single core of a typical CPU.

We provide the CPU figures only as a rough comparison. In practice the nature of these simulations are that multiple jobs can be usefully run to make better use of multi-cored conventional CPUs. When simulating very large systems, the consequent large memory requirements can cause contention between cores.

In practice we are bound at present by the memory available on the graphics cards—which is 896MB per GPU for the GTX 295 used for these measurements. Especially the memory requirements for the irregular small-world simulations increase quickly with the rewiring probability p, system size N and number of dimensions d, as these affect how much neighbour information has to be stored explicitly. With this graphics card, it is possible to simulate irregular graph systems of approximately up to $N = 8192^2$ with $P \leq 0.01$ or $N = 320^3$ with $P \leq 0.1$. With the use of bit-packing for

Simulated System Size	Monte Carlo Hits (million per second)	
	CPU	GPU
Regular $N = 8192^2$	40.8	5239.9
Regular $N = 512^3$	34.8	4360.1
Irregular $N = 4096^2$; $P = 10^{-5}$	35.9	1151.1
Irregular $N = 4096^2$; $P = 10^{-1}$	11.8	499.6
Irregular $N = 256^3$; $P = 10^{-5}$	28.8	572.3
Irregular $N = 256^3$; $P = 10^{-1}$	8.4	226.4

Table 1 Ising Kernel Performance Summary. The CPU results are for a single core of an Intel Core2 Quad CPU running at 2.66GHz, and the GPU results are for one GPU of the NVIDIA GTX 295.

the regular lattice simulations we can simulate meshes of $N = 8192^2$ and $N = 512^3$, an improvement on previously stated limits [10].

7 Conclusions

We have shown that GPUs are excellent cost effective accelerator platforms for Ising simulation work due to their high number of homogenous cores and the consequent data-parallel programming model which is well suited to this model. We have found that even for highly irregular problems we have been able to obtain considerable performance improvements over a conventional CPU implementation - typical a factor of 20-40 fold.

We conclude that a good architecture for managing production level investigations of critical phenomena might be a Beowulf style cluster with relatively cheap interconnect hardware between nodes and a suitable head node with appropriate storage capacity. Each individual compute node would have a medium priced CPU with an associated GPU card so that the combined CPU/GPU-Kernel programming model we have employed can be exploited to the full. Depending upon the size of system being investigated and the relative costs of memory, it might be preferred to have dual core CPUs and dual GPUs such as provided by the GTX 295 card. We would anticipate that the forthcoming GT300 based graphics cards—which reportedly have up to 512 cores and a new memory model—may be even better for this sort of simulation model.

We have found CUDA an entirely usable programming model for use with NVIDIA GPU cards. In practice, the general form of CUDA code has much in common with OpenCL program source code. However, we think it not unlikely that for this sort of application, where speed is of the utmost importance, simply due to the statistical requirements of the investigation, it may remain worthwhile running vendor optimised code such as CUDA for the immediate future at least.

We note that as in the supercomputer era of the 1980s the Ising model is very susceptible to algorithmic optimisations that make the particular

systems size or model variation fit the parallelism available on a particular architecture. It is an optimistic sign for parallelism in general that many of these ideas can now be exploited on a commodity chip and with a relatively open programming model.

We are aware of super-fast spin-cluster update algorithms such as that by Wolff [28] that considerably improve the Ising model Monte Carlo update in the vicinity of the critical temperature. We are presently experimenting with optimal implementations on GPUs.

References

1. Niss, M.: History of the Lenz-Ising Model 1920-1950: From Ferromagnetic to Cooperative Phenomena. Arch. Hist. Exact Sci. **59** (2005) 267–318
2. Ising, E.: Beitrag zur Theorie des Ferromagnetismus. Zeitschrift fuer Physik **31** (1925) 253258
3. Onsager, L.: Crystal Statistics I. Two-Dimensional Model with an Order-Disorder Transition. Phys.Rev. **65** (1944) 117–149
4. Baxter, R.J.: Exactly Solved Models in Statistical Mechanics. Number ISBN 0-12-083180-5. Academic Press (1982)
5. Anderson, P.W.: New approach to the theory of superexchange interactions. Phys. Rev. **115** (1959) 2–13
6. Bhanot, G., Duke, D., Salvador, R.: A fast algorithm for the Cyber 205 to simulate the 3d Ising Model. J. Stat. Phys. **44** (1988) 985–1002
7. Blöte, H.W.J., Compagner, A., Croockewit, J.H., Fonk, Y.T.J.C., Heringa, J.R., Hoogland, A., Smit, T.S., van Willigen, A.L.: Monte Carlo Renormalization of the Three-Dimensional Ising Model. Physica A (1989) 1–22
8. Pawley, G.S., Swendsen, R.H., Wallace, D.J., Wilson, K.G.: Monte-Carlo renormalization group calculations of critical behaviour in the simple cubic Ising model. Phys. Rev. B **29** (1984) 4030–4040
9. Baillie, C., Gupta, R., Hawick, K., Pawley, G.: Monte-Carlo Renormalisation Group Study of the Three-Dimensional Ising Model. Phys.Rev.B **45** (1992) 10438–10453
10. Preis, T., Virnau, P., Paul, W., Schneider, J.J.: GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model. Journal of Computational Physics **228** (2009) 4468 – 4477
11. Boyer, D., Miramontes, O.: Interface motion and pinning in small-world networks. Phys. Rev. E **67** (2003)
12. Pękalski, A.: Ising model on a small world network. Phys. Rev. E **64** (2001)
13. Jeong, D., Hong, H., Kim, B.J., Choi, M.Y.: Phase transition in the Ising model on a small-world network with distance-dependent interactions. Phy. Rev. E. **68** (2003)
14. Kim, B.J., Hong, H., Holme, P., Jeon, G.S., Minnhagen, P., Choi, M.Y.: *XY* model in small-world networks. Phy. Rev. E. **64** (2001)
15. Hong, H., Kim, B.J., Choi, M.Y.: Comment on "Ising model on a small world network". Phy. Rev. E. **66** (2002) 018101
16. Yi, H., Choi, M.S.: Effect of quantum fluctuations in an Ising system on small-world networks. Phy. Rev. E **67** (2003)
17. Herrero, C.P.: Ising model in small-world networks. Phys. Rev. E **65** (2002)

18. Hawick, K.A., James, H.A.: Ising model scaling behaviour on z-preserving small-world networks. Technical Report arXiv.org Condensed Matter: cond-mat/0611763, Information and Mathematical Sciences, Massey University (2006)
19. Hawick, K.A.: Domain Growth in Alloys. PhD thesis, Edinburgh University (1991)
20. Binder, K.: The Monte-Carlo method for the study of Phase Transitions: A review of some recent progress. J. Comp. Phys. **59** (1985) 1–55
21. NVIDIA® Corporation: NVIDIA CUDA^{TM} Programming Guide Version 2.3. (2009) Last accessed August 2009.
22. Leist, A., Playne, D., Hawick, K.: Exploiting Graphical Processing Units for Data-Parallel Scientific Applications. Concurrency and Computation: Practice and Experience **21** (2009) 2400–2437 CSTN-065.
23. Flanders, P., Reddaway, S.: Parallel Data Transforms. DAP Series, Active Memory Technology (1988)
24. Hawick, K.A., Playne, D.P.: Hypercubic Storage Layout and Transforms in Arbitrary Dimensions using GPUs and CUDA. Technical Report CSTN-096, Computer Science, Massey University (2009) Submitted to Concurrency and Computation: Practice and Experience.
25. Hawick, K.A., Playne, D.P.: Turning partial differential equations into scalable software. Technical report, Computer Science, Massey University (2009)
26. Marsaglia, G., Zaman, A.: Toward a universal random number generator. FSU-SCRI-87-50, Florida State University (1987)
27. Hawick, K.A., Leist, A., Playne, D.P.: Mixing Multi-Core CPUs and GPUs for Scientific Simulation Software. Technical Report CSTN-091, Computer Science, Massey University (2009)
28. Wolff, U.: Collective Monte Carlo Updating for Spin Systems. Phys. Lett. **228** (1989) 379