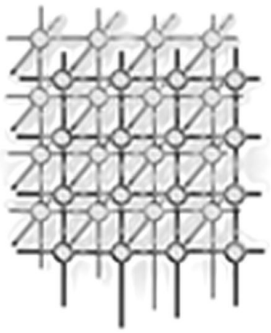
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2010; **00**:1–27 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

Hypercubic Storage Layout and Transforms in Arbitrary Dimensions using GPUs and CUDA



K.A. Hawick*and D.P. Playne†

Institute of Information and Mathematical Sciences, Massey University – Albany, North Shore 102-904, Auckland, New Zealand.

SUMMARY

Many simulations in the physical sciences are expressed in terms of rectilinear arrays of variables. It is attractive to develop such simulations for use in 1-, 2-, 3- or arbitrary physical dimensions and also in a manner that supports exploitation of data-parallelism on fast modern processing devices. We report on data layouts and transformation algorithms that support both conventional and data-parallel memory layouts. We present our implementations expressed in both conventional serial C code as well as in NVIDIA's Compute Unified Device Architecture (CUDA) concurrent programming language for use on General Purpose Graphical Processing Units (GPGPU). We discuss: general memory layouts; specific optimisations possible for dimensions that are powers-of-two; and common transformations such as inverting, shifting and crinkling. We present performance data for some illustrative scientific applications of these layouts and transforms using several current GPU devices and discuss the code and speed scalability of this approach.

KEY WORDS: data-parallelism; GPUs; CUDA; shifting; crinkling; hypercubic indexing.

1. INTRODUCTION

Data layout in memory of key variables is important for applications programs that make use of data-parallelism. Several well-known applications such as image processing and lattice models make use of rectilinear data that can be mapped closely to the parallel processing structure of a data-parallel computer. Many other applications and particularly numerical simulations in physics, chemistry and other areas of science and engineering are oriented around hypercubic data arrays where there is a high degree of potential parallelism to be exploited in the calculations. It is not always the case however that the logical data layout for the serial application program or from some file storage format is the

*Correspondence to: IIMS Massey University – Albany, North Shore 102-904, Auckland, NZ.
†E-mail: {k.a.hawick, d.p.playne}@massey.ac.nz

Received 23 June 2009
Revised 12 April 2010

optimal data-parallel layout for the target compute platform. Often a stencil-oriented algorithm, such as for image processing [1] or lattice-based Monte Carlo simulations [2,3], requires neighbouring adjacent data to remain unchanged during a synchronous update operation. Consequently some data-striding or array-crinkling is necessary to apply data-parallel concurrent updates.

Fine-grained data-parallelism is going through a revival period largely due to the low cost, wide availability and relative ease of programmability of processing accelerator devices such as Graphical Processing Units (GPUs) [4]. These devices can be programmed using a data-parallel approach using relatively high level languages such as NVIDIA's Compute Unified Device Architecture (CUDA) language. The term general-purpose GPU programming (GPGPU) [5] is now widely used to describe this rapidly growing applications-level use of GPU hardware [6, 7, 8].

Many of the ideas of data-parallelism were first invented and explored for the massively parallel, single-instruction multiple-data (SIMD) computers of the period between the mid 1970s and early 1990s. Computers such as the Distributed Array Processor (DAP) [9, 10], the Connection Machine (CM) [11] and MasPar [12] had software libraries [13, 14] with built-in routines for data layout transformations on regular rectilinear data structures. These machines often achieved notably high performance [15] on appropriately sized and shaped data structures [16] in scientific simulations [17] and related applications. The power of these machines also inspired the development of many data-parallel software libraries, kernels [18, 19] and programming languages [20, 21, 22, 23] that are still in use. We refer to the notion of data crinkling - a concept that appears to have its origins with array processing. The term comes from the notion of folding or crinkling a sheet of paper so that some elements are subsequently much closer together than before. Crinkling is an operation and as such described more than just data striding through memory. The DAP supported two dimensional crinkling [24] whereupon "normal" or serially laid-out data from the host computer's memory could be crinkled or uncrinkled to block it in different ways depending on what was optimal for the application's algorithms that ran on the array-processor accelerator [25].

Graphical Processing Units (GPUs) have much in common with computers such as the DAP, CM and MasPar. GPUs are programmed as special subroutines (known as kernels in CUDA) that are called from the conventional serial program running on the host processor. Thus an application consists of a CPU program linked to one or more GPU program components and will make use of various special purpose data allocation and transfer routines between CPU and GPU memory. Although there have been several programmer systems and tools for exploiting GPUs [26, 27, 28, 29], NVIDIA's CUDA language has emerged as one of the most powerful for scientific applications [30]. While CUDA is presently specific to NVIDIA hardware, the Open Compute Language (OpenCL) promises similar capabilities and applications programmer accessibility. Nevertheless, exploiting parallel computing effectively at an applications level remains a challenge for programmer. In this article we focus on data layout and storage transforms and explore the detailed coding techniques and performance optimisation techniques offered by CUDA [8] to build software for shifting, crinkling and re-indexing hypercubic data in arbitrary dimensions.

Managing optimal data layout on parallel computing systems is not a trivial problem [31] and in a complex application there may not be one single data layout that is optimal for all sub-algorithmic components of the code [32]. It is therefore valuable for a data-parallel system to have an appropriate library of data-transformation routines such as the parallel data transforms (PDTs) of Flanders and Reddaway [33]. A number of attempts have been made to optimise the PDT ideas using bit-wise operations that were customised for particular data sizes (such as powers-of-two) [34, 35] that naturally

fit onto particular hardware. Other work has also considered using indexing reshuffles to minimise memory usage by avoiding vacant elements in sparse or compressible structures [36]. Attempts have also been made to generalise the data transform ideas algebraically [37,38,39]. Most research went into optimising the ideas for two dimensional image data [40,41] on computers such as the MasPar [42,43] or DAP [44].

In this article we consider how the parallel data-transform ideas can be implemented using CUDA-like programming languages for performance, but also more generally in plain ordinary serial languages like C for portability reasons and for interoperability between CPU-Host/GPU-Accelerator program combinations. The multiple threads and grid-block model used for GPUs requires some important new approaches that depart from the indexing model that was optimal for computers like the DAP, CM and MasPar. The GPU memory model is in fact quite complex and comes in several different sorts with different layout and speed characteristics. We consider data that is logically treated as a hypercubic array of arbitrary dimensions, but which can be mapped to contiguous main memory for the CPU to access, yet suitably transformed to an optimal layout for an associated GPU to work with. We have investigated both dimensions that are powers-of-two and multiples of the GPU specialist memory length and size parameters, but also general unconstrained values.

In Section 2 we give a brief review of the GPU memory architecture. We outline the main data notation and hypercubic data layout ideas in Section 3 followed by a description of a generalised data indexing system in Section 4. We describe how the indexing can be optimised using bit-wise operations in Section 5 and discuss the issues and approaches to optimal data neighbour access and indexing in Section 6. In Section 7 we describe our CUDA implementation of the hypercubic indexing and also present code and algorithms for shifting and crinkling transforms in Section 8. We present some performance results using NVIDIA GeForce 9400/9600, 260 and 295 GPUs in Section 10. In Section 11 we discuss the implications for applications that use hypercubic storage transforms and offer some conclusions and areas for future optimisation work.

2. GPU ARCHITECTURE SUMMARY

Modern GPUs contain many processor cores that can execute instructions in parallel. A typical GPU (such as one of the NVIDIA GeForce 200 series) contains 16-60 multi-processors, each of which contains 8 scalar processor cores (giving a processor total of 128-480). These GPUs are capable of managing, scheduling and executing many threads at once using the SIMT paradigm [45]. To make use of the parallel computational power of a GPU, an application must be decomposed into many parallel threads. One difference between decomposing an application onto a common distributed or parallel architecture and a GPU is the number of threads required. GPUs provide the best performance when executing 1000+ threads.

These threads are organised as a grid of thread blocks. Each thread block contains a maximum of 512 threads and the grid size is limited to $\{65535, 65535, 1\}$. Each thread has a threadIdx which determines its id within a block and a blockIdx which determines the id of the block it belongs to. From these two identifiers, each thread can calculate its unique id.

GPUs also contain many memory types optimised for different tasks. These optimisations were developed for increased performance when processing the graphics pipeline but they can be easily

utilised for other purposes. Selecting the memory type best suited to the access pattern is vital to the performance of a GPU application [46], but equally as important is the memory access pattern used.

Global memory is the largest type of memory available on the GPU and can be accessed by all threads, unfortunately it is also the slowest. Access to global memory can be improved by a method known as **coalescing** [45]. If 16 sequential threads access 16 sequential (and aligned) memory addresses in global memory, the accesses can be combined into a single transaction. This can greatly improve performance of a GPU program that uses global memory.

Texture and **Constant** memory are both cached methods for accessing global memory. Texture memory caches the values in a spatial locality (near neighbours in the dimensionality of the texture) allowing neighbouring threads to access neighbouring values from global memory whereas constant memory allows multiple threads to access the same value from memory simultaneously. Each multiprocessor has its own texture and constant memory cache.

Shared memory is also stored on the multiprocessor but must be explicitly written to and read from by the threads in the multiprocessor. This memory is useful for threads that must share data or perform some primitive communication.

Registers and **Local** memory are used for the local variables used within the threads. The registers are stored on the multi-processor itself whereas local memory is actually just global memory but used for local variables. The threads will use the registers first and if necessary will make use of local memory, it is preferable if only the registers are used.

3. DATA LAYOUT ISSUES AND NOTATION

In this section we discuss the fundamental ideas behind expressing dimensions, shapes and layouts for hypercubic or rectilinear data arrays. We are considering a d-dimensional space $\mathbb{R}^d$, spanned by $\mathbf{r} = (r_0, ..., r_{d-1})$. In most cases we are interested in a physical space and $d = 2, 3$; however, in some cases we wish to simulate systems with $d = 4, 5$ and higher. Certainly the mapping problem of real space to data structure generally simplifies considerably when $d \equiv 1$ where we simply have an ordered list. A simple integer index i maps to the spatial region or 1-dimensional interval associated with $\mathbf{r}$ and in many cases the mapping is a simple one, with the interval spacing Δ being constant [47]. Thus we have:

$$r_i = r^0 + i \times \Delta, \forall i = 0, 1, ..., L - 1 \tag{1}$$

with the r^0 value chosen to determine the origin. However, when programming multi-dimensional spaces we typically want to use index i over the dimensions.

When working with real space with 2 or more dimensions, we define d-dimensions as: $r_{x_0}, ...r_{x_{d-1}}$. To work with these, we must employ some sort of spatial mapping between a discrete index-tuple $\mathbf{x_i}$ and the spatial cell or region associated with it.

$$\begin{aligned} r_{x_0} &= r^0_0 + x_0 \times \Delta_0, \forall x_0 &= 0, 1, ..., L_0 - 1 \\ r_{x_1} &= r^0_1 + x_1 \times \Delta_1, \forall x_1 &= 0, 1, ..., L_1 - 1 \\ &\dots \\ r_{x_{d-1}} &= r^0_{d-1} + x_{d-1} \times \Delta_{d-1}, \forall x_{d-1} = 0, 1, ..., L_{d-1} - 1 \end{aligned} \tag{2}$$

Often for a simple program the Δ_d values in each dimension are all the same value, perhaps even unity if the spatial units of the system modelled can be so adjusted. In all our examples we have made use of indexing from $0, ..., L-1$ as it is the generally accepted index methods by most modern programming languages.

Within a programming environment we can often define this d-dimensional space as a d-dimensional array. When defining a multi-dimensional array is it important to consider each programming language's built in multi-dimensional array indexing approach to allow for the best memory paging access. In Fortran we could implement `REAL SPACE(0:L1-1, 0:L2-1, 0:L3-1)` where the data is laid out in memory column major so the first index is fastest changing whereas in C/C++ we would declare: `double space[L3][L2][L1]` where we have had to reverse the order of dimensions so that index 0 is fastest changing in memory, based on the row-major ordering used by C.

One disadvantage of representing a multi-dimensional space as a multi-dimensional array is the significant restructuring process required when the number of dimensions is changed. Instead there is a more general method of representing multi-dimensional space. We want an unambiguous notation to specify these transformations and also an efficient set of algorithms to perform these operations, either in place or as a fresh data-set copy. Ideally the notation should map well to the array or operational syntax of a programming language. We also want to be able to perform combination operations on the transformations to make composites that can be executed in a single operation, again preferably in parallel.

$$[L_2, L_1, L_0] \tag{3}$$

implies data that is 3-dimensional $(d \equiv 3)$ with lengths L_i in each dimension.

We sometimes need to explicitly consider the type of the data. This specifies the size or storage requirement of each individual data element. We might think about it as unambiguous bits, and so write it as

$$[128, 128, 128, 32] \tag{4}$$

for an array of 32-bit ints for example. We could also write the type using some mnemonic such as:

$$[128, 128, 128, \text{int32}] \tag{5}$$

and we could treat this as

$$[128, 128, 128, 4, 8] \tag{6}$$

where we have 4 bytes per int and of course 8 bits per byte.

Although this notation might seem verbose or clumsy, it is useful because it is completely unambiguous - a bit is a bit is a bit and cannot be mistaken for something else, whereas a word or even the byte order in a word can differ between machine implementations. This ambiguity has been a barrier to portable use of some of the data-transform algorithms and implementations supplied by SIMD computer vendors in the past.

The notation implies a minimal sequential storage size

$$N = \Pi_{i=0}^{d-1} L_i \tag{7}$$

and we can think of the data as indexed in a fundamental way using a k-index where $k = 0, 1, ..., N-1$ with the usual convention for least significant dimension to the right, most significant dimension to the left.

This k-index idea is similar to the single array reference idea mentioned in [48] where Ciernak and Li use it to restructure linear algebra operations instead of using two array indices to access matrices. The idea can be generalised however so that we can write arbitrary dimensional simulations using a single k-index and therefore do not need to know how many explicit nested loops to write in the program. We can also ascribe spatial encoding information to the k-index as we discuss below in section 4.

We can combine these specifiers by multiplication and we get the same implied resulting dimension and storage size regardless of associative order. The interpretation is different however. It is useful for example to drop the data type part and decompose:

$$[128, 128, 128, 32] \mapsto [128, 128, 128] \cdot [32] \tag{8}$$

or $\mathcal{D}_2 \cdot \mathcal{D}_1$. We can then do operations or algebra on $\mathcal{D}_2$, while temporarily ignoring the notion that each individual element is of type $\mathcal{D}_1$.

We want to be able to apply transformations or remappings to the data to be able to interpret it in different ways. So data set $\mathcal{D}_2$ that is in layout $\mathcal{L}_1$ can be transformed into a different layout $\mathcal{L}_2$ with the same fundamental size or it can be filtered or reduced to a smaller data set. We might also allow duplications or insertions of padded zeros into the source for example and thus arrive at a larger target data set.

These general ideas lead onto a specific approach using a single compressed index into sequential contiguous storage space. This approach is well known and widely used in various applications, but does not seem to have a widely used name. We refer to it as k-indexing using a single integer k that compresses all information about all the data dimensions. It is particularly useful for writing simulation codes in arbitrary dimension without embedding a fixed number of nested `for`-loops in the source code.

4. HYPERCUBIC k-INDEXING

The technique we call k-indexing makes use of integer calculations within the user's application programming to index into a single-dimensional array or block of memory. In the case of a dense and regular space we can readily map between our model indices $x_i, i = 0, 1, ..., d-1$ and a memory index k which is solely a memory addressing index.

It is convenient to adopt notation so that each dimension is labelled by $i = 0, 1, ..., d-1$ and the lengths of the regular space, in units of the appropriate Δ_i, for each dimension are L_i such that x_i is in the range $[0...L_i - 1]$. Thus we can calculate the k-index completely generally from:

$$k = x_0 + L_0 \times x_1 + L_0 \times L_1 \times x_2 + L_0 \times L_1 \times L_2 \times x_3 + \cdots \tag{9}$$

We can decompose a k-index into individual dimensional indexes by an orthogonalisation process, assuming we have available a modulo operation (denoted throughout this article by %.)

$$x_i = \frac{k}{L_0 \times L_1 \cdots L_{i-1}} \% L_i \tag{10}$$

This is more complicated than it looks but can be implemented as an iteration over the dimensions:

Listing 1. Construction of a k-index from individual indexes and decomposition into indexes.

```
// construct a k-index from an index-vector
int toK( int *x ) {
  int k = 0, LP = 1;
  for( int i = 0; i < d; i++ ) {
    k += x[i] * LP;
    LP *= L[i];
  }
  return k;
}

// decompose a k-index into an index-vector
void fromK( int *x, int k ) {
  int LP = N;
  for(int i = d-1; i >= 0; i--) {
    LP /= L[i];
    x[i] = k / LP;
    k = k % LP;
  }
}
```

Alternatively we could pre-compute the L_d products as used in Equation 10 and these could available in a globally accessible array `LP[]`, bound to the hypercubic data set in question. We implement this as LP$[i] = \prod_{j=0}^{i} L_j$ so that LP$[i]$ is the stride (for normal data layout on the serial host memory) in dataset dimension i. For example, a data set of size $N = 64 = 4 \times 4 \times 4$ elements would have LP $\equiv \{1, 4, 16, 64\}$.

Also note our convention that our index-vector is indexed both in the programming language and the formulae given here using dimension $i = 0, 1, ..., d-1$. The k-index formed runs from $0, ..., N-1$ where $N = \prod_i L_i$ is the total number of spatial cells in our dense representation of space.

Figure 1 shows an indirect addressing scheme that is useful for hyper-cubic symmetry lattice models in arbitrary dimension d. The lattice lengths $L_i, i = 0, ..., d-1$ in each dimension are fixed and hence cell positions $x_i, i = 0, ..., d-1$ on the lattice can be encoded as a single integer $k = x_0 + x_1 \times L_0 + x_2 \times L_1 \times L_0$ and so forth. In this particular example the model payload is a single bit - such as in a spin model - and these k-indices or "pointers" can be used to identify the location of the m'th member particle in the j'th cluster that has been identified in this model configuration. The k-indices are convenient data pointers, so a cluster can be described compactly as a list of all the k-index values of its member "particles" with the spatial position information of particles thus conveniently compressed. This can be convenient for optimal memory utilisation for application algorithms that do component labeling [49] of a model.

A dual representation using both x-vectors and k-indices is remarkably powerful, since it is easy to formulate fast iterations over all cells using the k-index loop variable and yet also to randomly select individual elements to "hit" or update in a simulation model with some dynamical update scheme. It is also relatively easy to access an arbitrary cell indexed by arbitrary x-vector, by first composing it into its corresponding k-index and to construct the neighbouring cells from a chosen cell by its x-vector or k-index. Most powerfully however it means we can write a multi-dimensional simulation code without

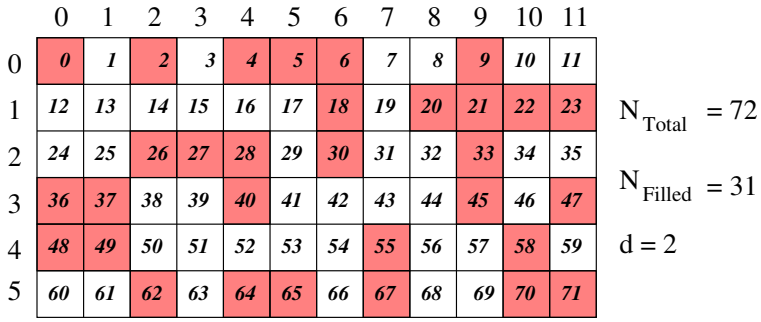


Figure 1. In a 12×6 system, where L[0]=12 and L[1]=6, we assign each lattice point a unique value called a 'k' value. Simple functions can convert between 'k' values and traditional (x_1, x_2) vectors. Using 'k' values in the body of our code removes any dependencies of the code on the actual dimensionality of the simulation being studies. In this model example the payload in each cell is just a single bit – shaded boxes represent those sites which are filled and white boxes are empty.

knowing how many nested for-loops to write - which we would need if we had to iterate of each x-vector component in turn.

The k-index approach also gives us a contiguous memory baseline for constructing transform operations between different layouts in CPU-Host memory and GPU-Accelerator memory. The equations above are expressed in terms of generalised or unconstrained array dimension lengths. If these are constrained to powers-of-two, then some even faster optimisations of the re-indexing calculations are possible.

5. POWERS-OF-TWO AND BIT OPERATIONS

In many simulations, the length of the data array does not particularly matter *a priori* and can be constrained to be a power of two. This is advantageous for many reasons: ease of Fourier transforms on a 2^N matrix, memory block alignment (important for CUDA). Power-of-two field lengths make some convenient optimisations possible for k-indexing and related approaches [33]. When the field length is a power of two, the bits of the k-index can be split to represent the indexes for the separate dimensions. This allows us to perform operations such as: extract dimension indexes, combine indexes to a k-index and calculate neighbours with the use of bit operations.

To extract a specific dimensions index as well as combine indexes back together into a single k-index, we need to compute several masks. We need to know how many bits each dimension's index will require, the total bits used by the indexes for lower dimensions as well as a mask that can be applied that will have a value of 1 for all bits that are used by the index. First of all we need to know how many bits of the k-index are required to store the index for each dimension which can be easily computed from the dimensions of the field (**L** is an array of the dimensions of the field which must be

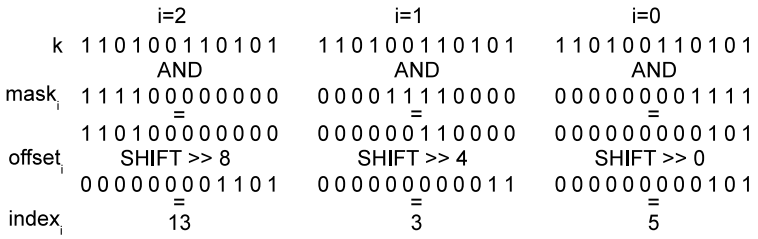


Figure 2. A illustration of the bit operations used to extract the indexes of individual dimensions from a k-index. Note that only the bits used by the indexes (the first 12 bits) are shown as all others remain 0 at all times.

specified by the user) using the equation: $\text{bits}_i = \log_2(\text{L}_i)$. We also wish to know the total number of bits used by the dimension indexes below us. This can simply be calculated by:

$$\text{offset}_i = \sum_{j=0}^{i-1} \text{bits}_j \tag{11}$$

We need a mask where each bit is 1 only if the corresponding bit in the k-index is used for the desired dimension. This can be calculated by taking the size of each dimension and subtracting 1 from it (so that all the bits used by the index are 1) and then shifting it to the left by the offset of the dimension. See Equation 12.

$$\text{mask}_i = (\text{L}_i - 1) \ll \text{offset}_i \tag{12}$$

These values can be pre-computed for each dimension during the initialisation of the program. They can then be used for the extraction and combination of indexes and k-indexes. Using the mask and offset values we can easily extract an individual index using an AND and a SHIFT bit operation. Take for example a field with the dimensions 16x16x16. Using Equations 11 and 12 we calculate the following sets:

$$\textbf{bits} = \{\textbf{4, 4, 4}\}\ \textbf{offset} = \{\textbf{0, 4, 8}\}\ \textbf{mask} = \{\textbf{0x00F, 0x0F0, 0xF00}\}$$

With these we can extract the index of any dimension with the simple equation (note that in this paper we use ! for NOT, & for AND and | for OR):

$$\text{index}_i = (\text{k}\ \&\ \text{mask}_i)\ \gg\ \text{offset}_i \tag{13}$$

We can calculate a k-index from a given set of indexes in a similar (but reversed) process. The equation for recombining indexes is:

$$\text{k} = \sum_{i=0}^{d} \text{index}_i \ll \text{offset}_i \tag{14}$$

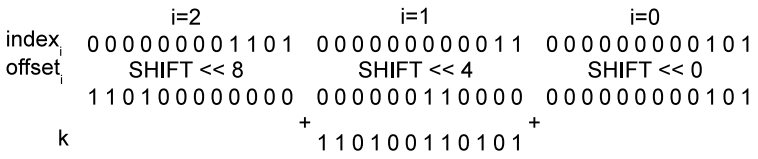


Figure 3. A diagram showing the process of recombining individual indexes into a k-index using bit operations. Note that once again only the bits used by the indexes are shown, all other bits remain 0.

The exact process for extracting individual indexes and recombining them can be seen in Figure 2 and Figure 3. These figures show the extraction of the z, y and x index for an example k-index and the reconstruction of the k-index from the individual indices. Note that the bits, offset and mask values are the same as given previously.

Within our C or CUDA simulations, these operations can be written as preprocessor macros that make use of pre-computed mask values. This allows us to extract and combine k-index values very quickly.

6. INDEXING AND NEIGHBOUR PROXIMITIES

Some algorithms are expressed entirely in terms of highly localised neighbour-neighbour interactions and in the case of a lattice or mesh model where the positions of neighbours is both known and unchanging, then k-index values for a cell's neighbours can be pre-computed and stored. Individual models will have different "degrees of freedom" to use the terminology common in the physics literature. For our purposes the degree of freedom (dof) is simply the data type and size (in terms of bytes) that resides on each hypercubic cell for the particular application. The degrees of freedom objects that are modelled on our dense space can in fact be implemented as various arrays, indexed by k-index values, or in Object-Oriented languages can be compound objects with all the relevant variables bundled within. Our hypercubic apparatus is then a framework to manage these dof entities in varius memory layouts appropriate to the serial host memory or a specific device memory such as one of the GPU memory layout models. The power of this approach is that once tested we do not need to reinvent the memory layout for each new application model and that when new compute devices such as GPUs or some future new array parallel accelerator become available it is straightforward to re optimise the application data layout in memory with trivial code changes

It is convenient for the unchanging neighbour model to embed a neighbour list of pre-computed k-index values within each degree-of-freedom object. The system is then implemented as a flat one-dimensional array of degree of freedom objects, the array being indexed by k-index value.

Typically in a model we need to address our degree-of-freedom object: either as a list so we apply some operation (an initialisation or a measurement) to all of them; or by spatial position so we can initialise or measure all the dofs within a chosen region of space. The former operation is done by looping over k-index; the latter by looping over each i-index component in turn.

In most models of our experience, the algorithm to update the dofs is specified in terms of spatial loop(s) while the dof initialisation or measurements are often made by flat all-inclusive k-index loop. Since the update algorithm is nearly always executed much more often during a "run" and will generally be more computationally expensive, it therefore makes sense to optimise a code from the perspective of the spatial looping. The remainder of this paper considers ways to build up algorithms or pre-computed data that will address optimal spatial looping.

6.1. Generalised Neighbour Index Calculations

In the general case we can calculate neighbouring positions with simple process. Extract the index, increment it will appropriate boundary handling conditions and then recombine it into a single k-index. Listing 2 shows a function that can calculate a neighbouring position in any dimension.

Listing 2. The calculation of a neighbouring k-index in the general case. Note that this method applies periodic boundary conditions.

```
int general_neighbour(int k, int i, int inc) {
  int *x = new int[d];
  fromK(x, k);
  x[i] = (x[i] + inc + L[i]) % L[i];
  return toK(x);
}
```

As with the previous extraction and combination operations, this process can be improved with the use of bit operations when the field length is a power of two.

6.2. Calculating Hypercubic Neighbour Positions using Bit Operations

For a hypercubic mesh, we can make use of bit operations to calculate the k-index of neighbouring cells. One major advantage of using bit operations to compute the neighbours is that the k-index does not need to be decomposed into separate indexes and then recombined. We can also handle boundary conditions directly within a simple bit expression.

We wish to define an operator that can calculate the k-index of a cell any distance away in any dimension. Periodic boundaries are a commonly used method to handle boundary conditions, we describe a logical expression that automatically wraps round should a neighbour calculation trigger a boundary condition. To do this we require another mask that is 1 for every bit that is used by the indexes of the dimensions other than the current one. This can be calculated by:

$$\text{inverse}_i = (N-1) - \text{mask}_i \tag{15}$$

With the example from Section 5 the sets would be calculated as:

bits = {4,4,4} offset = {0,4,8} mask = {0c00F, 0x0F0, 0xF00}
inverse = {0xFF0,0xF0F,0x0FF} LP = {0x001,0x010,0x100}

We can now describe an expression for calculating the k-index of a neighbour. The following expression gives the k-index of the n^{th} neighbour of k in the i^{th} dimension:

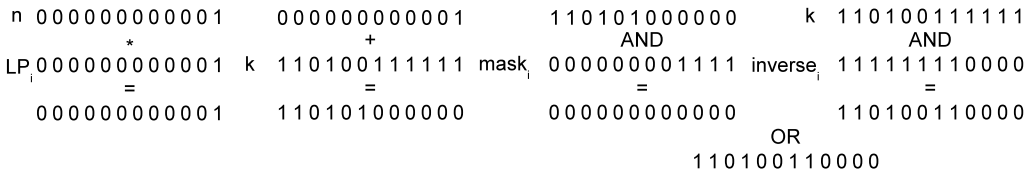


Figure 4. A diagram showing the process of calculating a neighbouring position in a hypercubic mesh using bit operations. This example calculates the neighbour in the +1 direction for the 0^{th} dimension.

$$\mathrm{k}_{i,n} = ((\mathrm{k} \; + \; (\mathrm{n} \; * \; \mathrm{LP}_i)) \; \& \; \mathrm{mask}_i) \mid (\mathrm{k} \; \& \; \mathrm{inverse}_i) \tag{16}$$

The workings of this expression calculating the $+1$ neighbour in the 0^{th} dimension can be seen in Figure 4.

The bits representing the indexes for $i = \{1, 2\}$ have remained unchanged while the index for $i = 0$ has been incremented, overflowed and suitably wrapped around. This expression works correctly for any value of i (up to the number of dimensions) and any value of n (positive or negative).

7. k-INDEXING IN CUDA

Making use of k-indexing within the CUDA environment is complicated by the fact that GPUs are highly optimized for image processing tasks and are thus best-suited to processing two-dimensional data. We can build a k-index from the block and thread indexes (see Section 2) so we can process a data-set regardless of the number of dimensions it has. The easiest way to do this would be to simply create the threads using only the x component of the thread and block indexes. This unfortunately is not optimal as the maximum k-index that can be created using only the x components is: $(512 * 65536) - 1 = 2^{25} - 1$. While this may be sufficient for many problems, our k-indexing method should be capable of accessing the maximum size possible.

To overcome this problem we must also use the y component of the block index. If the maximum value of the y component is limited to 128, then the maximum k-index possible is $(65536 * 128 * 512) - 1 = 2^{32} - 1$. This is the maximum possible value for the index and provides the functionality desired. Within these bounds there is a degree of flexibility, in some cases we may wish the number of threads within a block to be less than 512 (for memory reasons) in which case we can simply increase the maximum size of the y-component of the grid. Any maximum size of suitable provided that maximum size of the grid (gridDim.x and gridDim.y) and the maximum size of the block (blockDim.x) fulfill the following condition:

$$\mathrm{gridDim.x} \times \mathrm{gridDim.y} \times \mathrm{blockDim.x} = 2^{32} \tag{17}$$

Given these values we can calculate each thread's (local) k-index with:

$$k = (((\text{blockIdx.y} \times \text{gridDim.x}) + \text{blockIdx.x}) \times \text{blockDim.x}) + \text{threadIdx.x} \quad (18)$$

Once this k-index has been calculated we can then make use of the k-indexing techniques we have previously discussed in Sections 4,5 and 6. These k-indexes can index linear memory that represents a data-set with an arbitrary number of dimensions.

In a previous work [46] we found that for regular (unpadded) data such as we refer to here, it was definitely best to assign a single thread to each cell. This is not necessarily the case for algorithms that require data padding where other combinations are possible and will be more application-specific.

7.1. Texture Memory Access

Many simulations that perform neighbour operations on hypercubic meshes (which can make use of the k-index neighbour bit operations) have the best performance when using texture memory. Texture memory is a memory cache similar to constant memory with the difference what when a value is access the texture cache will load that value and all the values in the same spatial locality. This has been shown to be the optimal memory type for several applications processing hypercubic meshes [46, 49].

CUDA supports texture access for one-, two- and three-dimensions (although the code for three-dimensional texture is markedly more complex [50]). This presents the obvious problem that k-indexes designed for an arbitrary number of dimensions cannot easily work with texture memory that is limited to three-dimensions. While there is no way to overcome this problem entirely, it is possible to provide functions that can minimise the change required to move between texture and global memory access.

It remains the decision of the programmer whether to use texture or global memory, but to avoid undesirable changes to the code we can define access and initialisation functions that can easily be set to use global or texture memory. Global memory can be accessed using simply the k-index value, but a texture memory access requires the k-index to be split into individual indexes. The following functions show how the same call can be used to access data from global memory as well as one-, two- and three-dimensional textures (note these are implemented as `#defines` to create the fastest code possible).

```
#if defined GLOBAL
#define access(k,data) data[k]
#elif defined TEXTURE1D
#define access(k,data) tex1D(data, k)
#elif defined TEXTURE2D
#define access(k,data) tex2D(data, index(k, 0), index(k, 1))
#elif defined TEXTURE3D
#define access(k,data) tex3D(data, index(k, 0), index(k, 1), index(k, 2))
#endif
```

8. COMMON LAYOUT TRANSFORMATIONS

There are a number of commonly used transformations, that preserve data size that we consider first.

It is illustrative to use images to explain the ideas, but of course the fundamental notion behind these hypercubic transformations is that they apply to arbitrary dimensional data.

8.1. Mesh Inverting

Inverting or Flipping a mesh is useful for many applications, image processing, symmetry transforms and reflections. Performing these flips can be done easily and efficiently with GPUs and k-indexing. This transformation can be performed by each thread reading from one k-index and then writing to another. The first k-index to read from can be calculated as per normal (see Equation 18) and the second is this k-index flipped in the appropriate dimension. This could be done by extracting the dimension's index, inverting it and then recombining it back into the k-index. However, we can define a simple bit operation to calculate the k-index of the thread when the field is flipped in the i dimension:

$$\mathrm{k}_f = ((!k) \mid \mathrm{inverse}_i) \;\&\; (k \mid \mathrm{mask}_i) \tag{19}$$

This method can be used to flip a mesh in any dimension, however there is a performance enhancing technique we can use when flipping in the $i = 0$ dimension. CUDA performs coalesced reads/writes when sequential threads access sequential addresses in memory. This is the case when reading the values as the threads are assigned sequential addresses in the mesh. When the field is flipped in the $i >= 1$ case then the threads themselves will still write to sequential addresses because their $i = 0$ order will remain unchanged.

However, when flipped in the $i = 0$ case, the threads will write to sequential address but in the wrong order (back to front). To overcome this problem we can make use of shared memory. When the threads read the values from the field they can then write them into shared memory. The threads must then synchronise with the rest of the block and then effectively flip the values from shared memory. Then threads must also calculate a k-index that is flipped for the dimension index and also flipped at the block level. In this way the threads can read sequential values, swap values around in shared memory and then also write sequential values to the output. The code to perform this can be seen in Listing 3.

Listing 3. CUDA kernel to reverse the mesh in dimension i. If $i = 0$ the kernel will use shared memory to ensure coalesced global memory access.

```
#define reverse(k,i) (((~k) | inverse[i]) & ( k | mask[i]))

__shared__ int buffer[BLOCK_SIZE];
__global__ void flip( int* g_odata, int* g_idata, int i) {
  //The normal k-index to read from
  int k_read = (((blockIdx.y * gridDim.x) + blockIdx.x) * blockDim.x)
               + threadIdx.x;

  if(i == 0) { //The k-index of the thread with a reversed block
    int temp = (((blockIdx.y * gridDim.x) + blockIdx.x) * blockDim.x)
               + ((blockDim.x-threadIdx.x)-1);
    //The k-index of the thread with a reversed block and reversed dimension
    int k_write = reverse(temp, i);
    buffer[threadIdx.x] = access(k_read, g_idata);
    __syncthreads();
    g_odata[k_write] = buffer[ (blockDim.x - 1) - threadIdx.x ];

  } else {  //The k-index of the thread with a reversed dimension
    int k_write = reverse(k_read, i);
    g_odata[k_write] = access(k_read, g_idata);
```

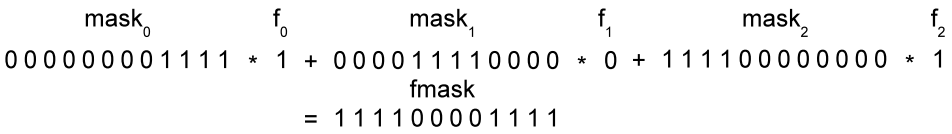


Figure 5. An example showing the construction of a flip mask (fmask) from a flip vector (f). This flip mask will flip the mesh in the 0^{th} and 2^{nd} dimensions.

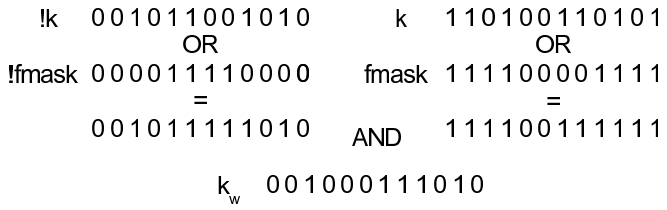


Figure 6. An illustration of a k-index being flipped in the dimensions defined by a flip mask (fmask). In this example the k-index is flipped in the 0^{th} and 2^{nd} dimensions.

```
    }
}
```

This method works well for flipping a field in a single direction. However, if a field must be flipped in multiple dimensions we must call this kernel multiple times. A better method is to develop a kernel that can flip a field in multiple dimensions in a single operation. As we can flip the index of a dimension simply by inverting it, we can define a mask that defines which indexes we wish to flip. This mask is best defined at the host level where we can make use of **mask** to define the flip mask (**fmask**). If we define a set **f** which is contains 1 for the dimensions we wish to flip and 0 for the ones we wish to left as they are we can define the **fmask** as:

$$\text{fmask} = \sum_{i=0}^{d} \text{f}_i \times \text{mask}_i \tag{20}$$

If we define **f** as $\{1, 0, 1\}$ then it indicates we wish to flip the field in the 0^{th} and 2^{nd} dimensions. Given the previous values of **mask** {0x00F, 0x0F0, 0xF00}, Equation 20 will calculate that fmask $=$ 0xF0F (See Figure 5).

The kernels can then calculate their k-index flipped according to this mask with (See Figure 6 for an example working):

$$\text{k}_w = ((!\text{k}) \mid (!\text{fmask})) \;\&\; (\text{k} \mid \text{fmask}) \tag{21}$$

The issue of coalesced memory access still applies when flipping in the 0^{th} dimension and can be overcome using the same shared memory buffer approach. The host code to generate the flip mask and the device code that flips a field in multiple directions can be seen in Listing 4.

Listing 4. Host code to create a flip mask and the CUDA kernel that uses the flip mask to flip the mesh in multiple dimensions in one operation.

```
int flipToK(int *flip) {
  int f_mask = 0;
  for(int i = 0; i < d; i++) {
    if(flip[0] == 1) {
      fmask = fmask | h_mask[i];
    }
  }
  return f_mask;
}

#define flipK(k,fmask) (((~k) | (~fmask)) & (k|fmask))

__shared__ float buffer[BLOCK_SIZE];
__global__ void flip( float* g_odata, float* g_idata, int fmask) {
  int k_read = (((blockIdx.y * gridDim.x) + blockIdx.x) * BLOCK_SIZE)
               + threadIdx.x;
  if((mask[0] & fmask) == mask[0]) {
    int temp = (((blockIdx.y * gridDim.x) + blockIdx.x) * BLOCK_SIZE)
               + ((BLOCK_SIZE-threadIdx.x) -1);
    int k_write = flipK(temp, fmask);
    buffer[threadIdx.x] = access(k_read, g_idata);
    __syncthreads();
   g_odata[k_write] = buffer[(BLOCK_SIZE - 1) - threadIdx.x];
  } else {
    int k_write = flipK(k_read, fmask);
    g_odata[k_write] = access(k_read, g_idata);
  }
}
```

8.2. Cyclic or Planar Shifts

Another common transform is shifting the field in one or more dimensions with periodic boundaries, an operation reminiscent of **pshifts** and **cshifts** on the DAP [33]. This is useful for situations such as shifting FFT images etc. To do this each thread must read in the value at its k-index, calculated the shifted k-index and then write to this address. This can be performed easily by using the periodic neighbour operation we have previously described in Section 6.

If we define a set representing the number of cells to shift in each dimension, we can calculate a k-index that represents the shift. This will be the new k-index of the 0^{th} element. As a k-index cannot contain a negative index, we must ensure that the values are all positive (effectively shifting the field around the field the other way).

The kernel to perform a shift in multiple dimensions along with the host code to calculate the k-index from a set of shifts can be seen in Listing 5.

Listing 5. Host code to construct a shift k-index and the CUDA kernel that shifts the mesh.

```
int shiftToK(int *shifts) {
  int k_shift = 0;

  for(int i = 0; i < d; i++) {
    if(shifts[i] < 0) {
      shifts[i] = h_L[i] - shifts[i];
    }
    k_shift = k_shift | (shifts[i] << h_bits[i]);
  }
  return k_shift;
}

__global__ void shift( float* g_odata, float* g_idata, int k_shift) {
  int k_read = (((blockIdx.y * gridDim.x) + blockIdx.x) * BLOCK_SIZE)
               + threadIdx.x;
  int k_write = k_read;
  for(int i = 0; i < d; i++) {
    k_write = periodic_neighbour(k_write, i, index(k_shift,i));
  }
  g_odata[k_write] = access(k_read, g_idata);
}
```

8.3. Crinkling or Sub-tiling

In the context of single dimensional data on vector processors, the terms gather/scatter are roughly analogous to crinkling/uncrinkling in arbitrary dimensions on an array processor or in a generalised data-parallel context.

Crinkling a data-set is used to re-arrange data within an array to allow for optimal access patterns to maximise the performance of data-parallel processes. The method in which the data-set is crinkled will depend on the device, in this section we describe a method for crinkling the data-set for a GPU. Crinkling can be performed in any dimension i with any step size n (however it is advisable to crinkle with a step size that divides evenly into the mesh size). This operation has taken the name crinkle as it is similar to the concept of folding or crinkling a piece of paper. A crinkle operation with i, n will rearrange the mesh in dimension i such that every nth element will be stored sequentially.

We can crinkle a mesh in any dimension using CUDA, each thread will read the value at its k-index, calculate its crinkled k-index and then write this value to the output. The calculation of a crinkled k-index (k_c) crinkled in any dimension i with a step of n can be calculated with the following equation:

$$\mathrm{k}_c = (((\mathrm{k}\&\mathrm{mask}_i)\%(\mathrm{n} \ll \mathrm{offset}_i)) \times \frac{\mathrm{L}_i}{\mathrm{n}}) + (\frac{\mathrm{k}\&\mathrm{mask}_i}{\mathrm{n} \ll \mathrm{offset}_i} \times \mathrm{LP}_i) + (\mathrm{k}\&\mathrm{inverse}_i) \qquad (22)$$

This equation can calculate the k-index crinkled in any dimension, with any step size for a mesh with any number of dimensions. An example of crinkling in $i = 0, 1$ with a step size of 2 for a 4x4 mesh can be seen in Figure 7.

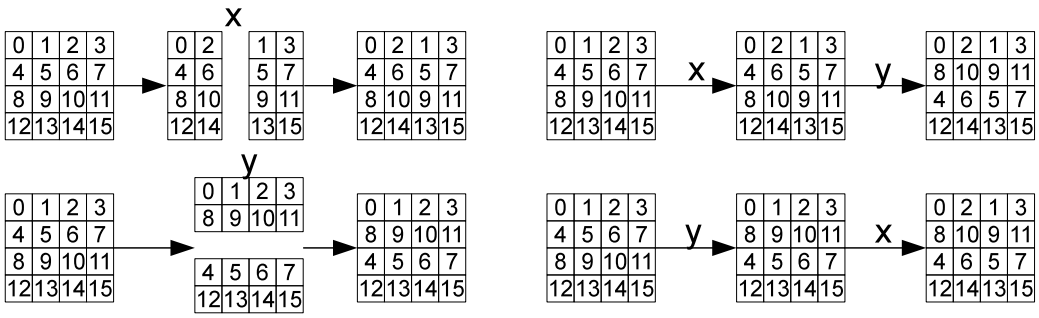


Figure 7. Crinkle operations on a 4x4 mesh. On the left, the mesh crinkled in the $i = 0$ (above) and in the $i = 1$ (above) showing the separate arrays and then their combination into a single array.

Figure 8. Our initial image of Lena (left) and the image crinkled with a step size of $n = 2$ in dimension $i = 0$ (middle) and $i = 1$ (right).Note the one-pixel width line superimposed over the image, this shows how some pixels are split by the crinkling. This image is arranged as a simple [128][128] image, this resolution was purposely chosen so that the individual pixel data can be clearly seen.

Examples showing the rearrangement process that takes place can be seen in the following test images. Our initial image has a single pixel width line superimposed over the image. This is to show the effects of the sampling process of the crinkling operation.

First we see the image crinkled in the $i = 0$ and $i = 1$ cases with step sizes of $n = 2$. The images appear as multiple copies of the same image, they do in fact contain entirely different pixels. The appearance of the images is an effect of the crinkling as it samples the original with a width of the step size and packs the output into a single array. This can be seen by the absence of the white line in some of the images. Since the white line is only one pixel thick, the images where the step size does not coincide with the line miss it out completely.

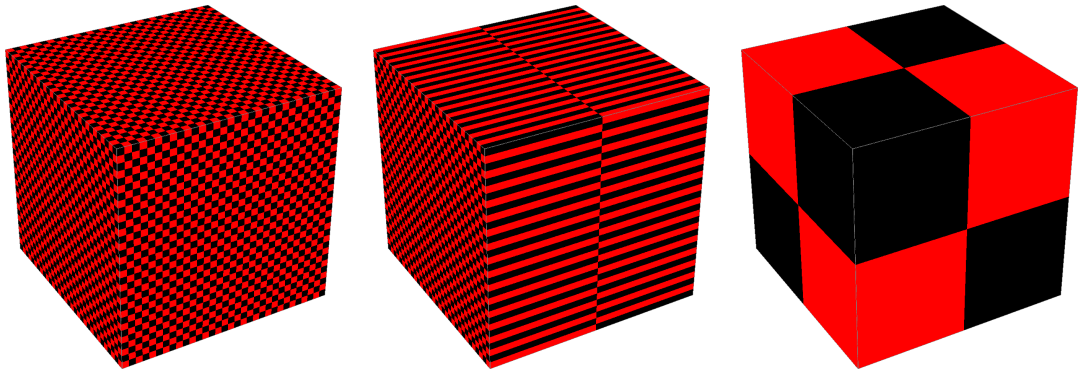


Figure 9. A three-dimensional cube coloured with a checkerboard pattern as it would be updated in an Ising model checkerboard update (left) and the same cube after it has been crinkled in the with $n = 2$ and $i = 0$ (middle) and in all three dimensions (right). The crinkled cube can be accessed more efficiently by CUDA as cells updated at once are stored sequentially.

Crinkling in itself is not a particularly useful process, however the rearrangement of the mesh can greatly improve the performance of some applications. The use of crinkling is described in Section 9.

8.4. Uncrinkling

Simulations on a crinkled data set is only useful if the data can be uncrinkled once the simulation has finished. Uncrinkling operations can be easily performed by simply reversing the k-indexes in the crinkle operation. Each thread calculates its k-index and its crinkled k-index using the Equations 18 and 22. However, unlike the crinkle operation, the threads read from the crinkled k-index and then write to the normal k-index. This process will undo the crinkle process or uncrinkle the data set. This is an easy process to test as a crinkle-uncrinkle operation will simply produce the original data set.

9. ISING SIMULATIONS WITH k-INDEXING

The Ising model [51, 52, 53] is a well-known model of magnetic spins used in the study of phase transitions. The number of dimensions the Ising model is simulated in has a drastic impact on the behaviour of the model. In one-dimension this model exhibits no phase-transition, two-dimensions phase-transition appears at the analytically calculated critical temperature and in three-dimensions around the experimentally determined critical temperature. The model is described by how the nodes interact with their nearest neighbours, according to a Hamiltonian or energy functional of the form:

$$H = -\sum_{i,j} J_{ij} S_i S_j \tag{23}$$

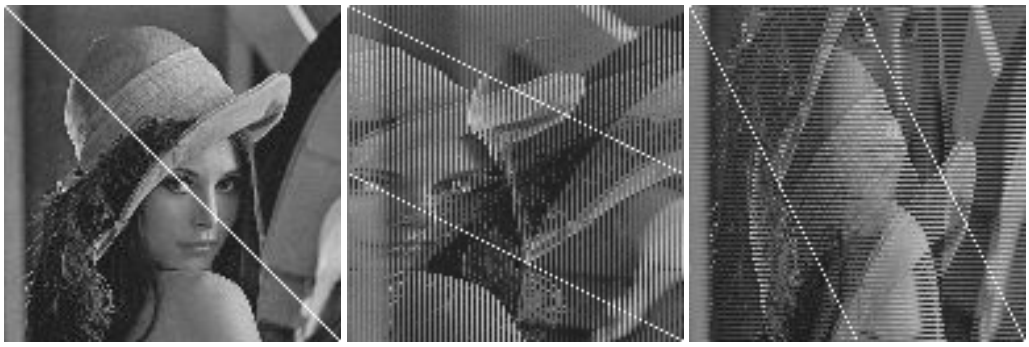


Figure 10. The same initial image as Figure 8 (left) and the image uncrinkled with step size $n = 2$ in dimension $i = 0$ (middle) and $i = 1$ (right).

We are not concerned here with the behaviour of the model which we describe elsewhere [54], but rather how it can be implemented easily using k-indexing techniques. The main advantage that k-indexing can provide is that the simulation can be written for an arbitrary number of dimensions. That is the same piece of code can be used for executing the simulation with any number of dimensions. This removes the need for maintaining multiple code versions for different numbers of dimensions.

The major difference in the different code for accessing neighbours and calculating their positions. When using k-indexing this can be easily replaced by a simple loop to calculate the neighbours in any number of dimensions. The loop to access the neighbours for this multi-dimensional code is shown in Listing 6.

Listing 6. Code that to access a cell's nearest neighbours in an arbitrary dimension hypercube.

```
for(int i = 0; i < d; i++) {
  kn = periodic_neighbour(k, i, -1);
  ...
  kn = periodic_neighbour(k, i, +1);
  ...
}
```

When simulating the Ising model on a parallel machine such as the GPU, the checkerboard update scheme is commonly used. This scheme updates alternating cells in the mesh such that no two neighbouring cells are updated at once (thus the term checkerboard). However, GPUs (and other parallel devices) perform best when reading/writing sequential values from memory, this is where crinkling becomes useful. When the mesh is crinkled, the cells belonging to each half of the checkerboard. A crinkled mesh will store the update cells sequential and the neighbouring cells sequentially which will improve performance when reading the values from memory.

The k-indexing implementation of on the GPU can simply make use of the neighbour and crinkle operations to calculate the position of its neighbouring cells in crinkled memory. The neighbours are calculated followed by their crinkled positions. This is shown in Listing 7.

Listing 7. Code to access the nearest neighbours in an arbitrary dimension hypercube which as been crinkled in $d = 0$ with a step size of 2.

```
//Spin to update
spin = mesh[crinkle(k,0,2)];

//Neighbours
for(int i = 0; i < d; i++) {
  kn = periodic_neighbour(k, i, -1);
  kn = crinkle(kn,0,2);
  neighbour = access(kn, mesh);
  ...
  //update model payload based on application
  ...
  kn = periodic_neighbour(k, i, +1);
  kn = crinkle(kn,0,2);
  neighbour = access(kn, mesh);
  ...
  //update model payload based on application
  ...
}
```

For architectures like the GPU, having to loop through the neighbours is not optimal, luckily we can make use of the **#pragma unroll** command in CUDA to tell the compiler to unroll the loop. This is possible because the number of dimensions will be defined at compile time and thus the number of iterations of the loop will be constant.

10. PERFORMANCE RESULTS

The performance of the k-indexing functionality has been tested by comparing several different implementations of the same operations. We compare results for several implementations of the simple mesh operations (flip, shift and crinkle) and the Ising model. These implementations are tested on the GPU and CPU for various field lengths. Details of the platform these are tested on are as follows - Linux distribution Kubuntu 9.10 64-bit running on an Intel® Core™2 Quad CPU at 2.66GHz with 8GB of DDR2-800 system memory and an NVIDIA® GeForce® GTX 295 graphics card.

10.1. Mesh Operation Results

The performance of the common mesh transformations are compared between the CPU and the various GPU implementations. The flip, shift and crinkle operations have been implemented as three different GPU kernels: k-indexing for arbitrary dimension size (A), k-indexing for power of two dimension size (B) and a manual kernel (C). These implementations are compared to a CPU k-indexing implementation (D) to provide a performance comparison between the GPU and CPU.

First we compare the performance of the GPU implementations. We compare the two k-indexing implementations to the manually written kernel for each mesh transformation. These results are shown in Figure 11, the graph shows the speeds of each kernel relative to the manually written

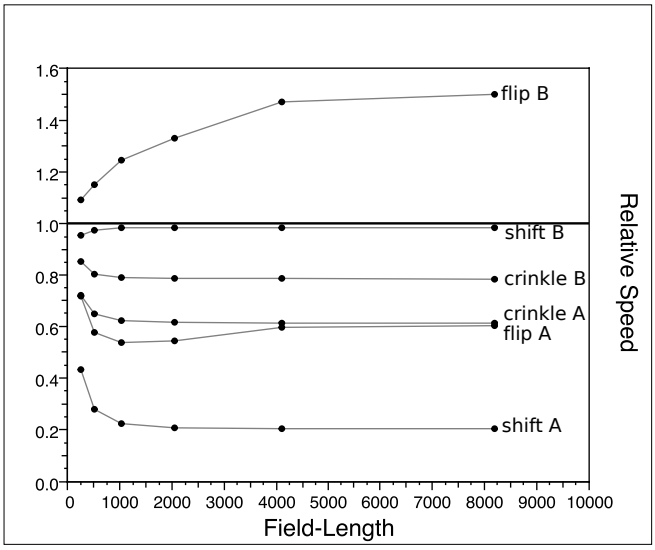


Figure 11. The performance results of the hypercube manipulation k-indexing kernels relative to the manually written kernel. The fixed line at 1 shows the performance of the manual kernels.

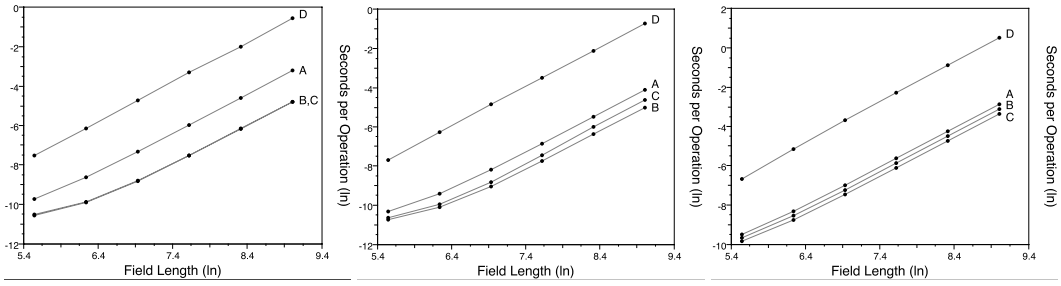


Figure 12. The absolute performance of the hypercube manipulation kernels for shift (left), flip (middle) and crinkle (right). Data shown is the three GPU kernels (A,B,C) and the CPU reference implementation (D) in ln-ln scale.

kernel. These results are gathered for thousands of operations on meshes with field lengths $N = \{256, 512, 1024, 2048, 4096, 8192\}$.

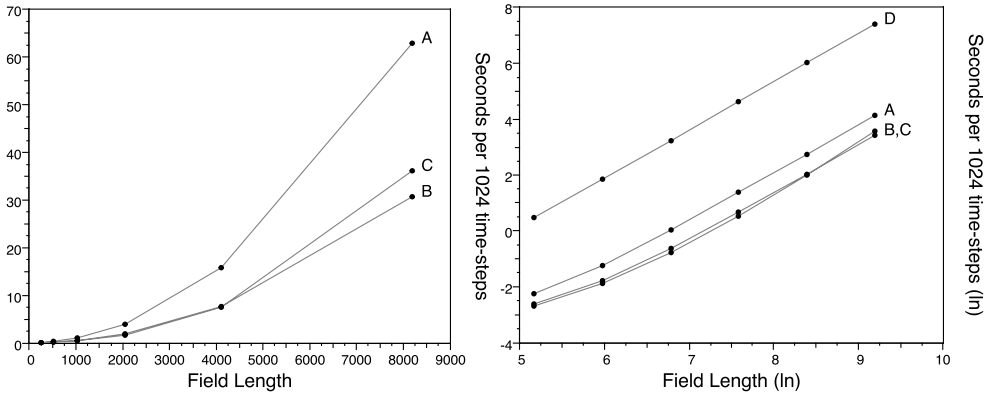


Figure 13. The performance results of the two-dimension Ising simulation implementations: arbitrary dimension length k-index (A), power-of-two dimension length k-indexing (B) and two-dimension specific implementation (C). The ln-ln plot (right) shows also shows the 2D specific CPU implementation (D) for reference.

10.2. Ising Simulation Results

To test the performance of the k-indexing techniques for use within the Ising model, we have again implemented three different simulations. The simulation implemented with k-indexing for arbitrary dimension lengths (A), with k-indexing for power-of-two lengths (B) and as a manual two-dimension specific implementation (C). The performance results of these implementations are presented in Figure 13.

As expected, the arbitrary dimension length k-indexing kernel (A) has the lowest performance of the three, taking approximately twice as long as the two-dimension specific implementation. The performance for the power-of-two k-indexing kernel (B) and the two-dimension specific implementation (C) have very similar performance with kernel B taking slightly longer with the exception of the largest field size. Interestingly for a field length of $N = 8192$, kernel B has the best performance and on the ln-ln plot in Figure 13 it would appear that kernel C does not scale as well as kernel B. It would be interesting to compute the simulation on the next power of two ($N = 16384$) to determine if this is some effect of memory access or that kernel B does indeed scale better. However, our GTX 260+ does not have sufficient memory to allocate the necessary arrays.

11. DISCUSSION AND CONCLUSIONS

We have described the general problem of re-laying out hypercubic arrays in memory in different ways and re-indexing them appropriately to achieve optimal data-parallelism at a fine-grained level. We have described both a general formulation for arbitrary lengths in arbitrary dimensions as well as particular optimisations for an arbitrarily dimensioned set of lengths that are all powers-of-two.

We have shown how these transformations can be expressed both as plain ordinary general serial C code implementations but also for use with CUDA code that runs on NVIDIA GPUs. We have implemented data shifts, stencil operations, array flipping or index-reversals, and crinkling and uncrinkling operations. These transformations allow different data layouts to be readily achieved in a single program even when different algorithmic parts may have different optimal data layouts on the host CPU and accelerator GPU. Two or more data-transforms can be composed into one by implementing a function that computes "an effective k-index" based upon the sequence of their associative application.

We have shown how optimal data layouts in CUDA support exploitation of the various special purpose memory (texture, global, and thread-local) that is available on GPUs for applications level programming. We have presented performance data for a number of simple applications that make use of our operations and transform codes and have compared different approaches on the GPU with one another as well as with reference versions running on the host CPU. Our CPU versions are not designed to be the absolute optimal cases and are provided more to give a performance context to the GPU codes.

We found that the power-of-two optimised versions work faster than the generalised versions. This is not entirely surprising but the quantifiable speedup advantages emphasise the advantage in choosing particular problem sizes for simulations and applications where the particular size is not directly constrained and is chosen for statistical and job scheduling reasons only.

Our CUDA kernel implementations B (the power-of-two implementations) and C (the arbitrary size implementations) for each transformation were both marginally slower than the custom-optimised kernel versions A, but do of course offer extension to arbitrary dimensions without code modification. All three kernel versions were considerably faster than the CPU reference implementations.

Although some of the primitive operations we have discussed seem relatively well-known, we believe it is worthwhile to state these transformations and data layout considerations explicitly, as while some past supercomputer proprietary libraries have partially implemented them, it has not always been clear how to do so portably. We have provided implementations using C-syntax, not only because this corresponds closely to CUDA syntax, but also because we believe anything that can be expressed in C-syntax can be readily ported to other appropriate data-parallel languages such as various modern Fortrans. The reverse is not necessarily true.

The hypercubic apparatus we have described is quite general and it has been worthwhile to develop since it can be used in several application models and simulations such as the arbitrary dimension Ising model simulations [54] that we have described. We were able to develop a single simulation code that could correctly support (subject to memory availability) arbitrary dimensions. This is valuable in computations involving phase transitions and other critical phenomena where one needs to investigate the same model in different dimensions to ascertain the critical dimension if there is one.

We have also developed a site-exchange Kawasaki dynamics [55] Ising model simulation on simple cubic and body-centred cubic lattices. This model has a larger neighbour interaction halo and therefore needs a different crinkling/uncrinkling block factor. Our apparatus makes it trivial to change the block factor and reuse code without the need for extensive re-testing for correctness. Other models such as the Heisenberg or Potts models [56] and spatial Prisoner's Dilemma [57] have a similar block structure but different individual data element sizes which our apparatus caters for. Partial differential equation based simulations using direct solvers for the: Cahn-Hilliard equation (real scalar field) [50]; Time-Dependent Ginzburg Landau equation (complex scalar field) [58]; and multi-species spatial Lotka-

Volterra equations (n- real scalars) [59] can also be rapidly implemented with the apparatus. Other lattices such as triangular meshes or face-centred cubic systems are also possible [60]. We have found it valuable to have **one** general implementation of this hypercubic framework with a **detailed exposition of the k-indexing terminology** as an aid to producing **many** different simulation applications without need to continually reinvent and implement the data-layout code.

We expect it may be possible to develop a run-time library of these transforms in CUDA or OpenCL and also that it is feasible to develop higher-level language preprocessors or macros that can automatically generate calls to them. Indeed it may be possible to adapt existing macro and directive language syntax such as that of OpenMP [61] to generate appropriate CUDA/OpenCL library routine calls.

These data storage indexing ideas also apply to modern multicore devices with multiple caches [62]. GPUs at present typically have no or little cached memory and work instead with the specific memory available to each core locally. We also expect to be able to use our k-indexing apparatus to investigate performance optimisation on conventional multi-core CPUs that do have various levels of cache which have non-trivial optimal data block sizes for different applications.

Another area for future work is the use of a more general set of prime factors to decompose arbitrary length transforms into. We have presented special implementations using powers-of-two and while it is possible to round up and pad data to the nearest power-of-two, a more efficient approach would be to use a prime-factors decomposition that is appropriately linked to a library of special-case implementations. This approach has been successfully used for optimising fast Fourier Transforms [63, 64] and we believe a similar approach is possible for data-parallel layout transformations.

Implementing these ideas for CUDA and GPUs has provided a particular motivation to achieve good performance on a modern parallel platform. There is also a strong likelihood of ideas that are implementable in CUDA having a projected longer lifetime since they will likely be easily implemented in the forthcoming OpenCL language standard too. We anticipate tools for fine-grained data-parallelism. such as we have discussed, having a relatively long usability lifetime as hyper-core architectures like GPUs become part of future CPU designs.

ACKNOWLEDGEMENTS

Thanks to A.Leist for useful discussions on GPU and CUDA issues.

REFERENCES

1. Lee C, Yang T, Wang YF. Partitioning and scheduling for parallel image processing operations. *Proc. IEEE Symp. Parallel and Distributed Processing*, 1995; 86–90.
2. Reddaway SF, Scott DM, Smith KA. A very high speed Monte-Carlo simulation on DAP. *Comp. Phys. Comms.* 1985; **37**:351–356.
3. Baillie C, Gupta R, Hawick K, Pawley G. Monte-Carlo Renormalisation Group Study of the Three-Dimensional Ising Model. *Phys.Rev.B* 1992; **45**:10 438–10 453.
4. Fatahalian K, Houston M. A Closer Look at GPUs. *Communications of the ACM* October 2008; **51**(10):50–57.
5. Owens JD, Luebke D, Govindaraju N, Harris M, Krüger J, Lefohn AE, Purcell T. A survey of general-purpose computation on graphics hardware. *Eurographics 2005, State of the Art Reports*, 2005; 21–51.

6. Langdon W, Banzhaf W. A SIMD Interpreter for Genetic Programming on GPU Graphics Cards. *Proc. EuroGP*, vol. LNCS 4971, O'Neill M, Vanneschi L, Gustafson S, Alcazar AE, Falco ID, Cioppa AD, Tarantino E (eds.), 2008; 73–85.
7. Messmer P, Mullowney PJ, Granger BE. GPULib: GPU computing in high-level languages. *Computing in Science & Engineering* September/October 2008; **10**(5):70–73.
8. Nickolls J, Buck I, Garland M, Skadron K. Scalable parallel programming with CUDA. *ACM Queue* March/April 2008; **6**(2):40–53.
9. Reddaway SF. DAP a Distributed Array Processor. *Proceedings of the 1st annual symposium on Computer Architecture,(Gainesville, Florida)*, ACM Press, New York, 1973; 61–65.
10. Flanders PM. Effective use of SIMD processor arrays. *Parallel Digital Processors*, IEE: Portugal, 1988; 143–147. IEE Conf. Pub. No. 298.
11. Hillis WD. *The Connection Machine*. MIT Press, 1985.
12. Green RC. Maspar mp-1 technolqgy and its applications. *IEE Colloquium on Parallel Processing: Industrial and Scientific Applications*, 1991.
13. Liddell HM, Bowgen GSJ. The DAP subroutine library. *Computer Physics Communications* June 1982; **26**(3-4):311–315.
14. Johnsson SL. Data parallel supercomputing. *Preprint YALEU/DCS/TR-741*, Yale University, New Haven, CT 06520 sep 1989. The use of Parallel Procesors in Meteorology, Pub. Springer-Verlag.
15. Parkinson D, Liddell HM. The measurement of performance on a highly parallel system. *IEEE Trans. on Computers* January 1983; **c-32**(1):32–37.
16. Reddaway SF. Distributed Array Processor, Architecture and Performance. *High-Speed Computation*, *NATO ASI Series*, vol. 7, Kowalik JS (ed.), Springer-Verlag, 1984; 89–98.
17. Wilding N, Trew A, Hawick K, Pawley G. Scientific modeling with massively parallel SIMD computers. *Proceedings of the IEEE* Apr 1991; **79**(4):574–585, doi:10.1109/5.92050.
18. Yau HW, Fox GC, Hawick KA. Evaluation of High Performance Fortran through applications kernels. *Proc. High Performance Computing and Networking 1997*, 1997.
19. Hawick KA, Havlak P. High performance Fortran motivating applications and user feedback. *Technical Report NPAC Technical Report SCCS-692*, Northeast Parallel Architectures Center January 1995.
20. Fox G. Fortran D as a portable software system for parallel computers jun 1991. Center for Research on Parallel Computation, Rice University, PO Box 1892, Houston, Texas, TX 77251-1892, CRPC-TR91128.
21. Chapman B, Mehrotra P, Zima H. Vienna Fortran - a Fortran language extension for distributed memory multiprocessors. *ICASE Interim Report 91-72*, Institute for Computer Applications in Science and Engineering, NASA, Langley Research Center, Hampton, Virginia 23665-5225 sep 1991.
22. Bozkus Z, Choudhary A, Fox GC, Haupt T, Ranka S. Fortran 90D/HPF compiler for distributed-memory MIMD computers: design, implementation, and performance results. *Proc. Supercomputing '93*, Portland, OR, 1993; 351.
23. Bogucz EA, Fox GC, Haupt T, Hawick KA, Ranka S. Preliminary evaluation of high-performance Fortran as a language for computational fluid dynamics. *Proc. AIAA Fluid Dynamics Conf*, AIAA 94-2262, Colorado Springs, 1994.
24. AMT Ltd, 65 Suttons Park Avenue, Reading, RG6 1AZ. *DAP Series FORTRAN PLUS Language (Enhanced)* 1990.
25. GSPawley, GWThomas. The implementation of lattice calculations on the DAP. *J.Comp.Phys.* Aug 1982; **47**(2):165–178.
26. Buck I, Foley T, Horn D, Sugerman J, Fatahalian K, Houston M, Hanrahan P. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.* 2004; **23**(3):777–786, doi:http://doi.acm.org/10.1145/1015706.1015800.
27. AMD. *ATI CTM Guide* 2006.
28. McCool M, Toit SD. *Metaprogramming GPUs with Sh*. A K Peters, Ltd., 2004.
29. Khronos Group. OpenCL 2008. URL `http://www.khronos.org/opencl/`.
30. Hwu WM, Rodrigues C, Ryoo S, Stratton J. Compute Unified Device Architecture Application Suitability. *Computing in Science and Engineering* 2009; **11**:16–26.
31. Herlihy M, Luchangco V, Martin P, Moir M. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. on Computer Systems* May 2005; **23**(2):146–196.
32. Liu YA, Stoller SD, Li N, Rothamel T. Optimizing aggregate array computations in loops. *ACM Trans. on Programming Languages and Systems* January 2005; **27**(1):91–125.
33. Flanders P, Reddaway S. Parallel Data Transforms. *DAP Series*, Active Memory Technology 1988.
34. Fraser D. Array permutation by index-digit permutation. *Journal of the ACM* April 1976; **23**(2):298–309.
35. Mackerras P, Corrie B. Index bit permutations for automatic data redistribution. *Proceedings of the IEEE symposium on Parallel rendering*, ISBN:1-58113-010-4, Phoenix, Arizona, USA, 1997; 23–30.
36. Ding CH. An optimal index reshuffle algorithm for multidimensional arrays and its applications for parallel architectures. *IEEE Trans. on Parallel and Distributed Systems* March 2001; **12**(3):306–315.
37. Fletcher P, Robertson P. A generalised framework for parallel data mapping in multidimensional signal and image processing. *International symposium on Signal Processing and its Applications*, 1992; 614–617.
38. Smith K, Fletcher PA. Status of parallel mapping functions. *Technical Report HJ/s/6-1*, CSIRO 1992.

39. Swan M, Coddington P, Hawick K. An implementation of k-tiling algebra. *Technical Report DHPC-098*, Adelaide University, South Australia October 2000.
40. Bosman O, Fletcher P, Tsui K. K-tiling: A structure to support regular ordering and mapping of image data. *APRS Workshop on Two and Three Dimensional Spatial Data: Representation and Standards*, 1992.
41. Tsui K, Fletcher P, Hungerford S. A flexible kernel image model. *Proceedings of the APRS DICTA Conference, Melbourne*, 1991.
42. Vozina G, Fletcher P, Robertson P. Volume rendering on the (maspar) (mp-1). *Workshop on Volume Visualisation*, 1992; 3–8.
43. Fletcher PA. Visualisation on a massively parallel supercomputer: Regular mapping and handling of multidimensional data on simd architectures,. *Proceedings of the Fourth Australian Supercomputer Conference, Bond University Qld.*, 1991.
44. AMT Ltd, 65 Suttons Park Avenue, Reading, RG6 1AZ. *DAP 500 Image Processing Library* 1988.
45. NVIDIA® Corporation. *CUDA™ 2.0 Programming Guide* 2008. URL http://www.nvidia.com/, last accessed November 2008.
46. Leist A, Playne D, Hawick K. Exploiting Graphical Processing Units for Data-Parallel Scientific Applications. *Concurrency and Computation: Practice and Experience* December 2009; **21**:2400–2437, doi:10.1002/cpe.1462. CSTN-065.
47. Knuth D. *The Art of Computer Programming: Fundamental Algorithms*, vol. 1. 3rd edn., Addison-Wesley, 1997.
48. Cierniak M, Li W. Unifying data and control transformations for distributed shared-memory machines. *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, ACM: New York, NY, USA, 1995; 205–217, doi:http://doi.acm.org/10.1145/207110.207145.
49. Hawick KA, Leist A, Playne DP. Parallel graph component labelling with gpus and cuda. *Technical Report CSTN-089*, Massey University June 2009. Submitted to Parallel Computing.
50. Playne D, Hawick K. Data Parallel Three-Dimensional Cahn-Hilliard Field Equation Simulation on GPUs with CUDA. *Proc. 2009 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'09) Las Vegas, USA.*, CSTN-073, 2009.
51. Niss M. History of the Lenz-Ising Model 1920-1950: From Ferromagnetic to Cooperative Phenomena. *Arch. Hist. Exact Sci.* 2005; **59**:267–318, doi:doi10.1007/s00407-004-0088-3.
52. Ising E. Beitrag zur Theorie des Ferromagnetismus. *Zeitschrift fuer Physik* 1925; **31**:253258.
53. Onsager L. Crystal Statistics I. Two-Dimensional Model with an Order-Disorder Transition. *Phys.Rev.* Feb 1944; **65**(3):117–149.
54. Hawick K, Leist A, Playne D. Damaged Lattice and Small-World Spin Model Simulations using Monte-Carlo Cluster Algorithms, CUDA and GPUs. *Technical Report CSTN-093*, Computer Science, Massey University 2009.
55. Kawasaki K. Diffusion constants near the critical point for time dependent Ising model I. *Phys. Rev.* 1966; **145**(1):224–230.
56. KBinder, DPLandau. Critical properties of the two-dimensional anisotropic Heisenberg model. *Phys.Rev.B* 1976; **13**(3):1140–1155.
57. Hawick KA, Scogings CJ. Roles of space and geometry in the spatial prisoners' dilemma. *IASTED Int. Conference on Modelling, Simulation and Identification (MSI'09), Beijing, China*, 2009. 659-010.
58. Playne D, Hawick K. Visualising vector field model simulations. *Proc. 2009 International Conference on Modeling, Simulation and Visualization Methods (MSV'09) Las Vegas, USA.*, CSTN-074, 2009.
59. Hawick KA. Erlang and distributed meta-computing middleware. *Technical Report CSTN-092*, Computer Science, Massey University 2009.
60. Johnson M, Playne D, Hawick K. Data-parallelism and gpus for lattice gas fluid simulations. *Technical Report CSTN-109*, Computer Science, Massey University March 2010.
61. Chapman B, Jost G, van der Pas R. *Using OpenMP - Portable Shared Memory Parallel Programming*. ISBN 978-0-262-53302-7, MIT Press, 2008.
62. Lu Q, Alias C, Bondhugula U, Henretty T, Krishnamoorthy S, Ramanujam J, Rountev A, Sadayappan P, Chen Y, Lin H, *et al.*. Data layout transformation for enhancing data locality on nuca chip multiprocessors. *PACT '09: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, IEEE Computer Society: Washington, DC, USA, 2009; 348–357, doi:http://dx.doi.org/10.1109/PACT.2009.36.
63. Kolba DP, Parks TW. A prime factor fft algorithm using high-speed convolution. *IEEE Trans. on Acoustics, Speech and Signal Processing* August 1977; **ASSP-25**(4):281–294.
64. Chan S, Hoe K. Prime-factor algorithm and Winograd Fourier transform algorithm for real symmetric and antisymmetric sequences. *IEE Proceedings* April 1989; **136 Part G**(2):87–94.