Parallel Computing 36 (2010) 655–678

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Parallel graph component labelling with GPUs and CUDA

K.A. Hawick [*], A. Leist, D.P. Playne

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

A R T I C L E I N F O

Article history:
Received 3 June 2009
Received in revised form 22 April 2010
Accepted 21 July 2010
Available online 10 August 2010

Keywords:
Graph
Component label
Mesh
GPU
CUDA
Data-parallel

A B S T R A C T

Graph component labelling, which is a subset of the general graph colouring problem, is a computationally expensive operation that is of importance in many applications and simulations. A number of data-parallel algorithmic variations to the component labelling problem are possible and we explore their use with general purpose graphical processing units (GPGPUs) and with the CUDA GPU programming language. We discuss implementation issues and performance results on GPUs using CUDA. We present results for regular mesh graphs as well as arbitrary structured and topical graphs such as small-world and scale-free structures. We show how different algorithmic variations can be used to best effect depending upon the cluster structure of the graph being labelled and consider how features of the GPU architectures and host CPUs can be combined to best effect into a cluster component labelling algorithm for use in high performance simulations.

1. Introduction

Graph component labelling is an important algorithmic problem that occurs in many applications areas. Generally, the problem involves identifying which nodes in a graph belong to the same connected cluster or component. In many applications the graph of interest has undirected arcs and all nodes in a connected cluster are fully and mutually reachable. Some applications problems will give rise to structural graphs that are generated as a number of separate clusters, or break up into separate clusters or coagulate into larger super-clusters as a result of some dynamical process. In some cases the graph is simply a structural one and connectivity is determined entirely by the presence or absence of arcs between nodes.

Many simulations problems are formulated in terms of a set of variables on a mesh and the value of the variables on the nodes can be used to differentiate between nodes that are connected or not. Fig. 1 shows two examples of graph colouring and labelling applications problems. On the left a set of variables on a regular mesh graph have been coloured according to their similarity properties, whereas on the right a structural graph has been coloured according to its cluster components. Examples of the former case are the Ising model, Potts model, other lattice physics models and field models where the structural arcs are fixed – in a nearest neighbour pattern for example – but where some property of each node that can change dynamically, indicates whether two adjacent nodes are part of the same cluster or not. Examples of the latter case include analysis of computer networks; sociological networks; and other relationship network graphs.

Fig. 2 illustrates how component labelling can be used to analyse the results of a simulation. The figure shows a Kawasaki spin-exchange Ising model simulation [1] where a random mixture of spins (coloured dark blue and white) has been quenched to a low temperature and clusters or components have formed in the model system. Models such as these exhibit phase transitions and the component size distribution can be used to examine and identify the location and properties of the resulting phase transitions.

* Corresponding author. Tel.: +64 9 414 0800; fax: +64 9 441 8181.
E-mail addresses: k.a.hawick@massey.ac.nz (K.A. Hawick), a.leist@massey.ac.nz (A. Leist), d.p.playne@massey.ac.nz (D.P. Playne).

0167-8191/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2010.07.002

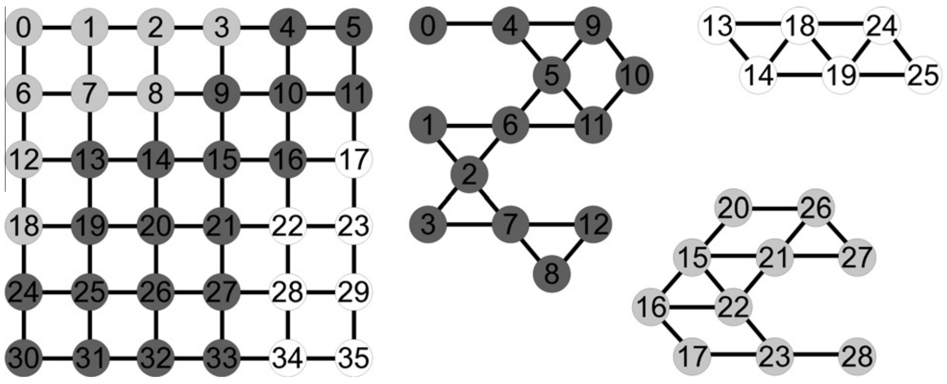

Fig. 1. Component labelling on a regular mesh (left) – where each site has a known number of neighbours but different properties – and on an arbitrary graph (right). Different colours (shades) denote different components. Mesh problems arise from simulations where each node has **different model specific properties**, e.g. spin, or velocity direction or some other model value that dictates whether two physically adjacent nodes are deemed to be part of the same "model component". The regular mesh has in this case 26 separate clusters, the largest with 3 nodes, the arbitrary graph has 3 separate components, the largest with 8 nodes.

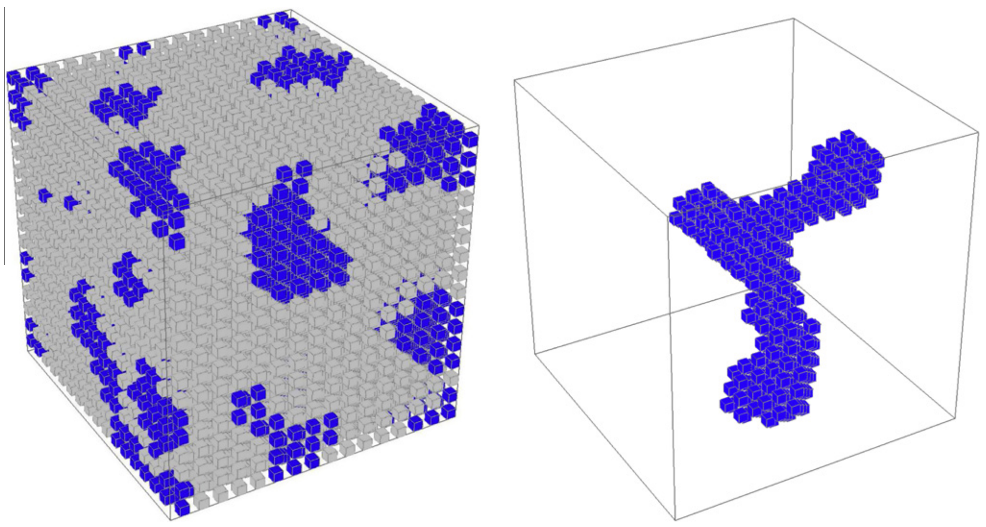

Fig. 2. The cubic system on the right is a body-centred cubic configuration of spins (blue and grey) of a Kawasaki spin-exchange Ising model that has been equilibrated below the critical temperature. The image on the right is the largest component of the blue minority (0.2 volume fraction) phase. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Until recently, use of component analysis was computationally expensive compared to the simulation update computational costs. It was therefore impractical as a metric of simulations except when used in-frequently during model evolution – such as to analyse static model configuration dumps. Fast component labelling algorithms such as we describe here for use on a cheap accelerator platform such as a GPU, enable use of component labelling more frequently in simulations.

Fig. 3 shows how the largest component measurements can be used to identify special properties of a model system such as the location in parameter space of a phase transition. The figure shows how the normalised fraction of the model system that is occupied by the giant largest component of the minority (dark spin) phase of the model shown in Fig. 2 varies with temperature and minority volume fraction. The crease in the surface locates the phase transition.

Graph colouring is of course not a new idea, although as mentioned it has previously been impractically expensive to compute regularly during a simulation. There are a number of well-known algorithms for graph colouring [2] including parallel algorithms for speeding up graph calculations [3] as well as those specifically on graph colouring such as [4,5]. In addition, a number of parallel component labelling applications have been described in the applications literature, particularly for use in lattice physics simulation models [6,7], studies of percolation [8], and studies of other phase transitions [9].

Parallel graph colouring is a problem that tends to work well on data-parallel systems and during the last golden age of data parallelism, computers such as the Distributed Array Processor (DAP) [10] and the Connection Machine examples [11] had software libraries with built-in routines for component labelling on regular rectilinear data structures such as images.

Graphical processing units (GPUs) such as NVIDIA's GeForce series have revitalised interest in data-parallel computing since they offer dedicated data-parallel hardware on the graphics chip but are also readily integrated into a compute server

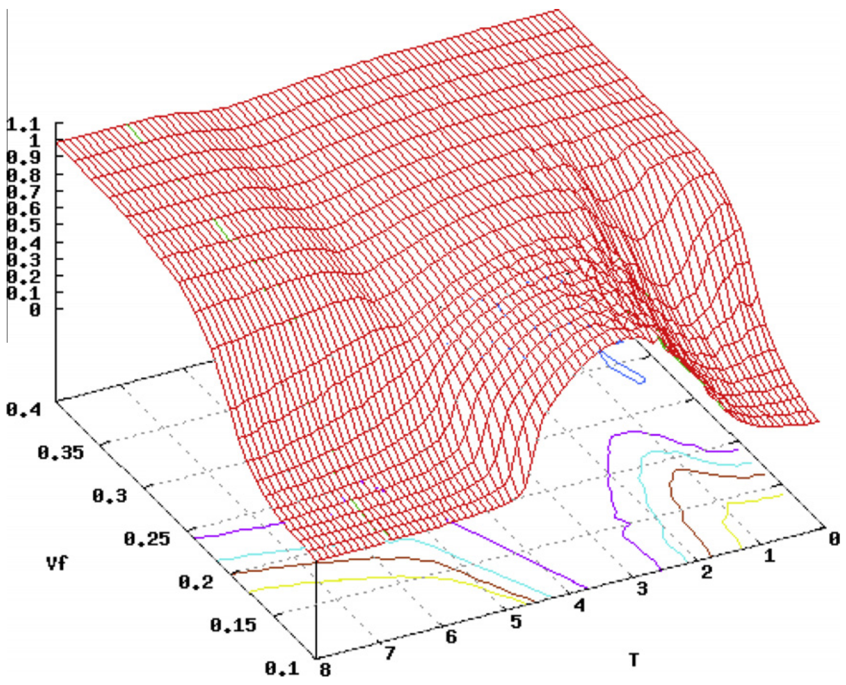

Fig. 3. Largest component analysis of model system such as shown in Fig. 2. The normalised fraction occupied by the largest component cluster of the minority (blue) phase is plotted as a surface as it varies with the model temperature T and the minority volume fraction V_f (percentage of blue spins). The phase transition is revealed elegantly from the component labelling analysis and identification of the size of the largest cluster component. Measurements shown are averaged over 100 independent model runs on a $32 \times 32 \times 32$ body centred cubic system. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

and make the parallel capability accessible to the applications programmer through use of NVIDIA's Compute Unified Device Architecture (CUDA). While CUDA is presently specific to NVIDIA hardware, the Open Compute Language (OpenCL) has similar capabilities and programming interface.

In this article we describe how some of the well-known data-parallel graph component labelling algorithms can be adapted to make specific and optimal use of GPU hardware and how this can be expressed in a relatively high-level manner using programming languages like CUDA. We describe the component labelling problem in Section 2 and the CUDA GPU architecture in Section 3. Our experience has been that applications that require component labelling fall into major classes – firstly those that employ arbitrarily structured graphs such as random graphs or scale-free and Watts–Strogatz "small-world" graphs and secondly those that use regular rectilinear data such as images or hypercubic lattice data. There are quite different optimisations that can be made in the two cases. We describe parallel algorithms both for arbitrarily structured graphs (Section 4) and more symmetric mesh and lattice graphs (Section 5) and show how they can be implemented on GPUs with a data-parallel language like CUDA. We present some performance results using NVIDIA GeForce 260 GPUs in Section 6 and in Section 7 we discuss the implications for component labelling applications and offer some conclusions and areas for future optimisation work.

GPUs are typically programmed using a conventional host calling program written in (for example) "C" that runs on the hosting CPU with specific fast 'kernel" programs that are called and which run on the GPU. A large part of our article describes these parallel kernel algorithms in detail. For clarity we refer to Kernels 1, 2, 3 that address the arbitrary graph component labelling problem and Kernels A, B, C, D which address hypercubic regular data component labelling.

2. Graph component labelling algorithms

In the context of general graphs and graph theory [12] graph labelling is a well known problem with relatively easy to understand (serial) algorithms [13]. Generally graph component labelling is a subset problem of the wider colouring problems [14] with a number of known algorithmic variations [15].

One perspective on the problem is in terms of equivalence classes and the popular formulation in Numerical Recipes books [16] shows how a predicate function that specifies equivalence or "connected-ness" can be used to iteratively sweep across all-pairs of graph nodes to identify clusters or equivalence classes. In the case of simulations or applications involving fixed graphs or meshes, we often know the adjacency-lists in advance and it is not necessary to conduct sweeps over all-pairs of nodes. In our work, we assume this is the case, and that some data structure exists that specifies the subset of nodes that are considered adjacent to a particular node.

Fig. 4 illustrates the component labelling problem for a d-dimensional rectilinear image. It is required that each vertex be coloured or labelled with each component having a unique integer label. Arbitrary graph vertices can be indexed explicitly by some integer k but even in the case of a hyper-cubic mesh, image or lattice, these indices can encode the spatial dimensions $(x_1, x_2, \ldots, x_d)$ in some order of significance [17].

The initial labelling of individual vertices is often the slowest algorithmic component as it may involve sweeping through the adjacency information iteratively. To make use of the correctly labelled component information however, an application may need to carry out some "book-keeping" operations to tidy up the label information into a structure such as that shown in the right of Fig. 4 – whereby subsequent iterations over each component cluster can be made – for example to compute cluster moments or average properties. It is also often desirable that the cluster structure be compactified, particularly where the number of components is relatively small compared to the total number of vertices.

In summary, starting from a graph, specified either as a mesh or as a set of adjacency information, we often want to: colour the vertices uniquely according to component; determine how many unique colours (components) we have used; gather pointers for each cluster to allow efficient subsequent iteration by cluster. In addition, depending upon the application we may also want to: compactify the colours to a contiguous and continuous integer list; sort the colours by cluster sizes; histogram the size distribution; apply some algorithm to one or more clusters, or remove them from the set.

Sequential algorithms for component labelling will typically be implemented as a sweep through all the graph nodes, adjusting their component "colours" according to adjacency information until each component has a single and unique colour – usually in the form of an integer label. The sweep is usually organised as a breadth first tree search, with various accelerating heuristics, depending upon properties of the data. Some formats lend themselves particularly well to various heuristics and also to parallel algorithms. The data-parallel GPU is well-suited to carrying out the data-parallel operations of component labelling and its host CPU can be employed fairly easily to carry out the book-keeping support operations serially, and usually with more ordinary or global memory available than the dedicated memory available on the GPU card itself.

3. GPU architecture

In this section we give a brief summary of GPU-specific terms used in the rest of this article. Graphics processing units or GPUs provide a high computational throughput due to their large number of processor cores. GPUs contain multiprocessors (MPs) which each contain eight Scalar Processors (SPs) and additional memory. These architectures use the SIMT (Single Instruction Multiple Thread) parallel programming paradigm. The SPs within a single MP must execute the same instruction synchronously while the MPs in the GPU can execute instructions independently of each other.

A large number of threads is required to make full use of a GPUs computing capability. The GPU can manage, schedule and execute many threads in hardware to avoid high thread management overheads. These threads are organised as a large grid of blocks. Each block is a 3D structure that can contain up to 512 threads with maximum sizes of $\{\mathbf{512, 512, 64}\}$ and the grid is a 2D structure with maximum sizes of $\{65535, 65535\}$. The threads are executed by assigning thread blocks to MPs. The multiprocessors will split the thread blocks into sets of 32 threads known as warps. When executing instructions, the MP will select a warp that is ready to execute and issue it an instruction.

Each thread has a unique id that it can work out from it's threadIdx and blockIdx. The blockIdx identifies which block the thread belongs to and the threadIdx defines the thread's position within the block.

There is no automatic memory caching on the GPU, instead a number of optimised memory types are made available to be explicitly used by the programmer. These memory types are:

Global memory is the main device memory on the GPU. Any input/output to and from the GPU must be placed in global memory. This memory also has the slowest access time. The number of necessary global memory transactions can be reduced by a technique known as coalescing. When 16 sequential threads access 16 sequential and aligned values in memory, the GPU will automatically combine them into a single transaction.

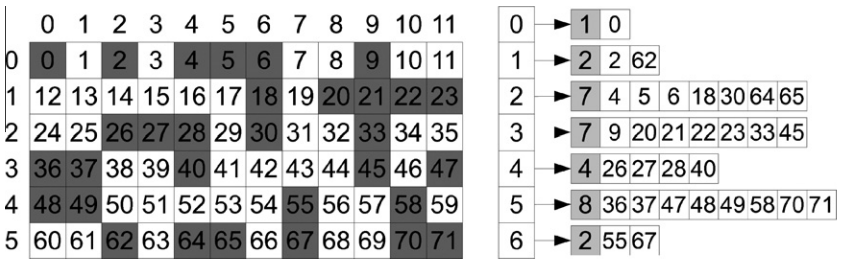

Fig. 4. A 12×6 periodic (opposite edges are connected) nearest-neighbour mesh where the cluster "sameness" also depends on the variables decorating the graph nodes. Graph node $k = 0$ has exactly 4 nearest neighbouring nodes that it is adjacent to (1, 11, 12, 60), but it has correctly been identified as the sole member of cluster $j = 5$ where nodes "need to be green" and be "nearest-neighbour connected" to be part of the same cluster. There are thus 8 clusters in total found and we want them labelled accordingly.

Shared memory is stored within the MPs and allows threads in the same block to share information. The main use of the memory is to act as an explicit access cache to allow the number of global memory transactions to be reduced by sharing common data between threads.

Texture memory is a cached method of accessing a designated piece of global memory. The texture cache is located on the MPs. Textures can be created in **1**-, **2**- and **3**-dimensions and will automatically cache values in the same spatial locality. This will improve performance when threads access values in some regular spatial neighbourhood.

Constant memory is another cached method of accessing a small section of global memory. This cache is designed to allow multiple threads to access the same value out of the cache at once. This is very useful when all the threads in a block must access the same value from memory.

4. Arbitrarily structured graphs

This section describes our (serial) CPU and (parallel) GPU algorithms used to label the components of arbitrary graphs. Such graphs are arbitrary in the sense that their structure is not known beforehand. Examples are random graphs [18] and complex networks, such as metabolic networks [19–21], human collaboration networks [22,23] and many other types of social networks [24–26], as well as computer networks like the World-Wide Web [27] to name but a few. These graphs require a data structure that has no built-in assumptions about graph structure and in particular, that supports arbitrarily long (and potentially widely differing) adjacency-list lengths for each vertex node.

4.1. Data structures

An arbitrary graph can be represented in different ways in memory. The data structure can have a critical impact on the performance of the labelling algorithm. Different hardware architectures with diverse memory hierarchies and processing models often require different data structures to achieve high performance. We restrict ourselves to undirected graphs for this article, but the data structures and algorithms described in this section can easily be adapted to work with directed graphs as well. Fig. 5 illustrates the data structures used to represent an arbitrary graph in the main memory of the host system or the device memory of the graphics card.

The graph representations in device memory are designed so that the CUDA kernels can maximise the utilisation of the available memory bandwidth. One important way to do this is to reduce the number of non-coalesced memory transfers. Coalescing is a feature of CUDA devices that enables them to combine the global memory reads or writes of 16 sequential threads into a single memory transaction. The threads have to access addresses that lie in the same memory segment of size 32-, 64- or 128-bytes for coalescing to work (CUDA compute capability 1.2 or higher). This memory block also needs to be aligned to the segment size. A more detailed description and analysis of coalescing and its performance implications can be found in the CUDA Programming Guide [28]. Trying to reduce the number of non-coalesced memory transactions and in general to optimise the device memory accesses is a challenge for arbitrary graphs on CUDA devices, as the structure of the graph is not known beforehand.

4.2. The CPU reference implementation

This section describes the single-threaded CPU implementation used as a comparison to the CUDA kernels. We realise, of course, that a multi-threaded implementation on a multi-core or multi-CPU system could achieve better performance than

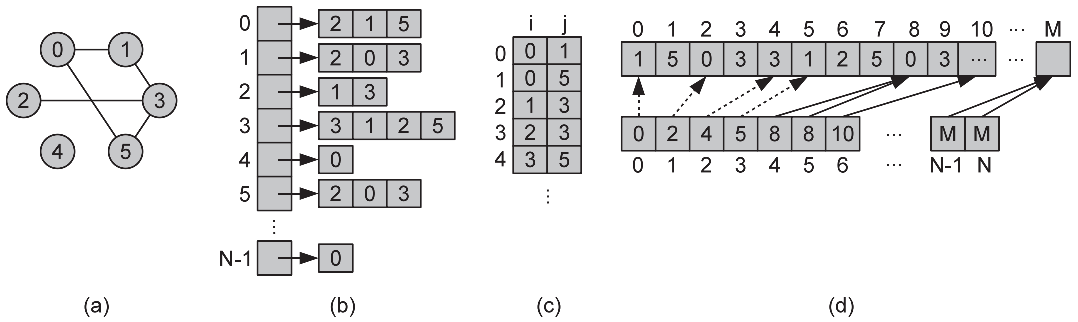

Fig. 5. The data structures used to represent the graph by the CPU and GPU implementations. All three examples represent the same undirected graph shown by (a). (b) Shows the adjacency-list representation used by the CPU algorithm. Every vertex $v_i \in V$ stores a list of its neighbours at index i of the vertex array. The first element of each adjacency-list is its length $|A_i|$, also called its out-degree k_{i_out} or simply degree k_i. The following elements are the indices of the adjacent vertices, that is the arc set $A_i \subseteq A$. To represent an undirected edge, each one of its end points needs to store the respective neighbour. (c) Illustrates the edge set E used by Kernel 1. Every edge $e_{\{i,j\}} \in E$ is represented by a 2-tuple, which stores the IDs of the respective end points. (d) Shows the vertex set V (bottom) and the arc set A (top) used by Kernels 2 and 3. Every vertex $v_i \in V$ stores the start index of its adjacency-list A_i at index i of the vertex array. The adjacency-list length $|A_i|$ can be calculated by looking at the adjacency-list start index of v_{i+1} ($V[i+1] - V[i]$). The vertex array contains $|V| + 1$ elements so that this works for the last vertex too.

the serial one, but we do not intend to compare parallel-processing techniques on different architectures in this paper. We are merely interested in investigating the performance of graph labelling algorithms on CUDA devices and provide the CPU performance for comparison.

The algorithm (see Algorithm 1) initialises the colours of all vertices to distinct values. It then iterates over the vertex set V and starts the labelling procedure for all vertices $v_i \in V$ that have not been labelled yet as indicated by the frontier array F. The labelling procedure iterates over the arc set $A_i \subseteq A$ of vertex v_i, comparing its colour value c_i with that of its neighbours. If it finds that the colour value c_j of a neighbour v_j is greater than c_i, then it sets c_j to the value of c_i and recursively calls the labelling procedure for v_j. If, on the other hand, it finds a neighbour v_j with a lower colour value, it sets the colour c_i to the value c_j and starts iterating over the adjacency-list from the beginning again. This means that once the labelling procedure reaches the end of the adjacency-list of vertex v_i, all vertices that are reachable from v_i are labelled with the same colour value.

Algorithm 1. The single-threaded CPU reference implementation of the component labelling algorithm for arbitrary graphs. Although the data structure used by this algorithm uses lists of directed arcs to represent the links between vertices, it assumes that if there is an arc (i,j) in the adjacency-list $A_i \subseteq A$ of vertex $v_i \in V$, then there is also an arc (j,i) in the adjacency-list $A_j \subseteq A$ of vertex $v_j \in V$. Thus, the algorithm essentially works on edges.

```
function LABEL_COMPONENTS_CPU($G$)
  Input parameters: The graph $G := (V,A)$ is an array of adjacency-lists, one for every vertex $v \in V$. The arc set $A_i \subseteq A$ of a
   vertex $v_i$ is stored in position $V[i]$. $|V|$ is the number of vertices in $G$ and $|A_i|$ is the number of neighbours of $v_i$ (i.e. its
   degree).
  declare integer $C[|V|]$ //$C[i]$ stores the colour of vertex $v_i$
  declare boolean $F[|V|]$ //$F[i]$ indicates if vertex $v_i$ still needs to be processed
  for all $v_i \in V$ do
    $C[i] \leftarrow i$ //initialise every vertex to a unique colour value
    $F[i] \leftarrow$ true
  end for
  for all $v_i \in V$ do
    if $F[i]$ = true then
      call PROCESS_CPU($G,C,F,v_i$)
    end if
  end for
  return $C$ //$C$ now contains the component ID for every vertex $v \in V$.
```

```
function PROCESS_CPU($G,C,F,v_i$)
  $F[i] \leftarrow$ false
  for all $(i,j) \in A_i$ do
    if $C[i] > C[j]$ then
      $C[i] \leftarrow C[j]$
      re-run the for-loop from the beginning of the adjacency-list
    else if $C[i] < C[j]$ then
      $C[j] \leftarrow C[i]$
      call PROCESS_CPU($G,C,F,v_j$)
    end if
  end for
```

4.3. The CUDA implementations

The three CUDA kernels solve the task of labelling the components of an input graph in very different ways. As described in Fig. 5, Kernel 1 uses a different data structure from Kernels 2 and 3. While Kernel 1 works on the edge set, executing one thread for every edge for a total of $|E|$ threads, Kernels 2 and 3 operate on the vertex set, executing one thread for every vertex for a total of $|V|$ threads. The kernels are described in the following paragraphs. As a convention for the pseudo-code, variables located in graphics device memory are identified with a subscript d (e.g. V_d).

4.3.1. Edge-based Kernel 1

In this approach, the colours array is initialised to a unique integer value for every vertex $v \in V$. Every thread loads one edge $e := \{i,j\} \in E$ from the edge set and compares the colours c_i and c_j of the two vertices v_i and v_j. If one of the colour values is smaller than the other, then the colour of the vertex with the higher colour value is updated. An `atomicMin` operation is used for this to ensure that simultaneous updates by different threads do not interfere with each other and do not rely on a particular ordering. In addition to updating the colour, the thread also sets a flag that indicates that there are still changes being made to the vertex colours. This last operation can also cause simultaneous writes to the same address by multiple threads. However, CUDA guarantees that at least one of them will succeed, which is all that is needed here. The kernel is called until no more changes are being made. The pseudo-code of this kernel is given in Algorithm 2.

Performance: The performance of the algorithm depends strongly on the graph diameter d—that is, the longest of the shortest paths connecting any two vertices within the same component of the graph—since the colour needs d steps to propagate between all vertices. However, this does not mean that the kernel has to be called d times, as the $|E|$ threads are not actually all executing at the same time. This means that the changes to the colour of a vertex v_i made while processing an edge $\{i,j\}$ can already be visible when processing an edge $\{i,k\}$ at a later time during the same kernel call. The diameter is thus the upper bound for the number of kernel calls. If the graph consists of more than one disjoint component, then the diameter of the component with the largest value for d is relevant.

CUDA specific optimisations: The kernel uses a boolean flag in shared memory to indicate if any changes have been made by one or more threads in this block. If this flag is set to true once all threads in the block have completed their work, then the first thread in the block sets the respective flag in global memory to true. This optimisation reduces the number of global memory writes and replaces them with much faster writes to shared memory.

Algorithm 2. Kernel 1: The first approach to labelling the components of an arbitrary graph in parallel on a CUDA-enabled device. This kernel operates on the edge set E, running one thread for every edge for a total of $|E|$ threads per kernel call.

```
function LABEL_COMPONENTS_KERNEL1_HOST($E,N$)
  Input parameters: $E$ is the list of all edges in a graph $G := (V,E)$. $N = |V|$ is the number of vertices.
  declare {integer,integer} $E_d[|E|]$ in device memory //allocate memory for the edge list on the device
  declare integer $C_d[N]$ in device memory //$C_d[i]$ stores the colour of vertex $v_i$
  copy $E_d \leftarrow E$ //copy the edge list to the device
  do in parallel on the device using $N$ threads: initialise $C_d[1 \cdots N]$ such that $C_d[i] \leftarrow i$
  declare boolean $m$ //$m$ indicates if the labels are still changing
  declare boolean $m_d$ in device memory
  repeat
    copy $m_d \leftarrow$ false
    do in parallel on the device using $|E|$ threads: call LABEL_COMPONENTS_KERNEL1($E_d,C_d,m_d$)
    copy $m \leftarrow m_d$
  until $m$ = false //repeat until the colours are not changing any more
  declare integer $C[N]$
  copy $C \leftarrow C_d$ //copy the colours to the host
  return $C$ //$C$ now contains the component ID for every vertex $v \in V$.
```

```
function LABEL_COMPONENTS_KERNEL1($E_d,C_d,m_d$)
  declare integer $t \leftarrow$ thread ID queried from the CUDA runtime
  declare {integer,integer} $e_t \leftarrow E_d[t]$ //read the edge $e_t := \{i,j\} \in E_d$ from device memory
  declare integer $c_i \leftarrow C_d[i]$ //read the colour $c_i$ of vertex $v_i$ from device memory
  declare integer $c_j \leftarrow C_d[j]$ //read the colour $c_j$ of vertex $v_j$ from device memory
  if $c_i < c_j$ then
    call atomicMin($C_d[j],c_i$)
    $m_d \leftarrow$ true
  else if$c_j < c_i$ then
    call atomicMin($C_d[i],c_j$)
    $m_d \leftarrow$ true
  end if
```

4.3.2. Vertex-based Kernel 2

Kernel 2, just like Kernel 1, initialises the colours array to a unique integer value for every vertex $v \in V$. Every thread t_i then loads the adjacency-list start index of vertex v_i into shared memory. This allows all but the last thread in the block to calculate the adjacency-list length $|A_i|$ of their vertex without another read from global memory. Every thread then checks if its vertex is flagged as active in the frontier array, which means that this is either the first kernel iteration (all vertices are flagged) or that its colour has changed in the previous kernel call and therefore needs to be compared to the colour values of its neighbours. The next step for all threads that are flagged is to load the current colour c_i of their vertex. Now they can iterate through their adjacency-lists, comparing c_i with the colour c_j of every one of their neighbours v_j and updating either one of the colours if its value is higher than that of the other vertex. The kernel is called until no more changes are being made. The pseudo-code of this kernel is given in Algorithm 3.

Performance: Just like with Kernel 1, the performance of this algorithm depends strongly on the graph diameter d, as the colour needs d steps to propagate between all vertices. Again, this does not mean that d iterations are needed for the colour to spread through the connected components.

CUDA specific optimisations: In addition to the shared memory flag already used in Kernel 1, this kernel uses an array in shared memory to store the adjacency-list start indices of the threads within the same thread block as described before. Furthermore, it uses a one-dimensional texture reference to iterate over the arc set A. Texture fetches are cached and can potentially exhibit higher bandwidth as compared to non-coalesced global memory reads if there is locality in the texture fetches. And since a thread t_i iterates over $|A_i|$ consecutive values and thread t_{i+1} iterates over the directly following $|A_{i+1}|$ consecutive values and so on, this locality is given. The CUDA code for these optimisations is shown in Algorithm 4.

The implementation of Kernel 2 described in this article uses one texture reference for the arc set. As the maximum width for a texture reference bound to linear memory is 2^{27} for all currently available CUDA devices (compute capabilities 1.0–1.3), it can only process graphs with up to 2^{27} arcs or half as many edges (each edge is represented by two arcs). The maximum size of the arc set is thus 512 MB (4 bytes per arc). As some graphics cards provide more device memory than 512 MB, they could be used to process larger graphs if it was not for this limitation. In that case, two or more texture references could be used to access the arc set when necessary. It is important to ensure that the size of the arc set does not exceed this limit, as the CUDA libraries do not give any warning or failure message when trying to bind more elements than supported to the texture reference. The kernel executes as if everything was normal and only the incorrect results, if spotted, show that something went wrong. This is a commonly encountered problem when using CUDA and requires particular care from the programmer.

Each thread in Kernel 2 iterates over the adjacency-lists of one vertex, which causes warp divergence if the degrees of the vertices that are being processed by the threads of the same warp are not the same. Due to the SIMD data model (or SIMT in CUDA terminology), all threads in a warp have to do k_{max} iterations, where k_{max} is the highest degree processed by this warp. This problem can be reduced by sorting the vertices by their degree. However, the time to sort the vertices on the CPU and the overhead in the CUDA kernel, which comes from the fact that the vertex ID is no longer equal to the thread ID and therefore has to be looked up from device memory, are considerable. Our experiments have showed that this technique can slightly improve the kernel execution time when processing scale-free graphs with their "fat tail" degree distributions, but the overall execution time, which includes the time required to sort the vertices by the CPU, increases considerably. These findings are in line with previous results reported in [29]. Consequently, the implementation of Kernel 2 reported here does not sort the vertices. Nevertheless, other graph algorithms may benefit from this technique, for example when they execute the same kernel multiple times without changes to the graph structure.

Algorithm 3. Kernel 2: The second approach to labelling the components of an arbitrary graph in parallel on a CUDA-enabled device. This kernel operates on the vertex set V, running one thread for every vertex for a total of $|V|$ threads per kernel call. Like in the CPU reference implementation described in Algorithm 1, the kernel operates on a set of arcs but assumes that for every arc (i,j) an arc (j,i) exists as well.

```
function LABEL_COMPONENTS_KERNEL2_HOST($V$,$A$)
  Input parameters: The vertex set $V$ and the arc set $A$ describe the structure of a graph $G := (V,A)$. Every vertex $v_i \in V$
   stores the index into the arc set at which its adjacency-list $A_i$ begins in $V[i]$. The adjacency-list length $|A_i|$ is calculated
   from the adjacency-list start index of vertex $v_{i+1}$ ($V[i+1] - V[i]$). To make this work or the last vertex $v_N \in V$, the
   vertex array contains one additional element $V[N+1]$. This additional element is not included in the number of
   vertices $|V|$. $|A|$ is the number of arcs in $G$.
  declare integer $V_d[|V|+1]$ in device memory //allocate memory for the vertex list $V$
  declare integer $A_d[|A|]$ in device memory //allocate memory for the arc list $A$
  declare integer $C_d[|V|]$ in device memory //$C_d[i]$ stores the colour of vertex $v_i$
  declare boolean $F1_d[|V|]$, $F2_d[|V|]$ in device memory //$F1_d[i]$ indicates if a vertex $v_i$ is to be processed in the current
   iteration and $F2_d[i]$ indicates if it is to be processed in the next iteration
  copy $V_d \leftarrow V$ //copy the vertex set to the device
  copy $A_d \leftarrow A$ //copy the arc set to the device
  do in parallel on the device using $|V|$ threads: init. $C_d[1\cdots|V|]$, $F1_d[1\cdots|V|]$, $F2_d[1\cdots|V|]$ such that: $C_d[i] \leftarrow i$,
   $F1_d[i] \leftarrow$ true and $F2_d[i] \leftarrow$ false
  declare boolean $m$ //$m$ indicates if the labels are still changing
  declare boolean $m_d$ in device memory
  repeat
    copy $m_d \leftarrow$ false
    do in parallel on the device using $|V|$ threads:
      call LABEL_COMPONENTS_KERNEL2($V_d$,$A_d$,$C_d$,$F1_d$,$F2_d$,$m_d$)
    $F1_d \leftrightarrow F2_d$ //swap the frontier arrays for the next iteration
    copy $m \leftarrow m_d$
  until $m$= false //repeat until the colours are not changing any more
```

Algorithm 3 (*continued*)

Algorithm 3. Kernel 2: The second approach to labelling the components of an arbitrary graph in parallel on a CUDA-enabled device. This kernel operates on the vertex set V, running one thread for every vertex for a total of $|V|$ threads per kernel call. Like in the CPU reference implementation described in Algorithm 1, the kernel operates on a set of arcs but assumes that for every arc (i,j) an arc (j,i) exists as well.

```
  declare integer $C[N]$
  copy $C \leftarrow C_d$ //copy the colours to the host
  return $C$ //$C$ now contains the component ID for every vertex $v \in V$.
```

```
function LABEL_COMPONENTS_KERNEL2($V_d$,$A_d$,$C_d$,$F1_d$,$F2_d$,$m_d$)
declare integer $i \leftarrow$ thread ID queried from the CUDA runtime //equal to the vertex ID of $v_i$
if $F1_d[i]$ = true then
  $F1_d[i] \leftarrow$ false
  declare integer $idx_i \leftarrow V_d[i]$ //the start index of $A_i \subseteq A$ in $A_d$
  declare integer $idx_{i+1} \leftarrow V_d[i+1]$ //the start index of $A_{i+1} \subseteq A$ in $A_d$ (the upper bound for $v_i$)
  declare integer $c_i$,$c_j$ //the colours of vertex $v_i$ and $v_j$
  $c_i \leftarrow C_d[i]$ //read $c_i$ from device memory
  declare boolean $c_{i(mod)} \leftarrow$ false //indicates if $c_i$ has been changed
  for all $j \in A_i$
    $c_j \leftarrow C_d[j]$ //read the colour of the adjacent vertex $j$ from device memory
    if $c_i < c_j$ then
      call atomicMin($C_d[j]$,$c_i$)
      $F2_d[j] \leftarrow$ true //$v_j$ will be processed in the next iteration
      $m_d \leftarrow$ true
    else if $c_i > c_j$ then
      $c_i \leftarrow c_j$
      $c_{i(mod)} \leftarrow$ true
    end if
  end for
  if $c_{i(mod)}$= true then
    call atomicMin($C_d[i]$,$c_i$) //$c_i$ was changed and needs to be written to device memory
    $F2_d[i] \leftarrow$ true //$v_i$ will be processed again in the next iteration
    $m_d \leftarrow$ true
  end if
end if
```

Algorithm 4. The CUDA specific optimisation techniques used for Kernel 2. It shows the use of the shared memory and texture caches.

```
//the one-dimensional texture reference
texture<int, 1, cudaReadModeElementType> arcsTexRef;
cudaBindTexture (0, arcsTexRef, d_arcs, nArcs * sizeof(int));
. . .
__shared__bool localChanges;
if (threadIdx.x == 0) {//the first thread in the block
  localChanges = false;
}
//load the start index of the adjacency-list into shared memory
__shared__ int arcListIndices[64];//the block size is 64
if (tid <= nVertices) {tid is the unique thread ID
  arcListIndices[threadIdx.x] = vertices[tid];
}
__syncthreads ();
. . .
int arcListEndIdx;
```

(*continued on next page*)

Algorithm 4 (*continued*)

Algorithm 4. The CUDA specific optimisation techniques used for Kernel 2. It shows the use of the shared memory and texture caches.

```
if (threadIdx.x < blockDim.x − 1) {//all but the last thread in the block
  arcListEndIdx = arcListIndices[threadIdx.x + 1];//read from shared memory
} else {
  arcListEndIdx = vertices[tid + 1];//read from global memory
}
...
for (int arcIdx = arcListIndices[threadIdx.x];
  arcIdx < arcListEndIdx; ++arcIdx){//iterate over the adjacency-list
  // do a texture fetch to get the ID of the neighbour
  int nbrId = tex1Dfetch (arcsTexRef, arcIdx);
  // compare the colours of this vertex and its neighbour
  ...
}
__syncthreads ();
if (localChanges && threadIdx.x == 0) { //set the global flag if necessary
  *globalChanges = true;
}
```

4.3.3. Breadth-First-Search Kernel 3

This approach is based on the CUDA algorithm for a breadth-first search proposed by Harish and Narayanan [3]. Instead of calculating the cost to reach all vertices in the same component as a given source vertex, however, our implementation assigns the same colour label to all vertices in the same connected component for all components in the graph. Kernel 3 described here also avoids a race condition that can occur in Algorithm 2 of [3], where one thread can evaluate line 5 and therefore assume that the neighbour has not been processed yet before that vertex sets the evaluated flag to `true` in line 3. Our implementation uses two frontier arrays to solve this problem, one for the current iteration and one for the next iteration, which are swapped between kernel calls.

Kernel 3 initialises the colours array to 0, which indicates that the respective vertex has not been coloured yet. The breadth-first search starts from a source vertex $v_{src} \in V$ with a new colour value c_{src} and spreads this colour further through the component $C_i \subseteq V$, where $v_{src} \in C_i$, until all reachable vertices have been labelled. During iteration $s = \{1,2,3,\ldots\}$ all vertices that are connected to v_{src} through a shortest path of $s - 1$ edges are coloured with the same colour c_{src} and the vertices at distance s are flagged as active in the frontier array. Only vertices that are flagged as active are processed during an iteration of the kernel. A new breadth-first search is started for every vertex that has not been coloured during one of the previous searches and is thus in a different, disjoint component.

Performance: It takes exactly $d + 1$ iterations to label all vertices of a connected component, where d is the diameter of the component. Furthermore, it takes $|C|$ breadth-first searches to label the whole graph, where $|C|$ is the number of disjoint components. Because this kernel is based on a breadth-first search, it performs better when the number of components is low and the degree is uniformly high, as this means that the number of active vertices processed during an iteration increases more quickly than for a lower degree, utilising more of the processing units available in the CUDA device. However, it turns out that this kernel performs much worse than the other CUDA implementations for almost every type of graph as the results section shows. We will therefore not cover it in as much detail as the other kernels in this article.

The use of different data structures and optimisation techniques means that the resource usage differs among the kernels. Table 1 lists these differences.

5. Hypercubic mesh graphs

A d-dimensional hypercubic mesh can be thought of as a graph with a very simple and regular structure where each vertex is connected to $2d$ neighbours. The data structure for storing such hypercubic meshes is usually a simple array and the connected neighbour addresses can be explicitly calculated rather than stored. Labelling these hypercubic meshes no longer takes the form of finding groups of vertices that are connected together (as all vertices in the mesh are ultimately connected) but instead finding connected groups of vertices that share some common property or fulfill some requirement. This subset of general graph labelling is applicable to any connected-component labelling (CCL) problem where the data conforms to a hypercubic mesh structure such as domain counting in field equation simulations and in the $d = 2$ case can be used for image partitioning and computer vision.

As hypercubic meshes have a very regular and strict structure, labelling algorithms can be optimised to take advantage of this structure. This is especially useful for GPU programs as their performance depends heavily on minimal branches and regular memory access. As the exact number of neighbours along with their position in memory is known, more efficient

Table 1
The resource usage of Kernels 1, 2 and 3 for arbitrary graphs (CUDA 2.1). None of the kernels uses any local memory or constant program variables. A link is either an undirected edge (Kernel 1) or a directed arc (Kernels 2 and 3), depending on the way the kernel represents the connections between vertices. The performance with different block sizes was measured for all multiples of 32 up to 512 and the best one was selected. This may be different on other hardware than our NVIDIA GTX260.

	Kernel 1	Kernel 2	Kernel 3
Global device mem. per vertex (bytes)	4	10	10
Global device mem. per link (bytes)	8	4	4
Other global device mem. (bytes)	1	5	5
Shared mem. per thread block (bytes)	33	304	53
Register usage per thread	4	9	5
Compiler generated constant mem. (bytes)	8	12	12
Thread block size	32	64	192

memory access patterns can be created allowing the speed of the labelling process to be greatly increased. We present, discuss and compare several hypercubic mesh labelling implementations on GPUs using NVIDIA's CUDA.

5.1. Connected-component labelling hypercubic meshes

Connected-component labelling algorithms in the 2-dimensional case have been widely studied for their use in image partitioning. Image partitioning is extremely useful in pattern recognition, computer vision and scientific simulation analysis. The real-time requirements of such applications continues to push the requirements for image partitioning algorithms to execute in the minimum time possible. Much of the research into two-dimensional CCL algorithms has been focused on optimising CPU CCL algorithms. Wu et al. [30] present a detailed description of CPU CCL algorithms and their optimisations that allow for very fast labelling of images. These algorithms can easily be extended to find connected components in higher-dimension meshes. In general, CCL algorithms can be classified as: multi-pass, two-pass or one-pass [30].

Multi-pass algorithms operate by labelling connected components over multiple iterations. These algorithms usually perform some local neighbour operations during scans through the mesh. The operations take the form of setting the label of the cell to the lowest label of the neighbours or record an equivalence between two labels. The performance of the multi-pass algorithms is highly dependent on the number of iterations through the data. However, some methods require very few iterations. The multi-pass method described in [31] has been shown to label 512×512 images in no more than four scans.

Two-pass methods work by detecting equivalences, resolving equivalence chains and then assigning a final label to each cell. The *Scanning phase* performs a single iteration through the image examining the neighbouring cells and recording equivalences between labels. The *Analysis phase* resolves equivalence chains (sequences of equivalent labels) to find the correct final label for each label. The *Labelling phase* scans through the image looking up the current label in the equivalence table and assigning it the correct final label. These methods generally have very high performance as they must scan through the data only twice. However, the performance is very dependent on how quickly the *Analysis phase* can resolve the label equivalences. Storing an entire equivalence structure is also very memory consuming.

One-pass is the last method of labelling. It works by scanning through the image until it finds a cell that does not yet have a label assigned to it. When one is discovered, it is assigned a new label and will then assign the same label to all the cells that are connected to it. This method has the advantage of only requiring one pass through the data, however, it does require irregular data access patterns to assign labels to connected cells (this is often performed recursively). This process of assigning label significantly reduces the performance of the algorithm and in general causes one-pass algorithms to be slower than the other two methods.

These classifications of CCL algorithms applies to both CPU and GPU implementations. We expect to find a difference in the optimal method for GPUs as compared to CPUs because the cost of parallel GPU iteration through the mesh should be less than a serial CPU iteration. Some implementations of CCL methods are also not possible on GPU due to the extra memory restrictions and parallel nature of GPU architectures. We have implemented and tested several GPU CCL algorithms to determine the optimal approach for GPU CCL.

5.2. CUDA implementations

As many scientific applications are being moved onto the GPU, we wish to perform CCL on the GPU without having to copy it back to the CPU. Thus we wish to find the fastest CCL algorithm for GPUs. We expect that a multi-pass method such as the one described by Suzuki et al. in [31] may execute faster on a GPU than a two-pass method because of the parallel nature of GPUs. Because an iteration through the data can be performed faster on a GPU than on a CPU, performing several iterations is no longer as much of an issue. The larger memory requirements and more irregular memory access of a two-pass labelling method will have a larger impact on the performance of a GPU algorithm than on a CPU because GPUs are highly optimised for regular memory transactions. We implement several multi-pass methods as well as a two-pass method and compare their results.

As hypercubic meshes have clearly defined structures and neighbours, there is no need to store neighbour lists separately. Instead the position of a cell's neighbours can be calculated explicitly. We can also make use of specialised memory types available on the GPU to improve memory access. In our previous work [29] it was shown that the optimal memory type to use for simulating field equations is the texture memory type. The texture memory type uses an on-chip memory cache that caches values from global memory in the spatial locality. This allows threads that access the values from memory in the same spatial locality to read from the texture cache rather than the slow global memory. The memory access patterns for CCL algorithms are very similar to the patterns for a field equation simulation and texture memory was found to perform the fastest for all CCL algorithms. Only the texture memory implementations of the Kernels are discussed.

5.2.1. Kernel A – Neighbour Propagation

Kernel A implementations a very simple multi-pass labelling method. It parallelises the task of labelling by creating one thread for each cell which loads the field and label data from its cell and the neighbouring cells. The threads will then find the cell with the lowest label and the same state as its allocated cell. If the label is less than the current label, the cell will be updated with the new lower label. Fig. 6 shows several iterations of this algorithm.

For each iteration, each cell will get the lowest label of the neighbouring cells with the same property. The kernel must be called multiple times until no label changes occur, then it is known that every cell has been assigned its correct final label. To determine if any label has changed during a kernel call, a global boolean is used. If a kernel changes a label it will set this boolean to true. The host program can then examine this boolean value to determine if another iteration of the algorithm is required. The host and device algorithms for this method can be seen in Algorithm 5.

Algorithm 5. The algorithm for the host and the kernel for the Neighbour Propagation CCL method. The host will call the kernel until m_d is not set to true at which point the mesh will have the correct labels.

```
function Mesh_Kernel_A_Host($D_d$,$N$)
  declare integer $L_d[N]$
  do in parallel on the device using $N$ threads: initialise $L_d[0 \cdots N-1]$ such that $L_d[i] \leftarrow i$
  declare boolean $m_d$ in device memory
  repeat
    do in parallel on the device using $N$ threads: call Mesh_Kernel_A($D_d$,$L_d$,$m_d$)
  until $m_d$= false
  return
function Mesh_Kernel_A($D_d$,$L_d$,$m_d$)
  declare integer $id$, $id_n$, $label$ in local memory
  declare integer $n_{id}[dim \times 2]$ in local memory
  $id \leftarrow$ threadID & blockID from CUDA runtime
  $n_{id} \leftarrow$ hypercubic neighbours of $id$ where the cells have the same state
  $label \leftarrow$ L[$id$]
  for all $id_n \in n_{id}$
    if L[$id_n$] <$label$
      $label \leftarrow L[id_n]$
      $m_d \leftarrow true$
    end if
  end for
  L[$id$] $\leftarrow label$
  return
```

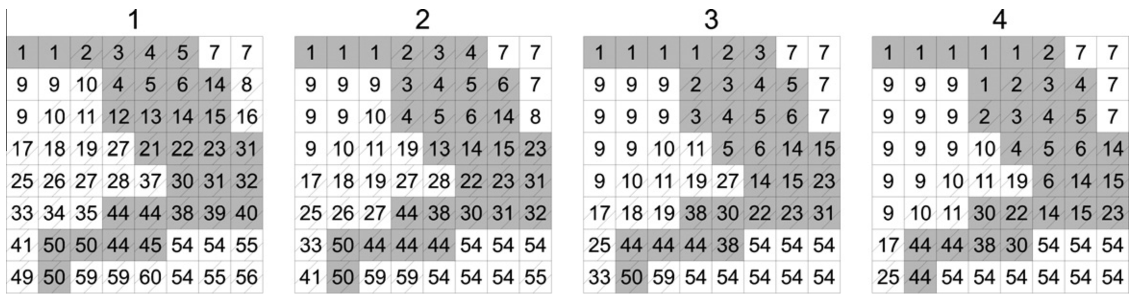

Fig. 6. Four iterations of the Neighbour Propagation algorithm. The colour shows the field state of the cell, the number shows the current label assigned to each cell. The lines through the cell shows that the cell does not yet have the correct final label.

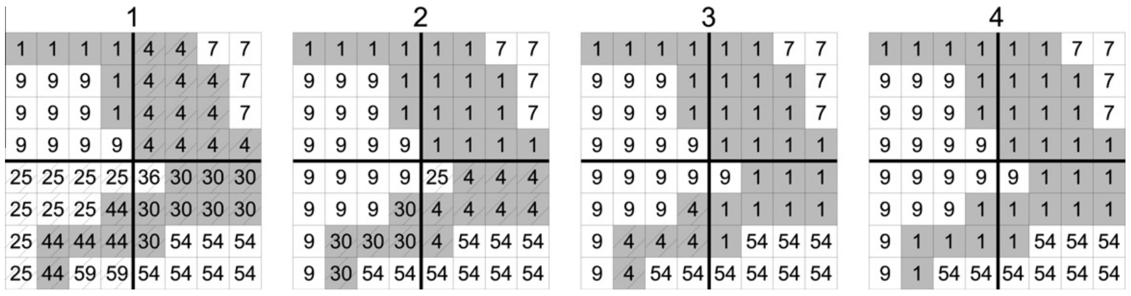

Fig. 7. Four iterations of the Local Neighbour Labelling algorithm. In this case the field is split into four blocks indicated by the heavy lines. Cells that are crossed out are yet to receive their final label.

Kernel A requires many iterations to label a mesh because a label can only propagate itself by a maximum of one cell per iteration. When there are large clusters in the mesh, the algorithm will require many iterations to label the entire mesh.

5.2.2. Kernel B – Local Neighbour Propagation

Kernel B is an improved version of Kernel A that allows labels to propagate themselves faster through a cluster by making use of Shared-Memory. Initially the threads fetch their cell's label and the labels of all four neighbours from global memory and choose the lowest of these (the same process as Kernel A). Instead of returning here, the threads then write these values into shared memory. The kernels then loop through updating the labels within shared memory until they stop changing. For each iteration of this loop the kernels will load the neighbours' labels from shared memory and choose the lowest appropriate label (see Algorithm 6). This effectively performs the entire labelling algorithm on each block at a time. The labels for four iterations of the kernel can be seen in Fig. 7. Note that the host code for Kernel B is effectively the same as for Kernel A.

Algorithm 6. The algorithm for Kernel B performing the Local Neighbour Propagation CCL method.

```
function Mesh_Kernel_B($D_d$,$L_d$,$m_d$)
  declare integer $id$, $id_L$, $label$ in local memory
  declare integer $n_{id}[dim]$ in local memory
  $id \leftarrow$ threadID & blockID from CUDA runtime
  $n_{id} \leftarrow$ hypercubic neighbours of $id$ where the cells have the same state
  $label \leftarrow L_d[id]$
  for all $id_n \in n_{id}$ then
    if $L_d[id_n] < label$ then
      $label \leftarrow L_d[id_n]$
      $m_d \leftarrow true$
  end if
end for
  declare integer $L_L[blockDim]$ in shared memory
  declare boolean $m_L$ in shared memory
  $id_L \leftarrow$ threadID from CUDA runtime
  $m_L \leftarrow true$
  $n_{id} \leftarrow$ neighbours of $id_L$ where the cells have the same state
  repeat
    $L_L[id_L] \leftarrow label$
    synchronise block
    $m_L \leftarrow$ false
    for all $id_n \in n_{id}$ do
        if $L_L[id_n] < label$ then
          $label \leftarrow L_L[id_n]$
          $m_L \leftarrow true$
        end if
    end for
    synchronise block
  until $m_L$ = false
  $L_d[id] \leftarrow label$
  return
```

As the labels on the borders of the block will change, multiple calls will be required for the entire mesh to be correctly labelled. However, Kernel B will require less iterations than Kernel A (although each kernel call will take longer). However, Kernel B is still limited in that a label can only propagate by the size of a block each iteration (Our block size is 16×16). For a large mesh where the use of GPUs is necessary (e.g. 4096×4096), this can still require many kernel calls.

5.2.3. Kernel C – Directional Propagation Labelling

Kernel C is designed to overcome the problem that a label can only propagate itself by one cell per iteration (Kernel A) or one block per iteration (Kernel B). Kernel C achieves this by breaking away from the method of allocating one thread to each cell. Instead, each thread is allocated a row (that is a row of cells in any of the dimensions of the hypercubic mesh). Each thread will iterate through this row propagating the lowest label along the row. By iteratively calling $2d$ such kernels (one going each direction for each dimension) the program can propagate the lowest labels through the mesh, see Fig. 8.

Each thread will calculate its unique position (with the dimension of iteration set to either 0 or $N-1$ depending on direction) and then iterate through the row. For each cell it will check to see if the last cell had the same state and if its label was lower, if this condition occurs the current cell will be assigned the last cell's label. This algorithm is described in detail in Algorithm 7.

Algorithm 7. The Kernel to iterate through the Ith dimension of a hypercubic mesh. This function must be called once for each direction in each dimension. Once all functions no longer set m_d to true, the mesh will be correctly labelled.

```
function Mesh_Kernel_C($I, D_d, L_d, m_d$)
  declare integer $id$, $label$ in local memory
  $id \leftarrow$ threadID & blockID from CUDA runtime
  $label \leftarrow L_d[id, 0]$ //$[id, 0]$ indexes the array at $id$ with the $I$th dimension component set to 0.
  for $n \leftarrow 1$ to $N-1$ do
    if $(D_d[id, n] = D_d[id, n-1])$&&$(label < L_d[id, n])$ then
      $L_d[id, n] \leftarrow label$
      $m_d \leftarrow true$
    else
      $label \leftarrow L_d[id, n]$
    end if
  end for
  return
```

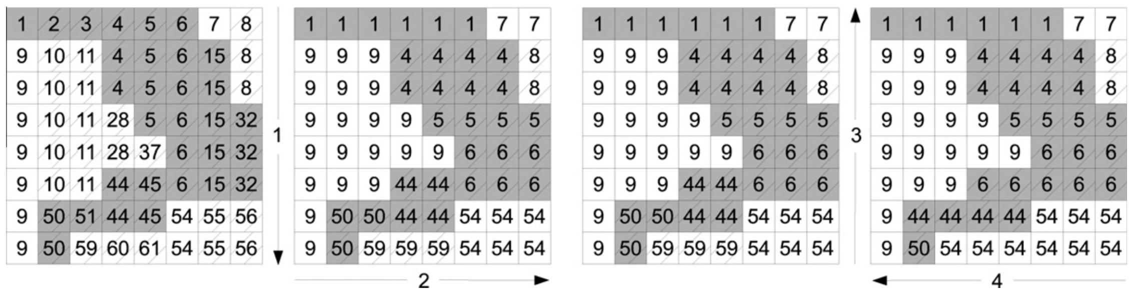

Fig. 8. One iteration of the Directional Propagation Labelling algorithm, one-pass for each direction in each dimension. Crossed out cells currently have an incorrect label.

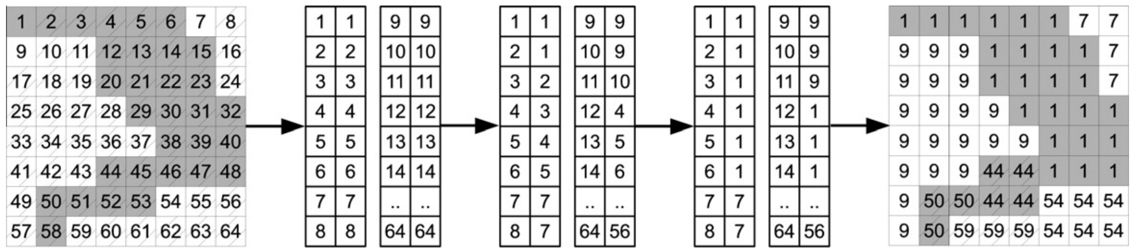

Fig. 9. A single iteration of the Label Equivalence method. From left to right: the initial scan and initial equivalence list, the list after the scan, the list after analysis and the mesh after labelling from the equivalence list. Crossed out lines indicate cells that have do not yet have their final label.

5.2.4. Kernel D – Label Equivalence

Kernel D is a multi-pass algorithm that records and resolves equivalences. However, unlike a two-pass algorithm this method does not record an entire equivalence table. Instead the kernel will record the lowest (and the most likely) label that each label is equivalent to. This reduces the memory complexity of the method while still allowing the equivalences to be resolved with a small number of iterations (4096×4096 systems were labelled in 9 or less iterations). This method is similar to the CPU algorithm described by Suzuki et al. in [31].

As with Kernel A and B, one thread is created for each cell. Each thread will examine the neighbouring labels and if it finds that it is next to a lower label, it will put that lower label into the equivalence list if the label is lower than the value currently in the list. At the end of this kernel call, the equivalence list will have a list of labels and the lowest other label that they are directly neighbouring, see Algorithm 8 and Fig. 9.

Algorithm 8. The algorithms for the host code and the three phases of Kernel D: scanning, analysis and labelling.

```
function Mesh_Kernel_D_Host($D_d$,$N$)
  declare integer $L_d[N]$, $R_d[N]$
  do in parallel on the device using $N$ threads: initialise $L_d[0 \cdots N-1]$ such that $L_d[i] \leftarrow i$
  do in parallel on the device using $N$ threads: initialise $R_d[0 \cdots N-1]$ such that $R_d[i] \leftarrow i$
  declare boolean $m_d$ in device memory
  repeat
    do in parallel on the device using $N$ threads: call Mesh_Kernel_D_Scanning_Phase($D_d$,$L_d$,$R_d$,$m_d$)
    do in parallel on the device using $N$ threads: call Mesh_Kernel_D_Analysis_Phase($D_d$,$L_d$,$R_d$)
    do in parallel on the device using $N$ threads: call Mesh_Kernel_D_Labelling_Phase($D_d$,$L_d$,$R_d$)
  until $m_d$ = false
  return
function Mesh_Kernel_D_Scanning_Phase($D_d$,$L_d$,$R_d$,$m_d$)
  declare integer $id$,$label_1$,$label_2$ in local memory
  declare integer $n_{id}[dim]$ in local memory
  $id \leftarrow$ threadID & blockID from CUDA runtime
  $n_{id} \leftarrow$ neighbours of $id$ where the cells have the same state
  $label_1 \leftarrow L_d[id]$
  $label_2 \leftarrow MAXINT$
  for all $id_n \in n_{id}$
    min($label_2$,$L_d[id_n]$)
  end for
  if $label_2 < label_1$ then
    atomicMin($R_d[label_1]$,$label_2$)
    $m_d \leftarrow true$
  end if
  return
function Mesh_Kernel_D_Analysis_Phase($D_d$,$L_d$,$R_d$)
  declare integer $id$, $ref$ in local memory
  $id \leftarrow$ threadID & blockID from CUDA runtime
  if $L_d[id] = id$ then
    $ref \leftarrow R_d[id]$
    repeat
      $ref \leftarrow R_d[label]$
    until $ref = R_d[label]$
    $R_d[id] \leftarrow ref$
  end if
  return
function Mesh_Kernel_D_Labelling_Phase($D_d$,$L_d$,$R_d$)
  declare integer $id$
  declare integer $ref$ in local memory
  $id \leftarrow$ threadID & blockID from CUDA runtime
  $L_d[id] \leftarrow R_d[L_d[id]]$
  return
```

The problem with this is that there will be equivalence chains, i.e. 16 is equivalent to 12 which is equivalent to 4 and so on. So there is another kernel that will resolve the equivalence lists so that the list will store the real lowest equivalent label. As each cell has a unique label at the start of the process, one thread is launched for each cell (which is the same as one thread for each initial label). These threads will retrieve the label currently assigned to that cell, it will then check to see if the label that cell has is the same as the label that would have been initially assigned to it. If the label is same then that thread is considered to own that label and will resolve the equivalence chain, if not it will simply return. The thread must then expand the equivalence chain to discover what the real lowest label it is equivalent to is. It will look up its label in the equivalence list and find the label it is equivalent to, it will then look up this label. This is repeated until the label it looks up in the list is equivalent to itself, this will be the lowest label in the equivalence chain. Once this value is found it overwrites the equivalence list with this value, see Algorithm 8.

This process sounds computationally expensive, however, most threads will not need to expand a long equivalence list. If a thread in the equivalence chain has already expanded and overwritten the equivalence list, other threads that have that value in their chain will not need to expand the entire chain as some part of it has already been done. While the order in which CUDA executes the threads is not controllable, the order of execution is known and repeatable. The scheduler assigns blocks to multiprocessors in order of block number. In general the threads with lower id's will be executed first and thus the equivalence list resolution process will remain relatively fast.

Once the equivalence chains have been expanded, the final stage of the labelling process can take place. One thread for each cell will be launched that will read that cell's label. Look up the label in the equivalence list and then assign the equivalent label to the cell, see Algorithm 8.

This process cannot label the entire mesh in a single pass and several iterations will be required. However, its method of combining labels rather than cells allows it to process large meshes with very few iterations. The advantage of this method over a two-pass system is that it does not require a large equivalence matrix (which uses too much memory) or linked-list processing (which is infeasible on a GPU).

5.3. CPU implementations

For comparison purposes we have also implemented two hypercubic mesh CCL algorithms on the CPU. These two algorithms are serial implementations of the algorithms we have previously discussed. The method CPU I is simply an alternating raster scan which updates the labels based on each cell's neighbours. The raster scan directions are alternated each iteration to ensure that labels are propagated in all directions, see Fig. 10. This raster scan CPU method is the CPU equivalent of Kernels A–C as it performs neighbour comparison operations each step (as with Kernels A and B) and the alternating scan directions are similar to the directional propagation method of Kernel C.

The labelling method CPU II uses the same algorithm as Kernel D, but simply finds and resolves the equivalences in serial on the CPU. Serial processing provides no advantage in the scanning or labelling phase, but can be more efficient in the analysis phase as the equivalence chains will never be more than two equivalences long. This method of labelling is very similar to the one described by Suzuki et al. in [31]. Although this method is not as fast as some CPU implementations of two-pass algorithms, it still provides a useful benchmark for comparing the GPU labelling methods.

6. Performance results

We now analyse the performance of the CUDA kernels and CPU reference implementations and compare them with each other. Table 2 briefly summarises the different implementations. Section 6.1 describes the test graphs used for the arbitrary graph performance analysis, which is given in Section 6.2. The results for the hypercubic mesh graphs are analysed in Section 6.3.

The system used for the performance measurements has the following configuration: an Intel Core2 Quad CPU running at 2.33 GHz and 4 GB of DDR2-800 system memory (≈2.8 GB accessible due to the use of a 32-bit operating system) with an NVIDIA GeForce GTX260 graphics card which has 216 CUDA cores and 896 MB of device memory. The Linux distribution Kubuntu 8.10 is used as the operating system. Version 2.1 of the CUDA toolkit and the corresponding developer drivers 180.22 for the GPU are installed on this system.

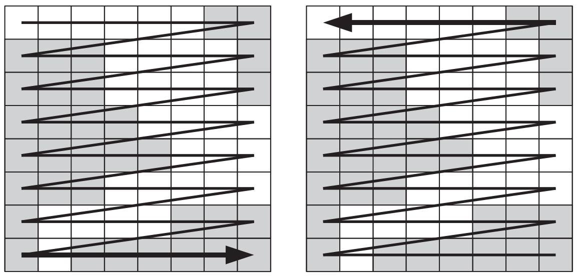

Fig. 10. The alternating raster scan patterns used by the CPU I method.

Table 2
This table gives a brief overview of the different CUDA implementations.

Arbitrarily structured graphs	
Kernel 1	Operates on the edge set. Compares the colour values of the two vertices connected by an edge and updates the higher one using `atomicMin`
Kernel 2	Operates on the vertices and their adjacency-lists. Uses a frontier array to flag vertices whose colour has been been changed in one iteration to be processed in the next iteration. When being processed, a vertex compares its colour value to those of its neighbours and updates the higher one using `atomicMin`
Kernel 3	Performs one breadth-first search for every connected component in the graph. Starting with a single vertex, it spreads the colour to all other vertices in the same component, processing all vertices at distance $s - 1$ from the source during iteration s
Hypercubic mesh graphs	
Kernel A	Neighbour Propagation – each iteration selects the lowest neighbouring label for each cell
Kernel B	Local Neighbour Propagation – each iteration resolves the labels within a block
Kernel C	Directional Propagation – each iteration propagates labels in a single direction
Kernel D	Label Equivalence – each iteration resolves the lowest (most likely) Label Equivalence chain and assigns all cells in the chain the final label

Table 3
This table lists the properties of the graphs that are used to analyse the performance of the different labelling algorithms. Disjoint 1 and 2 have 100 major components, most of the other components that are found in some of their graph instances are monomers (i.e. vertices with no neighbours). The line, scale-free and Watts–Strogatz (W–S) graph instances consist of only one component. They are used to determine how well the algorithms perform when they are used to verify that a graph is fully-connected. Furthermore, Barabási's scale-free and Watts–Strogatz's small-world graphs are well known network models that have been studied in many scientific publications and are therefore of particular interest. The abbreviation m stand for "millions".

	Vertices (m)	Edges (m)	Average degree	Components
Disjoint 1	2–16	50	6.3–50	100–31,044
Disjoint 2	1	10–60	20–120	100
Disjoint 3	5	60	24	1–1,000,019
Line	0.2–1	0.2–1	2	1
Scale-free	1–5	5–25	10	1
W–S, $p = 0.01$	2–14	10–70	10	1
W–S, $p = 0.1$	2–14	10–70	10	1
W–S, $p = 1.0$	2–14	10–70	10	1

6.1. Arbitrary test graphs

A number of graph instances are used to compare the performance of the CUDA kernels and to see how well they hold up against the CPU algorithm. Table 3 gives an overview of the graph properties.

Disjoint. This algorithm generates graphs with a specified number of major components. Vertices are assigned to these sets at random. Edges are created between randomly selected vertices from the same set. This is a variant of the Erdös–Rényi method of generating random graphs that differs in the initial partitioning of the vertices into different sets. The resulting graph has at least the specified number of components. It can have more components, most of them consisting of only one vertex (monomers). This is more likely to be the case when the number of edges in relation to the number of vertices is small.

Three different tests are performed with this algorithm, listed in the table as Disjoint 1, 2 and 3. The first one varies the number of vertices in the graph, while keeping the number of edges (50,000,000) and major components (100) fixed. The second one varies the number of edges, while keeping the number of vertices (1,000,000) and major components (100) fixed. And Disjoint 3 varies the number of major components ($10^0, 10^1, \ldots, 10^6$), while keeping the number of vertices (5,000,000) and the number of edges (60,000,000) fixed.

Line. A line graph is a simple one-dimensional graph where every vertex is connected to the vertex before and after itself. The first and last vertex are the only exceptions with only one neighbour. A line is the graph with the highest possible diameter $d = |V| - 1$. And as described in Section 4.3, the performance of the CUDA algorithms proposed in this paper is heavily dependent on the graph diameter. The line graph is the worst case scenario for them.

Scale-free networks have a power-law degree distribution for which the probability of a node having k links follows $P(k) \sim k^{-\gamma}$ [32]. Typically, the exponent γ lies between 2 and 3 [33,27]. Such a graph has a small number of vertices with a very large degree, also called the hub nodes, while most other vertices have a relatively small degree. The algorithm implements the Barabási–Albert model of gradual growth and preferential attachment described in [32], where new vertices are gradually added to the graph and connected to the already present vertices with a probability that is based on the degree of those vertices.

Watts–Strogatz. The Watts–Strogatz (W–S) small-world network model described in [34] is often used in the literature. It starts off with a regular graph where every vertex is connected to its k nearest neighbours. Then, one end of each edge is randomly rewired with probability p, considering every edge exactly once. The rewired edges are shortcuts or "long-distance"

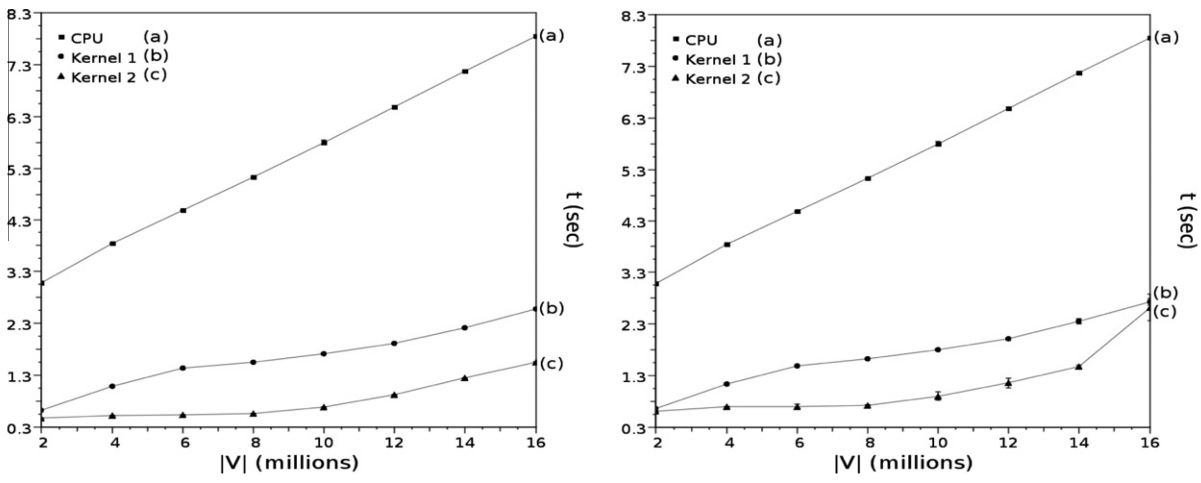

Fig. 11. The execution times t measured to process the **Disjoint 1** graphs excluding (left) and including (right) memory allocation, deallocation and host ↔ device memory copies. The number of vertices $|V|$ increases from 2,000,000–16,000,000. The number of edges and major components is set to 50,000,000 and 100 respectively. The execution times for Kernel 3 are in the range from ≈20 to 229 s.

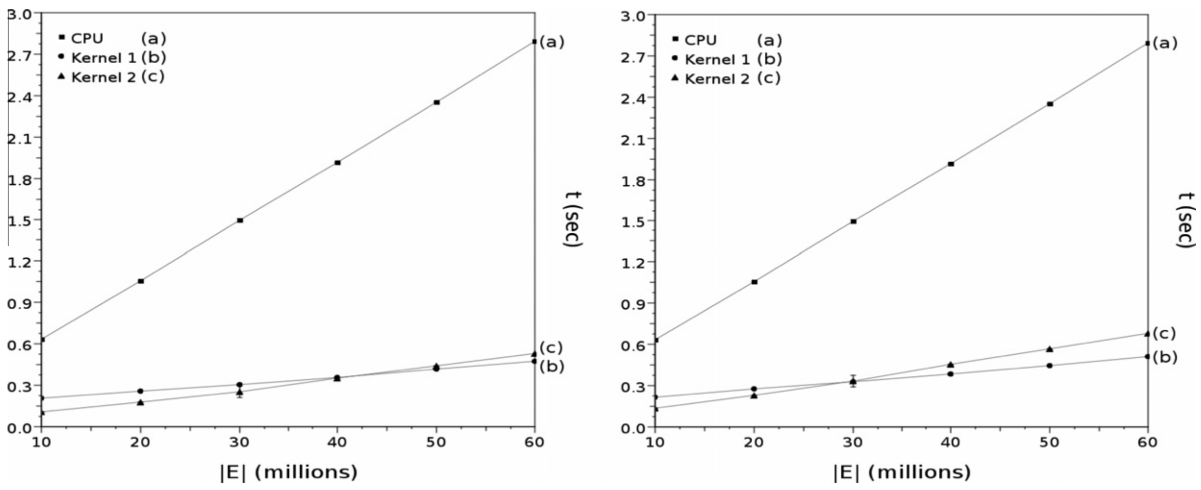

Fig. 12. The execution times t measured to process the **Disjoint 2** graphs excluding (left) and including (right) memory allocation, deallocation and host ↔ device memory copies. The number of edges $|E|$ increases from 10,000,000–60,000,000. The number of vertices and major components is set to 1,000,000 and 100 respectively. The execution times for Kernel 3 are in the range from ≈9.8 to 11.2 s.

links in the graph. For a large range of p, W–S have shown that the resulting networks have a high clustering coefficient and a small path length, which are characteristic properties of small-world networks.

6.2. Arbitrary graph performance analysis

This section discusses the performance of the different graph labelling algorithms using the graph instances described in the previous section as test cases. Each data point shown in the result plots given in Figs. 11–17 is the mean value of 30 measurements. The error bars illustrate the standard deviation from the mean. The performance results for Kernels 1 and 2, as well as for the CPU reference implementation, are shown for all of the graph instances. The implementation of Kernel 3 is much slower than the other approaches for all graphs with the exception of the line graphs. Its results have been omitted from all other result plots as they would otherwise dominate the plots and the performance differences between Kernels 1 and 2 and the CPU algorithm would get lost.

Most of the figures described in this section show two result plots. They illustrate, unless specifically mentioned otherwise, the CUDA kernel execution times without (left) or with (right) the time taken for device memory allocation, deallocation and memory copies between the host and device. The kernel execution times by themselves are of particular interest if the labelling algorithm is used in conjunction with other CUDA kernels that modify or analyse the same graph instance,

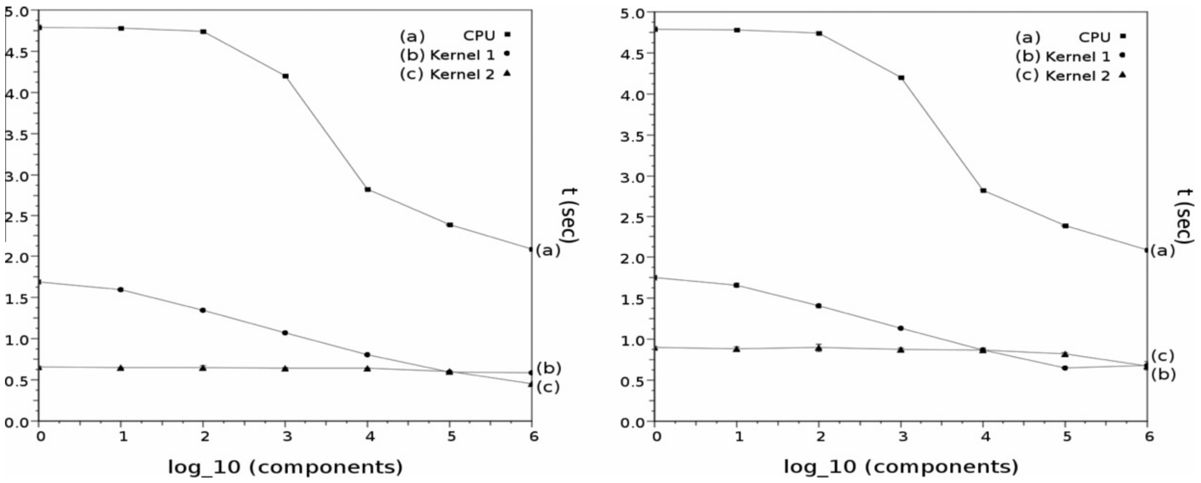

Fig. 13. The execution times t measured to process the **Disjoint 3** graphs excluding (left) and including (right) memory allocation, deallocation and host ↔ device memory copies. The number of major components increases from $10^0, 10^1, \ldots, 10^6$. The number of vertices and edges is set to 5,000,000 and 60,000,000 respectively. The execution times for Kernel 3 are in the range from ≈47 to 1515 s.

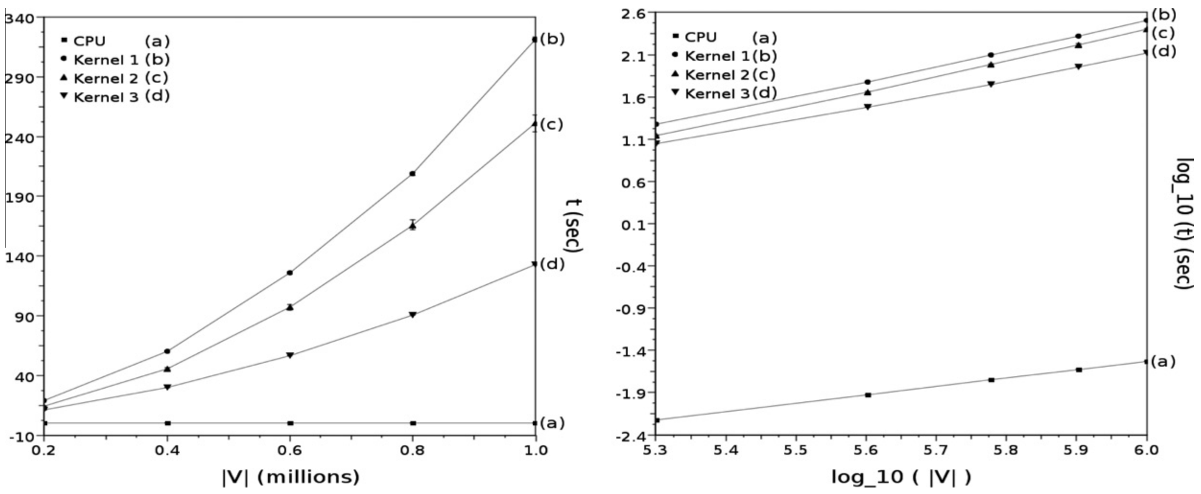

Fig. 14. The execution times t measured to process the **Line** graphs (left). Because the CUDA kernel execution times are so high and the graphs are rather small (compared to the other graphs used for the performance measurements), the device memory operations do not change the results in any significant way. The timing results increase logarithmically with the order of the graph $|V|$ (right), which increases from 200,000–1,000,000. The slopes of the least square linear fits to the data sets on the logarithmic plot, rounded to two significant digits, are: CPU ≈ 1.00, Kernel 1 ≈ 1.76, Kernel 2 ≈ 1.77 and Kernel 3 ≈ 1.57. This is the only graph type where Kernel 3 performs better than the other CUDA kernels. However, the CPU algorithm takes ≪1 s for all measured graphs of this type, beating the CUDA kernels by far.

which means that the data is already on the device. The plot on the right, on the other hand, is more relevant if only the component labelling is done on the device, which means that the graph data has to be copied to device memory before it can be processed.

Fig. 11 illustrates the graph labelling performance for the **Disjoint 1** graphs. The results demonstrate how even large graphs can be labelled in a matter of seconds. When it comes to the raw labelling performance, then Kernel 2 can clearly finish the task in the shortest amount of time. This is also true when the time for memory allocation, deallocation and host ↔ device memory copies is included. Only at the last data point, where $|V|$ = 16,000,000 (total device memory usage 443 MB for Kernel 1 and 534 MB for Kernel 2), does the mean value of the measurements of Kernel 2 make a sudden jump. The results are also much more spread out then before, as illustrated by the larger error bar. This behaviour can also be observed on some of the following result plots and will be discussed at the end of the section.

Disjoint 2 keeps the number of vertices consistently at 1,000,000, but increases the number of edges added between these vertices and therefore the average degree k of the graph instances. The performance measurements are illustrated

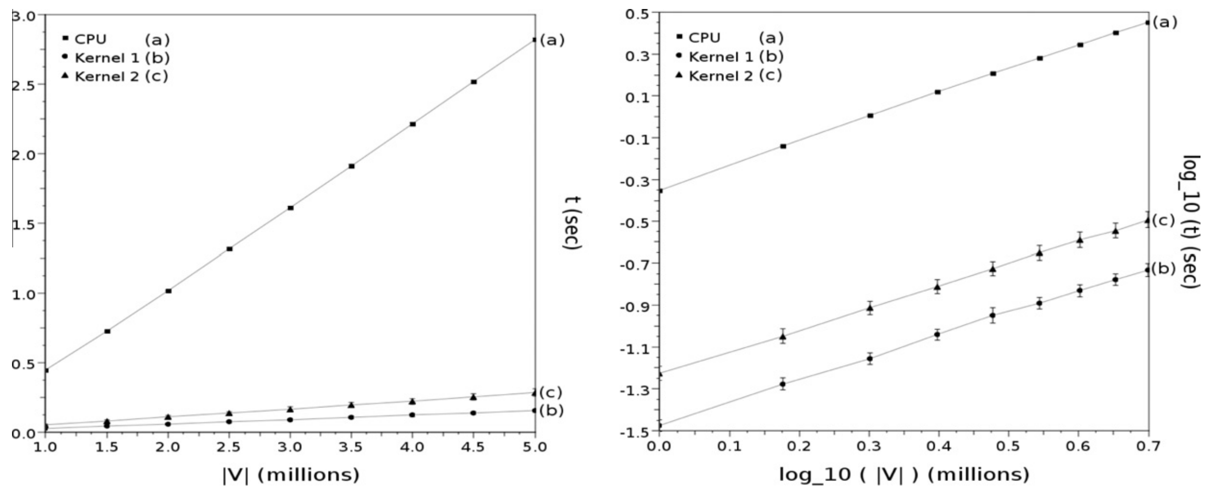

Fig. 15. The execution times t measured to process the **Scale-free** graphs excluding (left) and including (right) memory allocation, deallocation and host $\leftrightarrow$ device memory copies. The plot on the right is given on a logarithmic scale, illustrating that the results scale logarithmically with the order of the graph $|V|$, which increases from 1,000,000–5,000,000. The slopes of the least square linear fits to the data sets on the logarithmic plot, rounded to two significant digits, are: CPU ≈ 1.13, Kernel 1 ≈ 1.06 and Kernel 2 ≈ 1.05. The average degree for all graph instances is $k = 10$. The execution times for Kernel 3 are in the range from ≈ 9 to 47 s.

in Fig. 12. While Kernel 2 again shines for low degrees, Kernel 1 can outperform it once k exceeds ~ 90 when excluding and ~ 55 when including the memory allocation and copy times. Kernel 1 operates on the edge set E, whereas Kernel 2 operates on the arc sets that make up the adjacency-lists of the individual vertices. Even though Kernel 1 can not use some of the optimisation techniques used in Kernel 2, like the frontier array or texture fetches, it makes up for this once the degree is high enough by only having to traverse half as many connections than Kernel 2.

The performance measurements for **Disjoint 3**, illustrated in Fig. 13, show how the number of major components $(10^0, 10^1, \ldots, 10^6)$ effects the execution times of the algorithms. The number of vertices and edges is kept consistent at 5,000,000 and 60,000,000 respectively. Both the CPU algorithm and Kernel 1 finish the tasks significantly faster for a large number of components in the graph. The more components, the less vertices are in each component, which means the component diameters are smaller too. The execution times of Kernel 2 are much less effected by the number of components. Interestingly, the decrease of the execution times of Kernel 1 mostly stops at $\sim 10^5$, whereas the range from $\sim 10^5$ to 10^6 is where the execution times for Kernel 2 begin to decrease more noticeably.

The results for the **Line** graphs are shown in Fig. 14. As mentioned before, this is the worst case scenario for the CUDA kernels, because the diameter d of the graph is equal to $N - 1$. This requires the kernel to be called up to N times. The CPU implementation, on the other hand, can very quickly propagate the colour in $N - 1$ recursive calls from the first vertex to the last vertex.

Fig. 15 illustrates the performance measurements for the **Scale-free** graphs generated with Barabási's model. Kernel 1 and 2 both finish the labelling tasks considerably faster than the CPU implementation and scale well with the network size. The device memory allocation and host $\leftrightarrow$ device memory copies do not change the outcomes significantly.

Fig. 16 illustrates the performance results for the **Watts–Strogatz** graphs for three different rewiring probabilities. The CPU algorithm performs well for small values of p, but gets worse with increasing randomness, whereas Kernel 1 behaves the other way around. Data pre-fetching techniques employed by the CPU work better on regular, more predictable graphs than on the random data accesses that occur more frequently the higher the value of p is. Kernel 1, on the other hand, is much less affected by the storage locations of the vertices that form the end points of an edge. And it benefits a lot from the long-distance links created by rewiring edges, as these reduce the diameter of the graph.

The results for Kernel 2 are less consistent. Like Kernel 1, it benefits when p is increased from 0.01 to 0.1, as it also benefits from any reduction of the graph diameter. For $p = 1.0$, however, the performance goes back down again and is similar to the results obtained for $p = 0.01$. Graphs with $p = 1.0$ also appear to scale worse on Kernel 2 than the two lower values for p.

When the memory allocation, deallocation and host $\leftrightarrow$ device copies are included in the timings, then the CUDA kernels lose some ground compared to the CPU. Kernel 2 can again beat the other implementations for all values of p for graph instances up to $|V| \sim 12{,}000{,}000$ (total device memory usage is 503 MB for Kernel 1 and 572 MB for Kernel 2). Once the number of vertices exceeds this value, however, the mean performance suddenly drops and the results become very inconsistent for both Kernel 1 and Kernel 2.

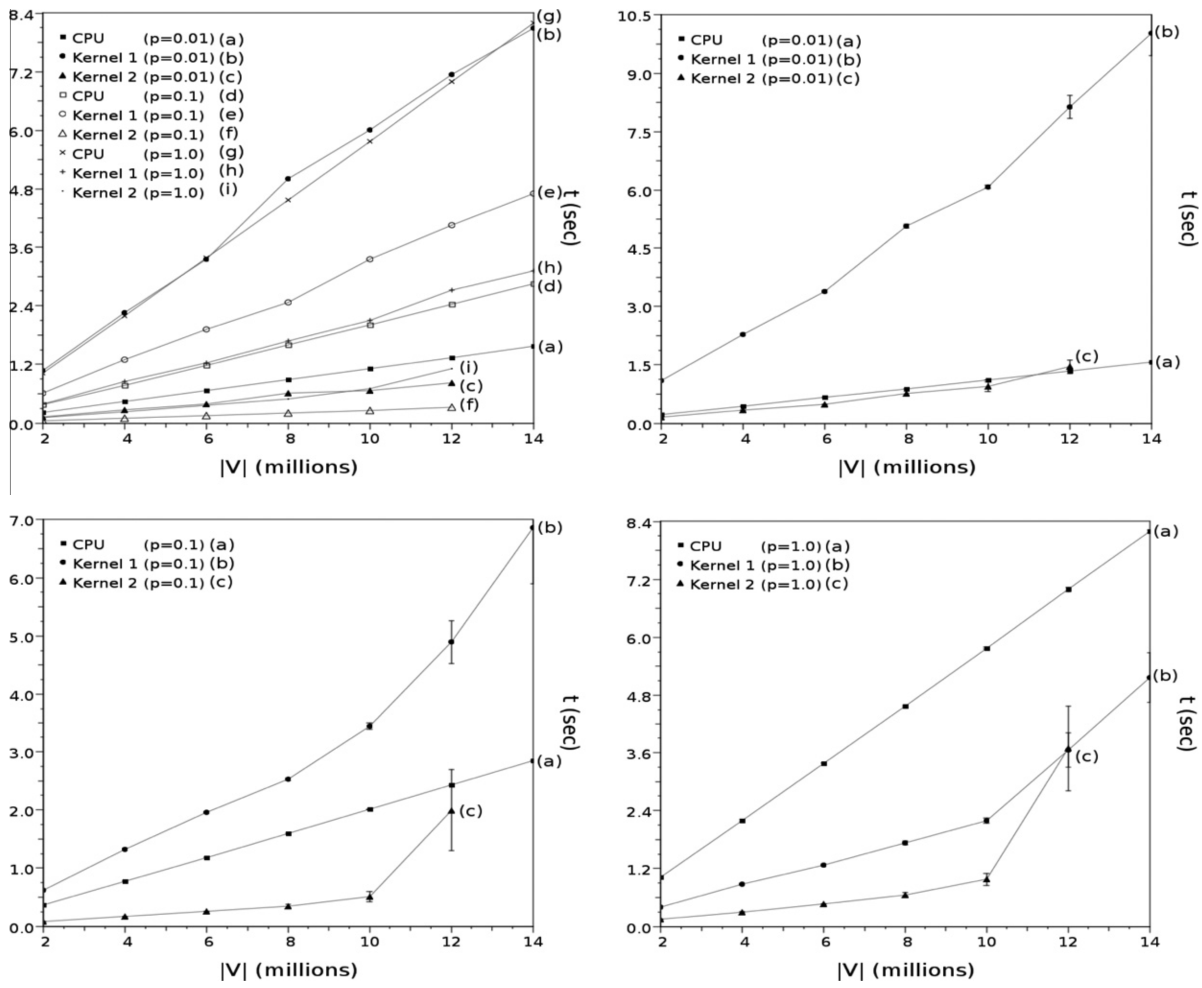

Fig. 16. The execution times t measured to process the **W–S** small-world graphs with the number of vertices $|V|$ increasing from 2,000,000–14,000,000. The average degree for all graph instances is $k = 10$. The plot on the top left shows the results for all three rewiring probabilities $p = \{0.01, 0.1, 1.0\}$ excluding memory allocation, deallocation and host ↔ device memory copies. The error bars in this plot are similar to the symbol size and are hidden to make it easier to distinguish the different symbols. The other three plots show the timing results for the different p-values including the time taken for memory allocation, deallocation and host ↔ device memory copies. No measurements with Kernel 2 are available for the graph instances with $|V| = 14{,}000{,}000$ ($|A| = |V| \times k$), as these graphs exceed the 2^{27} arc set size limit, which is due to the maximum size of a texture reference as described in Section 4.3.2. The execution times for Kernel 3 are in the range from ≈19 to 132 s for all p.

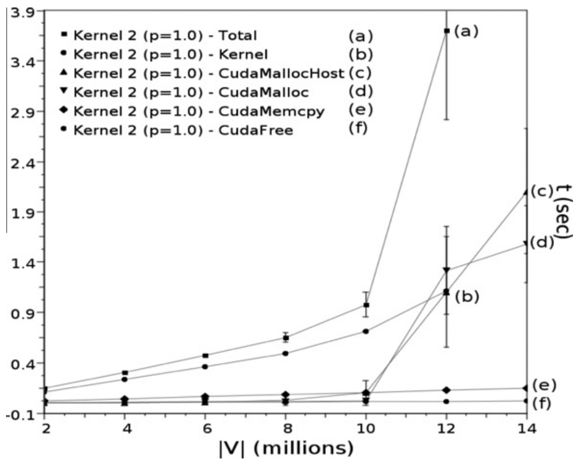

Fig. 17. The dissected execution times for Kernel 2 on the Watts–Strogatz graphs with $p = 1.0$. The datasets illustrate how long the individual CUDA operations (memory allocations, host to device memory copies, kernel executions and memory deallocations) take.

Now that we have looked at all the performance results, it is time to come back to the strange behaviour of the memory performance observed for large graphs with Kernel 1 and Kernel 2. Fig. 17 shows the dissected time measurements for Kernel 2 on the Watts–Strogatz graphs with $p = 1.0$ to give a better insight into what causes this sudden and inconsistent decrease of the result values. It is clearly the memory allocation operations *CudaMallocHost* and *CudaMalloc*. The former allocates page-locked memory on the host system, whereas the latter allocates memory on the device. Both the host and the device do not reach their memory limits yet (no swapping on the host). It is not clear why the memory allocation performance varies so much for large graphs, starting at $\sim$500 MB of total device memory usage on the test machine. This threshold may vary for different system configurations.

As expected, the CUDA profiler shows that the kernels are memory-bound and that non-coalesced memory accesses and divergent branches are the major problems when labelling arbitrary graphs. The kernel that can achieve the highest memory bandwidth for a given graph instance generally finishes the labelling task in the shortest amount of time.

6.3. Hypercubic performance analysis

The performance of the GPU labelling algorithms have been tested in two different ways. As the labelling algorithm was initially designed to find domains within phase-transition simulations, the algorithms were tested with data from such a simulation at various stages. We have also tested the algorithm on a generated spiral pattern – a test case designed to be specifically hard to label.

It should be noted here that although it was not described in this article, a two-pass GPU method was implemented and tested. However, the restrictions of the GPU kernels did not allow for linked-list based methods for resolving equivalences. This meant a large array was required to store the Label Equivalences (the equivalence table for 2D is N^4) which means the maximum power of two field length it can label (on a GTX 260 with 896 MB of memory) is $N = 64$. At this size it performed many times slower than the CPU implementation. The analysis phase of the two-pass algorithm was also hard to parallelise as many of the analysis phase algorithms depend on serial processing to find the correct set of equivalences. Although it was

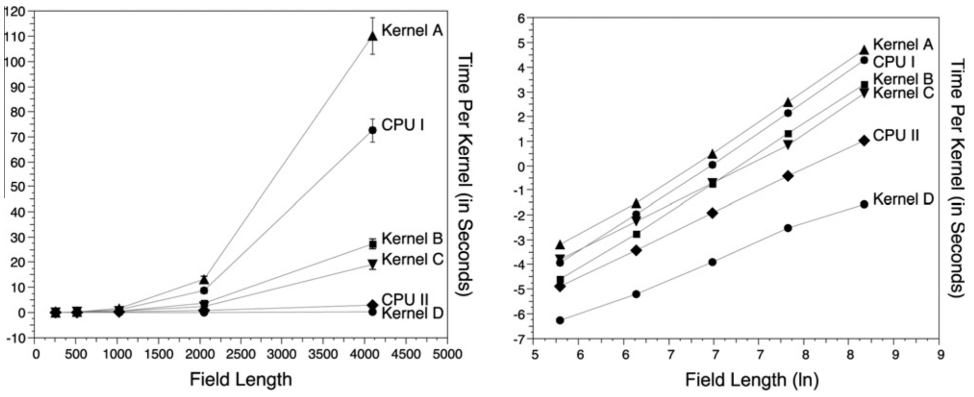

Fig. 18. Left: a comparison of the performance of the GPU and CPU CCL implementations labelling Cahn–Hilliard systems for field lengths $N = 256, 512, 1024, 2048, 4096$. Right: the same data presented in ln–ln scale.

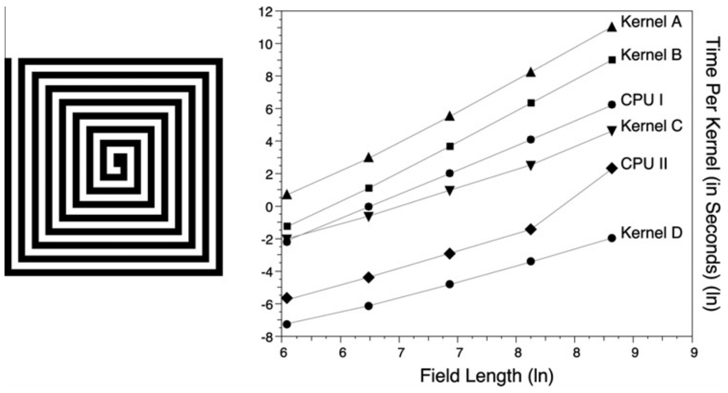

Fig. 19. Left: a sample Spiral image used for testing (32×32). Right: the performance results (in ln-ln scale) of the CPU and GPU CCL implementations labelling spiral images of size $N = 256, 512, 1024, 2048, 4096$.

possible to implement, the performance of the algorithm was very poor and could not complete most of the test sets (due to memory issues) used for the other methods, thus it has not been included in our performance analysis.

6.3.1. Field equation data

The labelling algorithms have been tested with the Cahn–Hilliard equation simulation to test their performance when operating in conjunction with a real simulation. The Cahn–Hilliard equation is a phase-transition model of a quenching binary alloy, component labelling can be used to determine which phase the field is currently in. The Cahn–Hilliard simulation used is the GPU-based simulation described in [29]. By testing the labelling algorithms in conjunction with a real GPU simulation, it will provide performance data of the algorithms as they would be used in a real situation. This will ensure that there are no unexpected effects of simulation and labelling kernels interfering with each other.

The simulation was ran for 16384 time-steps and the labelling algorithm was called every 128 steps. This is a common method of sampling how the number of clusters evolves over time. It also tests the effectiveness of the different algorithms at labelling fields from the initial random state to the interconnected clusters of various sizes after phase separation. The time per labelling call is averaged over the simulation. A plot of the results can be seen in Fig. 18.

Of the GPU Kernels A-C, only Kernel A performed worse than its equivalent CPU algorithm (CPU I). CPU I does not have the problem that each label can only be propagated by one cell per iteration, one major benefit of serial computation. With their optimisations, Kernels B and C provide similar performance gains over CPU I with Kernel C performing slightly better. It is interesting to note a cross-over between Kernels B and C in the ln-ln plot. At $N = 1024$ the performance lines of Kernels B and C intersect. This shows that the directional propagation algorithm of Kernel C is only more effective after the field reaches a large enough size ($N > 1024$).

Kernel D provides the best performance of any method outperforming the CPU II implementation of the same algorithm. In this case the algorithm can be decomposed to make use of the parallel processing power of the GPU. It can be seen from the ln-ln plot that Kernel D performs the best for every field length. At $N = 4096$ Kernel D provides a 13x speedup over CPU II method. This is not on the same scale as many GPU speedup factors but still provides an important performance gain.

6.3.2. The spiral model

The spiral data pattern is particularly hard to label as is contains two clusters that are each one pixel wide. This makes labelling extremely hard, especially for Kernel A which can only propagate a label by one cell each iteration. The algorithms have been tested with several $N \times N$ images with $N = \{256, 512, 1024, 2048, 4096\}$. A sample spiral image along with the performance results for the GPU Kernels along with the CPU implementations can be seen in Fig. 19.

The results from the spiral test case show that the labelling methods react differently as compared with the simulation results. The performance of Kernel D and CPU II when labelling a spiral was very similar to labelling a Cahn–Hilliard simulation. In fact Kernel D's labelling time for a spiral was actually less than the time for a simulation field.

The performance of the other Kernels (A–C) and CPU I were severely impacted. Kernel A was the most strongly affected by the spiral structure and performed the worst out of any method, for a 4096×4096 spiral was over $450{,}000\times$ slower than Kernel D. This extremely poor performance is a combination of the algorithm and the spiral, the time required for Kernel A to label a 4096×4096 Cahn–Hilliard system was over 600x less than for a spiral of the same size.

7. Discussion and conclusions

We have described our data-parallel algorithms and CUDA implementations of graph component labelling for the cases of arbitrarily structured graphs and hypercubic data graphs with regular neighbour related connectivities. We have measured execution times and reported performance gains using the NVIDIA GeForce 260 GPU card on 2.33 and 2.66 GHz Pentium Quad processors running (Ubuntu) Linux.

In the case of arbitrarily structured graphs, we found that the best CUDA implementation depends on the input graph. We do observe however that Kernel 2, which operates on the vertices and their adjacency-lists, performs best in most cases; Kernel 1, which operates on the edge set, performs better for graphs with high connectivity degrees.

We found that the best CUDA algorithm for a given graph can beat the CPU algorithm in most cases by a factor of up to around an order of magnitude. Our data suggest that a 5–10 times speed-up is likely in most cases for the largest graph instance when the data is already on the device. Slightly less speed-up is attained when the data has to be specifically copied to the device to do the labelling. Our CUDA kernels are much slower than the CPU when processing the line graph – not surprisingly given its structure and memory following behaviour. This was clearly the worst case scenario for our CUDA kernels. We found that in general, the breadth-first search based algorithm (Kernel 3) is not a good choice for the component labelling problem on CPU or GPU systems.

We also note that execution time for the CUDA memory management operations – `cudaMallocHost` and `cudaMalloc` – become inconsistent once the required amount of data becomes large – roughly 500 MB on our test machine, but the exact value most likely depends on the computer system configuration.

For applications where some graph properties are already known, then it is possible to select the best algorithm for the problem. In general, we found that: if the graph diameter is very large, then use the CPU; if the average graph degree is very large, then use Kernel 1 and in all other cases use Kernel 2.

In our hypercubic graph component labelling results we still found some worthwhile speed-ups although not at the stellar levels we have obtained for other applications. Cluster component labelling (CCL) algorithms often depend heavily on the serial memory traversal nature of CPU computation as well as the tree-based equivalence resolution methods. As neither are possible on the GPU, less efficient labelling (but parallelisable) algorithms must be used.

While we were unable to provide a dramatic speedup over CPU methods, the GPU implementation of our CCL algorithm for hypercubic data was comparable to the results described by Wu et al. in [30]. Unfortunately we were unable to test our algorithm with precisely the same test data as those authors, but for the same field length our GPU implementation indicates a slight performance increase. The main advantage that Kernel D provides over CPU II is that the simulation data can remain completely in the GPU memory and does not need to be copied out to host memory every time the field needs to be labelled.

This is useful for the sort of partial differential equation field models we simulate and wish to component-label. The main advantage for hypercubic data is that we avoid use of the memory hungry data structures necessary for arbitrarily structured graphs, thus allowing more physical memory available to support exploration of larger model data sizes. Graph cluster component labelling plays an important role in some of the cluster updating algorithms used in some Monte Carlo physics model simulations and we plan to employ our findings from this work to speed up phase-transition location work in this area [35]. In the introduction we presented some preliminary results of this work to motivate fast CCL in simulations.

We also anticipate emerging new GPU cards that contain multiple GPU devices and which can be used to serve separately threaded slave tasks of a CPU hosted application program. This approach will at worst allow CCL to be farmed out as a completely separate task from the simulation but at best may allow further problem decomposition so that different devices tackle different parts of the CCL problem. Finally, although we have developed our algorithms and implementations specifically for NVIDIA hardware and CUDA language, we anticipate they will port to the forthcoming OpenCL vendor/platform independent data-parallel language.

References

[1] K. Kawasaki, Diffusion constants near the critical point for time dependent Ising model I, Physical Review 145 (1) (1966) 224–230.
[2] V.C. Barbosa, R.G. Ferreira, On the phase transitions of graph coloring and independent sets, Physica A 343 (2004) 401–423.
[3] P. Harish, P. Narayanan, Accelerating large graph algorithms on the GPU using CUDA, in: S. Aluru, M. Parashar, R. Badrinath, V. Prasanna (Eds.), High Performance Computing – HiPC 2007: Proceedings of the14th International Conference, Goa, India, vol. 4873, Springer-Verlag, 2007, pp. 197–208.
[4] A.H. Gebremedhin, F. Manne, Scalable Parallel Graph Coloring Algorithms, Concurrency: Practice and Experience 12 (2000) 1131–1146.
[5] E.G. Boman, D. Bozdag, U. Catalyurek, A.H. Gebremedhin, F. Manne, A scalable parallel graph coloring algorithm for distributed memory computers, in: Proceedings of Euro-Par 2005 Parallel Processing, Springer, 2005, pp. 241–251.
[6] C.F. Baillie, P.D. Coddington, Cluster identification algorithms for spin models – sequential and parallel, Concurrency: Practice and Experience 3 (C3P-855) (1991) 129–144.
[7] C.F. Baillie, P.D. Coddington, Comparison of cluster algorithms for two-dimensional Potts models, Physical Review B 43 (1991) 10617–10621.
[8] R. Dewar, C.K. Harris, Parallel computation of cluster properties: application to 2d percolation, Journal of Physics A – Mathematical and General 20 (1987) 985–993.
[9] K. Hawick, W.C.-K. Poon, G. Ackland, Relaxation in the dilute Ising model, Journal of Magnetism and Magnetic Materials 104–107 (1992) 423–424. International Conference on Magnetism, Edinburgh, 1991.
[10] P.M. Flanders, Effective use of SIMD processor arrays, in: Parallel Digital Processors, IEE, Portugal, 1988, pp. 143–147, iEE Conf. Pub. No. 298.
[11] W.D. Hillis, The Connection Machine, MIT Press, 1985.
[12] R. Gould, Graph Theory, The Benjamin/Cummings Publishing Company, 1988.
[13] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, second ed., MIT Press, 2001.
[14] D. Brelaz, New methods to color the vertices of a graph, Communications of the ACM 22 (4) (1979) 251–256.
[15] J.A. Gallian, A dynamic survey of graph labeling, The Electronic Journal of Combinatorics 16 (2005) DS6.
[16] W. Press, B. Flannery, S. Teukolsky, W. Vetterling, Numerical Recipes in C, Cambridge University Press, 1988. pp. 252–254 (Chapter 12), Determination of Equivalence Classes.
[17] K. Hawick, D. Playne, Hypercubic storage in arbitrary dimensions, Tech. Rep., Massey University, 2009.
[18] P. Erdös, A. Rényi, On random graphs, Publicationes Mathematicae 6 (1959) 290–297.
[19] H. Jeong, B. Tombor, R. Albert, Z. Oltvai, A.-L. Barabsi, The large-scale organization of metabolic networks, Nature 407 (6804) (2000) 651–654.
[20] D.A. Fell, A. Wagner, The small world of metabolism, Nature Biotechnology 18 (11) (2000) 1121–1122.
[21] A. Wagner, D.A. Fell, The small world inside large metabolic networks, Proceedings of the Royal Society B 268 (1478) (2001) 1803–1810.
[22] M.E.J. Newman, The structure of scientific collaboration networks, PNAS 98 (2) (2001) 404–409.
[23] M.E.J. Newman, Ego-centered networks and the ripple effect, Social Networks 25 (1) (2003) 83–95.
[24] S. Milgram, The small-world problem, Psychology Today 1 (1967) 61–67.
[25] F. Liljeros, C. Edling, L. Amaral, H. Stanley, Y. Aberg, The web of human sexual contacts, Nature 411 (6840) (2001) 907–908.
[26] D. Liben-Nowell, J. Kleinberg, Tracing information flow on a global scale using Internet chain-letter data, PNAS 105 (12) (2008) 4633–4638.
[27] R. Albert, H. Jeong, A.-L. Barabsi, Diameter of the World-Wide Web, Nature 401 (6749) (1999) 130–131.
[28] NVIDIA®Corporation, CUDA®2.1 Programming Guide, 2009, <http://www.nvidia.com/> (accessed May 2009).
[29] A. Leist, D. Playne, K. Hawick, Exploiting graphical processing units for data-parallel scientific applications, Concurrency and Computation: Practice and Experience 21 (2009) 2400–2437. cSTN-065, doi:10.1002/cpe.1462.
[30] K. Wu, E. Otoo, K. Suzuki, Optimizing two-pass connected-component labeling algorithms, Pattern Analysis and Applications 12 (2009) 117–135, doi:10.1007/s10044-008-0109-y.
[31] K. Suzuki, I. Horiba, N. Sugie, Fast connected-component labeling based on sequential local operations in the course of forward raster scan followed by backward raster scan, in: Proceedings of the 15th International Conference on Pattern Recognition (ICPR'00), vol. 2, 2000, pp. 434–437.
[32] A.-L. Barabási, R. Albert, Emergence of scaling in random networks, Science 286 (5439) (1999) 509–512.
[33] D.J. d. Price, Networks of scientific papers, Science 149 (3683) (1965) 510–515, doi:10.1126/science.149.3683.
[34] D.J. Watts, S.H. Strogatz, Collective dynamics of 'small-world' networks, Nature 393 (6684) (1998) 440–442.
[35] K.A. Hawick, H.A. James, Ising model scaling behaviour on z-preserving small-world networks, Tech. Rep. arXiv.org Condensed Matter: cond-mat/0611763, Information and Mathematical Sciences, Massey University, February 2006.