Parallel Graph Component Labelling with GPUs and CUDA

K.A. Hawick[☆], A. Leist and D.P. Playne

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

Abstract

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.

Key words: graph, component label, mesh, GPU, CUDA, data-parallel

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. Figure 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.

There are a number of well-known algorithms for graph colouring [1] including parallel algorithms for speeding up graph calculations [2] as well as those specifically on graph colouring such as [3, 4]. In addition, a number of parallel component labelling applications have been described in the applications literature,

[☆]Author for correspondence

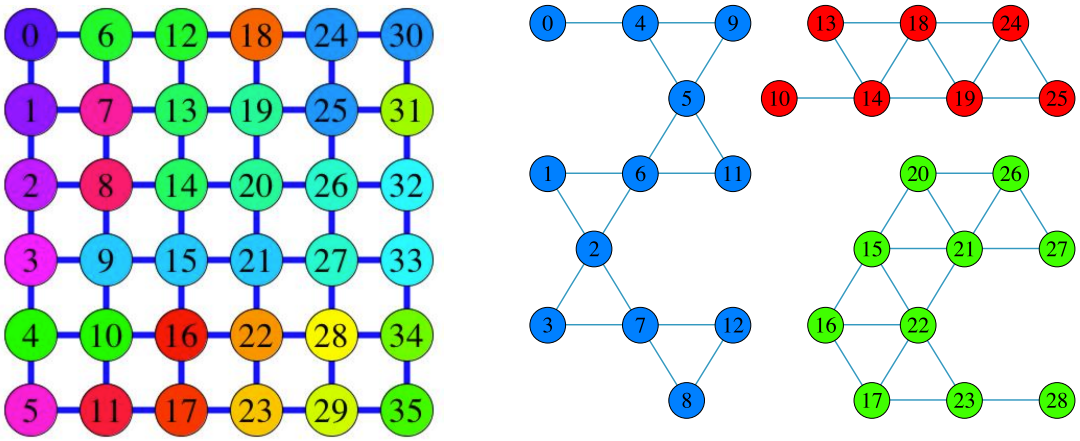


Figure 1: Component labelling on a regular mesh (left) - where each site has a known number of neighbours - and on an arbitrary graph (right). Different colours (shades) denote different components. The regular mesh has in this case 26 separate clusters, the largest with 3 nodes, the arbitrary graph has 3 separate components, the largest with 8 nodes.

particularly for use in lattice physics simulation models [5, 6], studies of percolation [7], and studies of other phase transitions [8].

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) [9] and the Connection Machine examples[10] 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 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 forthcoming Open Compute Language (OpenCL) promises similar capabilities and applications programmer accessibility.

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. 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 3) and more symmetric mesh and lattice graphs (Section 4) 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 5 and in Section 6 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 [11] graph labelling is a well known problem with relatively easy to understand (serial) algorithms [12]. Generally graph component labelling is a subset problem of the wider colouring problems [13] with a number of known algorithmic variations [14].

One perspective on the problem is in terms of equivalence classes and the popular formulation in Numerical Recipes books [15] 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.

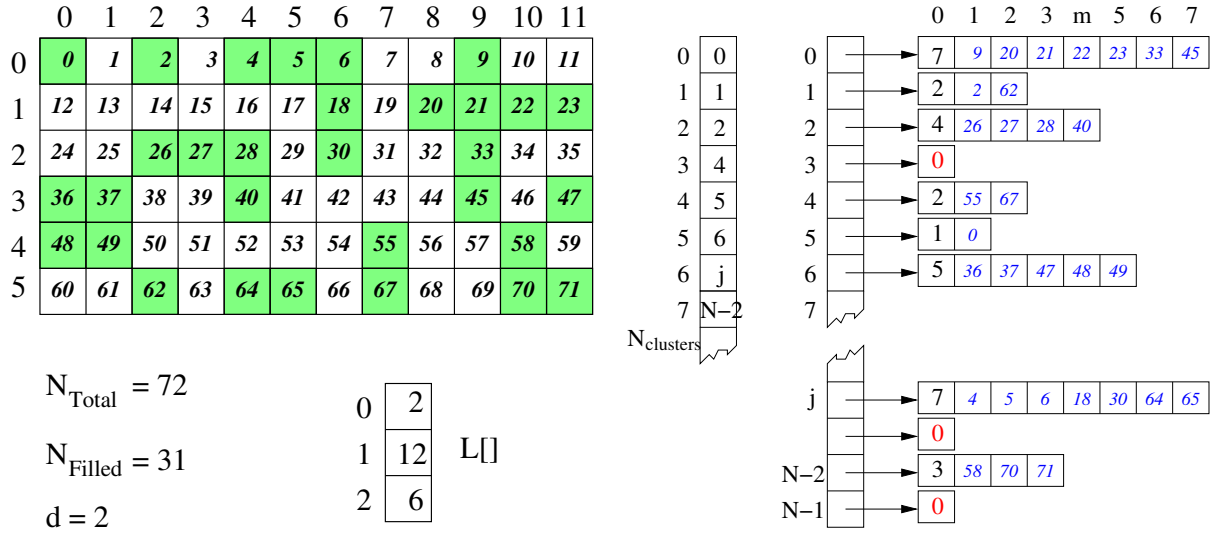


Figure 2: A 12×6 periodic nearest-neighbour mesh where the cluster "sameness" also depends on the variables decorating the graph nodes. Graph node $k = 0$ has exactly 4 nearest neighbouring nodes that it is adjacent to $(1, 11, 12, 60)$, but it has correctly been identified as the sole member of cluster $j = 5$ 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.

Figure 2 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 $(x_1, x_2, ..., x_d)$ in some order of significance [16].

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 on the right of Figure 2 - 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. 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 [17] and complex networks, such as metabolic networks [18, 19, 20], human collaboration networks [21, 22] and many other types of social networks [23, 24, 25], as well as computer networks like the World-Wide Web [26] 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.

3.1. 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. Figure 3 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 [27]. 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.

3.2. 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 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. 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

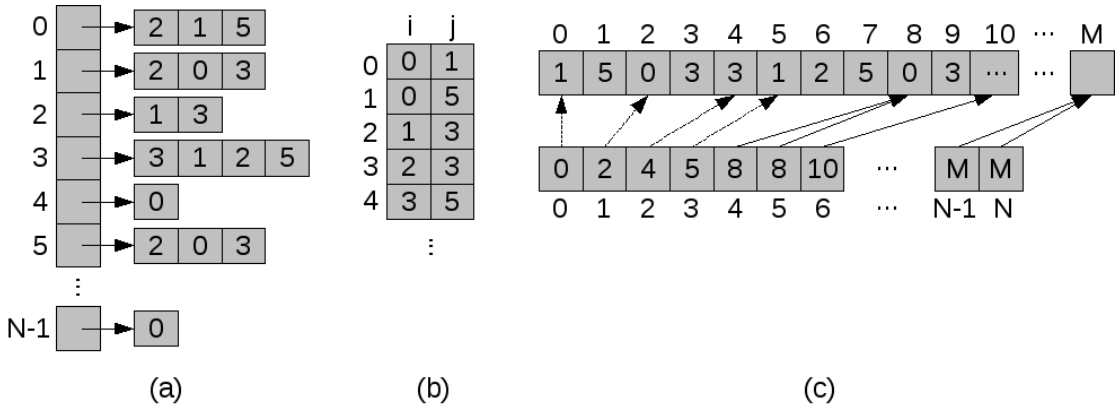


Figure 3: The data structures used to represent the graph by the CPU and GPU implementations. All three examples represent the same undirected graph. **(a)** shows the adjacency-list representation used by the CPU algorithm. Every vertex $v_i \in V$ stores a list of its neighbours at index i of the vertex 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. **(b)** 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. **(c)** 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.

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.

3.3. 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 Figure 3, 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.

3.3.1. 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. 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. 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

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 ∈ V. The
  arc set A_i ⊆ 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 ∈ V do
    C[i] ← i //initialise every vertex to a unique colour value
    F[i] ← true
  end for
  for all v_i ∈ 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 ∈ V. The number of distinct values
  equals the number of components in G.
```

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

true. This optimisation reduces the number of global memory writes and replaces them with much faster writes to shared memory.

3.3.2. 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

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 in G.
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 $C_d[1 \dots N]$ 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 $v \in V$. The number of distinct values equals the number of components in G.

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($C[j], c_i$) //atomicMin atomically replaces the value pointed to by its first parameter with the second parameter if it is smaller than the current value
  $m_d \leftarrow$ true
else if $c_j < c_i$ **then**
  call atomicMin($C[i], c_j$)
  $m_d \leftarrow$ true
end if

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 512MB (4 bytes per arc). As some graphics cards provide more device memory than 512MB, 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.

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 could be reduced if the vertices were sorted by their degree. However, we have shown in our previous research [28] that, depending on the network structure, the time to sort the vertices and even more the overhead of having to read the vertex ID from device memory, instead of being able to infer it from the thread ID, reduces the performance gain considerably or even slows the algorithm down. This optimisation technique has not been used here due to these mixed results, but may give a small speed-up for some graph structures.

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} ($V[i+1] - V[i]$). In order for this to work for 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 numer 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 with one additional element

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: initialise $C_d[1 \ldots |V|]$, $F1_d[1 \ldots |V|]$ and $F2_d[1 \ldots |V|]$ such that

$$C_d[i] \leftarrow i,\ F1_d[i] \leftarrow \text{true and } F2_d[i] \leftarrow \text{false}$$

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

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 $v \in V$. The number of distinct values equals the number of components in G.

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 //the thread ID is 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 $j \in A_i$ **do**
      $c_j \leftarrow C_d[j]$ //read the colour of the adjacent vertex j from device memory
      if $c_i < c_j$ **then**
        call atomicMin($C[j], c_i$) //atomicMin atomically replaces the value pointed to by its first parameter with the second parameter if it is smaller than the current value
        $F2_d[j] \leftarrow$ 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($C[i], c_i$) //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

3.3.3. Breadth-First-Search Kernel 3

This approach initialises the colours array to 0, which indicates that the respective vertex has not been coloured yet. This kernel is based on a breadth-first search, starting from a source vertex $v_{src} \in V$ with a new colour value c_{src} and spreading 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 $s = \{1, 2, 3, \dots\}$ 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 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.

CUDA specific optimisations: Kernel 3 uses the same flag in shared memory already described for the other two kernels. Unlike in Kernel 2, neither the shared memory array for the adjacency-list start indices nor the texture fetches improve the performance of this kernel - thus they are not used. The reason appears to be that the number of active vertices in each iteration is smaller compared to Kernel 2, meaning that much more adjacency-list start indices are fetched than actually needed, wasting bandwidth and thus the advantage of using the shared memory in the first place.

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;
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;
}
```

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

4. Hypercubic Mesh Graphs

A d-dimensional hybercubic mesh can be thought of as a graph with a very simple and regular structure where each vertex is connected to $2d$ neighbours. Labeling 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

Algorithm 5 Kernel 3: The third 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_KERNEL3_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} ($V[i+1] - V[i]$). In order for this to work for 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 with one additional element

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: initialise $C_d[1 \ldots |V|] \leftarrow 0$ and $F2_d[1 \ldots |V|] \leftarrow$ false

declare boolean m //m indicates if there are still vertices set to active in the frontier

declare boolean m_d in device memory

declare integer c_{curr} //the current colour

for all $v_i \in V$ **do**

  declare c_i //the colour of vertex v_i

  copy $c_i \leftarrow C_d[i]$ //only start a breadth-first search from v_i if it has not been coloured yet

  if $c_i = 0$ **then**

    $c_{curr} \leftarrow c_{curr} + 1$ //use a new colour for the vertices that are reachable from v_i such that $c_{curr} > 0$

    do in parallel on the device using $|V|$ threads: initialise $F1_d[1 \ldots |V|] \leftarrow$ false

    copy $F1_d[i] \leftarrow$ true //set the source vertex v_i to active

    repeat

      copy $m_d \leftarrow$ false

      do in parallel on the device using $|V|$ threads:

        call LABEL_COMPONENTS_KERNEL3($V_d, A_d, C_d, F1_d, F2_d, m_d, c_{curr}$)

      $F1_d \leftrightarrow F2_d$ //swap the frontier arrays for the next iteration

      copy $m \leftarrow m_d$

    until $m =$ false //repeat until all reachable vertices have been coloured

  end if

end for

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 $v \in V$. The number of distinct values equals the number of components in G.

function LABEL_COMPONENTS_KERNEL3($V_d, A_d, C_d, F1_d, F2_d, m_d, c_{curr}$)
  declare integer $i \leftarrow$ thread ID queried from the CUDA runtime //the thread ID is equal to the vertex ID of v_i
  if $F1_d[i] =$ true **then**
    $F1_d[i] \leftarrow$ false
    $C_d[i] \leftarrow c_{curr}$ //if the vertex is marked as active in the frontier, then set its colour to c_{curr}
    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)
    for all $j \in A_i$ **do**
      if $F1_d[j] \neq$ true AND $C_d[j] = 0$ **then**
        $F2_d[j] \leftarrow$ true //process v_j in the next iteration if it has not been coloured yet and if it is not being processed in this iteration
        $m_d \leftarrow$ true
      end if
    end for
  end if

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.

	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

heavily on minimal branches and regular memory access. As the exact number of neighbours along with their position in memory is known, more efficient 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.

4.1. Connected-Component Labeling 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 [29]. Much of the research into two-dimensional CCL algorithms has been focused on optimising CPU CCL algorithms. Wu Et al 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 [29].

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 [30] has been shown to label 512x512 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 *Labeling 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 it 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.

4.2. 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 [30] 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 GPU are highly optimised for regular memory transactions. We implement several multi-pass methods as well as a two-pass method and compare their results.

The performance of GPU applications are highly dependent on the memory type and access patterns used. In our previous work [28] showed 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.

4.2.1. Kernel A - Neighbour Propagation

Kernel A an implementation of 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. Figure 4 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

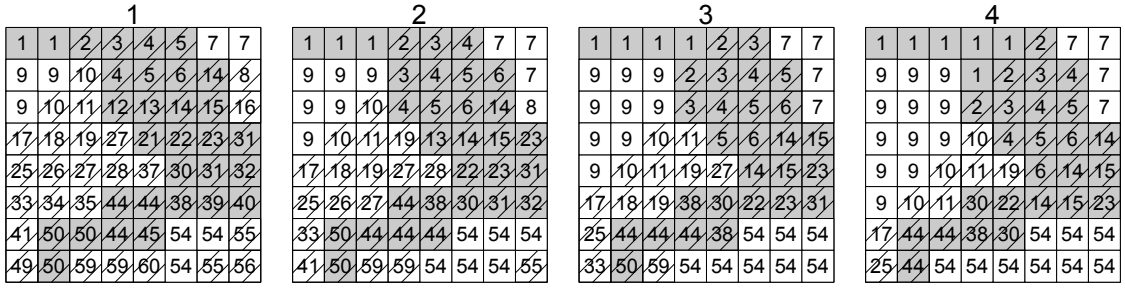


Figure 4: Four iterations of the Neighbour Propagation algorithm. The colour shows the field state of the cell, the number shows the current label assigned to each cell. The lines through the cell shows that the cell does not yet have the correct final label.

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 6.

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.

4.2.2. Kernel B - Local Neighbour Propagation

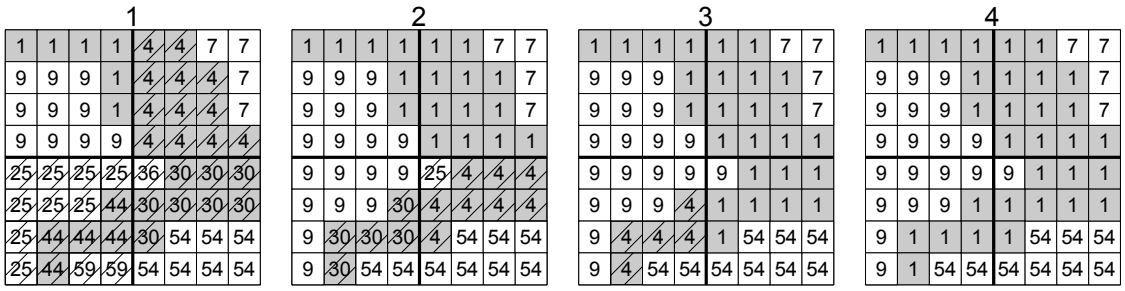


Figure 5: Four iterations of the Local Neighbour Labeling algorithm. In this case the field is split into four blocks indicated by the heavy lines. Cells that are crossed out are yet to receive their final label.

Kernel 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 7). 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 Figure 5. Note that the host code for Kernel B is effectively the same as for Kernel A.

As the labels on the borders of the block will change, multiple calls will be required for the entire mesh to be correctly labeled. 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 16x16). For a large mesh where the use of GPUs is necessary (eg. 4096x4096), this can still require many kernel calls.

Algorithm 6 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(E, N)
  declare integer $L_d[N]$
  do in parallel on the device using N threads: initialise $L_d[1 \dots N]$ such that $L_d[i] \leftarrow i$
  declare boolean m in host memory//m indicates if the labels are still changing
  declare boolean m_d in device memory
  repeat
    $m \leftarrow$ false
    copy $m_d \leftarrow m$
    do in parallel on the device using N threads: **call** Mesh_Kernel_A(D_d, L_d, m_d)
    copy $m \leftarrow m_d$
  until $m =$ false
  return
function Mesh_Kernel_A(D_d, L_d, m_d)
  declare integer $id, id_n, label_1, label_2$ in local memory
  declare integer $n_{id}[dim]$ in local memory
  $id \leftarrow$ threadID & blockID from CUDA runtime
  $n_{id} \leftarrow$ hypercubic neighbours of id where the cells have the same state
  $label_1 \leftarrow$ L[id]
  for all $id_n \in n_{id}$ **do**
    $label_2 \leftarrow L[id_n]$
    if $label_2 < label_1$ **then**
      $label_1 \leftarrow label_2$
      $m_d \leftarrow true$
    end if
  end for
  return

4.2.3. Kernel C - Directional Propagation Labeling

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 Figure 6.

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 8.

4.2.4. 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 (4096x4096 systems were labeled in 9 or less iterations). This method is similar to the CPU algorithm described by Suzuki Et Al in [30].

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

Algorithm 7 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_1, label_2 in local memory
  declare integer n_id[dim] in local memory
  id ← threadID & blockID from CUDA runtime
  n_id ← hypercubic neighbours of id where the cells have the same state
  label_1 ← L_d[id]
  for all id_n ∈ n_id do
    label_2 ← L_d[id_n]
    if label_2 < label_1 then
      label_1 ← label_2
      m_d ← true
    end if
  end for
  declare integer L_L[blockDim] in shared memory
  declare boolean m_L in shared memory
  id_L ← threadID from CUDA runtime
  m_L ← true
  repeat
    L_L[id_L] ← label_1
    synchronise block
    m_L ← false
    n_id ← neighbours of id_L where the cells have the same state
    for all id_n ∈ n_id do
      label_2 ← L_L[id_n]
      if label_2 < label_1 then
        label_1 ← label_2
        m_L ← true
      end if
    end for
    synchronise block
  until m_L = false
  L_d[id] ← L_L[id_L]
  return
```

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 9.

The problem with this is that there will be equivalence chains, ie 16 is equivalent to 12 which is equivalent to 4 etc. So there is another kernel that will update 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 9.

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

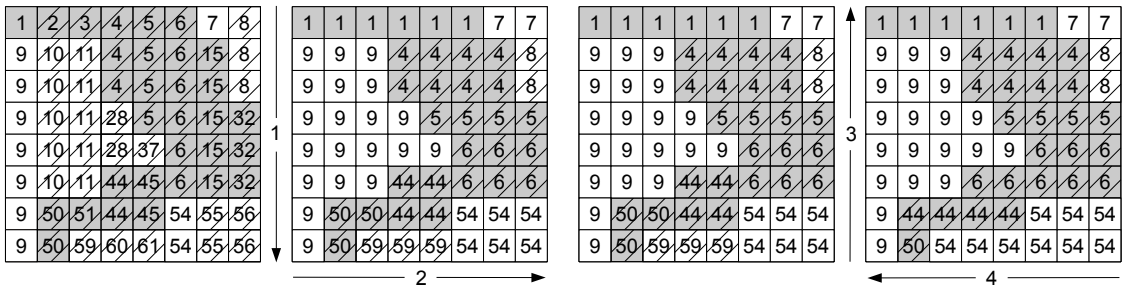


Figure 6: One iteration of the Directional Propagation Labelling algorithm, one pass for each direction in each dimension. Crossed out cells currently have an incorrect label.

Algorithm 8 The Kernel to iterate through the I^{th} 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 labeled.

function Mesh_Kernel_C(I, D_d, L_d, m_d)
  declare integer $id, label_1, label_2$ in local memory
  $id \leftarrow$ threadID & blockID from CUDA runtime
  $label_1 \leftarrow L_d[id, 1]$ //$[id, 1]$ indexes an array with the I^{th} index set to 1 and the other indexes defined by id.
  for $n \leftarrow 2$ **to** N **do**
    $label_2 \leftarrow L_d[id, n]$
    if $D_d[id, n] = D_d[id, n-1]$ **then**
      //$D_d[] = D_d[]$ tests if two cells have the same state
      if $label_1 < label_2$ **then**
        $L_d[id, n] \leftarrow label_1$
        $m_d \leftarrow true$
      else
        $label_1 \leftarrow label_2$
      end if
    else
      $label_1 \leftarrow label_2$
    end if
  end for
  return

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. The performance of this is dependent in the order that the threads are executed.

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 9.

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).

4.3. 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

Algorithm 9 The algorithms for the three phases of Kernel D: Scanning, Analysis and Labeling. The host will continue to call these three functions in order until the Scanning function no longer sets the value m_d (the global boolean) to true. At this point the mesh will have been correctly labeled.

function Mesh_Kernel_D_Scanning_Phase(D_d, L_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 $id_n \in n_{id}$ **do**
    min($label_2, L_d[id_n]$)
  end for
  if $label_2 < label_1$ **then**
    atomicMin(g_ref[id], $label_2$)
    $m_d \leftarrow true$
  end if
  return

function Mesh_Kernel_D_Analysis_Phase(D_d, L_d, R_d)
  declare integer $id, label$ in local memory
  $id \leftarrow$ threadID & blockID from CUDA runtime
  $label \leftarrow L_d[id]$
  if $label = id$ **then**
    declare integer ref in local memory
    $ref \leftarrow label$
    $label \leftarrow R_d[ref]$
    repeat
      $ref \leftarrow label$
      $label \leftarrow R_d[ref]$
    until $ref = label$
    $R_d[id] \leftarrow label$
  end if
  return

function Mesh_Kernel_D_Labeling_Phase(D_d, L_d, R_d)
  declare integer id
  declare integer ref in local memory
  $id \leftarrow$ threadID & blockID from CUDA runtime
  $ref \leftarrow R_d[L_d[id]]$
  $L_d[id] = R_d[ref]$
  return

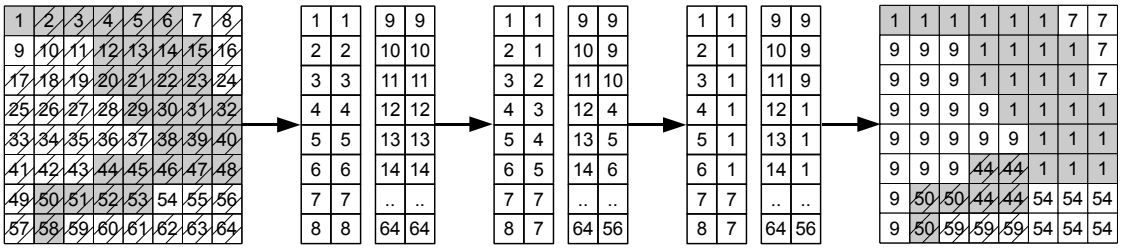


Figure 7: A single iteration of the Label Equivalence method. From left to right: the initial scan and initial equivalence list, the list after the scan, the list after analysis and the mesh after labelling from the equivalence list. Crossed out lines indicate cells that have do not yet have their final label.

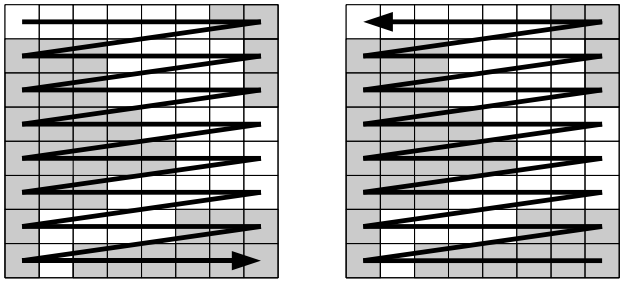


Figure 8: The alternating raster scan patterns used by the CPU I method.

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 Figure 8. 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 & 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 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 [30]. 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.

5. Performance Results

We now analyse the performance of the CUDA kernels and CPU reference implementation and compare them with each other. Section 5.1 describes the test graphs used in these simulations and Section 5.2 provides and discusses the performance outcomes of the individual simulations.

5.1. 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 2 gives an overview of the graph properties.

Disjoint This algorithm generates graphs with a specified number of major components. Vertices are assigned to the sets at random. Edges are created between randomly selected vertices from the same set. 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.

Table 2: 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 abbreviations k and m stand for "thousands" and "millions" respectively. Values are rounded to one significant digit if necessary.

	Vertices	Edges	Avg. degree	Components
Disjoint 1	$2 - 16$m	50m	$6.3 - 50$	$100 - 31044$
Disjoint 2	1m	$10 - 60$m	$20 - 120$	100
Disjoint 3	5m	60m	24	$1 - 1000019$
Line	$200 - 1000$k	199999-999999	2	1
Scale-free	$1 - 5$m	4999985-24999985	10	1
W-S, $p = 0.01$	$2 - 14$m	10-70m	10	1
W-S, $p = 0.1$	$2 - 14$m	10-70m	10	1
W-S, $p = 1.0$	$2 - 14$m	10-70m	10	1

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 3.3, 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 $P(k) \sim k^{-\gamma}$ [31]. Typically, the exponent γ lies between 2 and 3 [32, 26]. A graph generated with this algorithm 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.

Watts-Strogatz The Watts-Strogatz (W-S) small-world network model [33] 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 rewired with probability p, considering every edge exactly once. The rewired edges are shortcuts or "long-distance" 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.

5.2. 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 Figures 9 to 15 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 turns out to run 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

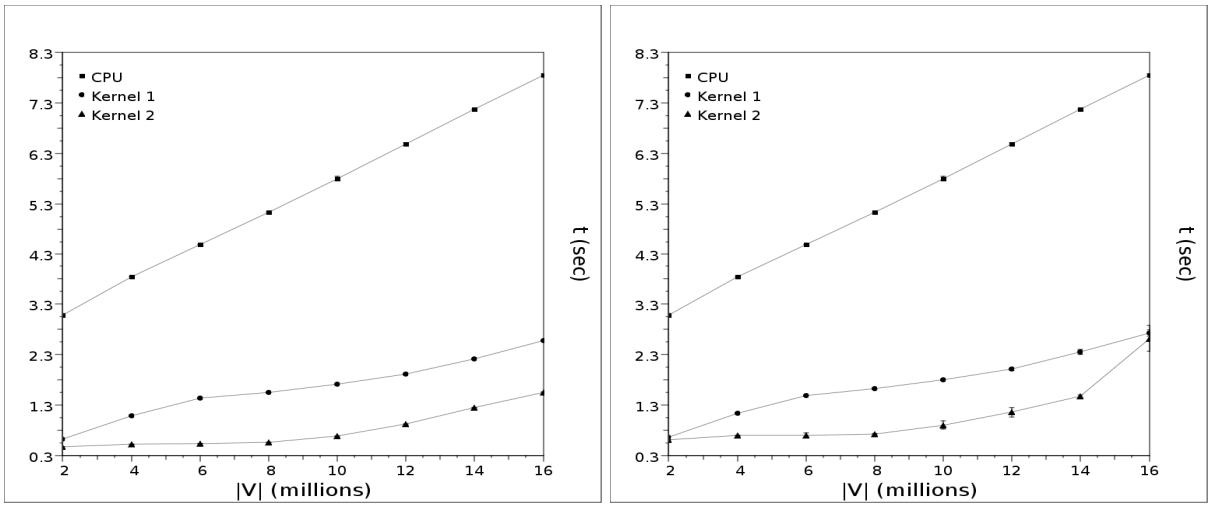


Figure 9: The execution times t measured to process the **Disjoint 1** graphs excluding (left) and including (right) memory allocation, deallocation and host $\leftrightarrow$ device memory copies. The number of vertices $|V|$ increases from $2,000,000 - 16,000,000$. The number of edges and major components is set to $50,000,000$ and 100 respectively. The execution times for Kernel 3 have been left out of the plots as they are much higher than the other results, ranging from $\approx 20 - 229$ seconds.

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, 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.

The system used for the performance measurements has the following configuration: An Intel® Core™2 Quad CPU running at 2.33GHz with an NVIDIA® GeForce® GTX260 graphics card and 4GB of DDR2-800 system memory ($\approx$ 2.8GB accessible due to the use of a 32-bit operating system). The linux distribution Kubuntu 8.10 is used as the operating system.

Figure 9 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 $\leftrightarrow$ device memory copies is included. Only at the last data point, where $|V| = 16,000,000$ (total device memory usage $443MB$ for Kernel 1 and $534MB$ 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 in Figure 10. 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 figure 11, 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

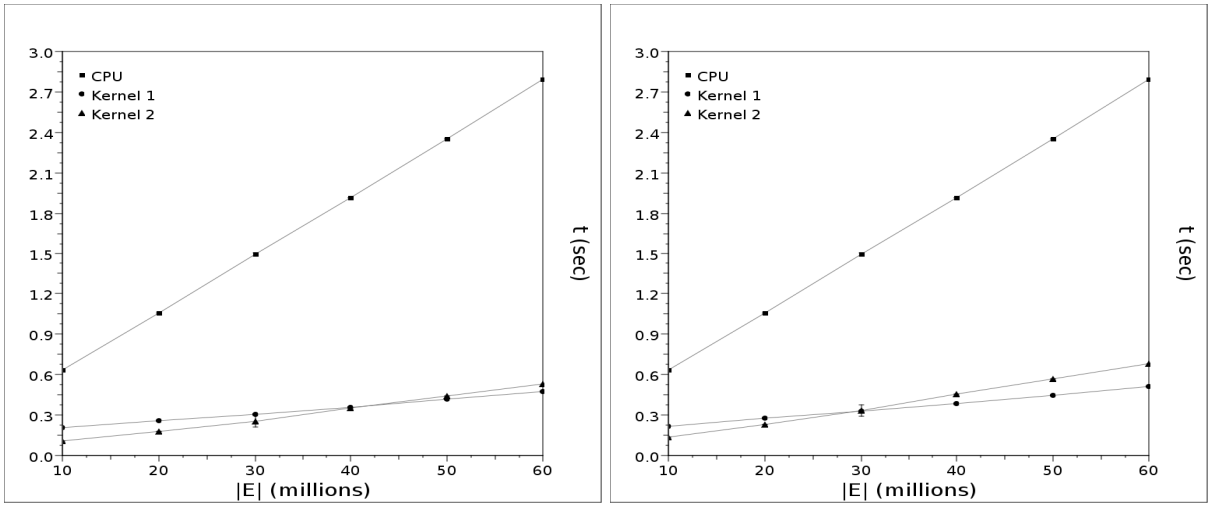


Figure 10: The execution times t measured to process the **Disjoint 2** graphs excluding (left) and including (right) memory allocation, deallocation and host $\leftrightarrow$ device memory copies. The number of edges $|E|$ increases from $10,000,000 - 60,000,000$. The number of vertices and major components is set to $1,000,000$ and 100 respectively. The execution times for Kernel 3 have been left out of the plots as they are much higher than the other results, ranging from $\approx 9.8 - 11.2$ seconds.

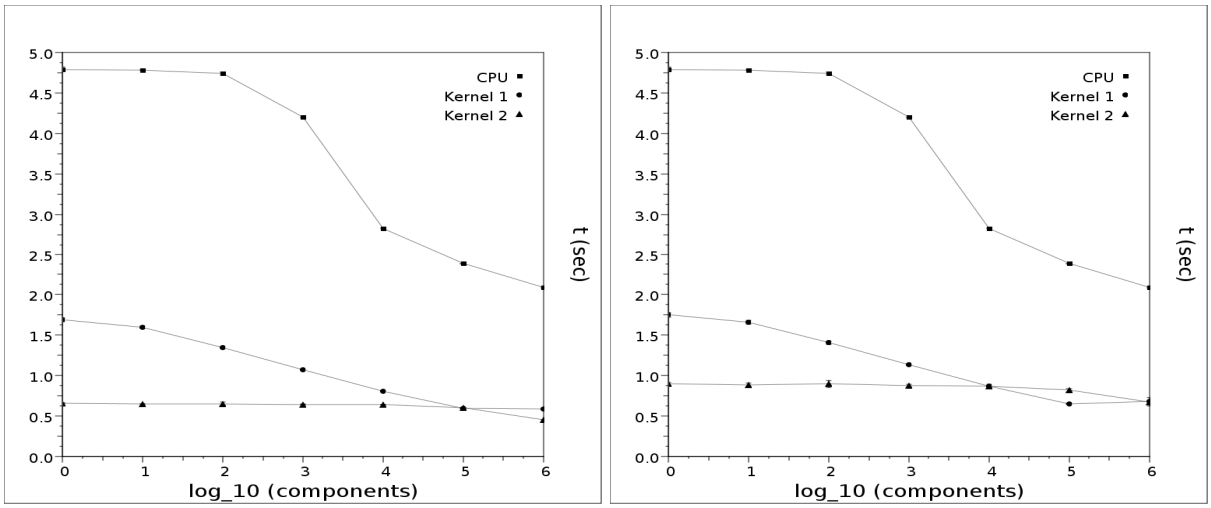


Figure 11: The execution times t measured to process the **Disjoint 3** graphs excluding (left) and including (right) memory allocation, deallocation and host $\leftrightarrow$ device memory copies. The number of major components increases from $10^0, 10^1, \ldots, 10^6$. The number of vertices and edges is set to $5,000,000$ and $60,000,000$ respectively. The execution times for Kernel 3 have been left out of the plots as they are much higher than the other results, ranging from $\approx 47 - 1515$ seconds.

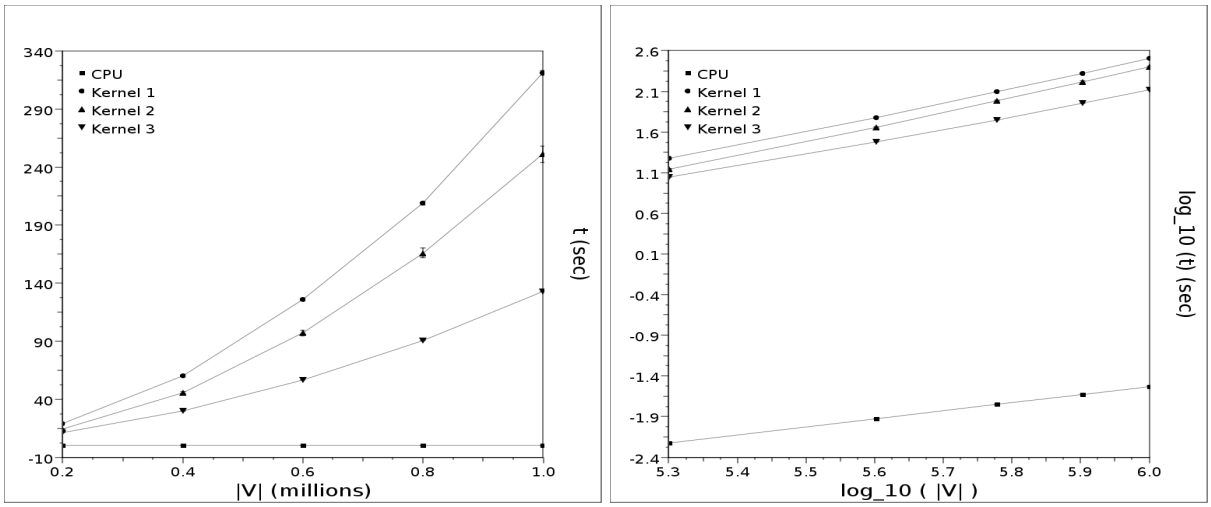


Figure 12: The execution times t measured to process the **Line** graphs (left). Because the CUDA kernel execution times are so high and the graphs are rather small (compared to the other graphs used for the performance measurements), the device memory operations do not change the results in any significant 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 $\ll 1$ second for all measured graphs of this type, beating the CUDA kernels by far.

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 execution times for Kernel 3, on the other hand, increase quickly with the number of components. The reason is that a new breadth-first search has to be started for each disjoint component. And small components keep the frontier of the search small, which means that most of the processing units are not being utilised well.

The results for the **Line** graphs are shown in Figure 12. 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. Even Kernel 3 performs better than Kernels 1 and 2. However, this does still not make it look good compared to the CPU algorithm.

Figure 13 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.

Figure 14 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

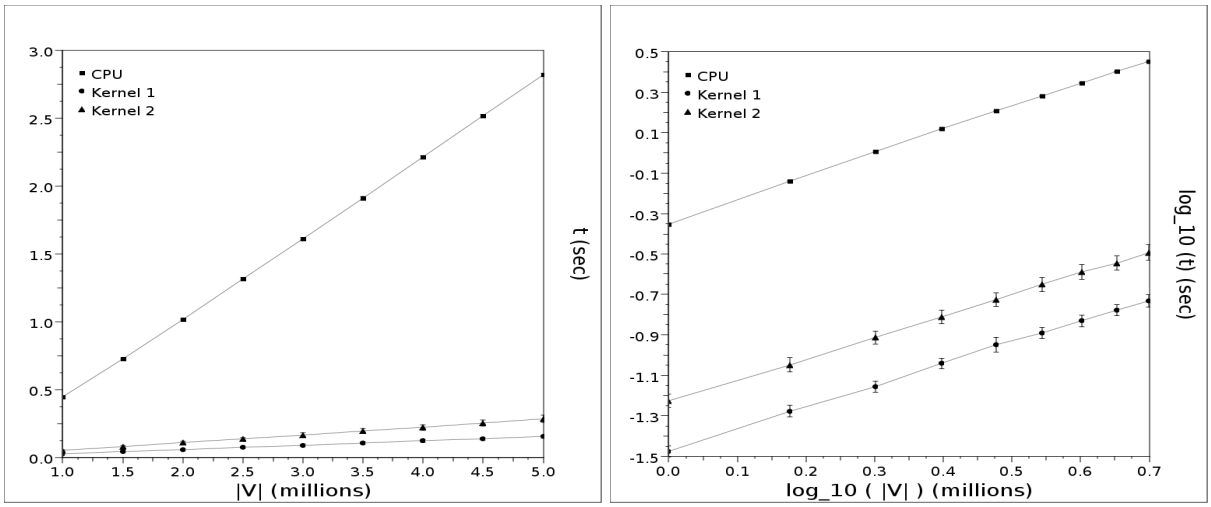


Figure 13: The execution times t measured to process the **Scale-free** graphs excluding (left) and including (right) memory allocation, deallocation and host $\leftrightarrow$ device memory copies. The plot on the right is given on a logarithmic scale, illustrating that the results scale logarithmically 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 have been left out of the plots as they are much higher than the other results, ranging from $\approx 9 - 47$ seconds.

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 $503MB$ for Kernel 1 and $572MB$ 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.

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. Figure 15 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 $\sim 500MB$ 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.

5.3. Hypercubic Performance Results

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

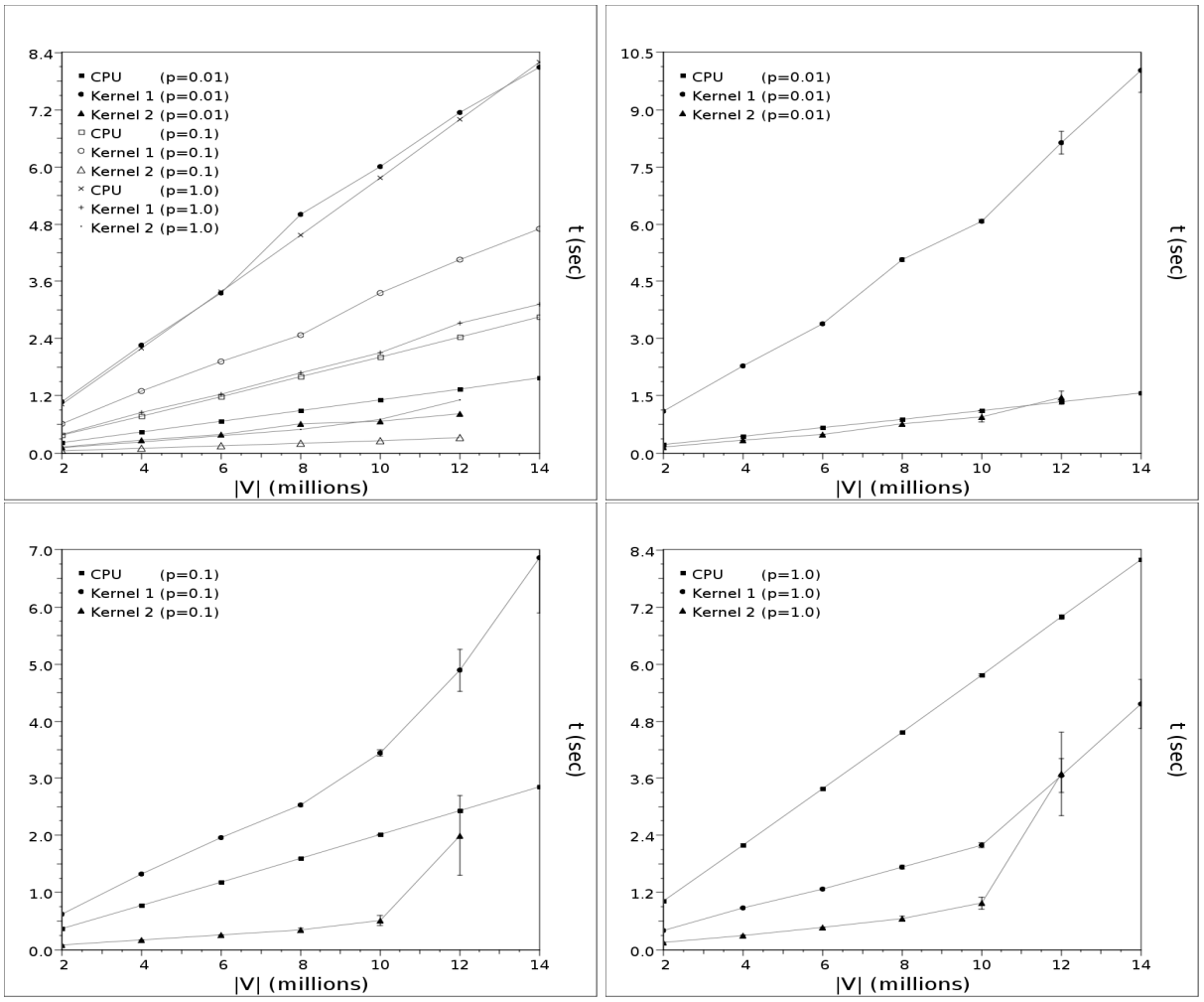


Figure 14: The execution times t measured to process the **W-S** small-world graphs with the number of vertices $|V|$ increasing from $2,000,000 - 14,000,000$. The average degree for all graph instances is $k = 10$. The plot on the top left shows the results for all three rewiring probabilities $p = \{0.01, 0.1, 1.0\}$ excluding memory allocation, deallocation and host $\leftrightarrow$ 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 $\leftrightarrow$ 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 on page 7. The execution times for Kernel 3 have been left out of the plots as they are much higher than the other results, ranging from $\approx 19 - 132$ seconds for all p.

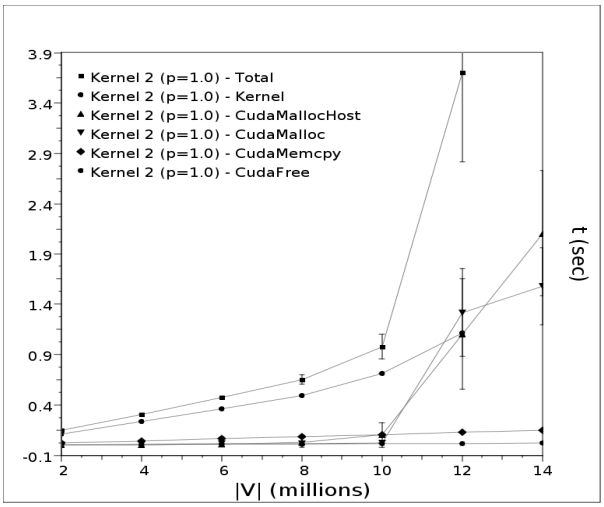


Figure 15: The dissected execution times for Kernel 2 on the Watts-Strogatz graphs with $p = 1.0$. The datasets illustrate how long the individual CUDA operations (memory allocations, host to device memory copies, kernel executions and memory deallocations) take.

which quickly overran the memory limitations of the GPU. 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 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.

5.3.1. 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 simulation used is the GPU-based simulation described in [28]. 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 Figure 16.

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.

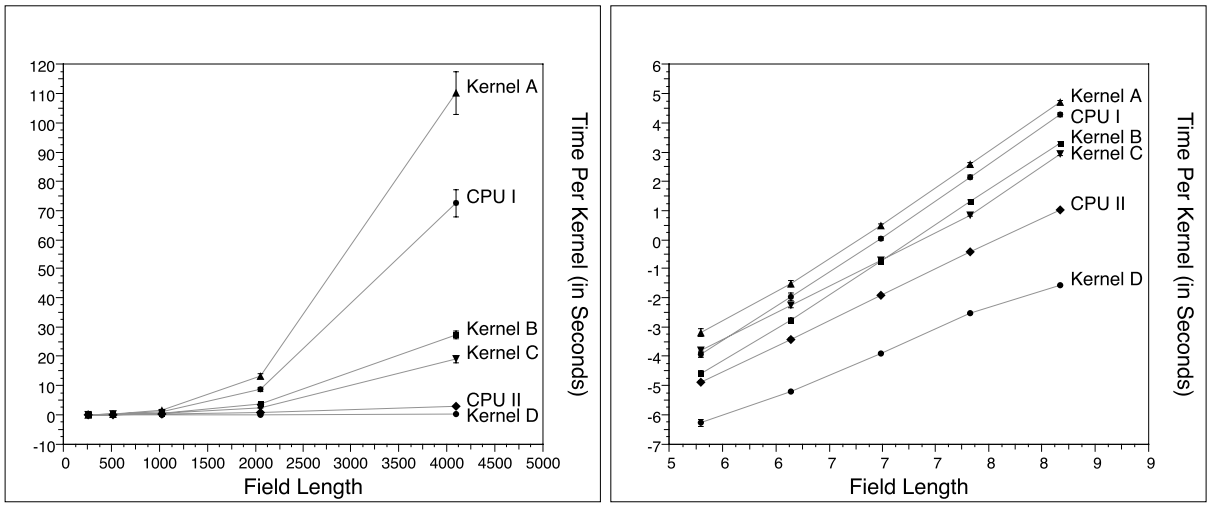
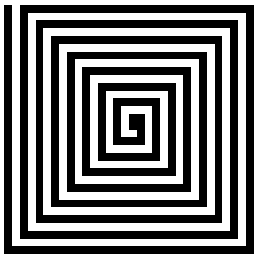


Figure 16: Left: a comparision of the performance of the GPU and CPU CCL implementations labelling Cahn-Hilliard systems for field lengths N = 256, 512, 1024, 2048, 4096. Right: the same data presented in ln-ln scale.

5.3.2. 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 NxN 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 Figure 17.



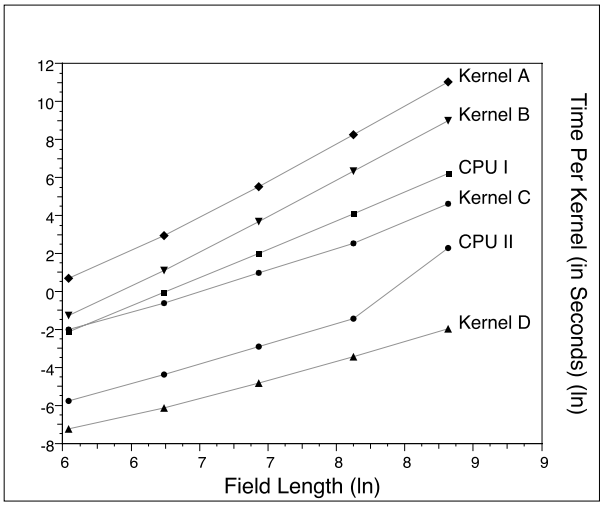


Figure 17: Left: a sample Spiral image used for testing (32x32). Right: the performance results (in ln-ln scale) of the CPU and GPU CCL implementations labelling spiral images of size N = 256, 512, 1024, 2048, 4096.

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 4096x4096 spiral was over 450,000x 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 4096x4096 Cahn-Hilliard system was over 600x less than for a spiral of the same size.

6. 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 500MB 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 [29]. 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 labeled.

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 [34].

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 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] V. C. Barbosa, R. G. Ferreira, On the phase transitions of graph coloring and independent sets, Physica A 343 (2004) 401–423.

[2] 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: 14th International Conference, Proceedings, Vol. 4873, Springer-Verlag, Goa, India, 2007, pp. 197–208.

[3] A. H. Gebremedhin, F. Manne, Scalable Parallel Graph Coloring Algorithms, Concurrency: Practice and Experience 12 (2000) 1131–1146.

[4] 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.

[5] 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.

[6] C. F. Baillie, P. D. Coddington, Comparison of cluster algorithms for two-dimensional potts models, Phys. Rev. B 43 (1991) 10617–10621.

[7] R. Dewar, C. K. Harris, Parallel computation of cluster properties: application to 2d percolation, J. Phys. A Math. Gen. 20 (1987) 985–993.

[8] 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.

[9] P. M. Flanders, Effective use of SIMD processor arrays, in: Parallel Digital Processors, IEE, Prtugal, 1988, pp. 143–147, iEE Conf. Pub. No. 298.

[10] W. D. Hillis, The Connection Machine, MIT Press, 1985.

[11] R. Gould, Graph Theory, The Benjamin/Cummings Publishing Company, 1988.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction To Algorithms, 2nd Edition, MIT Press, 2001.

[13] D. Brelaz, New methods to color the vertices of a graph, Communications of the ACM 22 (4) (1979) 251–256.

[14] J. A. Gallian, A dynamic survey of graph labeling, The Electronic Journal of Combinatoris 16 (2005) DS6.

[15] W. Press, B. Flannery, S. Teukolsky, W. Vetterling, Numerical Recipes in C, Cambridge University Press, 1988, Ch. 12, pp. 252–254, determination of Equivalence Classes.

[16] K. Hawick, D. Playne, Hypercubic storage in arbitrary dimensions, Tech. rep., Massey University (2009).

[17] P. Erdös, A. Rényi, On random graphs, Publicationes Mathematicae 6 (1959) 290–297.

[18] H. Jeong, B. Tombor, R. Albert, Z. Oltvai, A.-L. Barabsi, The large-scale organization of metabolic networks, Nature 407 (6804) (2000) 651–654.

[19] D. A. Fell, A. Wagner, The small world of metabolism, Nature Biotechnology 18 (11) (2000) 1121–1122.

[20] A. Wagner, D. A. Fell, The small world inside large metabolic networks, Proceedings of the Royal Society B 268 (1478) (2001) 1803–1810.

[21] M. E. J. Newman, The structure of scientific collaboration networks, PNAS 98 (2) (2001) 404–409.

[22] M. E. J. Newman, Ego-centered networks and the ripple effect, Social Networks 25 (1) (2003) 83–95.

[23] S. Milgram, The Small-World Problem, Psychology Today 1 (1967) 61–67.

[24] F. Liljeros, C. Edling, L. Amaral, H. Stanley, Y. Aberg, The web of human sexual contacts, Nature 411 (6840) (2001) 907–908.

[25] D. Liben-Nowell, J. Kleinberg, Tracing information flow on a global scale using Internet chain-letter data, PNAS 105 (12) (2008) 4633–4638.

[26] R. Albert, H. Jeong, A.-L. Barabsi, Diameter of the World-Wide Web, Nature 401 (6749) (1999) 130–131.

[27] NVIDIA® Corporation, CUDA^{TM} 2.1 Programming Guide, last accessed May 2009 (2009).
URL `http://www.nvidia.com/`

[28] A. Leist, D. Playne, K. Hawick, Exploiting Graphical Processing Units for Data-Parallel Scientific Applications, Tech. Rep. CSTN-065, Massey University, to appear in Concurrency and Computation: Practice & Experience (October 2008).

[29] K. Wu, E. Otoo, K. Suzuki, Optimizing two-pass connected-component labeling algorithms, Pattern Anal. Applic. 12 (2009) 117–135. doi:DOI 10.1007/s10044-008-0109-y.

[30] 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: Proc. 15th International Conference on Pattern Recognition (ICPR'00), Vol. 2, 2000, pp. 434–437.

[31] A.-L. Barabsi, R. Albert, Emergence of scaling in random networks, Science 286 (5439) (1999) 509–512.

[32] D. J. d. Price, Networks of Scientific Papers, Science 149 (3683) (1965) 510–515. doi:10.1126/science.149.3683.

[33] D. J. Watts, S. H. Strogatz, Collective dynamics of 'small-world' networks, Nature 393 (6684) (1998) 440–442.

[34] 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).