J. Parallel Distrib. Comput. 71 (2011) 615–620

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

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

Research note

Connected component labeling on a 2D grid using CUDA

Oleksandr Kalentev [a,*], Abha Rai [a], Stefan Kemnitz [b], Ralf Schneider [c]

[a] Max-Planck-Institut für Plasmaphysik, Wendelsteinstr. 1, D-17491 Greifswald, Germany

[b] Fachhochschule Stralsund – University of Applied Sciences, Zur Schwedenschanze 15, D-18435 Stralsund, Germany

[c] Ernst-Moritz-Arndt-Universität, Domstr. 11, D-17487 Greifswald, Germany

ARTICLE INFO

Article history:
Received 28 May 2010
Received in revised form
13 October 2010
Accepted 14 October 2010
Available online 26 October 2010

Keywords:
CUDA
GPU
Parallel
Connected component
Component labeling
Mesh

ABSTRACT

Connected component labeling is an important but computationally expensive operation required in many fields of research. The goal in the present work is to label connected components on a 2D binary map. Two different iterative algorithms for doing this task are presented. The first algorithm (Row–Col Unify) is based upon the directional propagation labeling, whereas the second algorithm uses the Label Equivalence technique. The Row–Col Unify algorithm uses a local array of references and the reduction technique intrinsically. The usage of shared memory extensively makes the code efficient. The Label Equivalence algorithm is an extended version of the one presented by Hawick et al. (2010) [3]. At the end the comparison depending on the performances of both of the algorithms is presented.

1. Introduction

Connected component labeling is an important problem appearing in different fields of research. Although one can consider this problem as a general one, namely arbitrary graph component labeling or coloring, often the specific task of labeling connected components on a grid is of great interest. This is needed for example in computer vision as a part of segmentation, namely separating the objects from the background. It reduces the image to a binary representation, in which objects are represented by 1's and the background by 0's. Another field of application are cellular automata (CA) models used for different kinds of simulation in physics, mathematics and biology.

Since the early 1970s, numerous approaches for connected component labeling have been introduced [4,8–10]. Most of these approaches are suitable for sequential processing, but also some parallel algorithms have been developed [5,1].

The use of GPUs with interfaces as CUDA [6] or OpenCL [7] opens a new perspective for many data processing approaches. The problem of graph component labeling with GPUs has already been addressed by Hawick et al. [3].

In the current work, we present two algorithms for connected component labeling on a 2D binary grid. For our implementations, we use CUDA and measure the performances of the two algorithms on an NVIDIA TESLA C1060. The first algorithm for directional propagation labeling uses the reduction technique. For the second algorithm, we improve and extend the implementation by Hawick et al. [3], which allows one to reduce the total memory usage and simplifies the original procedure. We demonstrate that our implementation can be easily extended to compute different connected component characteristics such as area or perimeter. This is of interest, e.g. in cases where binding energies of cluster atoms change with cluster sizes.

The paper is organized as follows. In Section 2, the two algorithms are described. Section 3 presents the comparison of the performances. In Section 4, we conclude the paper.

2. Algorithm description

Connected component labeling in our framework is the assignment of a unique label to each non-zero element on a 2D grid in such a way that all non-zero neighbors get the same label. In this work we consider 2D cases with four neighbors (north, south, east and west).

2.1. Row–Col unification

The first algorithm is similar to a "kernel C" algorithm described in [3]. This method implements the directional propagation labeling. In the initial "kernel C" approach, each thread is responsible for the whole row or column.

* Corresponding author.
E-mail addresses: okalenty@ipp.mpg.de, o.kalentev@googlemail.com (O. Kalentev).

0743-7315/$ – see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2010.10.012

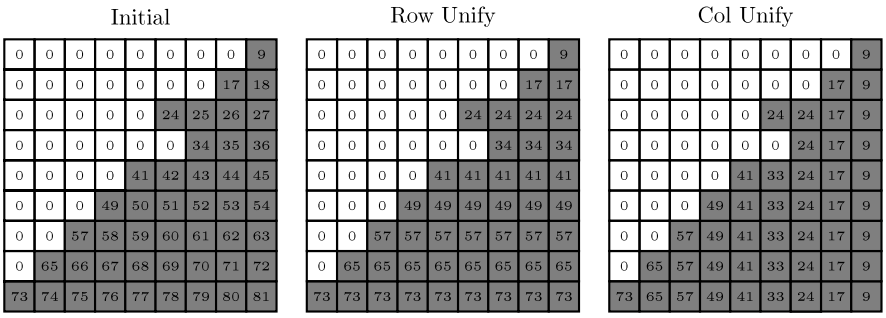

Fig. 1. Row–Col Unify procedure.

In our implementation, we intrinsically apply the reduction technique, which is well known in parallel computing [2]. This technique allows one to obtain integrated characteristics of a bunch of data, for example, the sum of array elements. Additionally, in our implementation we use shared memory and a local array of references. In the following section, we describe the main steps of the algorithm.

The main idea of the algorithm is the propagation of the label with the smallest value. The procedure has to initialize all non-zero elements with unique identifiers which are equal to the value of the corresponding index of the element in the label array. Thereafter, the smallest label value propagates along all rows and then columns (see Fig. 1). Making several iterations of such a label propagation procedure, the algorithm finally marks all elements in each connected component with the smallest label value of this connected domain.

Algorithm 1 Row–Col Unify algorithm. Host part.

Require: *mapLabelsHost*, *mapLabelsDev*
Require: *ContinueIndHost*, *ContinueIndDev*
  mapLabelsDev ← *mapLabelsHost*
  call: *InitializeLabels*
  while *ContinueIndHost* ≠ 0 **do**
    ContinueIndHost ← 0
    ContinueIndDev ← *ContinueIndHost*
    call: *UnifyRow*(*mapLabelsDev*, *ContinueIndDev*)
    call: *UnifyCol*(*mapLabelsDev*, *ContinueIndDev*)
    ContinueIndHost ← *ContinueIndDev*
  end while
  mapLabelsHost ← *mapLabelsDev*

The current implementation of the "Row–Col Unify" algorithm is suitable only for a map with size 1024 × 1024. The adaptation for other sizes was not done due to the fact that this algorithm is much slower than the second one (see Section 3). The algorithm is iterative and consists of two steps. The host part is described in pseudo-code in Algorithm 1. First, the label array (*mapLabels Dev*) is initialized with unique identifiers. Then, two unification procedures for rows (*UnifyRow*) and columns (*UnifyCol*), respectively, are called within the *while* loop. Each procedure requires two parameters: the first one is the label array mentioned above; the second one is an integer indicator for the loop termination. If *UnifyRow* or *UnifyCol* do not set this indicator to 1, the *while* loop is terminated. This is possible due to the fact that the operation of value assignment does not produce any synchronization problems for CUDA devices in this case.

The main routine kernel is described in Algorithm 2. It is identical for both row and column unification. The procedures for row and column processing differ from each other only in the way they access the global memory. The grid for this kernel is constructed in the following way: threads from the same block access the same row or column. The layout of the global memory is organized in such a way that the access for rows is coalesced whereas for columns it is not. In the routine, the label array is copied first from the global to the shared memory, where also the local array of references is allocated. We add one more element to the front of the label and reference arrays to avoid unnecessary conditioning during the update procedure.

Algorithm 2 Row–Col Unify algorithm. Main routine kernel.

Require: *mapLabelsDev*, *shmemLabels*, *shmemRefs*
Require: *ContinueInd*
  shmemLabels ← *mapLabelsDev*
  call: *UnifyRowI*((*threadId* ∗ 2 + 1), *ContinueInd*, *shmemLabels*, *shmemRefs*)
  if *threadId* < 256 **then**
    call: *UnifyRowL*((*threadId* ∗ 2 + 1) ∗ 2, *shmemLabels*, *shmemRefs*)
  end if
  call: *syncthread*()
  call: *Update*(*threadId* ∗ 2 + 1, *shmemLabels*, *shmemRefs*)
  call: *syncthread*()
  if *threadId* < 128 **then**
    call: *UnifyRowL*((*threadId* ∗ 2 + 1) ∗ 4, *shmemLabels*, *shmemRefs*)
  end if
  call: *syncthread*()
  call: *Update*(*threadId* ∗ 2 + 1, *shmemLabels*, *shmemRefs*)
  call: *syncthread*()
  . . .
  . . .
  . . .
  if *threadId* < 1 **then**
    call: *UnifyRowL*((*threadId* ∗ 2 + 1) ∗ 512, *shmemLabels*, *shmemRefs*)
  end if
  call: *syncthread*()
  call: *Update*(*threadId* ∗ 2 + 1, *shmemLabels*, *shmemRefs*)
  call: *syncthread*()
  mapLabelsDev ← *shmemLabels*

Secondly, we apply an initial unification procedure which is the first step of reduction. This is described in Algorithm 3. Each two neighboring elements are processed in each thread. If both elements in the label array are non-zero, then the one that has the smallest value is propagated, i. e., this value is assigned to both array elements. The references of these elements are initialized according to the indices of the elements: the one with the larger index value references the one with the smaller index value which has a reference to itself. This is shown in Fig. 2.

The *ContinueInd* indicator is set to 1, if two neighboring non-zero elements that are not equal to each other are found.

Then, a cascade of reduction and update steps is executed.

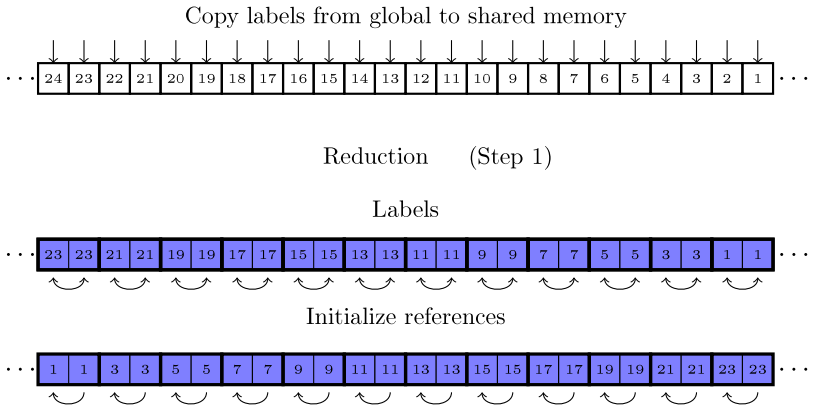

Fig. 2. Row–Col Unify algorithm. Initialization and first reduction step.

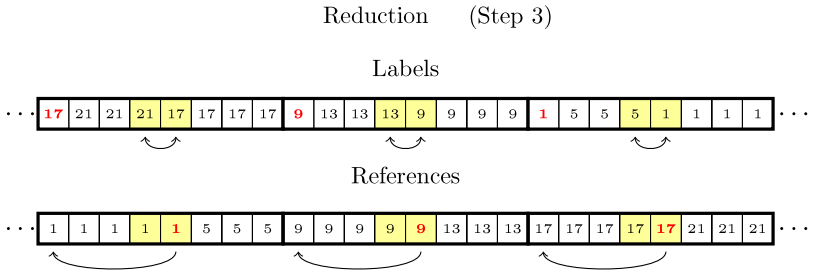

Fig. 3. Row–Col Unify algorithm. Reduction and update step 3. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Algorithm 3 Row–Col Unify algorithm. References initialization, 1st step of unification.

function *UnifyRowI*(*iPos*, *ContinueInd*, *shmemLabels*, *shmemRefs*)
  if *shmemLabels*[*iPos*] $\neq$ 0&&*shmemLabels*[*iPos* + 1] $\neq$ 0 **then**
    if *shmemLabels*[*iPos*] $\neq$ *shmemLabels*[*iPos* + 1] **then**
      shmemLabels[*iPos*] $\leftarrow$ *min*{*shmemLabels*[*i*] | *i* = *iPos*, *iPos*+ 1}
      shmemLabels[*iPos* + 1] $\leftarrow$ *shmemLabels*[*iPos*]
      ContinueInd $\leftarrow$ 1
    end if
  end if
  shmemRefs[*iPos*] $\leftarrow$ *iPos*
  shmemRefs[*iPos* + 1] $\leftarrow$ *iPos*
  return

Each subsequent step in the cascade requires half the numbers of active threads for the *UnifyRowL* reduction procedure compared to the previous step. However, it uses all threads in a block for an *Update* procedure. In Fig. 3, an example of one (third) step of reduction is shown. As mentioned before, in the first reduction step each thread processes two neighboring elements. In the subsequent steps, the number of neighboring elements processed by each thread doubles, i.e., in step 2 the number of elements is 4, in step 3 the number of elements becomes 8 and so on.

In Fig. 3, three groups of labels and references which are processed by three threads are shown. The separation of groups is indicated by thicker lines. Each thread checks the values of the two central elements of the corresponding groups (yellow). If they are non-zero, the minimum value of these two elements is assigned to the element which is referenced by the element with the smaller index value from the processed pair. The references are taken from the reference array. The references of the processed pair are also updated, i.e., the element with the larger index value receives the same reference as the element with the smaller index value. The updated values in both arrays are marked in red. A detailed description of the *UnifyRowL* procedure is given in Algorithm 4.

Algorithm 4 Row–Col Unify algorithm. *UnifyRowL* procedure.

function *UnifyRowL*(*iPos*, *shmemLabels*, *shmemRefs*)
Require: *iRef*
  iRef $\leftarrow$ *shmemRefs*[*iPos*]
  if *shmemLabels*[*iPos*] $\neq$ 0&&*shmemLabels*[*iPos* + 1] $\neq$ 0 **then**
    shmemLabels[*iRef*] $\leftarrow$ *min*{*shmemLabels*[*i*] | *i* = *iPos*, *iPos* + 1}
    shmemRefs[*iPos* + 1] $\leftarrow$ *shmemRefs*[*iPos*]
    shmemLabels[*iPos* + 1] $\leftarrow$ *shmemLabels*[*iRef*]
  end if
  return

After the procedure *UnifyRowL* is finished, all threads in a block must be synchronized and the *Update* function is called in order to complete the reduction step. Here, all threads update both labels and references for two processed neighboring elements, i.e., each element obtains the corresponding label and reference from the element to which it currently points. Algorithm 5 summarizes the *Update* procedure.

Algorithm 5 Row–Col Unify algorithm. *Update* procedure.

function *Update*(*iPos*, *shmemLabels*, *shmemRefs*)
Require: *iRef*
  iRef $\leftarrow$ *shmemRefs*[*iPos*]
  shmemLabels[*iPos*] $\leftarrow$ *shmemLabels*[*iRef*]
  shmemRefs[*iPos*] $\leftarrow$ *shmemRefs*[*iRef*]
  iRef $\leftarrow$ *shmemRefs*[*iPos* + 1]
  shmemLabels[*iPos* + 1] $\leftarrow$ *shmemLabels*[*iRef*]
  shmemRefs[*iPos* + 1] $\leftarrow$ *shmemRefs*[*iRef*]
  return

The weak point of this procedure is the bank conflicts which lead to the serialized requests for the reading of the value needed for the update. This gives rise of the performance drop for the whole procedure.

Table 1
The benchmark of the Row–Col Unify and the Label Equivalence algorithms.

		Time (ms)						
		Blobs				Spiral	Random	
		C1	C2	C3	C4	–	0.5	0.1
RC		87.16	87.89	87.27	87.65	4501.56	254.47	34.75
LE	NSZ	5.60	5.77	5.89	6.68	5.60	6.47	1.56
	SZ	16.25	16.28	15.22	17.30	38.86	14.23	2.11

2.2. Hoshen–Kopelman or Label Equivalence

The second presented algorithm is similar to the well-known Hoshen–Kopelman [4] algorithm. Recent publications by Suzuki et al. [10] and Wu et al. [11] give a nice description of the label equivalence procedure for the labeling of connected components in a binary image in the sequential case.

The parallel version of the Label Equivalence algorithm for GPUs has been presented by Hawick et al. [3].

According to their description, the multi-pass algorithm consists of three phases which are repeated in a loop: scanning, analysis, and labeling. The first phase constructs a forest of references, the second one connects all references in each tree to the root, and the third one assigns the corresponding labels according to the references.

Our implementation is significantly improved compared to the algorithm presented by Hawick et al. [3] in terms of memory consumption: there is no need for an additional reference array. Apart from that, it requires less steps: the labeling phase is omitted. Other features are the usage of padding which allows us to avoid extra conditioning for the border elements. Atomic operations which slow down the computation dramatically due to their synchronous nature are also omitted in the scanning phase. This is possible due to the iterative nature of the algorithm: if collision happens, it will be resolved during the next iterative step. Fig. 4 demonstrates the first step of the algorithm as well as usage of the padding. Here, the left picture shows initial labels with the value equal to the array index, starting from the left top added element. The right one shows labels after the first step of the algorithm. Considering labels as references one can find the forest consists of 8 trees (for convenience they are depicted with different colors). After the second step values of all the labels in those trees will be the same and equal to the corresponding root's label value, *i.e.*, 20 for the gray region, 31 for the orange, 41 for the blue etc.

Listing 1 shows the initialization procedure. 'UINT' stands for the 'unsigned int' type. 'SIZE*' and 'SIZE*PAD' are definitions of macros for the size of the working area and the whole area including padding, respectively. The label array must be filled with a binary image. The procedure initializes all non-zero elements with unique identifiers which are equal to the value of the corresponding index of the element in the label array. This allows us to use these values later as references.

After initialization of the label array, the main loop for the iterative labeling of connected components starts. Two functions are called on each step: "Scanning" and "Analysis". The first function represents the first phase of the algorithm, namely the linking of the elements. This is done by choosing the smallest nonzero label value within five elements: one central and its neighbors (see Listing 3). As labels with zero value are ignored during the procedure, constructed trees never connect to the background and, therefore also with each other. As a consequence of the locality of the procedure, the algorithm works for any number of disconnected components.

The indicator *IsNotDone* is only set to 1, if the label value is changed. This is done in order to stop execution of the algorithm when no further iterations are needed.

The second function represents the relabeling phase of the algorithm. Here, in a *while* loop each thread takes a sequence

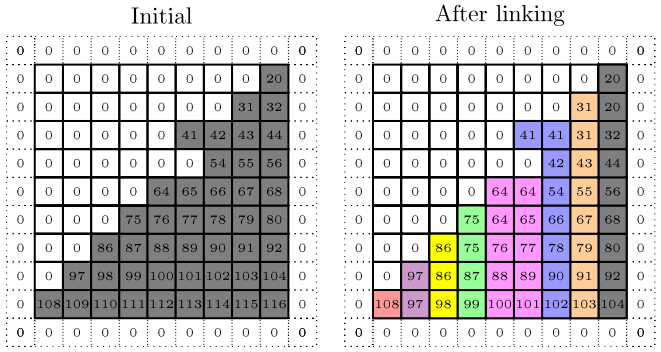

Fig. 4. Label Equivalence procedure. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

of labels starting from the current position considering them as references.

The loop is terminated when the label with the value that coincides with the index of that element is found (see Listing 2). This procedure works remarkably fast due to the fact that the steps which are performed previously make the later ones shorter.

Fig. 5 gives an example of such a situation. Here, the root element is shown in gray. All labels are linked to it if they are considered as references. For example, if for an element with index 6 (in blue) the relabeling was already done. Then, for an element with index 10 the sequence is shorter, because after processing element 6 the first element will be taken directly.

3. Benchmarks and discussion

To measure the performance of the two algorithms presented before several tests have been done. All tests presented in the following section are run on an NVIDIA TESLA C1060.

Table 1 summarizes the results for images with such a size and different topological and occupational settings of the image. Under topological settings we understand different shapes of the connected components, whereas occupational settings refer to the fraction of the connected components to the image size. Here, "RC" stands for the "Row–Col Unify" and "LE" for "Label Equivalence" algorithms. In Table 1, three different cases of the image filling are considered: spiral (see [3] for the details), random, and blob with random. The first one is traditionally considered to be a particularly complex case, whereas the second and the third ones are of great interest for tasks like cellular automata (CA).

For the CA procedure, one usually starts with a random distribution of the occupied cells and follows their dynamics which often results in some blobs of a certain size and a randomly distributed small noise. We compare two cases for the random cell distribution with occupation ratios of 0.5 and 0.1, which means that half and 10% of the image cells are occupied, respectively.

For the "blob with random" distribution we consider four cases with different blob sizes (C1-blob radius 10, C2-20, C3-50, C4-100). The occupation ratio is kept fixed at 0.5. Half of the occupied cells are assigned to blobs and the rest are randomly distributed. This choice of fixed occupation ratio and different blob sizes result in different numbers of blobs for different cases.

For the "Label Equivalence" algorithm, the performance of the extended version of the procedure (named in Table 1 "SZ") as well as that of the original one (named in Table 1 "NSZ") is measured.

To demonstrate the flexibility of our extension of the "Label Equivalence" algorithm in addition to the pure labeling the size of each connected component is also calculated, which is needed e.g. for calculating cluster-size dependent binding energies in CA applications. For that purpose another array *_CSize* is used. This array must be initialized with 0's in non-occupied cells and with

Table 2
Random distribution with different occupation ratio.

		Time (ms)							
		0.2	0.3	0.4	0.6	0.7	0.8	0.9	0.99
KB	NSZ	10.33	15.50	18.90	39.34	34.12	33.91	36.30	24.42
	SZ	16.74	27.81	38.84	84.21	95.70	113.60	137.95	317.06
Iter. num.		4	5	5	8	5	4	4	3

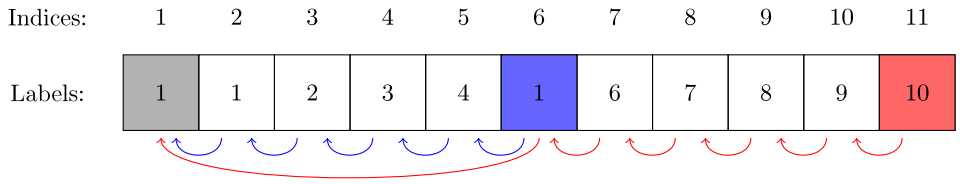

Fig. 5. Relabeling procedure. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Table 3
Scanning over picture size.

		Time (ms)			
		1024×1024	2048×2048	4096×4096	8128 × 8128
KB	NSZ	6.50	24.98	97.88	388.24
	SZ	14.30	55.75	217.89	870.70

Table 4
Scanning over the size of the block.

		Time (ms)					
		512	256	64	16	4	1
KB	NSZ	25.09	24.91	25.34	36.19	123.15	453.331
	SZ	55.41	55.73	55.13	62.13	154.91	522.085

1's in occupied cells. The only part which has to be modified is the "Analysis" function. Needed modifications are shown in Listing 4. The presented code has to be added after the assignment of the label at the end of the function. Here, after reconnecting the element with the root of the tree of references the size of the root is incremented by the size of the current element, which afterwards is set to 0. In order to synchronize the summation atomic operations are used. As a result of calculations one gets an array with connected component sizes stored in the element with the smallest index for each component. Other extensions to the algorithm can be done in a similar manner.

Table 1 shows that the "Label Equivalence" algorithm is faster in all cases. Blob sizes do not affect the speed of the second algorithm. The occupation ratio, on the other hand, influences the speed significantly.

The case with a spiral distribution is extremely difficult to handle for the first algorithm and very easy for the second one. It is interesting to mention that our modification of the "Label Equivalence" algorithm needs only 3 iterations for such a distribution regardless of the size of the map. This shows that the number of iterations needed for the second algorithm depends strongly on the topology of the connected component and only weakly on their size.

Table 2 shows the change of the performance of two modifications of the "Label Equivalence" algorithm with respect to different occupation ratio for the case of 2048 × 2048 grids. Here, the maximum number of iterations, and therefore calculation time, is needed for the case of an occupation ratio of 0.5–0.6. For other occupation ratio values the time consumption is less, although for the case of high occupation ratio it drops slower than for the lower ones. The reason for this is the first condition checking for the non-zero elements.

Table 3 shows the performance of the "Label Equivalence" algorithm with different sizes of the map. The execution time increases linearly with the total number of cells.

Table 4 shows the performance of the second algorithm with respect to the block size of the kernel. A significant drop of the performance occurs if the block size gets smaller than the warp size (32 in our case). The result is expected because the multiprocessor schedules and executes threads in groups of 32 parallel threads (see [6]).

4. Conclusions

Two type of iterative algorithms for labeling connected components on a 2D binary grid were described. The first one is a "Row–Col Unify" algorithm which implements the directional propagation labeling technique into CUDA. The second one is the modified version of "Label Equivalence" implemented by Hawick et al. in [3].

It was shown that there are two major advantages of the "Label Equivalence" algorithm over the "Row–Col Unify" one. The first advantage is the simpler implementation which leads to much less instructions. The second one is concerned with a reduced number of iterations needed for the procedure to expand the smallest label on a whole connected component. In the case of a spiral distribution of occupied cells for a 1024 × 1024 grid the number of iterations needed was 514 for the "Row–Col Unify" algorithm and 3 for "Label Equivalence". This demonstrates that the productivity of the second algorithm depends on the topology weaker than the productivity of the first one.

In general, the second algorithm is 15 ∼ 35 times faster compared with the first one. Another advantage of the "Label Equivalence" algorithm is its capability to be easily extended to calculate additional integral characteristics of each connected component, like size, perimeter or area.

Appendix. Label Equivalence procedure listings

See listings.

Listing 1. Labels array initialization

```
//// Initialization with unique id's
__global__ void InitLabels(UINT* _Labels)
{
  UINT id = blockIdx.y*gridDim.x*blockDim.x+
    blockIdx.x*blockDim.x+threadIdx.x;
  UINT cy = id/SIZEX;
  UINT cx = id - cy*SIZEX;
  UINT aPos = (cy+1)*(SIZEXPAD)+cx+1;
  UINT l = _Labels[aPos];
  l*=aPos;
  _Labels[aPos] = l;
}
```

Listing 2. Relabeling phase

```
__global__ void Analysis(UINT* _Labels)
{
  UINT id = blockIdx.y*gridDim.x*blockDim.x+
    blockIdx.x*blockDim.x+threadIdx.x;
  UINT cy = id/SIZEX;
  UINT cx = id - cy*SIZEX;
  UINT aPos = (cy+1)*(SIZEXPAD)+cx+1 ;
  UINT label = _Labels[aPos];
  if(label)
  {
    UINT r=_Labels[label];
    while(r!=label)
    {
      label = _Labels[r];
      r = _Labels[label];
    }
    _Labels[aPos] = label;
  }
}
```

Listing 3. Linking phase

```
__global__ void Scaning(UINT*_Labels,UINT*_IsNotDone)
{
  UINT id = blockIdx.y*gridDim.x*blockDim.x+
    blockIdx.x*blockDim.x+threadIdx.x;
  UINT cy = id/SIZEX;
  UINT cx = id - cy*SIZEX;
  UINT aPos = (cy+1)*(SIZEXPAD)+cx+1 ;
  UINT l = _Labels[aPos];
  if(l)
  {
    UINT lw = _Labels[aPos - 1];
    UINT minl = ULONG_MAX;
    if(lw) minl = lw;
    UINT le = _Labels[aPos + 1];
    if(le&&le<minl) minl = le;
    UINT ls = _Labels[aPos - SIZEX - 2];
    if(ls&&ls<minl) minl = ls;
    UINT ln = _Labels[aPos + SIZEX + 2];
    if(ln&&ln<minl) minl = ln;
    if(minl<l)
    {
      UINT ll = _Labels[l];
      _Labels[l] = min(ll,minl);
      _IsNotDone[0]=1;
    }
  }
}
```

Listing 4. Size calculation

```
//Size calculation
UINT n = _CSize[aPos];
if(n&&label!=aPos)
{
  atomicAdd(&_CSize[label],n);
  _CSize[aPos] = 0;
}
```

References

[1] R. Dewar, C. Harris, Parallel computation of cluster properties: application to 2d percolation, Journal of Physics A: Mathematical and General 20 (1987) 985–993.
[2] http://gpgpu.org/.
[3] K. Hawick, A. Leist, D. Playne, Parallel graph component labelling with GPUs and CUDA, Parallel Computing 36 (12) (2010) 655–678.
[4] J. Hoshen, R. Kopelman, Percolation and cluster distribution. i. cluster multiple labeling technique and critical concentration algorithm, Physical Review B 14 (8) (1976) 3438–3445.
[5] M. Manohar, H. Ramapriyan, Connected component labeling of binary images on a mesh connected massively parallel processor, Computer Vision, Graphics, and Image Processing 45 (2) (1989) 133–149.
[6] NVIDIA, Cuda programming guide 3.0, http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf.
[7] NVIDIA, Opencl programming guide 2.3, http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_OpenCL_ProgrammingGuide.pdf.
[8] W. Pratt, Digital Image Processing: PIKS Inside, John Wiley & Sons Inc., New York, NY, USA, 2001.
[9] A. Rosenfeld, A. Kak, Digital Picture Processing, Academic Press Inc., Orlando, FL, USA, 1982.
[10] K. Suzuki, I. Horiba, N. Sugie, Linear-time connected-component labeling based on sequential local operations, Computer Vision and Image Understanding 89 (1) (2003) 1–23.
[11] K. Wu, E. Otoo, K. Suzuki, Optimizing two-pass connected-component labeling algorithms, Pattern Analysis & Applications 12 (2) (2009) 117–135.

Oleksandr Kalentev received his M.S. Degree from Kharkov State University (Kharkov, Ukraine) in 2001, and in August 2008 he was awarded a Ph.D. in Physics from Ernst–Moritz–Arndt University (Greifswald, Germany). He is currently a Postdoctoral researcher at the Max–Planck-Institute of Plasma Physics (Greifswald, Germany). His research interests are computational plasma physics, plasma surface interaction, modeling of 3D plasma transport and Particle-in-Cell simulation of the Hall thruster.

Abha Rai did her Bachelor of Science at Delhi University, India and Master of Science from the Indian Institute of Technology, Delhi, India. Afterwards she obtained her Ph.D. from the Max–Planck Institute of Plasma Physics, Greifswald, Germany. She worked intensively in the field of fusion related material science. Currently she is working on transport in plasmas and GPU application for silver cluster dynamics.

Stefan Kemnitz has been studying computer science since 01.09.2008 at the University of Applied Sciences in Stralsund. He has a special interest in code optimization and GPU computing.

Ralf Schneider received the diploma degree in Physics from the Universität Würzburg in 1986, and the Ph.D. Degree in physics in 1989 from the Universität Bayreuth. He habilitated 2001 at the Universität Greifswald in Theoretical Physics, where he also lectures. He has worked since 1990 as a scientist at the Max–Planck-Institut für Plasmaphysik in Garching in the tokamak physics division and since 1998 in the stellarator theory division in Greifswald. From 1.1.2005 to 31.12.2009 he was Head of the Helmholtz Junior Research Group 'Materials in contact with plasmas'. Since 1.1.2010 he is professor for 'Computational Material Science' at the Institute of Physics at the Ernst–Moritz–Arndt University in Greifswald. His research is on computational physics: from the multi-scale modelling of plasma-wall interactions for fusion and low temperature plasmas including thrusters to quantum chemistry of hydrocarbons and modelling of sediment transport in the Baltic Sea. He is the author of more than 250 journal articles and book contributions.