DOI: 10.2352/ISSN.2470-1173.2016.2.VIPC-240

Block Equivalence Algorithm for Labeling 2D and 3D Images on GPU

Sergey Zavalishin; Ryazan State Radio Engineering University; Gagarin St., 59/1, Ryazan, Russia; ss.zavalishin@gmail.com
Ilia Safonov; National Research Nuclear University MEPhI; Kashirskoe Shosse St, 31, Moscow, Russia; ilia.safonov@gmail.com
Yury Bekhtin; Moscow State Technical University of Radio Engineering, Electronics and Automatics; Prospect Vernadskogo St, 78, Moscow, Russia; yury.bekhtin@yandex.ru
Ilia Kurilin; Samsung R&D Institute Russia; Dvintsev St, 12/1, Moscow, Russia; ilya.kurilin@samsung.com

Abstract

In this paper we propose a block equivalence algorithm for connected component labeling of 2D and 3D images on GPU. Usage of square pixel blocks in our solution allows reducing twice computational complexity in comparison with existing label equivalence methods. In contrast to well-known block-based algorithms, we don't rely on decision tables to reduce amount of memory accesses. Instead, we propose a different technique based on pixel scan mask that better suits to GPU architecture. We show, theoretically and experimentally, that our approach outperforms many existing CPU and GPU algorithms for connected component labeling. We also demonstrate, how to extend our method to label 3D volumetric images and that it has significant performance advantage over a simple label equivalence algorithm.

1. Introduction

Object detection is one of the most common problems in modern computer vision. In the case of binary images, it may be performed using connected component labeling (CCL) technique, which aims to assign each object in scene unique numeric label.

In the terms of CCL object is represented by a group of adjacent white pixels, while background consists of the black ones. We distinguish two types of pixel connectivity: four-connected and eight-connected pixels. The most of CCL algorithms were designed to deal with both types of connectivity, but in recent years we have seen many new approaches, based on assumption that all image pixels are eight-connected. While universal algorithms such as [1-3] are forced to deal with image pixels, eight-connectivity based algorithms may be applied to 2x2 blocks. This approach was first introduced by Grana in [4]. He noticed that all pixels within 2x2 block always share the same label (see figure 1), which allows reducing of labelling efforts. His idea was widely adopted by many authors, who proposed similar algorithms with different extensions [5-10].

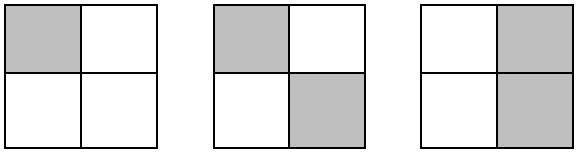

Figure 1. Examples of different 2x2 pixel blocks with the same label

The critical point of any block-based algorithm is a necessity to read each pixel multiple times: for instance, if there are two adjacent pixels A and B inside the same block, and pixel C, which belongs to another block is 8-connected to both of them, we will check it twice to evaluate relation between first A and C and then B and C. An obvious way to avoid that is addition of conditional checking: if we have checked pixel C, we don't need to check it again. Grana solves the problem using large decision tables. Later papers were aimed to improve his approach by increasing the effectiveness of pixel checking with better decision conditions [6, 7] and by applying different spatial apertures [5, 8].

Another important characteristic of CCL algorithm is a pixel access pattern, which may be regular [1-10] and irregular [11, 12]. Regular access is a perfect solution for single core CPU architecture, as it leads to a better cache utilization. But it also has a drawback: regular access provides very limited ability for concurrent processing. On the other hand, irregular access often requires perform multiple passes through image, while sequential algorithms need only one [2], two [1, 3-6] or one-and-a-half [8] scans.

The majority of attempts to adopt CCL algorithms to parallel architectures had very limited application. But fast development of modern GPGPU technologies, such as OpenCL and CUDA, make it possible to apply them for CCL processing using a broad range of devices [13-15]. There are two different ways to achieve that. The first one is a simple adaptation of sequential approaches to run on GPU [13]. Often it means, that we process individual image slices in parallel with distinct computing units and then merge them. The drawback of this approach is quite obvious: merging cannot be done in parallel, so it's hard to achieve good performance.

Another scenario adopts multi-pass methods with irregular access. In contrast to CPUs, GPU architectures can benefit from processing multiple pixels at once, so multi-pass algorithms can be easily adopted for them. In 2011 Kalentev proposed GPU labeling algorithm [14], that extends label equivalence technique. His approach utilizes two interleaving steps: on the first step each pixel is labeled with the lowest label among its neighbors; and on the second label equivalences are resolved by finding the roots of equivalence trees stored in label map, where equivalence trees represent hierarchy between neighbor labels. These two steps are performed iteratively until there will be no changes inside the label map.

In this work we propose a block equivalence algorithm, which applies 2x2 blocks to resolve label equivalence in eight-connected images. In contrast to other block-based methods, our algorithm doesn't rely on large decision tables. Instead, we propose another technique based on pixel connectivity mask, which fits well for GPU architecture. We show that our algorithm demonstrates comparable results to well-known CPU labeling algorithms. Our approach can be extended to label 3D images and it significantly outperforms label equivalence algorithm.

DOI: 10.2352/ISSN.2470-1173.2016.2.VIPC-240

2. Related Work

Let's briefly describe the label equivalence technique [14]. The algorithm requires initial binary image and label map to store output. It consists of three phases: initialization, scanning and analysis. Initialization is performed only once, while two other phases are executed iteratively, one by one, while no changes will be presented in label map. Each step is performed in parallel; each computing unit processes its own pixel. All intermediate changes are stored in the label map; thus algorithm requires no additional memory. It makes label equivalence a great choice for labeling large images, because algorithm capabilities are limited by available memory amount only.

In the initialization phase, each non-zero image pixel gets a unique label corresponding to its index in 1D pixel array. If X_i and Y_i are the coordinates of i^{th} pixel and W is an image width, label may be evaluated as following: $L_i := X_i + Y_i \cdot W + 1$ (we assume that index of first element on array is 0). To exclude processing of pixels, which are equal zero, we assign zero for corresponding labels.

The scanning phase is illustrated in *Algorithm 1*. In this phase we compare neighboring labels of each pixel with its current label and assign the lowest of them to it. At the step 5 function *FindMinLabel* gets the minimal label among of eight (or four) pixel neighbors, except the zero ones. To make this process more efficient, we access not the pixel label itself, but the label it references to, according to the algorithm step 7.

Algorithm 1. Scanning phase of label equivalence algorithm

```
1:  Pos ← Workitem ID;
2:  Labels ← Array of image labels;
3:  L := Labels[Pos];
4:  if L > 0 then
5:      L_min := FindMinLabel(Pos);
6:      if L_min < L then
7:          Labels[L - 1] := min(Labels[L - 1], L_min);
8:      end if
9:  end if
```

In the analysis phase (*Algorithm 2*) we walk through the label map and resolve label equivalences. Labels, assigned in the initialization phase, are array indexes. Thus, we can use them to find root of the equivalence tree. If the pixel index and its label value are different then we iterate through the label map as following: $L_{i,n} := Labels[L_{i,n-1}]$, where $L_{i,n}$ is a current label at n^{th} iteration and *Labels* is a label map. When the condition $L_{i,n} = Labels[L_{i,n-1}]$ is met, we stop and assign the final label to current pixel: $Labels[L_{i,0}] := L_{i,n}$. The second and the third phases are repeated iteratively until no changes occurred inside the label map.

Algorithm 2. Analysis phase of label equivalence algorithm

```
1:  Pos ← Workitem ID;
2:  Labels ← Array of image labels;
3:  L := Labels[Pos];
4:  if L > 0 then
5:      L_cur := Labels[L - 1];
6:      while L_cur ≠ L do
7:          L := Labels[L_cur - 1];
8:          L_cur := Labels[L - 1];
9:      end while
10:     Labels[Pos] := L;
11: end if
```

For the simplicity we illustrate that using a simple example (figure 2). Let's assume, that we process pixel with index 9. Its label is 9, so $9 - 1 \neq 9$. It means, the label is a reference; and we need to walk through the label map to find out equivalence tree root. Label of pixel with previous index $9 - 1 = 8$ also differs from its index ($8 - 1 \neq 8$), so, the next possible candidate for the root is 7. After several iterations we find out that label in position 0 is similar to its index ($0 - 1 = 0$). Thus, pixel at index 9 corresponds to label 1.

The process itself is very similar to an iterative label distribution; nevertheless, labels are distributed both across nearby neighbors and between different image parts. However, in the case of large-sized images, systems with a small number of parallel processing units cannot reach the claimed good performance. That is, applying label equivalence to 2x2 blocks rather than to individual pixels may increase performance by four times.

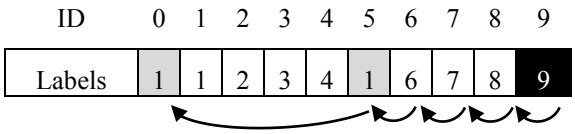

Figure 2. An example of analysis phase for the label equivalence algorithm.

Decision tables can reduce block processing complexity, as it was suggested by Grana in [4]. He utilized window, which is shown in figure 3. It's assumed, that we always process block X. Thus, in the case if we want to evaluate connectivity between this block and block Q, we should check connectivity between pixels o, p, i and j. Basically, we can first check o, i, j and then p, i, j, but it leads to doubling the number of memory reading operations. Decision tables allow reading each pixel only once. For instance, if pixel o is white and pixel i is also white, it means, that blocks X and Q are connected, thus there's no need to perform additional checks and pixels p and j may be ignored.

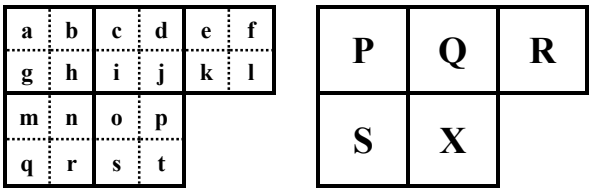

Figure 3. Search window proposed by Grana: left – pixel layout, right – super pixel layout

There are two ways to implement decision tables: using many nested if-then branches or by function pointers. Currently, GPU architectures don't support function pointers, so branching is the only alternative. But in contrast to CPUs, GPUs don't benefit from branches: due to architecture limitations, each workitem, which is similar to thread on GPU, executes all possible branches, but apply only those of them, which are valid according to the branch conditions (see [16]). Hence, utilization of large decision tables on GPUs is totally infeasible. To avoid this problem, we propose using pixel scan mask, which fits GPU architecture perfectly.

DOI: 10.2352/ISSN.2470-1173.2016.2.VIPC-240

3. Proposed Algorithm

3.1. 2D Images Labeling

Let's outline our algorithm prerequisites. Block equivalence algorithm takes binary image as an input. Depending on requested results, it may output either pixel label map or block label map. We also need to store intermediate block connectivity map to determine, which blocks are connected. That is, total memory occupancy is 6.25N bytes, where N is a number of input image pixels. Here 1.25N bytes are required to store input image and block connectivity map and another 5N bytes stand for block label map and pixel label map. We assume, pixel and its connectivity are stored using a single byte, and label requires at least 4 bytes.

Algorithm performs in four phases: 1) Initialization; 2) Scanning; 3) Analysis; and 4) Final labeling. Initialization phase is needed to obtain block connectivity map and to perform initial block labeling. Scanning and analysis are performed iteratively, one by one, until no changes are presented in the block label map. The last phase of final labeling converts block label map into pixel label map. This phase is optional and may be omitted. In this case algorithm memory occupancy is reduced to 2.25N bytes. All the phases are performed on image blocks, so the number of workitems is always 0.25N, which significantly reduces amount of work to be performed in scanning and analysis phases.

Initialization phase is the most complex phase in the whole algorithm. As it was mentioned above, decision tables are not feasible for GPUs, and we propose using pixel scan mask, which is shown in figure 4a. A-D are the pixels of the current block and all the other pixels are pixels of neighbor blocks, where 0x0 belongs to top-left block, 0x1 and 0x2 to top block, 0x3 to top-right block and so forth. The hexadecimal numbers (we use C-language notation) represent bit positions of the corresponding pixels in the mask. Depending on pixel configuration in A-D block, we initialize these bits with ones if we need to check them to evaluate connectivity and with zeroes otherwise.

0x0	0x1	0x2	0x3
0x4	A	B	0x7
0x8	C	D	0xB
0xC	0xD	0xE	0xF

a

1	1	1	0
1	1	1	0
1	1	1	0
0	0	0	0

b

Figure 4. Pixel layout (a) for typical 2x2 block and corresponding search pattern (b) for pixel A encoded by hexadecimal value 0x777

A trivial example of a scan mask is shown in figure 4b. Firstly, we check if there is a pixel in position A. If it is, then we set scan mask to 0x777, which has the following binary code: 11101110111. It means, we are going to check first three pixels of the first three rows (enumeration starts from the lowest bit). Pixels in positions 0x5, 0x6, 0x9 and 0xA belong to the current block, and the pixels are ignored while checking for connectivity. Scan masks for pixels B-D may be obtained in the following manner: if the pixel is located at the right from the A (pixel B), we shift pattern 0x777 one bit left. If it's located at the bottom (pixel C), we shift the pattern 4 bits left. Final search pattern for any combination of pixels A-D is obtained using bitwise OR operations, as it's shown in the steps 4-11 of algorithm 2 (SHL is a bitwise left shift function and OR' is a bitwise OR).

Once the pixel scan mask is obtained, we start checking adjacent blocks for connectivity. Steps 13-20 of algorithm 2 demonstrate this process. First we determine whether we need to scan neighbor pixel. Function *HasBit* returns true if search pattern has non-zero bit in specified position and false otherwise. Then we read pixel and in the case if it is white we set corresponding bit in block connectivity mask. The mask represents connectivity between adjacent blocks. Its layout is shown in figure 5. X stands for pixels A-D, and other bits represent adjacent blocks. Non-zero bit value means the central block and its neighbor are connected. If the first condition in conditional statement determines the result all the rest conditions are ignored, thus in the case if the first neighbor block pixel is white the second one won't be checked (see step 17 of algorithm 3). This approach guarantees, that each pixel is read only once by each workitem. One can see, that we don't use nested if-then statements, which makes such code suitable for GPU architectures.

Algorithm 3. Initialization of block map

```
1:  x, y ← Workitem x and y coordinates in block array;
2:  w ← Block array width;
3:  Pixels ← Array of image pixels;
4:  bLabels ← Array of block labels;
5:  bConn ← Array of block connectivity patterns;
6:  P := 0x0;
7:  P0 := 0x777;
8:  if Pixels[2x, 2y] > 0 then P := P OR' P0; end if
9:  if Pixels[2x+1, 2y] > 0 then P := P OR' SHL(P0, 1); end if
10: if Pixels[2x, 2y+1] > 0 then P := P OR' SHL(P0, 4); end if
11: if Pixels[2x+1,2y+1] > 0 then P := P OR' SHL(P0, 5); end if
12: if P > 0 then
13:     bLabels[x + y·w] := x + y·w + 1;
14:     if HasBit(P, 0x0) AND Pixels[2x-1, 2y-1] > 0 then
15:         SetBit(bConn[x+y·w], 0x0);
16:     end if
17:     if (HasBit(P, 0x1) AND Pixels[2x, 2y-1] > 0) OR
            (HasBit(P, 0x2) AND Pixels[2x+1, 2y-1] > 0) then
18:         SetBit(bConn[x+y·w], 0x1);
19:     end if
20:     ...
21: end if
```

Scanning and analysis phases are performed in the same manner, as it's shown in algorithms 1 and 2. The only difference is that now we work with blocks rather than with individual pixels; hence, *Labels* array becomes *bLabels*, which represents block label map. In contrast to label equivalence algorithm, our algorithm applies the function *FindMinLabel* only to those labels of adjacent blocks, which has non-zero bits in block connectivity map. These bits can be checked using *HasBit* function. It saves us additional processing time and reduces the number of memory accesses.

0x0	0x1	0x2
0x3	X	0x4
0x5	0x6	0x7

Figure 5. Block (and block label) layout

©2016 Society for Imaging Science and Technology
DOI: 10.2352/ISSN.2470-1173.2016.2.VIPC-240

Finding equivalence tree root for blocks has no significant difference from the same stage for pixels. Finally, the last phase takes block labels and copies them to the corresponding pixels inside the pixel label map.

Easy to see, that using 2x2 blocks for resolving equivalences reduces four times operations for scanning and analysis. Nevertheless, initialization and final labeling can significantly decrease performance of the proposed algorithm. Let's compare its complexity to label equivalence method. Memory reading and writing are the major bottlenecks for any labeling algorithm, thus we should minimize them as much as possible. Table 1 contains the number of memory operations for each phase of label equivalence and block equivalence algorithms. Here N is a number of image pixels and M is a maximal length of equivalence tree, which depends on image configuration. In the comparison we consider the worst case for our algorithm, which never occurs in real-life images. According to the table, in theory our method is at least two times faster than the label equivalence algorithm.

Table 1. Complexity analysis of 2D algorithms

Kernel	LE	BE
Initialization	2N	3.25N
Scanning	10N	2.5N
Analysis	2N + M	0.5N + 0.25M
Final labeling	-	1.25N
Total	**14N + M**	**7.25N + 0.25M**

3.2. 3D Images Labeling

Labeling of volumetric 3D images using our algorithm includes the same phases as in 2D case, but with larger number of comparisons. Similar to 2D, we firstly apply initialization phase to obtain connectivity between cubes and set initial cube labels. Scanning and analysis are performed iteratively, one by one, until no changes will be presented in cube label map. Final labeling is optional and it's used to fill voxel label map with labels from the cube map.

Memory requirements for the algorithm are the following: in 3D we operate with 2x2x2 cubes, thus we need 6N bytes for all intermediate data, where 1.5N bytes are required for initial image voxels and cube connectivity map and 4.5N bytes for voxel labels and cube labels. Here we should note that 3D images use 26-connected voxels. It means, that we cannot store connectivity map for each cube using a single byte, because we need at least 26 bits. The nearest data type is 32 bits long.

Another difference from 2D algorithm is that we need apply three-dimensional voxel search pattern while performing initialization step. Its layout is shown in figure 6. In the case if we need to check all neighbors around voxel A1, we apply search pattern 0x77707770777, which means that we want to check voxels in left-top cube 3x3. This pattern may be adapted to any voxel from A1-D2 by applying bit shift operations. We can move it along x by shifting its bits by one bit left, along y by shifting by four bits left and along z by shifting by sixteen bits left. Similar to 2D case, we can get any combination of search patterns using bitwise OR operation.

Obtained search pattern is applied to evaluate connectivity mask between cubes in 3x3x3 cell as it is shown in steps 14-20 of algorithm 2. We save connectivity masks for each cube in three-dimensional array to use them in scanning phase. Along with it we store initial labels, which are initiated according to the cube index: $L_i := X_i + Y_i \cdot W + Z_i \cdot W \cdot H + 1$. Here X_i, Y_i and Z_i are the coordinates of i^{th} cube in an image, W and H – cube map width and height and L_i – cube label.

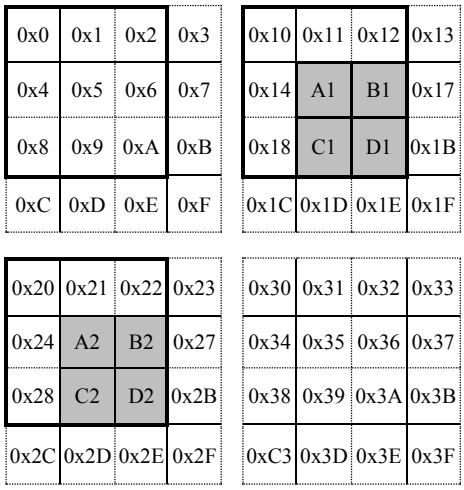

Figure 6. Layout of 2x2x2 cube with its neighbors, aligned by slices. Dark cells contain cube voxels. Outlined area 3x3x3 illustrates search pattern for voxel A1, which is encoded with hexadecimal value 0x77707770777

Scanning and analysis are performed in the same manner as in 2D case, except the fact that we work in 3D. Both of them require eight times less work to perform labeling in comparison to the label equivalence technique, so we save a huge amount of time while performing proposed algorithm. Table 2 contains estimations of complexity of these two algorithms by the means of the number of memory reads and writes. Easy to see, that our algorithm has more than two times lower complexity than the label equivalence.

Table 2. Complexity analysis of 3D algorithms

Kernel	LE3D	BE3D
Initialization	2N	8.125N
Scanning	30N	3.75N
Analysis	2N + M	0.25N + 0.125M
Final labeling	-	2.125N
Total	**34N + M**	**14.25N + 0.125M**

4. Results and Discussion

4.1. 2D Images Labeling

We compared our block equivalence algorithm (*BE*) with multiple well-known labeling algorithms for CPUs and GPUs. We analyzed the following algorithms: *sBBDT* from [6], which is an improvement of original block-based algorithm proposed by Grana in [4]; *BOS*, *BTS* and *FOS* algorithms from [8], where *BOS* and *BTS* are block-based algorithms and *FOS* is a pixel-based one-and-half scan algorithm; pixel-based *EFS* from [17]; and GPU-based methods *CCLC* from [15] and *LE* from [14].

We performed all tests on the following hardware configuration: CPU Intel Core i7-2600 with 4 cores and 3.4 GHz processor frequency; GPU Nvidia GTX650 with 384 CUDA cores, 1024 MB 128-bit GDDR5 and 1058 MHz processor frequency; 16 GB RAM. All algorithms were compiled on PC with Windows 7

©2016 Society for Imaging Science and Technology
DOI: 10.2352/ISSN.2470-1173.2016.2.VIPC-240

64-bit operating system using Microsoft Visual Studio 2013 compiler and OpenCL. Each figure was obtained by averaging the execution time for 100 runs. Minimal and maximal execution times are the best and the worst results across these 100 runs, respectively. Start and end time stamps were evaluated before algorithm initialization and after obtaining the final label map. It means, that GPU-based algorithm results were measured including overhead for running GPU kernels. Source code for *BOS*, *BTS*, *FOS* and *EFS* was downloaded from [18]. Source code for *sBBDT* was borrowed from [19].

To perform tests on natural images, we have downloaded images from [20], including aerials, misc. and textures datasets. We have also downloaded tobacco document dataset from [21] and created additional document dataset by ourselves, consisting of various document types with size larger than 5 Mpx. In contrast to many authors, we have inverted document images to label text letters rather than background. To make images binary, we have applied Otsu binarization.

Comparison was performed using multiple datasets. Firstly, we have compared aforementioned algorithms using synthetic dataset, which includes multiple noise patterns with different white pixel densities from 0.05 to 0.95. Similar to many other GPU algorithms, ours demonstrates results, comparable to CPU algorithms, starting from 1024x1024 image size. It happens because of massive overhead of GPU-based methods for running code on device. Figure 7 illustrates results of noise patterns labeling by different algorithms. Our algorithm demonstrates the best results for high and mid densities. For low densities *CCLC* GPU algorithm performs better.

The results of algorithm comparison on natural images are presented in table 3. One can see that our algorithm outperforms all the other algorithms for images larger than 2048x2048 pixels for all datasets, except aerials. Each aerial image contains a single large object, which is hard to label using our algorithm, as it tends to propagate labels from bottom-right to top-left object pixel.

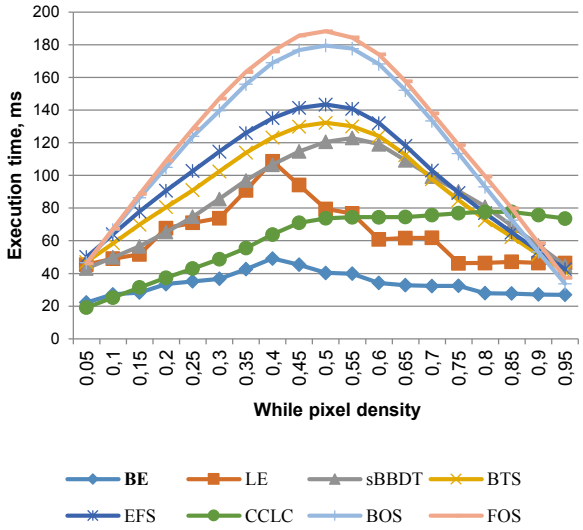

Figure 7. Performance of 2D algorithms on noise patterns with different densities, ms. Image size is 4096x4096.

For the smaller images *BOS* algorithm is the best one, but our algorithm demonstrates very close outcomes.

Table 3 also contains our algorithm results with final labeling disabled (see *BL (no FL)*). These results are given for the reference purposes only, because in this case the algorithm generates block label map, which is four times smaller than label maps from the other methods. Thus, *BL (no FL)* cannot be directly compared to them. Still, there are many cases, when block label map may be useful. One can see that with this little trick our algorithm demonstrates significant performance increase, which makes it faster for the majority of datasets.

Table 3. Performance of 2D algorithms on natural images, ms

Data set	Size	BE	BE (no FL)	LE	sBBDT	BOS	BTS	EFS	FOS	CCLC
Aerials	2048	10.64	9.56	23.21	10.56	**7.25**	9.81	10.25	7.45	15.48
	3072	20.60	18.02	49.19	24.68	**16.89**	22.94	23.99	17.78	32.87
	4096	34.06	**29.68**	72.04	44.63	**30.83**	41.29	44.55	31.91	55.25
	5120	**46.34**	**40.35**	102.86	70.40	48.83	64.85	69.23	50.41	-
Misc.	2048	7.21	**5.98**	12.90	10.56	**7.17**	9.73	10.16	7.39	18.09
	3072	**15.27**	**12.38**	26.13	24.68	17.17	22.67	23.84	17.49	40.18
	4096	**24.99**	**20.06**	44.51	44.55	30.52	41.60	43.07	31.74	69.85
	5120	**35.66**	**29.30**	68.48	70.42	48.40	66.02	68.23	50.52	-
Textures	2048	7.37	**6.20**	12.90	10.59	**7.06**	9.81	10.18	7.39	18.06
	3072	**14.88**	**12.11**	25.43	24.68	16.66	22.96	23.81	17.51	40.04
	4096	**24.64**	**19.64**	43.06	44.60	30.60	41.10	48.12	31.83	69.69
	5120	**34.73**	**28.23**	67.69	70.29	48.69	61.17	68.47	50.51	-
Tobacco		**6.79**	**5.86**	11.10	11.72	7.53	10.61	11.09	8.12	-
Docs		**10.32**	**8.84**	19.10	16.85	10.94	15.24	16.06	11.79	-

©2016 Society for Imaging Science and Technology
DOI: 10.2352/ISSN.2470-1173.2016.2.VIPC-240

4.2. 3D Images Labeling

We have compared two 3D labeling algorithms: 3D label equivalence (*LE3D*) and ours 3D block equivalence (*BE3D*). First, we performed a series of tests on synthetic images, containing noise patterns with different voxel densities. It's assumed that each 2D slice of voxel image has unique noise pattern, thus in this test algorithms label a large number of 3D noise structures, which occupy several slices. Depending on noise density, number of slices may vary from 2-3 for each structure to 200-250. The results for the noise pattern 256x256x127 are shown in figure 8. Easy to see that for all the densities our algorithm outperforms *LE3D*.

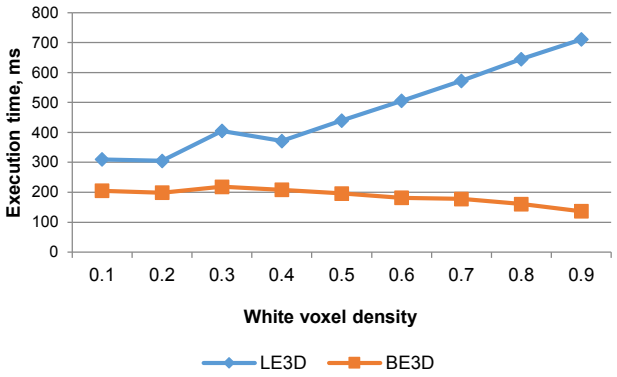

Figure 8. Performance of 3D algorithms on noise patterns with different densities, ms

Note, that starting from density 0.5 LE3D demonstrates significant performance degradation. This problem remains for any type of noise pattern with density higher than 0.4, which makes us think that it is related to hardware limitations. We were unable to perform a large number of tests with different GPUs to prove this, but our conclusions can be confirmed by results obtained using uniformly filled 3D image (see table 4). In the case of BE3D this image has the same number of object blocks as the number of object voxels in noise pattern with density 0.5. And similarly to LE3D, our algorithm demonstrates significant performance degradation while processing this image in comparison to noise patterns with smaller densities.

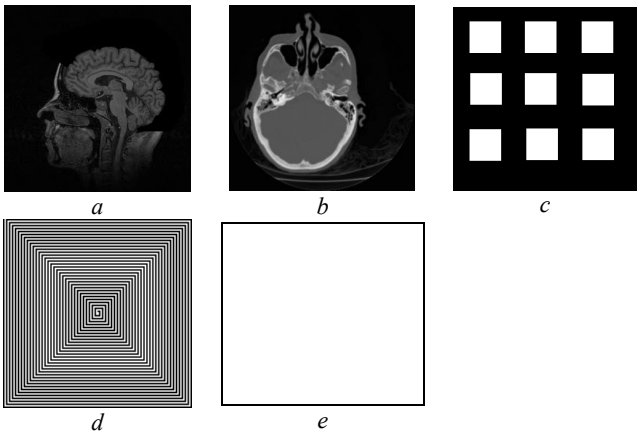

Figure 9. 3D image samples: a – mrbrain, b – cthead, c – cubes, d – spiral, e - uniform

For another test we have downloaded 3D images from [22]. Because of lack of available natural 3D volumetric images, we were forced to generate some images by ourselves. All the images are shown in figure 9. Image 9c has size 256x256x255 and it contains 45 cubes; image 9d has size 128x128x63 and it contains spiral-like object with no intersections; image 9e has size 256x256x255 and it contains only white pixels. According to our experiments, our *BE3D* algorithm outperforms *LE3D* on all natural images. As it was shown in complexity analysis section, our algorithm is about 2.5 times faster than *LE3D*. Similar to 2D case we also give *BL3D* results without final labeling stage, which is denoted as *BL3D (no FL)*.

Table 4. Performance of 3D algorithms on natural images, ms

Image	Time	BE3D	BE3D (no FL)	LE3D
cubes	Min	**191.4**	**132.5**	330.7
	Avg.	**192.0**	**132.9**	331.2
	Max	**193.4**	**134.0**	333.4
cthead	Min	**140.2**	**111.2**	483.2
	Avg.	**140.7**	**111.8**	484.5
	Max	**141.2**	**113.0**	487.9
mrbrain	Min	**113.1**	**88.1**	336.8
	Avg.	**113.3**	**88.7**	337.8
	Max	**114.0**	**91.8**	340.5
spiral	Min	**23.0**	**17.6**	51.4
	Avg.	**23.5**	**18.2**	52.0
	Max	**24.1**	**19.0**	52.1
uniform	Min	**405.2**	**305.7**	1018.2
	Avg.	**406.2**	**306.4**	1019.3
	Max	**406.9**	**307.4**	1022.5

5. Conclusion

We proposed a block equivalence algorithm for connected component labeling of 2D and 3D volumetric images on GPU. A new approach based on pixel scan mask, which reduces the number of pixel comparisons was designed to fit GPU architecture. We demonstrated that our algorithm complexity is about 2.5 less that complexity of label equivalence algorithm. The experimental results demonstrated that our approach outperforms existing CPU-based and GPU-based algorithms for a wide range of artificial and natural images. Our 3D labeling algorithm demonstrated 2.5 times better performance in comparison to 3D label equivalence algorithm.

Reference code for the algorithm is available at https://github.com/szavalishin/Labeling.

References

[1] Lifeng He, Yuyan Chao, Kenji Suzuki, A Run-Based Two-Scan Labeling Algorithm Image Processing, IEEE Transactions on , vol.17, no.5, pp.749,756, May 2008.

[2] Lifeng He, Yuyan Chao, Kenji Suzuki, and Hidenori Itoh. A Run-Based One-Scan Labeling Algorithm. In Proceedings of the 6th International Conference on Image Analysis and Recognition (ICIAR '09), 2009.

[3] He, L., Chao, Y., Yang, Y., Li, S., Zhao, X., & Suzuki, K. (2013). A Novel Two-Scan Connected-Component Labeling Algorithm. In *IAENG Transactions on Engineering Technologies* (pp. 445-459). Springer Netherlands.

DOI: 10.2352/ISSN.2470-1173.2016.2.VIPC-240

[4] C. Grana, D. Borghesani, and R. Cucchiara, "Optimized block-based connected components labeling with decision trees," IEEE Trans. Image Process, vol. 9, no. 6, pp. 1596–1609, Jun. 2010.

[5] He, L., Zhao, X., Chao, Y., & Suzuki, K. (2014). Configuration-Transition-Based Connected-Component Labeling. *Image Processing, IEEE Transactions on*, *23*(2), 943-951.

[6] Sutheebanjard, P., & Premchaiswadi, W. (2011). Efficient scan mask techniques for connected components labeling algorithm. *EURASIP Journal on image and Video Processing*, *2011*(1), 1-20.

[7] Chang, W. Y., & Chiu, C. C. (2014, June). An efficient scan algorithm for block-based connected component labeling. In *Control and Automation (MED), 2014 22nd Mediterranean Conference of* (pp. 1008-1013). IEEE.

[8] Santiago, D. J., Ren, T. I., Cavalcanti, G. D., & Jyh, T. I. (2013, May). Fast block-based algorithms for connected components labeling. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on* (pp. 2084-2088). IEEE.

[9] Grana, C., Borghesani, D., Santinelli, P., & Cucchiara, R. (2010, August). High Performance Connected Components Labeling on FPGA. In *Database and Expert Systems Applications (DEXA), 2010 Workshop on* (pp. 221-225). IEEE.

[10] Zhao, H. L., Fan, Y. B., Zhang, T. X., & Sang, H. S. (2010). Stripe-based connected components labelling. *Electronics letters*, *46*(21), 1434-1436.

[11] F. Chang, C.J. Chen, and C.J. Lu, "A linear-time component- labeling algorithm using contour tracing technique," Comput Vis Image Underst, vol. 93, n. 2, pp. 206-220, 2004.

[12] J. Martín-Herrero, "Hybrid object labeling in digital images. Machine," Vision and Applications, vol. 18, pp. 1-15, 2007.

[13] V. M. A. Oliveira, R. A. Lotufo, A study on connected components labeling algorithms using GPUs, SIBGRAPI (2010).

[14] Kalentev, O., Rai, A., Kemnitz, S., & Schneider, R. (2011). Connected component labeling on a 2D grid using CUDA. Journal of Parallel and Distributed Computing, 71(4), 615-620.

[15] Stava, O., and Benes, B.: 'Connected component labeling in CUDA' in 'GPU computing gems emerald edition' (Morgan Kaufmann, 2011), Chap. 35, pp. 569–581

[16] (2015, Aug) Nvidia OpenCL Best Practices Guide Version 1.0 [Online]. Available: http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf

[17] L. He, Y. Chao, and K. Suzuki, "An efficient first-scan method for label-equivalence-based labeling algorithms," Pattern Recognition Letters, vol. 31, n. 5, 2010.

[18] (2015, Aug) Source code of FOS, EFS, BTS and BOS algorithms [online]. Available: http://cin.ufpe.br/~djcs/labeling

[19] (2015, Aug) Source code of sBBDT algorithm [online]. Available: http://phaisarn.com/labeling

[20] (2015, Aug) USC-SIPI Image Database [online]. Available: http://sipi.usc.edu/database/

[21] (2015, Aug) Tobacco800 Document Image Database [online]. Available: http://www.umiacs.umd.edu/~zhugy/tobacco800.html

[22] (2015, Aug) The Stanford volume data archive [online]. Available: https://graphics.stanford.edu/data/voldata/

Author Biography

Sergey S, Zavalishin *received his MS degree in Computar Science from Moscow Engineering Physics Institute/University (MEPhI), Russia in 2012. Currently he is a post graduate student of Ryazan State Radio Electronics University. His research interests include machine learning, computer vision and image processing.*

Ilia V. Safonov *received his MS degree in automatic and electronic engineering from Moscow Engineering Physics Institute/University (MEPhI), Russia in 1994 and his PhD degree in computer science from MEPhI in 1997. Since 1998 he is an associate professor of National Research Nuclear University MEPhI while conducting researches in image segmentation, features extraction and pattern recognition problems.*

Yury S. Bekhtin *received his PhD degree from the Ryazan State Radio Electronics University (RSREU) in 1993, and a doctoral degree (habilitation) from RSREU in 2009. Currently he is a professor of RSREU and Moscow State Technical University of Radio Engineering, Electronics and Automatics (MIREA). He is the author of more than 120 journal papers. His current research interests include wavelet-based processing of noisy still images and video.*

Ilya V. Kurilin *received his MS degree in radio engineering from Novosibirsk State Technical University (NSTU), Russia in 1999 and his PhD degree in theoretical bases of informatics from NSTU in 2006.Since 2007, Dr. I. Kurilin has join Image Processing Group, Samsung RnD Institute Russia where he is engaged in photo and document image processing projects.*