J Real-Time Image Proc
DOI 10.1007/s11554-016-0652-5

SPECIAL ISSUE PAPER

GLSC: LSC superpixels at over 130 FPS

Zhihua Ban[1] · Jianguo Liu[1] · Jeremy Fouriaux[1]

Received: 22 June 2016 / Accepted: 3 November 2016

Abstract Superpixel has been successfully applied in various computer vision tasks, and many algorithms have been proposed to generate superpixel map. Recently, a superpixel algorithm called "superpixel segmentation using linear spectral clustering" (LSC) has been proposed, and it performs equally well or better than state-of-the art superpixel segmentation algorithms in terms of several commonly used evaluation metrics in superpixel segmentation. Although LSC is of linear complexity, its original implementation runs in few hundreds of milliseconds for images with resolution of 481 $\times$ 321 stated by the authors, which is a limitation for some real-time applications such as visual tracking which may needs, for instance, 30 FPS for standard image resolution (e.g., 480 $\times$ 320, 640 $\times$ 480, 1280 $\times$ 720 and 1920 $\times$ 1080). Instead of inventing new algorithms with lower complexity than LSC, we will explore LSC to modify its structure and make it suitable to be implemented by parallel technique. The modified LSC algorithm is implemented in CUDA and tested on several NVIDIA graphics processing unit. Our implementation of the proposed modified LSC algorithm achieves speedups of up to 80$\times$ from the original sequential implementation, and the quality, measured by two commonly used evaluation metrics, of our implementation keeps being similar to the original one. The source code will be made publicly available.

Keywords Real time · GPGPU · CUDA · Superpixel · Image segmentation

✉ Jianguo Liu
liujg11@126.com

Zhihua Ban
zhihua_ban@hust.edu.cn

Jeremy Fouriaux
jeremy.fouriaux@gmail.com

[1] National Key laboratory of Science and Technology on Multi-spectral Information Processing, School of Automation, Huazhong University of Science and Technology, Wuhan 430074, Hubei Province, China

1 Introduction

Superpixel (see Fig. 1 for illustration) is a perceptually homogeneous atomic image region. It is usually used to reduce the number of entries of high complexity algorithms and thus, improves their performance.

Superpixel algorithms have been successfully applied to many computer vision tasks, such as image segmentation [12], image parsing [13], visual tracking [18], saliency detection [6] and edge detection [17]. After the advent of superpixel in the work of [16], many superpixel algorithms [3, 4, 11] with different model have been proposed in the literature, each with its own advantages and drawbacks that optimally suit particular applications. For instance, superpixels algorithms with high quality may be paramount importance in medical image segmentation. If superpixels are used to track moving objects in real-time applications, computational efficiency should be the priority over quality criteria.

We believe that the execution speed of an implementation of a superpixel algorithm, as a preprocessing step, is quite critical for subsequent applications. There are two trends to improve the execution speed of implementations of existing algorithms: by reducing the number of calculations in the implementations and by exploiting parallelism that is inherently given in the existing algorithms and further implementing them with parallel technique to

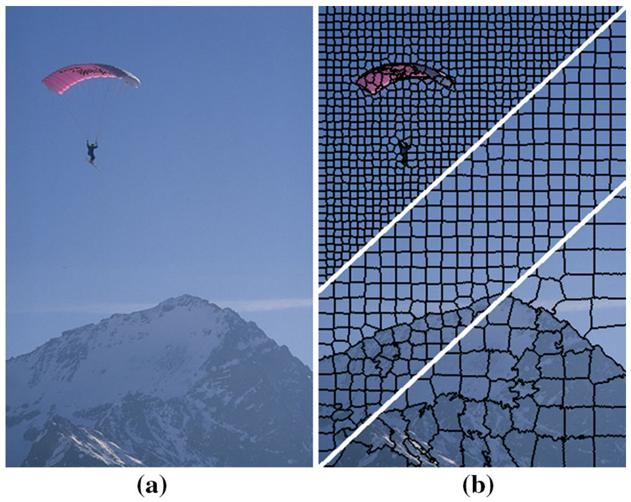


Fig. 1 Superpixel illustration. The original image is from the Berkeley segmentation database. **a** Original image. **b** Superpixels

exploit better the compute potential of architectures equipped with multi- or many-core processors. Both the two trends have achieved success. As an example because we know already we speak about superpixel, the execution speed of the well-known superpixel algorithm simple linear iterative clustering (SLIC) [1] has been successfully improved by two researches [9, 15]. The work of [9] reduces the number of distance calculation by exploiting spatial redundancy within natural images. A parallel modification is proposed in [15], and their implementation with CUDA is up to 83× faster than the original CPU implementation of SLIC as stated by the authors.

GPU has been becoming a well-known parallel computing architecture since NVIDIA introduced compute unified device architecture (CUDA) [14] and many works [5, 7] have achieved significant improvement in terms of time consumption using NVIDIA GPU. This work focuses on how to modify the recently proposed superpixel algorithm superpixel segmentation using linear spectral clustering (LSC) [11] to make it suitable to be implemented using parallel technique. The modified LSC is implemented using CUDA. In order to keep consistent with the original LSC algorithm [11] in terms of the quality of the generated superpixels, we have tuned the parameters of our modified LSC and comparisons between our modified LSC and the original LSC [11] are presented. The execution speed of the implementation of our modified LSC algorithm is evaluated on several different NVIDIA GPUs. It is shown that the differences in quality between superpixels produced by our modified LSC algorithm and the original LSC [11] algorithm are small and our implementation of the modified algorithm is up to 80× faster than the original implementation provided by the authors. Since our implementation tests on GPU, the modified LSC algorithm will be referred to as GLSC sometimes.

The remainder of this paper is organized as follows. Related works are presented in Sect. 2. Section 3 presents the original LSC [11], and our modified LSC targeted to GPU. We discuss in Sect. 4 the performances of our modified algorithm. Section 5 states our conclusion.

2 Related work

SLIC [1] and LSC [11] both use *k*-means like method to iteratively refine a superpixel. CIELAB color and spatial location of a pixel are used as features in SLIC [1] and LSC [11] transforms these five features into a ten-dimensional feature space. While in SLIC [1], *k*-means is performed on the 5D feature space and a limited region is searched for each cluster when updating the label of pixels, LSC [11] applies a weighted *k*-means method in the transformed 10D feature space with the same searching method when updating the label of pixels (see Sect. 3.1 for details).

Although the searching method saves computation with respect to the standard *k*-means or standard weighted *k*-means, this searching method results in dependency that a specific pixel in overlapping search region of several clusters may change label when any of them are processed. In order to make SLIC [1] being suitable to be implemented with parallel technique, gSLIC [15] updates the label of each pixel by comparing nine fixed neighboring clusters to remove that dependency. The similar updating method is applied into our modification to the original LSC algorithm [11]. To update clusters' representation or, in other words, to compute the mean value of each cluster, we also share a similar method to reduce redundant accessing by searching in a region with fixed size around the cluster location center. However, there are also many differences between gSLIC [15] and GLSC in terms of both algorithm modifications and implementation. For example, at algorithm level, there is no need to transform feature space for gSLIC [1] but our GLSC needs. And, at implementation level, we use CUDA warp to update an individual cluster's representation, and gSLIC [15] uses a whole CUDA thread block to update the representation. Our modification details are presented in Sect. 3.2.

3 The GPU-based superpixel algorithm

LSC [11] has achieved state-of-the-art performance in terms of quality of generated superpixels, (e.g., boundary adherence, regularity and perceptual satisfaction). Although it can run in few hundreds of milliseconds for images with size of 481 × 321, the original implementation is still not fast enough to comply with real-time applications such as object tracking in videos and image retrieval systems. Our work focuses on exploring parallel potential of the LSC algorithm [11] and taking advantage of many-core GPU architecture. In this section, we firstly introduce the original LSC algorithm [11] and then describe our modification and implementation on NVIDIA GPU with CUDA.

3.1 LSC superpixel algorithm

LSC [11] construct a bridge between normalized cuts and weighted k-means by mapping the input space to a feature space with higher dimensions in order to reduce the time complexity of normalized cut to linear. By further limiting the search space of each cluster to a fixed size which is related to the size of initial superpixel, LSC [11] can achieve a faster execution speed while retaining a good superpixel quality. There are several steps in the pipeline of LSC [11]: converting color space from RGB to CIELAB; mapping them into a high-dimensional feature space; performing weighted k-means with fixed search space to produce coarse superpixel map (pixels within the same superpixel may be disconnected in the image plane); small superpixels merging with their neighbors and relabeling pixels by considering their connectivity.

Let $w(p)$ be the weight of pixel p and $\phi(p)$ be the feature of pixel p mapping from CIELAB color space. The calculation of $w(p)$ and $\phi(p)$ is defined by Eqs. (1) and (2).

$$\begin{aligned}\phi'(p) =&\Big(C_c \cos\frac{\pi}{2}l_p,\ C_c \sin\frac{\pi}{2}l_p,\\ &2.55C_c \cos\frac{\pi}{2}\alpha_p,\ 2.55C_c \sin\frac{\pi}{2}\alpha_p,\\ &2.55C_c \cos\frac{\pi}{2}\beta_p,\ 2.55C_c \sin\frac{\pi}{2}\beta_p,\\ &C_s \cos\frac{\pi}{2}x_p,\ C_s \sin\frac{\pi}{2}x_p,\\ &C_s \cos\frac{\pi}{2}y_p,\ C_s \sin\frac{\pi}{2}y_p\Big),\\ w(p) =&\phi'(p)\cdot\frac{1}{|V|}\sum_{q\in V}\phi'(q) = \phi'(p)\cdot\bar{\phi}',\end{aligned} \tag{1}$$

$$\phi(p) = \frac{1}{w(p)}\phi'(p), \tag{2}$$

where C_c, C_s, V and $|\cdot|$ are correspondingly color factor, spatial factor, image pixel set and the number of elements in a specified set. l_p, α_p and β_p are the three components of pixel p in CIELAB color space. x_p and y_p are coordinates of pixel p on the image grid.

In LSC [11], the desired number of superpixel K or the initial superpixel size v_x in horizontal direction and v_y in vertical direction of the image plane is required. If K is as input, v_x and v_y are computed by equation below.

$$\begin{aligned}&n_x = \left\lfloor\sqrt{\frac{KV_x}{V_y}}\right\rfloor, \quad n_y = \lfloor K/n_x\rfloor,\\ &v_x = \lfloor V_x/n_x\rfloor, \quad v_y = \lfloor V_y/n_y\rfloor,\end{aligned} \tag{3}$$

where $V_{x/y}$ is the number of pixels on the horizontal/vertical direction of the image plane. Then, the initialized K is reset to $n_x \times n_y$. K seed pixels are then sampled uniformly over the whole image with horizontal and vertical intervals v_x and v_y. After moving each seed to its lowest gradient neighbor in the 3×3 neighborhood, these seeds are used as the search centers and their feature vectors are used as initial weighted means of the corresponding cluster or superpixel in this case. Each pixel is assigned to the cluster of which the weighted mean is closest to the pixel's feature vector in terms of Euclidean distance. After the pixel assignment, the weighted mean m_k and search center c_k of each cluster are updated by Eqs. (4) and (5).

$$c_{x,k} = \frac{\sum_{p\in\pi_k} x_p}{|\pi_k|}, \quad c_{y,k} = \frac{\sum_{p\in\pi_k} y_p}{|\pi_k|}, \quad c_k = (c_{x,k}, c_{y,k}), \tag{4}$$

$$m_k = \frac{\sum_{p\in\pi_k} w(p)\phi(p)}{\sum_{p\in\pi_k} w(p)}, \tag{5}$$

where π_k is the k-th cluster. The search space of each cluster is limited to the size of $2v_x \times 2v_y$. Repeat pixel assignment and cluster updating steps until the stopping criteria are met. A post-processing step will be used to enforce superpixel connectivity. The LSC [11] algorithm is summarized in Fig. 2.

3.2 Modified LSC algorithm for CUDA

In the original LSC algorithm [11], the updating of labels, as shown in Fig. 2 from line 9 to line 17, is cluster-wise. Imagine that one pixel is assigned to a label when the pixel locates in the search region of certain cluster, its label may be changed when processing another cluster's search region in which the pixel is also located if the condition of line 12 is met. Therefore, parallel threads cannot be distributed among clusters because of the potential dependencies among them. Similar to gSLIC [15], we update labels at pixel level. Each individual pixel is compared with its neighboring cluster centers and is assigned to the nearest cluster or superpixel here. By doing this, the dependence can be removed.

The updating of weighted means and search centers is another critical part in our modification. According to Eqs. (4) and (5), if we distribute parallel threads among clusters, updating phase needs to access all pixels for each single cluster. However, given a specific superpixel, most of the accessing contributes nothing to its weighted mean

```
Input  : Image data with CIELab color space and $K$.
Output: Label set $L(p)$.
 1: Map each image pixel $(l_p, \alpha_p, \beta_p, x_p, y_p)$ to a ten dimensional
    vector $\phi(p)$ in feature space.
 2: Compute $v_x$ and $v_y$ using equation 3.
 3: Sample seeds over the image uniformly at fixed horizontal and
    vertical intervals $v_x$ and $v_y$.
 4: Move each seed to its lowest gradient neighbor in the $3\times3$
    neighborhood.
 5: Initialize weighted mean $m_k$ and search center $c_k$ of each cluster
    using the corresponding seed.
 6: Set $L(p) =$ No Label.
 7: Set $D(p) = \infty$.
 8: repeat
 9:    for each weighted means $m_k$ and search center $c_k$ do
10:       for pixel $p$ in the $2v_x \times 2v_y$ neighborhood of $c_k$ in the
          image plane do
11:          $d =$ Euclidean distance between $\phi(p)$ and $m_k$.
12:          if $d < D(p)$ then
13:             $D(p) = d$.
14:             $L(p) = k$.
15:          end if
16:       end for
17:    end for
18:    Updates weighted means and search centers for all clusters
       using equation 4 and equation 5 with $\pi_k = \{p|L(p) = k\}$.
19: until weighted means of $K$ cluster converge.
20: Enforce superpixel connectivity.
```

Fig. 2 LSC superpixel algorithm

and search center due to the fact that the superpixel contains locally small ratio of the image pixels. Aiming to save computations, we only search a limited region around the search center of a given superpixel. This searching technique results in fewer pixels in the processing cluster being accessed when updating the weighted mean and search center of that cluster. When we update a certain superpixel, some pixels with the label of that superpixel may be ignored, since we access only a limited space around its cluster center. Fortunately, this optimization retains very small quality difference based on our experiments described in Sect. 4.

With these modifications mentioned previously, the computation of updating labels is independent at pixel level and the updating of weighted means and search centers are independent at cluster level. It becomes easier to move the computations to GPU by assigning parallel threads at pixel level or at cluster level. To explain our modification clearly with the purpose that discovering the parallel potential of the original LSC algorithm [11] for the parallel programming model SIMT (single instruction multiple threads), details of our modifications on each step in the pipeline of LSC [11] will be presented sequentially.

Converting color space from RGB to CIELAB is naturally capable of parallel implementation since the computation is independent for each pixel. We specify one thread per pixel for this task, and no additional modification is needed.

Compute weight $w(p)$ and feature vector $\phi(p)$ We split this task into three sequential substeps: computing $\phi'(p)$, computing the mean vector of $\phi'(p)$ and computing $w(p)$ and $\phi(p)$ with the output of the first two substeps. The computation of $\phi'(p)$ and the third substep are pixel independent, and each pixel is assigned with one thread. To get the mean vector of $\phi'(p)$ with $p = 0, \ldots, |V| - 1$, we use a parallel reduction algorithm to sum ϕ' and to compute its mean implemented on GPU similar to [8]. The parallel reduction procedure is composed of two kernels. G_y (e.g., $G_y = 32$) blocks with B_x (e.g., $B_x = 256$) threads in each block are created to launch the first kernel. As shown in Fig. 3, threads in an individual block sum up one part of ϕ' and store the partial sums to $\tilde{\phi}'$. The second kernel, showing in Fig. 4, is used to sum the G_y partial results produced by the first one and compute the mean of ϕ'.

Initialize weighted means m_k and $L(p)$ This step is running in cluster level independently. After the computation of n_x, n_y, v_x and v_y on CPU, the weighted means and search centers are initialized on GPU showing in Fig. 5. We prefer to use this initialization without moving to lowest gradient neighbor because it will not reduce the quality obviously but will save computing time. The label of each pixel in a $v_x \times v_y$ neighborhood of a search center is initialized to the index of that search center.

Assign each pixel to its nearest cluster Instead of using search space in the original LSC [11], we compute the Euclidean distance from current pixel to weighted means of nine clusters (cluster of current pixel in the previous iteration plus its eight neighboring clusters). Then the pixel is assigned to the nearest cluster. Therefore, this assignment

```
Input  : $\phi'$, $V_x$, $V_y$.
Output: $\tilde{\phi}'$.
 1: this kernel is executed by creating $G_y$ blocks with $B_x$ threads in
    each block for a partial sum of $\phi'$.
 2: $t$ = get_thread_id();
 3: $b$ = get_block_id();
 4: $\tilde{S}(t) = 0$; $x_p = t$; $y_p = b$;
 5: while $y_p < V_y$ do
 6:    while $x_p < V_x$ do
 7:       $\tilde{S}(t)$ += $\phi'(p)$; $x_p$ += $B_x$;
 8:    end while
 9:    $y_p$ += $G_y$;
10: end while
11: $B_x = B_x / 2$;
12: while $t < B_x$ do
13:    $\tilde{S}(t)$ += $\tilde{S}(t + B_x)$; $B_x = B_x / 2$;
14: end while
15: if $t == 0$ then
16:    $\tilde{\phi}'(b) = \tilde{S}(0)$;
17: end if
```

Fig. 3 Compute partial sums of $\phi'(p)$ with $p = 0, \ldots, |V| - 1$ and store them into $\tilde{\phi}'(b)$ with $b = 0, \ldots, G_y - 1$

```
Input :  $\tilde{\phi}'$, $|V|$.
Output:  $\bar{\phi}'$.
 1: this kernel is executed by creating one block with $\frac{G_y}{2}$ threads.
 2: $t$ = get_thread_id(); $G_y = G_y$ / 2;
 3: while $t < G_y$ do
 4:     $\tilde{\phi}'(t)$ += $\tilde{\phi}'(t + G_y)$; $G_y = G_y$ / 2;
 5: end while
 6: if 0 == $t$ then
 7:     $\bar{\phi}' = \tilde{\phi}'(0)$ / $|V|$;
 8: end if
```

Fig. 4 Add the sequence of partial sums of $\tilde{\phi}'(t)$ with $t = 0, \ldots, G_y - 1$ and get the mean vector of ϕ'

```
Input :  $\phi$, $v_x$, $v_y$, $n_x$.
Output:  $m$, $c$.
 1: this kernel is executed by creating one thread for each weighted
    mean in parallel.
 2: $k_x$ = threadIdx.x + blockIdx.x × blockDim.x;
 3: $k_y$ = threadIdx.y + blockIdx.y × blockDim.y;
 4: $k = k_y \times n_x + k_x$;
 5: $x_p = k_x \times v_x + v_x$ / 2; $y_p = k_y \times v_y + v_y$ / 2;
 6: $m_k = \phi(p)$; $c_k = (x_p, y_p)$;
```

Fig. 5 Initialize the weighted mean m_k and the search center c_k with $k = 0, \ldots, K - 1$

can be executed in parallel for all pixels and suitable for SIMT. The procedure is shown in Fig. 6.

Update m_k and c_k In this step, we use one warp per cluster. Each warp threads will perform a parallel reduction independently to compute the weighted mean m_k and search center c_k. Instead of accessing all pixels for a single weighted mean, only a fixed search space which is related to the initial superpixel size and cluster search center in the previous iteration. Some pixels in certain superpixels may be ignored in this searching method. Practically, it will not degrade the quality of generated superpixels visually. However, execution time will vary with 2 parameters: the number of clusters and the size of search space chosen (see Sect. 4 for more details). For a specific application, it is encouraged to adjust trade-off between quality and execution speed as our implementation is fast enough to allow some real-time adjustments and the algorithm remains of linear complexity. The algorithm is presented in Fig. 7.

These operations stated in Figs. 6 and 7 are repeated a fixed number of iterations or until we found stopping conditions are fulfilled. This iterative process is identical to LSC algorithm [11]; the maximum number of iterations done is a fixed parameter. Since the connectivity enforcement is not suitable for GPU, we transfer back the labels from device memory to host memory and enforce the connectivity with CPU using the same method with the original LSC [11].

```
Input :  $\phi$, $m$, $L$, $n_x$
Output:  $L$
 1: this kernel is executed by creating one thread for each pixel in
    parallel.
 2: $x_p$ = threadIdx.x + blockIdx.x × blockDim.x;
 3: $y_p$ = threadIdx.y + blockIdx.y × blockDim.y;
 4: $k_x = L(p)$ % $n_x$; $k_y = L(p)$ / $n_x$; $D_m = \infty$;
 5: for all $y \in \{-1, 0, 1\}$ do
 6:     for all $x \in \{-1, 0, 1\}$ do
 7:         $k = (k_y + y) \times n_x + (k_x + x)$;
 8:         $D$ = distance from $m_k$ to $\phi(p)$;
 9:         if $D < D_m$ then
10:             $D_m = D$; $L(p) = k$;
11:         end if
12:     end for
13: end for
```

Fig. 6 Assign each pixel to its nearest cluster among its nine neighboring clusters

```
Input :  $\phi$, $w$, $L$, $c$, $v_x$, $v_y$.
Output:  $m$, $c$.
 1: this kernel is executed by creating one warp of threads for each
    cluster.
 2: $t$ = get_thread_id_in_a_warp();
 3: $k$ = get_warp_id(); $s$ = get_warp_size();
 4: $x_p = c_{x,k} - v_x$; $y_p = c_{y,k} - v_y$;
 5: $\tilde{c}_{x,t} = 0$; $\tilde{c}_{y,t} = 0$; $\tilde{m}_t = 0$; $\tilde{n}_t = 0$; $\tilde{w}_t = 0$;
 6: while $y_p < c_{y,k} + v_y$ do
 7:     while $x_p < c_{x,k} + v_x$ do
 8:         if $L(p) == k$ {$p \in \pi_k$ in other words} then
 9:             $\tilde{c}_{x,t} = \tilde{c}_{x,t} + x_p$; $\tilde{c}_{y,t} = \tilde{c}_{y,t} + y_p$; $\tilde{n}_t = \tilde{n}_t + 1$;
10:             $\tilde{m}_t = w(p)\phi(p)$; $\tilde{w}_t = \tilde{w}_t + w(p)$;
11:         end if
12:         $x_p = x_p + s$;
13:     end while
14:     $y_p = y_p + 1$;
15: end while
16: $s = s$ / 2;
17: while $t < s$ do
18:     $\tilde{c}_{x,t} = \tilde{c}_{x,t} + \tilde{c}_{x,t+s}$; $\tilde{c}_{y,t} = \tilde{c}_{y,t} + \tilde{c}_{y,t+s}$; $\tilde{n}_t = \tilde{n}_t + \tilde{n}_{t+s}$;
19:     $\tilde{m}_t = \tilde{m}_t + \tilde{m}_{t+s}$; $\tilde{w}_t = \tilde{w}_t + \tilde{w}_{t+s}$;
20:     $s = s$ / 2;
21: end while
22: if $t == 0$ then
23:     $m_k = \tilde{m}_t$ / $\tilde{w}_t$; $c_{x,k} = \tilde{c}_{x,t}$ / $\tilde{n}_t$; $c_{y,k} = \tilde{c}_{y,t}$ / $\tilde{n}_t$;
24: end if
```

Fig. 7 Update the weighted mean m_k and the search center c_k with $k = 0, \ldots, K - 1$

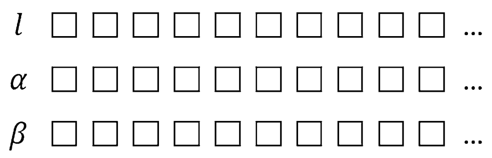


Fig. 8 The storage structure of CIELAB colors. l, α and β are the three components of CIELAB color

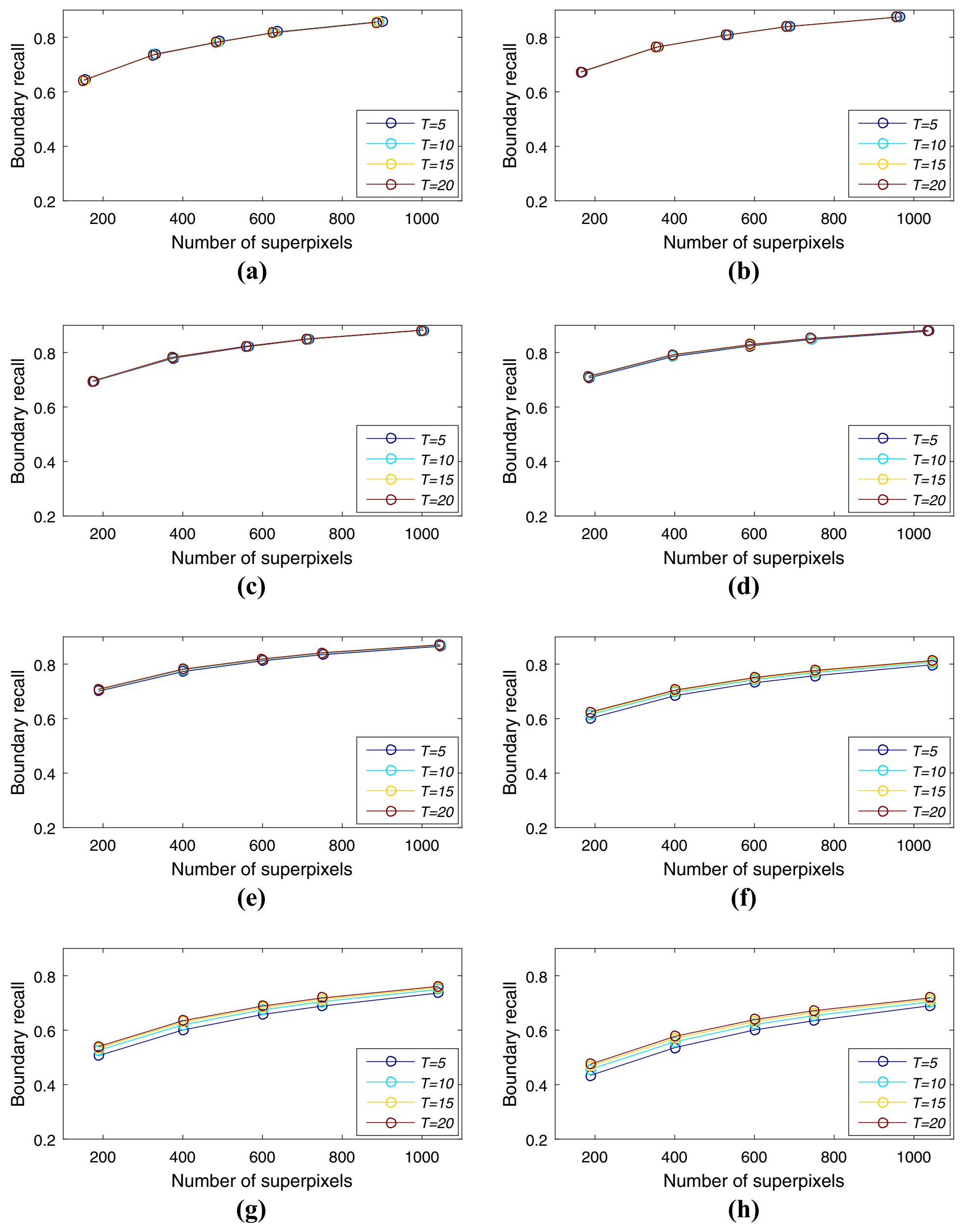


Fig. 9 Results of boundary recall. In each subfigure, r_c is set to one of the constants [0.05 (**a**), 0.075 (**b**), 0.1 (**c**), 0.15 (**d**), 0.2 (**e**), 0.4 (**f**), 0.6 (**g**), 0.8 (**h**)] and the number of iterations T varies from 5 to 20 with a step of 5. Once r_c and T are set, 5 experiments are done on all images of BSDS500 with $v_x = v_y \in \{27, 19, 16, 14, 12\}$. In each experiment, the average of the number of the generated superpixels and the average of boundary recall over 500 images are used to plot the figures

3.3 Implementing consideration

With the explanation of Sect. 3.2, people can implement their own CUDA code. For performances and precision measures, we have implemented GLSC using CUDA. In this section, we discuss implementation details including data type, storage structure and memory type in this section. Note that it is strongly encouraged to implement code by tuning our implementation provided publicly for target GPUs as our purpose here was not to heavily tune GLSC on CUDA, but to demonstrate feasibility of our parallel implementation.

We use single floating point precision for our computations, GPUs performances are enhanced, and we do not loose quality. In order to enable coalescing access on GPU global memory, components of colors and components of the 10D features are stored separately, and it is commonly referred as structure of array organization instead of array of structure. Figure 8 gives an intuitive explanation. These are common techniques that can be applied to implementations of other algorithms.

In the implemented kernels that perform reduction, variables, such as $\tilde{S}(t)$ in line 4 of Fig. 3 and all variables in line 5 of Fig. 7, which are shared among threads in the same thread block should be stored in shared memory. In the implemented kernel of algorithm presented in Fig. 6, it is recommended to read m_k using texture fetch functions when the distance is computed at line 8. Neighboring

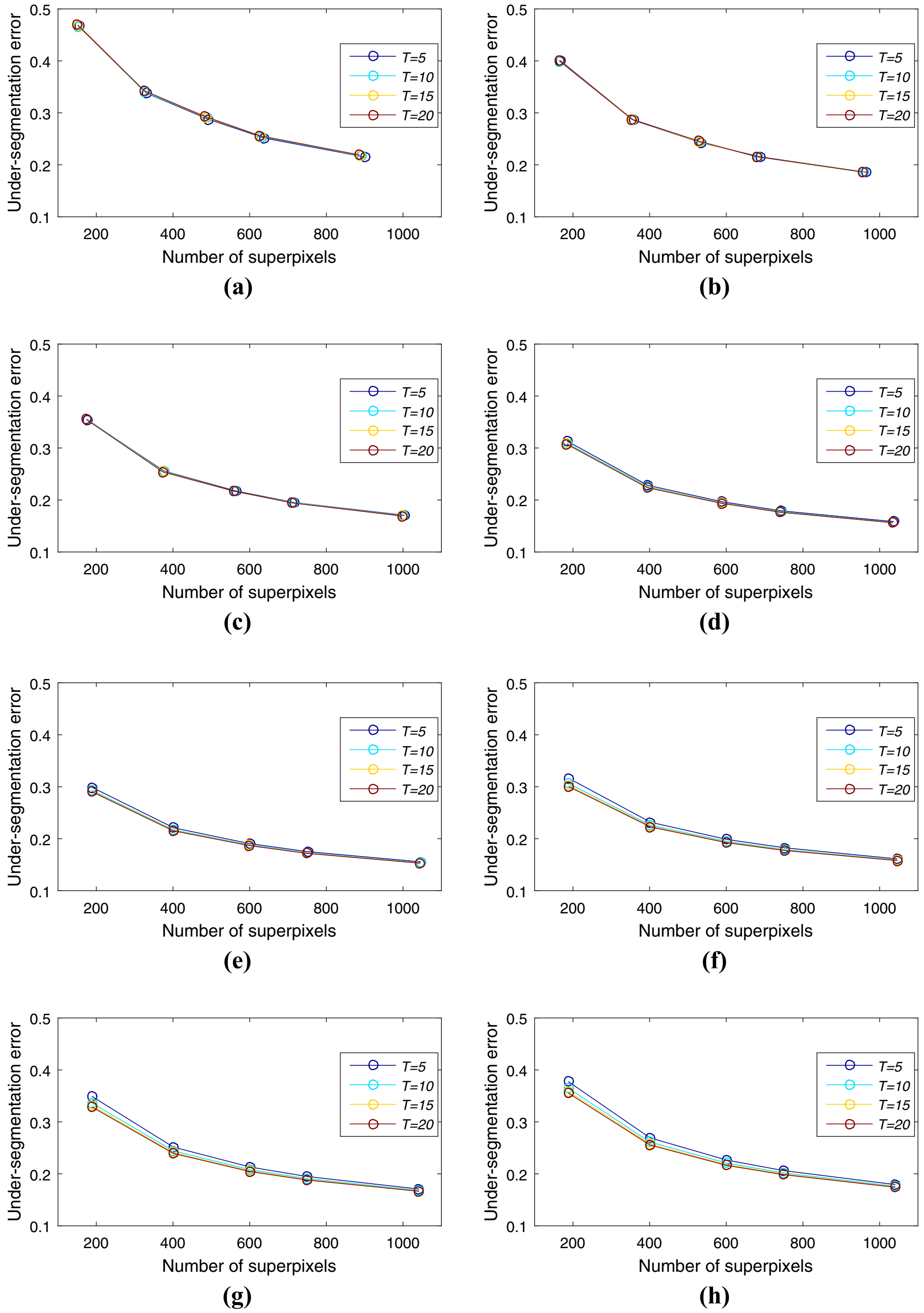


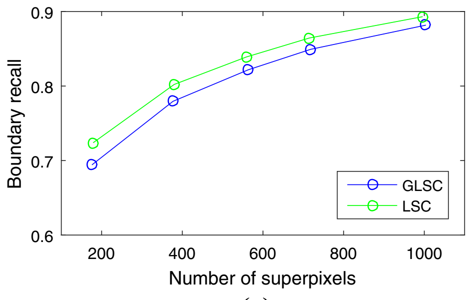
Fig. 10 Results of under-segmentation error. In each subfigure, r_c is set to one of the constants [0.05 (**a**), 0.075 (**b**), 0.1 (**c**), 0.15 (**d**), 0.2 (**e**), 0.4 (**f**), 0.6 (**g**), 0.8 (**h**)] and the number of iterations T varies from 5 to 20 with a step of 5. Once r_c and T are set, 5 experiments are done on all images of BSDS500 with $v_x = v_y \in \{27, 19, 16, 14, 12\}$. In each experiment, the average of the number of the generated superpixels and the average of under-segmentation error over 500 images are used to plot the figures

pixels may access the same cluster centers; therefore, the access to global memory may be saved since the access may hit the cache of texture.

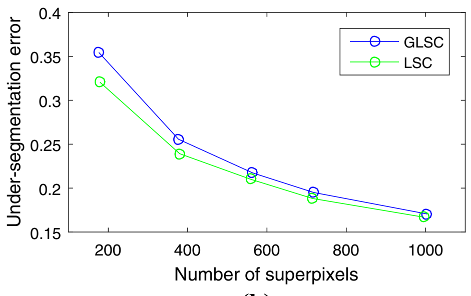
4 Experiments

In order to keep consistent with the original algorithm in terms of quality of the output, we have tested different values of GLSC parameters in order to optimize its results and ended up with a fixed set of parameters that are beneficial for all tested inputs. Our modified LSC algorithm is compared to the original LSC algorithm [11] in terms of the quality of the generated superpixels. The results show that there is no significant difference of quality between them. The execution time of our parallel implementation of the modified LSC algorithm and the original serial implementation[1] of the original LSC algorithm [11] provided by

[1] http://jschenthu.weebly.com/projects.html.



(a)



(b)

Fig. 11 The results of boundary recall and under-segmentation error for GLSC and LSC. In each subfigure, 5 experiments are done on all images of BSDS500 with $v_x = v_y \in \{27, 19, 16, 14, 12\}$. The average of the number of the generated superpixels and the average of each metric over 500 images are used to plot the figures. The average absolute difference of boundary recall and under-segmentation error are 0.0192 and 0.0136

the authors is also compared. Our implementation exposes important acceleration, reaching a speed up to 137 frames per second (FPS) making it suitable for real-time application on images of dimensions 480 $\times$ 320.

4.1 Quality of superpixels

Both the original LSC algorithm [11] and our modified LSC algorithm use two parameters C_s and C_c to control the relative significance of the color similarity and spatial proximity in measuring distance between pixels. In our experiments, C_c is set to 20, and C_s is defined as $r_c \times C_c$ for both the two algorithms. Generally speaking, larger r_c leads to superpixel map with higher regular shape.

Experiments are performed on the Berkeley Segmentation Data Set 500 (BSDS500) [2]. There are 500 481 $\times$ 321 or 321 $\times$ 481 color images. Each image in BSDS500 provides at least four ground truth segmentations which are obtained from different persons. The effect of parameters for our modified LSC algorithm is tested firstly. To evaluate the quality of superpixels, the boundary adherence of superpixels generated by our modified LSC algorithm and the original LSC algorithm [11] is compared using two commonly used evaluation metrics in superpixel segmentation methods: boundary recall (BR) [10] and under-segmentation error (UE). We also present visual comparison.

Boundary recall represents the fraction of ground truth boundary pixels correctly detected by the superpixel algorithm. A true boundary pixel is considered as being correctly detected if it falls within 2 pixels from at least one superpixel boundary pixel. This measure ranges from 0 to 1. A high BR indicates that few ground truth boundaries are missed. Conversely, a low BR implies that the segmentation results are poor because too many natural boundaries are missed.

In our modified LSC algorithm, there are three key factors, the spatial ratio r_c, the number of iterations T and the initial superpixel size or the desired number of superpixels that can affect the results of boundary recall. Experiments are conducted to evaluate how these factors affect the quality. The results are plotted in Fig. 9. With the increase of r_c, the importance of the number of iterations T also increases. When r_c is small, such as Fig. 9a, b, there is no obvious difference among 5, 10, 15 and 20 iterations visually. When r_c grows, the difference between different iteration number becomes large. In other words, the more the relative importance of spatial distance is, the more the number of iterations needs to get the algorithm converged. Since the number of iterations significantly affect the execution time of our implementation, r_c and T should be carefully considered when user applies our implementation to their situation.

UE is another measure of the quality of the generated superpixels. Given a region from the ground truth segmentation g_p and a group of superpixels required to cover it, $|s_k \cap g_p| > B_k$, UE measures how many pixels "leak" across the boundary of g_p from s_k. In our experiments, B_k is set to $0.05|s_k|$. To compute UE for one image with one ground truth, the formula of UE is defined in Eq. (6). Lower UE implies that fewer superpixels cross multiple objects.

$$\mathrm{UE} = \frac{1}{N}\left[\sum_{p=1}^{M}\left(\sum_{|s_k \cap g_p| > B_k} |s_k|\right) - N\right], \tag{6}$$

The results of UE are plotted in Fig. 10. With the increase of r_c, the influence of iteration number T becomes more and more obvious but less obvious than boundary recall (see Figs. 9h, 10h).

According to our analysis above, we set $r_c = 0.1$ and $T = 5$ for our modified LSC algorithm and keep the default parameters (in which $r_c = 0.075$ and $T = 20$) of the original LSC algorithm when comparing them using boundary recall and under-segmentation error. As shown in Fig. 11, LSC has a better superpixel quality. The measured average

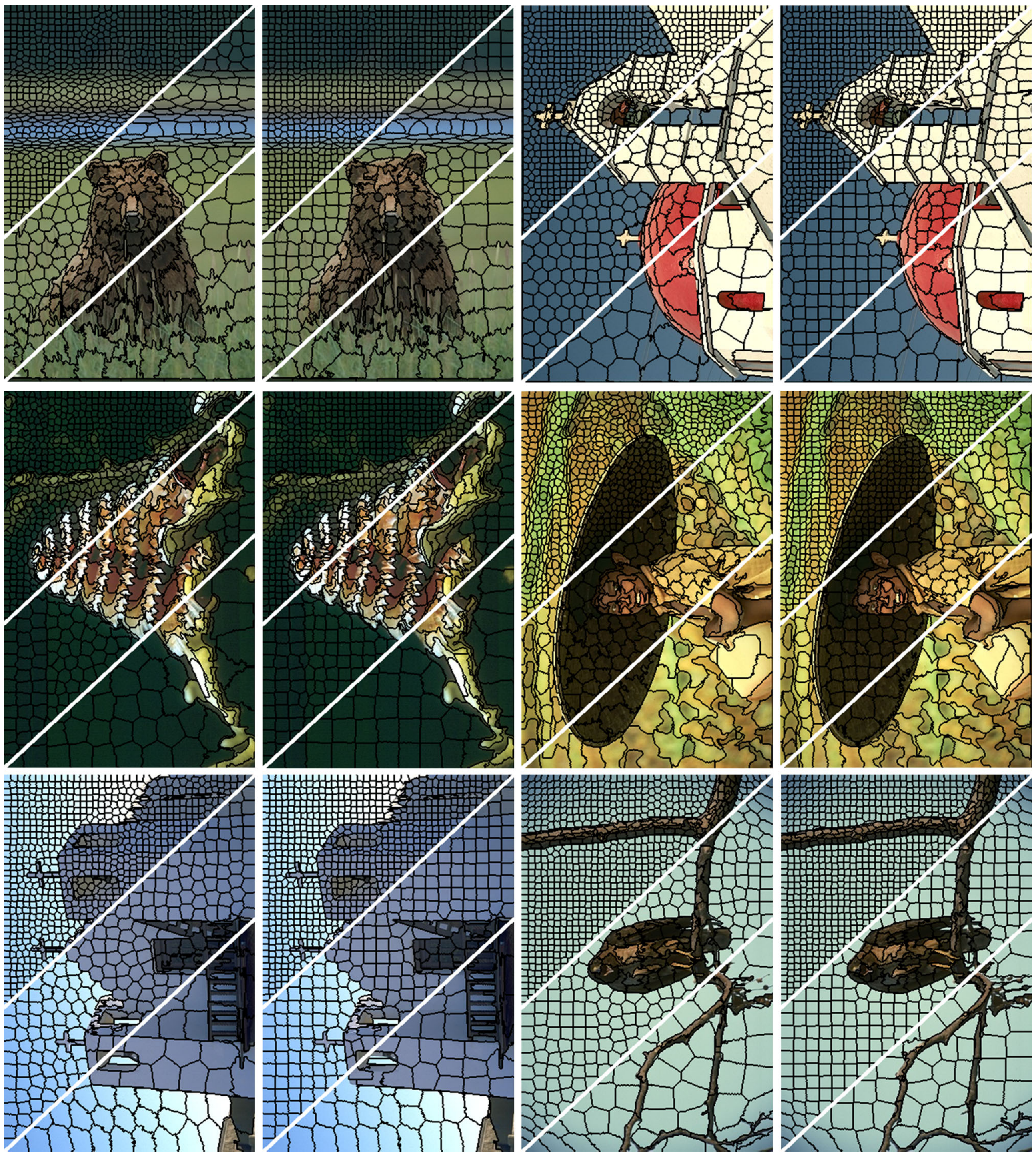


Fig. 12 Visual comparison. In each pair of images, the results of original LSC are on the *left* and the results of GLSC are plotted on the *right*

Table 1 CUDA capable devices used in experiments

Device model	CUDA cores	Memory bandwidth (GB/s)	GPU clock (MHz)
GTX 650	384	82.4	1084.5
GTX 960	1024	107.0	1273.4
GTX 760	1152	183.4	1123.0
GTX TITAN X	3072	320.9	1050.8

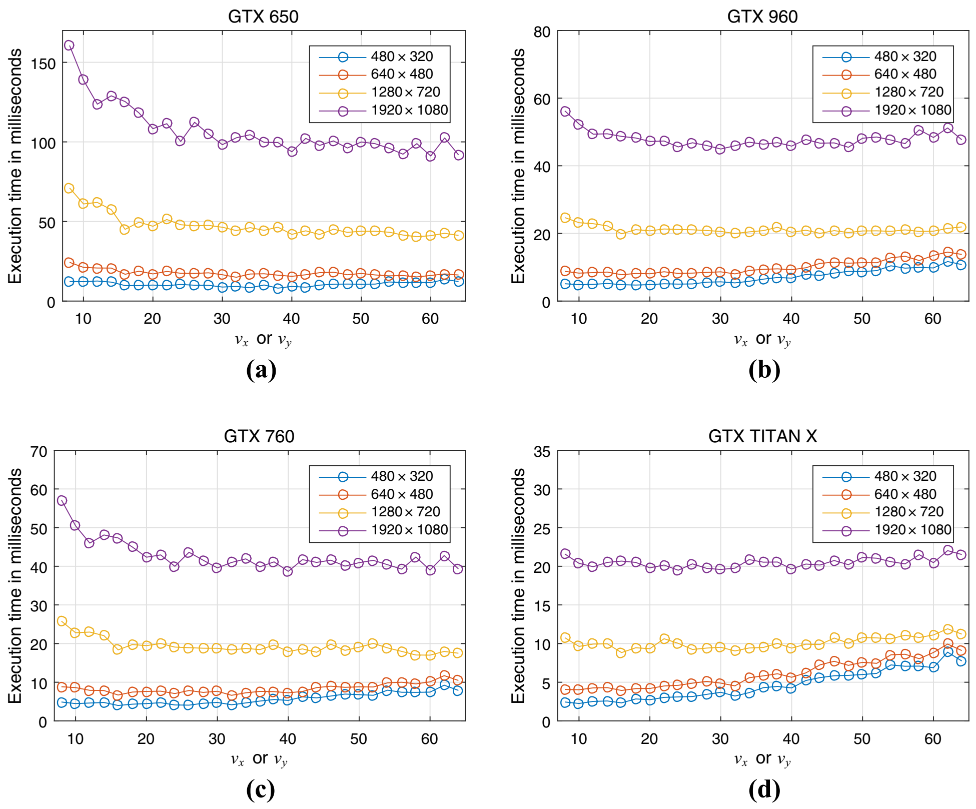


Fig. 13 Effect of v_x on four GPUs. The execution time is stable on GTX 650 when image size is small. When image size is big, the smaller the size of superpixel is, the more the execution time needs on lower performance GPU such as GTX 650 (**a**), GTX 960 (**b**) and GTX 760 (**c**). When image size is small, the execution time is not quite stable on higher performance GPU such as GTX TITAN X (**d**)

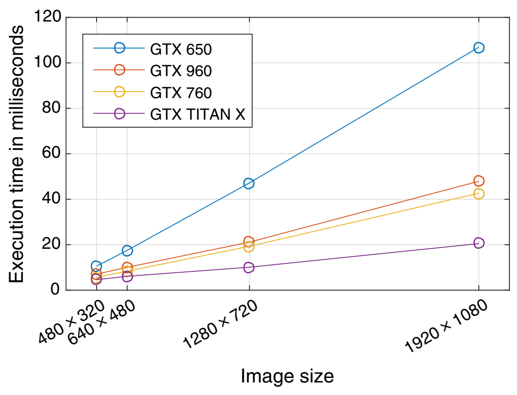


Fig. 14 Image size versus execution time. It shows that with the increase of image size, the execution time grows near linearly on the four GPUs

absolute difference of boundary recall and under-segmentation error are 0.0192 and 0.0136. With the increase of the number of superpixels, the qualities of GLSC and LSC [11] tend to approach each other. Visual comparison are plotted in Fig. 12.

4.2 Execution time

Since LSC superpixel algorithm [11] is of a linear complexity $O(|V|)$, the execution time of our implementation is directly influenced by the number of pixels in the input image. As mentioned in Sect. 3.2, the desired number of superpixels K or v_x/v_y can also influence the execution time of our parallel implementation. To show the effects

Table 2 Execution time of our implementation on GPU and execution time of the original implementation provided by the authors

Image size	Execution time (ms)					Speedup			
	GTX 650	GTX 960	GTX 760	GTX TITAN X	I5-4590	GTX 650	GTX 960	GTX 760	GTX TITAN X
480 × 320	10.60	7.10	5.70	4.60	108.70	10.3	15.3	19.1	23.6
640 × 480	17.50	10.1	8.30	6.10	206.40	11.8	20.4	24.9	33.8
1280 × 720	47.20	21.1	19.3	10.1	699.00	14.8	33.1	36.2	69.2
1920 × 1080	106.9	47.9	42.6	20.5	1637.6	15.3	34.2	38.4	79.9

Speedups are measured by our GPU implementation dividing the original version running on I5-4590

Table 3 Execution speed measured using FPS with post-processing added

Image size	Execution time (ms)					Speedup			
	GTX 650	GTX 960	GTX 760	GTX TITAN X	I5-4590	GTX 650	GTX 960	GTX 760	GTX TITAN X
480 × 320	75.2	102.0	119.0	137.0	9.0	8.4	11.4	13.3	15.3
640 × 480	44.6	66.70	75.80	90.90	4.7	9.4	14.1	16.0	19.2
1280 × 720	16.3	28.50	30.00	41.50	1.4	11.7	20.3	21.4	29.9
1920 × 1080	7.00	12.00	12.80	17.90	0.6	11.8	20.1	21.4	29.9

the two factors on runtime of our parallel implementation, it is tested on four CUDA devices: GTX 650, GTX 960, GTX 760 and GTX TITAN X. Configuration of these GPUs is listed in Table 1. The original implementation (see Footnote 1) is tested on an Intel Core i5 4590 3.3 GHz CPU to compute the speedups compared with the four GPUs.

We use four images with different sizes (480 × 320, 640 × 480, 1280 × 720 and 1920 × 1280) to test execution time of the modified part of the original LSC algorithm. v_x varies from 8 to 64 with a step of 2, and we set $v_y = v_x$. Experiments are performed on the four GPUs, and the results are plotted in Fig. 13. Although execution time is not quite stable with the variation of superpixel size, the average execution time is near linear with respect to image size as shown in Fig. 14.

Since we use the same method to perform the post-processing step and no algorithm modification has been made for this step, the execution time of our implementation is measured without the post-processing step when evaluating the performance of our implementation of the modified part. Devices on our test platform use independent memory system; therefore, the time of data transfer between CPU memory and GPU memory is part of the measurement. To evaluate the improvement on execution speed of our parallel implementation, the average execution time of each image size and their speedups are summarized in Table 2. As stated before, the time of post-processing is not included in this test but we add the time of data transfer. It turns out that LSC [11] is suitable for massively parallel architectures and exhibits notable speedup. In Table 2, it is shown that the GPU version is processing an input in less than 6ms for the faster device and it can achieve almost 80× performance boost related to the original version running on CPU I5-4590.

Our modified part of the LSC algorithm plus data transfer plus the post-processing step constructs a complete superpixel algorithm. It is necessary to evaluate the performance of the full implementation. As shown in Table 3, the execution time of post-processing step which runs on the same CPU is added and the speed is measured in FPS. If 30 FPS is real time for a specific application, our implementation can run in real time for image resolution 480 × 320 and 640 × 480. Only higher performance GPU can run our implementation in real time for image with resolution 1280 × 720. For image resolution 1920 × 1080, the test devices cannot run the implementation in real time. Our implementation can be found publicly.[2]

5 Conclusion

The superpixel algorithm LSC [11] was modified to make it suitable for massively parallel architecture such as GPU. The modified LSC has been implemented using CUDA (GLSC), and a series of experiments have been conducted to evaluate its performance. The modified algorithm was evaluated to see the effects of r_c and T for different initial superpixel size. It turns out that $r_c = 0.1$ and $T = 5$ is enough to get a better superpixel quality. GLSC and the original LSC algorithm were compared using two standard metrics, boundary recall and under-segmentation error. Execution time between our implementation of GLSC and the original implementation of LSC [11] provided by the authors has also been compared. Even though the execution time can be influenced by the initial superpixel size, it is near linear with respect to image size. Our implementation can run almost 80× faster than the serial implementation of LSC when the post-processing step was not counted. Moreover, we have demonstrated that the rather simple implementation of GLSC on commercially available GPU platforms such as NVIDIA GTX TITAN X can run at over 130 FPS for images with size 480 × 320 pixels.

References

1. Achanta, R., Shaji, A., Smith, K., Lucchi, A., Fua, P., Süsstrunk, S.: Slic superpixels compared to state-of-the-art superpixel

[2] https://1drv.ms/u/s!ApN11A41BjdggQMr1jhCu6pfmc3D.

methods. IEEE Trans. Pattern Anal. Mach. Intell. **34**(11), 2274–2282 (2012)
2. Arbelaez, P., Maire, M., Fowlkes, C., Malik, J.: Contour detection and hierarchical image segmentation. IEEE Trans. Pattern Anal. Mach. Intell. **33**(5), 898–916 (2011)
3. Van den Bergh, M., Boix, X., Roig, G., Van Gool, L.: Seeds: superpixels extracted via energy-driven sampling. Int. J. Comput. Vis. **111**(3), 298–314 (2015)
4. Felzenszwalb, P., Huttenlocher, D.: Efficient graph-based image segmentation. Int. J. Comput. Vis. **59**(2), 167–181 (2004)
5. Garcia-Garcia, A., Orts-Escolano, S., Garcia-Rodriguez, J., Cazorla, M.: Interactive 3D object recognition pipeline on mobile GPGPU computing platforms using low-cost RGB-D sensors. J. Real Time Image Process. (2016). doi:10.1007/s11554-016-0607-x
6. Gong, C., Tao, D., Liu, W., Maybank, S.J., Fang, M., Fu, K., Yang, J.: Saliency propagation from simple to difficult. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2531–2539 (2015)
7. Guler, P., Deniz, E.: Real-time multi-camera video analytics system on GPU. J. Real Time Image Process. **11**(3), 457–472 (2016). doi:10.1007/s11554-013-0337-2
8. Harris, M.: Optimizing Parallel Reduction in CUDA. NVIDIA Developer Technology (2007)
9. Kesavan, Y., Ramanan, A.: One-pass clustering superpixels. In: 2014 7th International Conference on Information and Automation for Sustainability (ICIAfS), pp. 1–5. IEEE (2014)
10. Levinshtein, A., Stere, A., Kutulakos, K.N., Fleet, D.J., Dickinson, S.J., Siddiqi, K.: Turbopixels: fast superpixels using geometric flows. IEEE Trans. Pattern Anal. Mach. Intell. **31**(12), 2290–2297 (2009)
11. Li, Z., Chen, J.: Superpixel segmentation using linear spectral clustering. In: 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 1356–1363 (2015)
12. Liu, X., Xu, Q., Ma, J., Jin, H., Zhang, Y.: MsLRR: a unified multiscale low-rank representation for image segmentation. IEEE Trans. Image Process. **23**(5), 2159–2167 (2014)
13. Nguyen, T.V., Lu, C., Sepulveda, J., Yan, S.: Adaptive nonparametric image parsing. IEEE Trans. Circuits Syst. Video Technol. **25**(10), 1565–1575 (2015)
14. NVIDIA Corporation: NVIDIA CUDA Compute Unified Device Architecture Programming Guide. NVIDIA Corporation (2007)
15. Ren, C.Y., Prisacariu, V.A., Reid, I.D.: gSLICr: SLIC superpixels at over 250Hz. ArXiv e-prints (2015)
16. Ren, X., Malik, J.: Learning a classification model for segmentation. In: Proceedings. Ninth IEEE International Conference on Computer Vision, 2003, pp. 10–17 (2003)
17. Sun, X., Shang, K., Ming, D., Tian, J., Ma, J.: A biologically-inspired framework for contour detection using superpixel-based candidates and hierarchical visual cues. Sensors **15**(10), 26654–26674 (2015)
18. Wang, S., Lu, H., Yang, F., Yang, M.H.: Superpixel tracking. In: 2011 IEEE International Conference on Computer Vision (ICCV), pp. 1323–1330 (2011)

Zhihua Ban received the B.S. degree from China University of Petroleum, Qingdao, China, in 2012. He is currently pursuing the Ph.D. degree with the State Key Lab for Multispectral Information Processing Technology, School of Automation, Huazhong University of Science and Technology. His research interests are in the areas of general purpose computations on GPU, image processing.

Jianguo Liu received his B.S. degree in mathematics from the Wuhan University of Technology in 1982 and M.S. degree in computer science from the Huazhong University of Science and technology in 1984, and Ph.D. degree in electrical and electronic engineering from the University of Hong Kong in 1996, respectively. He was a visiting scholar with the medical image processing group of the department of radiology at the University of Pennsylvania from December, 1998 to July, 2004. He is currently a professor of School of Automation at Huazhong University of Science and Technology in China. His interests include signal processing, image processing, parallel algorithm and structure, and pattern recognition.

Jeremy Fouriaux received his M.Sc. degree from RENNES University in 2005. He is founder of a startup: H4E computing, Hongkong, specialized in GPGPU, compilation and software engineering. His research interests include general purpose computations on GPU and compilation applied to image processing, computer vision and scientific computing.