A Work-Optimal Parallel Connected-Component Labeling Algorithm for 2D-Image-Data using Pre-Contouring

Henning Wenke, Sascha Kolodzey, Oliver Vornberger
University of Osnabrueck, Germany, 49069 Osnabrueck
Email:hewenke@uos.de, skolodze@uos.de, oliver@uos.de

Abstract—Connected-component labeling (CCL) is a well-known problem with many applications, e.g. in image processing. In this paper, we describe a parallel algorithm to solve 2d-image-data CCL-problems resulting in linear overall work. It can be classified as an one-pass algorithm, since no temporary labels are required. Our algorithm initially extracts the connected components’ independent contour-segments. Then, these are unified and labeled. Finally, the image is filled in order to label all non-contour pixels. Our approach is motivated by the observation that a line has, independent of its complexity, exactly two ends. Thus, two independently extracted lines (e.g. due to the parallel process) can be unified by only adjusting their ends, resulting in constant costs for this operation. Additionally, all contours are extracted as directed cyclic linked-lists, where the in and out degree of each node is always one. This topology is a simpler to deal with when compared to the original pixel grid.

I. INTRODUCTION

One very common method in image processing is Connected Component Labeling (CCL). The CCL procedure assigns a unique label to each set of connected pixels in an image. In this paper, we will limit ourselves to 2d-rectilinear image data. There, a pixel’s neighborhood can be defined in two ways: 4-connected or 8-connected [1]. Important applications of CCL algorithms are pattern recognition, computer vision and image processing. This includes, for example, character recognition [2], [3]. In addition, according to [4], the object’s contours are also helpful for 2d-object recognition in many cases. Given an image consisting of n pixels, a CCL-algorithm is considered optimal if it executes in O(n) time. In general, considering the immediate vicinity of a pixel is not sufficient to determine global labels. Therefore, most approaches assign temporary labels also known as provisional labels. Hernandez-Belmonte [5] and He [6] identify three classes of CCL-algorithms: The first category is multi-pass algorithms, like [7]. Because the number of passes depends on the image’s content, variants of this approach are typically not optimal. In contrast, two-pass algorithms make two distinct passes through the image. In a first pass, the image is scanned to determine equality of neighboring pixels, assign provisional labels and record the related equivalence information. Then, label equivalence information can be analyzed in order to determine the final labels. At last, a second pass assigns the final labels to the corresponding pixels. A theoretically optimal example of this class is the union-find algorithm [8]. The final category are one-pass algorithms, which go once through the image. Examples are algorithms based on contour-tracing, like [9] and its linear-time successor [4]. They scan a binary image from top to bottom and from left to right. Each time a contour is encountered in this process, a new label is generated. Then, the full contour is traced and the new label is applied to all the contour’s points. Afterwards, pixels within a contour are labeled equal to the contour, using a scan-line algorithm. In case of unclassified regions completely surrounded by a connected-region, an inner-contour is generated and receives the same label as the corresponding outer contour. Thus, the scan-line algorithm can label pixels between the inner-contour and outer-contour using the outer-contour’s label, while pixels within the inner-contour remain unlabeled. In the early 1990s, there was an interest in parallelizing CCL-algorithms to use them in image analysis applications. [10] considered CCL-algorithms to belong to the most time consuming algorithms used in pattern recognition. Recently, in line with the growing utilization of graphics processing units (GPUs) for general computation, again some attempts have been made to parallelize CCL-algorithms. Hawick [11] implemented a union-find CCL-algorithm for image data in CUDA. Their implementation was faster than an optimized CPU implementation for most of the tested instances. A drawback of their approach was the inability to guarantee that a single equivalence-tree is constructed for each connected component (CC) in the first pass, due to the parallelism. To solve this problem, the first pass has to be repeated several times, until all trees belonging to a single component are merged into a single one. The exact number of necessary passes is data-dependent and thus not known in advance, so after each iteration the labels’ correctness has to be checked. At last, Stava et al. [12] have further enhanced this approach and optimized it for the Nvidia Fermi architecture. They noted a significant speedup when compared to Hawick.

Our contribution: We present a novel deterministic CCL-algorithm for a PRAM. As far as we know, there is no parallel contouring based CCL-algorithm in literature. Our algorithm presented in this paper preserves all properties of an optimal line-following CCL-algorithm like [4]. Plus, it is parallel and as such delivers a better asymptotic running-time. Furthermore, we have not found a CCL-algorithm in literature, which is both parallel and has an optimal asymptotic total complexity on a CRCW-PRAM. At last, we compare an early OpenCL implementation of our algorithm on a GPU to results of Stava [12] in practice.
II. ALGORITHM

A. Overview

We pursue a parallel one-pass approach to extract the CCs' inner and outer contours in a first step. Therefore, contour-line-segments (CS) are extracted in parallel. These segments are subsequently unified, using a data-dependent or a data-independent conflict-free parallelism pattern. During the unification process, necessary information to distinct inner- from outer contours is aggregated and stored in the surviving elements. Then, surviving segments belonging to an outer contour receive a label. Afterwards, the lines' unification is reversed, using a scheme exactly reverse to their prior unification. In this process the surviving CSs' labels are passed to each splitted CS. So, all segments of one contour will receive this label. Finally, the image is filled in order to label all non-contour pixels, using a column-wise parallel and stack based scan-line approach. Our approach shares some properties, e.g. of the resulting lines, with the work of [4]. However, we do not classify our algorithm as line-following, since parallel extracted CS are never followed but merged in parallel to a complete contour.

B. Properties of the used contours

Our first sub-algorithm extracts all CSs of all CCs for all pixels in parallel. In order to enable subsequent sub-algorithms to unify these to complete contours, distinct inner from outer contours, label and finally fill them, several properties are required:

- A contour is always closed by definition.
- A contour (primarily) consists of the contained CC's outer pixels' outer edges.
- All properties of a CS can be computed based on its pixel's 8-way neighborhood.
- Each CS has one and only one unique successor and is predecessor to exactly one CS.
- The definition above results in distinct rotational directions for inner and outer contours.
- A pixel may contain up to 4 CS belonging to one or more contours. The final fill-algorithm decides within a pixel if a CC is entered and/or left. The in-out state-change must be recognized based on the entirety of a pixel's CSs' properties.

As a start, we describe one possible way to model the CS, which fits the specifications above. If the classification of one or more pixels of a pixel's 4-connected neighborhood differs from its own, one or more CSs for this pixel need to be extracted. The exact configuration of these extracted segments additionally depends on the pixel's 8-connected neighborhood. There are four potential edge-parallel-CS-parts, which are parallel to one of the pixel's left, right, upper or lower edges, respectively. One of these parts is generated if the pixel adjacent to the respective edge is classified differently. This is demonstrated left in figure 1. Light gray shaded squares indicate an identical classification or segmentation. Dark gray shaded pixels are not classified and therefore not part of any CC. Please note that this coloring scheme remains true for the rest of this paper. In addition to edge-parallel-CS-parts link-CS-parts are needed, if two cyclic consecutive pixels of a pixel's 4-connected neighborhood share the classification with the central pixel under consideration and the pixel of the 8-neighborhood, which resides between the other two neighbors, has a different classification. As an illustration, see right part of figure 1. Link-CS-parts consist of a horizontal and a vertical half, which can be thought of as an extension of the neighboring pixel's edge-parallel-CS-parts, respectively. In order to unify individual CS, a unique successor has to be determined for every single CS. This has to be done in a parallel fashion, thus this operation is restricted to rely on local data. Thus, for each possible CS-part type a direction is defined as follows:

- Upper segments are directed left
- Lower segments are directed right
- Left segments are directed down
- Right segments are directed up

A unified CS within a pixel, consisting of edge-parallel-CS-parts and link-CS-parts, can be identified based on local data and without need of synchronization with other CS. Thus, the algorithm extracts unified CS within a pixel directly. Only these are considered in the remaining paper. All possible combinations of four succeeding and four previous directions result in 16 cases, which need to be distinguished, as shown in figure 2. Additionally, up to four of these CSs can appear within the same pixel. All combinations are possible where different CSs do not have a CS-part in common and do not cross each other. From this follows that at most one segment

![Fig. 1](image1.png)  ![Fig. 2](image2.png)
can head to or come from the right, left and up and down pixel, respectively. Thus, it is possible to limit our further considerations to four types of CSs, based on the leaving-direction of their pixel. They are hence simply called right, left, down and up. Additional information, such as their course within their associated pixel can be stored as a property of a CS. However, one special case does not fit in this scheme. If a CC consists of a single pixel, no CS leaves the pixel. This results in an undefined leaving direction. Since this situation can obviously be identified locally, no CS is extracted and the pixel can directly be labeled instead. Some possible combinations are shown in fig. 3. Notably, the right example shows the special case, where all four CSs exist. At last, inner and outer contours need to be dealt with in a different fashion. In case of one or multiple unclassified or differently classified areas enclosed completely by a CC, one or several inner contours occur (see figure 4). Since these contours are independent from each other, they would receive different provisional labels. Thus, finding a distinct label for each associated pixel is difficult. So, we chose an approach based on the observation, that each CC has exactly one outer contour and zero, one or multiple inner contours. Obviously, once inner contours are classified as such and removed, the remaining outer contours can directly receive their final label. A line-following-approach, like [4], be identified locally, no CS is extracted and the pixel can directly be labeled instead. Some possible combinations are shown in fig. 3. Notably, the right example shows the special case, where all four CSs exist. At last, inner and outer contours need to be dealt with in a different fashion. In case of one or multiple unclassified or differently classified areas enclosed completely by a CC, one or several inner contours occur (see figure 4). Since these contours are independent from each other, they would receive different provisional labels. Thus, finding a distinct label for each associated pixel is difficult. So, we chose an approach based on the observation, that each CC has exactly one outer contour and zero, one or multiple inner contours. Obviously, once inner contours are classified as such and removed, the remaining outer contours can directly receive their final label. A line-following-approach, like [4], whose computation-scheme begins at the image’s border and proceeds sequentially, can identify outer contours by simply finding them when firstly entering a new CC. Obviously, in case of a parallel approach, as desired here, this method of classification is not an option. Instead, a possible solution suitable for parallel execution may use the contour’s rotational direction. The CSs’ direction definition implies a counter-clockwise rotational direction in case of inner contours and a clockwise rotational direction in case of outer contours. As an illustration, see figure 4 (right).

C. Extraction of contour-segments (CS)

As outlined above, a CS is of type left, right, up or down, depending on the pixel, which contains the next CS in the defined contour-direction. For pixel p, a CS of one direction-type is created, if p shares the classification with the next pixel in the direction. Additionally, the next two pixels of the 8-neighborhood of pixel p starting after this neighbor-pixel and proceeding clockwise, need to be analyzed. Only if one of these has a different classification, the condition is completely met. Figure 5 shows all cases resulting in the generation of a CS of type right, for example. Other direction-types arise from rotationally-symmetric considerations. Some calculations require closed contours to operate. Following the above described procedure, CCs, which are in touch with an image’s edge, would not receive a closed contour. Thus, these contours are closed by force. In order to do that, imaginary neighbors beyond the image’s edges are considered. They are unclassified per definition. In the algorithm described in this paper, each CS has to be unified with its successor, which is contained in the pixel of the 4-connected neighborhood, matching its directional-type. This, for example, is the pixel residing left of its containing pixel if a CS’s directional-type is left (see. figure 6). For this neighbor-pixel, the suitable of

![Fig. 3. Some combinations of CSs within a single pixel. Numbers: ioCnt (v)](image)

![Fig. 4. Example of a CC containing enclosed unclassified areas. One outer contour and two inner contours are extracted.](image)

![Fig. 5. All cases of a CS of direction-type right in pixel p being created](image)

![Fig. 6. CS of type right seeking a successor. Chosen is the first existing CS in a CCW-manner starting from down-direction.](image)
Algorithm 1: ExtractCS

Global Data: Pixel[] pa, CS[] ca;
pixelCnt n, image-width a, image-height b;
Precondition: p.class ∀ p of pa set;
for each Pixel p of pa in parallel do
  if p.class ≠ UNCLASSIFIED then
    for each Dir d ∈ {right, left, down, up} do
      if Condition for CS c in d fulfilled then
        c.status ← EXISTING;
        c.props ← calcProps(p, c);
        c.ioCnt ← calcIoCnt(p, c);
        c.head ← c; c.tail ← c;
        c.suc ← calcSuccessor();
      else
        c.status ← NOT_EXISTING;
      endif
    end
    storeCS(p, d, c);
  endif
end

Function unifyOp(CS c)
if c.status = EXISTING then
  CS cn ← c.suc;
  if c ≠ cn.head then
    c.tail.props ← aggregateInfo(c.tail.props,
      cn.props);
    c.tail.head ← cn.head;
    cn.head.tail ← c.tail;
  else if checkIfOuterContour(cn.props) then
    cn.label ← get1dAddress(cn);
  endif
endif

As only the CS-strips’ ends are of interest for the unification operation, we call this operation simply the unification of two CSs for the rest of this paper. From this follows also that two CS-strips’ unification results, independently from their complexity, in constant costs. The parallel unification of multiple CSs is possible, but one restriction has to be fulfilled: The unification of a CS cs0 with its successor cs1 must not take place simultaneously to the unification of cs1 with its successor cs2. If so, the correct assignment of references to the resulting CS’s head and tail cannot be guaranteed. In order to resolve potential synchronization problems, a data-independent scheme, processing as many CSs in parallel as are well-known in every step without any data-analysis is applied. In order to explain that, at first we define a tile as follows:

- A tile represents a rectangular pixel-region.
- All CSs within it, that can be unified, are unified.
- All CSs crossing one or more of its edges are not unified.
- Two tiles that share one edge can be unified and the result is one tile.
- Initially, after the extractCS sub-algorithm is executed, each pixel is a tile.

Then, the tile-merging sub-algorithm called unify is applied (see algorithm 2). Additionally, figure 7 shows all iterations of this pattern applied to a 4x4 pixel-grid. The algorithm should be read together with the descriptions below. There, in every iteration tiles are defined, which contain CSs, that can be processed independent from any CS contained in any other
tile. Thus, these tiles can be processed conflict-free in parallel. Tiles are defined in a way that in even iterations horizontal CS (directional-types right/left) and in uneven iterations vertical CS (directional-types up/down) within tiles associated with the iteration are to be processed. In the first iteration the pixel-grid is divided into tiles of size $2 \times 1$ pixel sized tiles, where CSs within every two adjoined pixels are united. Once the first iteration is accomplished, the CSs of the left pixels with direction-type right and the CSs of the right pixels with direction-type left of all tiles from the first iteration have been unified with their successors. As the pattern is alternating in the second iteration, a vertical unification is performed. Here every pair of tiles from the previous iteration lying on top of each other are unified. Obviously, all tiles of the second iteration have a size of $2 \times 2$ pixels. There are at most two CSs of directional-type down within the upper pixels and similarly two CSs of directional-type up within the lower pixels of each tile to be united with their respective successors, if these are existing. In the third iteration CSs of every pair of tiles of the previous iteration laying side by side are united. Hence, tiles associated with the third iteration are $4 \times 2$ pixel in size, and so on. Finally, we get one tile occupying all pixels. Since none of the CSs crossing one iteration’s tiles borders have been united with their successors in iterations prior to this one, tiles of each iteration can be processed independently in parallel. Though the processing of tiles associated to one iteration can be done in parallel, the processing of CSs within one tile may lead to synchronization problems. For the remaining part of the paper, let direction-types right and down be called forth-directions and direction-types up and left be called back-directions. In case all CSs of direction-type forth of all tiles within an iteration are processed before all of the CSs of direction-type back are processed, no conflicts can occur during forth-CSs' processing. Consider the scenario of the unification of CS-strips with CSs of types back- and forth alternating along the separating edge between both tiles (see figure 8). Since all CSs of type back are not unified until the unification of all CSs of type forth is finished, the simultaneous unification of three directly succeeding CSs is impossible during the forth-CSs' processing phase. As an illustration, see left part of figure 8. Thus, in each iteration, all CSs of direction-type forth can be unified in parallel if the CSs of direction-type back are not unified yet. However, the following unification of each tile’s CSs of direction-type back cannot be done in parallel, as all CSs of direction-type forth associated with the current iteration already have been unified. Thus, possibly three CSs of type back, belonging to three directly subsequent CS-strips are to be unified in parallel within one tile. As an illustration, see figure 8 (right). The shown situation here results in a conflict. The postulated static pattern resolves these conflicts by sequential processing all CSs of type back per tile. The parallelism-scheme is described in pseudocode representing the unify sub-algorithm contained in the pseudocode-snippet of sub-algorithm 2.

Until now, exactly one CS per complete outer contour has received a label. These need to be passed subsequently to all other CSs of their corresponding contours. In order to achieve that, a parallelism-pattern exactly inverse to the one used in conjunction with the unify sub-algorithm is applied and in this process the surviving CS’s label is passed on. The associated function passLabelOp is described in the pseudocode-snippet below. It is called instead of function unifyOp.

```plaintext
Function passLabelOp(CS c)
    if isLabeled(c.tail) then
        c.suc.label ← c.tail.label;
    endif
end
```

E. Contour-Labeling: Data-Dependent Approach

As an alternative to the data independent grid based contour-labeling approach, we want to introduce a data-dependent one, which enables a higher degree of parallelism. As a start, let’s take a look at the pseudo-code of algorithm 3. It is very similar to the basic parallel list-ranking algorithm using a pointer-jumping technique. Input are directed cyclic linked-lists, where each node is a CS, as extracted by algorithm 1. At first, it aggregates data of all nodes of a linked list and applies it to each node. Afterwards, all nodes of each linked list share the minimal memory address of all nodes of the corresponding linked list in the field minId. Then, the information aggregated in the field props can be used to decide if a CS belongs to an outer contour. If so, it receives the aggregated minimal memory address as a label. Unfortunately, this algorithm does $O(n \cdot \log_2(n))$ computations and is thus not work-optimal. Therefore, we apply a technique similar to Cole-Vishkin’s[Quelle] optimal parallel list-ranking algorithm [13]. It can be described at a high level by the following three steps[13]:

1) Given a list L of size $O(n)$, create a list L’ of length $O(n/\log_2(n))$ by iterative deletion of at least half of each

![Fig. 8. Conflicts due to parallel unification of 2 tiles’ CSs. Left: No conflicts possible in forth-direction. Right: Conflict in back-direction](image-url)
lists nodes within each step. The selection of appropriate nodes can be done with a to be computed 2-ruling set in each iteration.

2) Solve the list-ranking problem for L by application of the basic pointer jumping algorithm. This now takes O(n) operations.

3) Insert the nodes of L back into L, using a pattern exactly reverse to their prior deletion from L. Thereby compute the ranks of all nodes of L.

Cole-Vishkin’s parallel algorithm is able to solve the list-ranking problem in \( O(n/p) \) time using \( p \leq n/\log(n) \) processors, which is optimal. This scheme can be used for contour-labeling, too. All list-ranking specific computations need to be replaced by the operations given in algorithm 3. And in step 3, the found labels only need to be passed on. However, special care needs to be taken in case of small linked lists, which may be removed completely within step 1. If so, they will not reach step 2 and hence cannot be labeled. This is possible, since \( n \) represents the data-set’s size and not the linked-list’s length. To prevent that, whenever a node is to be removed, it is checked if it is its own successor. If that is the case, the node can be labeled directly as described in our data-independent approach.

As in Cole-Vishkin’s original algorithm, all changed computations always compare two nodes, resulting in constant costs for these operations. Thus, all theoretical properties of Cole-Vishkin’s algorithm remain true in case of our contour labeling algorithm.

### F. Fill CCs within contours

Finally, pixels belonging to a certain CC and contained within a contour are to be labeled. Therefore, a filling-algorithm, originating from two opposing edges of the image, is applied. Here, this is done by processing half-columns starting from the upper and lower image’s edges and proceeding sequentially to the center. These half-columns are independent from each other and can thus be dealt with in parallel. It is of importance to consider cases, where CCs are nested with one or multiple other CCs or unclassified regions. Regarding this, the algorithm described in this paper utilizes a stack-based fill-algorithm to manage labels of nested contours. The associated sub-algorithm 4 is described in the pseudocode-snippet below. Here, we assume a vertical processing direction, but a horizontal approach is also possible. Whenever a new pixel is entered, data from its associated four CSs needs to be gathered in case of their existence. This includes their label, which is guaranteed to be the same if more than one CS exists. Since the algorithm operates in a vertical direction, CCs can exclusively be entered or left by passing horizontal edge-parallel-CS-parts, which may be called io-CS-parts (io stands for in/out). Their total number needs to be determined for the current pixel, too. All possible courses of a single CS were already shown in figure 2. There, the numbers indicate the associated io-CS-part-counts (ioCnt for short). Obviously, it is 0, 1 or 2 in case of a single CS. Additionally, figure 3 (left) shows the only possible combination of CSs, where the ioCnt of more than one CS is nonzero. So, the total ioCnt of a pixel is also 0, 1 or 2. In case of a pixel containing exactly one io-CS-part, a CC is either entered or left. The proper operation can be identified by considering the label-stack. If the CS’s label is currently not on top of the stack, the CC is entered and thus the label has to be put on top of the stack. Otherwise, the CC is left and the stack’s topmost label is removed. In case of an ioCnt different from one, the stack remains unchanged, but in case of an existing CS-label it is applied to the pixel. If a pixel does not contain an io-CS-part but is classified, it receives the stack’s top label.

### G. Asymptotic analysis

Be \( n \) the number of pixels. We use the data-structures pixels and CS, consisting of a fixed number of fields each. And there are \( 4 \cdot n \) CS. Thus, our algorithm’s memory consumption is linear with respect to the number of pixels. Asymptotic numbers of computations and running-times are data-independent and depend on the pixel-grid resolution only. These are:

**Contour-segment extraction:** All pixels can be processed independently in parallel. Thereby, up to four CS per pixel are extracted, resulting in constant costs per pixel. So, using \( p \leq n \) processors delivers a running-time \( O(n/p) \) on a PRAM.

**Contour labeling:** Here, we only consider our data-dependent approach, since it delivers better asymptotic properties due to its higher degree of parallelism. As described before, it preserves all properties of Cole-Vishkin’s optimal list-ranking algorithm by which it was inspired: Given any number of \( p \leq n/\log(n) \) processors, all contours can be labeled in \( O(n/p) \) time on a CRWC-PRAM.

**Contour filling:** All columns can be filled independently in parallel but each single column is processed sequential. If there are more rows than columns, it is reasonable to process in a row-wise scheme instead. Let \( a \) be image-width and \( b \) image-height. Then, any number of \( p \leq \max(a, b) \) processors can be utilized, resulting in \( O(n/p) \) running-time on a PRAM.

Altogether, all sub-algorithms have a running-time guarantee of \( O(n/p) \), and the processor-number is limited by the contour-filling algorithm to \( p \leq \max(a, b) \). Our algorithm
is work-optimal as it obviously does not more than $O(n)$ operations for any number of processors utilizel.

III. IMPLEMENTATION DETAILS

We have implemented our algorithm utilizing OpenCL so it can be evaluated on a GPU or CPU. However, in this paper we limit our considerations to a GPU. The pseudo-code given before describes our implementation from a quite high level point of view. References are in most cases implemented as 1d memory-addresses. Others are only globally unique if the 2d-pixel-grid structure is taken into account, too. The stacks used by algorithm 4 are implemented as fixed-sized data-structures with a pointer to a current element. The 2-ruling sets necessary for Cole-Vishkin’s processing scheme are implemented similar to their earlier paper [14], because a $\log(\log(\log(n)))$-n ruling set already is a 2-ruling set for all earthly values of $n$. In order to implement our algorithm, dozens of distinct OpenCL-kernels are necessary. Especially Cole-Vishkin’s algorithm requires numerous global synchronization points. Since OpenCL does not allow global synchronization within a kernel, whenever such a point is reached, the kernel is terminated and another one started thereafter. Additionally, GPUs typically offer some sort of user managed cache called local memory (or shared memory, CUDA term). Its use is important in many cases to receive good memory access patterns. For example, Stava [12] identify their shared memory use as one important reason delivering the speedup when their algorithm is compared to Hawick[11]. However, in case of linked-lists it is less straightforward to use local memory as it is in case of an algorithm working directly on a rectangular pixel-grid.

Contour labeling is done using a hybrid approach consisting of the data-independent and the data-dependent approach described in this paper. The data-independent one has no overhead to identify CS to be processed in parallel and it is possible to achieve convenient memory-access patterns at least to a certain degree. Both is not true in case of the our implementation of the data-dependent pattern. In exchange, it delivers a much higher degree of parallelism. Thus, our contour labeling algorithm always starts with the data-independent pattern and uses it as long as the delivered parallelism is sufficiently high to utilize the executing device. Due to the fact that it is a data-independent pattern, a proper number of iterations can be calculated in advance. Then, we switch to the data-dependent pattern. Of course, on the way back, the last iterations must use the static pattern again, since the unification needs to be exactly reversed. This hybrid approach runs about 50 percent faster than the faster approach alone. The overall memory-consumption of our current implementation is 77 bytes per pixel.

IV. EXPERIMENTAL RESULTS

In order to get an idea how the running-time measured is composed of the CS-extraction, Contour labeling and Contour filling, the running times of the corresponding kernels are measured using OpenCL-events and aggregated to the three categories. In case of the 4096 x 4096 Spiral, we get the following results for our algorithm:

- CS-extraction: 21 percent of running-time measured
- Contour labeling: 61 percent of running-time measured
- Contour filling: 18 percent of running-time measured

Now, let’s take a look at data-dependence. In order to do that, we determine the throughput when processing a checkerboard-pattern, where the squares’ size can be configured. Here, four different classifications are applied to separate all squares as distinct CCs. In doing so, the impact of different numbers and sizes of CCs can be evaluated. Table 1 shows the results compared to Stava. Here, Stava’s algorithm delivers the highest throughput in case of the smallest CCs and it decreases as the

![Fig. 9. Comparison of throughput [GPixel/sec] to Stava using spiral-data in different resolutions](image-url)
CCs’ size increases. This behavior was already reported in their original paper. Contrasting these results, the throughput of our algorithm increases, if the CCs size increases. An exception to this behavior is the CC size one, where our algorithm does not extract any CS. So, our algorithm performs better, if the processed data consists of less contour-segments.

V. CONCLUSION

We have presented a parallel algorithm to solve 2d-image-data CCL problems in linear overall complexity. Similar to contour-tracing approaches, the extracted object-contours may also be helpful for certain applications like 2d-object recognition. Our current OpenCL-implementation performs comparable to the fastest previous approaches, when executed on a GPU, if a sufficiently high resolution is applied. Additionally, it scales superior with respect to the data-resolution in case of the tested data-set.

VI. FUTURE WORK

First and foremost, we will research alternatives to the fill contour sub-algorithm, since it limits the asymptotic running-time due to the small number of processors utilizable, when compared to our other sub-algorithms. Perhaps a per-column unification of linked-lists is a better idea. However, these would have to be re-arranged at first in order to solve problems regarding nested connected-components.

Besides, we will improve our implementation. One problem identified is the amount of OpenCL kernel calls (hundreds per processed image). And another one is the limited use of a GPU’s local memory provided by OpenCL.

REFERENCES