A Study on Connected Components Labeling algorithms using GPUs

Conference Paper · October 2010

CITATIONS
26

READS
1,744

2 authors:

Victor Oliveira
University of Campinas
3 PUBLICATIONS **58** CITATIONS
SEE PROFILE

Roberto de Alencar Lotufo
University of Campinas
215 PUBLICATIONS **4,491** CITATIONS
SEE PROFILE

A Study on Connected Components Labeling algorithms using GPUs

Victor M. A. Oliveira, Roberto A. Lotufo
Department of Computer Engineering and Industrial Automation
School of Electrical and Computer Engineering - UNICAMP
Campinas, Brazil
Email: victormatheus@gmail.com, lotufo@unicamp.br

***Abstract*—Connected Components Labeling (CCL) is a well-known problem with many applications in Image Processing. We propose in this article an optimized version of CCL for GPUs using GPGPU (General-Purpose Computing on Graphics Processing Units) techniques and the usual Union-Find algorithm to solve the CCL problem. We compare its performance with an efficient serial algorithm and with Label Equivalence, another method proposed in the literature which uses GPUs as well. Our algorithm presented a 5-10x speedup in relation to the serial algorithm and performed similar to Label Equivalence, but proved to be more predictable in the sense that its execution time is more image-independent.**

***Keywords*-Image processing; Connected Components Labeling; Connected Components Analysis; GPU; CUDA**

I. Introduction

Since the origin of Computer Science, Connected Components Labeling's algorithms have been used in several fields [1]. One of these fields is Image Processing [2].

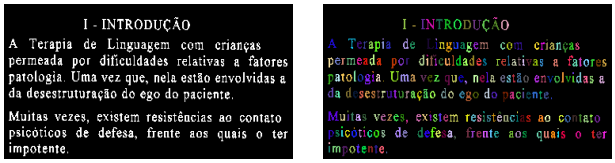
Connected Components Labeling (CCL) is a very important tool in Image Processing, Engineering, Physics, and others, therefore there have been many proposed algorithms and implementations. However, there are many factors like computer architecture and programming languages that can render a previously efficient algorithm inefficient, and vice versa.

This article's motivation is to understand how the use of GPUs (Graphical Processing Units) can improve the performance of complex Image Processing algorithms. We will proceed to explain, analyze and compare the performance of three CCL implementations:

- *Stephano-Bulgarelli's algorithm* [3], which is a very efficient serial algorithm, used here as a "control group" to be compared with GPU algorithms.
- An optimized version of the traditional Union-Find algorithm for GPUs.
- *Label Equivalence*, a hybrid version of Union-Find and Neighbor Propagation for GPUs based on [4].

As a running example of CCL, we are going to segmentate letters in an image with text. Figure 1 shows the result of this procedure. In the end, each letter is marked with a different label.

A motivation for this work is improving performance of some Image Processing algorithms in order to make Real-Time Video Processing more suitable in common machines.



(a) (b)

Figure 1. (a) Binary image with text. (b) Letters segmentation.

In Section II we explain some details about GPU architecture, Section III introduces some basic concepts about the CCL problem, in Section IV we explain our approach to the CCL problem in GPUs using Union-Find, Sections V and VI present other two parallel approaches to CCL problem, while experimental results are given in Section VIII. Finally, we come to a conclusion in Section IX.

II. GPUs

Graphical Processing Units (GPUs) are graphics-specialized hardware which can be used not only for Computer Graphics tasks (ex: rendering and mesh operations), but also for other applications. Obviously, this is possible only if we can adapt the problem to the massively parallel nature of GPUs. So, although feasible, it is not recommended to implement serial algorithms in GPUs, as the performance hit would be unacceptable.

With the development of GPUs, it became feasible to use them for tasks traditionally handled by CPUs, the set of techniques that allowed this is called General-Purpose Computing on Graphics Processing Units (GPGPU).

As said before, GPUs have a massively parallel architecture, which means some of the parallel algorithms in the literature can be easily adapted to it, specially the ones made for the Parallel Random Access Machine (PRAM) computational model [5].

In this work, an NVIDIA GPU is used with CUDA language [6]. The main objective of this article is not to deeply analyze their architecture, but the understanding of some of its basic aspects beforehand is necessary to fully comprehend this article:

- Kernel: A function which is executed in parallel in the GPU.
- Thread: Context (memory and registers) and execution of serial instructions in a GPU processor.
- Block: Group of threads that shares memory and has some hardware-based synchronization primitives. A GPU processor executes Blocks' Threads in parallel.
- Coalesced memory access: A phenomenon in which Threads in the same Block access contiguous positions in the memory. All memory transfers are done at the same time. This is vital for performance.
- Texture cache: Memory cache which can be used for speeding memory accesses and to force memory coalescing.
- Shared memory: Fast processor memory which is shared by all threads in a Block.

So, we can see GPUs use a hybrid hardware model, inside Blocks we have a SIMD architecture [7], while between Blocks it is impossible to ensure synchronization. Also, if two or more threads try to write at same memory position at the same time, at least one of them will be successful, but there is no guarantee about which one will be successful. Therefore it is possible to say that inside a Block the Arbitrary Asynchronous PRAM model is valid.

III. CCL Concepts

All algorithms here presented share the same structure in the sense that all of them are based in the idea of an image I that should be labeled, and an auxiliary structure L, with the same size of I, with *labels*. A label is a value in L which points to a pixel in I. All algorithms will have the task of trying to find the labels L such that a pair of pixels $p, q \in I$ are in the same connected component if and only if they have the same label in L [2].

A raster value of a pixel is an unidimensional index which locates it in the image. Labels will have the raster value of the pixel in I they point to.

The concept of *root* will be necessary in this article too, the definition of root used here is a pixel whose label points to itself. A connected component can be defined as all pixels in the image which have the same root. The root of a connected component C will be the pixel with smaller raster value in C.

IV. Union-Find

Our approach to the CCL problem is a parallel version of the usual Union-Find algorithm [8] optimized for GPUs and will be described as follows.

The Union-Find structure is used in this article in the same manner as for graph labeling algorithms; i.e., a node (pixel) verifies its neighbors and merges with them using the *merge* operation. In our case, it will be utilized with an adjacency relation (ex.: 4-Connectivity). When the algorithm is complete for all pixels, the connected components of the image graph will be encoded in the structure, in the sense that for every pair of pixels p and q they only belong to the same connected component if and only if they share a parent in the structure.

Also, it is easy to see that a common Union-Find algorithm will not fit well this architecture, because it may need to follow a long path until reaching the root of two pixels at the merge process, as Figure 2 reveals. A distant root means a long path in memory, which generates a heavy performance loss, because memory accesses are not coalesced. Also, it is a good idea to use as much Shared Memory as possible, thus increasing the process speed. So, our main concern must be organizing memory accesses in order to take advantage of the GPU's architecture. See Figure 4 for a running example.

A. Phase 0: Arrangements

We index each pixel by its raster value and divide the image in rectangular pieces of pixels. It will be considered that each piece will be assigned to a Block. Each Block will be given to a different GPU processor, in fact, there is a hardware Block's scheduler in NVIDIA GPUs, so it is not an unrealistic point of view. The start label of each pixel is its raster value as well.

B. Phase 1: Local merge

Each GPU processor receives its Block and makes the merge procedure in its shared memory, creating a Union-Find structure of the image piece assigned to the Block. In Algorithm 1 is presented a detailed description of this step. All operations are performed in Shared Memory, then the results are transfered to the global memory. It can be seen in Algorithm 2 there is a heavy use of Atomic Operations in the merge procedure. Basically, this means all operations that use the same memory position are serialized, so that race conditions and other issues that might happen in concurrency are avoided.

Atomic Operations are needed because merges are done in parallel. In Figure 3.a there is an example of concurrent merging, pixels a, b, and c have roots r, s, and t, with labels l_r, l_s, and l_t, respectively. Suppose $l_r < l_s < l_t$ and $merge(a, c)$ and $merge(c, b)$ are called in parallel. Depending on the order of execution of $merge(a, c)$ and $merge(c, b)$, l_t will be l_r or l_s. For example, if we have $l_r = 0, l_s = 1, l_t = 2$ and $merge(c, b)$ is done after $merge(a, c)$, the final value will be $l_t = l_s = 1$. Else, if $merge(a, c)$ is done after $merge(c, b)$, it will be $l_t = l_r = 0$.

That is exactly what we want to avoid, with $atomicMin$ the result will be l_r (the smaller label) in both cases, but there is a side effect. We can see in Figure 3.b that pixel s was not linked to its right root, that is why there is a loop in Algorithm 2 that stops only when labels are correctly assigned for both input pixels (Figure 3.c).

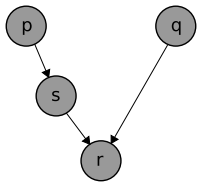


Figure 2. Pixel p and pixel q share a parent r, so they belong to the same connected component, but in order to discover this, it is required to solve p's label chain and bypass s.

C. Phase 2: Global merge

In this step all partially Union-Find structures of the image are merged in the same way as in last section. These structures were stored in Texture Memory, thus taking advantage of Texture Cache and improving performance in memory accesses.

D. Phase 3: Path compression

All paths in the image are compressed and the result is the labeled image. Figure 3.d depicts the result of this step for the example pixels of the Section IV-B.

Algorithm 1 Union-Find: Local merge

Require: Image I of size $N \times M$.
Require: Adjacency relation $\mathcal{N}$.
Require: "Block" structure with index in the image and size of the Block.
Ensure: All pixels in the Block that are in the same connected component have the same label.

1: $bid \leftarrow$ Block 2D index of this thread
2: $id \leftarrow$ global 2D index of this thread
3: $shared_{mem}[bid] \leftarrow bid$ {Use shared memory, Indexes are in Block coordinates}
4: **for all** $id_n \in \mathcal{N}_{id}$ **do**
5: **if** $I(id) = I(id_n)$ **then** $merge(shared_{mem}, bid, bid_n)$
6: **end if**
7: **end for**
8: $V \leftarrow find(shared_{mem}, bid)$
9: $V_x \leftarrow V mod Block.Width$
10: $V_y \leftarrow V div Block.Width$
11: $L[id] \leftarrow (Block.Y \times Block.Height + V_y) \times M + Block.X \times Block.Width + V_x$

V. Neighbor Propagation

Now a completely different approach will be presented. Instead of trying to solve the problem in one pass in the image, the problem will be dealt with in many passes (until convergence). Each of these passes will be a simpler operation in the image. In Figure 5 is possible to see an example of the algorithm's execution. This algorithm may

Algorithm 2 Union-Find: Merge procedure

Require: Partially labeled image L.
Require: Raster index p in L.
Require: Raster index q in L whose pixel is going to be merged with p.
Ensure: p and q share a parent in L.

1: $done \leftarrow$ **false**
2: **repeat**
3: $p \leftarrow find(L, p)$
4: $q \leftarrow find(L, q)$
5: **if** $p < q$ **then**
6: $atomicMin(L[q], p)$
7: **else if** $p > q$ **then**
8: $atomicMin(L[p], q)$
9: **else**
10: $done \leftarrow$ **true**
11: **end if**
12: **until** $done =$ **true**

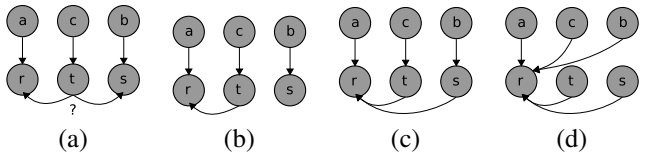


Figure 3. Example of merging: (a) Parallel merging of a-c and c-b; (b) Atomic operations link pixels a and c, but not b; (c) Another iteration of the algorithm successfully labels b; (d) Path compression.

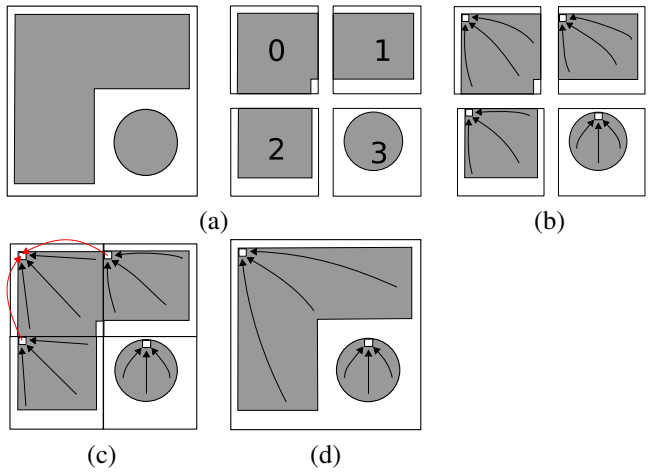


Figure 4. Union-Find execution: (a) Phase 0: A piece of pixels is assigned to a Block, this Block is given to a GPU processor. (b) Phase 1: Union-Find is done in each Block. (c) Phase 2: All pieces are merged. (d) Phase 3: Path compression

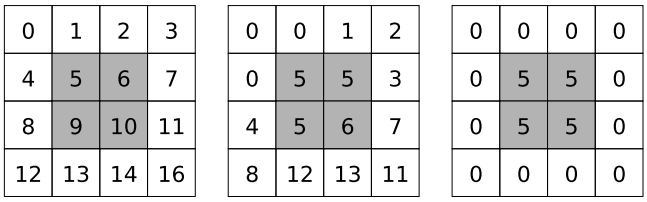


Figure 5. Neighbor Propagation execution: (a) Image that will be labeled, initial labels are the raster values of pixels. (b) Labels after first iteration. (c) Result is obtained after convergence.

sound naive, but it will help to understand the algorithm in Section VI.

A. Phase 0: Arrangements

As in Union-Find, the image is divided in rectangular pieces and each one is assigned to a Block, which is given to a GPU processor. Start labels are pixels' raster values as well.

B. Phase 1: Verify neighborhood

In this phase, every pixel p in the image will change its label l to the smaller label in its adjacency l', but only if $l' < l$. This step can be completely done in Shared Memory, which is faster, as said before in Section II.

C. Phase 2: Convergence test

If there was no replacement of labels in Phase 1, then the algorithm is over and the result is a labeled image; if not, return to Phase 1.

VI. Label Equivalence

Label Equivalence is an algorithm proposed for parallel hardware in [4] and implemented in GPU architecture in [9]. It can be described briefly as a hybrid version of Union-Find (Section IV) and Neighbor Propagation (Section V), in the sense that it has elements from both algorithms.

A. Phase 0: Arrangements

As in Union-Find and Neighbor Propagation, the image is divided in rectangular pieces and each one is assigned to a Block. While in Section IV and V, each pixel has just one label, now each pixel will have two labels. $L1$ will be the image composed by the first labels and $L2$ the image composed by the second ones. Both start values will be the raster values of their correspondent pixels in the image.

B. Phase 1: Verify neighborhood

For each pixel p, the smaller label in p's $L1$ neighborhood will be written in $L2$.

C. Phase 2: Path compression for $L1$'s roots in $L2$

For all pixels in $L1$ that are roots, a path compression (in the same way as Section IV-D) in the correspondent pixel in $L2$ is done, so all pixels which were roots in $L1$ have their final label assigned, as their chain of labels were resolved, notice this is a very expensive operation. See Algorithm 3 for a detailed description of this step.

D. Phase 3: Find $L1$'s new values in $L2$

This is the main step in the algorithm. Let's suppose the set of pixels that are roots in $L1$ is $R1$, the correspondent set of these pixels in $L2$ will be called $R2$. In Phase 1 and 2, the chain of labels for all pixels in $R1$ is solved and stored in the labels of $R2$.

Now, if r is a pixel in $R1$, there is a set C of pixels in $L1$ which have r as root and, therefore, they form a connected component. So, for all pixels in C, their new root is the new label of r, stored in $R2$.

Thus, this step consists in finding for all pixels in $L1$ the root of the correspondent pixels in $L2$, as we can see in Algorithm 3. The propagation step with a selective compression of paths is what essentially makes the performance of this algorithm better than Neighbor Propagation.

E. Phase 4: Convergence test

If there is no change in Phase 3, it is done and the labeled image is in $L1$. If not, return to Phase 1.

Algorithm 3 Label Equivalence: Path compression for $L1$'s roots in $L2$

Require: An image L of size $N \times M$.
Require: An image R of size $N \times M$.
1: $id \leftarrow$ global 2D index of this thread
2: $label \leftarrow L[id]$
3: **if** $L[id] = id$ **then**
4: $oldlabel \leftarrow L[id]$
5: $label \leftarrow R[oldlabel]$
6: **while** $oldlabel \neq label$ **do**
7: $oldlabel \leftarrow label$
8: $label \leftarrow R[oldlabel]$
9: **end while**
10: **end if**
11: $R[id] \leftarrow label$
12: **return** R

VII. Counting the number of different connected components

When labels are numbered as $1, 2 \cdots K$, it is easy to discover the number of connected components (K). But our components's labels are the smaller raster value of its pixels. However, there is a very efficient parallel algorithm to find the number of different connected components in an image. The algorithm uses *Prefix Sum* [10], a common

(a) (b)

Figure 6. (a) Image from the dataset. (b) Binarized and simplified version

primitive in parallel programming which suits very well GPU architecture. See Algorithm 4 for a description of the algorithm, note the initialization of vector R can be done in parallel either.

Algorithm 4 Count CC

Require: Labeled image L of size $N \times M$.
Ensure: The number of different connected components.

```
for i = 1 to N × M do
    if L[i] = i then
        R[i] ← 1 {Pixel i is root}
    else
        R[i] ← 0
    end if
end for
return PrefixSum(R)
```

VIII. Performance Results and analysis

Our benchmark used The Berkeley Segmentation Dataset [11]. Although the purpose of this article is not segmentation, but just to analyze algorithms' performance, this dataset was used because it is large enough (100 images) to provide a good sample of different images.

Moreover, all images were binarized and simplified (see Figure 6), as the labeling algorithms presented work only with binary images. Images were also resized to have at least 16 megapixels, the reason for this is that a big portion of execution time in small images is spent in memory transfers between the CPU and the GPU.

Execution times of Union-Find and Label Equivalence were compared with Stephano-Bulgarelli's algorithm [3], which is a very efficient two-pass serial labeling algorithm. This benchmark was executed in a computer with an Intel Core i5 processor and with a NVIDIA GeForce GTX 285 GPU. The results are shown in Table I. Neighbor Propagation's results were omitted because its running time was very long compared to the others presented methods.

Analyzing this results, Union-Find and Label Equivalence performances are similar, our result obtained for Label Equivalence is confirmed by data presented in [9]. Also, Union-Find is more *stable*, in the sense that its execution time is more predictable, as its running time standard deviation is much smaller than Label Equivalence's. Another interesting result is that GPU algorithms' speedup in relation with the serial algorithm is heavily dependent of the type of image being labeled. Figure 7 shows the distribution of data in this experiment.

Table I
Performance results

	Stephano-Bulgarelli CPU (serial)	Union-Find GPU	Label Equiv. GPU
Running time (ms)	$955 \pm 205ms$	$185 \pm 3ms$	$173 \pm 53ms$
Speedup	1	5.15 ± 1.09	5.78 ± 1.51
Min speedup	-	3.43	1.18
Max speedup	-	7.79	10.39

Table II
Spiral image

	Stephano-Bulgarelli CPU (serial)	Union-Find GPU	Label Equiv. GPU
Running time (ms)	$233ms$	$48ms$	$129ms$
Speedup	1	4.85	1.80

Both GPU algorithms performed bad with the "blank image" (an image with just one connected component which covers all image). This can be explained because it is necessary to merge and compress long chains of labels in memory to join Blocks, which is an expensive operation in GPUs. Also, Label Equivalence did not perform well in some cases, like the "line image", an image with horizontal lines, which is expected, as the propagation of labels in a line serializes the algorithm.

To check this hypothesis, we tested the algorithms with a spiral image (Figure 8), which has two connected components that are very interlaced with each other. This is a difficult test for local CCL algorithms, because it is very hard to propagate labels between blocks. The used image has 16 megapixels and each connected component has 1 pixel of thickness. In Table II we can see the result that Label Equivalence performed very bad, while Union-Find suffered a smaller performance hit.

These results indicate that using GPUs and GPGPU techniques for complex image processing tasks is feasible. Although our speedup (5x-10x) is far from the obtained in other fields like Molecular Dynamics [12], it is relevant, specially if we consider there are much more powerful GPUs in the market at the moment. Also, the algorithms presented here have no serial steps in them, thus the increase of performance with the number of processors will probably be linear.

In fact, this results can be explained because GPU hardware is designed for problems with a high locality in data access, while CCL problem is by nature non-local (global). That is why all efforts in this article were in the direction of breaking the problem in smaller and more local pieces and putting them together in the end to make the solution.

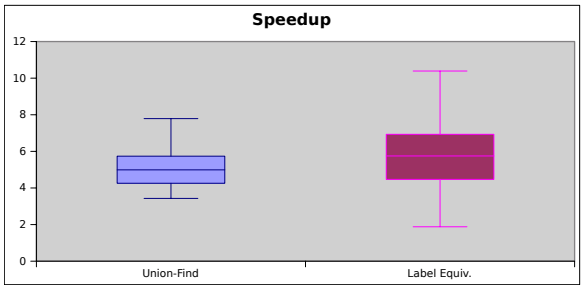


Figure 7. Speedup of the two CCL algorithms in a set of 100 images.

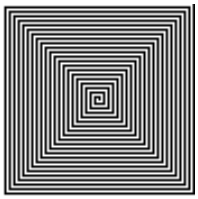


Figure 8. Spiral image

IX. Conclusion

We presented a comparative study of GPU algorithms using GPGPU techniques for the Connected Components Labeling problem. We proposed an Union-Find algorithm optimized for GPUs where the merge is executed locally first and globally later. We showed a method for counting the number of connected components in GPUs as well. Also, we compared this algorithm with Stephano-Bulgarelli's algorithm, an efficient serial one, and with another two algorithms for GPUs, Neighbor Propagation and Label Equivalence [9]. The results showed that our algorithm performed as well as Label Equivalence, and is more predictable, suffering less in specific cases (e.g., the "line image"), while Neighbor Propagation was much slower than both.

Also, one special point about the parallel algorithms here presented is that they have very good *scalability*, as there are not serial steps in them. Therefore, it can be concluded that both Union-Find and Label Equivalence algorithms will perform successfully in future GPUs.

All source code and image tests used in this article can be accessed at the collaborative scientific programming platform Adessowiki [13] in the link: *www.adessowiki.org/wiki/victor/view/*.

Acknowledgment

The authors thank Rubens Campos Machado, Bruno Forcioni, Diego Pereira Rodrigues, and Thiago Vallin Spina for support and collaboration. Also, the authors would like to thank FAPESP for financial support.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, second edition*. Cambridge, MA, USA: MIT Press, 2001, ch. Chapter 21: Data structures for Disjoint Sets, pp. 498–524.

[2] A. Rosenfeld, "Connectivity in digital pictures," *Journal of the ACM*, vol. 17, no. 1, pp. 146–160, 1970.

[3] L. di Stefano and A. Bulgarelli, "A simple and efficient connected components labeling algorithm," in *ICIAP '99: Proceedings of the 10th International Conference on Image Analysis and Processing*. Washington, DC, USA: IEEE Computer Society, 1999, p. 322.

[4] K. Suzuki, I. Horiba, and N. Sugie, "Linear-time connected-component labeling based on sequential local operations," *Computer Vision and Image Understanding*, vol. 89, no. 1, pp. 1–23, 2003.

[5] J. H. Reif, *Synthesis of Parallel Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[6] NVidia, "Cuda website," May 2010. [Online]. Available: http://www.nvidia.com/cuda

[7] M. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, pp. 948+, 1972. [Online]. Available: http://en.wikipedia.org/wiki/Flynn's_taxonomy

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, second edition*. Cambridge, MA, USA: MIT Press, 2001, ch. Chapter 22: Elementary Graph Algorithms, pp. 552–556.

[9] A. L. K.A. Hawick and D. Playne, "Parallel graph component labelling with gpus and cuda." Massey University, Computational Science Technical Note Series, April 2010. [Online]. Available: http://www.massey.ac.nz/ kahawick/cstn/089/cstn-089.html

[10] W. D. Hillis and G. L. Steele, Jr., "Data parallel algorithms," *Communications of the ACM*, vol. 29, no. 12, pp. 1170–1183, 1986.

[11] D. Martin, C. Fowlkes, D. Tal, and J. Malik, "A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics," in *ICCV '01: Proceedings of the 8th International Conference on Computer Vision*, vol. 2, July 2001, pp. 416–423.

[12] P. H. Colberg and F. Hfling, "Accelerating glassy dynamics using graphics processing units," Tech. Rep. arXiv:0912.3824. LMU-ASC 55-09, Dec 2009, comments: 18 pages, 8 figures, HALMD package licensed under the GPL, see http://colberg.org/research/halmd/.

[13] R. A. Lotufo, R. C. Machado, A. Körbes, and R. G. Ramos, "Adessowiki on-line collaborative scientific programming platform," in *WikiSym '09: Proceedings of the 5th International Symposium on Wikis and Open Collaboration*. New York, NY, USA: ACM, 2009, pp. 1–6.