Accelerated Connected Component Labeling Using CUDA Framework

Fanny Nina Paravecino and David Kaeli

Northeastern University, Boston MA 02115, USA
fninaparavecino@coe.neu.edu, kaeli@ece.neu.edu

Abstract. Connected Component Labeling (CCL) is a well-known algorithm with many applications in image processing and computer vision. Given the growth in terms of inter-pixel relationships and the amount of information stored in a single pixel, the time to run CCL analysis on an image continues to increase rapidly. In this paper we present an accelerated version of CCL using NVIDIA's Compute Unified Device Architecture (CUDA) framework to address this growing overhead. Our parallelization approach decomposes CCL while respecting all global dependencies across the image. We compare our implementation against serial execution and parallelized implementations developed on OpenMP. We show that our parallelized CCL algorithm targeting NVIDIA's CUDA can significantly increase performance, while still ensuring labeling quality.

Keywords: connected component labeling, CUDA, HYPER-Q, dynamic parallelism.

1 Introduction

Image analysis plays an important role in many applications in biomedical, manufacturing and security applications. Connected Component Labeling has been used to identify blobs or regions in a graph. In many of these applications, the graph represents the contents of an image. Different approaches of Connected Component Labeling have been proposed [1,3,5].

Parallelization has been used effectively in image analysis implementations to accelerate a number of data-parallel tasks. Image analysis is typically easily parallelized, especially given the large amount of data that typically needs to processed using a common set of operations. Previous work has focused on parallelization of CCL [2,4], even though CCL involves some global synchronization. While speedups can be achieved, synchronization limits the amount of speedup that can be achieved.

However, recent advances in parallel architectures, such as NVIDIA's Kepler graphics processor [7], have improved support for the class of global synchronization operations present in CCL. The focus of our work here is to exploit this parallelism and global synchronization mechanism effectively. We present a new implementation of CCL which offers better performance compared against previous serial and parallel approaches [6].

L.J. Chmielewski et al. (Eds.): ICCVG 2014, LNCS 8671, pp. 502–509, 2014.

CCL has been using in a number of image analysis settings. Specifically, in the field of physical security, there are particular tasks such as luggage scanning at airports that require near real-time response with a very high rate of accuracy. Labeling is one of the first steps in the scanning pipeline; the accuracy of labeling will strongly affect the quality whole system. Unfortunately, labeling is a time-consuming operation. For this reason, we propose an efficient parallel CCL implementation which leverages the parallel architecture of NVIDIA's Kepler using the CUDA programming framework.

This paper is organized as follows. In the next section we review the state of the art in CCL algorithms, as well as consider previous work on parallel implementations. In Section 3, we review the Kepler GK110 architecture (equipped with compute capability 3.5), the target system used in our evaluation. In Section 4, we present the proposed algorithm and in Section 5 present results of running CCL on Kepler and compare against a serial implementation and a parallel OpenMP implementation. Finally, we conclude the paper in Section 6, and discuss directions for future work.

2 Connected Component Labeling

Connected Component Labeling utilizes hierarchical data structures and union-trees. CCL can be applied to graphs or images. When used on an image, typically CCL uses two scans of an image and performs an analysis of every pixel [4]. There has been previous work that attempts to perform a single scan of the image [3], but this form of CCL lacks inherent parallelism, making it difficult to tune execution efficiency.

When working with images, the CCL algorithm begins by labeling each pixel of an image I based on its neighbors. If a pixel belongs to the background, the label 0 is assigned to it. If the pixel is not part of the background, its label is determined by the labels of the neighboring pixels. In a sequential implementation of connectivity, we would consider the $North$ and $West$ pixels first in order to determine X's label. CCL constructs a tree in the first scan, by choosing a root label for the region. Then CCL performs a second scan, updating temporary labels based on their smallest neighboring labels.

There have been a number of attempts improve performance of CCL [1,2,3,4,5]. Stripe-based Connected Component Labeling [1] makes 3 passes over the image, performing: 1) *stripe extraction representation* during the first pass, 2) *stripe union* during the second scan, and 3) *label assignment* during the final step. The first scan can be run in parallel, processing multiple rows concurrently. However, the *stripe union* phase runs a *find root* task which explores all regions until a root label is found. This exploration is inherently sequential, and therefore very costly.

Klaiber et al. presented a memory-efficient parallel single pass CCL implementation targeting an FPGA [4] in 2012. Their approach improved the execution speed of Bayley et al.'s Single Pass CCL FPGA-based earlier implementation [3], reducing overhead due to memory allocation of labels while searching regions.

When processing large images, they tiled images into small slices and evaluated neighbors between slices. This approach reduced the memory requirements significantly.

A fast CCL implementation was presented in [5]. The regions are connected at region boundaries, which are identified during the second scan. This approach considers performing only a half scan, since in the previous step the image was divided into subregions and they used these subregions to merge components using boundary overlaps. The disadvantage of this approach is it inherently sacrifices accuracy. The approach uses semi-supervised learning, since the user needs to calibrate which regions she would like to segment.

2.1 GPU-Based CCL Implementations

Previous work has evaluated a GPU-based implementation [2] which builds off of a classical hierarchical structure, generates a root label for each component, locally merges them using block-based division of the image, and then performs a global merge to rejoin blocks. Since the *local merge* can be run in parallel, an *atomic* operation is used to control the many threads updating the same pixel label. Atomic operations impact performance due to thread waiting, and the multi-step merging process increases the number of memory accesses.

In [10] Mehta et al. presented a parallel implementation of a video surveillance algorithm run on a NVIDIA GPU using CUDA. Their CCL implementation divides input video frames into tiles and computes labels sequentially. To merge tiles, they choose the heaviest label from among the different tiles.

Riha et al. presented a promising GPU-based implementation of CCL using a single scan of an image [11]. Their approach creates an intermediate dynamic structure to store properties of groups of pixels on a per row basis. Their approach is very close to our own, though in their design, in order to merge and create the objects they need to run an extra step to update the connections between contiguous rows. The main problem with their approach is the sequential update of properties for every object in their dynamic structure.

3 NVIDIA's Compute Unified Device Architecture (CUDA)

With the advent of GPU computing, a number of computational barriers have been overcome, enabling researchers to push the limits of applications that were previously limited by performance. GPU computing architectures have been used across a wide range of applications [6]. NVIDIA's previous generation of GPUs, the Fermi family, has been used in a number of applications, promising peak single-precision floating performance of up to 1.5 TFLOPS. However, NVIDIA's Kepler GK110 GPU offers more than 3.5 TFLOPs of single-precision computing capability. The newest features provided on the Kepler enable programmers move a wider range of applications to the CUDA framework [8].

3.1 Kepler Advanced Features

Given the level of sophistication provided in the Kepler, we have focused our work on accelerating CCL while run on this particular architecture. Two new features, introduced on the Kepler GK110, are utilized in our approach:

1. Dynamic Parallelism: Kepler adds the capability to launch child kernels within a parent kernel. One typical pattern in sequential algorithms are nested loops. Dynamic parallelism allows us to implement a nested loop with variable amounts of parallelism.
2. Hyper-Q: Kepler provides the ability to run multiples kernels assigned to different streams, concurrently. The Kepler GK110 supports up to 32 concurrent streams (as compared to 16 on the Fermi). Each stream is assigned to a different hardware queue.

Additional features included on the Kepler GK110 include texture memory, pinned memory, coalesced memory accesses, and new block/thread settings. Many of these features will be exploited in order to produce our optimized CCL implementation.

4 Accelerated Connected Component Labeling

The neighborhood operations present in CCL make it challenging to parallelize. We must modify the structure of the underlying algorithm if we want to exploit the massive thread-level parallelism present on the Kepler. Stripe-based CCL [1] modifies the first step of the algorithm by using row-level parallelization. We followed this approach in our design, but instead of only working on two rows at a time, we launched as many threads as rows in the image (I) during the first scan.

Formally, our Accelerated CCL (ACCL) uses two scanning phases, but our second phase simply updates a matrix that is half the size of the original image I. An intermediate stage reduces the size of labels associated with I.

4.1 Phase 0: Find Spans

We define a span as a group of pixels that have the same intensity in image I, and are located contiguously in the row. The structure of a span has two elements (y_{start}, y_{end}), which correspond to the column indices of the *starting* pixel and the *ending* pixel, respectively. $span_x$ defines a span in row x, and it is defined as follow:

$$span_x = \{(y_{start}, y_{end}) | I_{(x,y_{start})} = I_{(x,y_{start+1})} = \ldots = I_{(x,y_{end})}\} \qquad (1)$$

An intermediate matrix stores spans on a per row basis. Once a span is found, a label is immediately assigned to the span in matrix L. Matrix L has half the number of columns of I, since labels are only associated with spans, not pixels. Processing I in this fashion, we reduce the number of updates performed during *Phase 1*.

A kernel *findSpans* is launched with as many threads as rows in the original image I. Each thread will process its corresponding row and fill the *Spans* matrix with indices. At the same time, the *Label* matrix L is assigned values found for each span. Figure 1 shows the behavior of this kernel.

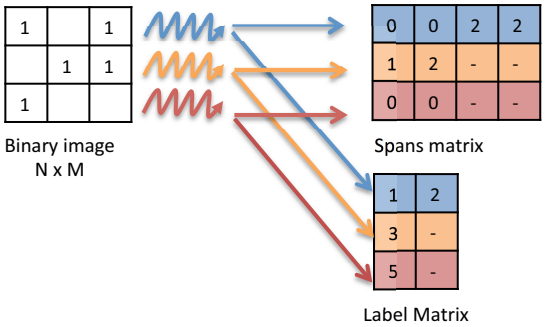


Fig. 1. The flow of the *findSpans* kernel

Our grid configuration attempts to maximize GPU occupancy. We launched 256 threads per block, where each image allocates 2 blocks. We can process up to 8 images per SMX before reaching full occupancy.

4.2 Phase 1: Merge Spans

In order to reduce the time taken for the *Union-Find* phase (which involves an exploration of previous regions to find the root), we record in matrix L the corresponding root label associated with each span.

Our second kernel *mergeSpans* joins two spans from contiguous rows, only if the indices of the spans overlap and both spans have the same intensity.

$$merge(span_x, spans_{x+1}) = \begin{cases} 1 & \text{merge if indexes } span_x \text{ and } span_{x+1} \text{ overlap} \\ 0 & \text{skip otherwise} \end{cases}$$

If the condition described above is satisfied, a *child* kernel is spawned with the one thread per label in L. These threads will update the respective labels of the newly added segment. Figure 2 shows the memory access pattern when the child kernel is launched. Spawming is enabled exploiting dynamic parallelism, which only became available in CUDA compute capability 3.5. The time to update multiple labels is the same as to update one single element of matrix L. We set up two grid configurations for this kernel. The parent kernel associates one block per image. The child kernel has 256 threads per block and 512 blocks.

Most of the memory accesses generated from threads of the same block perform read-only accesses of contiguous memory locations. This memory behavior is well-suited for the GPU *texture memory*. Texture memory is a read-only memory that is cached on-chip on the GPU. Utilizing texture memory can speed-up data transfer when threads of the same block access contiguous memory locations.

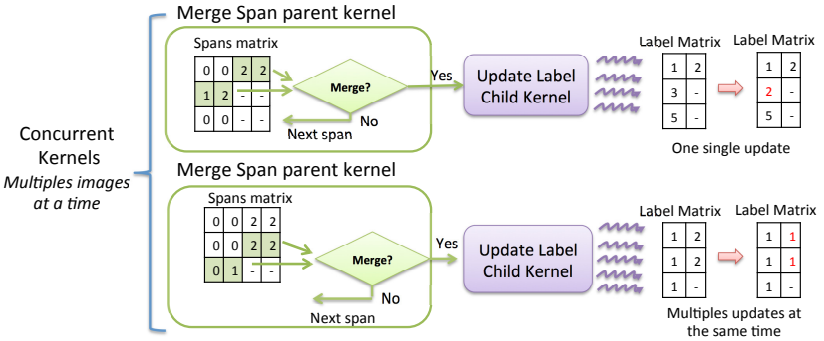
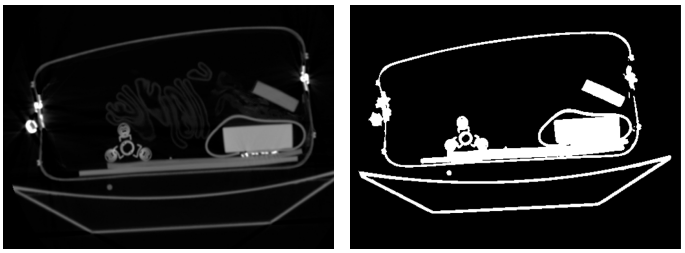


Fig. 2. The flow of the *mergeSpans* kernel

5 Performance Results and Analysis

Next, we present results of running real-world applications on an NVIDIA GK-110. Our experiments use DICOM images [9] of actual luggage scanned at airports. Each DICOM set contains more than 700 images (512x512) of the same luggage. All images were binarized and simplified (see Figure 3), even though ACCL works well with varying intensity values.

Our experiments were run on an Intel Core i7-3770K processor and with a NVIDIA GTX Titan GPU, compute capability 3.5 and CUDA 5.5. We used gcc compiler 3.7 and OpenMP 3.0.



(a) DICOM image (b) Binarized and simplified version

Fig. 3. The input image used in this work

We provide results for optimized serial, OpenMP and CUDA implementations in Table 1, all achieving the same labeling accuracy. Computation was recorded for one image at a time. The OpenMP configuration that uses two threads and shared memory for matrix L obtains the best performance.

We can see that our proposed parallel ACCL algorithm achieves an average speedup of 5x over CCL serial. Our parallel implementation ACCL reduces the complexity of *Phase 0* from CCL Serial $O(N*M)$, where N and M are rows and columns of input matrix, to $O(N)$. Furthermore, for the relabeling in *Phase 1*,

we reduced the CCL serial complexity of $O(N * M)$ to $O(1)$. However, in ACCL *mergeSpans* is still stuck with the same serial complexity in order to respect all global dependencies.

We compared our results against Stava's CUDA algorithm [6]. This algorithm processes 1542 Mpixels/s of for a 512^2 CT image, while our algorithm processes 5242 Mpixels/s. This is a speedup of over 3.3x.

Table 1. Performance results

Method	Running Time(s)	Speedup
CCL Serial	0.25	1.00x
CCL OpenMP	0.18	1.39x
ACCL	0.05	5.00x

In Table 2 the runtime our ACCL implementation using NVIDIA's Hyper-Q feature for processing multiple images concurrently is compared to the serial CCL runtime. Since we are using dynamic parallelism, there is a limitation in terms of the hardware, which restricts the number of child kernel threads that can be spawned. In the case where multiple images are processed concurrently, child kernel threads will not be able to be spawned, and labeling quality may suffer. In order to maintain labeling accuracy, we have added a sequential loop inside of the child kernel, and thus reduced the number of child kernel threads spawned. Adding the sequential loop reduces the speedup of our implementation (as is seen in Table 2 for Stream 1). Introducing the loop allows us to use the Hyper-Q feature effectively to process multiples images in parallel, and therefore achieve a degree of speedup as we increase the number of processed streams.

Table 2. Performance results comparing ACCL with Hyper-Q against CCL Serial

#Streams	CCL Serial(s)	ACCL(s)	Speedup
1	0.25	0.05	5.00x
2	1.08	0.10	10.80x
3	2.16	0.14	15.36x
4	4.18	0.19	21.44x
5	6.09	0.23	25.91x

We further investigated the cause of slowdown related to the loop added in the child kernel. Due to the nature of *mergeSpans* (parent kernel), a sequential analysis explores all spans. Thus, complexity of *mergeSpans* is $O(N * M)$. Since complexity of the child kernel is now increased from $O(1)$ to $O(M)$, the overall complexity of parent kernel becomes $O(N * M^2)$. Therefore, performance slowdown is reflected for stream 1.

The speed-up of any implementation that utilizes multiple streams greatly depends on the number of concurrent streams running, and partially depends on the image structure. As more streams run concurrently, we can continue to see benefits, though the speedup begins to tail off. The ideal performance for

Hyper-Q would be linear speedup (in the number of concurrent streams), but the work distributor inside the GPU starts distributing blocks as soon as the kernel is launched. If one kernel dominates the GPU, based on the intensity of the kernel (which is the case for our implementation for dynamic parallelism), then the second kernel will have to wait until first kernel is completed, at which point the second kernel can start execution. Thus, in this case we will only see a small amount of concurrent parallelism.

6 Conclusion

In this paper we presented Accelerated Connected Component Labeling (ACCL) using the CUDA framework. We experimented with new features of the NVIDIA Kepler GPU. Our proposed algorithm achieves improved performance versus prior serial and OpenMP parallel implementations. We also compared the serial CCL to processing multiples images concurrently using Hyper-Q running ACCL. The results showed that our algorithm could scale well as long as we increased the number of streams.

One interesting observation related to the parallelized algorithm we have presented here is that while dynamic parallelism improves performance for a small number of child kernels, it turns out to be a disadvantage when trying to use a larger number of child kernels. Hyper-Q, with dynamic parallelism, provides some benefits, but is not a perfect match for ACCL to scale performance.

References

1. Zhao, H.L., Fan, Y.B., Zhang, T.X., Sang, H.S.: Stripe-based connected components labelling. Electronics Letters 46(21), 1434–1436 (2010)
2. Oliveira, V., Lotufo, R.A.: A study on connected components labeling algorithms using GPUs. XXIII Sibgrapi, Graphics, Patterns and Images (2010)
3. Bailey, D., Johnston, C.: Singles Pass Connected Components Analysis. Image and Vision Computing (2007)
4. Klaiber, M., Rockstroh, L.Z., Wang, B.Y., Simon, S.: A memory-efficient parallel single pass architecture for connected component labeling of streamed images. Field-Programmable Technology (FPT), 159–165, 10–12 (2012)
5. Paralic, M.: Fast connected component labeling in binary images. In: 35th Telecommunications and Signal Processing (TSP), vol. 709, pp. 3–4 (2012)
6. Hwu, W.-M.: GPU Computing Gems Emerald Edition. M. Kaufmann (2011)
7. NVIDIA's Next Generation CUDA Compute Architecture Whitepaper: Kepler GK110. Nvidia (2013)
8. Foley, J.: Migrating your code from Tesla Fermi to Tesla K20X, with examples from QUDA Lattice QDC library. Microway, Inc. (2013)
9. National Electrical Manufacturers Association: Digital Imaging and Communications in Medicine (DICOM)., `http://medical.nema.org/standard.html`
10. Mehta, S., Misra, A., Singhal, A., Kumar, P., Mittal, A., Palaniappan, K.: Parallel implementation of video surveillance algorithms on GPU architectures using CUDA. In: 17th IEEE Int. Conf. Advanced Computing and Communications, ADCOM (2009)
11. Riha, L., Manohar, M.: GPU accelerated one-pass algorithm for computing minimal rectangles of connected components, pp. 479–484. IEEE Computer Society Press (2011)