DEGREE PROJECT IN TECHNOLOGY,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2016

Is GPGPU CCL worth it?

A performance comparison between some GPU and CPU algorithms for solving connected components labeling on binary images

ALEX SUNDSTRÖM

VICTOR ÄHDEL

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION

Is GPGPU CCL worth it?

A performance comparison between some GPU and CPU algorithms for solving connected components labeling on binary images

Alex Sundström Victor Ähdel

May 8, 2016

CSC, KTH
Degree Project in Computer Science, DD143X
Supervisor: Mårten Björkman
Examiner: Örjan Ekeberg

Abstract

Connected component labeling (CCL) is a traditionally sequential problem that is hard to parallelize. This report aims to test the performance of solving CCL using massively parallel hardware through GPGPU. To achieve this several CCL algorithms were researched and implemented using C++ and OpenCL. The results showed an improvement of up to a factor of 2, which is insignificant when also considering memory transfer. In conclusion, performing CCL on the GPU is not worth it if the data has to first be transferred to and from the GPU.

Sammanfattning

Lönar sig GPGPU CCL?

Etikettering av sammansatta komponenter (CCL) är ett traditionellt sekventiellt problem som är svårt att parallellisera. Denna rapport ämnar att testa prestandan av att lösa CCL med användning av massivt parallell hårdvara genom metoden GPGPU. För att uppnå detta undersöktes och implementerades ett flertal CCL algoritmer i C++ och OpenCL Resultaten pekar på en förbättring upp till en faktor av 2, vilket är obetydligt när man också tar hänsyn till minnesöverföringen. Sammanfattningsvis så är det ej värt att utföra CCL med GPGPU om data även måste överföras till och från GPU.

Contents

Terminology

GPU	Graphics Processing Unit
GPGPU	General-Purpose computing on Graphics Processing Unit
SIMD	Single Instruction Multiple Data
CCL	Connected Components Labeling
Pixel	An indivisible location on some 2-dimensional data grid.
Foreground pixel	A pixel with a value of 1, often displayed as white.
Background pixel	A pixel with a value of 0, often displayed as black.
Computing unit	A part of an OpenCL device that can execute work groups. A device usually has several of these.
Work-group	A collection of related work units.
Work unit	One execution of a kernel, usually on a specific pixel.

Chapter 1

Introduction

In recent history, a huge amount of algorithms have been developed, most of which are sequential in nature due to the hardware it was constructed for. However, nowadays performing calculations on hardware such as a GPU is growing more popular as they are massively parallel. This means they have access to a large amount of cores (1000+) to use in their calculations. As such a GPU offers more efficiency in terms of cost per GFLOPS and similar measurements than their sequential counterparts. Using this extra power generally leads to faster solutions to the problems [9]. If the sequential algorithm only uses one core of this new type of hardware, the execution time will suffer. So, it would be preferable if we could develop parallel algorithms for these problems, such that we can use the computing power available in parallel hardware. Even naive parallel algorithms may, due to this difference in computational power, be quicker in reality.

However, certain problems seem to be inherently sequential; computations that need to be completed before some other computation can create dependencies that are hard to solve in a parallel manner. Still, a more brute-force algorithm might outperform the sequential algorithm. The problem of connected components is one where the normal naive recursive algorithm would seem to be easily parallelizable, but recursion depth and similar issues create problems. Similarly, there are a few well-known optimizations that mostly makes the problem even more sequential. But since parallelization is so important, especially for real-time systems such as many using this algorithm, we investigated it.

1.1 Purpose

The purpose of this paper is to present some possible GPGPU algorithms for the problem of CCL and to elucidate the performance difference that may be had by using a GPU for the problem. It is of interest as replacing a CPU

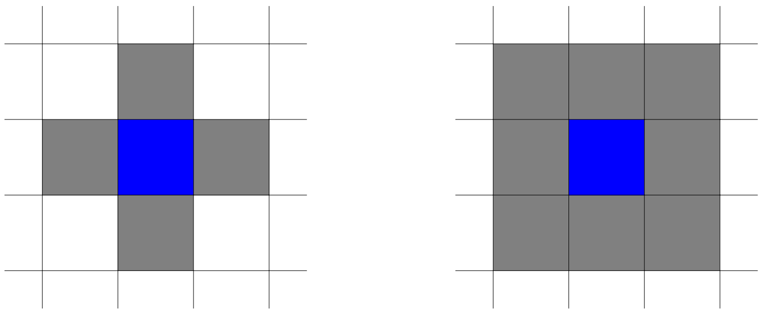

(a) 4-connected mask. (b) 8-connected mask.

Figure 1.1: 4- and 8-connected masks, the blue pixel is the one currently being processed, the gray are those that it is connected to.

version is a trade-off between a possible performance gain and code complexity inherent in current GPGPU methods.

1.2 Scope

The problem of connected components mathematically regards graphs. We will consider only the special case where the graphs represent an image.

The problem requires a definition of which pixels are connected to one another. A 4-connected system means that a pixel is considered to be connected to the pixel immediately to the north, south, west or east (in the image). A 8-connected system is the same as 4-connected with the addition of pixels to the northwest, northeast, southwest and southeast. Figure 1.1 shows the pixel masks of the two connectedness definitions. We will only consider 4-connected components as the 8-connected problem is very similar. Choosing the 4-connected problem also leads to slightly more succinct algorithms.

Certain implementations of CCL manage segmented images or full color images by using some measure of similarity. For most uses one can instead first threshold the image into a binary image, where the pixels either belong to the foreground or background. Two pixels are then connected if they both belong to the foreground and are 4-connected. This paper will only treat CCL of binary images.

1.3 Problem statement

This paper attempts to resolve whether implementing CCL using GPGPU techniques instead of using existing sequential algorithms can result in an improvement in performance.

Chapter 2

Background

2.1 GPGPU

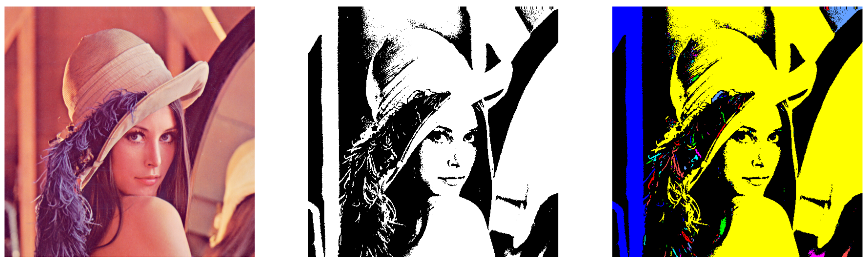
GPGPU is a technique where a program can be executed on the GPU. GPUs are multi core processing units which specialize in floating point calculations. While that lends itself to graphics processing it can also be useful for more general algorithms. That is because the GPU structure allows for very potent SIMD performance.

To utilize GPGPU certain frameworks can be used and an example of such a framework is CUDA [3]. CUDA however only works on Nvidia GPUs. An alternative that works on GPUs by other manufacturers is OpenCL [5]. There are some differences between the two frameworks, but they perform similarly [4].

2.1.1 OpenCL

When using GPGPU techniques one cannot treat the GPU as a platform with large amounts of completely separate threads. This is since the threads of execution really are small parts of several bigger cores, usually hosting around 64 threads. If no care is taken to group the threads together one might end up executing only one thread on each core. And doing so often leads to horrible performance.

While what the GPU has may very well be likened to a CPU thread, it is not entirely correct and different terminology needs to be used. The useful OpenCL terminology for these concepts is that a single thread is called a *work item*, belonging to some *work-group* [11]. Each *work-group* is executed on a *compute unit*, which is the equivalent of a CPU core. In this way, most or all of the work units in some work-group are executed simultaneously. A key part of optimizing OpenCL code is to ensure the work-groups are of a reasonable size such that the compute unit can work fully.



(a) Original image. (b) Thresholded image (Otsu's). (c) Labeled image.

Figure 2.1: Process of connected components labeling. Original image taken from the USC-SIPI database.

Each work unit can access both a global id that is dependant on the problem size and a local id. This local id is thus $0 < id < workgroupsize$, for each dimension of the problem. For our problem we will most often let each pixel of the image be processed by one work unit. It is then reasonable to call the rectangular block of pixels that some work-groups' work units process that work-groups local area.

2.2 Connected components

Connected components is an interesting problem of graph theory. A subgraph is a connected component if for every vertex in the subgraph there exists a path to every other vertex in the subgraph. A path is defined as a set of distinct vertices v_1, $v_2 \ldots v_n$ where there exists an edge between v_i and v_{i+1} for every $i \in N$, $1 \leq i < n$.

This notion is easily extended to images since we can imagine an image being a graph where every pixel is a vertex, and edges are between immediately neighbouring pixels. There should then be an edge between two pixels if they are 4-connected and they are both foreground. This specific instance of the problem, concerning images, has plenty of uses. For example: any time one wants to process a logical "object" in the image, such as a car, we would like some way of finding larger clusters of pixels that are somehow related.

Figure 2.1 shows an example thresholding and connected components labeling. Note that some labels share colors, all separate components are actually labeled differently.

2.3 Existing algorithms

There exists several completely different approaches to solving this problem, we will mention briefly the most important ones. Note that any algorithm that executes in $\mathcal{O}(n)$ is considered theoretically optimal. While there will be a difference in performance between these algorithms, most are considered "optimal".

2.3.1 One component at a time

The perhaps simplest sequential algorithm only scans through the (binary) image once [1]. Once it finds a foreground pixel without a label it pauses the scanning of the whole image. It instead starts to map out the entire component connected to which that one pixel belongs to. The algorithm accomplishes this by pushing pixel positions onto a stack whenever new ones are discovered. It then continues to pop from the stack in order to visit new pixels until the stack is exhausted.

2.3.2 Two-pass

The traditional way to label connected components when the image is stored as a binary array is to iterate over it at least twice [14]. The goal of the first iteration is to find and label as many connected components as possible. To accomplish that, each pixel is assigned a temporary label. They receive some label dependant on their neighbours if at least one neighbour belongs to the foreground. Otherwise the pixel is assigned a previously not introduced provisional label. For exceptionally simple images this might be enough for a correct labeling.

The greater problem occurs whenever the available neighbours have several different labels. In that case, something needs to be done to ensure that those neighbours' labels are considered to both belong to the same component. So the algorithm somehow registers the equivalence between the relevant neighbouring labels and gives the current pixel either one of the neighbouring labels or some special flag. Subsequent passes then aim to merge labels that really should be the same.

A few variations of the algorithm exist, which mostly handle the label equivalence in different ways. The old solution to the label equivalence is to put n-tuples ($n = 2$ for 4-connected, $n = 4$ for 8-connected) into a table whenever two labels should be equivalent. After the first step the table is then transformed into a new table where the first label is the smallest label belonging to the set of equivalent labels. The transformation is accomplished through something similar to algorithm 2.1. As can be seen, such a transformation requires quite a bit of work depending on the size of the table.

Algorithm 2.1: Equivalence table transformation for 4-connected.

```
input: $T$ //Starting table
output: $T_2$

sort $T$ on first element
foreach $x$ in $T$
  move $x$ to $T_2$
  foreach $y$ in $T$
    if $y_1 = x_2$
      $y_1 = x_1$
    if $y_2 = x_2$
      $y_2 = x_1$
```

A newer version instead employs a union-find data structure in order to not need to post-process any tables. It instead leads to the data structure automatically recording equivalences between all equivalent labels due to the union part. Such data structures have two functions: $union(A, B)$, merging the set belonging to A with that belonging to B, and $find(A)$. $find(A)$ returns a representative of the set such that if A and B are in the same set, $find(A) = find(B)$. No post-processing step is then necessary, and in the second pass the algorithm simply replaces every label with $find(label)$.

2.3.3 Parallel versions

An example of an existing GPGPU solution to the problem of CCL is an extension of the algorithm that uses union-find to merge pixels with the same label [12]. One problem presented with the regular union-find method is the GPUs reliance on locality of the data it is processing. Connected components can span the entirety of the input, as in the image. Therefore it is important to think ahead of how to design an algorithm that can try and break the problem into smaller parts to make the most of the GPUs power. One that treats each pixel as such a small part is neighbour propagation [6]. But it instead suffers from different problems due to looking at a too small area, possibly leading to bad performance.

Some parallel algorithms also exist that are not optimal for a GPU, due to techniques that can only use (or otherwise don't benefit from more than) 2–8 cores. Among those are an image adaptation of Tarjan's [8], and a strategy dealing with one row at a time and then merging them [10]. Several parallel versions also exist for the more general case of graphs [15, 2]. While these algorithms might not straightforwardly solve our problem, an adaptation of them might.

Chapter 3

Method

The first step in this study was to perform a literature study on algorithms relating to the problem of Connected Components. A number of known CPU algorithms were studied and then implemented in order to establish a baseline. Then a few known naive GPGPU algorithms and some CPU algorithms adapted for the GPU were implemented to elucidate any performance differences. We then compared the execution time of the algorithms.

The algorithms themselves are explained in chapter 4.

3.1 Language

To accomplish any of the GPGPU programming, we had to choose between the two most used frameworks, OpenCL and CUDA. But since CUDA is locked to NVIDIA hardware, we chose OpenCL.

To record the execution time a testing environment was created using C++. C++ was also used to implement the CPU based algorithms and to gain access to OpenCL functionality. The GPU algorithms were then coded as 'OpenCL C' kernels.

3.2 Hardware

The CPU algorithms were tested on a system with an Intel i7-4770k. In order to have some robustness regarding OpenCL vendor implementations, we tested using both an AMD and an Nvidia graphics card. The Nvidia GPU is an Asus GTX 960, and the AMD is an MSI Radeon R9 280X.

3.3 Data

We used The USC-SIPI Image Database [17]. The *Textures*, *Aerials*, and *Miscellaneous* volumes were downloaded and split into two categories depending on their size. Sizes of 512x512 pixels were kept in one category, and those of 1024x1024 in another. Images whose sizes did not fit into one of these categories were simply discarded.

3.4 Testing

The testing environment records the wall-clock time that it takes for the algorithm to execute. It also separately records the same time but including the time it takes for the memory to copy to and from the relevant locations. In the case of GPU algorithms this memory transfer time was expected to be quite significant, since the GPU memory is physically separate from the RAM.

The entire tests were repeated 5 times for each of the hardware configurations. This is in order to reduce variability of the testing environment itself, where OS scheduling and similar problems may affect the timings slightly.

The environment also checks the output after the timers are stopped in order to validate that the labeling is correct. It first looks only at the output and verifies that no two different labels are right next to another, and similar sanity checks. It then compares the output to a previous output from a verified algorithm, verifying that they are equivalent.

For a realistic thresholding of the images we first used a smart thresholding scheme before running the testing program. For this we chose Otsu's method [13], which is a thresholding that minimizes the intra-class variance.

In the code a choice often has to be made about how large the different workgroups are. For each system we manually tested different sizes until we found the optimal values.

Chapter 4

Algorithms

Here we will briefly explain the algorithms used in the tests. Some algorithms are fairly naive, and are mostly included for the sake of having some easily relatable algorithm in the comparison.

All the algorithms receive a 2D image with numbers 0 for background and 1 for foreground. As output they are expected to provide a labeling with arbitrary numbers in a similar image. As such, there is no requirement that they are consecutive in some low numbering. The numbers may not, however, be 0 or 1 as those are reserved for their special property of representing background and foreground. Not allowing 1 is not generally necessary, but choosing to start some numbering at 2 instead of 1 is trivial, and it assists in debugging. Background pixels are expected to remain 0 in the output.

We do not actually store the images as 2D images. This is due to various requirements of OpenCL regarding 2D images, such as requiring a sampler and two images in order to both read and write. We instead use a 1-dimensional buffer of the same size as the 2-dimensional one, but indexed slightly differently. So instead of $image[y][x]$ we use $image[w \cdot y + x]$ where w is the width of the image.

4.1 CPU

All the CPU algorithms provided use only a single core.

4.1.1 One-pass

The one-pass algorithm is the same as, and works just like the one described in section 2.3.1. That is, it iterates through the image looking for unlabeled foreground pixels [1]. When it finds one it pauses and labels the entire component that it is connected to. The implementation is accomplished with the use of a vector that stores the position of pixels contained in the component.

The vector is initialized to only contain the first pixel's position. That position is then removed from the vector and used to check if neighbouring pixels belong to the foreground. If that is the case then they are assigned their label and their position added to the vector. The process is then repeated until the vector contains no new elements at which point the entire component has been explored and labeled.

4.1.2 Union-find

Union-find as used in this report is implemented as a two-pass algorithm. In the first pass when a foreground pixel is found the algorithm makes note of the values contained in the pixels north and west of the found pixel. This is to compute which label to assign the current pixel.

1. If both neighbours are background pixels then the pixel is assigned a new label according to a formula of it's position ($location + 2$).
2. If only one of the neighbouring pixels belongs to the foreground then the current pixel is assigned the same label.
3. If both neighbours are in the foreground then their root-pixels are calculated using algorithm 4.1 and the lowest root is used when assigning the label.

Once the first scan is completed there are still some label equivalences to solve however. This is done by performing a second scan where the Find set function is used again on every foreground pixel.

Algorithm 4.1: Find set

```
input $image[], location$

while $location \neq image[location] - 2$
  $location \leftarrow image[location] - 2$

return $location$
```

4.1.3 Linear Two-scan

An efficient two-scan algorithm for labeling connected components makes use of three tables to solve the label equivalence in the first scan [7]. This is done by using three different tables to store the label relation, the next label in the component and the tail of the component (these tables are referred to as *rl_table*, *n_label*, and *t_label* respectively in the pseudo code). If a group of labels are in the same component they will all have the same representative label, usually the lowest in the component. The next label table is used to iterate through a component when solving the label equivalence. The tail table instead stores which label is the last in the component for a specific representative label. Since all label equivalence is taken care of in the first iteration the second is simply used to assign all labels the value of their representative label.

Algorithm 4.2: Linear two-scan

```
input: $image[], imageHeight, imageWidth$
$m \leftarrow 2$
$w \leftarrow imageWidth$

//Out of range checks for the image[] array accesses were omitted
for $0 \leq y \leq imageHeight - 1$
  for $0 \leq x \leq imageWidth - 1$
    if $image[y \cdot w + x] \in foreground$
      if $image[y \cdot w + (x-1)] \in background$ and $image[(y-1) \cdot w + x] \in background$
        $image[y \cdot w + x] \leftarrow m$
        $rl\_table[m] \leftarrow m$
        $n\_label[m] \leftarrow -1$
        $t\_label[m] \leftarrow m$
        $m = m + 1$
      else if $image[y \cdot w + (x-1)] \in foreground$
        $image[y \cdot w + x] \leftarrow image[y \cdot w + (x-1)]$
      else
        $image[y \cdot w + x] \leftarrow image[(y-1) \cdot w + x]$

    $u \leftarrow rl\_table[image[y \cdot w + (x-1)]]$
    $v \leftarrow rl\_table[image[(y-1) \cdot w + x]]$
    if $u \neq v$
      if $v < u$
        $swap(u, v)$
      $i \leftarrow v$
      while $i \neq -1$
        $rl\_table[i] \leftarrow u$
        $i \leftarrow n\_label[i]$
      $n\_label[t\_label[u]] \leftarrow v$
      $t\_label[u] \leftarrow t\_label[v]$
```

4.1.4 Front Back

To solve the problem of connected components algorithms employing the use of multiple scans can be implemented [16]. Here an initial scan in the forwards raster direction is used to create provisional labels and store which component they are connected to in a label connection table. This table is used for comparison when the algorithm computes the value of a foreground pixel. The completion of the first scan is followed by one or more pairs of scans both in the backwards and forwards raster direction. These scans aim to further solve label connectivity and will continue until no change is recorded and the algorithm completes. Figure 4.1 shows a small part of the initial forward scan.

Similar to other implementations the label connection table points keeps track of root labels. During computation the algorithm will assign label values equal to the lowest root of the neighbouring labels. Both the current and the neighbouring pixels will receive this value.

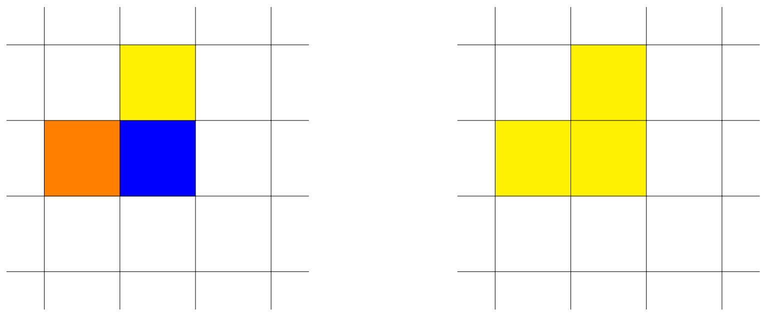

(a) Before computing center pixel (b) After computing center pixel

Figure 4.1: Example of the initial iteration of the front-back algorithm. The blue pixel is the current foreground pixel (lacking a label) being processed and the yellow and orange are already assigned provisional labels, with yellow representing a lower label value

If both neighbouring labels are 0 as in background pixels then a new provisional label will be assigned. When performing the initial forwards iteration (in the raster direction) the neighbouring pixels will be the western and northern labels. During the subsequent pairs of backwards and forwards iterations that follow the current pixel's label will also be taken into account. A backwards iteration will compare the southern and eastern labels.

4.2 GPU

4.2.1 Neighbour propagation

Neighbour propagation uses the simplest possible kernel, that only checks its immediate neighbours for some label that is lower than its own [6]. Algorithm 4.3 shows the relevant pseudo code. The code is run for each pixel in the image.

Algorithm 4.3: Neighbour propagation kernel

```
input: $t$ (this label)
foreach $n$ in connected neighbours
  if $1 < n < t$
    $t \leftarrow n$
```

Unfortunately, this is not sufficient as one iteration almost never is enough to complete a labeling. We must therefore use some method of reaching convergence. So we slightly modify the algorithm to write true to *changed* whenever it changed anything. Then we can use algorithm 4.4 until reaching convergence.

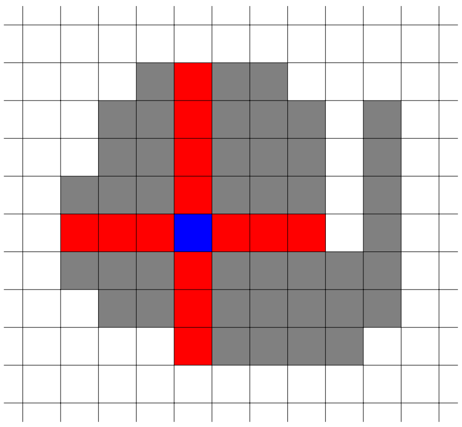


Figure 4.2: Which pixels a certain run of plus propagation will consider. Gray pixels are foreground, blue is current pixel. Red pixels are those that it considers for minimum.

Algorithm 4.4: Generic convergence algorithm

```
changed ← true
while changed
  changed ← false
  execute kernel
```

Since there is no method of synchronizing across work-groups in OpenCL, algorithm 4.4 has to be implemented in the C++ host code, while algorithm 4.3 is implemented as a kernel.

4.2.2 Plus propagation

Plus propagation works very similarly to neighbour propagation. The main difference is that it considers all the pixels on a straight line to the north/east-/south/west when calculating the minimum. The straight line stops as soon as it encounters some background pixel. Figure 4.2 shows the "plus" that this creates.

Similarly to neighbour propagation, one iteration of this does not guarantee a correct labeling. Therefore, the convergence algorithm 4.4 is used in the same manner as previously.

4.2.3 Line editing

Unlike the previous algorithms line-editing is run on all rows/columns simultaneously, instead of each pixel. The algorithm attempts to merge as much as possible of the row/column it is processing, while going in some direction. For the directions we use four different kernels, one for each of north, east, south, and west. Each of these are run after one another in the hopes that the lowest label of a component is propagated quickly in all directions. Like the previous strategies we cannot guarantee a quick solution, so we must again use the convergence scheme previously mentioned. Algorithm 4.5 shows the necessary steps for the kernel going east. The kernels for the other directions are constructed similarly.

Algorithm 4.5: Line editing, east

```
input: $image[]$, $y$, $w$ //row, width
$x \leftarrow 0$
$m \leftarrow \infty$ //Lowest found label
while $x < w$
  $c \leftarrow image[w \cdot y + x]$ //Current
  if $c = 0$
    $m \leftarrow \infty$
  else
    if $c < m$
      $m \leftarrow c$
    else if $c > m$
      $image[w \cdot y + x] \leftarrow m$
  $x \leftarrow x + 1$
```

4.2.4 Lookahead line editing

Lookahead line editing is very similar to line editing. The difference is that we don't write anything until we have explored until the end of any component on the line. After we have found the minimum of that line of the component it then writes the entire thing. This implies that every pixel of that line gets the correct minimum of that line immediately. In terms of line editing, the result is equivalent to a, for example, east run followed by a west run. That then means that we only need two different kernels for north+south and east+west. This should, in most cases, be an optimization, but unfortunately it causes image access to have slightly worse locality due to long lines and starting over when writing.

4.2.5 Union-find

Union-find works pretty much like the CPU version with the same name [12]. The difference is that we cannot guarantee which order the pixels are processed, and thus that the north and west pixel has already completed processing. This means that cannot rely on the first pass correctly joining the trees of labels. Therefore we again need to iterate until convergence.

Since a pixel to the left of another might not have completed its labeling first, it is beneficial to let the left pixel look at the value of the right one. Due to this, we also modify the algorithm such that it also considers the pixels to the east and south.

4.2.6 Stack-based exploration

Stack-based exploration is a GPU adaptation of the CPU one-pass algorithm. Limitations such as local memory size however mean that we cannot fit any possible component onto one stack. This breaks a necessary invariant of the one-pass algorithm; that a component is completely labeled after first being discovered. We also cannot use the global memory for this stack since that would either require synchronization between work-groups, or letting a single work-group deal with possibly huge components, making the algorithm too sequential for GPU usage.

This adaptation thus foregoes any such invariants, and lets the work-groups attempt to improve the labeling, while at least not weakening it. Assuming a work-group cooperates internally and picks the best label it can see locally, there are still two problems that occur:

- A neighbouring work-group actually has a better label that it is trying to spread.
- It runs out of the limited local stack space.

The first item has a fairly easy, but costly, solution. If we never write to a pixel that has a better label, the better label will eventually spread into this work-groups local area. We can then iterate like the rest of the algorithms, and pick and spread this better label the next time around. Note that this means that this algorithm is not one-pass, in any sense of the word.

The second item is solved similarly. As long as we don't add any pixels to the stack without also modifying its value, we cannot run out of stack without any changes. This then means that a new iteration is guaranteed, and the stack will in the next iteration continue where it left off. As a pleasant surprise, running out of stack space is thus sometimes beneficial since the next iteration is more quickly started. It is then very likely that some other work-groups local area has been touched by this superior label. Those work-groups can then cooperate with continuing to spread it. The algorithm might then perform well whenever the best label of some component is quickly picked up and spread by the cooperative efforts of several work-groups.

The implementation works in a few different phases. First up is the eligibility phase. Among the labels of pixels inside the work-groups area a choice has to be made on which to start spreading. Obviously a lower label is generally better than a higher one, but it is possible that the lower label cannot be spread further. A pixel is called eligible if any of its connected neighbours are foreground and have a label that is worse than its own. If that label is picked it could then spread to its neighbour, resulting in a slightly better solution.

The eligibility phase can thusly also support the convergence algorithm since we are guaranteed change if there exist eligible pixels.

Pixels with different labels may then be eligible. Since we never checked whether a lower label is right by the eligible pixels, a lower choice of label is wise. We therefore pick the lowest eligible label. It is possible that a more complex choice of eligible label would result in faster execution time.

Implementing a stack that the work-items cooperate on without race conditions is a bit tricky. For instance, one work-item could increment the index outside of the stack, and have some other unit read the data there before being able to write the correct data. Since there is no atomic operation that both increments/decrements an index and writes/reads an array at that index, we need some other solution. It can be solved by requiring that all units read or write during some phase, but not both. Therefore we alternate between a push phase and a pop phase.

During the push phase the algorithm has some pixel that it needs to use to process its neighbouring pixels. The current pixel is either fetched from the global id (x, y) (for the first push phase) or from a previous pop phase. It then looks at the neighbouring pixels and reduces their labels to the spreading label, if it previously was higher than that. If it was higher that pixel is pushed since its neighbours may have higher labels as well. In this way between 0 and 4 new pixels are added to the stack for every unit running during this phase.

During the pop phase we need to get new values of x and y for this work item, so we pop from the stack. It is possible that some work items receive no new pixel. In that case they are marked invalid and does nothing until the next pop phase. If no pixels are on the stack at all, we instead stop this iteration of the larger part of the algorithm. During this phase the stack therefore shrinks by 1 for every work item, though not becoming negative.

Finally we need to actually implement push and pop, since the GPU doesn't actually have stacks and we need it to be atomic. A push first has to manage where in the array to put things, using some kind of index. We manipulate this index using `atomic_inc`, which atomically increments the integer and returns the old value. This is since no two work items will get the same index returned to them when using it except if any `atomic_dec` is used in-between. When they then have their unique index which is inside the bounds of the current stack, they can manipulate the values there in peace. Care has to be taken to revert the index with an `atomic_dec` in case the returned value surpasses the array size. All subsequent work units will also end up not writing to the stack since it is full, leading to no clashes between `atomic_incs` and `atomic_decs`. Pop works the same way but opposite.

4.2.7 Local plus propagation

This algorithm only solves the labeling inside the work-groups respective local area. The *local solving* is a labeling that would be correct if the local area spanned the entire image. But in general it does not imply a global solution since labels across work-group borders may have completely different labels.

The reasoning behind such an algorithms is that we want to be able to use the low latency of local memory access. If we copy over only a small area representing the work-groups area to the local memory it will fit into the limited space available. We can then do a fair amount of processing of the area for little total execution time. It is therefore reasonable to completely solve that area locally.

However, it is thus only usable as a modification of another algorithm. We simply need some other algorithm to cross the work-group borders. In order to implement it as a modification one execution of the kernel is run on every iteration of a converging algorithm. It does not necessarily have to modify the changed flag, since the original algorithm should be responsible for the convergence.

Unfortunately we cannot use union-find for this local solving as the root pixels (and other parent pixels) are very likely to be outside the local area. In such a situation there is not much of a point in not simply using neighbour propagation. The stack-based algorithm is believed to have too much overhead for such a local solver, though it is unverified.

We then found that plus propagation performed better than neighbour propagation in the small space inside such a local area. The implementation is very straightforward and pretty much as described in the regular plus propagation.

Chapter 5

Results

Table 5.1 displays the timings of the algorithms on the 512x512 volume of images. As can be seen, linear two-scan performs the best of the CPU algorithms, outperforming one-pass and union-find slightly. The preparation (copying) time for the images on the CPU is fairly constant at approximately 250 microseconds.

On the GPU side the union-find algorithm vastly outperforms most of the others. The Nvidia card performs about twice as well on most algorithms for this size of image. The preparation time is once again fairly constant, with the AMD taking approximately 1200 microseconds, and the Nvidia card taking approximately 750.

The maximum speedup for the GPU against the CPU is $\frac{1232}{745} \approx 1.65$ for the Nvidia card and $\frac{1232}{1253} \approx 0.98$ for the AMD.

Table 5.2 shows the times for the 1024x1024 image set. For this category there is a slightly larger speedup of $\frac{4245}{1930} \approx 2.20$ for the Nvidia card and $\frac{4245}{2051} \approx 2.07$ for the AMD. The preparation time increased by 4 times for the CPU, which is consistent with a 4 times as large image. For the GPU the same number is approximately 2 for the AMD and 2.5 for the Nvidia, which implies a considerable overhead for the memory transfer.

Figure 5.1 displays a summary of the best CPU and GPU algorithms, including the memory transfer time.

Make	Strategy	µs	Std.Dev.	µs prep
CPU	Front back scan	5018	1354	263
	Linear two-scan	1232	449	273
	One-pass	1663	463	265
	Union-find	1490	460	236
AMD	Line editing	60267	44560	1479
	Lookahead line editing	41124	29951	1327
	Neighbour propagation	90982	27073	948
	Neighbour propagation +local	20207	6915	1185
	Plus propagation	23169	13380	1239
	Stack-based	4732	1977	1327
	Union-find	1253	395	1211
	Union-find +local	1377	1144	1289
Nvidia	Line editing	24684	18307	759
	Lookahead line editing	22365	17422	759
	Neighbour propagation	37858	12353	691
	Neighbour propagation +local	19069	6992	776
	Plus propagation	24801	14747	758
	Stack-based	2469	814	759
	Union-find	745	94	762
	Union-find +local	1301	218	757

Table 5.1: Resulting times of the tests ran on the database of images of size 512x512. Shown is times in microseconds and one standard deviation, as well as the difference in time between with and without counting preparation time.

Make	Strategy	µs	Std.Dev.	µs prep
CPU	Front back scan	18388	5267	993
	Linear two-scan	4245	1136	982
	One-pass	7099	2309	1056
	Union-find	5093	1210	964
AMD	Line editing	199934	190150	3427
	Lookahead line editing	141350	134386	2817
	Neighbour propagation	214404	518810	2151
	Neighbour propagation +local	68539	23837	2745
	Plus propagation	167226	224958	3075
	Stack-based	10613	6041	2705
	Union-find	2051	3762	2488
	Union-find +local	2386	3602	2004
Nvidia	Line editing	184501	174393	1887
	Lookahead line editing	149799	131142	1886
	Neighbour propagation	184659	63322	1843
	Neighbour propagation +local	104038	36531	1881
	Plus propagation	287598	419592	1881
	Stack-based	7223	4179	1888
	Union-find	1930	303	1880
	Union-find +local	3890	745	1882

Table 5.2: Resulting times of the tests ran on the database of images of size 1024x1024.

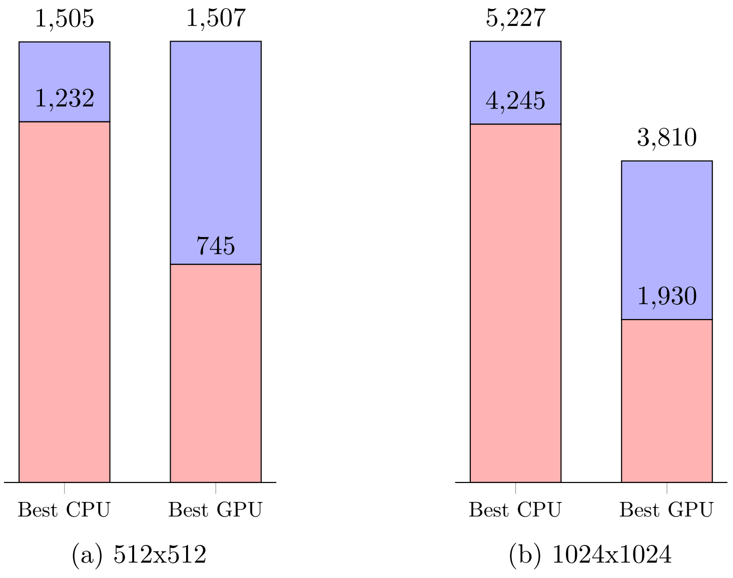


Figure 5.1: Comparison of best results from the CPU and GPU, including memory transfer. Memory transfer is in blue while execution time is in red.

Chapter 6

Discussion

The main point to consider when reviewing the results is that we obviously cannot use the same hardware for the CPU and GPU measurements. This implies that any relative speed difference also depends on how the difference of performance is between the hardware. A practical reasoning is then to attempt to match the cost of the different parts, as well as the cost of operation. Due to this, we have attempted to use consumer hardware that is in the same price range. There is still a bit of a difference, but it should be within a factor of 1.5 (approximately 2000-3000 SEK).

One also need to take into consideration that the used CPU algorithms only used a single core. Modern CPUs often have between 4 to 8, and while there is (as has been shown) some overhead in parallelizing the algorithms, better results are possible. If such an algorithm is used, the slight speedup of the GPUs gained here will be mostly nullified.

Since the larger dataset resulted in a larger difference in speed than the smaller one, a dataset with even larger images might show even better results. Unfortunately, we could not find such a dataset with quality images. Additional research might be required to completely explore that possibility.

While linear two-scan was the fastest algorithm, it also required the most memory. It required three additional tables to resolve label equivalence which required prior knowledge of the image to specify their sizes [7]. Among others the amount of background pixels present in the image as two of the tables were recommended to take on the same size. The safest option is to have them be the same size as the input image. If memory is a limited resource union-find might be more reasonable.

The times of the slower GPU algorithms may look erroneous, but they are simply a bit naive. There is, of course, a slim possibility that we have made mistakes in the coding of certain algorithms. However, the results in execution time are reasonable, which lessens this possibility.

We didn't manage to create any algorithms focusing on local memory that were better than the regular global ones. This is likely due to the problem at hand not having enough work to be done in such a local area, leading to low reusability. And if the data is brought from the global memory into the local one only to quickly be brought back, there is nothing to be gained from using local strategies. There is also a clear trend in the results where the AMD performs better than the Nvidia on the implemented local algorithms. This means that there is also the possibility that some local algorithm would perform better than global one on the AMD, but not the Nvidia. But unless more complex strategies are used such that more reusability is possible, we cannot recommend focusing on that for this particular problem.

The reasonable GPU algorithms appear to all scale approximately linearly. This means that synchronization overhead of the parallel algorithms is not too great for this problem. That implies that we likely could use extra power from even greater parallelism, if such hardware is popular in the future. It may then be more worthwhile in the future to use GPGPU for CCL.

Chapter 7

Conclusions

The difference in execution time is fairly insignificant. Without counting preparation time a speedup of approximately 2 was achieved, and with preparation time it is instead approximately equal. This means that there is little point to transfer the data to the GPU memory in order to perform CCL. However, it also means that there is little reason to do the opposite, if the data was originally on the GPU.

The union-find algorithm was fairly easy to implement on both types of hardware. Since it also performed very well on both, we note that the different versions do not require considerably different efforts of coding. All in all, our recommendation is that the implementation is for the device the data is likely to be on from the start.

So whether it is worthwhile to use GPGPU techniques to solve the CCL depends purely on whether the data is on the GPU already.

Bibliography

[1] Ahmed Abubaker et al. “One scan connected component labeling technique”. In: *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on.* IEEE. 2007, pp. 1283–1286.

[2] Jiři Barnat et al. “Computing strongly connected components in parallel on CUDA”. In: *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International.* IEEE. 2011, pp. 544–555.

[3] NVIDIA Corporation. *CUDA Parallel Computing Platform.* 2016. URL: `http://www.nvidia.com/object/cuda_home_new.html` (visited on 02/22/2016).

[4] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. “A comprehensive performance comparison of CUDA and OpenCL”. In: *Parallel Processing (ICPP), 2011 International Conference on.* IEEE. 2011, pp. 216–225.

[5] Khronos Group. *The open standard for parallel programming of heterogeneous systems.* 2016. URL: `https://www.khronos.org/opencl/` (visited on 02/22/2016).

[6] K. A. Hawick, A. Leist, and D. P. Playne. “Parallel Graph Component Labelling with GPUs and CUDA”. In: *Parallel Computing* 36.12 (Dec. 2010), pp. 655–678. DOI: `10.1016/j.parco.2010.07.002`. URL: `www.elsevier.com/locate/parco`.

[7] Lifeng He, Yuyan Chao, and Kenji Suzuki. “A linear-time two-scan labeling algorithm”. In: *Image Processing, 2007. ICIP 2007. IEEE International Conference on.* Vol. 5. IEEE. 2007, pp. V–241.

[8] Wim H Hesselink, Arnold Meijster, and Coenraad Bron. “Concurrent determination of connected components”. In: *Science of Computer Programming* 41.2 (2001), pp. 173–194.

[9] Victor W Lee et al. “Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU”. In: *ACM SIGARCH Computer Architecture News.* Vol. 38. 3. ACM. 2010, pp. 451–460.

[10] Petr Matas et al. “Parallel algorithm for concurrent computation of connected component tree”. In: *Advanced Concepts for Intelligent Vision Systems*. Springer. 2008, pp. 230–241.

[11] Aaftab Munshi et al. “The opencl specification”. In: *Khronos OpenCL Working Group* 1 (2009).

[12] VMA Oliveira and RA Lotufo. “A study on connected components labeling algorithms using GPUs”. In: *Workshop of Undergraduate Works, XXIII Sibgrapi, Conference on Graphics, Patterns and Images, September*. 2010.

[13] Nobuyuki Otsu. “A threshold selection method from gray-level histograms”. In: *Automatica* 11.285-296 (1975), pp. 23–27.

[14] Azriel Rosenfeld and John L Pfaltz. “Sequential operations in digital picture processing”. In: *Journal of the ACM (JACM)* 13.4 (1966), pp. 471–494.

[15] Jyothish Soman, Kishore Kothapalli, and PJ Narayanan. “Some GPU algorithms for graph connected components and spanning tree”. In: *Parallel Processing Letters* 20.04 (2010), pp. 325–339.

[16] Kenji Suzuki, Isao Horiba, and Noboru Sugie. “Linear-time connected-component labeling based on sequential local operations”. In: *Computer Vision and Image Understanding* 89.1 (2003), pp. 1–23.

[17] Allan G. Weber. *The USC-SIPI Image Database*. Tech. rep. Los Angeles, CA 90089-2564 USA, 3740 McClintock Ave: University of Southern California, Signal and Image Processing Institute, Department of Electrical Engineering, Oct. 1997.