GPU image processing

suite of techniques of image processing

University of Catania

Lecturers: Dario Allegra, Filippo Stanco

Giorgio Locicero, Daniele Alma

March 8, 2021

Abstract

A lot of image processing techniques can be considered embarrassingly parallel, so an implementation on GPU is straightforward, but others are built in a linear pattern and are inherently sequential, so that the process of parallelization is not as simple.

Most of the times, complex problems can be divided in small problems that share a common pattern of resolution, where known techniques of parallelization are used to implement the sub-problems and rebuild the complex problems by piecing together the solutions.

In this study a suite of techniques for image processing is presented along with a toolkit of executables organized in a GUI for visual testing.

1 Introduction

During this project, a number of requirements needs to be satisfied in order to run the examples, in particular OpenCL needs to be working on the machine (so a platform needs to be present) for the OpenCL build (main build), or CUDA needs to be working (so a compatible GPU is needed as well). Other dependencies need to be satisfied, other requirements are described directly in the repository of the project at `https://github.com/josura/GPU-imageprocesing-suite`.

With the project come a command line interface and a GUI: the command line interface is a binary executable written in C that calls OpenCL kernels, the GUI is a graphical tool that calls the executables for every technique.

2 Interface

Along with the executables, a GUI is provided as an interface to see the results.

The GUI is implemented in QT. the whole project was tested on a Linux computer, so it should work without too many problems if the project is built on a Linux system (especially on Debian-based systems for some libraries, Arch-based distros also work).

3 Morphology

For morphology, only grayscale morphology was considered (because It generalizes the concept, encapsulating binary morphology with the right changes).

Only PNG (RGBA) images are supported(tested), other formats need to have 4 channels(color space for 3 channels and transparency), It must be taken into account that the framework used to read images (`https://github.com/nothings/stb`) will not decode some formats with the same output, and the code used (with the framework associated) won't work the same way.

3.1 Structuring elements

The structuring elements considered in this project are Non-flat and are represented by images. For structuring elements, images are used because the computation behind morphology in GPU (OpenCL in specific) is easier done with images. The structuring elements support **don't care** items (that most of other tools does not support, especially for grayscale morphology) identifiable by transparency channel of the image of the structuring element (0 if the pixel should not be considered for computation, otherwise it will be considered, further development could improve the concept by weighting the pixels based on the transparency of the channel, this possibility is not considered in the main branch of the project).

Some misunderstanding could rise with the definition and implementation of the structuring element, because is not a conventional way of representing it (other authors use null values for don't care items and negative values for items that should not be in the image, but the definition that they give is most of the time strictly binary, with the operations, that will be presented in this presentation, only implemented for 2 levels with binary logic).

3.2 Erosion and Dilation

Erosion and dilation considered in this project are the one described in [1]. Exceptions of theory definition are used for the hit or miss morphological operator, where the structuring element and image will be seen and manipulated to recreate the effect in binary images and in the original theory, everything will be explained in 3.3 .

The definitions of grayscale erosion and dilation are the same as in the theory. Grayscale erosion is defined in the following way:

$$[f \ominus b](x,y) = min_{(s,t)\in b}\{f(x+s,y+t) - b(s,t)\}$$

Grayscale dilation is defined in the following way:

$$[f \oplus b](x,y) = max_{(s,t)\in \hat{b}}\{f(x-s,y-t) + \hat{b}(s,t)\}$$

Grayscale erosion makes the image darker and more defined and grayscale dilation makes the image brighter and slightly blurred.

After the execution of erosion or dilation, values that are not in the normal range of values for images are returned to valid values: this concept introduce some error in the theory because some operations that are **idempotent** lose the idem-potency for multiple applications of the morphological operation.

Along with erosion and dilation as base morphological operators, the implementation of pointwise maximum, minimum, image difference, a modified version of erosion (useful for Hit or Miss) and complement (that in grayscale becomes the negative of an image) was done in OpenCL.

3.2.1 Derived morphological operators

Almost all morphological operators that could be built by a combination of erosion and dilation have been implemented, some of the binary ones have been left out because they do not have a real application in grayscale, others(like hit or miss) have been implemented by changing the theory behind morphological operators in grayscale a little.

Morphological gradient	$[f \oplus b - f \ominus b](x, y)$
Closing ($\bullet$)	$[(f \oplus b) \ominus b](x, y)$
Opening ($\circ$)	$[(f \ominus b) \oplus b](x, y)$
Tophat	$[f - (f \circ b)](x, y)$
Bottomhat	$[(f \bullet b) - f](x, y)$
Hit or Miss ($\otimes$)	$[(f \ominus b_1) \wedge (f^c \ominus b_2)](x, y)$
Geodesic erosion	$\begin{cases} E_g^{(k)}(f) = E_g^{(1)}(E_g^{(k-1)}(f)) \\ E_g^1(f) = (f \ominus b) \vee g \end{cases}$
Geodesic dilation	$\begin{cases} D_g^{(k)}(f) = D_g^{(1)}(D_g^{(k-1)}(f)) \\ D_g^1(f) = (f \oplus b) \wedge g \end{cases}$

Table 1: Morphological operators implemented

Presented in this project, the following operations are implemented:
for the properties of these operators(along with base operators) refer to [1].

3.3 Hit or Miss(EXPERIMENTAL)

For the hit or miss operation, a different type of erosion is used, where values are shifted by $-maxvalue$, where maxvalue is the maximum intensity of a channel in an image, therefore the new range of values is $[-maxvalue, 0]$ where 0 values represent white in grayscale and $-maxvalue$ represent black (for binary images $[-1, 0]$).

This representation is not possible with normal images because values are in the positive range and negative values are not accepted: to overcome this obstacle, a new method for erosion was coded to sum the value of the structuring element (not subtract it) because the structuring element image is seen as formed with negative values ($SE_f = -SE$ and the same structure described in 3.2 for the use of the operations and 3.1 for the structuring element specifications).

This operation was not thought to be made for grayscale images, but this implementation was built with theory in mind to implement something that does what Hit or Miss does in binary images.

3.4 Geodesic operations and image reconstruction

For the topic of image reconstruction, geodesic erosion and dilation have been implemented.

The image starting point for image reconstruction is called a **marker**. Geodesic morphological operators use a mask image to limit the erosion or dilation of the recursive call; for erosion, pixel values cannot go lower than the pixels in the mask image, while for dilation they cannot exceed the values in the mask.

Geodesic erosion is defined as:

$$\begin{cases} E_g^{(k)}(f) = E_g^{(1)}(E_g^{(k-1)}(f)) \\ E_g^1(f) = (f \ominus b) \vee g \end{cases}$$

where $\vee$ is the operator of pointwise maximum of two images/signals.

With the same structure as the operation described above, Geodesic dilation is defined as:

$$\begin{cases} D_g^{(k)}(f) = D_g^{(1)}(D_g^{(k-1)}(f)) \\ D_g^1(f) = (f \oplus b) \wedge g \end{cases}$$

where $\wedge$ is the operator of pointwise minimum of two images/signals.

It is to be noted that $E_g^0(f) = D_g^0(f) = f$

The recursive calls of geodesic erosion and dilation are tail recursive and can be translated to a cycle (in fact, in the implementation a cycle is used).

3.5 Performance analysis for morphology

For the experimental analysis to see the performance of the parallel algorithms used, two images were used, both with the same number of channels (RGBA) but with different size. The algorithms are compared with some famous implementation of morphological operators in Matlab. Measure and experimental data was taken on a computer with a Ryzen 4600H CPU and an NVIDIA GTX 1650 GPU.

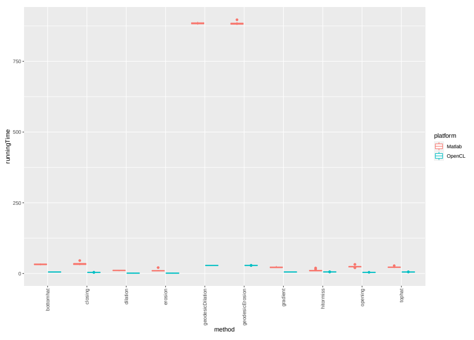

Figure 1: Times for 1920x1080 image

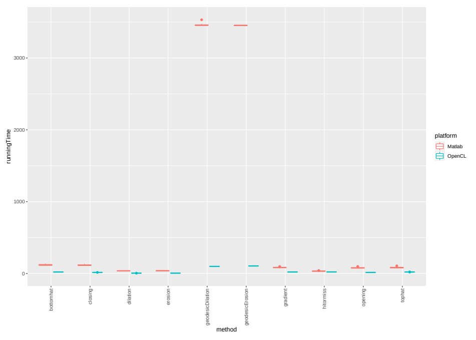

Figure 2: Times for 3840x2160 image

As it can be seen from Figure 1 and 2, even with a naive implementation of morphological operators, the running times are always lower than the implementations from Matlab libraries. It should be noted that the image reconstruction that Matlab does is more complex, but the implementation used in OpenCL (that alternates phases of geodesic erosion and dilation or uses only one of these) can give more or less the same results of reconstruction if applied correctly.

4 Dithering

Dithering is a noise signal added to a signal on purpose, usually before quantization: when an image is quantized, all color levels in a certain range get assigned to the same level and the result has a color-banding effect. Dithering gives the illusion of spatiality and color gradients, even with a low number of levels.

4.1 Random dithering

Random dithering generates a random signal and adds it to the image. Usually, if P is the quantization step, a value between $-\frac{P}{2}$ and $\frac{P}{2}$ is randomly chosen, so that pixels that would have been assigned to the same level L will sometimes get assigned to different levels.

To implement random dithering on GPU, the MWC64X RNG[2] was used: it is overall a very efficient RNG except for the seeding, which was customized by passing 8 seeds from the host to the device and combining them with the work-item global id with various operations.

Each work-item generates 4 different random numbers between $-\frac{P}{2}$ and $\frac{P}{2}$ and sums it to the RGBA values, though alpha is restored after quantization.

4.2 Ordered dithering

Ordered dithering alters the image based on fixed values in a matrix. The matrix used is the **Bayer Matrix**, which is defined recursively for sizes which are powers of 2 as follows:

$$M_2 = \begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix}$$

$$M_{2n} = \begin{bmatrix} 4M_n & 4M_n + 2 \\ 4M_n + 3 & 4M_n + 1 \end{bmatrix}$$

The new image value is:

$$I'(x, y) = I(x, y) + w * (\frac{1}{m^2} M(x \mod n, y \mod n) - 0.5)$$

Usually, a fixed size is chosen and the matrix is computed recursively beforehand. This would be highly inefficient for the GPU, so the corresponding matrix element is computed on the fly by each work-item using an approximation by Joel Yliluoma[3]:

$$M(i, j) = \frac{bit\_reverse(bit\_interleave(bitwise\_xor(i, j), i))}{n^2}$$

4.3 Performance analysis for dithering

The performances of the GPU implementations of random and ordered dithering kernels were compared to two simple Matlab implementations. Measure and experimental data was taken on a computer with a Ryzen 5600X CPU and a Radeon RX 6800 GPU.

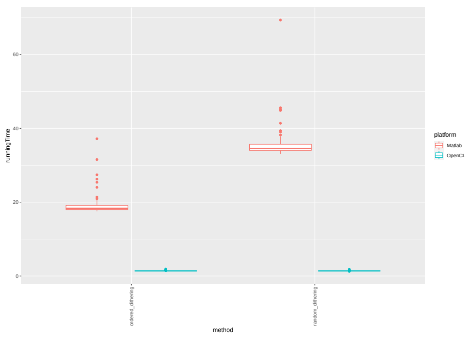

Figure 3: Times for 1920x1080 image

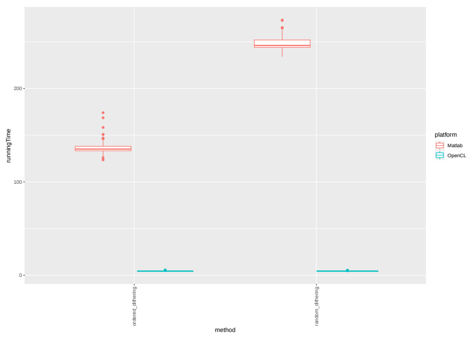

Figure 4: Times for 3840x2160 image

5 Segmentation

Image segmentation is a process in which an image is partitioned in disjoint regions, so that for a certain predicate Q all adjacent regions R_i and R_j:

$$Q(R_i) = TRUE$$

and

$$Q(R_i \cup R_j) = FALSE$$

The main strategies used for segmentation are:

- Edge-based: edges define region borders.
- Thresholding: by thresholding with an appropriate value, a good enough representation of the image regions can be found (mainly if there are just two regions).
- Region growing: if a similarity criteria between pixels is given, a single region can be found by starting with a seed pixel and consequently expanding the region by checking similarity with the 4 or 8 adjacent pixels. This can be repeated multiple times to find all regions in an image.

5.1 Edge detection

5.1.1 Laplacian of Gaussian

Usually, to find edges of a given image, a convolution is applied between the image and a kernel matrix like Sobel. A more accurate result can be obtained with Gaussian smoothing and by considering edges on both sides: this is both done by a LoG (Laplacian of Gaussian) convolution kernel, which can blur the image and find edges at the same time. The implementation uses a fixed 5x5 kernel size and computes the kernel values by sampling a LoG function with a standard deviation value passed as a parameter.

$$LoG(x,y) = \frac{x^2 + y^2 - 2\sigma^2}{\sigma^4} * e^{\frac{-(x^2+y^2)}{2\sigma^2}}$$

The chosen default value for σ is 0.5. A generic convolution kernel and method were implemented on the device code, so that they can be used with any convolution kernel matrix of any size. The kernel matrix is passed to the device constant memory, so that all work-items can access it efficiently.

5.1.2 Canny's algorithm

Canny's algorithm is a popular edge detector that tries to find all real edges and discards those that are considered not strong enough. First of all, images are converted to grayscale: the method is mainly used with a single channel. There are three main steps to the actual algorithm and each one of those was implemented in a separate kernel.

Derivative of Gaussian The DroG operator consists in convolution kernels generated by partial differentiation with respect to each direction of a Gaussian function and sampling each result. The convolution kernel used in the implementation is computed on host by sampling the derivative with respect to x with a standard deviation value passed as a parameter:

$$DroG\_x(x,y) = -\frac{x}{\sigma^2} * e^{\frac{-(x^2+y^2)}{2\sigma^2}}$$

The chosen default value for σ is 0.7. The kernel for the other direction is just the first one transposed, so for space efficiency a "transposed" method was implemented, that just applies the convolution using the kernel matrix element at position (j,i) instead of (i,j). Both convolutions are applied to the image simultaneously and the two results $v = (x, y)$ for each pixel are used to compute the magnitude and the potential edge angle.

The magnitude is computed with the norm of the vector v and is saved in a new image that is used in the following step.

The angle is computed with:

$$edge\_angle = arctg(\frac{y}{x})$$

The angle is then converted to degrees and "quantized" to either 0, 45, 90 or 135 degrees to simplify the next step.

Non-maxima suppression Non-maxima suppression consists in removing pixel magnitudes that are not local maxima for the DroG operator: this can be checked using the angle obtained in the previous step.

- If the angle is 0 degrees, the edge is perfectly vertical and the left and right neighbors magnitudes must be smaller than the current pixel magnitude.
- If the angle is 45 degrees, the edge is diagonal and directed from south-east to north-west and the top-right and bottom-left neighbors magnitudes must be smaller than the current pixel magnitude.
- If the angle is 90 degrees, the edge is perfectly horizontal and the top and bottom neighbors magnitudes must be smaller than the current pixel magnitude.

- If the angle is 135 degrees, the edge is diagonal and directed from south-west to north-east and the top-left and bottom-right neighbors magnitudes must be smaller than the current pixel magnitude.

If a point is a local maximum, the magnitude is copied to a new image that is used in the next and final step.

Hysteresis and edge-linking Hysteresis is a thresholding method that uses two thresholds, with one bigger than the other.

Suppose $high\_threshold > low\_threshold$ and $magnitude$ is the corresponding work-item pixel magnitude.

- If $magnitude \geq high\_threshold$, the pixel is part of a strong edge and is colored white.
- If $magnitude \leq low\_threshold$, the pixel is considered to not be part of an edge and is colored black.
- If $low\_threshold < magnitude < high\_threshold$, the pixel is considered a strong edge if at least one of its 2 neighbors in the same direction is part of a strong edge and has the same angle. This is the linking phase.

The linking method is very efficient, but it might not be good enough in some cases, with it not being able to link distant pixels. More effective linking methods can be applied by checking more distant neighbors or applying this "weak" linking phase many times, but both methods can turn out to be really inefficient, especially for a parallel implementation.

5.2 Otsu's method

5.2.1 Introduction

Otsu's method is a segmentation thresholding method that finds the optimal threshold by consequently dividing pixels in two disjoint classes: if L is the maximum level value, Otsu divides it into class 1 with values in $[0, k]$ and class 2 with values in $[k+1, L-1]$. For each possible partition corresponding to a level k, the interclass variance is computed which is defined as:

$$\sigma_B^2 = P_1(m_1 - m_G)^2 + P_2(m_2 - m_G)^2$$

P_1 is the probability of a pixel being in class 1, P_2 is the probability of a pixel being in class 2, m_1 and m_2 are the color means of each class and m_G is the global mean of the image. The ideal threshold k is the one corresponding to the maximum interclass variance.

After getting the histogrammed frequencies for each color, the needed values for the interclass variance of partition k can be computed as:

$$P_1(k) = \sum_{i=1}^{k} p_i$$

$$P_2(k) = 1 - P_1(k)$$

$$m_1(k) = \frac{\sum_{i=1}^{k} ip_i}{P_1(k)}$$

$$m_2(k) = \frac{m_G - m_1(k)}{P_2(k)}$$

A more efficient way of computing the interclass variance is the following:

$$\sigma_B^2 = \frac{m_G P_1 - m}{P_1(1 - P_1)}$$

With this formula there is no need to compute P_2 and m_2.

5.2.2 Implementation

Like Canny's algorithm implementation, the image is first converted to grayscale. Otsu's method is not embarrassingly parallel: to get color frequencies the histogram is needed, after that cumulative sums need to be computed and stored, finally after computing the interclass variance a maximum between all values needs to be found. This is really straightforward on a sequential implementation, in which the histogram computation just cycles through all pixels, the value of the sums can be easily memorized at each step and the maximum can be computed with simple checks.

The parallel implementation does the following steps:

- Compute the histogram: each work-item is assigned to a pixel and the corresponding level occurrences in the histogram buffer are incremented using the OpenCL *atomic_inc*() method.

- Scan: scan is the procedure which, given a vector with n values, returns a new vector with n values in which at each position k there is the cumulative sum of all elements at position $i \leq k$. After normalizing the histogram to get the frequencies, the computation of all class means and probabilities is, in fact, a scan.

- Compute interclass variance: the more efficient interclass variance formula involves a lot of products and divisions between small numbers, which can turn out to be really imprecise with floating-point arithmetic. This is solved by applying the natural logarithm on both sides, so that the formula turns into simple sums and subtractions. Then, the actual value of the interclass variance will be:

$$\sigma_B^2 = exp(2\log{(m_G P_1 - m)} - \log{(P_1)} - \log{(1 - P_1)})$$

- Reduction: after each work-item computes its corresponding interclass variance, a reduction with local memory to find the maximum is applied.

While this is a really convoluted procedure for just 256 colors, it can be easily generalized when there are many more levels. Also, the histogram computation with atomics turned out to be really efficient, even more than the sequential CPU counterpart.

5.3 Region growing

5.3.1 Introduction

Region growing is a segmentation method that finds one region at a time, starting from seed pixels. The seed pixel is assigned to a region and checks for similarity on its 4 adjacent pixels: if they are similar, they are set to be a part of the region and they themselves can check for similarity on their neighbors. Two region growing programs were implemented: the first one just creates a single region from a starting pixel which position is chosen by the user, the second one uses random seed pixels to generate a number of regions defined by a parameter. In the second one, the seed pixels for each region are chosen on host with the C standard library function $rand()$. The seed pixel is discarded if it is already part of a region.

5.3.2 Similarity between pixels

The criteria used for checking similarity between two pixels is ΔE: this defines the perceptive distance between two colors in the $CIELAB$ color space. The RGB values of the image are first converted to the $CIELAB$ color space, then the converted values are used in the region growing kernel. A distance threshold can be passed as a parameter from host: this decides the level of acceptance for a pixel to enter a region.

The default value for the distance threshold was set to 3, which seems to also be the ideal value: smaller values resulted in difficulties to find regions and higher values ended up in excessive overmerging.

5.3.3 Region growing kernel

The region growing kernel is launched multiple times for each region: in the first iteration, the work-item corresponding to the seed pixel will explore is neighbors, in the second iteration its neighbors that got into the region will explore their neighbors, and so on.

Along with the converted image, the kernel uses two buffers and a global flag: the two buffers are used to store region labels and explore labels (if a pixel explore label is 1 it can check for similarity with its neighbors), the global flag is a termination flag that is set to 0 on host before each launch of the kernel, and by work-items to 1 every time a pixel tries to check for similarity with its neighbors. This makes it possible to stop launching instances of the kernel when the flag has not been set to 1 by any work-item.

Finally, after region growing, the region labels are used to color the image: each pixel is colored with its corresponding seed pixel color.

5.3.4 Advantages and disadvantages

Region growing, with the right distance threshold, can be really efficient in segmenting the image and well-defined edges between regions can often be seen.

Unfortunately, it is really expensive, even on GPU and overmerging is common because similarities are not checked for all pixels in the region, only adjacent ones. This can be partially mitigated with other region growing techniques based on Statistical Region Merging.

5.4 Performance analysis for segmentation

The performances of the GPU implementations of Canny's algorithm, Otsu's method and the LoG convolution kernel were compared to the Matlab implementations in the image processing libraries. Measure and experimental data was taken on a computer with a Ryzen 5600X CPU and a Radeon RX 6800 GPU.

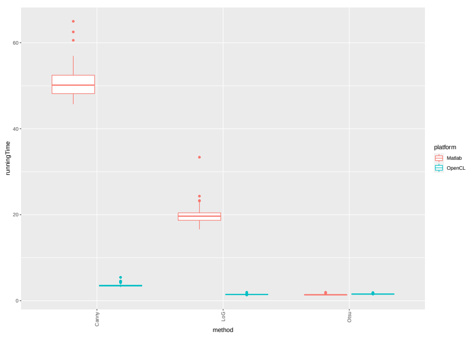

Figure 5: Times for 1920x1080 image

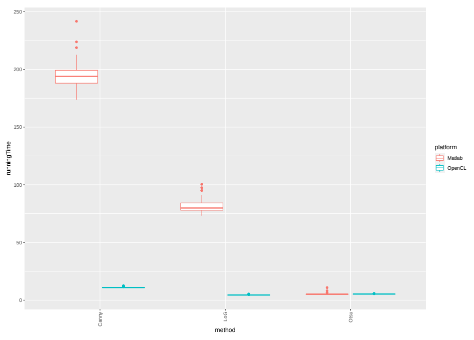

Figure 6: Times for 3840x2160 image

Regarding region growing with multiple regions, here is the scatterplot (number of regions, running time) of multiple executions on Lena, from 1 region to 60.

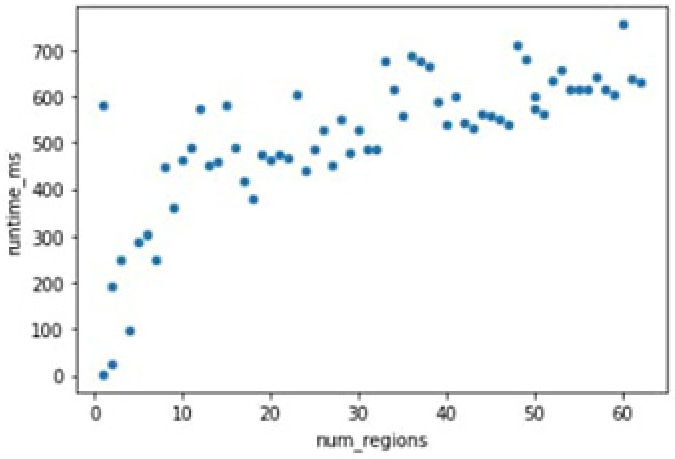

Figure 7: Running times depending on the number of regions

The running time initially increases very quickly, then it tends to stabilize after about 20 regions. The turning point is probably the number of big regions in the image.

6 Conclusion

During this project, the most common algorithms for image processing were discussed and implemented accordingly to specification of [1][5] and [4][6].

At the end of all the experiments, the results were clear, a good GPU implementation is always better than a CPU version, especially for this type of algorithms that rely on an inherently parallel infrastructure, easily exploitable with GPGPU.

For **morphology**, the structuring elements were passed as images: this consideration brings some limitations, especially on the values of the structuring element (where every intensity is limited by the specifications of the image); however, with this concept, the possibility of using unconventional structuring elements is intriguing (along with the possibility of using transparency as don't care items).

Some morphology operations were not thought for grayscale images (Hit or Miss) but, with some addition to the theory to generalize the problem for grayscale images while maintaining the concept behind for binary images, the problem of generalizing and implementing the operation was done with some considerations regarding the choice previously made for structuring element and the basic operations of dilation and erosion, as explained in 3.3.

Operations for image reconstruction were implemented as well according to the theory found in [1].

For **dithering**, a GPU RNG such as MWC64X[2] makes a random dithering implementation really straightforward and highly efficient; ordered dithering is just as good with the Bayer Matrix approximation, with few computations needed by each work-item.

For **segmentation**, it really depends on the method: Otsu's method might not be worth it but it still holds up well, Canny's algorithm and Laplacian of Gaussian that have convolution steps are really efficient, even if memory accesses might not be contiguous, because the OpenCL image API handles all the caching. Region growing is generally too expensive, even with a parallel implementation: for segmentation purposes, the other methods are preferred; it certainly has other uses, like the magic wand selection tool often found in software like Photoshop.

7 Future outlooks

For **morphology**, the theory is most of the times not formalized, authors of other algorithms that expand the concept of morphology do not define an explicit and straightforward infrastructure to use for most problems that could arise during research, but focus only on a type of problem without generalizing the concept.

For future works, additional theory could be added to the original articles, along with better implementations and modern theories about dynamic morphology and applications.

Most of the algorithms used in this project were not optimized, future works with newer theories will be optimized.

For **dithering**, an error-diffusion dithering implementation was considered since it is generally considered the best method, but the algorithm is not parallel-friendly: one might try various approaches like turn-based or row-based, but performance will suffer greatly compared to the other two methods anyways. Ordered dithering is probably good enough for most situations.

For **segmentation**, the similarity criteria in region growing could be improved by considering information about the whole region; other clustering-based approaches might be interesting, like a parallel k-means that finds the most similar colors in the image.

References

[1] Rafael c. Gonzalez, Richard E. Woods, *Digital Image Processing*, Pearson, 4th edition, 2018.[667-671,688-691]

[2] D. B. Tomas, *MWC64X RNG.*

[3] Joel Yliluoma, *Joel Yliluoma's arbitrary-palette positional dithering algorithm.*

[4] Akarun, Lale and Yardunci, Y and Cetin, A Enis *Adaptive methods for dithering color images*

[5] *Mathematical morphology in image processing*, Dougherty, Edward, 2018, CRC press

[6] *Image analysis using mathematical morphology*, Haralick, Robert M and Sternberg, Stanley R and Zhuang, Xinhua, IEEE transactions on pattern analysis and machine intelligence, 1987, IEEE