Porting the Laplacian filtering application to the Vulkan API using OpenCL and OpenGL programming models

Υλοποίηση του Laplacian filter στην προγραμματιστική διεπαφή Vulkan με χρήση των προγραμματιστικών μοντέλων OpenCL και OpenGL

Gkeka Maria Rafaela

Supervisor: Assoc. Prof. Bellas Nikolaos

2nd committee member: Assoc. Prof. Antonopoulos Christos

A Thesis submitted in fulfillment of the requirements
for the degree of Diploma Thesis
in the
Computer Systems Lab (CSL)
Department of Electrical and Computer Engineering
University of Thessaly
Volos, Greece

October 2018

Porting the Laplacian filtering application to the Vulkan API using OpenCL and OpenGL programming models

Υλοποίηση του Laplacian filter στην προγραμματιστική διεπαφή Vulkan με χρήση των προγραμματιστικών μοντέλων OpenCL και OpenGL

Gkeka Maria Rafaela

Supervisor: Assoc. Prof. Bellas Nikolaos

2nd committee member: Assoc. Prof. Antonopoulos Christos

Εγκρίθηκε από την διμελή εξεταστική επιτροπή την ....

....	....
Ν. Μπέλλας	Χ. Αντωνόπουλος
Αναπληρωτής Καθηγητής	Επίκουρος Καθηγητής

Dedicated to

my mother

Υλοποίηση του Laplacian filter στην προγραμματιστική διεπαφή Vulkan με χρήση των προγραμματιστικών μοντέλων OpenCL και OpenGL

Περίληψη

Όταν μιλάμε για GPUs στην ουσία αναφερόμαστε σε εξειδικευμένες μονάδες εκτέλεσης που δημιουργήθηκαν με σκοπό την επιτάχυνση διαδικασιών που αφορούν γραφικά. Στις μέρες μας, η GPU αποτελεί βασικό μέλος των σύγχρονων υπολογιστικών συστημάτων, τα οποία μπορούν να προσφέρουν σημαντικά οφέλη απόδοσης στις παράλληλες εφαρμογές επεξεργασίας δεδομένων. Τα OpenCL, OpenGL και Vulkan είναι προγραμματιστικές διεπαφές για GPUs. Σκοπός αυτής της διπλωματικής εργασίας είναι η διερεύνηση των διαφορών μεταξύ των διαφορετικών API εφαρμόζοντας ένα σύνθετο αλγορίθμο επεξεργασίας εικόνας. Η OpenCL αποτελεί μία τυποποιημένη διεπαφή για παράλληλες υπολογιστικές εφαρμογές που χρησιμοποιείται σε πάρα πολλές υλοποιήσεις οι οποίες εκτελούνται σε ετερογενείς πλατφόρμες. Η OpenGL συνήθως αλληλεπιδρά με τη μια GPU με στόχο την επιτάχυνση των γραφικών. Το Vulkan είναι η ένωση των προηγούμενων δύο APIs. Αποτελεί τη νέα γενιά διεπαφών λοω-οερηεαδ, ςροσσ-πλατφορμ γραφικά και ςομπυτε ΑΠΙ. Το Laplacian Filter, είναι μια εφαρμογή επεξεργασίας εικόνων που εστιάζει στην αναγνώριση ακμών της εικόνας. Συγκεκριμένα, επεξεργάζεται τον τόνο ή τις λεπτομέρειες της εικόνας εισόδου εφαρμόζοντας ένα σύνολο ισχυρών επιδράσεων χωρίς να καταστρέφει την εικόνα. Εφαρμόζωντας πληθώρα βελτιστοποιήσεων στις παραλληλισμένες εκδόσεις της εφαρμογής, έχουμε την δυνατότητα εξέτασης της απόδοσης μιας GPU . Τα αποτελέσματα της συγκεκριμένης εφαρμογής δείχνουν ότι η OpenCL δίνει τον καλύτερο χρόνο εκτέλεσης.

Porting the Laplacian filtering application to the Vulkan API using OpenCL and OpenGL programming models

Abstract

A Graphics Processing Unit (GPU) is a dedicated parallel processor optimized for accelerating graphical computations. Nowadays, GPU has become one of the most important components in modern computer systems, that can provide significant performance benefits to data parallel applications. OpenCL, OpenGL and Vulkan offer three different interfaces for programming GPUs. The target of this thesis is to investigate the differences between Application Programming Interfaces (APIs) by porting a complex image processing algorithm. OpenCL is used by a huge variety of software projects which execute across heterogeneous platforms in order to provide a standard interface for parallel computing. OpenGL is typically used to interact with a GPU to achieve hardware-accelerated graphics rendering. Vulkan is a union of the previous two APIs, new generation low-overhead, cross-platform graphics and compute API. The Laplacian Filter is an edge aware image processing application that produces a wide range of strong effects for both detail manipulation and tone mapping of an image without corrupting the image. This thesis examines the performance of a GPU in different optimizations tested in the parallelized versions of the application. The results of the specific application show that the OpenCL gives better execution time.

Acknowledgements

First and foremost, I would like to express my immeasurable appreciation and deepest gratitude to my advisor, Prof. Nikolaos Mpellas, for his continuous support, and motivation. His guidance helped me during my research and writing of this thesis. I am deeply grateful to Prof. Christos Antonopoulos for his support, valuable comments and technical advices on the department of this thesis.

To my friends, thank you for all your support, understanding and love during the past five years. Our experiences will accompany me throughout my life.

To my parents and my sister, thank you for encouraging me and supporting me in so many ways to do my best and always believing in me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Since the early days of computing, there has been an obvious desire to increase the processor performance. The key enabler for performance improvement is to exploit parallelism. Instruction-level parallelism (ILP) is a measure of how many of the instructions in a computer program can be executed simultaneously. A hardware design approach to achieve increased ILP is pipelining, a technique that divides the processing of an instruction into a series of sequential steps and increases the instruction throughput of a Central Processing Unit (CPU). The next step of hardware improvements was a superscalar processor that is capable of issuing multiple instructions for execution during a clock cycle because of multiple execution units. Today's superscalar processors mostly rely on hardware techniques such as dynamic instruction scheduling, advanced branch prediction, optimizing caches, and so on. Another important method for improving performance is static ILP in which the compiler decides which instructions to execute in parallel. [1] [2]

With instruction-level parallelism at its limit because of "Brick wall" [3], we turn to other forms of parallelism such as thread-level parallelism (TLP) which exploits parallelism across different threads of execution. The improvement in performance gained by the use of a multicore processor depends on the implementation of the software algorithms which are used. The applications executed in this type of systems use a method of parallelizing whereby a master thread forks a specified number of slave threads and the system divides a task among them. A multicore system relies heavily on memory bandwidth because of more threads go to memory more often but this can hide the latency. Furthermore, the concurrency level is dependent on the number of cores.

The limitations of multicore processors led to the need for many-core processors. Many-core processors are distinct from multicore processors designed for a high degree of parallel processing and for higher throughput. Graphics Processing Units (GPUs) may be considered a form of manycore processor, only being suitable for highly parallel code. GPUs have a large number of simple processing units for massively parallel calculation for rendering graphics on a computer quickly, and this has generated the idea of general purpose GPU (GPGPU) computing.

Because of the ability to use GPUs for general purpose computing, many interfaces created by companies such as Apple developed a specification for parallel programming of heterogeneous systems called Open Computing Language (OpenCL). OpenCL is an extension to existing languages and provides a standard interface for parallel computing using task-based and data-based parallelism. Code that gets executed parallel on the GPU is called a kernel [4].

Open Graphics Library (OpenGL) is a cross-language, cross-platform Application Programming Interface (API) for 2D and 3D rendering. The API is typically used to interact with a GPU, to achieve hardware-accelerated rendering. It is supported on essentially every GPU. OpenGL execution takes place as a pipeline with both programmable and state-driven, fixed hardware stages. The programmable stages are called shaders and each of them has a separate set of inputs and outputs. One of the shader types is the compute shader which takes some values as input and it is up to the shader itself to fetch that data. Shaders of this type used by OpenGL API to execute computations [5].

OpenGL and OpenCL is managed by the non-profit technology consortium Khronos Group. Khronos also fully define a cross-API intermediate language SPIR-V with native support for shader and kernel features used by APIs such as OpenCL, OpenGL and Vulkan. SPIR-V is the first multi-API intermediate language for parallel compute and graphics [6].

At Game Developers Conference (GDC) 2015, Khronos Group announced the new Vulkan API, which was initially referred to as the ”next generation OpenGL initiative”. When releasing OpenCL version 2.2, Khronos announced that OpenCL would be merging into Vulkan in the future. Vulkan is a low-overhead cross-platform 3D graphics and compute API. Vulkan targets high-performance realtime 3D graphics applications such as video games and interactive media across all platforms [7].

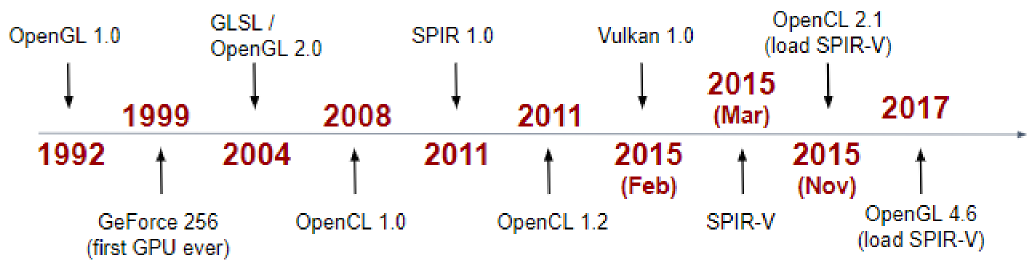

Figure 1.1: API's versions timeline

1.2 Problem statement & Contributions

The purpose of this thesis is to implement a complex image processing algorithm, Local Laplacian Filters, in technologies for GPGPU (OpenCL, OpenGL, Vulkan) and compare the details and the differences between the implementations as well as the execution time.

Using the GPU for general-purpose programming is effective, but also challenging exactly like the use of the Vulkan API for compute. Vulkan provides more flexibility in designing applications with better performance and lower energy consumption than was possible using OpenGL. This is one reason that we want to use it for complex compute applications. In contrary to its great progress in game development and graphics design, its use in compute applications is not common. Therefore, applying an application like Local Laplacian Filters in Vulkan API is great of interest.

1.3 An overview of the content

This thesis is organized into 6 chapters, each one of those includes smaller sections and possibly subsections.

Chapter 2, provides a description of the algorithm Local Laplacian Filters by providing more information about the general technique of the Gaussian and Laplacian pyramids.

Chapter 3 presents the introduction in LLF implementations. It describes a single threaded implementation which is based on the initial Matlab code.

Chapter 4 is an important section of the document which describes the methodology which is used to parallelize the initial sequential implementation. It also introduces our OpenCL implementations which are compared with the execution time of sequential implementation. These implementations is the base of the other APIs implementations.

Chapter 5 describes the use of the OpenGL API and the compute pipeline implementations of the LLF algorithm. It introduces OpenGL specific optimizations and methods.

Chapter 6 is an introduction in the next generation Vulkan API and intermediate representation Spir-V. It also presents a Vulkan application of LLF algorithm and uses Spir-V toolchain.

Chapter 7 presents the conclusion which includes the future work.

Chapter 2

The Laplacian filter

The Local Laplacian Filters (LLF) is an edge-aware image processing algorithm based on the direct manipulation of Laplacian pyramids [8]. Given an input image, the algorithm applies detail or tone enhancements. An example of Laplacian filter transformation is shown in Fig. 2.1.

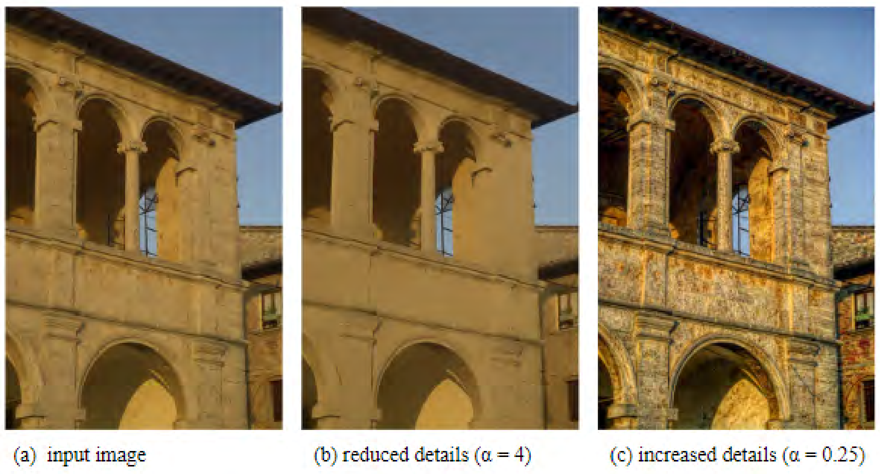
(a) input image
(b) reduced details ($\alpha = 4$)
(c) increased details ($\alpha = 0.25$)

Figure 2.1: LLF algorithm when (b) image details are smoothed, and (c) image details are enhanced

2.1 Background on Laplacian Pyramids

Laplacian Pyramids have been used to analyze a variety of applications such as compression, texture synthesis, and harmonization. In this section, we introduce the steps of Laplacian Pyramids construction. The concept is to scale down the initial image, and then to expand the pyramid images to recreate the input image [9].

The first step of Laplacian pyramids creation is generation of Gaussian pyramids. The Gaussian pyramid is constructed by smoothing the image with an appropriate smoothing filter and then subsampling the smoothed image. The subsequent images are weighed down using a Gaussian average (Gaussian blur) and then scaled down.

Gaussian blur or Gaussian smoothing is a well known transformation in image processing, is commonly used in conjuction with edge detection to reduce the level of noise in the image. It is also used as a pre-processing stage in computer vision algorithms in order to enhance image structures at different scales. It simply blurs the image and reduces the sharpness and the detail by making the transition from one pixel to another very smooth. A Gaussian blur effect is generated by convolving an image with a kernel of Gaussian values. In image processing a kernel (also called convolution matrix or mask) is a small matrix of numbers used to apply effects across an image, such as blurring, sharpening, outlining or embossing. Figure 2.2 shows the output of Gaussian smoothing when the size of the convolution matrix size is 5x5. A larger convolution matrix increases the size of blurring because more pixels are used in calculations for each pixel of the new picture.

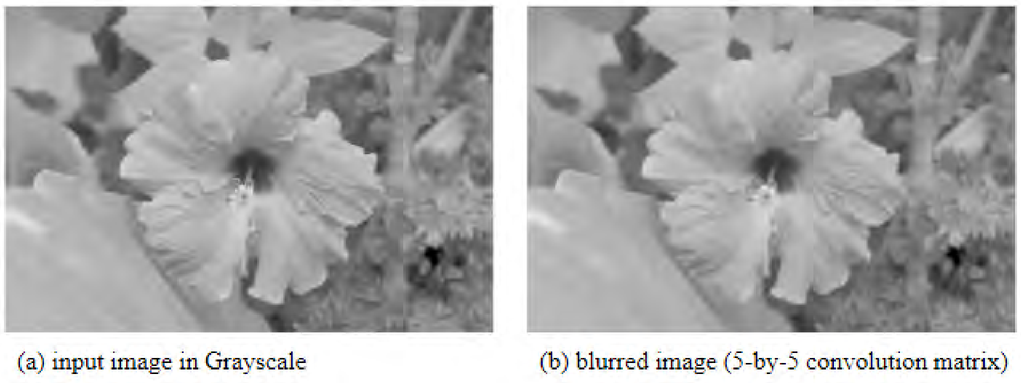
(a) input image in Grayscale
(b) blurred image (5-by-5 convolution matrix)

Figure 2.2: Gaussian convolution blur example with a 5-by-5 kernel which is used to Gaussian pyramid computation

The lowest level (level 0) of the Gaussian pyramid consists of the input image. For the creation of the remaining levels, each pixel value within every level is computed as a weighted average of values in the previous level within 5-by-5 neighbor pixels. Note that each level has a different size in this pyramid (thus, the concept of the pyramid).

After blurring, each lower level of the pyramid is reduced by half (compared with the previous level) by subsampling the pixels of the previous level. If $C * R$ is the size of the initial image (pyramid L0), then the size of the image in level 1 is $(C/2) * (R/2)$. Consequently, a pixel of the new image represents four pixels of the previous pyramid level image.

If we want to create Gaussian pyramids with N levels for an image I, then every level is performed by the following equation, in which REDUCE is a function that represents the process which generates a Gaussian pyramid (Fig. 2.3).

$$G_0 = I \tag{2.1.1}$$

$$G_k = REDUCE(G_{k-1}), \quad k = [1, N] \tag{2.1.2}$$

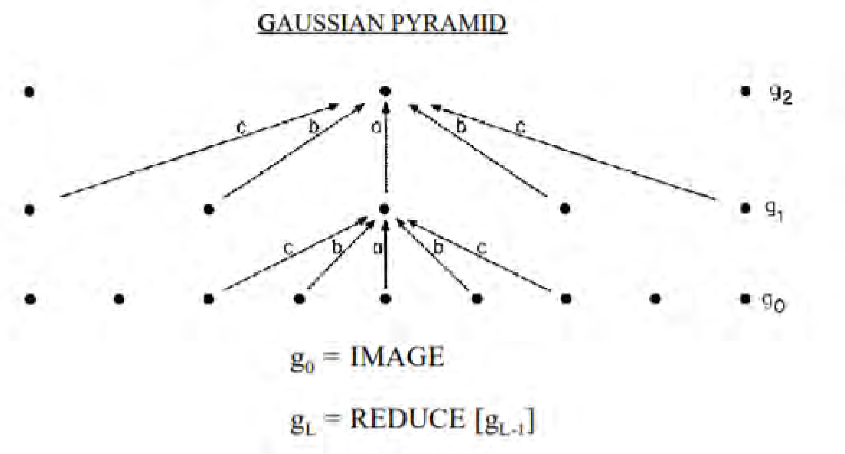

Figure 2.3: Use of blurring and subsampling processes for Gaussian pyramid generation [9]

Then we introduce the EXPAND function, the inverse process of REDUCE, which doubles the size of the image (which takes as a parameter) in every dimension using a smooth kernel. The smooth kernel is applied as we describe previously in the EXPAND process.

$$G_{k,0} = G_k \tag{2.1.3}$$

$$G_{k,l+1} = EXPAND(G_k, l), \quad k = [0, N], l = [0, k] \tag{2.1.4}$$

The parameter l of the function declares the number of expansions we apply. If it has zero value, then we double the dimension size one time. If we apply EXPAND l times to image G_l, we obtain $G_{l,l}$, which is the same size as the initial image G_0.

The Laplacian pyramid (which is similar to the Gaussian pyramid), is a sequence of error images L_0, L_1, ..., L_N (Fig. 2.4) and is equals to the difference image of the blurred versions of Gaussian pyramid between each levels. The smallest level (L_N) is not an error image but equals the smallest level image of the Gaussian pyramid. The N-th level is responsible for the reconstruction of the original image by using the difference images on higher levels.

$$L_N = G_N \tag{2.1.5}$$

$$L_k = G_k - EXPAND(G_{k+1}, 0) = G_k - G_{k+1,1}, \quad k = [0, N-1] \tag{2.1.6}$$

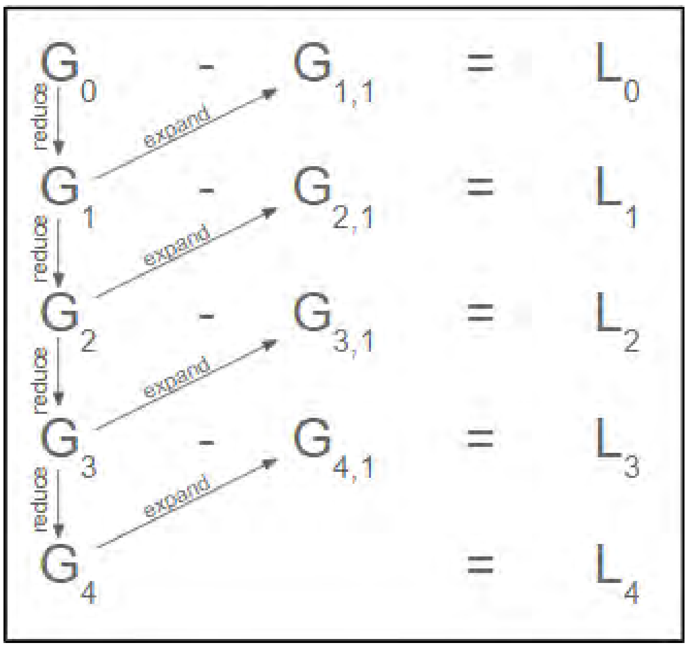

Figure 2.4: Laplacian five levels pyramid generation

The concept of the Laplacian pyramid technique is to save the source image in small structures, apply a variety of transformations and create a new image with the same size. When no transformations are applied, then the new image is similar to the original. Figure 2.5 shows the method without intermediate transformations. If we consider that S is the source image, then is mathematically accepted that is equal to the sum of the fully expanded Laplacian pyramid images.

$$S = G_0 = \sum_{k=0}^{N} L_{k,k} \tag{2.1.7}$$

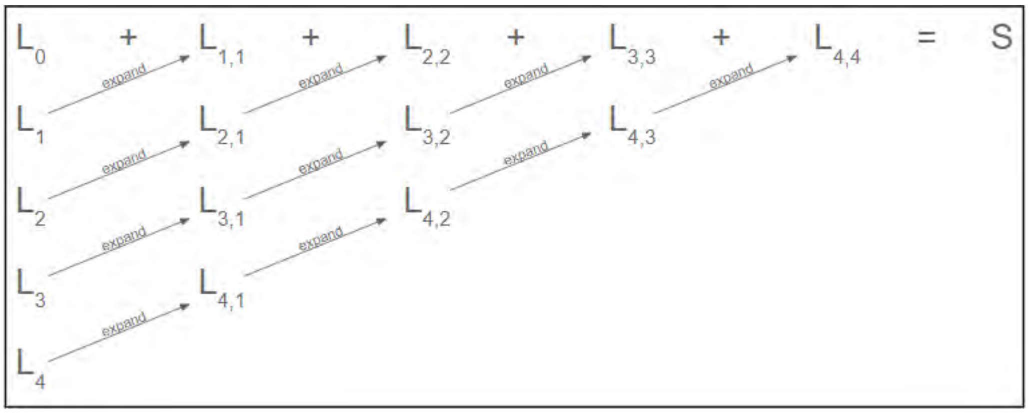

Figure 2.5: Input image regeneration

The Fig. 2.6 shows another approach of initial image recreation, which limits the memory resources and improves the performance of the algorithm.

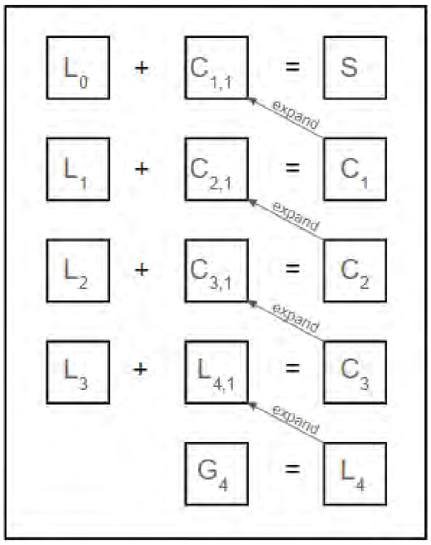

Figure 2.6: Input image regeneration

2.2 The LLF algorithm

The Local Laplacian Filters is a flexible approach to achieve edge-aware image processing through simple point-wise manipulation of Laplacian pyramids. The idea is to find the way that the edges represented in Laplacian pyramids.

The algorithm based on Laplacian pyramid generation. Initially, we calculate the Gaussian pyramid, which is the same as we describe in the previous section 2.1. In the next step, for each pixel in the Gaussian pyramid, we based on its value g_0 for creating a sub-image by remapping a specific range R_0 of the input image using a point-wise function. We consider the resulting image as the base level of a new intermediate pyramid in which we apply the pyramid creation process (subsampling and upsample). From the intermediate Sub-Pyramid, we pick the appropriate pixels for Laplacian pyramid images (Fig. 2.7).

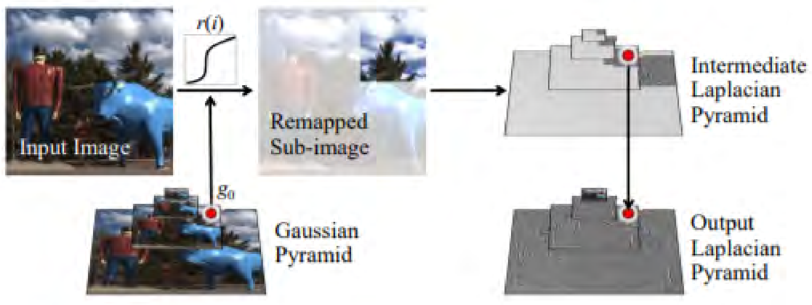

Figure 2.7: Overview of the basic idea of LLF [8]

The algorithm 1 describes the previous analysis of Local Laplacian Filters method. The algorithm takes as input an image S, a parameter σ_r and remapping function r. Let (x_0, y_0) be the position of pixels within level l_0 of the Gaussian pyramid. The use of the σ_r parameter is introduced in the section below.

2.2.1 Remapping and Parameterization

The remapping function is a point-wise function used to create an intermediate image S' for every pixel (x_0, y_0) of every Gaussian pyramid level l_0. The intermediate image has a specific size that depends on the l_0 and the position of the pixel. The

Algorithm 1 LLF [8]

```
1: compute input Gaussian pyramid G[S]
2: for $(x_0, y_0, l_0)$ do
3:     $g_0 \leftarrow G_{l_0}(x_0, y_0)$
4:     determine sub-region $R_0$ needed to evaluate $L_{l_0}(x_0, y_0)$
5:     apply remapping function: $R'_0 \leftarrow r_{g_0,\sigma_r}(R_0)$
6:     compute sub-pyramid $L_{l_0}[R'_0]$
7:     update output pyramid: $L_{l_0}[S'](x_0, y_0) \leftarrow L_{l_0}[R'_0](x_0, y_0)$
8: end for
9: collapse output pyramid: $S' \leftarrow collapse(L_l[S'])$
```

function applied to the original image (W: width, H: height) from which compares the pixels of a $K*K$ window ($K = 3(2^{l_0+2}-1)$) with the (x_0, y_0) element of Gaussian pyramid level l_0. The pixel in position (x_0, y_0) is in the centre of the $K*K$ region, so $3*2^{l_0+1}-2$ elements are located on each side in both directions of the coefficient (x_0, y_0). In the case that the distance of one of the x_0 or y_0 from the input region is less than $K/2$, the intermediate sub-image has smaller size.

Figure 2.8 shows the data used to calculate one pixel of a Laplacian pyramid level image. The intermediate sub-image has the same size with that of the original image window but not the same information. At this step of the LLF algorithm, we process the edge and detail detection and we decide if we need to amplify or to smooth the result. This decision depends on the value of the user provided parameter α.

The most significant parameter is σ_r because it determines the effect of each pixel in the output image. Intensity variations smaller than σ_r should be considered details whereas larger variations should be considered edges. We compare every appropriate pixel of the original image with the pixel of Gaussian pyramid image and we decide if it belongs to edges or details. This decision determines the remapping function r_e or r_d. The remapping function has two versions, one for grayscale and another for color images. In the next chapter, we describe our implementation that uses RGB-information for the images in all the stages of algorithm and the following color remapping equations.

$$r_d(i) = g_0 + unit(i - g_0)\sigma_r f_d(||i - g_0||/\sigma_r) \tag{2.2.8}$$

$$r_e(i) = g_0 + unit(i - g_0)[f_e(||i - g_0|| - \sigma_r) + \sigma_r) \tag{2.2.9}$$

The user also provides two parameters α, β related to f_d and f_e functions. The function f_e (used in 2.2.9) takes α as parameter and is called when a pixel i of the input image gives to the Gaussian pyramid pixel under consideration the edge characteristic. Consequently, the value of the i pixel of the intermediate sub-image depends on parameter α. Figure 2.1 indicates the effect of the parameter. Similarly, f_d is a point-wise function that controls the amplification or attenuation of details and takes β as parameter.

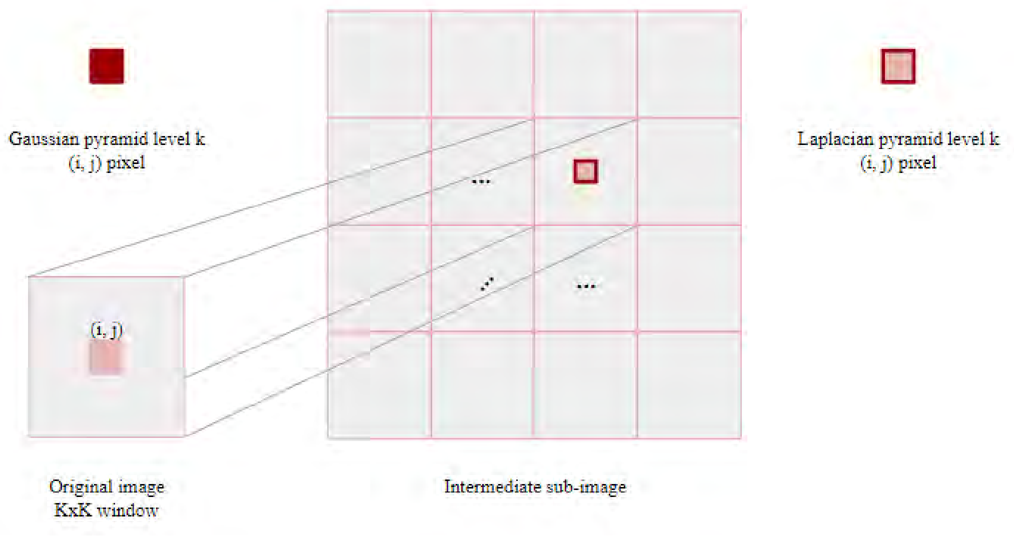

Figure 2.8: Creation of Intermediate sub-image and Laplacian pyramid. Input and output of LLF remapping function. Pixels with the same color has the same value.

2.2.2 Algorithm complexity

We assume that N is the number of pixels in the original image. Then the method of Local Laplacian Filters, yields a complexity in $O(N^2)$, since each coefficient (x_0, y_0, l_0) entails the construction of another pyramid with $O(N)$ pixels. In order to reduce the cost of the implementation, we can decrease the number of the intermediate sub-pyramids which are processed, as it's shown in Fig. 2.6. According to section 2.2.1, in which we examine the size of an intermediate sub-image $(K * K)$, this size is $O(2^{l_0})$. Each level requires the manipulation of $O(N)$ coefficients in total as a results of a level l_0 contains $O(N/2^{l_0})$ coefficients. Since there are $O(logN)$ levels in the pyramid, the overall complexity of our algorithm is $O(NlogN)$ [8].

Chapter 3

Local Laplacian Filters Sequential Implementation

In chapter 2, we introduce the edge-aware detail and tone manipulation Local Laplacian Filters algorithm [8]. In this chapter, we port the initial Matlab version of the algorithm to a single-threaded C implementation and we report performance results [10]. Figure 3.1 shows a call graph of the algorithm.

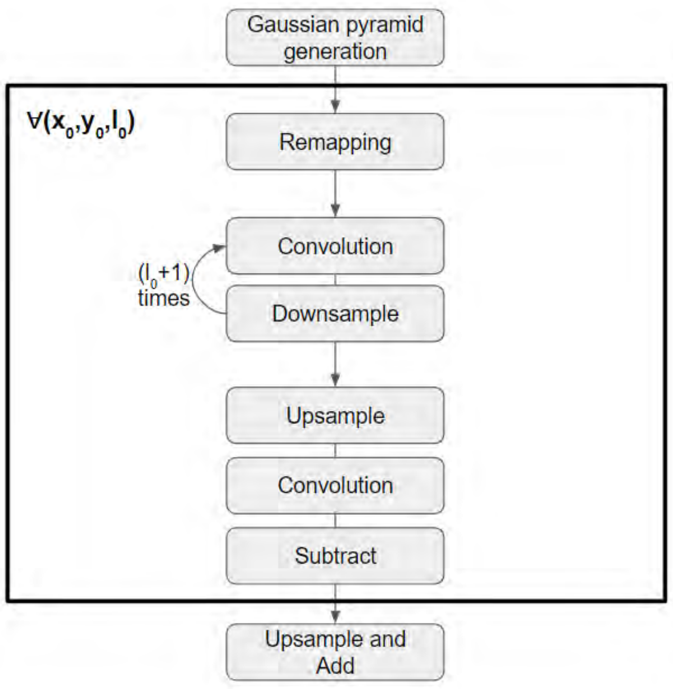

Figure 3.1: LLF call graph

The first step of implementation is to read the input image which saved in Red-Green-Blur (RGB) format. If N is the size of image in pixels, the size of buffer in which the input image copied is $3 * N$. Each image pixel is represented by three float numbers.

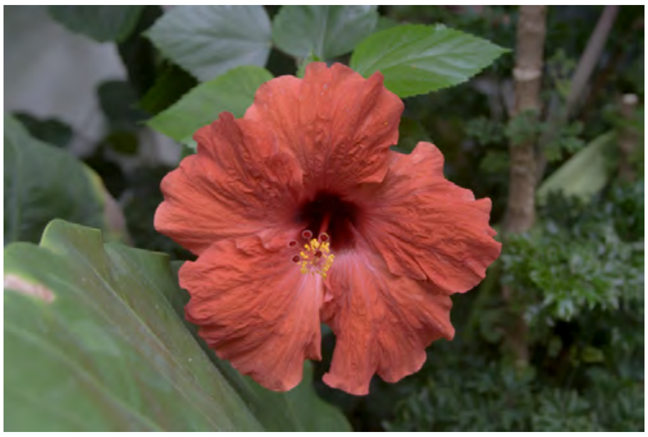

Figure 3.2: Input image

According to [8], using five levels of pyramid typically produces the best results. Figure 3.3 shows the reduce of each level size of Gaussian pyramid in grayscale for the input image in Figure 3.2.

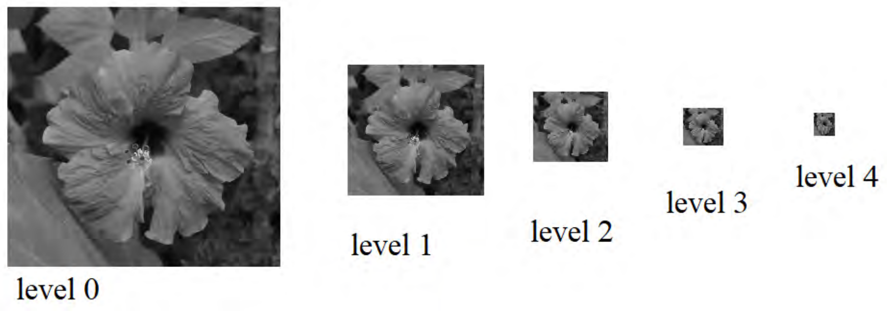

Figure 3.3: Gaussian pyramid with five levels in grayscale for cropped input Fig. 3.2

The convolution blurring function is frequently called in several phases of the algorithm, specifically, in subsampling of the Gaussian pyramid and every sub-image pyramid and upsampling of Laplacian pyramid and sub-image pyramids. The convolution matrix is the result of $v^T * v$, with v a vector of 5 elements.

$$v = \{c \quad b \quad a \quad b \quad c\}, \quad where \quad a + 2c = 2b, \quad a + 2b + 2c = 1 \tag{3.0.1}$$

Therefore, in our implementation the convolution vector is:

$$v = \{0.05 \quad 0.25 \quad 0.4 \quad 0.25 \quad 0.05\} \tag{3.0.2}$$

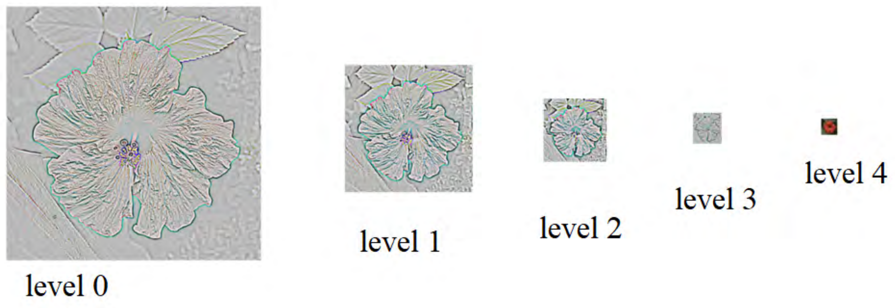

Figure 3.4: Laplacian pyramid with five levels in RGB for cropped input Fig. 3.2

The Laplacian Pyramid consists of error images which shows the difference between the same level of initial image and the new detail (in our implementation) manipulated image. Figure 3.4 shows in RGB format this difference.

According to [10], we apply the LLF to the input image (Fig. 3.2) for twelve combinations of parameters $(\alpha, \beta, \sigma_r)$. Figure 3.5 shows many output images of our implementation. There are not exactly the same with the Matlab manipulated images but the differences are vissually indistinguishable because of the PSNR[1] value is on the order of 30 to 40dB [8]. In our case of color image with three RGB values per pixel, the PSNR defined as the Mean Squared Error (MSE) which is the sum over all squared value differences divided by image size and by three.

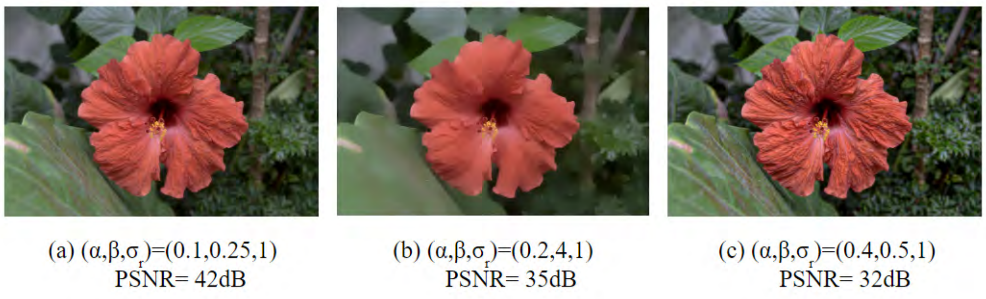
(a) (α,β,σ_r)=(0.1,0.25,1)
PSNR= 42dB
(b) (α,β,σ_r)=(0.2,4,1)
PSNR= 35dB
(c) (α,β,σ_r)=(0.4,0.5,1)
PSNR= 32dB

Figure 3.5: Detail manipulated images

[1]Peak Signal-to-Noise Ration (PSNR) is a metric commonly used to measure the quality of a signal transformation.

Local Laplacian Filters is a complicated algorithm that is shown by the execution time of the sequential C implementation (Fig. 3.6). We run the application at an Intel(R) Core i7-4820K CPU running at 3.70GHz with 16GB DRAM. The code runs using a single thread.

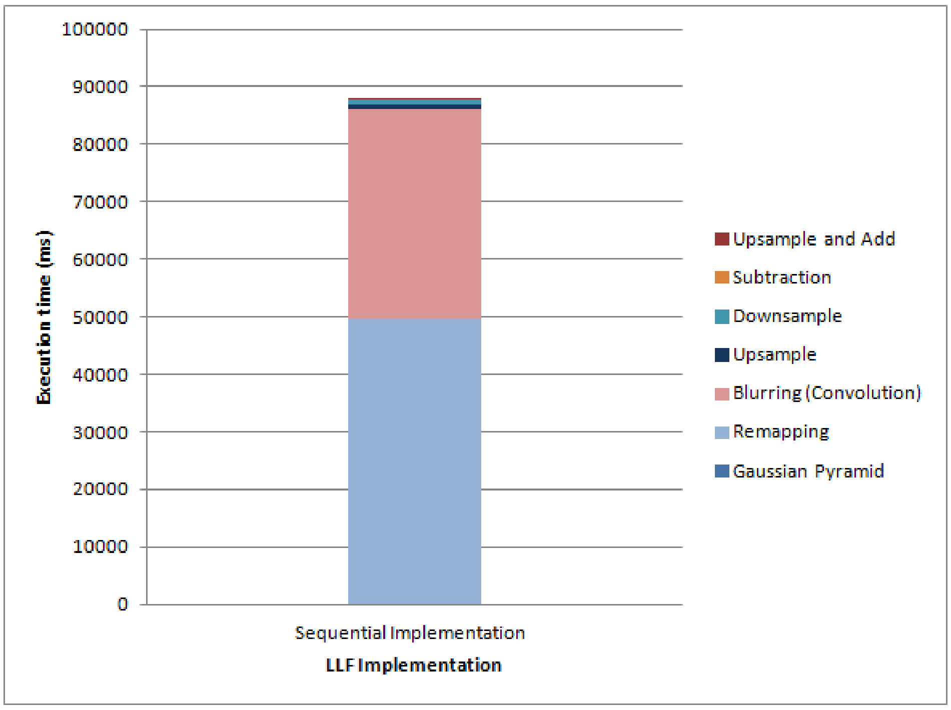

Figure 3.6: Execution time of all phases of sequential LLF

Figure 3.7 shows that the execution time per level increases at the smaller levels (pyramid L0 to L4), although the number of pixels decreases. The difference is that each pixel on a smaller level uses more pixels to calculate it. In fact, each Laplacian pyramid pixels uses $K * K$ pixels, where K depends on the pyramid level ($K = 3(2^{l_0+2} - 1)$).

The values of the user-supplied parameters affect the calling of certain functions and thus the type of commands executed. The difference in execution time for different input parameters relative to the algorithm execution time is negligible. This is because of the functions f_d and f_e have different functionality but similar computational complexity.

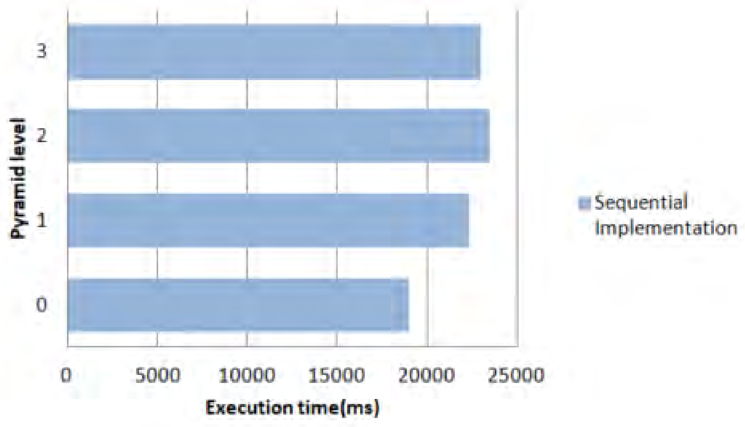

Figure 3.7: Execution time of each level of sequential LLF

The execution time in previous figures measured for the user parameters $(\alpha, \beta, \sigma_r)$ $= (0.25, 1, 0.4)$. Figure 3.8 shows the corresponding output image. These parameter values are assumed to apply the technique of LLF to the input image in the best way.

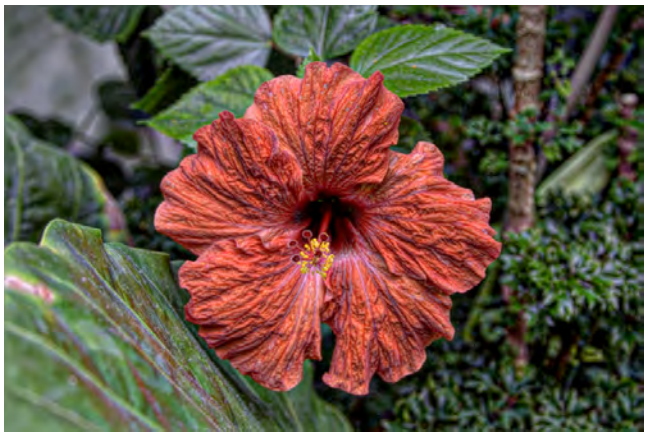

Figure 3.8: Detail manipulation $(\alpha, \beta, \sigma_r)$=(0.4,0.25,1)

Chapter 4

Local Laplacian Filters OpenCL Implementation

4.1 OpenCL Overview

Open Computing Language (OpenCL) provides a standard interface for parallel computing using task- and data-based parallelism. It is a framework that provides many benefits in the field of high-performance, and one of the most important is portability. In order to implement an OpenCL application, we need one host and one or more compute devices. Host is the computational unit on which the host program runs, as a CPU of the computer system and the device (another computational unit) which is accessed via OpenCL library. OpenCL routines, called kernels, can execute on devices across heterogeneous platforms. Heterogenious systems use more than one kind of devices like CPUs, GPUs, field-programmable gate arrays (FPGAs) and other processors or hardware accelerators. These systems gain performance or energy efficiency by adding dissimilar coprocessors, usually incorporating specialized processing capabilities to handle particular tasks. OpenCL host specifies programming languages (based on C99 and C++11) to program these devices and Application Programming Interfaces (APIs) to control the platform and execute programs on the compute devices, such as:

Platform Layer API Query, select and initialize compute devices. Create compute contexts and work-queues.

Runtime API Launch compute kernels. Set kernel execution configuration and manage scheduling, compute and memory resources.

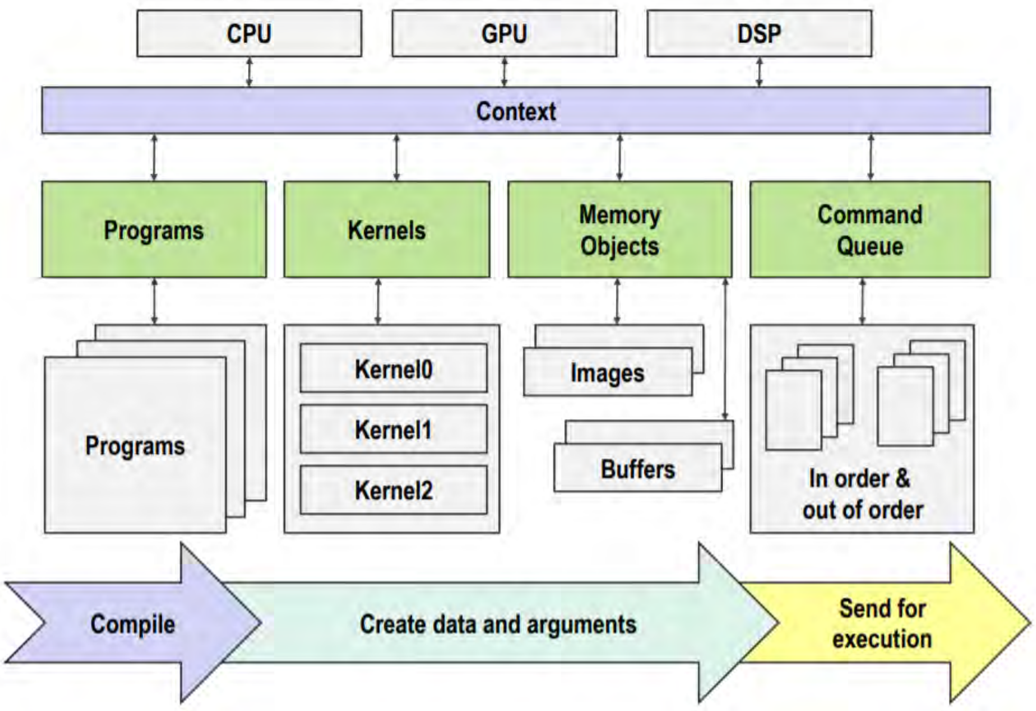

Figure 4.1: OpenCL execution model [11]

An OpenCL kernel is the basic unit of parallel code that can be executed on the target device and each OpenCL application contains one or more kernels. It is executed parallel by an 1D, 2D or 3D array (work group) of work items (all work items run the same code). Work items within a work group cooperate via shared memory, atomic operations and barrier synchronization, in difference with the work items in different work groups which cannot cooperate with each other.

The figure 4.1 shows the OpenCL execution model which contains the context with one or more devices. The context is used by the OpenCL runtime for managing objects such as command-queues, memory, program and kernel objects and for executing kernels on one or more devices specified in the context [12]. It is also used for command queue creation. The command queue is used to control the device, any command from the host to the device is performed through this command queue. In order to execute a kernel, all the data which are being processed must be saved on the device memory. However, the kernel does not have the capability to access memory outside of the device. Therefore, this action must be performed on the host-side. The kernel code must be loaded to another OpenCL object, the program. The program is the final binary which is executed through the platform, after the just-in-time build. In case of the application use multiple devices and consequently multiple command queues, the programmer can use events to synchronize kernel executions between queues.

Multiple types of memories are supported. The global memory saves the data between host and device and it is visible to all the workitems in difference with the shared memory which is accessible from the workitems in the same work group. The global memory is larger and has longer latency than shared memory. Moreover, the private memory is also supported for each thread and contains on-chip device registers. The last type of memory is the constant which saves variables allocated in global memory which are accessed inside a kernel, as read-only variables.

The previously introduced flow is listed below:

- Get a list of available platforms
- Select device
- Create context
- Create command queue
- Create memory objects
- Read kernel file
- Create program object
- Compile kernel
- Create kernel object
- Set kernel arguments
- Execute kernel (Enqueue task): kernel function is called here
- Read memory object
- Free objects

Each OpenCL device has its own information. Some basic capabilities of our device, a NVIDIA GeForce GTX 770 GPGPU, are shown in Table 4.1. All these values restrict our implementation.

DEVICE_NAME	GeForce GTX 770
DEVICE_VENDOR	NVIDIA Corporation
PLATFORM_NAME	NVIDIA CUDA
PLATFORM_VERSION	OpenCL 1.2 CUDA 9.1.83
MAX_WORK_ITEM_DIMS	3
MAX_WORK_ITEM_SIZES	1024 / 1024 / 64
MAX_WORK_GROUP_SIZE	1024
GLOBAL_MEM_SIZE	1994MB
LOCAL_MEM_SIZE	48KB

Table 4.1: Device query

4.2 Parallelization of the algorithm

In order to parallelize the algorithm, we create the call graph (Fig. 3.1) to show the dependencies between the phases of the algorithm. The remapping phase calculates the data which are used in the creation of Laplacian pyramids by downsampling (l_0+1) times and upsampling once every sub-image pyramid. Calculations of each level are independent, but because of the device global memory restrictions (Table 4.1), we can't save all the algorithm intermediate data in global memory. As a result, we use five kernels which are execute sequentially to each other, every one of them for remapping, blurring, downsampling, upsampling and subtraction. Each kernel calculates concurrently the data used for one line of Laplacian pyramid. The call graph in figure 4.2 shows the level of algorithm parallelization.

The sequential implementation has warned us that a large part of the execution time is consumed during the remapping and blurring (Fig. 4.3). The remapping kernel contains expensive calculations and the blurring kernel called two more times than the other kernels during the calculation of one Laplacian pyramid image line.

We apply multiple techniques to the kernels code for better performance, such as different methods of convolution (two kernels, with an one dimension vector for horizontal and vertical convolution), loop unrolling, function inlining, strength reduction and common sub-expression elimination. In the context of these changes, we add the upsampling, downsampling and subtraction kernels despite the small impact of the corresponding phases of the sequential implementation on the total execution time. By creating these kernels we minimize the data which are transferred from GPU global memory back to CPU memory. We will introduce only two implementations (Implementation I and Implementation II).

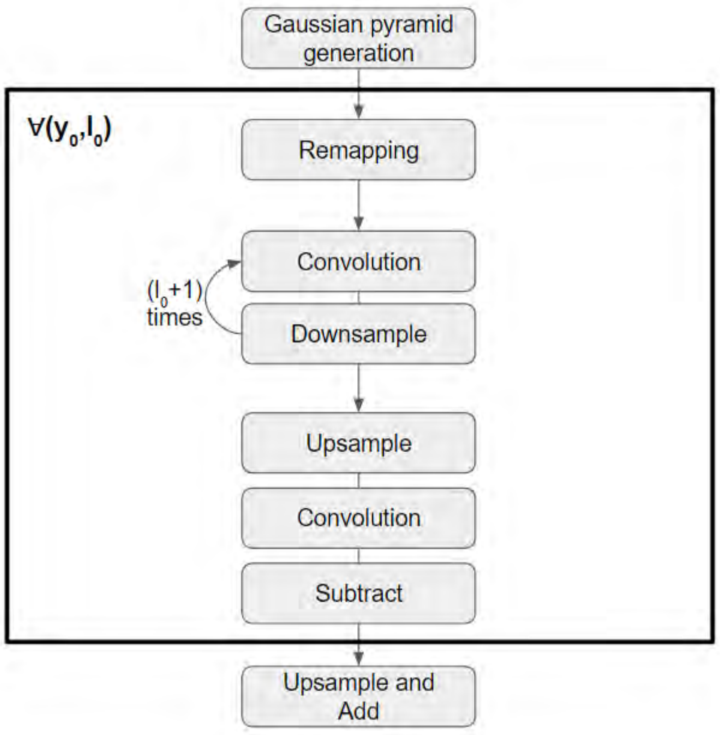

Figure 4.2: Call graph of parallelized LLF implementation

4.2.1 Implementation I

The remapping kernel is a point-wise function and takes as input the original image (use of one line in every kernel call) and the corresponding level of Gaussian pyramid, so for an input image $800 * 533$ we need the local workgroup size has the maximum 800 work items in x dimension and 1 in y. Each remapping kernel thread write a sub-image with $(3 * (2^{l_0+2} - 1)) * (3 * (2^{l_0+2} - 1))$ pixels.

The blurring kernel takes as input #Gaussian_pyramid_width sub-images and calculates an output with the same size. Each blurring kernel thread calculates one pixel of the output but it needs the neighboring 5-by-5 pixels of the input sub-image. It is optimal to have the data of each sub-image in the shared memory of each work-

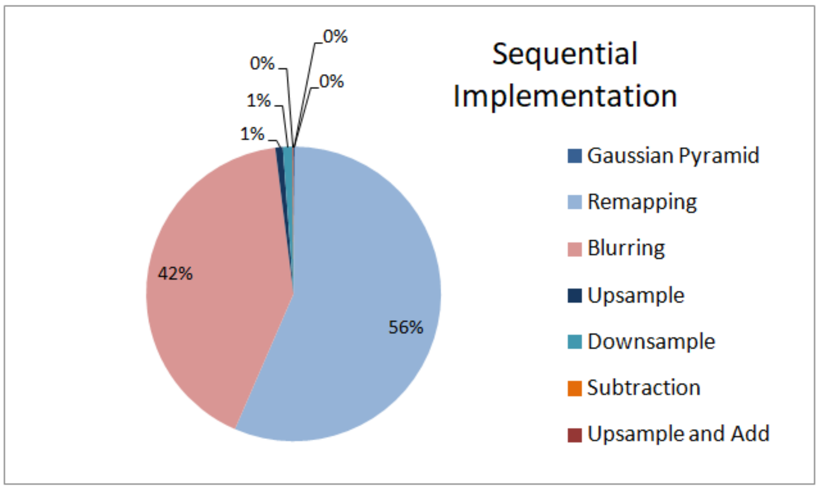

Figure 4.3: Percentage execution time of sequential implementation (Total = 88142.76ms)

group. This means that each workgroup is mapped to one sub-image, the workgroup has $3 * (2^{l_0+2} - 1)$ work items in each of x and y dimensions. For bigger values of level pyramid, such as 2 or 3, the local work group size is calculated bigger than the maximum thread invocations of each workgroup for our device (1024 invocations per workgroup). Therefore, we use a workgroup with $(3 * (2^{l_0+2} - 1)) * 1$ workitems, without using shared memory.

We try to use shared memory and the performance of implementation decreases. Each workgroup calculates a line of the sub-image by using the neighboring 5-by-5 pixels. Every one of the $3 * (2^{l_0+2} - 1)$ workitems in x dimension loads to shared memory the 5 heighboring pixels in y dimention. These colomn-wise memory operations cost to the program execution time more than the advantage of using shared memory faster operations.

The downsample kernel takes as input the output of the blurring kernel and returns an output array with the half size. Each downsample kernel thread (work-item) calculates one pixel of the kernel result, so the number of threads have been invoked is the half of the size of input sub-image. We use a local workgroup size of $((3 * (2^{l_0+2} - 1)/2^r) * 1)$ workitems, with r is the number of reduces which are applied to the initial sub-image. As we already mentioned, the times we apply downsampling in each sub-image is $l_0 + 1$, with l_0 is the level of Gaussian pyramid which is being processed.

In the same way, we approach the upsample kernel local workgroup size. The output sub-image has the double size of the input, then the total number of threads which are invoked in each upsample kernel launching is equal to the double size of the input image. The upsample kernel is called once, after the creation of each sub-image pyramid by downsampling each sub-image. If we assume that in one sub-image is applied r reductions, then the local workgroup contains $((3 * (2^{l_0+2} - 1)/2^{r-1}) * 1)$ workitems.

Concerning the subtract kernel, it calculates the final Laplacian pyramid by subtracting the intermediate sub-image pyramid from Gaussian pyramid. Each call of the kernel calculates one line of the Laplacian pyramid. We need #Gaussian_level_width workitems, each one of them to calculate the index of the sub-image pixels we use. It only depends on the corresponding pixel, so we invoke #Gaussian_level_width$*$1 workitems per local workgroup.

Executing this OpenCL **Implementation I** for an image $800 * 533$ pixels and algorithm parameters $(\alpha, \beta, \sigma_r) = (0.25, 1, 0.4)$ on NVIDIA GeForce GTX 770, we achieve speedup equals to 12 and *Total execution time = 7379.07ms* (Fig. 4.4).

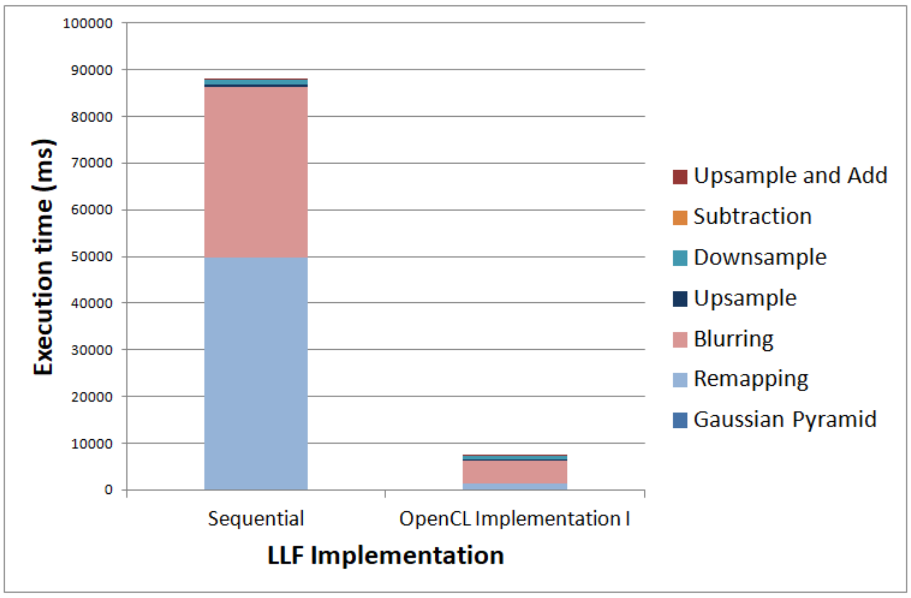

Figure 4.4: Execution time of sequential and OpenCL initial implementations (Speedup = 12)

OpenCL **Implementation I** achieves better execution time for each phase of the algorithm. All the workgroups introduced in the previous analysis of the local work-

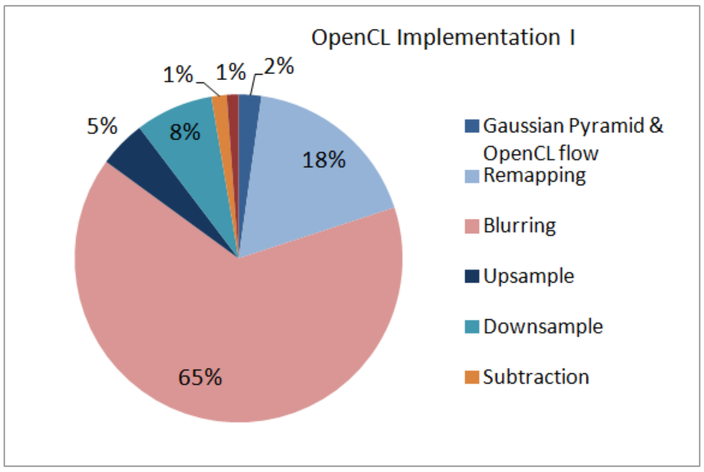

Figure 4.5: Percentage execution time of OpenCL **Implementation I** (Total = 7379.07ms)

group sizes can be improved. The execution time of dowsampling, upsampling and subtraction doesn't affect the total because it is very small in relation to the blurring kernel execution time. The figure 4.5 shows that the execution time of blurring kernel constitutes the 65% of the total execution time of OpenCL Implementation I. Then, we focus to the convolution kernel execution time decrease.

4.2.2 Implementation II

The purpose of this implementation is to minimize the execution time of blurring kernel by using all the available workitems per workgroup. As a result, we create two versions of the blurring kernel. The first version launched in the case of a sub-image is able to calculated by the threads of one workgroup. In different case we call the second version of convolution kernel in which multiple workgroups used for one sub-image production. This implementation has the advantage of the use of shared memory.

The algorithm is using blurring before every time we apply size reduction to an image. The size of sub-images is $(3 * (2^{l_0+2} - 1)) * (3 * (2^{l_0+2} - 1))$. Blurring of sub-images of pyramid levels equals to 0 or 1, need the first version of kernel, such as in the upsampling phase. Sub-images of levels 2 and 3 launch the second kernel in the first levels of sub-image pyramid creation and in the other case call the first version. Figure 4.6 shows the improvement of **Implementation II** which is solely due to the reduction of blurring execution time.

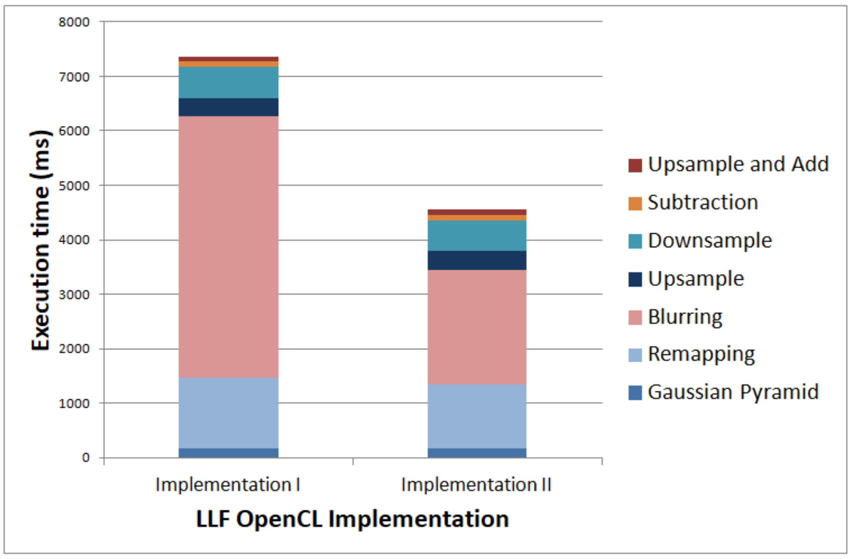

Figure 4.6: Comparison of the two OpenCL implementations

In comparison of the sequential implementation, we achieve Speedup = 19.3 and Total execution time of optimized OpenCL implementation equals to 4567.40ms (Fig. 4.7).

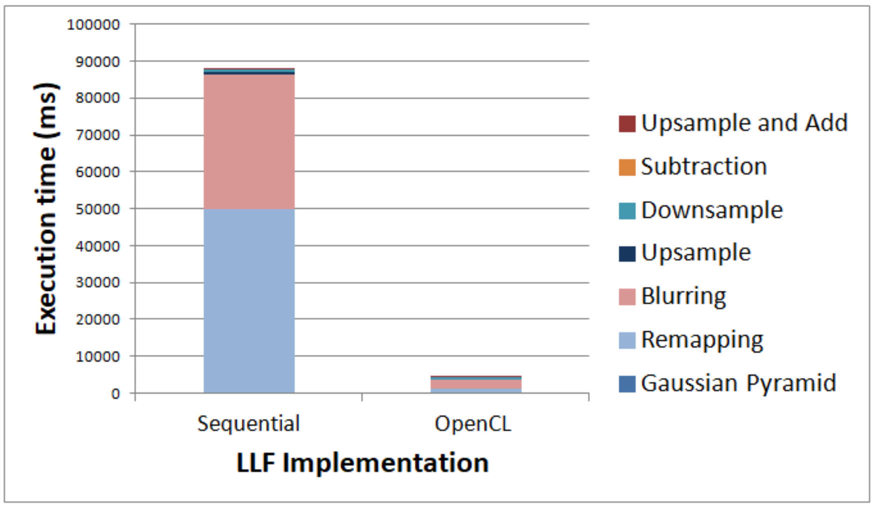

Figure 4.7: Comparison of sequential and optimized OpenCL implementations

Figure 4.8 shows the execution time of the construction of each Laplacian pyramid level image. We observe that the execution time of sequential implementation is increasing as (the index of) the pyramid level increases. Although the size of the pyramid decreases, the size of sub-image $(3 * (2^{l_0+2} - 1)) * (3 * (2^{l_0+2} - 1))$ increases and the calculations in all phases of the algorithms increase too. The sequential

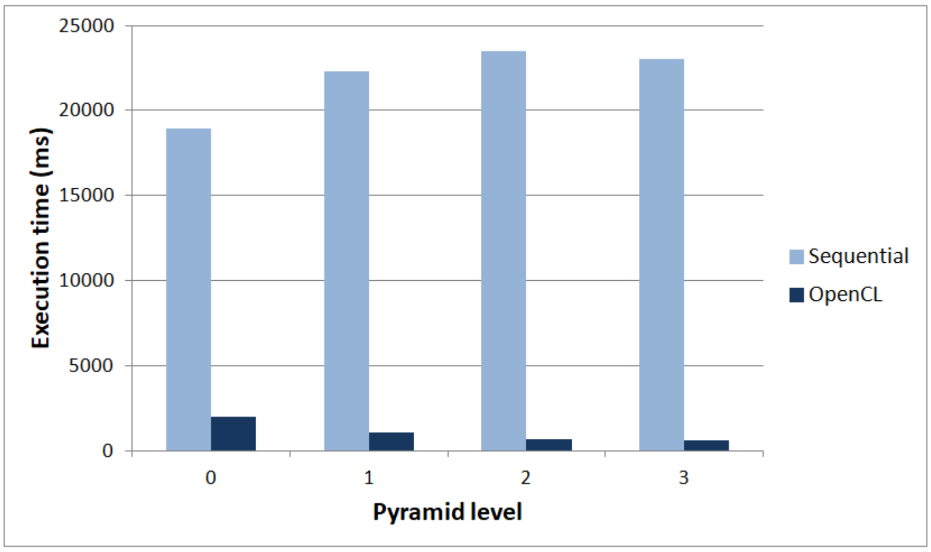

Figure 4.8: Comparison of sequential and optimized OpenCL implementations ET per level

implementation needs more time for all these calculations. In other side, the elements of each Laplacian pyramid line are calculated in parallel. This fact explains the decrease in execution time as the pyramid image size decreases.

Chapter 5

Local Laplacian Filters OpenGL Implementation

5.1 OpenGL Overview

Open Graphics Library (OpenGL) is a cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphics. The API is typically used to interact with a graphics processing unit (GPU), to achieve hardware-accelerated rendering. OpenGL is a large state machine, a collection of variables that define how OpenGL should currently operate. The state of OpenGL is commonly referred to as the OpenGL context. When using OpenGL, we often change its state by setting some options, manipulating some buffers and then render using the current context. An onject in OpenGL is a collection of options that represents a subset of OpenGL's state. For example, we could have an object that represents the settings of the drawing window. One could visualize an object as a C-like struct and an OpenGL's context as a large struct [13].

In OpenGL we use shaders instead of the kernels which are used in OpenCL. A Shader is a user-defined program designed to run on some stage of a graphics processor. Shaders are written in the OpenGL Shading Language (GLSL). The OpenGL rendering pipeline defines the following shader stages:

- Vertex Shaders
- Tessellation Control and Evaluation Shaders

- Geometry Shaders
- Fragment Shaders
- Compute Shaders

A program object can combine multiple shader stages (built from shader objects) into a single one. A program pipeline object can combine programs that contain individual shader stages into a whole pipeline. For the our OpenGL implementation of Local Laplacian Filters we use a compute pipeline in which the compute shaders are loaded.

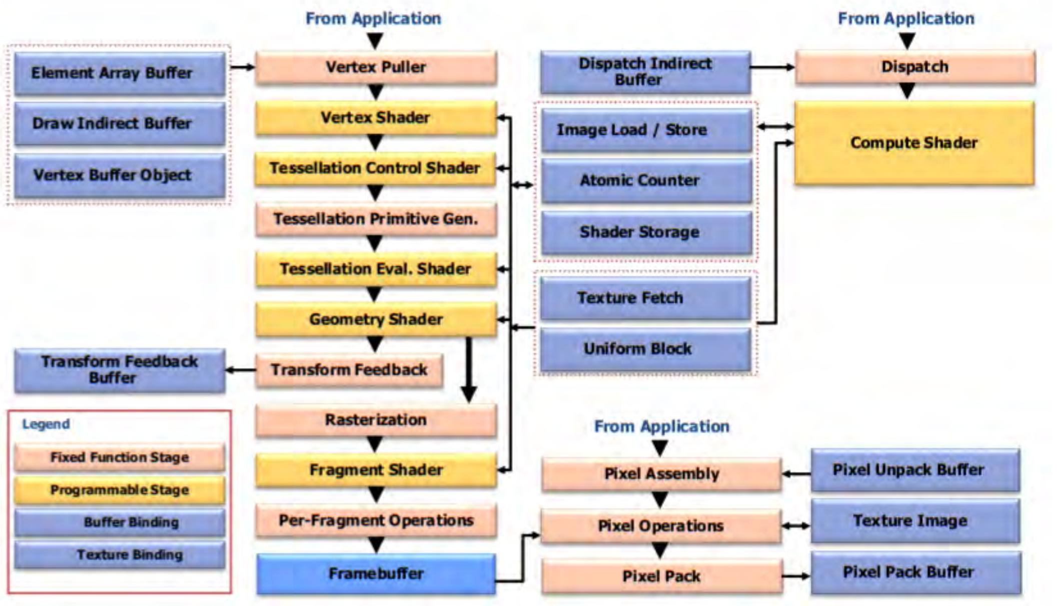

Figure 5.1: Block diagram of the OpenGL pipeline [5]

The following steps used to create an OpenGL application by creating only compute pipeline.

- Create window
- Create context
- Generate memory (buffer) objects
- Read shader file and create shader object
- Compile shader
- Create program object

- Attach shader to program
- Link program
- Create kernel object
- Set kernel arguments
- Execute compute shader
- Read memory object
- Delete buffer objects and programs
- Destroy window

The newest versions of OpenGL support the loading of Spir-V binaries instead of shader files. The binaries are compiled and attached to the program such as compute shaders do.

5.2 LLF Implementations

In this section, we implement the OpenGL implementation of the Local Laplacian Filters which introduced in the previous chapter. The improvements applied to OpenCL code are based on the characteristics of the device. We execute the new code in the same device (NVIDIA GeForce GTX 770), so we use the same functionality as the initial OpenCL implementations. Each one of the two following subsections presents the results of the two best parallel implementations in OpenGL. In section 5.3, we present some OpenGL specific optimizations that were tested.

We create five shaders by using OpenGL Shading Language (GLSL) version 4.5 [14], each one for the phases of the algorithm (remapping, blurring, downsampling, upsampling, subtraction). The two APIs have many common features. The way that the arguments of shaders are passed is different in OpenGL, like how they are defined in the shader code. Uniform variables and buffer objects used for this purpose. We have created a recursive function called from the kernels/shaders when they need to determine the bounds of the output buffer in which every of the shaders can write. OpenGL doesn't support recursion, so we recreate this function by using "for loop".

5.2.1 Implementation I

The number of compute shader executions is defined by the function which is used to execute the compute program, in some way like a kernel is executed in OpenCL. The function that dispatch the shader set the number of workgroups. The size of workgroups is set locally inside the shader code. Newest versions of OpenGL support an extension "ARB_compute_variable_group_size" that allows applications to write generic compute shaders that operate on work groups with arbitrary dimensions.

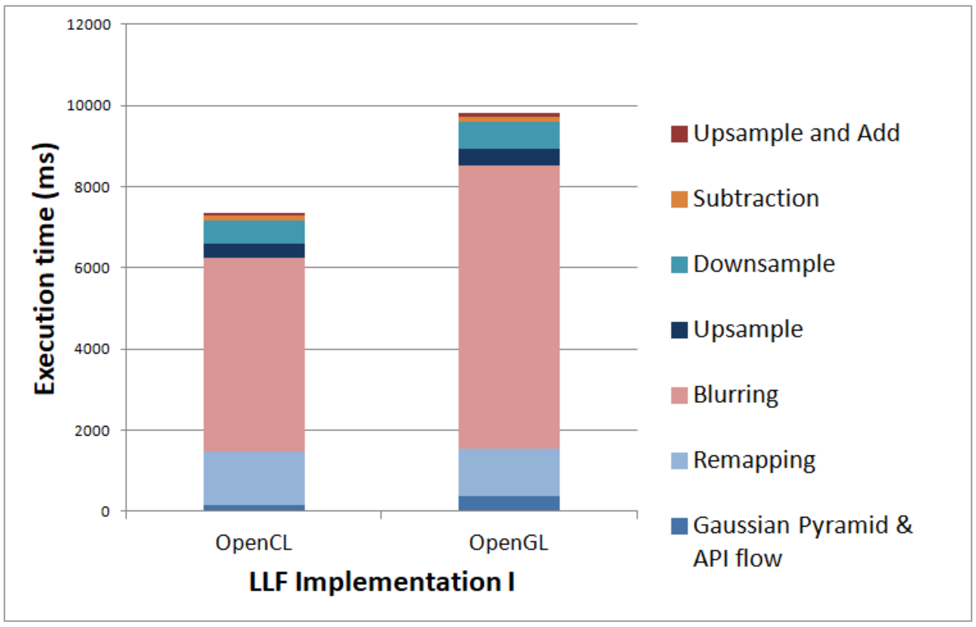

Figure 5.2: Execution time of Implementation I for OpenCL and OpenGL (use of "ARB_compute_variable_group_size" extension)

Implementation I uses the previous extension which define the needed local work-group size in every launch of shader. The execution time of this OpenGL implementation is bigger than the corresponding OpenCL implementation.

Total execution time of OpenCL = 7379.07ms

Total execution time of OpenGL = 9862.06ms

The speed up of the new OpenGL implementation is 9 (Fig. 5.3).

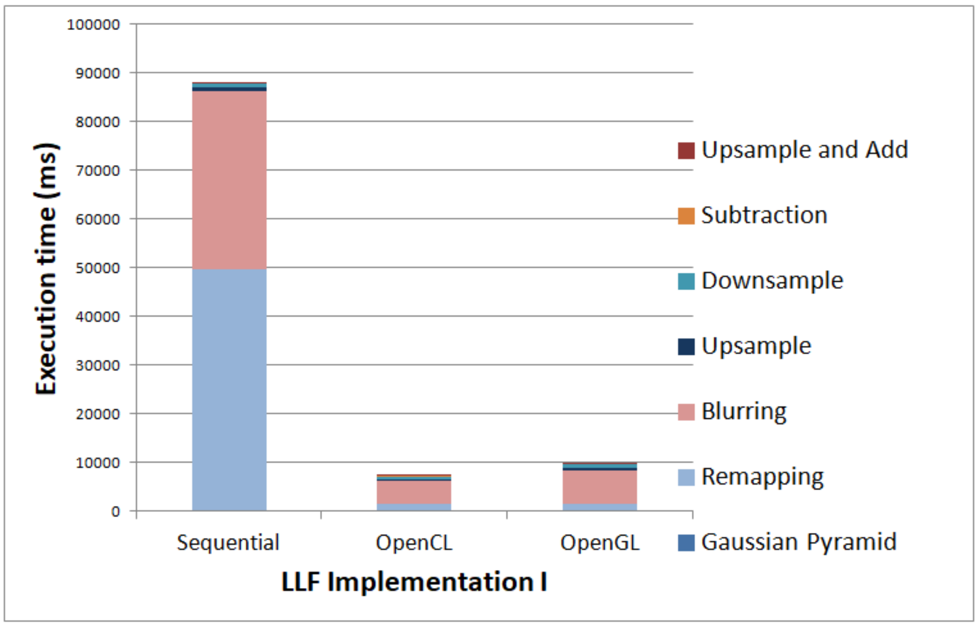

Figure 5.3: Comparison of sequential implementation with the Implementation I in OpenCL and OpenGL

It is known that when the invoked threads don't execute the same code or some threads are useless then the performance of application decreases. In our implementation, every time that one shader called the number of workitems is different. Assuming that the "ARB_compute_variable_group_size" extension isn't supported, then we have to define a constant value for local workgroup size of each shader. We analyze the different local workgroup sizes which are used for each shader. If we find K different values of workgroup sizes then we create K copies of shaders code files. Each one of the new files define a specific value of lcoal workgroup size.

As a result of the previous analysis, we have five shaders (because of the five level Laplacian pyramid) with different local workgroup size for every phase of the algorithm. The workgroup size of remapping kernel equals to (pyramid_width,1,1) and the workgroup size of the remaining shaders (blurring, downsample, upsample) equals to (subimage_width,1,1). Each shader (blurring, downsample, upsample) calculates a subset of the corresponding sub-image. The execution time of this approach is bigger than the previous implementation (use of the "ARB_compute_variable_group_size" extension) (Fig. 5.4) because some of the sub-images have smaller size than the defined size.

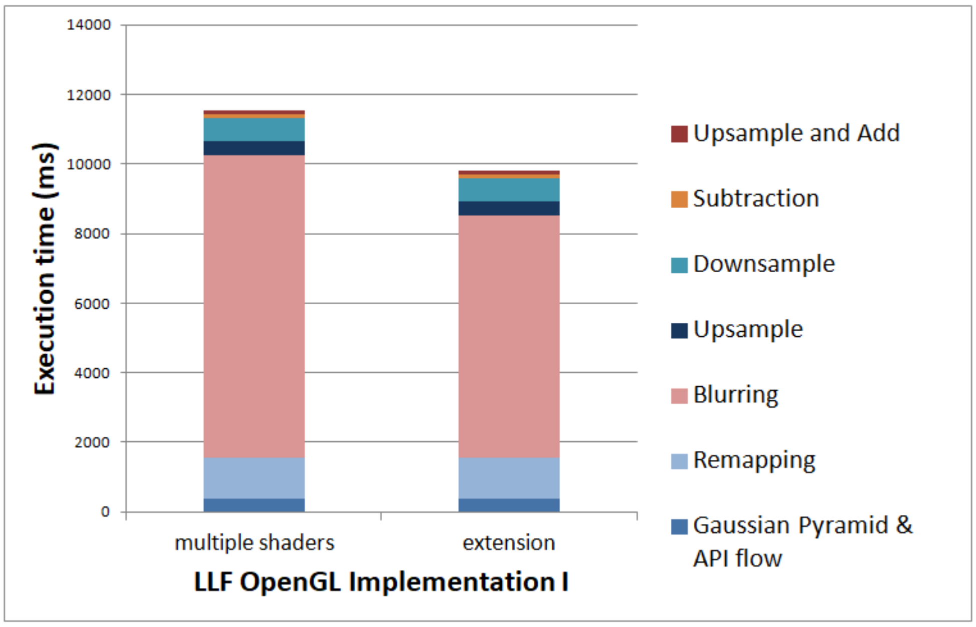

Figure 5.4: The effect in execution time of not using the exact workgroup size

5.2.2 Implementation II

The optimized OpenGL implementation achieves speedup 16.6 and *Total execution time = 5317.45ms* (Fig. 5.5).

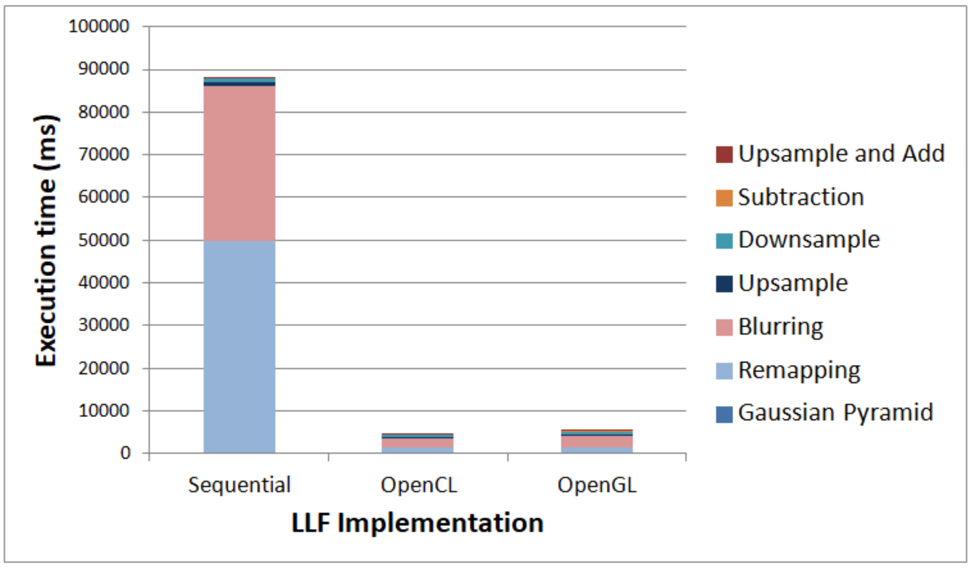

Figure 5.5: Summarize the optimized LLF implementations

5.3 Comparison between OpenGL and OpenCL implementations

We based on the execution time of the optimized OpenCL and OpenGL implementations as they were introduced in sections 4.2.2 and 5.2.2.

Total execution time of OpenCL = 4567.40ms

Total execution time of OpenGL = 5317.45ms

Figure 5.6 shows this difference. The time we need to create the initial flow (read shader files, compile shader objects and attach to the programs, link programs, allocate memory buffer objects) of OpenGL API is bigger than the OpenCL (about 165ms). As mentioned before, OpenGL compiler does not support recursion in functions called from shaders. This costs to the implementation execution time about 225ms.

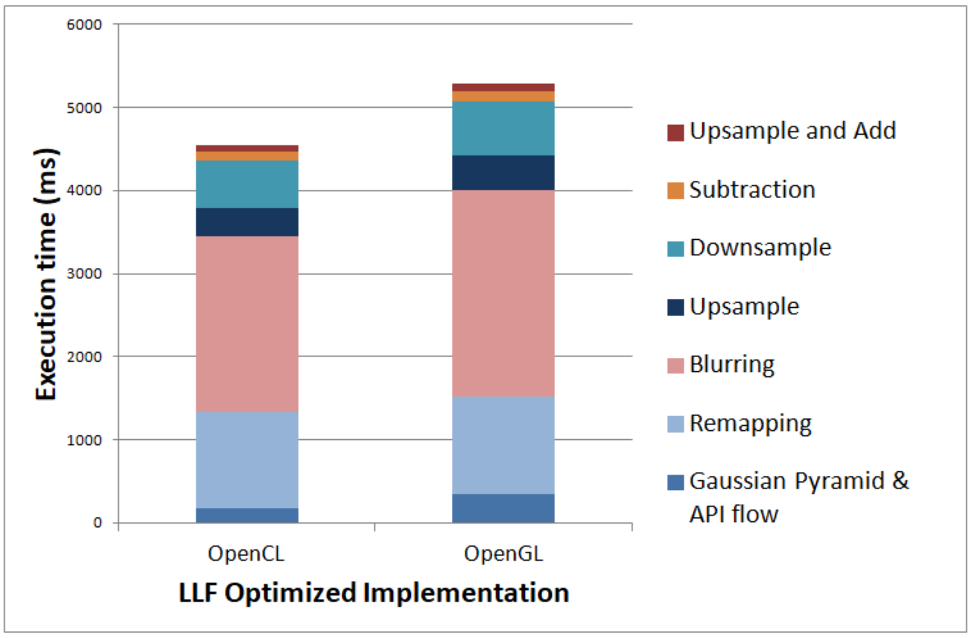

Figure 5.6: Comparison between optimized OpenCL and OpenGL implementations

In order to explain precisely the difference in the results, we try to load the OpenCL kernel and OpenGL shader assembly from the applications. The attempt to load the OpenGL shader assembly was not successful because of the older available context version.

Our OpenGL implementations use Shader Storage Buffer Objects (SSBOs). SS-

BOs are mostly used by compute applications because they can be larger than other buffer objects. Also the reads and writes use incoherent memory accesses, so they need the appropriate synchronization barriers. We also try to apply OpenGL specific optimizations, such as use of other types of buffers like Pixel Buffer Objects and Uniform Buffer Objects. However, the application memory requirements could not be met by the allowed sizes of these types of memory.

OpenCL is created for computing specifically. When we do computing using OpenGL we always have to think about how to map out computing problem to the graphics context (i.e. talk in terms of textures and geometric primitives like triangles etc.). In conclusion, compute shaders are easier to use if we need to add a bit of compute to an OpenGL application, because we don't need to deal with all the complications of sharing devices and resources between OpenGL and OpenCL.

Chapter 6

Local Laplacian Filters Vulkan Implementation

6.1 Vulkan Overview

Vulkan is a new API that provides better abstraction of modern graphics cards. The advantage of this new API is that allows the programmer to better describe what the application intends to do. The possible results are better performance and less surprising driver behavior compared to existing APIs like OpenGL and OpenCL. However, the performance of a compute system is based primarly on the quality of its implementation. OpenGL and OpenCL are expecting to merge into a single API (Vulkan) for compute and graphics. This work is in progress.

As mentioned in previous chapters, the parallel code is represented by the kernels in OpenCL and the shaders in OpenGL. The Vulkan API support a different type to defining the functionality of the shaders and the kernels, an intermediate representation, the SPIR-V. With the use of an external compiler, shaders written in any shading language and kernels can be converted to SPIR-V. The implementations which use SPIR-V, like Vulkan applications, avoid the overhead of code files parsing, compiling and linking the parallel code. Newest versions of OpenCL and OpenGL are able to load SPIR-V binaries instead of read kernel/shader files. The Table 6.1 shows the compatibility of the APIs versions, which of them are able to load specific version of SPIR-V IR.

SPIR-V is the first open intermediate language for parallel compute and graphics. It can take as input all the types of shading languages and specific versions of OpenCL kernels (Table 6.2). The SPIR-V purposes are to: [15]

- Provide a binary intermediate language for kernels/shaders to be the form passed by an API into a driver.
- Map easily to other intermediate languages.
- Allow the first steps of compilation (just-in-time compilation) and reflection to be done offline.
- Be low-level enough to require a reverse-engineering step to reconstruct source code.
- Can be targeted by new front ends for languages can access multiple production quality backends.
- Improve portability by enabling shared tools to generate or operate on it.
- Reduce compile time during application run time.

API version	**IR version**
Vulkan X.XX	SPIR-V 1.X
OpenCL 2.1/2.2	SPIR-V 1.X
OpenCL 2.0 (Extension)	SPIR 2.0
OpenCL 1.2 (Extension)	SPIR 1.2
OpenGL 4.6	SPIR-V 1.X

Table 6.1: Versions compatibility: APIs load SPIR/SPIR-V binary

Tables 6.1 and 6.2 presents another IR (Intermediate Representation), SPIR. SPIR was initially developed for use by OpenCL in order to decrease the time for the online compilation. It is based on the LLVM IR and has now evolved into the cross-API intermediate language SPIR-V. However, the SPIR-V has nothing to do with SPIR but used for the same purpose.

API feature set version	**IR version**
OpenCL 1.2/2.X	SPIR-V 1.X
OpenCL C++ (2.X)	SPIR-V 1.X
OpenCL C 1.2	SPIR 2.0
OpenCL C 2.0	SPIR 2.0
OpenCL C 1.2	SPIR 1.2
GLSL	SPIR-V 1.X

Table 6.2: Versions compatibility: APIs parallel code compiled by SPIR/SPIR-V version

A Vulkan application pass a SPIR-V module containing any of the following operands declared by OpCapability (type defined by Vulkan API):

- Matrix
- Shader
- InputAttachment
- Sampled1D
- Image1D
- SampledBuffer
- ImageBuffer
- ImageQuery
- DerivativeControl

Therefore, Vulkan applications don't support the SPIR-V binaries created by compiling OpenCL kernels. There are many projects in action that work in this path. One of them is the CLSPV compiler [16] which is work in progress and converts the OpenCL kernels to GLSL compute shaders (Fig. 6.1). It consists of a set of LLVM module passes to transform a dialect of LLVM IR into a SPIR-V module containing Vulkan compute shaders.

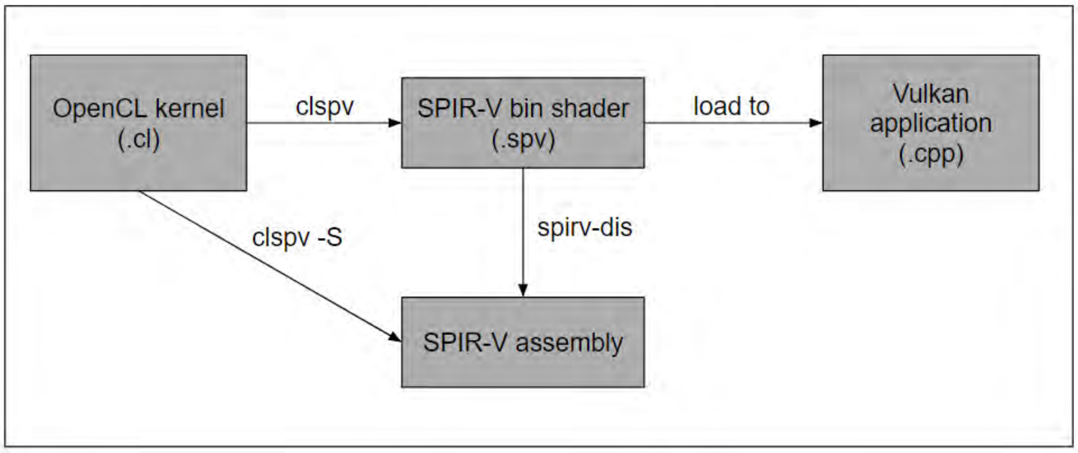

Figure 6.1: Porting OpenCL to Vulkan

We use the SPIR-V in our Local Laplacian Filters Vulkan implementation by compiling the OpenGL shaders. Glslang is the official reference compiler for the OpenGL shading languages [17]. The figure 6.2 shows the way that all types of GLSL shaders compiled. The Khronos Group also provides a set of tools [18] that create or transform SPIR-V modules. Supported features of the tool package that we use during this thesis:

Assembler, Dissambler Supports SPIR-V versions 1.0, 1.1, 1.2 and 1.3 and instraction sets of GLSL std450 version 1.0 Rev3

Optimizer It is under development. Supports many optimizations, some of which we will use in next section

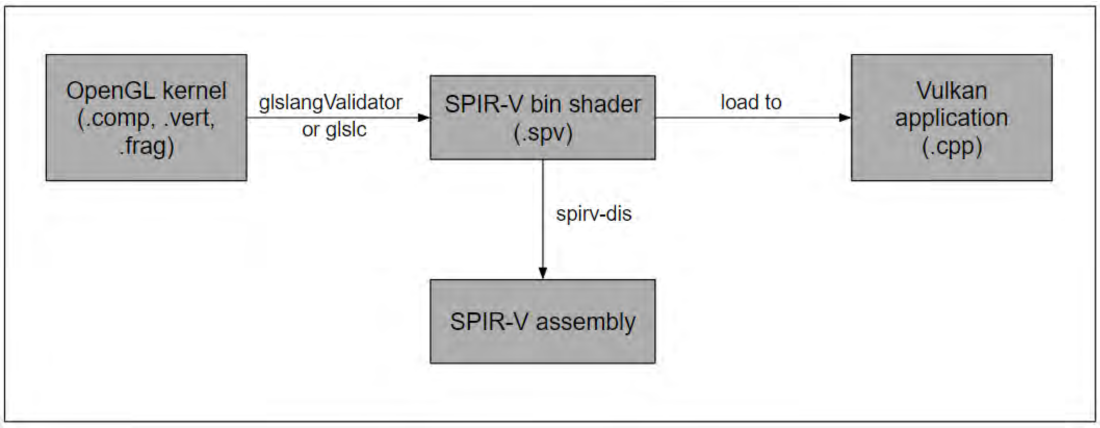
OpenGL kernel (.comp, .vert, .frag)
glslangValidator or glslc
SPIR-V bin shader (.spv)
load to
Vulkan application (.cpp)
spirv-dis
SPIR-V assembly

Figure 6.2: Porting OpenGL to Vulkan

6.1.1 Description of a Vulkan application flow

Vulkan is a layered architecture, consists of the following elements:

The Vulkan Application is the user application (Local Laplacian Filters).

The Vulkan Loader interfaces directly with the application and the ICDs. Between them it can inject a number of optional layers.

Vulkan Layers are optional components used for debugging, validation, and other purposes.

Installable Client Drivers (ICDs) control the Vulkan-capable hardware. Each of them support one or more devices. The loader discovers available physical devices and return this information to the application.

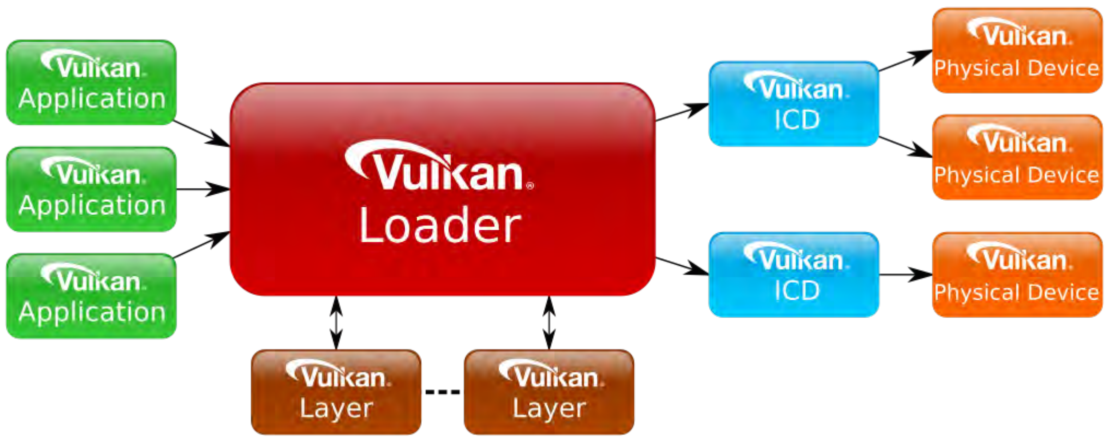

Figure 6.3: Vulkan architecture [19]

The first step is the Vulkan initialization by creating an instance. At this point we specify some simple information including which layers and extensions we want to activate. We also create a handler of available devices, in order to choose one of them. The next step is the communication with the device we create by creating image and/or buffer objects. Compute applications usually require buffers.

The buffers and images can't be used immediately after creation as no memory has been allocated for them. It is the application's responsibility to allocate GPU memory for resources. The memory types have different properties. Some will be

CPU visible or not, coherent between GPU and CPU access, cached or uncached, etc. We can find out all of these properties by querying from the physical device.

When we already allocate the memory we want, we have to assign to the buffers a specific region of the memory. The Vulkan API introduce some new terms, such as command buffers which are allocated from a command pools. We issue all the application GPU commands into command buffers which are sent to a queue for execution. Work is executed on queues belonging to devices. Multiple queues can be synchronized against each other as they can run out of order or in parallel to each other.

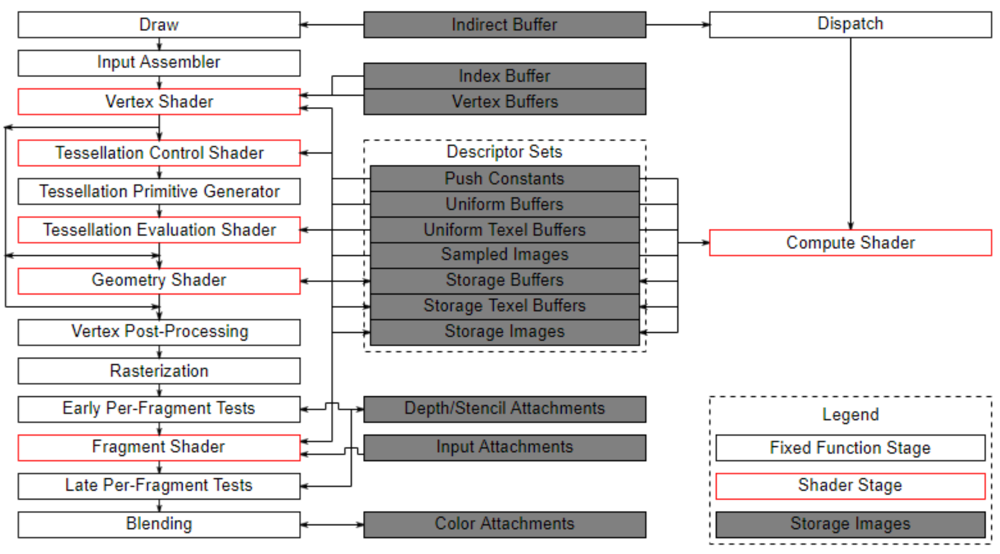

Figure 6.4: Vulkan pipeline [20]

Compute pipeline isn't so complicated as the graphics pipeline is (Fig. 6.4). It consists of a single static compute shader stage and the pipeline layout. As mentioned before, shaders are specified as SPIR-V binary. Once pipeline is created, data has to be ready for execution. Vulkan resourses are presented by descriptors which are arranged in sets. Sets are allocated from pools. Each set has a layout, which is known at pipeline creation time and is shared between sets and pipelines. We can switch pipelines which use sets of the same layout.

6.2 LLF Implementations

The figure 6.2 describes the way that our five shaders compiled, the functionality of which is the same with this of the OpenGL implementation. The SPIR-V doesn't support the "ARB_compute_variable_group_size" extension that we use for parameterizable local work group size, so each shader file must define a constant size of workgroup. Because of this, we categorize the calls of each shader according to the number of workitems which invoked per workgroup, such as the test case in section 5.2.1. It is important to remember that these numbers are limited by the hardware that we use.

6.2.1 Implementation I

The work group size equals to (pyramid_width,1,1) for the remapping shader and (subimage_width,1,1) for the blurring, downsample, upsample shaders. So, we create five copies of each initial shader. Each copy has its own local workgroup size. Because of this, the execution time of the Vulkan implementation I (Fig. 6.5) is expected to be higher than the execution time of the other APIs. However, it achieves speedup equals to 8.

In order to optimize Vulkan implementation, we use the standalone optimizer of the supported SPIR-V Tools. We test all the available compiler flags, but the execution time stays in the same values. We are listing the flags which change the size of the code:

- strip-debug
- eliminate-dead-const
- eliminate-local-single-block
- eliminate-local-single-store
- eliminate-dead-code-aggressive

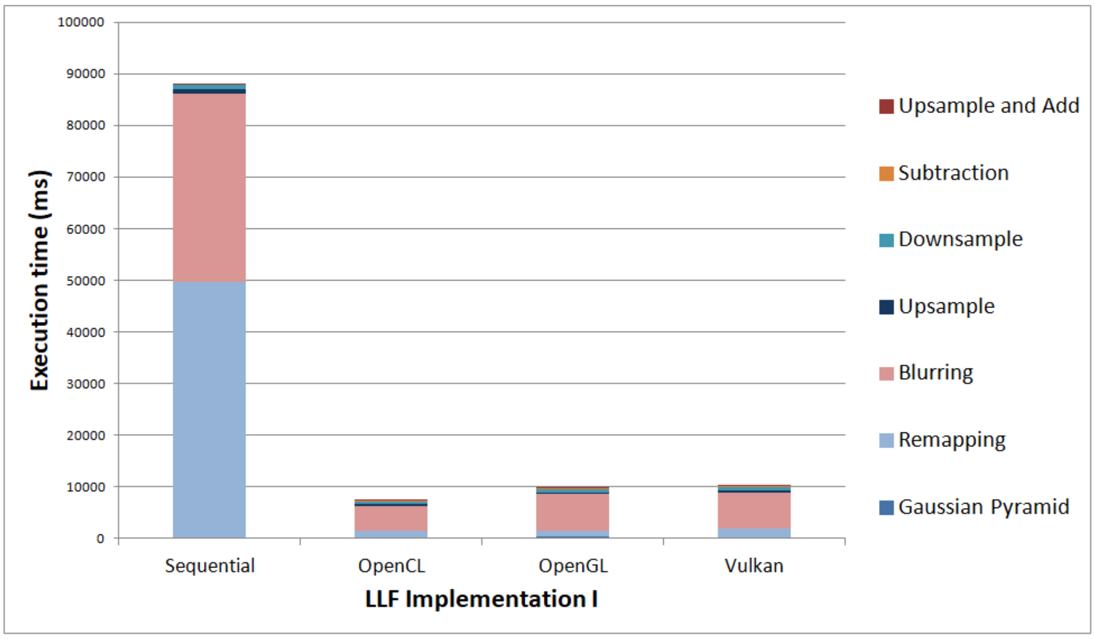

Figure 6.5: Comparison of sequential implementation with the Implementation I in OpenCL, OpenGL and Vulkan

$$Total\ execution\ time\ of\ OpenCL = 7379.07ms$$
$$Total\ execution\ time\ of\ OpenGL = 9862.06ms$$
$$Total\ execution\ time\ of\ Vulkan = 10924.68ms$$

6.2.2 Implementation II

The optimized implementation is more complicated because of the two versions of convolution kernel. The local workgroup size of remapping, downsampling and upsampling kernels are the same with these of Implementation I. Table 6.3 includes all the sizes of convolution local workgroup, in case of each sub-image of an initial image $800 * 533$ is calculated by one workgroup. We choose the workgroups $64 * 16$, $28*28$, $78*13$ and $78*13$, and create four copies of the first version of the convolution shader. In the same way, we choose the workgroups $10 * 97$ and $20 * 51$ of the second version of the convolution kernel. We could try to implement one shader code for every one workgroup size we use, but then the cost of Vulkan flow creation would be increased.

Pyramid level	Downsampling	Upsampling
level 0	$78 * 13(1014)$	$70 * 14(980)$
level 1	$25 * 25(625)$	$64 * 16(1024)$
	$60 * 15(900)$	
level 2	$27 * 27(724)$	$64 * 16(1024)$
	$64 * 16(1024)$	
level 3	$28 * 28(784)$	$64 * 16(1024)$
	$64 * 16(1024)$	

Table 6.3: Convolution kernel local workgroup size calculates one or more sub-images

Pyramid level	Downsampling	Upsampling
level 2	$20 * 49(980)$	
level 3	$10 * 97(970)$	
	$20 * 51(1020)$	

Table 6.4: Convolution kernel local workgroup size calculates a part of a sub-image

Figure 6.6 shows the execution time of the optimized implementation (**Implementation II**) for the three APIs. The execution time of the Vulkan implementation is about 600ms higher than the remaining implementations. This is explained by the fact that the SPIR-V IR requires the definition of the local workgroup size in time of shader compilation.

Total execution time of OpenCL = 4567.40ms

Total execution time of OpenGL = 5317.45ms

Total execution time of Vulkan = 6561.15ms

Figure 6.7 shows the total change in execution time, where the Vulkan implementation achieves a speedup of 13.4.

Total execution time of Sequential implementation = 88142.76ms

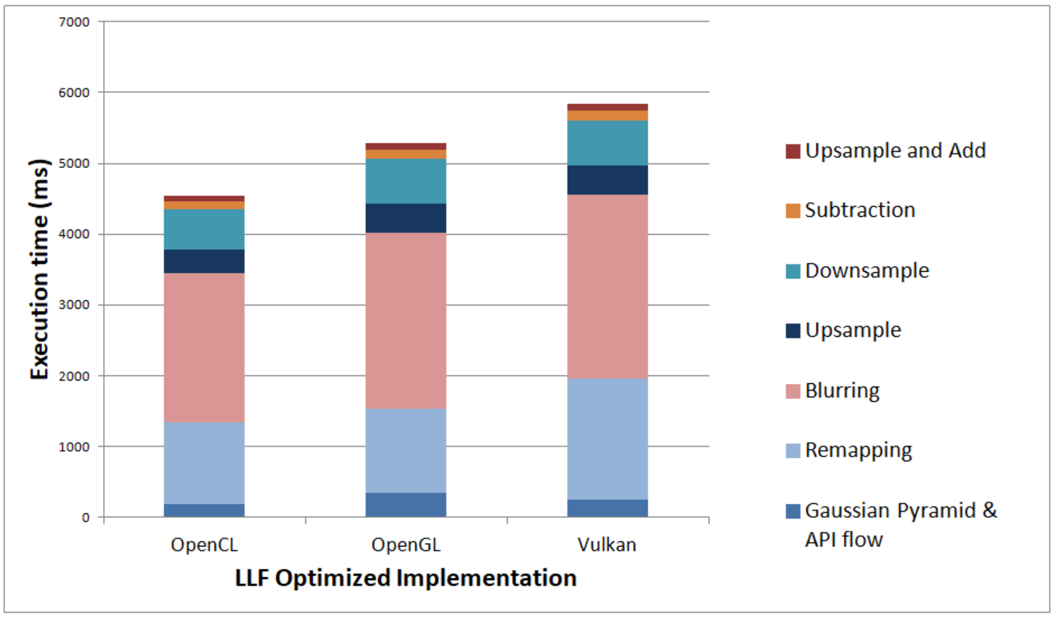

Figure 6.6: Comparison between OpenCL, OpenGL and Vulkan Implementation II

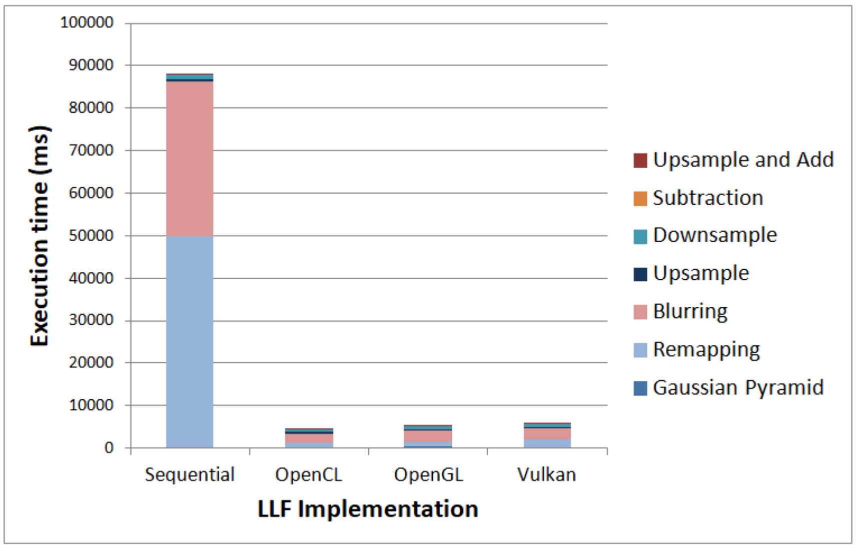

Figure 6.7: Execution time of Implementation II (Vulkan speedup = 13.4)

Chapter 7

Conclusions

In conclusion, the execution time of our implementations in the three APIs (OpenCL, OpenGL, Vulkan) shows that the OpenCL implementation achieves better performance. This is partially expected because the OpenCL API is created specifically for computing. However, a key advantage of Vulkan over OpenGL is the ability to generate GPU work in parallel using many CPU cores, making Vulkan particularly useful for CPU-bound developers. It is import to refer that the attempt of OpenCL and OpenGL to merge into the Vulkan API is work in progress. This means that any new tool is able to redefine the situation.

7.1 Future work

In this thesis a sequential and an OpenCL, an OpenGL and a Vulkan implementation of Local Laplacian Filters are presented. The execution time of parallel implementations is 19x better than the sequential implementation. There are some ideas that we could try to apply to the LLF implementation.

Each pixel of the input RGB image is represented by three float values. In order to minimize the size of memory we use and the number of floating point operations, we can use single point variables. Also we could create an intensity image, one pixel of which is computed by the function I_i below [8]. After the calculation of the output image, the pixels presents the I_r information. We need to translate this information to RGB. Because of this we save the color ratios (ρ_r, ρ_g, ρ_b) of every initial pixel.

$$I_i = (20I_r + 40I_g + I_b)/61 \tag{7.1.1}$$

$$(\rho_r, \rho_g, \rho_b) = (I_r, I_g, I_b)/I_i \tag{7.1.2}$$

Therefore, we could use approximation techniques that inserts error to the output image which is not viewable in order to gain in performance.

Bibliography

[1] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990. ISBN: 1-55880-069-8.

[2] Victor Pankratius, Ali-Reza Adl-Tabatabai, and Walter Tichy. *Fundamentals of Multicore Software Development*. 1st. Boca Raton, FL, USA: CRC Press, Inc., 2017. ISBN: 1138114375, 9781138114371.

[3] Krste Asanovic et al. *The landscape of parallel computing research: A view from berkeley*. Tech. rep. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[4] Khronos Group. *OpenCL Overview:The open standard for parallel programming of heterogeneous systems*. URL: https://www.khronos.org/opencl/.

[5] Khronos Group. *The OpenGL Graphics System: A Specification Version 4.5 (Core Profile)*. June 29, 2017. URL: https://www.khronos.org/registry/OpenGL/specs/gl/glspec45.core.pdf.

[6] The Khronos Group. *SPIR-V Specification version 1.00*. URL: https://www.khronos.org/registry/spir-v/specs/1.0/SPIRV.html.

[7] Khronos Group. *Khronos Releases Vulkan 1.0 Specification*. 2016. URL: https://www.khronos.org/news/press/khronos-releases-vulkan-1-0-specification.

[8] Sylvain Paris, Samuel W Hasinoff, and Jan Kautz. "Local Laplacian filters: Edge-aware image processing with a Laplacian pyramid." In: *ACM Trans. Graph.* 30.4 (2011), pp. 68–1.

[9] Peter J Burt and Edward H Adelson. “The Laplacian pyramid as a compact image code”. In: *Readings in Computer Vision*. Elsevier, 1987, pp. 671–679.

[10] Sylvain Paris, Samuel W Hasinoff, and Jan Kautz. *Local Laplacian Filters: Edge-aware Image Processing with a Laplacian Pyramid Matlab source code and samples*. 2011. URL: https://people.csail.mit.edu/sparis/publi/2011/siggraph/.

[11] Khronos Group. *OpenCL Details,Khronos Group Presentations*. 2012. URL: https://www.khronos.org/assets/uploads/developers/library/2012-pan-pacific-road-show-June/OpenCL-Details-Taiwan_June-2012.pdf.

[12] Khronos Group. *OpenCL 1.2 Reference Pages*. URL: https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/.

[13] *Learn OpenGL*. URL: https://learnopengl.com/Getting-started/OpenGL.

[14] Khronos Group. *The OpenGL Shading Language, Language Version: 4.50*. May 09, 2017. URL: https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.50.pdf.

[15] Google John Kessenich and Intel Boaz Ouriel. *SPIR-V Specification, Version 1.00*. January 16, 2018. URL: https://www.khronos.org/registry/spir-v/specs/1.0/SPIRV.pdf.

[16] Google and Codeplay. *CLSPV prototype compiler, open source github project*. URL: https://github.com/google/clspv.

[17] Khronos Group. *OpenGL / OpenGL ES Reference Compiler*. URL: https://www.khronos.org/opengles/sdk/tools/Reference-Compiler/.

[18] Khronos Group. *SPIRV-Tools, open source github project*. URL: https://github.com/KhronosGroup/SPIRV-Tools.

[19] Khronos Group. *Architecture of the Vulkan Loader Interfaces*. URL: https://github.com/KhronosGroup/Vulkan-Loader/blob/master/loader/LoaderAndLayerInterface.md.

[20] The Khronos Group. *Vulkan 1.0.87 - A Specification*. URL: https://www.khronos.org/registry/vulkan/specs/1.0/html/vkspec.html.

[21] Jason Ekstrand. *Vulkan in Open-Source:A discussion of the new Vulkan graphics API and its impact on Open-source software*. 2016. URL: https://archive.fosdem.org/2016/schedule/event/vulkan_graphics/.

[22] NVIDIA Corporation. *Introduction to GPU Computing with OpenCL*. 2009. URL: http://developer.download.nvidia.com/CUDA/training/NVIDIA_GPU_Computing_Webinars_Introduction_To_OpenCL.pdf.

[23] Ryoji Tsuchiyama et al. *The OpenCL Programming Book*. January 2012. URL: https://us.fixstars.com/opencl/book/.

[24] Benedict Gaster et al. *Heterogeneous computing with openCL: revised openCL 1*. Newnes, 2012.

[25] Aaftab Munshi et al. *OpenCL Programming Guide*. 1st. Addison-Wesley Professional, 2011. ISBN: 0321749642, 9780321749642.

[26] Khronos Group. *SPIR Overview:The Industry Open Standard Intermediate Language for Parallel Compute and Graphics*. URL: https://www.khronos.org/spir/.

[27] The Khronos Group. *Khronos Releases OpenCL 2.2 With SPIR-V 1.2*. May 16, 2017 - IWOCL 2017, Toronto. URL: https://www.khronos.org/news/press/khronos-releases-opencl-2.2-with-spir-v-1.2.

[28] Khronos Group. *glslang:Khronos reference front-end for GLSL and ESSL, and sample SPIR-V generator, open source github project*. URL: https://github.com/KhronosGroup/glslang.

[29] Chris Hebert and Christoph Kubisch. *Vulkan Memory Management*. URL: https://developer.nvidia.com/vulkan-memory-management.

[30] Baldur Karlsson. *Vulkan in 30 minutes*. URL: https://renderdoc.org/vulkan-in-30-minutes.html.

[31] Timothy Lottes. *Vulkan Device Memory*. URL: https://gpuopen.com/vulkan-device-memory/.

[32] Neil Henning Codeplay. *An Introduction to SPIR-V, Game Developers Conference*. March 2016. URL: http://www.cogsci.rpi.edu/~destem/gamearch/gdc16/AnIntroductionToSPIR-V.pdf.

[33] Harold Serrano. *Understanding OpenGL Objects*. URL: https://www.haroldserrano.com/blog/understanding-opengl-objects.

[34] Neil Henning. *A simple Vulkan Compute example*. URL: http://www.duskborn.com/a-simple-vulkan-compute-example/.

[35] Khronos Group. *Memory Model*. URL: https://www.khronos.org/opengl/wiki/Memory_Model.