Πανεπιστήμιο Θεσσαλίας
Πολυτεχνική Σχολή
Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Υλικό και λογισμικό framework για επεξεργασία εικόνας

Hardware and Software framework for image processing

Διπλωματική Εργασία

Γκάρας Αθανάσιος

Επιβλέποντες καθηγητές : Νικόλαος Μπέλλας
Αναπληρωτής Καθηγητής

Χρήστος Αντωνόπουλος
Επίκουρος Καθηγητής

Βόλος, Ιανουάριος 2018

Πανεπιστήμιο Θεσσαλίας
Πολυτεχνική Σχολή
Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Υλικό και λογισμικό framework για επεξεργασία εικόνας

Hardware and Software framework for image processing

Διπλωματική Εργασία
Γκάρας Αθανάσιος

Επιβλέποντες καθηγητές : Νικόλαος Μπέλλας
Αναπληρωτής Καθηγητής

Χρήστος Αντωνόπουλος
Επίκουρος Καθηγητής

Εγκρίθηκε από την διμελή εξεταστική επιτροπή την

…………………..	…………………..
Νικόλαος Μπέλλας	Χρήστος Αντωνόπουλος
Αναπληρωτής Καθηγητής	Επίκουρος Καθηγητής

Βόλος, Ιανουάριος 2018

……………………
Γκάρας Αθανάσιος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Πανεπιστημίου Θεσσαλίας

Στην οικογένεια μου και τους φίλους μου

Ευχαριστίες

Με την περάτωση της συγκεκριμένης εργασίας, θα ήθελα να ευχαριστήσω τους επιβλέποντες της διπλωματικής εργασίας , τον κύριο Μπέλλα Νικόλαο και τον κύριο Χρήστο Αντωνόπουλο για την εμπιστοσύνη που έδειξαν απέναντι μου με την ανάθεση του Θέματος της εργασίας , την συνεργασία και την συνεχή καθοδήγηση τους , που χωρίς αυτή η εκπόνηση της διπλωματικής θα ήταν πολύ πιο δύσκολη , αν όχι αδύνατη.
Επίσης θα ήθελα να ευχαριστήσω την οικογένεια μου και τους φίλους μου που με βοήθησαν , ο καθένας με τον δικό του τρόπο σε όλα τα χρόνια των σπουδών μου, αλλά και κατά την εκπόνηση της διπλωματικής εργασίας .

Contents

List of figures

Περίληψη

Το φίλτρο Local Laplacian , είναι μια εφαρμογή επεξεργασίας εικόνας , που επεξεργάζεται τις λεπτομέρειες και τις ακμές μίας εικόνας και παράγει ένα μεγάλο εύρος από έντονα εφέ , χωρίς όμως να αλλοιώνει την εικόνα. Το φίλτρο παράγει πολλές , μικρές Laplacian πυραμίδες από επεξεργασμένες εκδοχές της εικόνας εισόδου. Δυστυχώς αυτό έχει ως αποτέλεσμα υψηλούς χρόνους εκτέλεσης. Στην αρχική C υλοποίηση η επεξεργασία μιας εικόνας 1024x768 χρειάστηκε 70 δευτερόλεπτα .Σε αυτή τη διπλωματική έγινε μια προσπάθεια να μειωθεί ο χρόνος εκτέλεσης , αξιοποιώντας τον παραλληλισμό που υπάρχει στον αλγόριθμο , με την χρήση μιας multi thread CPU ,μιας GPU και μιας FPGA οι οποίες ελέγχονται από ένα Heterogeneous-aware Runtime System, το Centaurus runtime. Με την χρήση της γλώσσας OpenCl έγιναν οι εκδόσεις για την multithreaded CPU και την GPU , ενώ για την υλοποίηση σε FPGA χρησιμοποιήθηκε το Vivado HLS , μέσω του οποίου παράξαμε έναν accelerator σε Verilog. Γενικά επικεντρωθήκαμε περισσότερο στην FPGA υλοποίηση Τελικά καταφέραμε να πάρουμε πολύ καλά αποτελέσματα για την GPU και την FPGA μειώνοντας τον χρόνο εκτέλεσης σε 2.9 δευτερόλεπτα για την FPGA και 3.6 δευτερόλεπτα για την GPU.

Abstract

The Local 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. The filter constructs many small Laplacian pyramids of processed versions of the input image. Unfortunately this results in high execution times. In our first C implementation processing one 1024x768 image was done in about 70 seconds .In this thesis we tried to reduce the execution time of the application by exploiting the parallelism of the algorithm , with the use of a multi thread CPU a GPU and an FPGA device controlled by a Heterogeneous-aware Runtime System, the Centaurus runtime. For the multi thread CPU and the GPU we used OpenCL and for the FPGA the Vivado HLS tool was used to produce an accelerator in Verilog. We focused on the FPGA implementation and both the FPGA and the GPU implementations gave a very good speedup reducing the execution time to 2.9 seconds for the FPGA and 3.6 seconds for the GPU.

1 Introduction

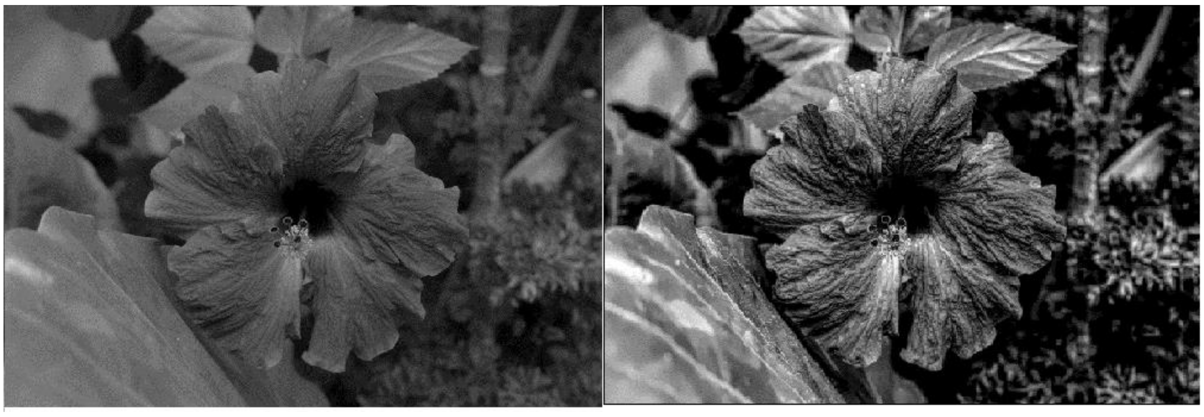


Figure 1 Example of the Local Laplacian filtering for a gray scale image input (left) and the processed output (right)

The local Laplacian filters, introduce a method for edge aware image processing, using Laplacian pyramids. The algorithm can produce results for detail enhancement or detail smoothing (edge manipulation), and for tone mapping or inverse tone mapping (tone manipulation) of an image. «The output is calculated as the construction of the Laplacian pyramid of the filtered output. For each output pyramid coefficient, we render a filtered version of the full-resolution image, processed to have the desired properties according to the corresponding local image value at the same scale, build a new Laplacian pyramid from the filtered image, and then copy the corresponding coefficient to the output pyramid. The advantage of this approach is that while it may be non-trivial to produce an image with the desired property everywhere, it is often easier to obtain the property locally" [(PARIS, HASINOFF, & KAUTZ, 2011)]

Methods like anisotropic diffusion [(Perona & Malik, 1990); (Aubert & Kornprobst, 2002)], neighborhood filtering [(Tomasi & Maduchi, 1998); (Kass & Solomon, 2010)], edge-preserving optimization [(Bhat, Zitnick, Cohen, & Curless, 2010); (Farbman, Fattal, Lischinski, & Szeliski, 2008)] and edge-aware wavelets [(Fattal, 2009)], are considered to be more suitable than Laplacian pyramids for detail manipulation. Unfortunately, some of them have difficult parameters to set or they suffer from edge artifacts like haloes in large decreases on the details.

1.1 Problem description

The local Laplacian filters method does not generate edge artifacts even for large increases in details, performs well on all cases and produce a wide range of effects. The basic shortcoming of this approach is the high running time (1 min in a 1MP image) of the algorithm. In this thesis we tried to minimize the running times of the local Laplacian filter, using the Centaurus runtime (Vassiliadis, et al., 2015), a Heterogeneous-aware Runtime System that runs on top of OpenCL and supports kernel execution on multicore CPU, GPU and FPGA. In this approach C and OpenCL code was produced, for the CPU and the GPU and C++ code as input for the Vivado HLS in order to generate Verilog and finally a bitstream for the FPGA thought Vivado. We tried different implementations in different devices in order to decide which one was the best in terms of accuracy and performance. The implementations will be presented and explained later along with the profiling numbers.

1.2 Background on Gaussian and Laplacian Pyramids.

In order to explain the algorithm an introduction to the Gaussian and Laplacian pyramids [(Burt & Adelson)] is necessary. The Gaussian pyramid is a set of images that derive from one input image. Each of these images (called levels) represent a version of the input image (which is also the level 0 image) at a lower resolution. In this approach each level has half the width and height of its parent level. Every level, is produced by its previous level by applying a Gaussian blur, and a downsample kernel. In this implementation the Gaussian blur is a 2D convolution filter (with zeroes for padding) with the 5 length kernel w= {.05, .25, .4, .25, .05}.Then downsample is applied by copying half the pixels(in width and height) from the output (t0) of the Gaussian filter (g1(i,j)=t0(2*i,2*j)). As a result of these operations ($g_l(i,j) = \sum_{m=-2}^{2}\sum_{n=-2}^{2} w(m,n) g_{l-1}(2i+m, 2j+n)$) the higher frequency details of the image progressively disappear from one level to another.
The Laplacian pyramid on the other hand uses the same concept of pyramids but in each level, only the higher frequency details are preserved. Every level of the Laplacian pyramid derives from the Gaussian pyramid by expanding the next level (smaller version) and subtracting it from the current level L`= G`- expand (G`+1).

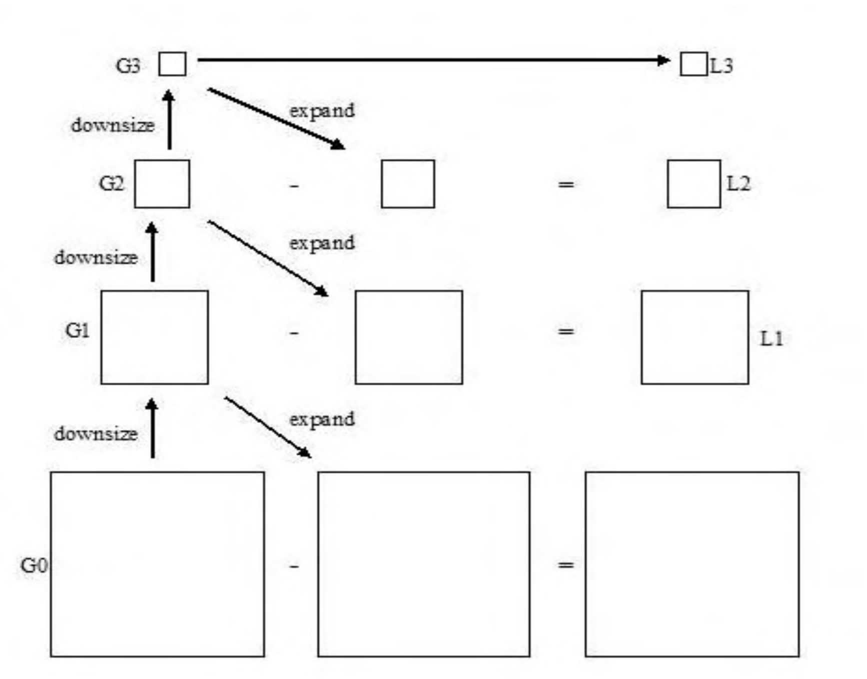


Figure 2 Example of a Gaussian {G} and a Laplacian {L} pyramid.

The expand() operation consists of an upsample operation and a Gaussian blur. The upsample doubles the dimensions of the image in width and height. More specific if t0 is the upsample output and g1 the input then t0(i,j) = 4*g1(i/2,j/2) if(i%2=j%2 == 0) ,0 else .The Gaussian blur will then smooth the upsample output using the same kernel as with the Gaussian pyramid. The smaller image (top level) of the Laplacian pyramid, called the residual, is always the same as the smaller image (top level) of the Gaussian Pyramid.

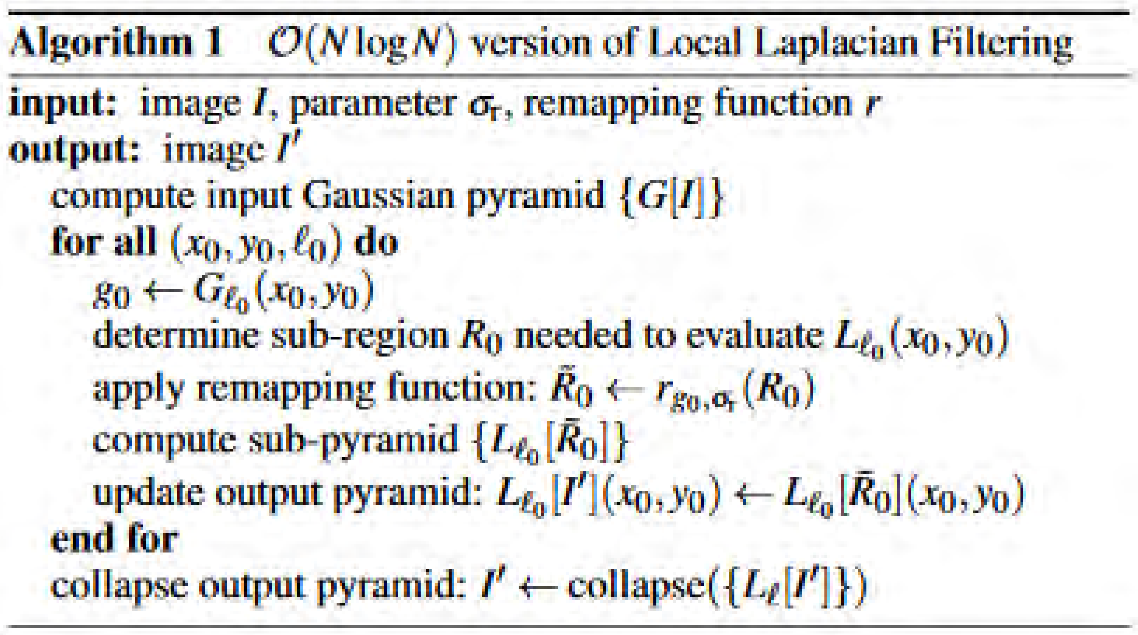
An interesting property of the Laplacian pyramid is that starting from the residual, we can collapse the pyramid, in order to get the image of the desired Gaussian pyramid level, or in the case where we fully collapse the pyramid, get the input image. Given that the residual contains all the low frequencies, we can expand it and add the Laplacian image of the previous level (same as add the high frequencies that we subtracted when we constructed the level) in order to get the Gaussian image of that level. Recursively if we consider the output of this procedure as the new “residual” we can continue to collapse until we get the input image.

2. Algorithm description

2.1 Preprocessing

The Local Laplacian algorithm, operates by filtering the luminance of an image, as well as the RGB channels, by processing each channel separately. The first preserves the original colors while the latter will also modifies the color contrast. The input image must also be scaled from [0,255] to [0, 1] floating point numbers (the Matlab code given uses double precision, but float is enough).In this work only the luminance filtering is supported. So as a preprocessing we convert the RGB values to grayscale (and save the ratios, to reintroduce colors after the processing) and scale them to [0, 1].

2.2 The algorithm



Algorithm 1 $O(N \log N)$ version of Local Laplacian Filtering

```
input: image $I$, parameter $\sigma_r$, remapping function $r$
output: image $I'$
  compute input Gaussian pyramid $\{G[I]\}$
  for all $(x_0, y_0, \ell_0)$ do
    $g_0 \leftarrow G_{\ell_0}(x_0, y_0)$
    determine sub-region $R_0$ needed to evaluate $L_{\ell_0}(x_0, y_0)$
    apply remapping function: $\tilde{R}_0 \leftarrow r_{g_0,\sigma_r}(R_0)$
    compute sub-pyramid $\{L_{\ell_0}[\tilde{R}_0]\}$
    update output pyramid: $L_{\ell_0}[I'](x_0, y_0) \leftarrow L_{\ell_0}[\tilde{R}_0](x_0, y_0)$
  end for
  collapse output pyramid: $I' \leftarrow \text{collapse}(\{L_\ell[I']\})$
```

Figure 3 The algorithm, as described by [(PARIS, HASINOFF, & KAUTZ, 2011)]

First we need to construct the Gaussian pyramid of the input image. On every level the value of the Gaussian image in a specific position will represent a global value of one window in the input image. If the absolute difference of a pixel value and the global value in such a window is bigger than a user defined parameter (sigma) then it will be considered an edge , if it is smaller a detail.
In Local Laplacian Filtering we want to calculate a Laplacian pyramid that when it is collapsed, it will give us the output image with the desired properties. In order to achieve that, each coefficient of each level, is calculated separately, one at a time taking into account only its local properties.
For every coefficient a different version of the input image is produced through the remapping function (which will be explained later).Since this coefficient is affected

only by local values , we only need a small region around it and therefore we process a window of the input image with the remapping function each time.
From the output image of the remapping function we construct a temporary Laplacian pyramid and we copy the corresponding coefficient, from the corresponding level of this pyramid to the output Laplacian pyramid. The procedure is repeated for every pixel of every level (apart from the residual which is copied from the Gaussian pyramid).
When the output Laplacian pyramid is completed, it is collapsed, producing the output image.

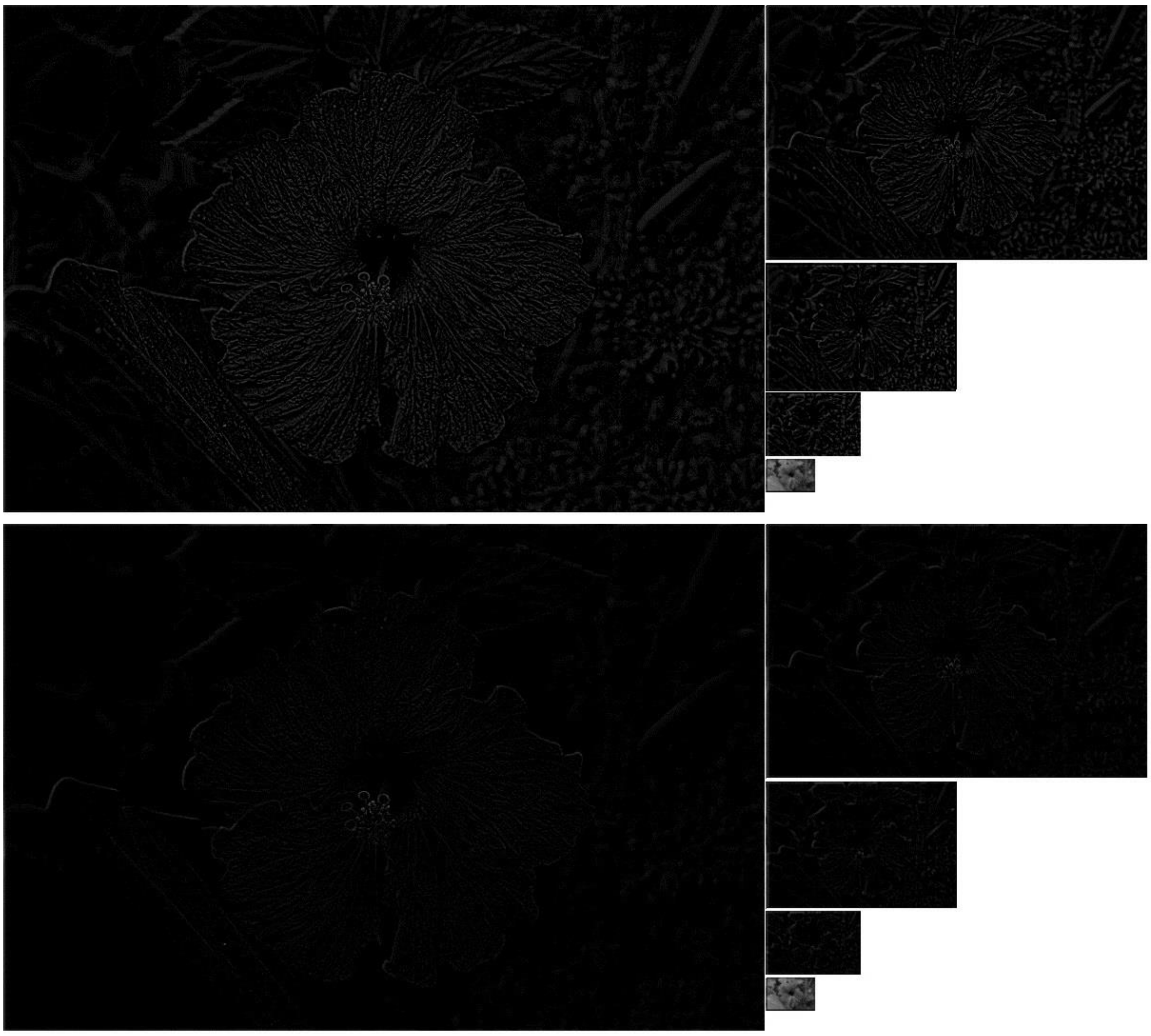


Figure 4 Example of a Laplacian pyramid produced with the Local Laplacian filtering method (top) versus a typical Laplacian pyramid (bottom)

2.2.1 Remapping function

The remapping function, is the function responsible for detecting (locally) edges and details in small windows and amplifying, or smoothing them. The inputs to the remapping function is a global value from the Gaussian pyramid and a pixel value from the window that is to be remapped .The global value g0= Gl0(x0, y0) is copied

from the position and scale/level that we calculate the coefficient. If we try to calculate the pixel (4, 5) position, on the third level of the Laplacian pyramid, then g0 should be the pixel at (4, 5) position on the third level of the Gaussian pyramid. This value, is global in a sense that it represents the intensities of all the pixels in the window that is currently remapped. Given a user defined parameter sigma, all values closer to g0 than sigma are considered edges and all those further than sigma are considered edges.
Edges and details are processed in a different way, which is defined from the user defined parameters alpha and beta .Alpha controls the details, and beta the edges throughout the functions Rd (), Re ().More specific:

$$r(i) = r_d(i) \; if \; |i - g_0| < \sigma_r \; and \; r(i) = \; r_e(i) \; otherwise$$

$$r_d(i) = \; g_0 \; + sign(i - g_0)\sigma_r f_d\left(\frac{|i - g_0|}{\sigma_r}\right)$$

$$r_e(i) = \; g_0 \; + sign(i - g_0)(f_e(|i - g_0| - \sigma_r) + \sigma_r)$$

where r(i) is the output of the remapping function and i is the windows input pixel value.
$f_d()$ is an S-shaped pointwise function where $f_d(\Delta)=\Delta^{alpha}$ .For 0<alpha<1 details are increased and for alpha > 1 details are smoothed. Finally $f_e(x) = beta * x$ 0<beta<1, for tone mapping and beta>1, for inverse tone mapping. In this implementation, and especially for the FPGA the beta is fixed at 1 so it does not process edges. Choosing a different parameter, will still work in most cases, but the results have not been tested thoroughly.

2.2.2 Determine the sub-region of the input image needed to evaluate the Laplacian pyramid coefficient

Since the window corresponds to one coefficient of the Laplacian output every time, we must decide the position and size of that window, in the input image. According to the algorithm, from the remapped version of that window we will have to construct an intermediate Laplacian pyramid, with level depth the same as the level of the output Laplacian pyramid that we try to evaluate. To calculate the level 0 for the output Laplacian pyramid for example we need to construct the Laplacian level 0 of the remapped window. For this , a 8x8 sized window would suffice , as only two levels of Gaussian pyramid will be constructed (The level 0 is of size 8x8 and the level 1 with size 4x4) and only one (8x8 sized) level of Laplacian pyramid .To determine the position of the window we must consider that the centermost pixels of this window will have bigger impact on the value of the coefficient .For the level 0 for example the pixel at (4,4) of the intermediate Laplacian pyramid , will be copied to the output Laplacian pyramid. This means that when we try to calculate the pixel (10,10) of the level 0 output Laplacian pyramid for example, the window must have column range

(6-14) and row range (6-14) on the input image. The center of the window, will now be the pixel (10,10) of the input image. To calculate the pixel (11,10) the window must be (6-15,6-14) and so on. For the next levels we must consider that one pixel of the Gaussian pyramid level 1 for example has a global value for a bigger window in the image (basically, it stores information about 4 pixels of the input image).The size of the level 1 window is 20x20 .Also the g0 must map to the center most pixel of the window when upsampled. This means that for g0 at (40,40) of the level 1 , we get a window centered at (80,80) of the input image which means that it has range (70-90,70-90).We could simply say that the stride of the windows for level 1 is 2.The next windows for the next levels are produced in the same fashion.

Level	Windows size	Stride
0	8x8	1
1	20x20	2
2	44x44	4
3	92x92	8

Remapped widow size and stride per level

2.3 Post processing

Finally we rescale the filtered output from [0, 1] to [0,255] and reintroduce colors using the ratios saved during the preprocessing.

2.4 Parallelism

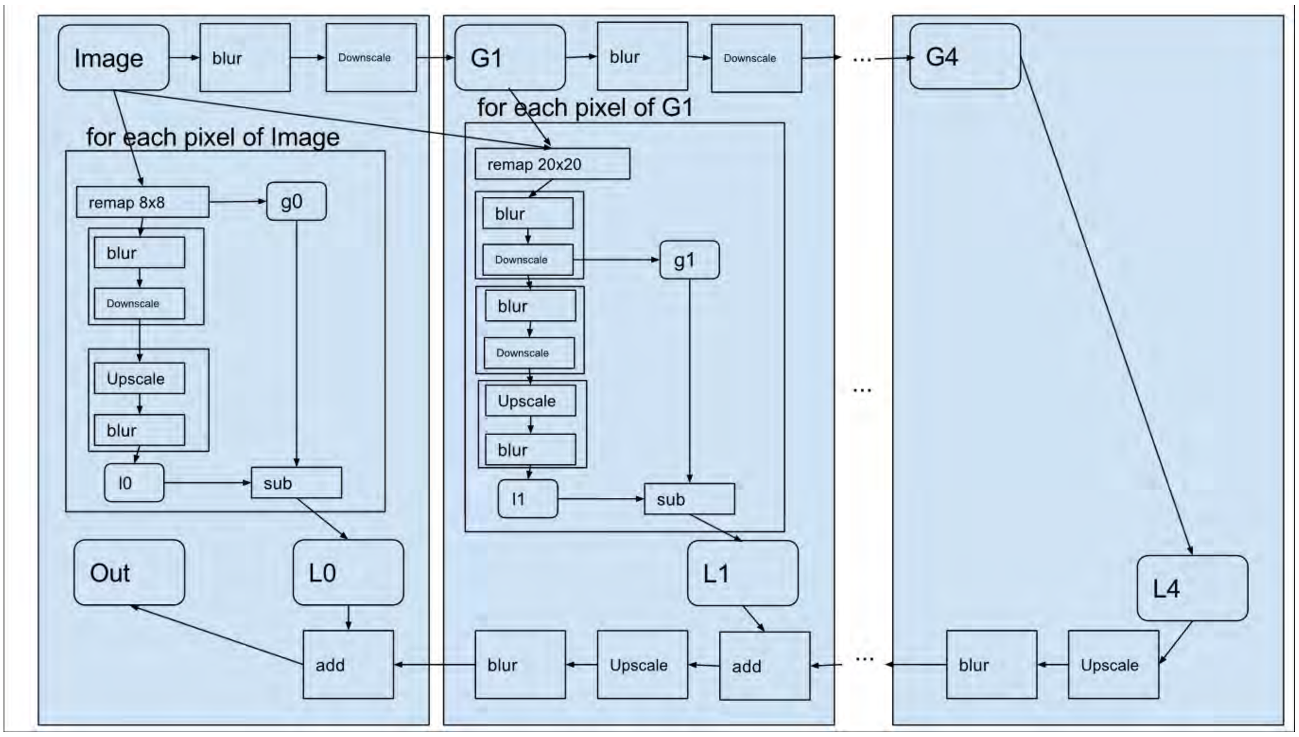


Figure 5 Block diagram of the algorithm.

The rectangles represent kernels, while the rounded rectangles represent stored data. The arrows show both the order of execution, and the data flow between kernels (these data only pass from one kernel to the next).

As we can see (from the block diagram image) the Gaussian pyramid construction , does not support so much parallelism between kernels, as every level requires its previous level completion,(unless a steaming like parallelism is allowed).The same is true for the Laplacian pyramid reconstruction , and for the local Laplacian pyramid construction as well.
However the bottleneck of this algorithm lies in the construction of the output Laplacian pyramid. In each iteration of this triple nested loop a small image is remapped and its (local) Laplacian pyramid is computed. Assuming that the Gaussian pyramid is complete, every one of these iterations is completely data independent from one another and therefore could execute in parallel if there were enough resources. The next bottleneck, is the “latency” of each loop. Fortunately more parallelism is supported inside every kernel in order to reduce it.
The kernel for the construction of the intermediate remapped image/window can produce each pixel of the image in parallel since every output pixel depends from one global value g0, the same for every pixel in the same window, and one pixel value, from the input image.

The Gaussian blur requires one column convolution, and one row convolution. In each of these kernels, parallelism is available since there is no accumulation like process, in any of them and each output purely depends on the inputs. Unfortunately in every pyramid construction, (unless a steaming like parallelism is allowed), every kernel has data dependencies from the one executed before it.

3. Implementation

3.1 Resources

3.1.1 The runtime

The runtime, uses the LLVM 3.7 compiler and the kernels run on top of OpenCL-1.2-4.5.0.8
All of the kernels were implemented in OpenCL and are invoked from a C program, using the Centaurus runtime. Using the runtime, data transfer and kernel invocations become quite simple
E.g.

```
#pragma acl task in(A) out(B) workers(numW) groups(numG)
label(“kernel”) device(0)
kernel(A,B,params);
#pragma acl taskwait label(“kernel”)
```

In this example the OpenCL kernel is executed in the device 0 (device 0 is the GPU and device 1 is CPU in out installation of the runtime) with numW as the number of workers-threads per block and numG as the total number (global) of workers in the device. The data transfer is automatically handled by the runtime and any data dependency is resolved before the kernel execution. For the data transfers in() and out() can be used to declare the direction of the data (in() means transfer data to the kernel, and out(), from the kernel).In the case above the data will only be transferred once to and from the device, meaning that in the next kernel invocation that will use the data A for example, the A data will not be updated, in the case that they will be changed from another device. In order to force the data to update we can use device_in(A)/device_out(B) instead. Finally when we don’t need to transfer data from one device to another we can use buffer(), to deny any transfer.
With the use of the taskwait pragma, we force synchronization on the execution flow.

3.1.2 The CPU

The system used for developing and testing has an i7-4820K @ 3.70GHz Intel CPU with 4 cores and 2 threads per core. Ideally this should give us at least 6x speedup, provided there is a good memory access pattern.

3.1.3 The GPU

The use of GPGPU (General Purpose GPU) computing is on the rise the last years. Thanks to the massive parallelism that GPU's natively offer, many applications including image processing can get a significant increase to performance if they can exploit this parallelism. In our system we use a GeForce GTX 770 GPU by NVidia. This GPU offers 8 multiprocessors and each multiprocessor can have up to 192 threads (called CUDA cores).The multiprocessors execute the threads in groups of 32 threads (called warps) that execute the same operations but with different data .The threads in each warp run completely concurrently.
From the side of the OpenCL programmer, the threads correspond to OpenCL work-items and they can be organized to work groups. All the threads inside a work group , can have synchronization points declared by the programmer .It is important for the overall performance that in each group every 32 of the work items , have the same execution flow , otherwise there will be divergence and less than 32 threads will run concurrently , resulting in larger execution times.
Of course every application needs data to process. Apart from divergence the bottleneck of an execution on a GPU is the high data transfer times to and from the memory. There are 3 basic types of memory in an NVidia CPU. The local memory of a thread, which is very fast and can only be accessed by the same thread. The shared memory of a group which can be accessed by all the threads inside a group, but not from threads of a different group and finally the global memory. The global memory is a slow but large memory that can be accessed by any thread of any group and also by the CPU. However it can perform very well if there is a coalesced access pattern. Before an application's execution starts, the CPU or a previous executed GPU kernel loads data to the global memory. During the application's execution the threads use that memory to load and store data. It is important to avoid unnecessary global loads and stores, and try to use the shared memory when the same data are needed by many threads inside a group.

As a GPU, the system uses a GeForce GTX 770.
The deviceQuery for this GPU returned the results below:

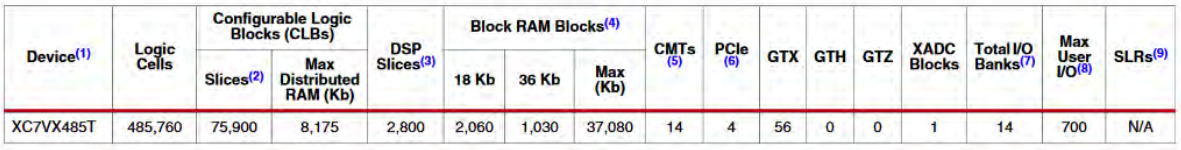
```
 CUDA Driver Version / Runtime Version                  7.5 / 7.5
 CUDA Capability Major/Minor version number:    3.0
 Total amount of global memory:                 2047 MBytes (2146762752 bytes)
 ( 8) Multiprocessors, (192) CUDA Cores/MP:     1536 CUDA Cores
 GPU Max Clock rate:                            1110 MHz (1.11 GHz)
 Memory Clock rate:                             3505 Mhz
 Memory Bus Width:                              256-bit
 L2 Cache Size:                                 524288 bytes
```

```
Maximum Texture Dimension Size (x,y,z)            1D=(65536), 2D=(65536, 65536),
                                                  3D=(4096, 4096, 4096)
Maximum Layered 1D Texture Size, (num) layers     1D=(16384), 2048 layers
Maximum Layered 2D Texture Size, (num) layers     2D=(16384, 16384), 2048 layers
Total amount of constant memory:                  65536 bytes
Total amount of shared memory per block:          49152 bytes
Total number of registers available per block:    65536
Warp size:                                        32
Maximum number of threads per multiprocessor:     2048
Maximum number of threads per block:              1024
Max dimension size of a thread block (x,y,z):     (1024, 1024, 64)
Max dimension size of a grid size    (x,y,z):            (2147483647, 65535, 65535)
Maximum memory pitch:                             2147483647 bytes
Texture alignment:                                512 bytes
Concurrent copy and kernel execution:             Yes with 1 copy engine(s)
Run time limit on kernels:                        Yes
Integrated GPU sharing Host Memory:                      No
Support host page-locked memory mapping:          Yes
Alignment requirement for Surfaces:               Yes
Device has ECC support:                           Disabled
Device supports Unified Addressing (UVA):         Yes
Device PCI Domain ID / Bus ID / location ID:       0 / 4 / 0
```

3.1.4 FPGA

FPGAs are also used in high performance computing and they also allow high parallelism in a different way than GPUs. An FPGA (Field Programmable Gate Arrays) is a device that consists of: an array of configurable logic blocks (CLB), ram blocks and a hierarchy of reconfigurable blocks .All these components can be programmed (and programmed) by a bitstream file, with the use of a hardware description language and a bitstream synthesis tool, to produce custom hardware applications. The advantage of FPGAs over GPU’s and CPU’s is its low energy consumption , and the fact that the programmer can define most of the system’s architecture and therefore exploit furthermore the parallelism of an application, much like creating an ASIC. The basic disadvantages are, that there must be enough resources in the FPGA for the application and that there are timing constraints that must be met, in order for the design to be functional. Unfortunately developing an FPGA application in HDL is very time consuming because it requires much more analysis of the application in order to achieve the minimum possible latency of a design and mostly because hardware descriptions language are hard to use , compared to software languages. The development of high level synthesis tools comes to solve this problem by transforming code written in a higher level language to an HLD. In this implementation and since there is an FPGA by Xilinx on the system, I used the Vivado HLS 14.4 tool with the help of the Vivado documentation [(Xilinx, Xilinx. Vivado Design Suite User Guide, High-Level Synthesis.)] to transform C code to Verilog and the Vivado 14.4 tool to produce the bitstream from the Verilog files. In order for the Vivado HLS to produce Verilog code, some coding conventions must be followed and in order for this code to be efficient, some pragmas and directives provided by the tool are used. Their use and the impact to the results will be explained along with the implementation.

The system has a VC707 FPGA board connected via PCIe. The FPGA on the board is a XC7VX485T virtex-7 FPGA and it has more than enough resources. The resources of the FPGA are presented below.



Device(1)	Logic Cells	Configurable Logic Blocks (CLBs)		DSP Slices(3)	Block RAM Blocks(4)			CMTs (5)	PCIe (6)	GTX	GTH	GTZ	XADC Blocks	Total I/O Banks(7)	Max User I/O(8)	SLRs(9)
		Slices(2)	Max Distributed RAM (Kb)		18 Kb	36 Kb	Max (Kb)									
XC7VX485T	485,760	75,900	8,175	2,800	2,060	1,030	37,080	14	4	56	0	0	1	14	700	N/A

Figure 6 Virtex 7 specifications by (Xilinx, 7 Series Overview)

For the implementation, I also used the 1 GB DDR3 ram of the board.

3.2 C Implementation

The first step of the implementation, was to translate the code from Matlab, where the filter is originally implemented, to C. In this implementation I set the level of pyramids to 5 .I noticed that more than 5 levels do not contribute so much to the result , especially for smaller images. After that the following kernels were distinguished:

The Gaussian pyramid kernel

As described above, the first step of the algorithm is to construct the Gaussian pyramid of the input image and for that a Gaussian blur followed by a downsample kernel were used, for each level. The construction of the Gaussian pyramid is very fast compared to the construction of the Laplacian pyramid so I did not tried to optimize this step of the algorithm.

The remap kernel

This kernel has as inputs, the input image, the location and size of the window, and one value from the Gaussian pyramid. It uses the remapping function to produce the intermediate remapped window.

The local Gaussian blur kernel(s)

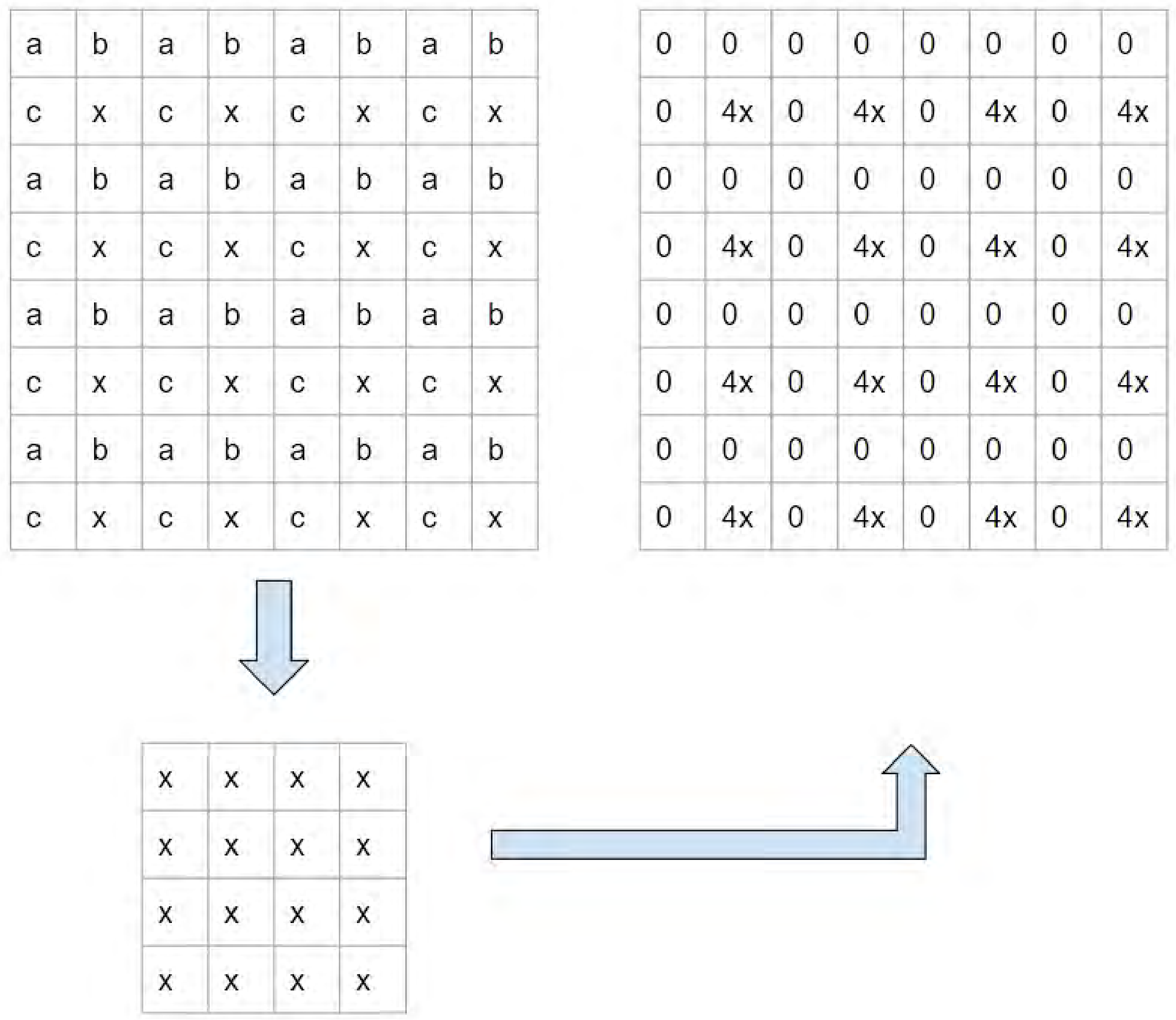
Applies the Gaussian blur, as explained, to a window. In C it is the same kernel, used to construct the Gaussian pyramid. The Gaussian blur consists of two kernels (convolution row and convolution column) executed in series

The local downsample kernel

The local downsampling kernels halves a sub-image in both width, and height. To achieve that it copies one pixel from a 2x2 window of the input to the output sub-image. It is interesting here that the kernel must choose to copy the one pixel from the 2x2 window that would appear in the input Gaussian pyramid as well and decimate the others.
For example if the remap kernel , remaps from the location (0-8,0-8) of the input image then the local downsample kernel would copy the pixels (0,0),(2,0),(4,0),(6,0) of its input to create the first line of the downsampled output. But if the remap kernel , remaps from the location (1-9,0-8) of the input image , then the kernel would have to copy the pixels (1,0),(3,0),(5,0),(7,0) of its input because the pixel (0,0) of the input sub-image which corresponds to the pixel (1,0) of the input image would never appear to the global Gaussian pyramid.

The local upsample kernel

Doubles the size of an image by matching one pixel value (multiplied by 4) of the input image to a 2x2 window on the output image .Same as the downsample kernel, the upsample much match this value to the correct location in that 2x2 window in order to match the Gaussian image that it will be subtracted from. All the other values in that window must be zero (the ones decimated by the downsample kernel).



a	b	a	b	a	b	a	b
c	x	c	x	c	x	c	x
a	b	a	b	a	b	a	b
c	x	c	x	c	x	c	x
a	b	a	b	a	b	a	b
c	x	c	x	c	x	c	x
a	b	a	b	a	b	a	b
c	x	c	x	c	x	c	x

x	x	x	x
x	x	x	x
x	x	x	x
x	x	x	x

0	0	0	0	0	0	0	0
0	4x	0	4x	0	4x	0	4x
0	0	0	0	0	0	0	0
0	4x	0	4x	0	4x	0	4x
0	0	0	0	0	0	0	0
0	4x	0	4x	0	4x	0	4x
0	0	0	0	0	0	0	0
0	4x	0	4x	0	4x	0	4x

Figure 7 Example of the downsample and the upsample operations

As an example if the downsample kernel chooses the x from the 2x2 widow (a,b,c,x) they must return to their original positions from the upsample kernel

The local subtract kernel

Subtracts 2 images, to produce the Laplacian level

The kernel for collapsing the output Laplacian pyramid

The last step of the algorithm expands the residual, adds the expanded image with its previous level and repeats this process with the output image, as the new residual. For this a Gaussian blur was used, followed by an upsample kernel, and a kernel that adds the values of two images.
The first and the last kernels were not given much attention because they are not the bottleneck of the execution time and their implementation is trivial.

3.2 CPU and GPU runtime Implementations

In order to use the massive parallelism that the GPU offers, I tried to produce a whole line of an output Laplacian pyramid level, in parallel. Therefore, the loop that goes across the columns of the output Laplacian pyramid is now embedded inside the kernels and the triple nested loop became a double nested loop (one loop for each pyramid level and one loop for each row output).
Because each value of the Gaussian pyramid corresponds to a window, the memory requirements grow after the execution of the remap kernel. It is only reasonable that the data should stay in the device that executes the kernels as buffers (using the buffer() pragma) until the intermediate windows and pyramids are no longer needed in order to avoid data transfer costs .For this reason heterogeneous kernel execution was not preferred.
The inputs to the device are the input image (only once) and the Gaussian level (again once for each level) that we need, in order to produce the output Laplacian level. The output is the Laplacian output level, but produced and transferred from the device one line at a time.

After this short analysis, I implemented all the local kernels in OpenCL, to assign their execution to the CPU and GPU devices.

The remap kernel is the most interesting kernel here. Because the windows are overlapping, I used the shared memory of the GPU to cache the input image and reduce the access to the global memory, which is very slow compared to the shared memory. Since the size and the number of the windows changes from level to level. The number of threads in each group changes dynamically from level to level, to match the number and the size of the windows, but it is always bigger than 132.The principal of the kernel stays the same. Every time, the creation of a number of windows is assigned to a thread group.
The first stage of the kernel is to load from the global memory , the input image pixels required to create one line of the windows , as well as the g0 value (stored in a register).At this stage, every thread loads and each of the loads from the global , and the stores to the shared memory , are coalesced. At the next stage a number N of threads , is responsible to create one line of the output window through an iteration with stride N and send the output data , back to the .Then , the control flow of the kernel returns to stage 1 and creates the next line of the remapped windows. The total number of thread groups is the total number of windows divided by the number of windows per group.

The Gaussian blur , is composed of a vertical(column) convolution , and a horizontal (row) convolution , with the filter {.05, .25, .4, .25, .05}.Basically , the implementation of these kernels and especially of the column convolution , is an adaptation of the 2D convolution , that is offered as an example by NVIDIA (NVIDIA, n.d.) , to the concept of windows that we use. More specific, in the column convolution every thread group will process one line of the window. Every thread will store from the global memory or the zero padding, to the shared memory. Then every thread, but the four responsible to store the padding, will calculate, one output pixel, of the convolution operation. The memory stores and loads are coalesced in the column convolution. The row convolution has the same principals, but it loads/stores rows of the input image. Because of this, the accesses are not coalesced and there is a loss in performance.

The upsample, and downsample, kernels are mostly data transfers and logistics in order to decide the locations, of the reads and writes. Because there is no data reusability, the shared memory is not used.

The subtract kernel consists of two loads (one from the Gaussian, and one from the expanded image) from the centers of two sub-images, one subtraction, and the coalesced store of the result. It is a small kernel in terms of operations, but it has increased execution time, because in that point the data, are transferred from the device memory, back to the system memory.

3.2.1 Profiling

In this chapter I present the performance numbers for the implementations so far, along with the observations that led to the optimizations.

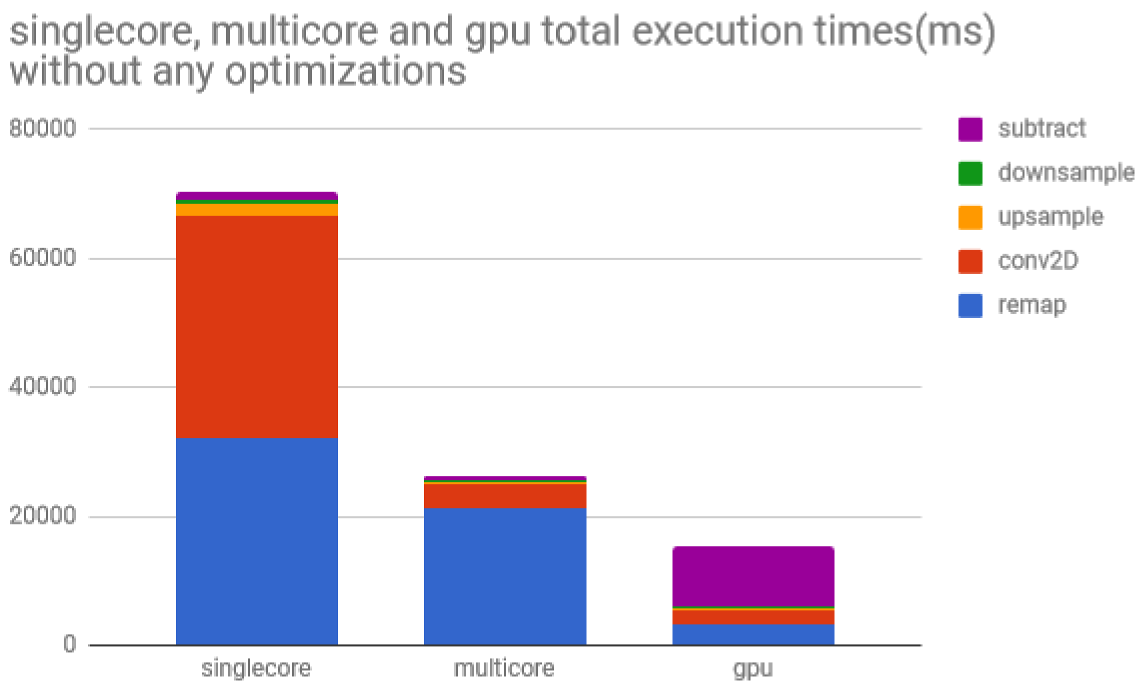


Figure 8 Total execution time per device for a 1024x768 image

Time in ms	Single core	multicore	GPU
remap	32270,31336	21246,20368	3129,932547
conv2D	34438,03265	3627,996718	2349,585979
upsample	1887,202613	437,333315	263,075615
downsample	444,220504	407,834407	252,746314
subtract	1277,633131	466,213931	9327,063624
total	70531,79294	26211,11913	15357,10428
Pure kernel execution			4.898

As we can see in the diagram and the spreadsheet above in the naive implementation the remap kernel is the bottleneck in the CPU. This is reasonable, even though it is executed only once per window, because there are many floating point operations in double precision.The blur kernel (Conv2D) is the second most computational intense kernel and it is executed multiple times per window. Therefore in the single core version of the code, where there is no parallelism to make up to the floating point operations, it occupies a significant percentage of the execution time.
In the GPU the subtract takes up to 60% of the execution time as we can also see on the diagram below. This happens because at that point the data are also transferred back to the CPU.

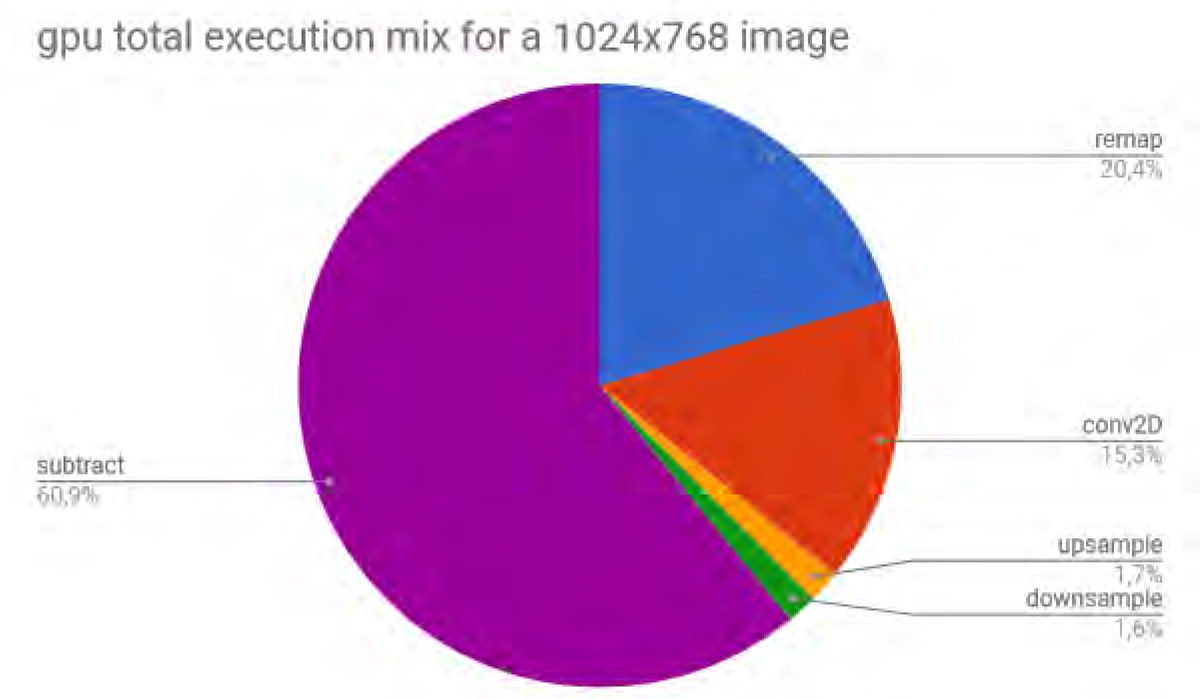


Figure 9 GPU kernel's execution mix for the first implementation

By measuring only the kernel execution time, and not data transfers or kernel overheads we can see that the above are the basic problem in this implementation, since the kernel execution is only 32% of the total execution time.

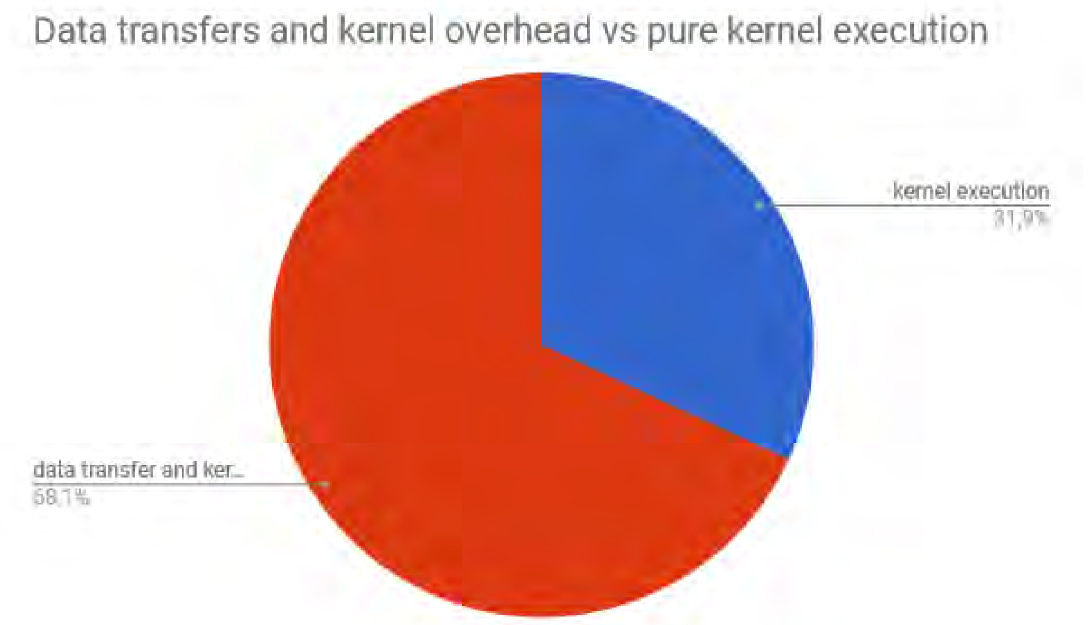


Figure 10 GPU memory transfers and kernel overhead vs pure kernel execution

3.3 The first optimization

The first optimization was to decimate unnecessary operations that came up from the Matlab code. Since in the construction of the output Laplacian pyramid only one level of the intermediate Laplacian pyramid is required, the calculations of all the other levels is redundant. The most extreme example is the case of the level 3.In the Matlab code (and in the c code that came up).An intermediate Laplacian pyramid of 3 levels is constructed, and only one pixel from the third level is used. Also, the lower levels of the output pyramid are the most time and memory consuming to create and store. After the optimization only one level of the intermediate Laplacian pyramid is produced. Another optimization in the same concept is in the local subtract kernel. The local subtract kernel produces a Laplacian level by subtracting a Gaussian level and an expanded image. From this Laplacian level, only the center most pixel will be used/stored to the output Laplacian pyramid. The unnecessary subtractions were decimated.
So far, in the implementation all the windows of a level, have the same size. The windows on the sides of the image used a padding to cover the missing pixels, and the results of that were noticeable in the output image. In order to correct this I decimated any remapping or blurring operation at those pixels, reducing the size of the window.

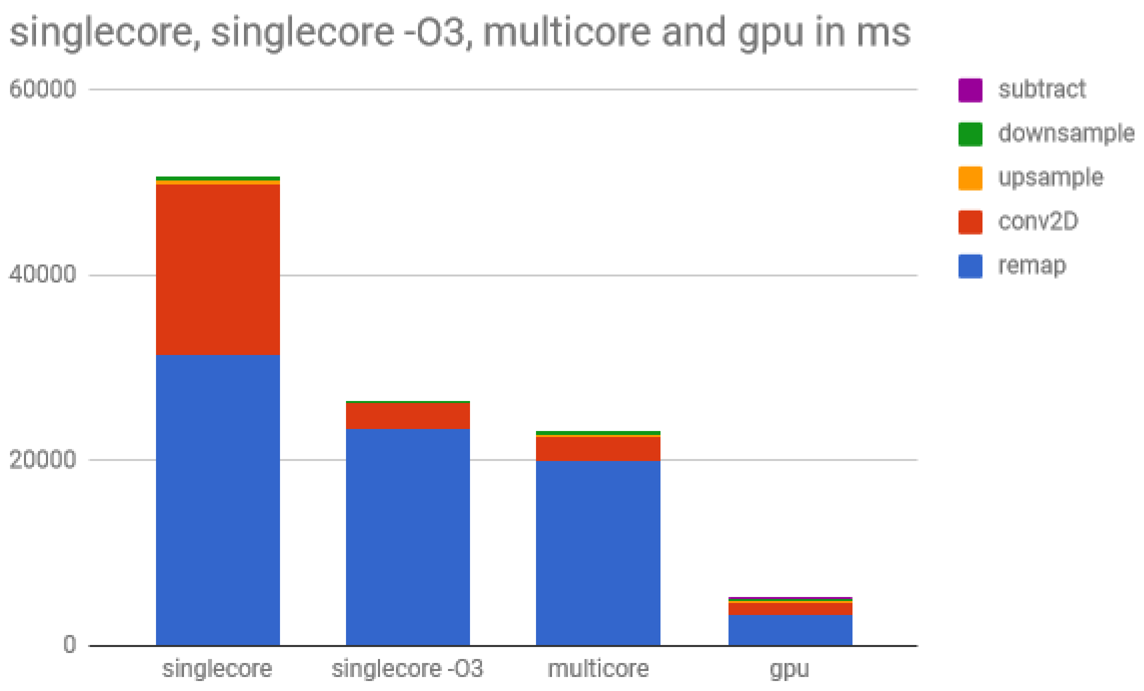


Figure 11 Total execution time per device for a 1024x768 image on the optimized version

	Single core	Single core -O3	Multicore	GPU
remap	31424,67386	23450,23936	19933,63111	3369,512597
conv2D	18415,64916	2704,891878	2661,9427	1261,508697
upsample	380,474309	90,235924	209,893086	175,603085
downsample	442,000749	155,411974	444,714919	250,618178
subtract				172,292132
total	50879,76709	26576,69183	23289,38039	5237,937497
clean kernel execution				4063,442976

In this experiment I also added the single core version, compiled with the -O3 flag the compiler gave very good results and this was expected mostly because this flag enables vectorization which is another form of parallelism.
As we can see the GPU benefits the most from the optimization. The clean kernel execution has stayed almost the same , but the kernel overhead and the data transfers have been reduced from 10.459ms to 1175 ms .As result the percentage of the subtract on the execution time has dropped significantly , and now the GPU execution mix looks as expected , with the remap kernel being the bottleneck of the implementation

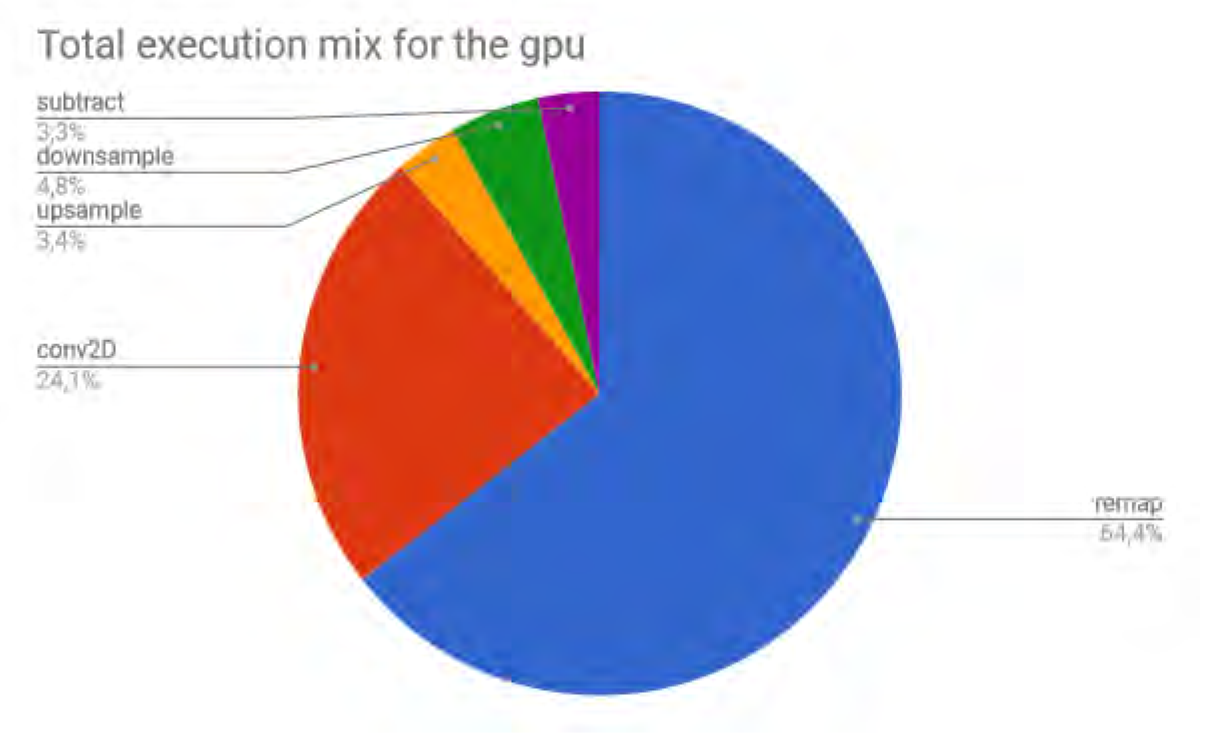


Figure 12 Total kernel execution mix for the GPU after the first optimization

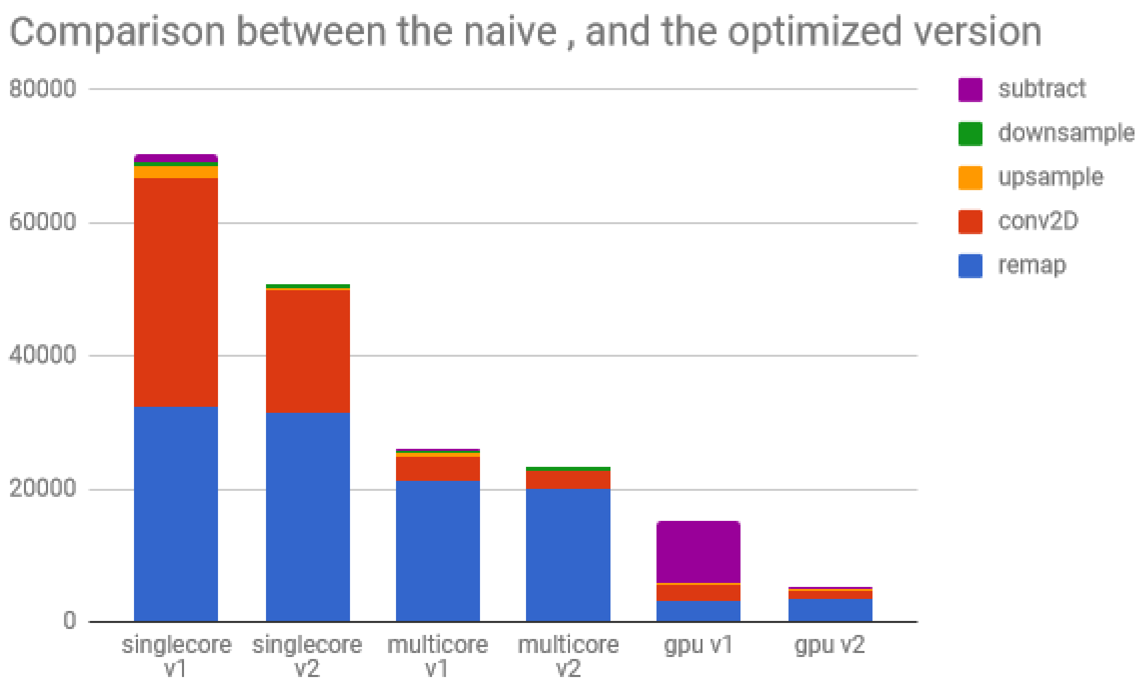


Figure 13 Initial implementations (v1) versus optimized (v2)

Comparing the execution times before and after the optimization, we can see that the real bottleneck of the implementation, the remap kernel, remains almost the same. In the CPU version there is a slight reduction on its execution time because some windows are now smaller, but on the GPU for the same reason we have a slight increase in this kernels execution time. The smaller windows simply add more logistics in the kernel's execution and cause some of the threads to be inactive when the GPU processes the edges of the image. On the other hand we can notice a decrease in the execution times of every other kernel, and this happens because they are not executed so many times unnecessarily.

3.4 Reducing accuracy

The final optimization I tried was to reduce the accuracy of the values and the operations from doubles to floats. In terms of performance the only the GPU had some noticeable speedup, while the difference in the results accuracy was not visually observable in any of the devices.

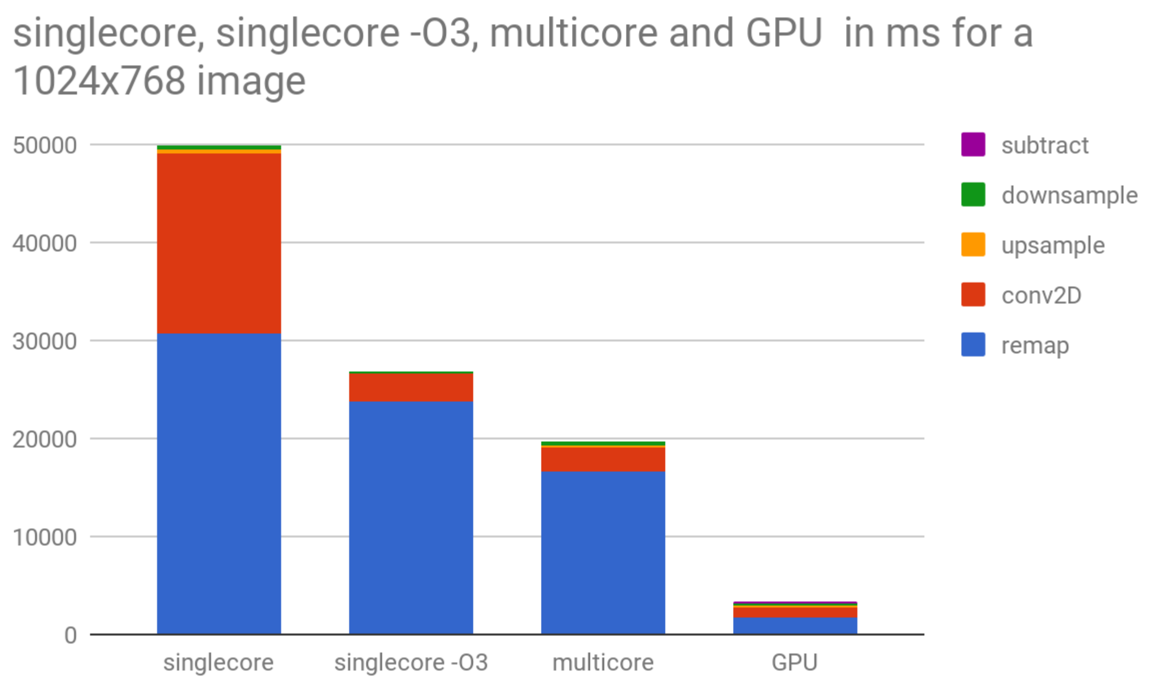


Figure 14 Total execution time per device for a 1024x768 image for the floats version.

	Single core	Single core -O3	multicore	GPU
remap	30624,03055	23731,19522	16647,40052	1758,865629
conv2D	18362,20489	2858,628081	2486,355789	1064,223982
upsample	387,392409	99,398928	213,78107	176,418784
downsample	435,827845	197,734728	425,542343	249,138234
subtract				165,97846
total	50025,41502	27065,71838	19808,04367	3673,615492
clean kernel execution				2.325

As we can see here this optimization was also more beneficial for the GPU implementation, mostly in terms of clean execution time. The GPU has almost 2x speedup in the execution just by decreasing the accuracy from doubles to floats.

3.4.1 Effect on accuracy

With the result of the single core of the first optimization as the golden output, we notice that the multicore and the GPU outputs present a drop in psnr, even though the results are not visually noticeable. This is probably caused by the changes in the order of operations, induced by the parallelism. In the floats version, the psnr decreases only in the single core implementation, since the multithreaded and the GPU already produced not so accurate results probably due to the differences between the LLVM - OpenCL compilers.

	Single core	Single core -O3	Multicore	GPU
doubles	inf	inf	76	76
floats	144	144	76	76

Psnr of the different versions for doubles and floats after the first optimizations, with respect to the single core output for doubles (golden).

Speedup

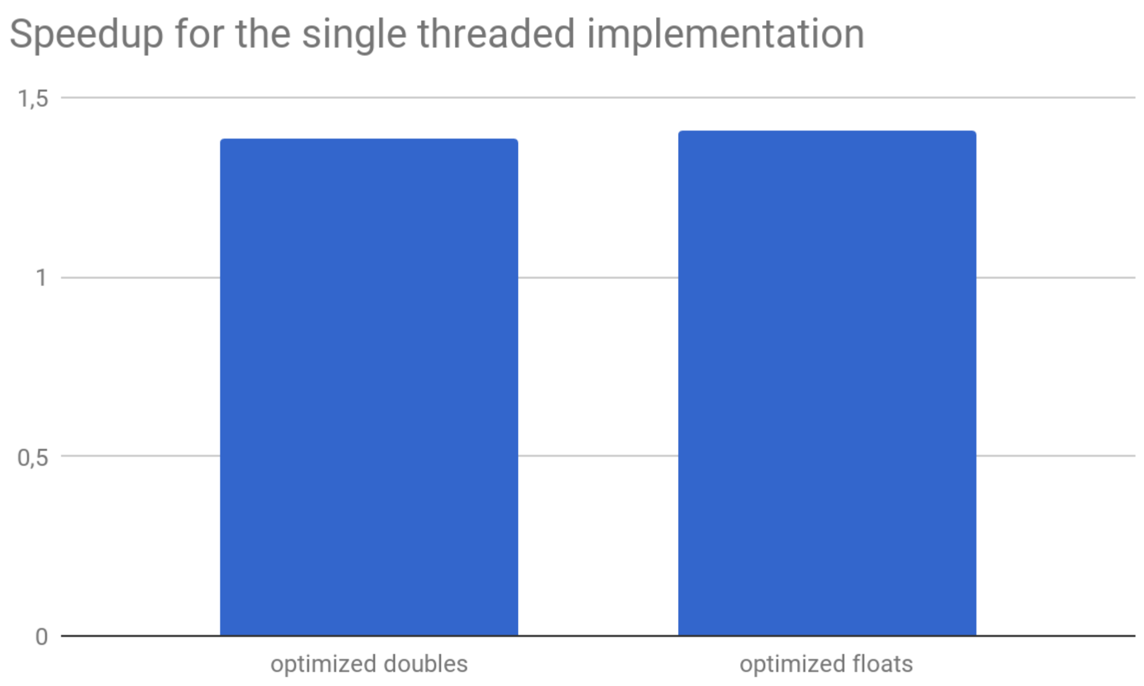


Figure 15 Speedup of the optimized versions, compared to the initial implementation (comparison between single thread versions)

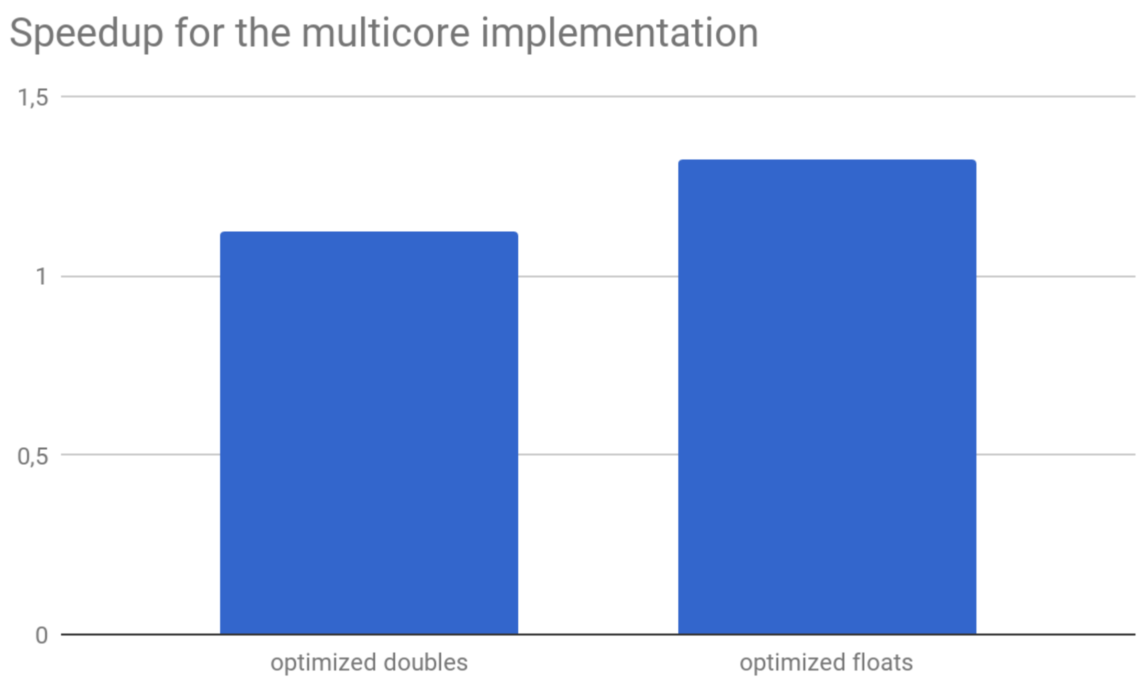


Figure 16 Speedup of the optimized versions, compared to the initial implementation (comparison between multicore versions)

As expected, the speedup in the single core version was very little (x1,38 for the doubles and x1,4 for the floats), since the bottleneck was the remap kernel, that was not optimized. The same and even worse occurs in the multicore version with x1,125453692 for doubles and x1,323256328 for floats.

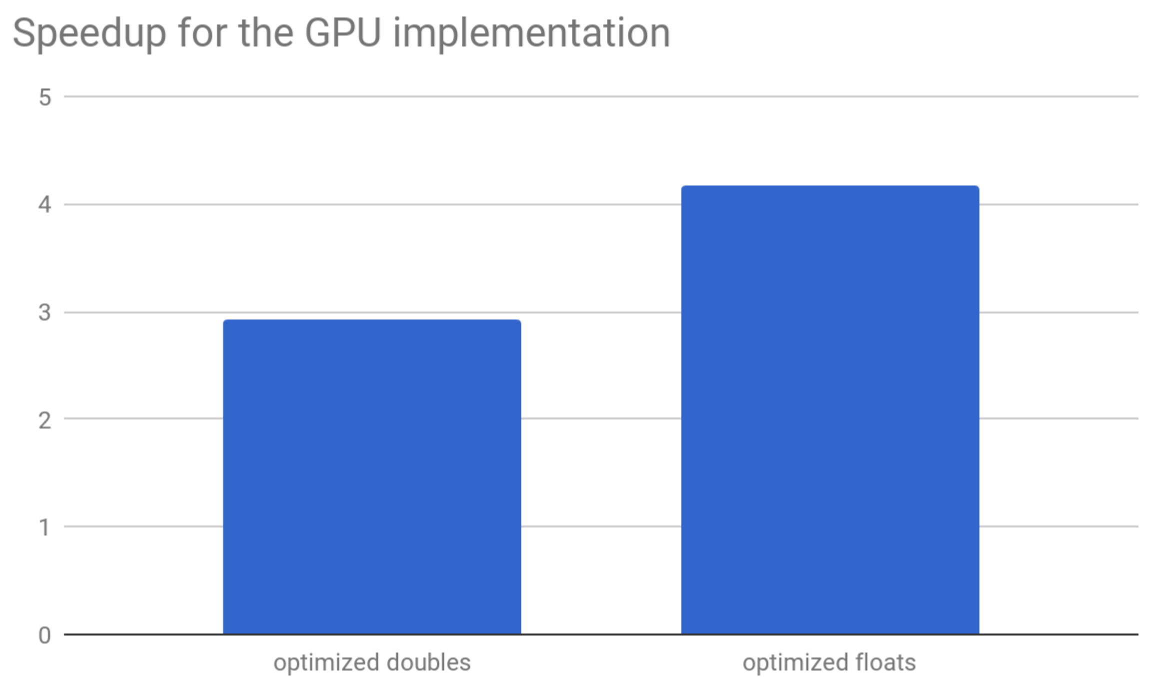


Figure 17 Speedup of the optimized versions, compared to the initial implementation (comparison between GPU versions)

For the GPU on the other hand there is enough parallelism to make up to the time consuming floating point operations in the remap kernel , we notice a 3x speedup for the doubles and a 4x speedup for the floats implementation making the GPU significantly faster than the CPU. The GPU has a 19x speedup versus the single core faster version, and 5x versus the multicore version.

3.5 FPGA Implementation

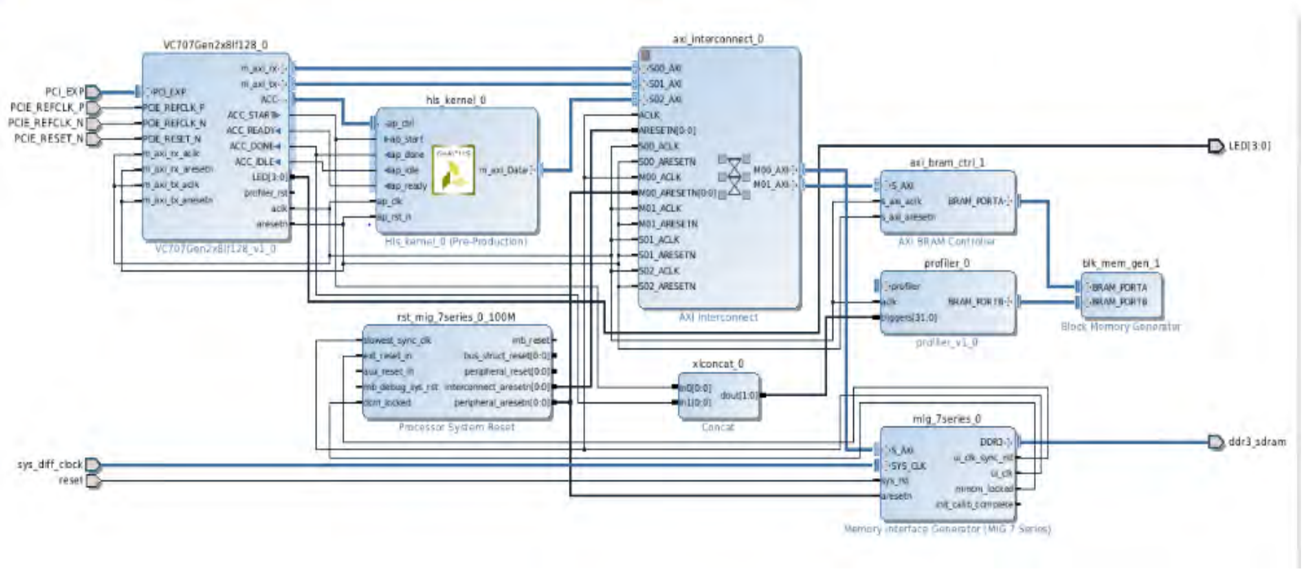


Figure 18 Block diagram of the system, provided by Vivado HLS.

The VC707Gen2x8lf128_0 is actually the RIFFA that is responsible for the PCIe transfers and the start of the accelerator. The hls_kernel_0 is the accelerator for the local Laplacian filter.

For the FPGA implementation , I used a RIFFA framework provided by the runtime and I built the accelerator using Vivado HLS .RIFFA (Reusable Integration Framework for FPGA Accelerators) is a simple framework by USCD for communicating data from a host CPU to a FPGA via a PCI Express bus (Jacobsen & Kastner, 2013). Through the RIFFA, I sent the required data from the host to the FPGA's ddr via PCIe .When the data are sent the accelerator starts and stores the output data to the ddr as well. By the end of the accelerators execution, the output data, are sent back to the host thought the RIFFA. In this implementation the accelerator is executing the whole flow of the construction of the output Laplacian pyramid. That said, the input is the Gaussian pyramid of the input image and the output the Laplacian pyramid. The whole design runs on the same clock with 4ns period.

3.5.1 The accelerator architecture

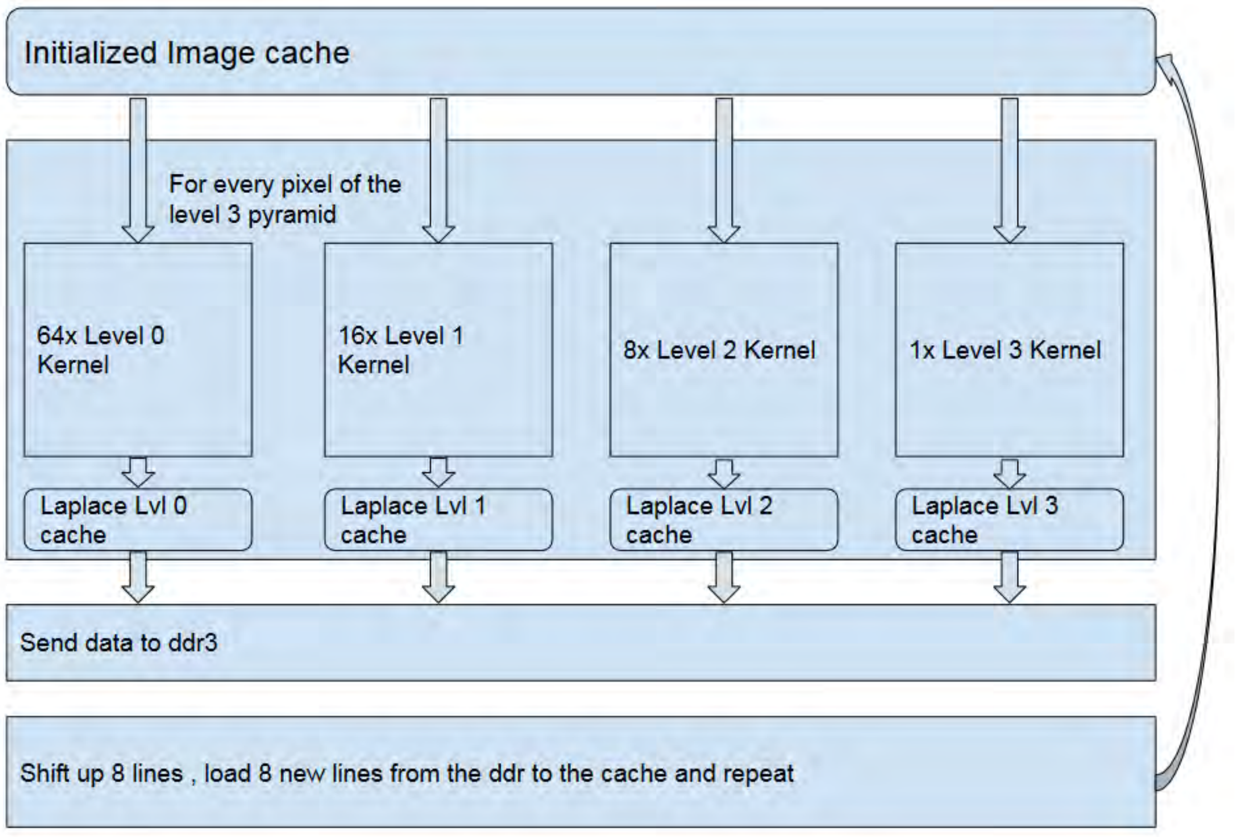


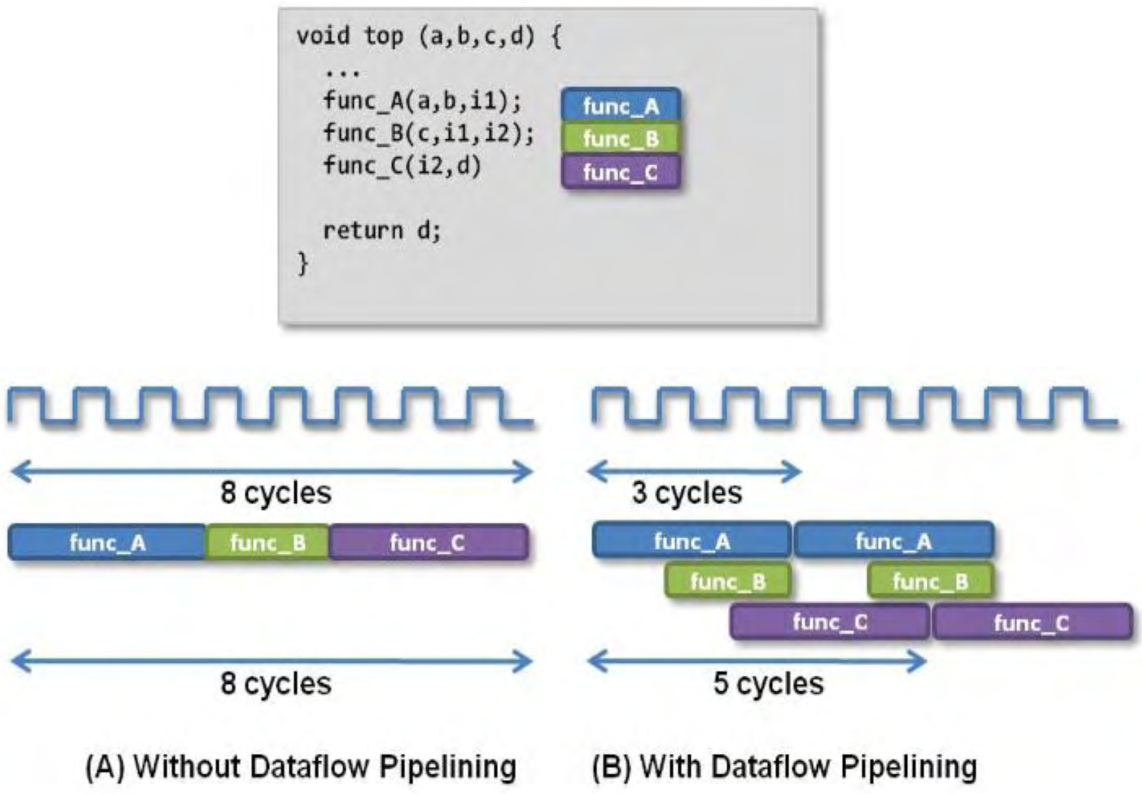
Figure 19 Block diagram of the FPGA accelerator implementation.

The basic hardware kernels, produce one output coefficient of the Laplacian image. Therefore there are 4 hardware kernels on the design (one for every level) executing iteratively , each time with different coordinates on the image as input .Basically , each hardware kernel consists of the body of the triple nested loop in the c code for the various levels. The smaller software kernels described in the previous implementations still exist, but are inlined in the hardware kernels. Since every one of the software kernels does not need the previous to finish its execution and it can start as long as the previous kernel produces some results, the dataflow pipeline type was preferred, implemented using the pragma #pragma HLS dataflow .For example the first column blur kernel can start its execution as long as the remap kernel produces its first remapped output pixel. The row blur kernel can start as long as the first 3 lines of the column blur kernel are produced, and the downsample and upsample can start when the first output pixel of every previous kernel is produced. Since there is little data reuse inside this triple nested loop , the whole flow of the kernels is implemented with FIFOs and the data are streaming for kernel to kernel using the “HLS_stream.h” library and the class hls::stream<> , provided by Vivado. Some buffers were used though, most of them in the Gaussian blur kernel, to store the 4 lines required by the row blur kernel and others to store the center most pixel of the temporary Gaussian pyramid level required to calculate the Laplacian coefficient. The four kernels that produce the Laplacian coefficient for each level, could execute in parallel, if it wasn't for the timing restrictions that the memory accesses set. Because all of these kernels load most of their data from the input image, and because the access to the ddr is expensive I decided to cache some data to a block ram. The cache stores all the data of the input image required by the kernel that produces the level 3 of the output pyramid for a whole output line. Therefore, the size of the cache is 92*image_width. This cache is also used by the other three kernels and provides the data required to produce 8 lines of the level 0 Laplacian output, 4 lines of level 1 and 2 lines of level 2.After the above outputs are produced the cache is refreshed by shifting its values by 8 lines (line 8 becomes line 0) and the 8 next lines load from the input image in the ddr, to the cache to fill the “empty” indexes .This process is repeated until the whole output Laplacian pyramid is constructed.

For the optimization of the software kernels I used the #pragma HLS pipeline optimization pragma. Each of these kernels achieved an II=1 (Iteration Interval) which means that it can process new data in every clock cycle. The latency of the software kernels however, adds a bottleneck to the accelerator.

3.5.2 The dataflow optimization

The use of the optimization directive #pragma HLS dataflow greatly improved the performance of the accelerator. According to Xilinx the dataflow directive ensures that each function will start its operation, as soon as input data are ready and it will not wait until the previous function finishes its execution.


```
void top (a,b,c,d) {
  ...
  func_A(a,b,i1);
  func_B(c,i1,i2);
  func_C(i2,d)

  return d;
}
```

Figure 20 Example of Dataflow pipelining by the Vivado Design Suite User Guide for High-Level Synthesis (Xilinx, Xilinx. Vivado Design Suite User Guide, High-Level Synthesis.)

The Dataflow pragma optimization improved the latency of the hardware kernels by far:

	Dataflow	No Dataflow
Level 0	205	938
Level 1	546	2820
Level 2	2082	10803
Level 3	8663	44528

Kernel latency and iteration interval for dataflow and non-dataflow design.

The transfer times to and from the FPGA are relatively small 20ms for a 1028*768 image and 452 ms for a 4096x4096 image.

3.5.3 Area

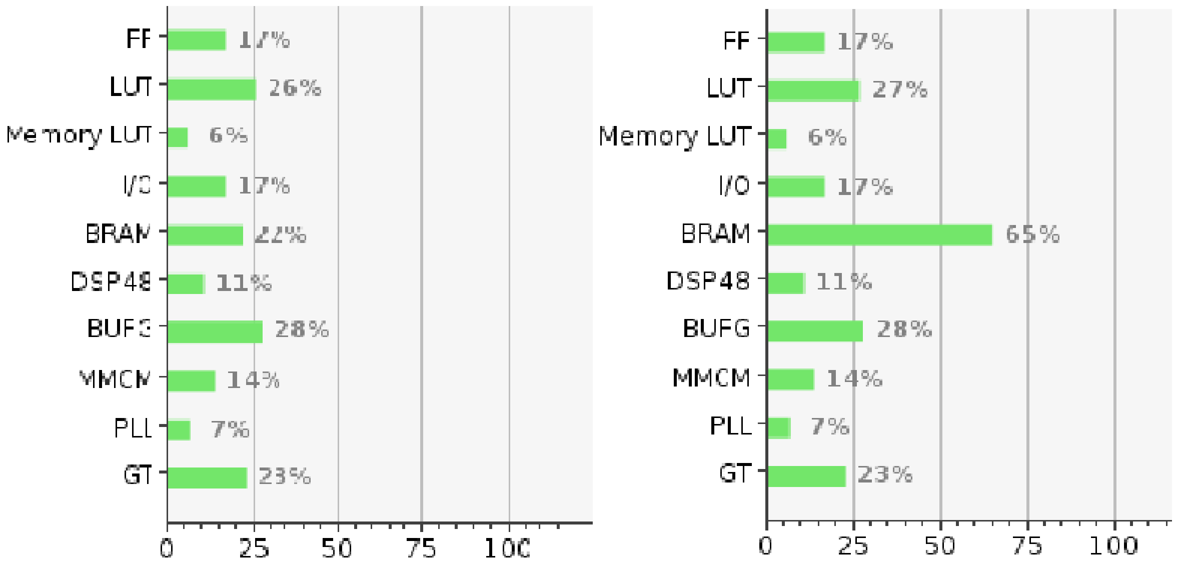


*Figure 21 Utilization % of the Virtex 7's resources for a 1024*768 image (left) and a 4096*4096 image (right)*

As we can see from the post-Implementation utilization results of the Vivado tool, there is about 30% utilization for a small image and a 65% utilization for a big image. The image cache in the accelerator results in more Bram usage, the for bigger image row sizes.

3.5.4 Accuracy

The FPGA performed well in terms of accuracy with not visible differences from a 1024x768 image processed from the CPU

	Single core	Multicore	GPU	FPGA
doubles	Inf	76	76	--
floats	144	76	76	77

The remapping function uses the pow(x, alpha) function to process details. Unfortunately this function is not supported for synthesis from the Vivado HLS. To overcome this problem we set the alpha parameter to 0.5 and swapped from pow, to sqrt .This is possibly one of the reasons for this insignificant drop in psnr.

3.5.5 Execution time and comparisons

The execution time of the FPGA was 2943,289219 ms for a 1024x768 image which gave a 24x speedup compared to the initial C implementation. For this and similar sized images the FPGA had the lower execution time.

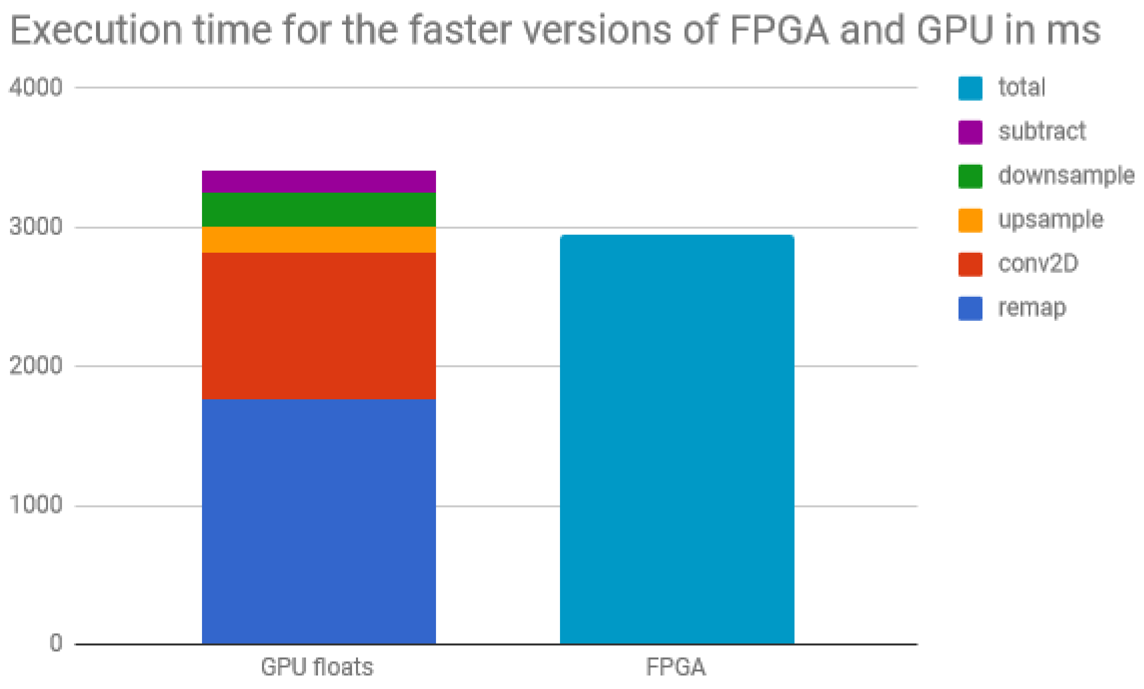


Figure 22 Execution time for a 1024x768 image, for the faster implementations

Finally I present a graph of all the implemented versions for the execution times of the same 1024x768 image

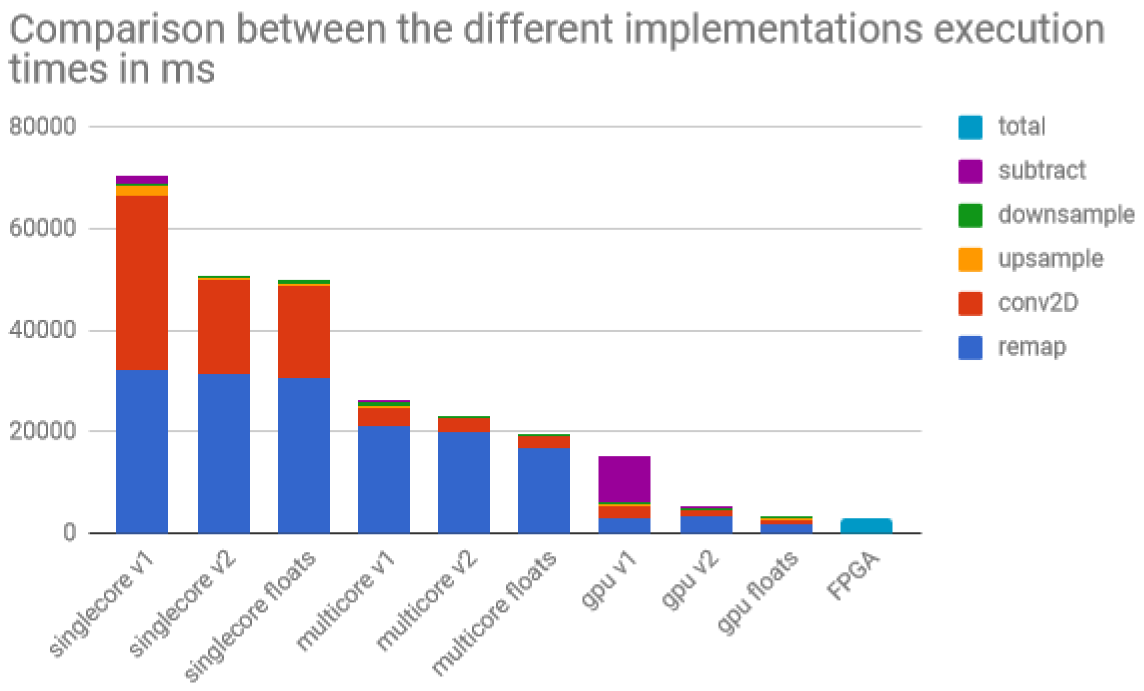


Figure 23 Execution times for a 1024x768 image

For smaller images, the FPGA performs even better compared to the GPU

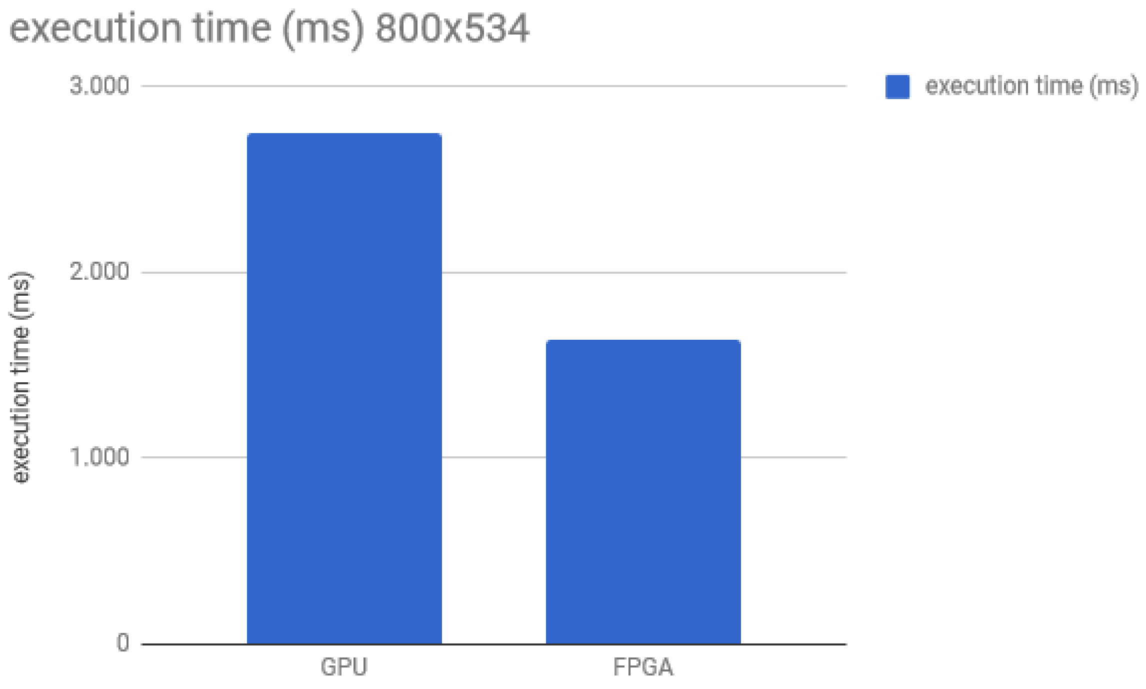


Figure 24 Comparison between FPGA and GPU for a small image

But for bigger images the GPU is faster

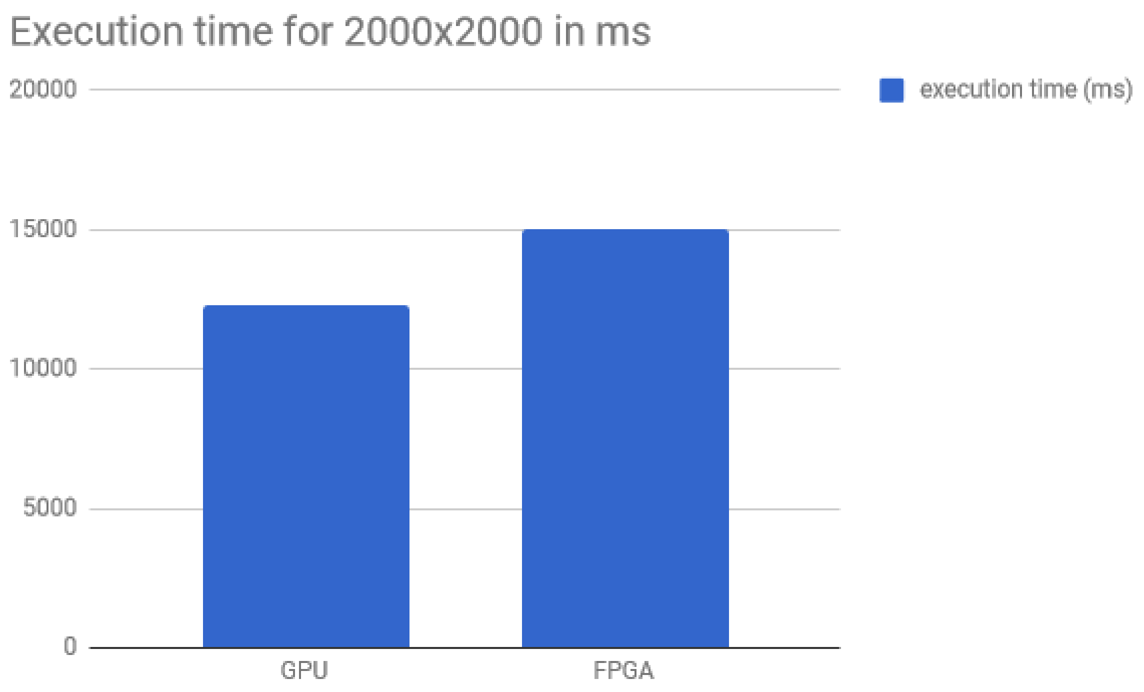


Figure 25 Comparison between GPU and FPGA for a big image

4.1 Conclusion

In this thesis we made an attempt to accelerate the execution time of the Local Laplacian Filter application. The first step was to write the C code. After that we implemented an OpenCL version, to target the GPU and the same code was also used for the multicore execution for the CPU. For this purpose the Centaurus runtime was used and helped the developing process, mostly by providing a level of abstraction for the data transfer and the kernel execution in the different devices
After achieving a satisfying execution time for the GPU, we implemented the FPGA version and we tried to surpass the GPU speedup.
The results of the profiling show that the FPGA and the GPU implementations are by far faster than the CPU .The FPGA and the GPU have similar execution times, with the FPGA being faster for small images, and the GPU faster for large images.

Execution time in ms	Multicore	GPU	FPGA
800x534	8827.707	2.754	1.637
1024x786	19.808,04367	3.673,615492	2.943
2000x2000	81.868	12.291,6601	15.031,570192

4.2.Future work

There is a lot of room for implementations for all devices but this thesis showed that the CPU (both single threaded and multithreaded) will not achieve better results than the GPU and the FPGA.
Porting the GPU version to CUDA would give access to better profiling tools, in order to improve the occupancy of the GPU.A simple way to achieve this, is to allow the GPU to process more windows in every kernel execution. Also, some of the kernels could execute asynchronous in order to overlap data transfers, with kernel executions.
For the FPGA, the first step would be an upgrade of the tools from 2014.4, to a newer version .Besides the better results in performance that could come up, the newer versions, support the synthesis of the pow() function .The use of fixed point arithmetic and user defined data types in general would increase the performance, and the utilization of the accelerator. A lookup table for the remapping function would also yield better results, as the remap kernel is one of the implementation bottlenecks. A better partitioning of the image cache to smaller caches, would allow many of the hardware kernels to execute completely in parallel. Finally two or more accelerators, could be instantiated in the block design, and process different parts of the image in parallel.
Another way to improve the execution time in every device, is to use the features that the Centaurus runtime provides for approximate computing and implement an approximate version of the filter with less levels, or smaller windows in each level.

5. Bibliography

Aubert, G., & Kornprobst, P. (2002). Mathematical problems in image processing: Partial Differential Equations and the Calculus of Variations. *Applied Mathematical Sciences, 147*.

Bhat, P., Zitnick, C. L., Cohen, M., & Curless, B. (2010). Gradientshop: A gradient-domain optimization framework for image and video filtering. *ACM Transactions on Graphics, 29*(2).

Burt, P. J., & Adelson, E. H. (n.d.). The Laplacian pyramid as a compact image code. *IEEE Transactions on Communication*, (pp. 532-540).

Farbman, Z., Fattal, R., Lischinski, D., & Szeliski, R. (2008). Edge-preserving decompositions for multi-scale tone and detail manipulation. *Proc. SIGGRAPH No 67*.

Fattal, R. (2009). Edge-avoiding wavelets and their applications. *ACM Transactions on Graphics, 8*(3).

Jacobsen, M., & Kastner, R. (2013). RIFFA 2.0: A reusable intergration Framework for FPGA accelerators. *In 23rd International Conference on Field programmable Logic and Aplications FPL '13*, (pp. 1-8).

Kass, M., & Solomon, J. (2010). Smoothed local histogram filters. *ACM Transactions in Graphics, 29*(3).

NVIDIA. *Code Samples*. Retrieved from http://docs.nvidia.com/cuda/cuda-samples/index.html#cuda-separable-convolution.

PARIS, S., HASINOFF, S. W., & KAUTZ, J. (2011). Local Laplacian Filters: Edge-aware image processing with a Laplacian pyramid. *ACM Transactions on Graphic.* Proc. SIGGRAPH.

Perona, P., & Malik, J. (1990). Scale-space and edge detection using anisotropic diffusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 12*(7).

Tomasi, C., & Maduchi, R. (1998). Bilateral fitering for gray and color images. *In Proceedings of the IEE International Conference on Computer Vision.* (Bombay,india).

Vassiliadis, V., Parasyris, K., Chalios, C., Antonopoulos, C. D., Lalis, S., Bellas, N., . . . Nikolopoulos, D. S. (2015). A Programming Model and Runtime System for Significance-aware Energy-effient Computing. *In 20th ACM Symposium on Principles and Practice of Parallel Programming , PPoPP 2015.*

Xilinx. *7 Series Overview.* Retrieved from https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf.

Xilinx. *Xilinx. Vivado Design Suite User Guide, High-Level Synthesis.* Retrieved from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_4/ug902-vivado-high-level-synthesis.pdf .