Interest Point Detection and Matching on SIMD Architectures

Torsten Sommer

Diploma thesis

Interest Point Detection and Matching on SIMD Architectures

Torsten Sommer

April 30, 2010

Lehrstuhl für Datenverarbeitung
Technische Universität München

Commenced on November 4, 2009
Submitted on April 30, 2010

by Torsten Sommer

Matr.-Nr. 2714705

Im Freihöfl 42
D-85057 Ingolstadt
GERMANY

Supervised by Prof. Dr.-Ing. K. Diepold, Prof. Dr. rer. nat. habil. T. Bräunl
and Dipl.-Ing S. Hawe

Fakultät für Elektrotechnik und Informationstechnik
Technische Universität München

Executed at Robotics and Automation Lab, Centre of Intelligent Information Processing Systems,
University of Western Australia, Perth

Abstract

With the soaring number of transistors per chip, graphic processing units have developed from coprocessors specialized in 3D graphics into fully programmable parallel processors providing superior floating point performance and memory bandwidth for data-parallel computations. This type of data processing is commonly found in computer vision algorithms that process images and thus are predestined for a parallel implementation running on a GPU which offers a significant speed-up without requiring additional or expensive hardware.

In this thesis an OpenCL implementation of the popular SURF algorithm for interest point detection and matching is presented. After a comprehensive review of the algorithm's details and underlying theory, a closer look is taken at the interface provided by the OpenCL framework and the specifics of the GPU. These sections are followed by a description of the kernel code running on the parallel processor and the supporting framework that facilitates development and testing. Finally the runtime and detection/matching performance of the proposed implementation is evaluated based on a set of standard image sequences and a reference implementation running on the CPU.

Contents

List of Symbols

CPU	Central Processing Unit
GPU	Graphics Processing Unit
GPGPU	General-Purpose Computing on Graphics Processing Units
RAM	Random Access Memory
FPGA	Field-Programmable Gate Array
PC	Personal Computer
GFLOPS	Billion (Giga) Floating Point Operations Per Second
SIMD	Single Instruction Multiple Data
Cg	C for Graphics (High-Level Shading Language)
GLSL	OpenGL Shading Language
HLSL	High Level Shading Language
CUDA	Compute Unified Device Architecture
OpenGL	Open Graphics Library
OpenCL	Open Computing Language
SURF	Speeded Up Robust Features
SIFT	Scale-Invariant Feature Transform
OpenCV	Open Source Computer Vision
XML	Extensible Markup Language
$\mathbf{A}$	Bold Uppercase Letter denotes a Matrix
$\mathbf{v}$	Bold Lowercase Letter denotes a Vector
$Tr(\mathbf{A})$	$\equiv \sum_{i=1}^{n} a_{ii}$ (Trace of a $n \times n$ Matrix)
$sign(x)$	Sign of the Scalar x

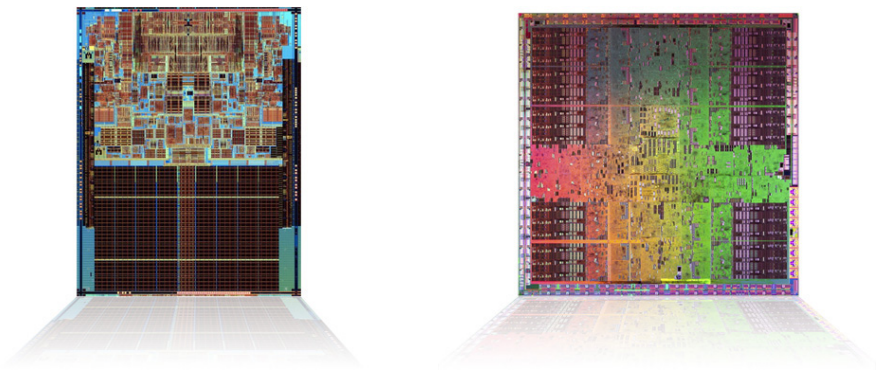
Chapter 1

Introduction

This chapter shows the motivation for the thesis' topic and after presenting its objectives a brief outline is given in Section 1.3.

1.1 Motivation

Over the last decade graphics processors have developed from simple coprocessors providing a limited and very specific instruction set focusing on the acceleration of 3D graphics into general purpose processors that can be programmed using high-level programming languages. The increase in computational power was for a long time driven by increasing the clock frequency of the processors which causes higher power consumption and is more and more approaching the physical limits. However it is still possible to increase the performance by switching to even smaller structure sizes which allows for more transistors on the same chip.



(a) Intel Core 2 Duo (b) NVIDIA GT200

Figure 1.1: Dies of a modern CPU (left) and GPU (right) [Har07, Tru08]

While this possibility has been used on the CPU side to increase the sizes of first

and second level caches the GPUs were given more cores instead of more on chip memory which results in a higher number of transistors devoted to calculations instead of data storage. By looking at the processor dies of a state of the art CPU and GPU this becomes obvious[1]. Almost half the available surface area is consumed by memory whilst the caches on a modern graphics processor take up only roughly one eighth. This has a direct effect on the number of operations that can be performed per second and the GFLOPS/Watt ratio which is also becoming increasingly important especially for mobile, battery powered applications.

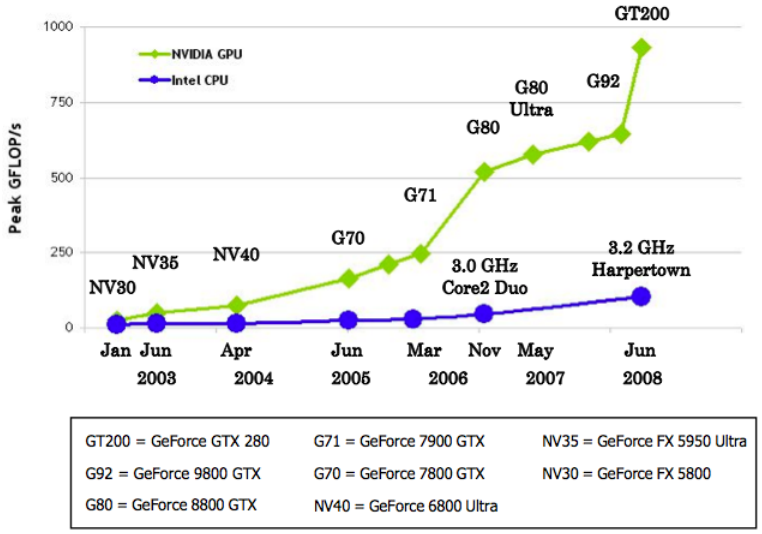


Figure 1.2: GFLOPS: CPU vs. GPU from [Cor09b]

First introduced by NVIDIA in 1999 it took another four years until the vast performance of the GPU was exploited for non-graphics applications like the fast Fourier transform [BS08]. After a number of proprietary shader languages such as Cg, GLSL, HLSL and CUDA just to name a few the Khronos group released version 1.0 of the OpenCL standard [Mun08] in 2009 that allows programmers to harness the power of a variety of parallel processors without having to use vendor specific APIs and libraries. The first implementation was provided by Apple with the release of Mac OS 10.6 "Snow Leopard" [Inc08].

Computer vision applications on the other hand have always demanded higher performance to allow for faster processing. One example is real-time processing in automotive and mobile robotics applications such as object recognition and tracking where more operations per second mean higher possible resolution, lower response times and more accurate results. Also, for problems that can be processed offline

[1]The dark rectangular regions are on-chip memory in figure 1.1

such as image stitching or 3D reconstruction where an exact homography estimation is required, this approach allows for shorter runtime and more images to be taken into account that can be processed at higher resolutions using more accurate and therefore computationally expensive filters and descriptors.

(a) 50 percent of the images registered

(b) Blended panoramic image

Figure 1.3: Panoramic image stitching using homography estimation. Images taken from [BL07]

Another advantage of the GPU approach is that computations run entirely on the graphics card and thus leave the CPU available to other tasks that cannot take advantage of a data-parallel implementation and are more efficiently carried out there. Also there is no need for additional hardware or FPGAs as a GPU ships with almost any modern PC.

Considering the soaring performance of the GPUs on the one hand, and the large amount parallel data processing found in today's computer vision algorithms on the other, graphic cards seem predestined to execute these computationally intense operations. In addition to this it has to be evaluated how the limitations brought by the new API and memory model affect the runtime and accuracy of the implement and how much additional effort is required to map a given algorithm to the execution model used by the GPU.

1.2 Objectives

The goal of this thesis is to develop a platform independent framework that allows for fast and reliable detection and matching of interest points taking advantage of the vast computational power found in today's GPUs without relying on a specific

manufacturer or product. In order to parallelize the computation and run it on graphics processors it is necessary to carefully analyze the algorithm and subdivide it into a number of steps that can be efficiently carried out on the GPU.

To be able to quantize the reduction of the runtime and to evaluate the performance in the feature detection and matching process the results need to be compared using a reference implementation and a commonly used test dataset that is based on ground truth. This facilitates the decision whether or not using this approach will have the desired effect in a particular application.

1.3 Outline

The thesis is subdivided into four chapters. Chapter 3 provides a comprehensive review of the SURF algorithm [BTG06] and shows the fundamental ideas that are used to extract, classify, describe and eventually match interest points from different images. In the following (Chapter 4) a closer look is taken at the OpenCL framework and the programming language that will be used for the implementation of the algorithm. A detailed description of the implementation is presented in Chapter 5 followed by an overview of the framework that supports the development and testing of the kernel programs in Chapter 6. Finally the runtime performance of the proposed implementation is evaluated and compared to a reference implementation in Chapter 7.

Chapter 2

Related Work

The following sections give a brief review of three papers that are concerned with feature detection and GPGPU computations and point out the achievements and flaws of the respective methods and implementations.

2.1 MinGPU

Released in 2007, MinGPU [BS08], short for "minimal GPU C++ library", is a C++ library providing a variety of functions for processing images in parallel on graphics hardware. By providing an abstract interface to the graphics processing unit it aims at facilitating the conversion of existing CPU code to code that can be run on the graphics board. It already provides several computer vision algorithms, including the homography transformation between two 3D views. In their research paper the authors provide timing charts showing that their MinGPU implementation of homography transformations runs approximately 600 times faster than the C++ CPU implementation.

Since the library is built on top of OpenGL it runs on virtually all modern graphics cards installed in today's computers. By encapsulating the complicated GPU programming using OpenGL into an easy-to-use interface it makes the the powerful resources provided by the GPU available even to less experienced programmers that are working on computer vision problems. Due to the encapsulation paradigm, users do not need any knowledge of its inner structure; therefore they do not need any knowledge of the details of how the fragment processor or the OpenGL drivers operate. The library contains only two classes: Array and Program. The Array Class defines a 2D array in the GPU memory, while the Program class represents a Cg program in the GPU memory. The data processing is implemented in a straightforward way: both array and Cg programs are created, the array holding the data to be processed is moved to a GPU, the program parameters are set and eventually the program is executed.

Even though the library performs well at filter-like algorithms, which process every pixel independently of the others such as Gaussian smoothing, convolutions, image pyramids, geometric transformations, image de-noising and cross-correlation there is no possibility of interaction between the program instances processing every output element. The programmer is also very limited in terms of data types since MinGPU is based on OpenGL and its shader language Cg which is only intended for graphics processing.

There exist two major problems with this approach. The first one is the limitation to "image like" data structures that can be processed by the Cg programs and the second being the fact that only operations implemented by MinGPU can be carried out on the graphics board and thus adding functionality would require modifications of the library.

2.2 OpenSURF

OpenSURF [Eva09] is as the name suggests an open source implementation of the SURF algorithm by Christopher Evans and was released early 2009. It also includes an extensive documentation and commented source which facilitates the understanding of the complicated code. The library itself is written in C++ and depends only on the OpenCV library for compilation which makes it virtually platform independent since there are compilers and versions of OpenCV for almost any operating system used in image processing amongst others Linux, Window and MacOS X. The author has taken care not to use any non-standard extensions to keep OpenSURF portable between Windows and Linux but the library also compiles on MacOS X without any major changes to the source code.

The library is organized in five different modules which are briefly described in the following. The first one is the integral image. It takes the input image and creates the integral image representation of the supplied input and computes the sum over the pixel values of upright rectangular areas within the image.

The main processing step of the SURF algorithm is performed by the Fast-Hessian module. It takes the integral image as an input and builds the Hessian response map. Next the interest points are extracted using a non-maximum suppression and finally interpolated. The result is a C++ std::vector of accurately localized interest points.

To describe and match the detected interest points the descriptor module extracts the Haar wavelet based structure vectors using the previously computed integral

image and returns a vector of "SURF described" interest points. The matching procedure is not part of the library but can easily be implemented as done in the demonstration code that ships with OpenSURF.

All detected interest points are stored in a class provided by the interest point module. It provides accessor and mutator methods for the associated data. Last but not least the utility module contains auxiliary functions that are not SURF specific.

Due to its portability and efficient implementation in C++ OpenSURF will serve as a benchmark for both processing speed and quality of the obtained results.

2.3 Efficient Keypoint Matching

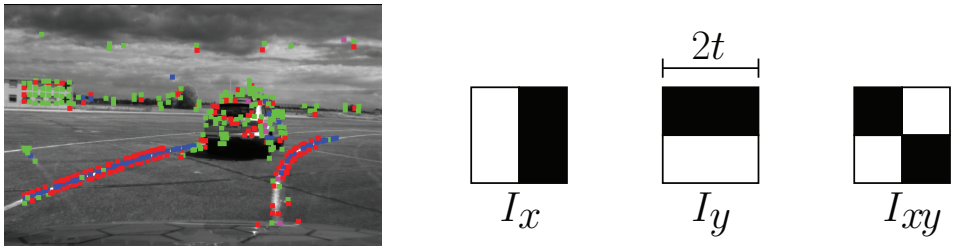
In "Efficient Keypoint Matching for Robot Vision using GPUs" [SW09] Michael Schweitzer and Hans-Joachim Wuensche from the Institute for Autonomous Systems Technology of the University of the Armed Forces Munich present a new approach for interest point detection and matching that focuses on robot vision applications where a very low execution time of the extraction process is required that allows for frame rates up to 200 Hz.

To reduce the computational effort a corner detection was chosen in contrast to the approaches taken in SURF [BTG06] and SIFT [Low99] that provide greater robustness to changes in viewpoint [MS05] but require localization in scale space which adds to the amount of operations needed. This loss in robustness can to some extent be compensated by using additional sensor information from the robot i.e. speed and gyration rates.

An additional difference to previous approaches is the fact that the Haar wavelet responses are not only used to detect the interest points (here called "keypoints") but also for their classification and description. Like in SURF an integral image is used to efficiently compute the Haar wavelet responses.

The interest points are detected at local extrema of the response map of I_{xy} similar to SURF by applying a non-maximum suppression. The I_{xy}, I_x and I_y components of the "Scale Invariant Descriptive Cells" (short SidCell) together with I_{xy} calculated at half the scale t are then used for the classification into one of the following groups: corner, chessboard, edge and flat. Figure 2.1 shows the three different Haar wavelets used where t denotes the scale and a scene after the features have been classified.

The descriptor is computed as a vector of surrounding SidCells weighted according



(a) (b)

Figure 2.1: Strong/weak corners of an automotive scene (left) and Haar wavelets for SidCell components (right) [SW09]

to their distance from the corresponding interest point. These descriptors are then used to match the detected interest points against a list of candidates detected in the previous frame.

The whole process is implemented in CUDA and runs entirely on the GPU and thus makes the CPU available to other tasks. However this CUDA implementation requires the program to run on NVIDIA hardware and thus isn't platform independent in terms of supported hardware. Further, the presented approach is not robust to even smaller changes in viewpoint and relies on additional information from the vehicle to constrain the matching which renders the algorithm useless in environments where the camera configuration is not known or the distance between two viewpoints and the corresponding images is too large. Finally it is worth mentioning that the enormous frame rate of over 200 Hz was achieved using a NVIDIA C1060 Tesla device - four times the price of a standard laptop.

Chapter 3

Interest Point Detection, Description and Matching

The following sections review the methods used in [BTG06] and provide the necessary mathematical background to understand the following chapters. It is subdivided into two major parts: The interest point detection that includes the computation of the integral images and determinant of the Hessians, finding the extrema and interpolation of their position. The second part is concerned with the description and matching of the detected features.

3.1 Integral Images

Integral images were originally introduced by Crow et al. [Cro84] as "summed-area tables" for the efficient calculation of mipmaps (i.e. stacks of scaled versions of a texture that can be used to reduce aliasing artifacts and increase the rendering speed in 3D applications) but didn't attract a lot of attention until the introduction of the object detection framework [VJ01] proposed by Viola-Jones in 2001.

The integral image can be computed quickly from an input image and reduces the time required to retrieve the sum over any rectangular upright region within the image. With an input image I the integral image $I_\Sigma(x)$ at a location $x = (x, y)^T$ is the sum over all values within the rectangle spanned by x and $(0,\ 0)^T$. The integral image can therefore be defined as

$$I_\Sigma(x, y) = \sum_{i=0}^{x} \sum_{j=0}^{y} I(i, j) \tag{3.1}$$

For computational efficiency this summation can be separated into two sums for the x- and y-direction:

$$I_{\Sigma,x}(x,y) = \sum_{i=0}^{x} I(i,y) \text{ and } I_{\Sigma,y}(x,y) = \sum_{i=0}^{y} I(x,i) \tag{3.2}$$

The integral can then be written in terms of $I_{\Sigma,x}$ and $I_{\Sigma,y}$ as

$$I_{\Sigma}(x,y) = \sum_{i=0}^{x} I_{\Sigma,y}(i,y) = \sum_{i=0}^{y} I_{\Sigma,x}(x,i). \tag{3.3}$$

which reduces the computational complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. Using the integral image, the effort of calculating the sum over any upright, rectangular area Σ within the image can be reduced to three additions:

$$\Sigma = A + C - B - D. \tag{3.4}$$

Therefore the calculation is independent of the size of the area which is beneficial when computing large integrals corresponding to large filter sizes.

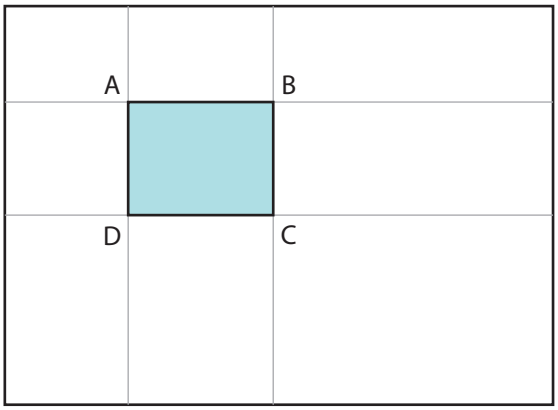


Figure 3.1: Only the values at the four corners that are necessary to calculate the sum over the green rectangle.

3.2 Hessian Matrix Based Interest Points

The method to detect interest points that is used in [BTG06] is based on the Hessian (sometimes also referred to as Jacobian matrix) that contains the derivatives of an n-dimensional function with respect to all its dimensions and thus has a size of $n \times n$. It detects blob-like structures within an image. The Hessian of a 2-dimensional function $f(x,y)$ is given as

$$\mathcal{H}(f(x,y)) = \begin{bmatrix} \frac{\delta^2 f}{\delta x^2} & \frac{\delta^2 f}{\delta x \delta y} \\ \frac{\delta^2 f}{\delta x \delta y} & \frac{\delta^2 f}{\delta y^2} \end{bmatrix}. \tag{3.5}$$

The interest points are detected at the locations where the determinant of the Hessian for the 2-dimensional image function $f(x, y)$

$$det(\mathcal{H}) = \frac{\delta^2 f}{\delta x^2}\frac{\delta^2 f}{\delta y^2} - \left(\frac{\delta^2 f}{\delta x \delta y}\right)^2. \tag{3.6}$$

is maximal. The sign of the trace of $\mathcal{H}$

$$sign(Tr(\mathcal{H})) \tag{3.7}$$

can be used to distinguish the detected interest points into two groups. Light blobs on dark ground are indicated by a negative and dark blobs on light ground by a positive sign as depicted in figure 3.2. The second order partial derivates in 3.9 can be computed by convolution of the image function $f(x, y)$ with scale normalized second order derivates of Gaussians [BTG06].

Figure 3.2: The two types of contrast: Light blob on dark ground and vice versa.

Thus the Hessian can then be expressed in terms of the second order derivatives of the Gaussian $g(x, \sigma)$ at scale σ ($\frac{\delta^2 g(\sigma)}{\delta x^2}$, $\frac{\delta^2 g(\sigma)}{\delta y^2}$ and $\frac{\delta^2 g}{\delta x \delta y}$) as

$$\mathcal{H}(\mathbf{x}, \sigma) = \begin{bmatrix} L_{xx}(\mathbf{x}, \sigma) & L_{xy}(\mathbf{x}, \sigma) \\ L_{yx}(\mathbf{x}, \sigma) & L_{yy}(\mathbf{x}, \sigma) \end{bmatrix}. \tag{3.8}$$

where $L_{xx}(\mathbf{x}, \sigma)$ denotes the convolution with $\frac{\delta^2 g(\sigma)}{\delta x^2}$ and similarly for $L_{yy}(\mathbf{x}, \sigma)$ and $L_{yy}(\mathbf{x}, \sigma)$.

In order to apply the convolution to the finite and discrete image I the infinite Gaussians have to be discretized and cropped. Even though this leads to a loss in repeatability under image rotations around odd multiples of $\pi/4$ which is caused by the square shape of the filters it greatly speeds up the computations of the response

map [BTG06].

These filters are even further simplified by approximating the Gaussian derivates by box filters as shown in 3.3 (bottom row) where the white regions have a weight of 1 and the black regions a weight of -2 for the xx- and yy-filters (left and center) and -1 for the xy-filter (right). Gray means a weight of 0. Using integral images as discussed in 3.1 this approximation allows for great computational efficiency when calculating the filter responses, especially for bigger filter sizes corresponding to a larger scale σ.

Let $L_{xx}(\mathbf{x}, \sigma)$ be the filter response of the approximated box filter for the xx-direction and similarly for $L_{yy}(\mathbf{x}, \sigma)$ and $L_{xy}(\mathbf{x}, \sigma)$ the Hessian can now be written as

$$\mathcal{H}(\mathbf{x}, \sigma) = \begin{bmatrix} L_{xx}(\mathbf{x}, \sigma) & L_{xy}(\mathbf{x}, \sigma) \\ L_{yx}(\mathbf{x}, \sigma) & L_{yy}(\mathbf{x}, \sigma) \end{bmatrix}. \tag{3.9}$$

Defining the convolution of the image I with the box-filters as D_{xx}, D_{yy} and D_{xy} the determinant of the Hessian then becomes

$$\det(\mathcal{H}_{approx}) = D_{xx}D_{yy} - (wD_{xy})^2. \tag{3.10}$$

To assure energy conservation between the Gaussian filters and the corresponding approximations a relative weight

$$w = \frac{||L_{xy}(1.2)||_F \, ||D_{yy}(9)||_F}{||L_{yy}(1.2)||_F \, ||D_{xy}(9)||_F} = 0.912... \simeq 0.9 \tag{3.11}$$

is applied to the expression where $||A||_F$ is the Frobenius norm of a $m \times n$ matrix A [GVL96]:

$$||A||_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2}. \tag{3.12}$$

Even though this weight depends on the scale it can in practice be kept constant as proposed by [BTG06] since it doesn't have a significant impact on the results. In order to guarantee a constant Frobenius norm for all filter sizes the filters are additionally normalized with respect to their size which is important when comparing filter responses of different scales.

Using the above definitions the function used for the detection can finally be written as

$$D = sign(D_{xx} + D_{yy}) \, (D_{xx} \, D_{yy} - 0.81 \, D_{xy}^2) \tag{3.13}$$

Figure 3.3 shows the three steps of the approximation of the filters used for the convolution. The first three filters are the scale-normalized second order derivatives of

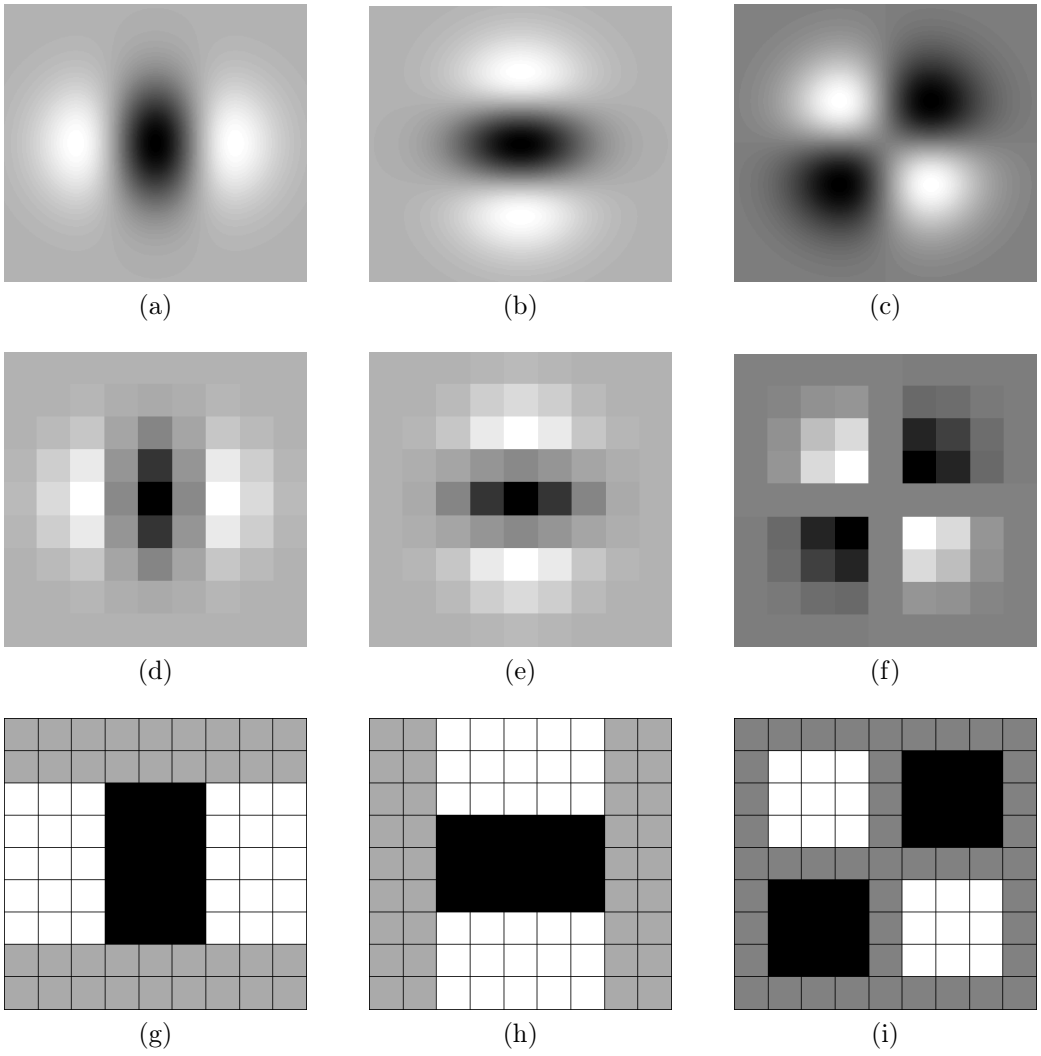


Figure 3.3: First row: second order Gaussian derivatives for $\sigma = 1.2$. Second row: discretized and cropped Gaussians. Third row: corresponding box filter approximations with size 9×9.

the Gaussians for the xx-, yy- and xy-directions corresponding to a scale of $\sigma = 1.2$. The second row shows the discretized and cropped version with filter size 9×9. The bottom row depicts the box-filter approximations of the previous row. The two distinct shades of gray (symbolizing a weight of 0) are due to a different offset caused by a higher weight of the black regions in the xx- and yy- compared to the xy-filters.

3.3 Scale Space Representation

An inherent problem when using the determinant of the Hessian is that only blobs corresponding to the filter size are detected but on the other hand the detected interest points tend to be more stable across changes in viewpoint or lighting [MS04]. The scale-space is a concept introduced by Witken in [Wit83] that addresses this problem. In order to construct the scale-space an image is iteratively filtered with Gaussian filters where the size of the filter depends on the corresponding scale of the filter. The filtered images are stacked on to of each other and form the scale domain and subsampling of the higher levels yields the image pyramid. This approach has been successfully applied in [Low04] to find the extrema of blob responses across the different scales.

However because of the necessity of repeated filtering (to avoid aliasing effects) this approach is computationally expensive. Using the box filters the blob responses for all levels of the pyramid can be computed in parallel using the integral image. It also saves the iterative smoothing and subsampling which would otherwise be necessary. Figure 3.4 shows the two methods to calculate the scale-space pyramid from the original image.

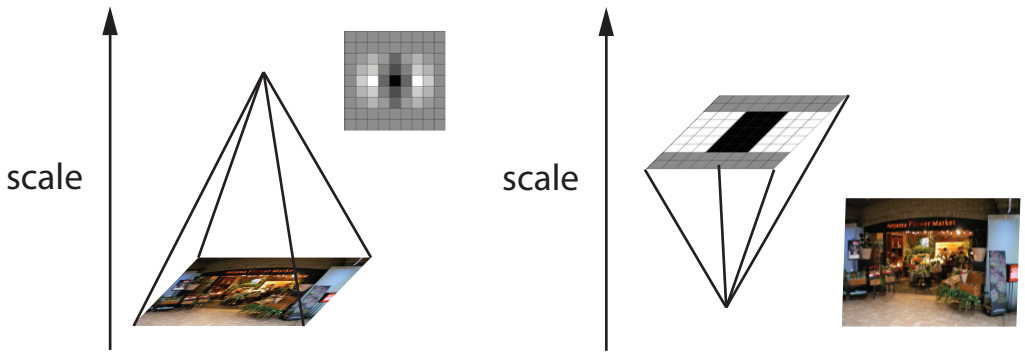


Figure 3.4: Computation of the scale-space pyramid through smoothing and subsampling (left) and varying the filter size (right)

By using box filters together with integral images as described in 3.1 a significant increase in performance can be achieved. Calculating the response for a 9×9 filter causes 81 memory accesses and additions compared to 8 for the box filter. Thus the computational cost remains constant over the different scales which allows for efficient and parallel computation of the layers. Although the use of box filters preserves high-frequency components that would get lost in down-scaled versions of an

image experiments in [BTG06] have shown that this effect is negligible.

The scale space starts with the response of the smallest filter of size 9×9 corresponding to a Gaussian with $\sigma = 1.2$. It is subdivided into octaves each containing four successive filter sizes that are used to retrieve the blob-responses for the different scales. The value of the approximated Gaussian scale increases with the filter size according to the following formula:

$$\sigma_{approx} = \text{Filter Size} \cdot \frac{1.2}{9} \tag{3.14}$$

The scale space starts with the response of the smallest filter of size 9×9 followed by filters with sizes 15×15, 21×21, 27×27. The higher octaves are constructed similarly by doubling the increase in filter size when going to the next higher octave starting with a value of 6. At the same time the resolution of the image can be reduced by a factor of two for every new octave which reduces the computational effort to a quarter compared to the preceding level. Figure 3.5 shows the first three and (optional fourth) octaves with filter sizes 9, 15, 21, 27 for the first, 15, 27, 39, 51 for the second and 27, 51, 75, 99 for the third octave.

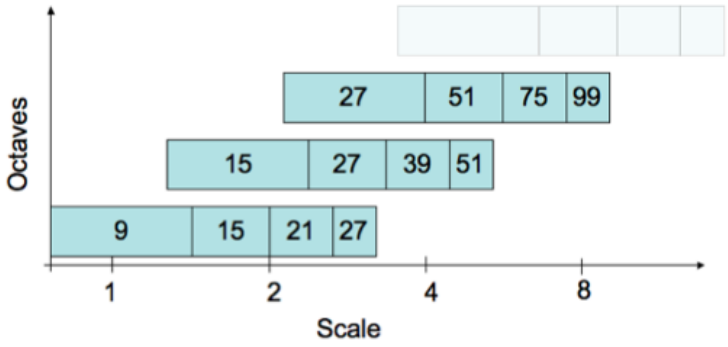


Figure 3.5: Overlapping filter sizes for the first three octaves after [BTG06]

The overlap between successive octaves is necessary since the responses from the first and last filter in every octave are only used for comparison when detecting the extrema (as shown in Section 3.4).

There are however some constraints that have to be taken into account when increasing the size of the filters. As a central pixel is required the lobe size (i.e. the width of the white and black regions of the filters D_{xx} and D_{yy}) can only be increased by multiples of 2 resulting in a minimum increase of 6 for the whole filter.

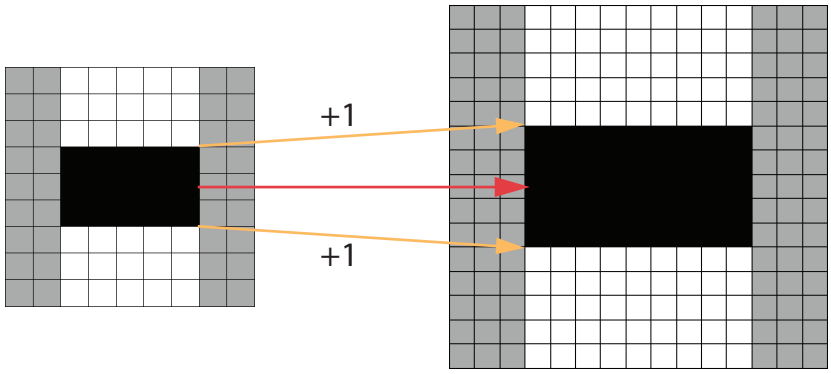


Figure 3.6: Increasing filters from 9×9 to 15×15 whilst preserving a central pixel

3.4 Interest Point Localisation

To locate the interest points within the previously constructed scale-space a 3D non-maximum suppression is applied to the four layers of every octave. Additionally a threshold can be applied to control the number of detected interest points. The lower the threshold the higher number of detected features. However a higher threshold leaves only the most stable results in place which generally allows for more reliable matching of the detected interest points.

To find the local maxima within the xy- and scale-space the non-maximum suppression is applied in all three directions. Therefore each element is compared to its $3 \cdot 3 \cdot 2$ neighbors in the next higher and lower scale and the 8 neighbors in the same scale. This process is visualized in figure 5.10 where the orange element in the center is compared to its 26 neighbors (blue).

To get a more accurate position for the detected interest points it is necessary to interpolate the position in both xy- and scale space because the scale change between the first filters in every octave is fairly big. Another problem is that the second and third octave have sampling intervals of 4 and 8 which means the accuracy of the detected position is only 4 and 8 pixels respectively.

To overcome this problem the position of the detected interest points is interpolated in both scale- and xy-space using the method proposed in [BL02]. Therefore the $H(x, y, \sigma)$ is written as a Taylor series up to second order with its center at the position where the point was detected:

$$H = H + \frac{\delta H}{\delta \mathbf{x}}^T \mathbf{x} + \frac{1}{2}\mathbf{x}^\mathbf{T}\frac{\delta^2 H}{\delta \mathbf{x^2}}\mathbf{x} \tag{3.15}$$

In order to find the distance vector between the interpolated location and the de-

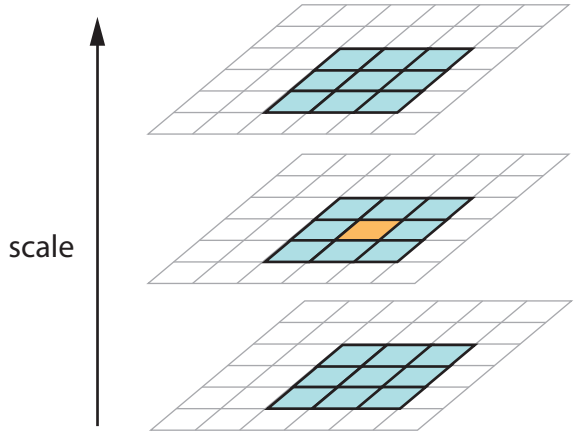


Figure 3.7: 3D non-maximum suppression

tected interest point $\hat{\mathbf{x}} = (x, y, \sigma)$ the derivative of this function is set to zero:

$$\hat{x} = -\frac{\delta^2 H}{\delta \mathbf{x^2}}^{-1} \frac{\delta H}{\delta \mathbf{x}} \tag{3.16}$$

The first order derivatives of H are approximated by the finite differences of adjacent pixels as

$$D_x(x, y, \sigma) = (H(x + 1, y, \sigma) - H(x - 1, y, \sigma))/2 \tag{3.17}$$

and for the second order derivative as

$$D_{xx}(x, y, \sigma) = H(x - 1, y, \sigma) + H(x + 1, y, \sigma) - 2H(x, y, \sigma) \tag{3.18}$$

$$\begin{aligned} D_{xy}(x, y, \sigma) \quad = \quad & (H(x + 1, y + 1, \sigma) - H(x - 1, y + 1, \sigma) \\ & -H(x + 1, y - 1, \sigma) + H(x - 1, y - 1, \sigma))/4 \end{aligned} \tag{3.19}$$

and similarly for D_y and D_σ as well as D_{yy}, $D_{\sigma\sigma}$, $D_{x\sigma}$ and $D_{y\sigma}$. To avoid instabilities caused by the interpolation that can only be carried out with limited accuracy the elements of $\hat{x}$ are being limited to ± 0.5.

To motivate this approximation method consider the one dimensional quadratic function $f(x)$ and its first and second order derivatives

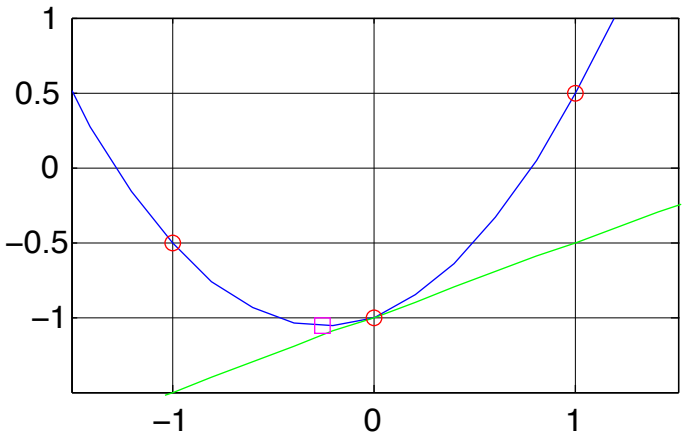


Figure 3.8: Functions $f(x)$ (blue), $f[n]$ (red), the tangent at $y = 0$ (green) and the extremum (magenta)

$$\begin{aligned} f(x) &= x^2 + \frac{1}{2}x - 1 \\ f'(x) &= x + \frac{1}{2} \\ f''(x) &= 1 \end{aligned} \tag{3.20}$$

and the discretized version $f[n] = f(n)$ with $n \in \mathbb{N}_0$ shown in figure 3.8. Applying the approximation of the derivates to $f[0]$ and substituting them in formula 3.16 yields the position of the extremum of $f(x)$:

$$\begin{aligned} f'[0] = &\quad (\tfrac{1}{2} + \tfrac{1}{2})/2 &&= \frac{1}{2} = f'(0) \\ f''[0] = &\quad -\tfrac{1}{2} + \tfrac{1}{2} + 2 &&= 2 = f''(0) \\ \hat{x} = &\quad -f''[0]^{-1}\, f''[0] &&= -\frac{1}{4} \end{aligned} \tag{3.21}$$

The inverse of $\frac{\delta^2 H}{\delta \mathbf{x}^2}$ can then be computed using the following formula [Mat10] where $|\mathbf{A}|$ denotes the determinant of $\mathbf{A}$:

$$\mathbf{A}^{-1} = \frac{1}{|A|} \begin{bmatrix} \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} & \begin{vmatrix} a_{13} & a_{12} \\ a_{33} & a_{32} \end{vmatrix} & \begin{vmatrix} a_{12} & a_{13} \\ a_{22} & a_{23} \end{vmatrix} \\ \begin{vmatrix} a_{23} & a_{21} \\ a_{33} & a_{31} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{13} \\ a_{31} & a_{33} \end{vmatrix} & \begin{vmatrix} a_{13} & a_{11} \\ a_{23} & a_{21} \end{vmatrix} \\ \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix} & \begin{vmatrix} a_{12} & a_{11} \\ a_{32} & a_{31} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} \end{bmatrix}, \text{with } \mathbf{A} \equiv \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}. \quad (3.22)$$

3.5 Interest Point Description and Matching

The descriptor used by the SURF algorithm describes the distribution of the pixel intensities in a scale dependent area around the detected interest points. It is similar to the descriptor proposed by Lowe [Low99]. But through the integration of gradient information within the subregions as explained in the following it is more robust towards noise. The use of integral images and Haar wavelets [Haa10] which are simple filters that are used to find gradients in the x- and y-directions allows for a significant speed-up. Figure 3.9 shows the two Haar wavelets used to calculate the gradients in x- and y-direction.

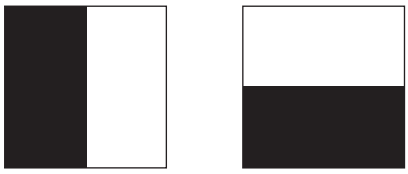


Figure 3.9: Haar Wavelets for the x- (left) and y-direction (right)

In order to achieve scale invariant results the descriptor needs to be dependent on the scale at which the interest point was detected. Therefore a window of size 20σ is defined. This window is divided into 4×4 equally sized subregions. For each of the 16 subregions the Haar wavelet responses in x- and y-direction are calculated at 5×5 regularly spaced points inside the subregion. These are summed up according to equation 3.23 and the results are stored in the four-dimensional vector v_{sub}.

$$v_{sub} = \left[\sum dx \ \sum dy \ \sum |dx| \ \sum |dy| \ \right] \quad (3.23)$$

The vectors of all subregions are then concatenated which results in a vector of length $4 \times 4 \times 4 = 64$. This descriptor is invariant to changes in scale, offsets in

illumination and after turning the descriptor into a unit vector invariant to changes in contrast [BTG06]. Figure 3.10 visualizes the extraction process of the SURF descriptor.

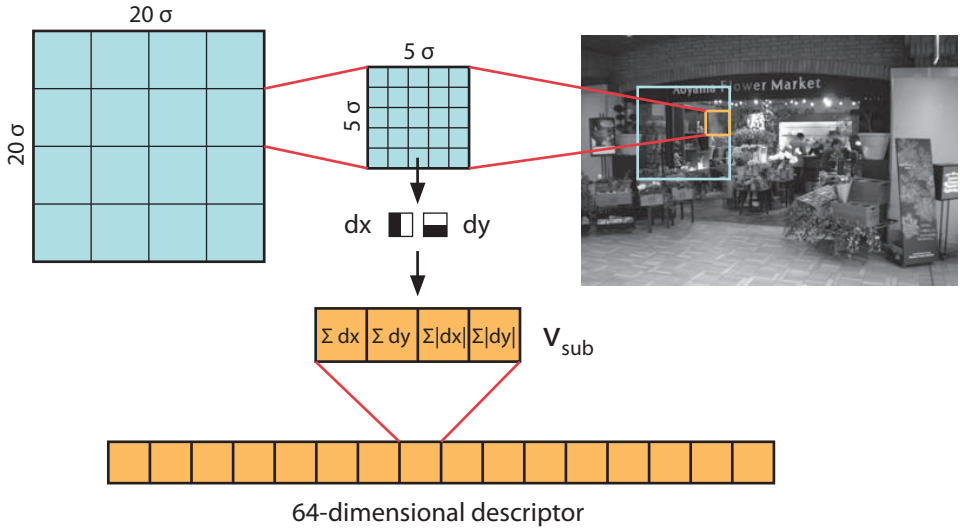


Figure 3.10: Computation of the 64-dimensional SURF descriptor.

To speed up the matching process the features can be categorized into two groups: light blobs on dark ground and vice versa (see figure 3.2). As the Hessian based detector typically detects blob-like structures the determinant can be used to distinguish the two kinds of interest points as described in 3.2. This fact can be further exploited for more advanced indexing methods.

Chapter 4

OpenCL

OpenCL (Open Computing Language) is a open programming language that allows the programmer to write parallel programs that can run on CPUs, GPUs, DSPs, Cell processors and other devices without the need to map the underlying algorithms to 3D graphics APIs like DirectX and OpenGL. The OpenCL API provides a platform independent and yet efficient, close-to-hardware interface to these devices which allows the same code to run on embedded devices as well as high performance computing hardware. Version 1.0 of the OpenCL Specification was released in 2009 [Mun08]. Another important aspect are the language bindings to OpenGL that allow programs that are using OpenCL to combine their parallel algorithms with graphics applications that are executed on the graphics pipeline.

The OpenCL framework consists of four major components that are described in detail in the following sections: The platform model, memory model, execution model and programming model. These sections are followed by a review of the runtime components and programming language that are necessary to understand the ideas and principles used in the implementation. The following sections presents the four fundamental models that form the basis of the OpenCL framework and provide the necessary abstraction to write platform independent code that runs on all supported devices. Since the OpenCL code eventually has to run on the graphics board a closer look is taken at the constraints introduced by the GPU's architecture and how the models introduced by OpenCL map to its design.

4.1 Platform Model

The platform model consist of a host with a number of compute-devices attached to it as depicted in figure 4.1. Each compute-device is divided into a number of compute units which again contain an arbitrary number of processing elements which is where the actual computation is performed.

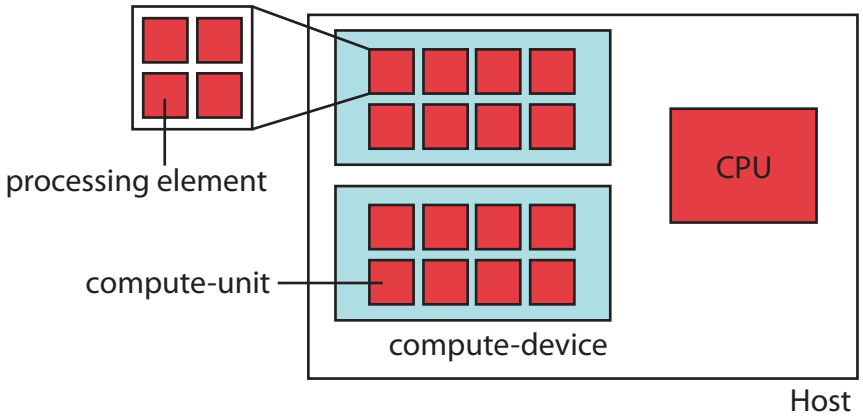


Figure 4.1: OpenCL platform model: A host with connected compute-devices each containing compute units that are comprised of a number of processing elements.

To execute calculations on the processing elements of a compute device the OpenCL application enqueues the commands from the host on which it runs. The processing elements then process a single stream of instructions as either SIMD (Single Instruction Multiple Data) where all processing elements share one stream of instructions or SPMD (Single Program Multiple Data) [Mun08].

4.2 Execution Model

An OpenCL program can generally be divided into a part that runs on the host and the kernels that execute on the compute-devices connected to it.

An index space is used when the host application submits a kernel to the compute device for execution. The index-space can be either one, two or three dimensional. Every instance of that kernel the so-called work-item is then executed for each element within the index space. Each work-item uses the same code. However the specific data used by every work-item will be different depending on its particular position within the index space.

The global index-space is subdivided into work-groups. Every work-group has the same dimensionality as the index space and each work-group has a unique ID as well as every work-item having a local ID inside the work-group. All work items in a work-group execute concurrently on one compute unit. Figure 4.2 shows an example of a two dimensional index-space with 12×12 work-items subdivided into 9 work-groups of 4×4 work-items.

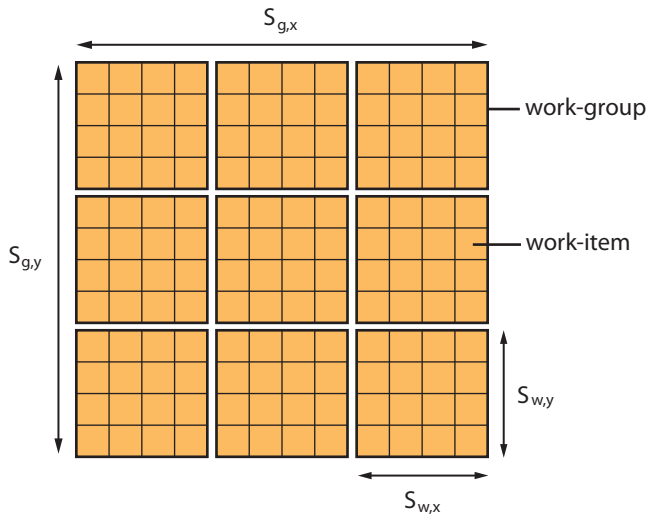


Figure 4.2: Two dimensional index space

The extent of the global index-space in every dimension denoted by $S_{g,x}$ and $S_{g,y}$ must be a multiple of the work-group sizes $S_{w,x}$ and $S_{w,y}$. Given the id of the work group (x_w, y_w), the size of the work-groups $(S_{w,x}, S_{w,y})$ and the local id of a work item $(x_{i,l}, y_{i,l})$ its global id $(x_{i,g}, y_{i,g})$ and the number of work-groups $(N_{g,x}, N_{g,y})$ can be calculated as follows:

$$(x_{i,g}, y_{i,g}) = (x_w S_{w,x} + x_{i,l},\ y_w S_{w,y} + y_{i,l}) \tag{4.1}$$

$$(N_{g,x}, N_{g,y}) = (\frac{S_{g,x}}{S_{w,x}},\ \frac{S_{g,x}}{S_{w,x}}) \tag{4.2}$$

On a CUDA device all work-groups are distributed over the available compute-units such that all work-items of a work group execute concurrently on the same compute-unit. Thus the execution time scales with the number of compute-units on the device. Figure 4.3 visualizes the process for two compute-devices with 2 and 4 compute-units [Cor09b].

The work-items are grouped into so-called warps of 32 elements on CUDA devices that are executed in half-warps of 16 work-items. This further subdivision is not part of the OpenCL execution model but plays an important role for the bandwidth efficiency as will be seen in the following section.

When a work-group of work-items is executed on a CUDA device the compute-device tries to hide the latencies caused by memory transactions by starting a number of warps simultaneously. While one warp is waiting for the read/write operations to finish another warp can perform calculations. The ratio between the maximum number of concurrent warps and started warps is referred to as the

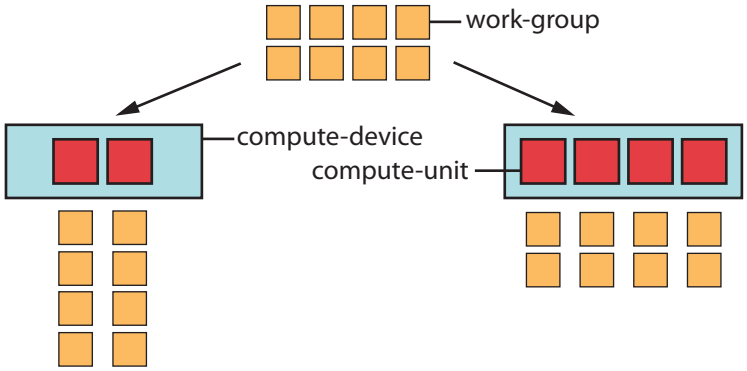


Figure 4.3: Distribution of 8 work-groups on compute-devices with 2 and 4 compute-units

$$\text{occupancy} = \frac{\text{active warps}}{\text{maximum number of warps}}. \tag{4.3}$$

This metric depends mainly on the amount of private memory (see next section) used by each work-item that maps to the registers of a compute unit. As the number of available registers is limited the occupancy decreases with the number of used registers. In order to completely hide memory latency the occupancy should be at least 25 percent [Cor09a].

The runtime of a given kernel is also affected by the sizes of the index-space and work-groups. There should be at least two work-groups per compute-unit to allow the processor to hide memory latencies efficiently. The maximum work-group size is limited by the number of registers on the compute-unit and the usage of private memory by each kernel. The number of registers for one block R_{block} is given as

$$R_{block} = \lceil \frac{R\lceil \frac{T}{32} 32 \rceil}{\frac{R_{max}}{32}} \rceil \frac{R_{max}}{32} \tag{4.4}$$

where R is the number of registers used by the kernel, R_{max} the amount of available registers on the compute-unit and T the number of work-items per work-group [Cor09a].

4.3 Memory Model

There exist four different types of memory a kernel can access: global, constant, local and private memory. Any work-item can read from and write to any element of global memory whereas the globally allocated constant memory can only be read

by the work-items. Both types are initialized by the host and may be cached depending on the implementation.

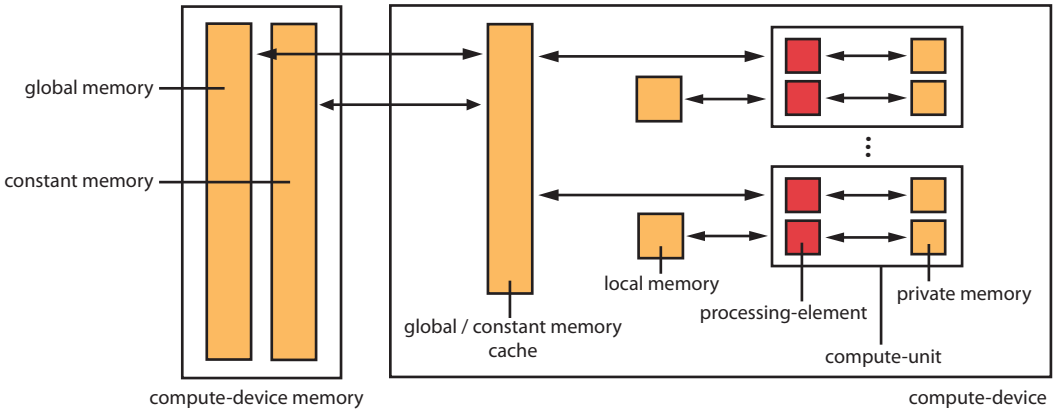


Figure 4.4: Memory model with processing-elements and compute-units

The local memory is only available to the work-items of the same group and can be used to share data among the work-items for fast access. The visibility of private memory is limited to the corresponding work-item executing the kernel. Figure 4.4 shows a schematic model of the different memory types.

On CUDA devices the different memory types in the OpenCL memory model map to the actual memory as follows: Global, local, constant and texture memory reside in the DRAM of the graphics board that corresponds to the compute-device memory whereas the shared and private memory are located directly on the GPU together with the cache for constants and textures which allows for high memory bandwidth and fast access. A look at figure 4.4 shows the motivation for the abstraction in the OpenCL model: The compute-device memory symbolizes the video RAM of the graphics card and the memory inside the compute-device the on-chip memory.

When accessing global memory from within a kernel that is executed on a CUDA device this access can be coalesced into one memory operation for every half-warp. To take advantage of the coalesced memory access a number of criteria have to be met that allow the GPU to combine the 16 memory operations into one. First every work-item within the half-warp must read a word from memory that consists of either 4, 8 or 16 bytes of aligned memory. These words must be aligned in the order of the work-items such that the first work-item in the half-warp reads the first word, the second work-item the second word and so on.

Further, the memory regions that contain the group of words accessed by each half-warp must be aligned similarly such that the first group starts at memory address

0 the second at address $0 + 16 \times word\_size$ etc.

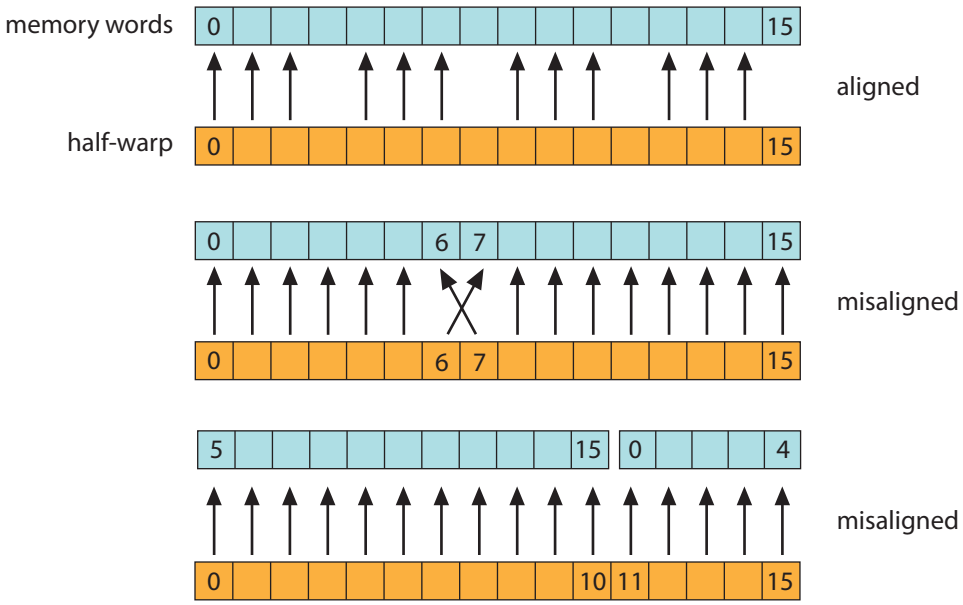


Figure 4.5: Aligned and misaligned access patterns

However not every work-item must take part in the memory access in order to allow coalescing it. If the GPU cannot coalesce the memory access because any of the above criteria is not met every word is read sequentially from global memory which reduces the effective memory bandwidth by a factor of 16.

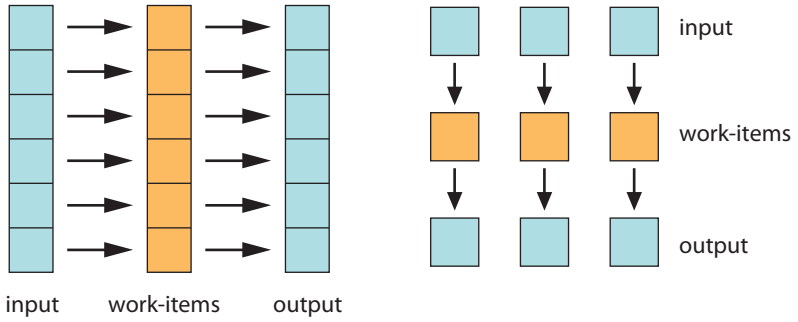
Figure 4.5 shows three different access patterns. In the first row the memory access can be coalesced into one operation even though every fourth work-item is not taking part. The second line shows a misaligned access where work-items with IDs 6 and 7 are accessing memory words with IDs 7 and 6 respectively. This access cannot be coalesced. In line three the access spreads across the boundary of two memory segments which leads to 16 consecutive operations. The latest versions of CUDA devices (compute capability 1.2 and higher [Cor09b]) have overcome most of these limitations.

4.4 Programming Model

As mentioned above there exist two programming models in OpenCL: data parallel and task parallel (as well as combinations of both types) where the preferred one is the data parallel model [Mun08].

In a data parallel model a sequence of operations is executed over a number of memory elements using a set of work-items defined by the index-space and all kernels can be executed in parallel and independently of each other. However the strict binding where every work-item maps to exactly one memory object is not necessary.

The task parallel programming model is essentially equivalent to a single work-item that executes without depending on an index-space. Any number of these work-items can then execute in parallel. Figure 4.6 visualizes the two approaches: While the work-items in the data-parallel model are processing portions of the same data every work item processes chunks of data that do not depend on each other.



(a) Data-parallel processing (b) Task-parallel processing

Figure 4.6: The two parallel programming models

Data can be synchronized at the level of work-groups and and command-queues of the same context. To synchronize the work-items within a work-group a barrier is used. Every work-item in the work-group will perform all operations up to that barrier before any work-item is allowed to proceed. This barrier is however limited to the work-group and does not provide means to synchronize work-items across work-groups. Listing 4.1 shows a barrier in the kernel code.

Listing 4.1: Barrier to synchronize the work-items within the same work-group

```
// ... before barrier

barrier(CLK_LOCAL_MEM_FENCE); // synchronize

// ... after barrier
```

To achieve synchronization among the commands in the command-queue barriers can be enqueued. All commands up to that barrier will be executed before any command beyond the barrier is allowed to execute. The barrier is limited to the command-queue where it was enqueued. Another option to synchronize the commands is to wait for events emitted by any of the enqueued commands that can be

read through the OpenCL API.

4.5 Runtime

During runtime all interaction of the host application with compute devices is handled through the functions provided by the OpenCL API. First a context has to be created. Within this context command-queues are used to communicate with the attached devices and retrieve information about the devices.

Since the compute-devices have their own memory it is necessary to allocate space inside the graphic cards memory that will hold the input and output data to the kernels. After memory has been allocated on the device using the previously created context a memory-object is returned that represents the data on the device. This memory-object is essentially a pointer inside the graphic cards memory and is stored by the host application for late use. The contents of these memory-objects or portions of it can be copied from or to the host or within the graphics memory by enqueuing the corresponding commands in the command-queue.

Similarly to the memory-objects the kernels are represented by kernel-objects that are compiled from a program-object. Using the context the program-object is compiled from a source string presented to the OpenCL compiler during runtime. The kernel-objects are then retained from the program.

With all the input loaded to the graphics cards memory and the kernels-objects retained the kernel can be executed over an index space. This process involves three steps: First all the arguments have to be set to the kernel function using the previously allocated memory-objects and other parameters like size information. After all parameters have been set the kernel is executed by calling the function clEnqueueNDRangeKernel().

Listing 4.2: Enqueuing a kernel

```
cl_int clEnqueueNDRangeKernel (
                        cl_command_queue command_queue,
                        cl_kernel kernel,
                        cl_uint work_dim,
                        const size_t *global_work_offset,
                        const size_t *global_work_size,
                        const size_t *local_work_size,
                        cl_uint num_events_in_wait_list,
                        const cl_event *event_wait_list,
                        cl_event *event )
```

This function is a typical example of a function provided by the OpenCL API. All functions are prefixed with "cl" and most require a large number of arguments. The

error code is returned as an cl_int and is the only way of finding out whether enqueuing the command succeeded or not.

Since the enqueued commands are executed asynchronously the programmer has to ensure that the execution of the previously enqueued commands has finished before the content of the memory-object holding the result is read back. There are two means to achieve this goal. The cl_event pointers returned by the API calls can be used to wait for the corresponding event or alternatively a call to clFlush() is issued that returns after the last command has been enqueued. The function clFinish() achieves the same except it returns after the execution of all enqueued commands has finished.

Another application for the event-objects is the retrieval of runtime information that includes the accurate execution times taken by the GPU. This information can be very helpful when profiling and optimizing the kernel code.

4.6 Programming Language

The OpenCL C programming language is based on the ISO/IEC 9899:1999 C language specification also known as C99 with a number of specific extensions and restrictions [Mun08]. This section gives a brief introduction to the modifications and restrictions of OpenCL C that are needed for the understanding of the code listings in Chapter 5.

In addition to the scalar vector types supported in C OpenCL introduces a number of built-in vector data types. These vector data types are defined with the type name i.e. char, int, float etc. followed by a number n which can be either 2, 4, 8 or 16. Listing 4.3 exemplifies how to access the values inside a float2 similar to a C-struct with fields x and y.

Listing 4.3: Accessing the values of a built-in vector type

```
int2 coords;
coords.x = 3;
coords.y = 7;
```

There also exist built-in types for the 2- and 3-dimensional images (image2d_t, image3d_t), samplers (sampler_t) and events (event_t) and a number of special scalar and matrix types that are not used in this implementation.

To qualify the address space of a variable or pointer OpenCL provides the following address space qualifiers that correspond to the memory spaces introduced in 4.3: __global, __local, __constant and __private. These qualifiers can be used as a

prefix to function arguments and when declaring pointers inside the kernel function or when calling auxiliary functions.

The qualifiers for objects of type image2d_t and image3d_t are __read_only and __write_only where read only is the default and may thus be omitted.

4.7 OpenCL on the Mac Platform

Since version 10.6 MacOS X provides a native implementation of the OpenCL 1.0 specification. It is implemented as a framework (OpenCL.framework) containing the API, OpenCL compiler as well as an interface to the runtime engine. Since OpenCL provides a C API the library can be used in any C, C++ and Objective-C program.

The included OpenCL compiler uses internally the LLVM (Low Level Virtual Machine, [Lat02]) and its front end "clang" to compile the kernels and translate those instructions into optimized machine code for the hardware of the compute-devices. The same technology is used to compile OpenGL code. In this process the code converted to an intermediate representation then optimized and finally passed to the compiler that translates it into instructions for the specific device [Inc09b].

Chapter 5

Kernel Programs

The kernel-code forms the heart of any OpenCL application. This chapter gives a detailed insight to the workings of the OpenCL implementation by walking step by step through the code. The following section gives an overview of the data flow, memory buffers and datatypes used throughout the different stages of the extraction and matching process whilst each of the subsequent sections focus on one processing step. Wherever possible the intermediate results are presented to achieve a better understanding of the underlying data used to find the interest points and matches. Every section features a schematic chart visualizing the operations performed by each work item. For convenience some placeholders are used that correspond to #define statements in the source code. Table 5.1 lists these definitions and gives some typical values for each of them.

Definition	Description	Typical Value
NUM_IPTS	maximum number of interest points	256
WIDTH	width of the source image	720
HEIGHT	height of the source image	480

Table 5.1: Definitions used in the source listings and figures.

5.1 Data Flow

This section gives an overview of the kernels and memory buffers used during the different processing stages from the interest point detected to the matching. The first step consists in loading the intensity map of the image to device memory. If the image data to be processed comes in RGB format it is first converted to grayscale according to

$$I = \begin{bmatrix} 0.299 & 0.587 & 0.114 \end{bmatrix} \begin{bmatrix} R & G & B \end{bmatrix}^T \tag{5.1}$$

where I is the intensity (luminance) value and R, G and B are the red, green and blue values of the corresponding pixel (as defined in [BT95]). This is done beforehand to reduce the amount of data being transferred from RAM to device memory which is the greatest bottleneck along the data flow path.

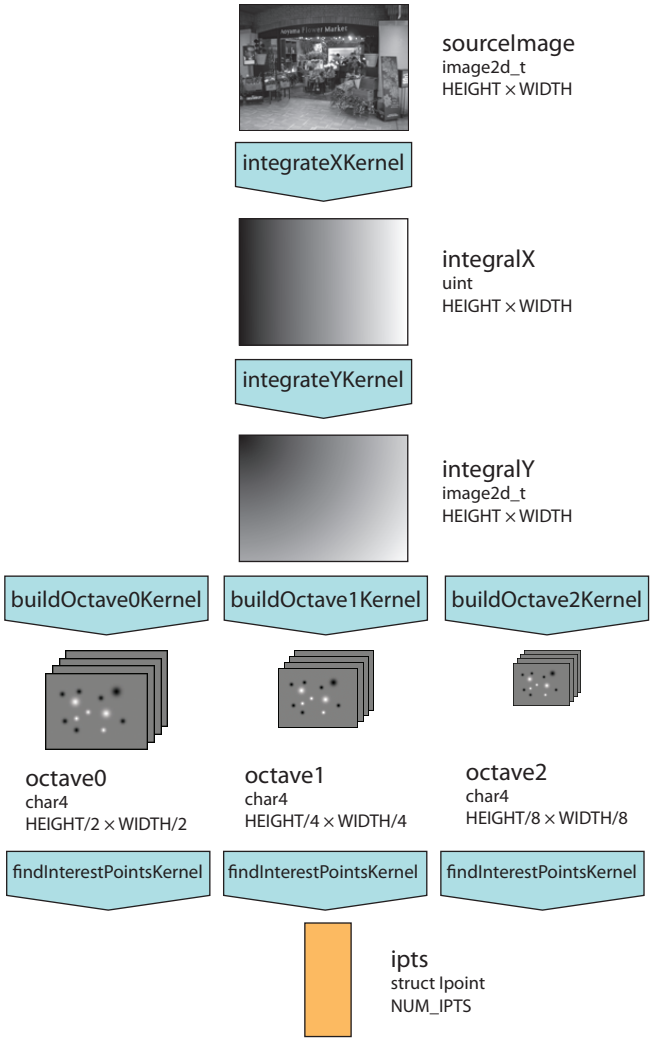


Figure 5.1: Kernels and data flow for interest point detection.

After the intensity map has been loaded to device memory integrateXKernel() performs as the name suggests an integration of this map in x-direction and writes the results to an array of unsigned integers of the same size as the input image (HEIGHT $\times$ WIDTH). IntegrateYKernel() takes this array as an input and integrates all values along the y-axis. The intermediate result, an array of the of unsigned integers of the same size as the input, is then copied to a two dimensional texture for fast access.

This texture image is used by the three kernels buildOctave0Kernel() through buildOctave2Kernel() to construct the three octaves each containing the four blob responses corresponding to the four filter sizes within each octave. Every four responses belonging to a position within an octave are stored in a char4. Finally three sets of work-items of findInterstPointsKernel() are spawned to "collect" the maxima of the response maps each processing one of the "stacks" computed in the previous step. The interpolated results are written back to an array of Ipoints (see listing 5.1) and length NUM_IPTS.

Listing 5.1: Struct used to store interest points

```
typedef struct {
        float x;
        float y;
        float scale;
        int sign;
} Ipoint;
```

Figure 5.1 shows the data flow up to the detected interest points where blue arrows symbolize the processing by the kernel and the boxes above and beyond the arrows the input and output. The three lines of text next to every orange box that symbolize the memory buffers give the name of the buffer used in the compute engine, its data type and dimensions.

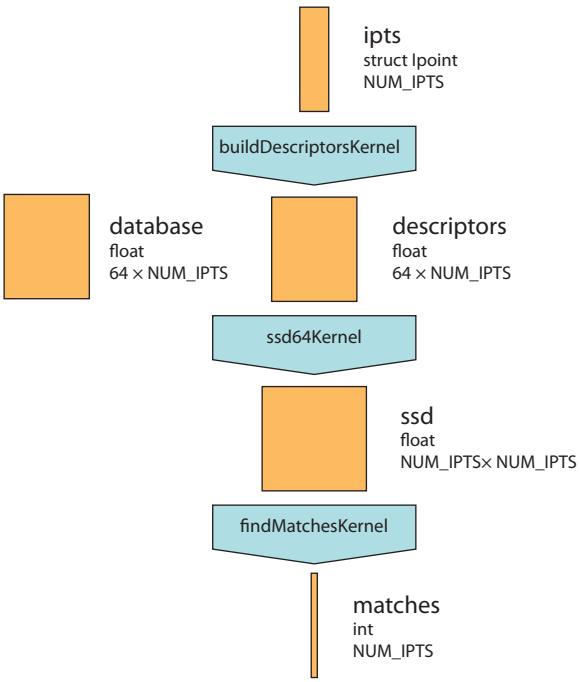


Figure 5.2: Data flow continued: interest point description and matching.

To find matches between interest points from different images it is first necessary to

extract the descriptors. BuildDescriptorsKernel() takes the list of detected interest points and writes the descriptors to an array of the same length where every interest point in the input array corresponds to one row of the output. The correlation matrix between the descriptors of the current image and a database of descriptors is calculated by ssd64Kernel(). Every element of this matrix is computed by one work-item. In this example the database is a copy of the descriptor array computed from the preceding image which can be the previous frame of a video sequence or an image taken of the same scene from a different perspective and therefore has the same size as the array holding the descriptors of the current image. Figure 5.1 visualizes the memory buffers and kernels used during the matching process.

5.2 Integration

To obtain the integral of the source image two kernels are used. The first kernel integrateXKernel() performs an integration in x-direction. It takes an image2d_t as an input which contains the intensity values of original image as 8-bit unsigned chars for each pixel and has the same dimensions as the original. In a first step the global position, the position inside the group and the group size are determined [1]. An unsigned integer is used to hold the sum of intensities [2] which allows for a maximum image size of $\frac{\text{UINT\_MAX}}{\text{UCHAR\_MAX}} = \frac{2^{32}-1}{2^8-1} > 16$ MPixels.

Listing 5.2: Source Code of kernelIntegrateX()

```
__kernel void integrateXKernel (
        __read_only image2d_t sourceImage,
        __global uint * integral,
        __local uint * area,
        const uint width )
{
        const int y = get_global_id(0); // [1]
        const int gi = get_group_id(0);
        const int ls = get_local_size(0);
        const int li = get_local_id(0);

        uint v = 0; // [2]

        for (int s = 0; s < width/ls; s++) { // for every segment

                // integrate [3]
                for (int xl = 0; xl < ls; xl++) {
                        int2 coords = (int2)(s*ls+xl, y);
                        uint4 pixel = read_imageui(sourceImage,
                                                   CLK_ADDRESS_CLAMP, coords);
                        v += pixel.x;
```

```
                area[li*ls+xl] = v;
            }

            // synchronize [4]
            barrier(CLK_LOCAL_MEM_FENCE);

            // write back results [5]
            for (int yl = 0; yl < ls; yl++) {
                integral[(gi*ls+yl)*width+(s*ls+li)] = area[yl*ls+li];
            }
        }
}
```

The kernel is executed over an index space of dimension HEIGHT $\times$ 1 which means one instance of the kernel is processing one row. Since every half-warp consist of 16 kernels trying to access 16 vertically aligned pixels a image_2d_t was chosen to avoid the loss in memory bandwidth that would occur when reading the values directly from a byte array. Another problem arrises from the fact that this access would have to be 32-bit aligned which requires that the loading is split up among the work-items of each group leaving one fourth of them idling during the loading. The result however consists of an array of unsigned integers which allows for an efficient store of the results.

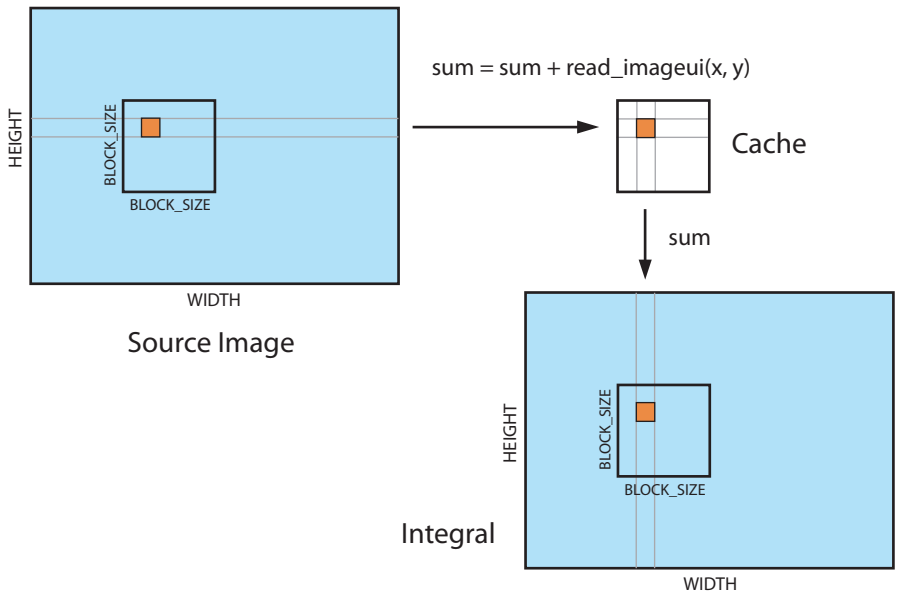


Figure 5.3: Block-wise calculation of the integral in x-direction using shared memory

The integral is therefore cached in an array of size BLOCK_SIZE $\times$ BLOCK_SIZE where BLOCK_SIZE is chosen to be 16. Every kernel runs two nested loops [3]. The outer loop iterates over the WIDTH / BLOCK_SIZE segments of width

BLOCK_SIZE. For each of these segments the inner loop accumulates the intensity values read from the source image. Every kernel writes one line of the segment according to its local id. After that the work group is synchronized to ensure that all kernels within the work group have finished their computations [4]. Finally the integral values are written back to global memory [5]. Therefore each kernel copies the column corresponding to its local_id_0 from the local cache to global memory. This process is repeated for every segment until the image has been processed.

Figure 5.3 shows process of reading the intensity value from the image, accumulating and writing it to the cache and finally writing the results row by row to global memory where every instance of the kernel is responsible for one column.

Figure 5.4: Image after integration in x-direction.

The result of the x-wise integration is shown in figure 5.4. In order to visualize the memory contents the values were divided by biggest possible value (WIDTH * UCHAR_MAX) such that after normalization black corresponds to a value of 0 and white to 1. The intensity map shown in figure 5.8 was used as an input for the integration.

Listing 5.3: Source Code of kernelIntegrateY

```
__kernel void integrateYKernel (
                __global uint * data,
                const uint width,
                const uint height )
{
        const uint x = get_global_id(0);
        const uint limit = x + (height-1) * width + 1;
```

```
        uint sum = 0;

        for(uint i = x; i < limit; i += width) {
                sum += data[i];
                data[i] = sum;
        }
}
```

The integration in y-direction works very similar to the integration in x-direction but can be implemented more efficiently since all kernels of a work group access aligned memory for every integration step in both source and destination memory corresponding to the intensity image to integrate and the integral respectively. Every instance of kernelIntegrateY() calculates one column of the integral where the column index corresponds to the global id of the kernel. After the global id has been determined the limit for the integration is calculated from the WIDTH and HEIGHT values of the image.

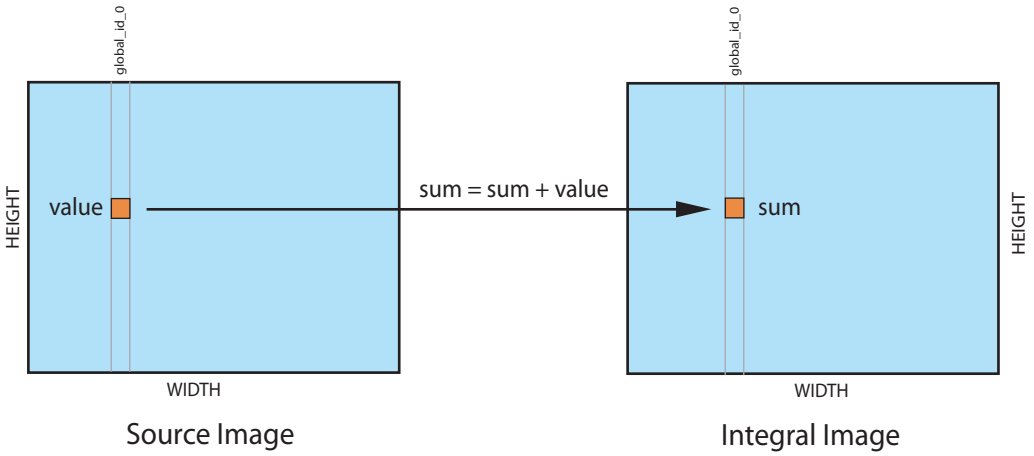
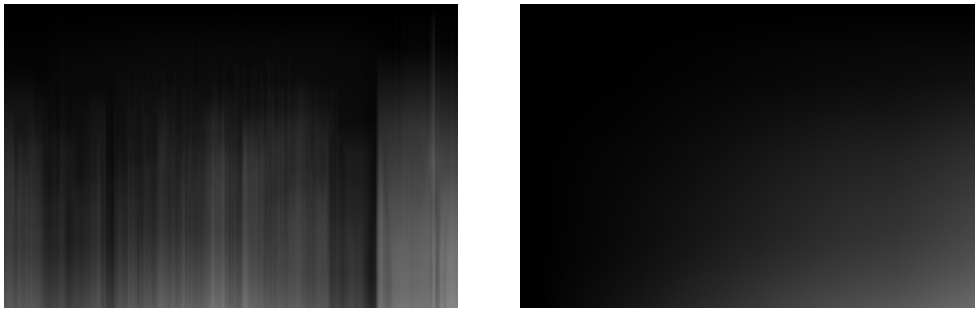


Figure 5.5: Element-wise computation of the integral in y-direction. Every kernel processes one column.

The integration is then performed in a for-loop as shown in figure 5.7. For every cycle of the integration loop one intensity value is read from the source image. This value is added to the sum and finally the new value of the sum is written directly to global memory. The loop terminates when the bottom of the image is reached. Since the integration is separable as shown in equation 3.2, the sum over the previous values is stored in a register and since the same data type is used for both input and output there exists only one argument holding the data.

Figure 5.6 shows the intensity map from figure 5.8 after integration in y- and both x- and y-direction. Similar to the x-wise integration the values where divided by the biggest possible value HEIGHT * UCHAR_MAX and WIDTH * HEIGHT *



(a) (b)

Figure 5.6: Image after integration in y-direction (left) and both x- and y-direction.

UCHAR_MAX such that after the normalization black corresponds to a value of 0 and white to a value of 1.

5.3 Computation of the Determinant of the Hessian

After the integral image has been computed the determinants of the Hessians for all octaves and intervals within the octaves need to be calculated. Therefore one set of kernels is spawned for every octave. Since every kernel operates on an output array that is one quarter of the size of its predecessor a different version of the kernel and output exist for every octave i.e. WIDTH / 2 × HEIGHT / 2 for the first octave, WIDTH / 4 × HEIGHT / 4 for the second and WIDTH / 8 × HEIGHT / 8 for the third. For simplicity only the kernel for the first octave is discussed here (buildOctave0Kernel) since they are very similar.

Like before (5.2) an image2d_t is used to hold the integral image because the integrated intensity values needed for the computation of the filter responses cannot be read in an aligned fashion as will bee seen.

The first step consists again in determining the global position of the kernel instance and defining a border around the edge of the input array [1]. This border is half the size of the biggest filter used in this octave rounded towards the next bigger integer value. For this example this would be $\lceil \frac{27}{2} \rceil = 14$. The global position is then checked against the border value and the response is only calculated for the inner area where it is defined [2].

Listing 5.4: Source Code of buildOctave0Kernel()

```
__kernel void buildOctave0Kernel (
                __read_only image2d_t integral,
                __global char4 * determinant,
```

```
                const uint width,
                const uint height )
{
        const uint x = get_global_id(0) * 2; // [1]
        const uint y = get_global_id(1) * 2;

        const uint border = 14;

        if (x < border || x > width-border || y < border || y > height-border) // [2]
                return;

        char4 det;
        det.s0 = detHessian(integral, x, y, 3);
        det.s1 = detHessian(integral, x, y, 5);
        det.s2 = detHessian(integral, x, y, 7);
        det.s3 = detHessian(integral, x, y, 9);

        determinant[(y / 2)*(width / 2)+(x / 2)] = det; // [3]
}
```

The response for the four filter sizes are then calculated by the function detHessian() and stored in a char4 where every 8 bits hold one response value. Finally the result is written to aligned global memory since every kernel in the half warp of the group writes $8 * 4 = 32$ bits [3]. The source code of buildOctave0Kernel() is given in listing 5.4.

To calculated the blob response the pointer to the integral image is passed on to the function detHessian() together with the global position of the kernel in x- and y-direction and the lobe size corresponding to the filters belonging to each interval within the octave. In this example the lobe sizes for the first octave are 3, 5, 7 and 9 corresponding to the filter sizes 9, 15, 21 and 27 i.e. one third of the filter size. A shortened version of the function is given listing 5.5.

Listing 5.5: Source Code of kernelIntegrateY()

```
inline char detHessian(__read_only image2d_t integral, uint x, uint y, uint l)
{
        uint w = 3*l;
        uint b = 2*l-1;
        uint b2 = b / 2;
        uint l2 = l / 2;

        const float area2 = w*w*w*w;

        float  Dxx = ( ImageBoxIntegral(integral, x-l-l2, y-b2, 3*l, b)
                        - 3 * ImageBoxIntegral(integral, x-l2, y-b2, l, b) );

        /* ... */

        float D = ( Dxx * Dyy - 0.81f * Dxy * Dxy ) / area2;
```

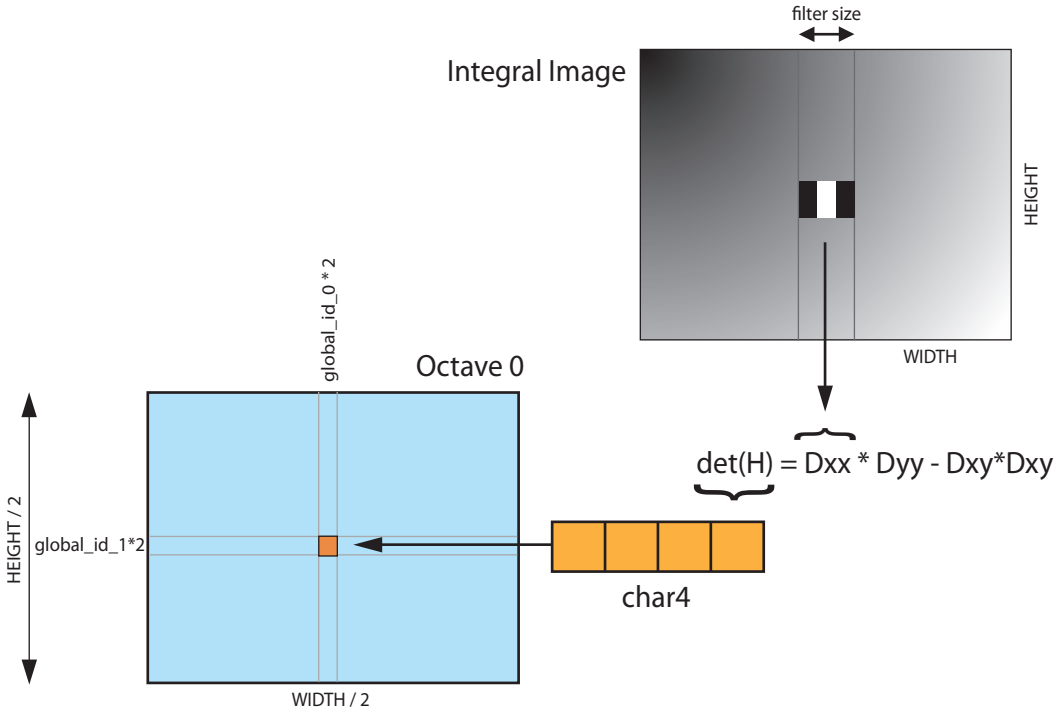


Figure 5.7: Calculation of the filter responses of the four filters in the first octave.

```
        // Get the sign of the laplacian
        int ls = Dxx + Dyy >= 0 ? 1 : -1;

        D = D < 0 ? 0 : ls * D;

        return (D*16.0f)*CHAR_MAX;
}
```

In a first step the lengths needed to construct the approximated filters of the second order gaussians are determined where w is the filter width and b the height of the boxes for the filter that yields the response in x-direction and the width of the filter that yields the response in y-direction respectively. The sum over the intensity values within each box is calculated by calling the function ImageBoxIntegral() which takes the integral image, the position in x- and y-direction and the size of the rectangle to integrate as arguments where col and row are the top left coordinates within the source image and columns and rows the number of pixels in x- and y-direction of the rectangle. Listing 5.6 shows the source of ImageBoxIntegral().

Listing 5.6: Source Code of ImageBoxIntegral

```
inline uint ImageBoxIntegral (
```

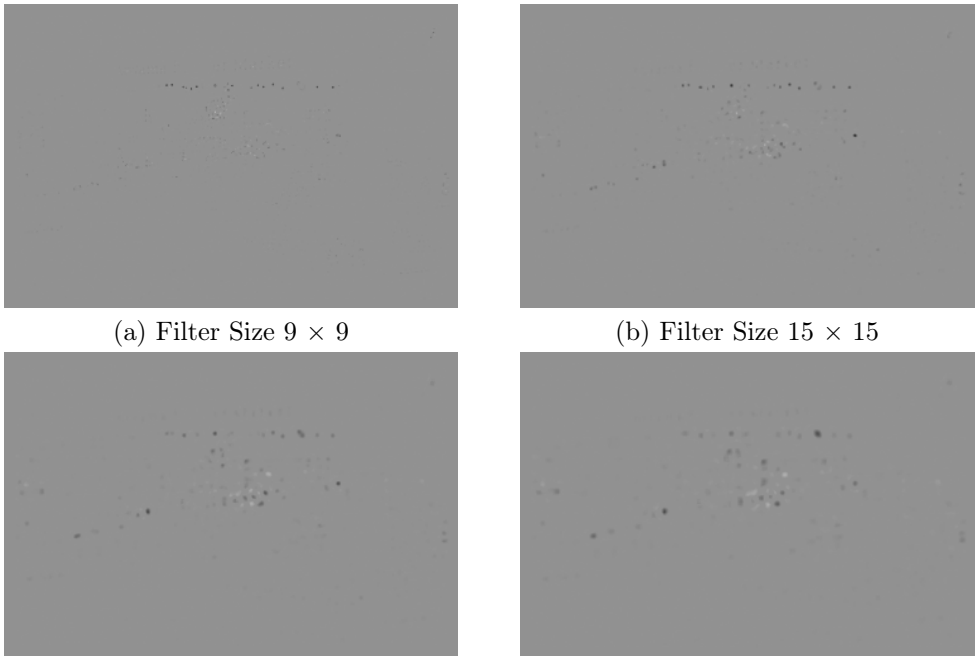
```
                __read_only image2d_t image,
                int col, int row, int cols, int rows )
{
        // The subtraction by one for row/col is because row/col is inclusive.
        int r1 = row - 1;
        int c1 = col - 1;
        int r2 = row + rows - 1;
        int c2 = col + cols - 1;

        uint A = read_imageui(image, CLK_ADDRESS_CLAMP, (int2)(c1, r1)).x;
        uint B = read_imageui(image, CLK_ADDRESS_CLAMP, (int2)(c2, r1)).x;
        uint C = read_imageui(image, CLK_ADDRESS_CLAMP, (int2)(c1, r2)).x;
        uint D = read_imageui(image, CLK_ADDRESS_CLAMP, (int2)(c2, r2)).x;

        return A - B - C + D;
}
```

The responses obtained by filtering the test image depicted in figure 5.8 are given in 5.9. Every image represents the filter responses for one filter size where black corresponds to the minimum value of -128, grey to a value of 0 and white to the maximum value of 127. One can see how the response of the smaller light bulbs on the ceiling of the shop slowly fades towards the bigger filter sizes whereas the bigger ones yield more response as the filter sizes grow. Note that the determinant of the Hessian is computed for every second pixel of the source image in the first octave and thus has only half the width and height of the source image.

Figure 5.8: Intensity map of the flowermarket test-image.



(a) Filter Size 9 × 9 (b) Filter Size 15 × 15

(c) Filter Size 21 × 21 (d) Filter Size 27 × 27

Figure 5.9: Determinants of the Hessian for the different filter sizes of the first octave.

5.4 Interest Point Detection

The task of detecting a set of interest points across the three octaves can be split up into two distinct sub-tasks. First the maxima within the response map need to be found. This is achieved by applying a three dimensional non-maximum-suppression which is explained in the following. In the second step the maxima are localized up to sub-pixel accuracy by interpolating their position in both the spacial and scale domain.

For efficiency both steps are combined into one kernel which allows for the reuse of the data that is used for the non-maximum suppression and thus reduces the amount of bandwidth consumed for the localization of the interest points. This kernel is applied to every response map computed in the previous step with appropriate parameters.

5.4.1 Detection

After the global position of the kernel has been determined the corresponding char4 is loaded from the response map [1] where every 8 bit value represents a response within the range $[-128; 127]$. This information is used to check if one of the central

values is a maximum in scale-direction [2]. The read access to global memory is aligned because all kernels in the half-warp load neighboring 32-bit values from the input array.

If one of the central values is a maximum in scale space and exceeds a predefined threshold this value is compared to all the remaining $8 * 3 = 24$ neighbors in the scale and spacial domain [3]. The neighboring char4 values loaded for the comparisons are stored into a 2D array of char4 and size 3×3. After the last comparison this array contains the entire 3D-neighborhood of the detected maximum including the maximum itself. If however a greater value is found among the candidates the function returns immediately [4]. Figure 5.10 visualizes the loading process.

Listing 5.7: Source Code of findInterestPointsKernel

```
__kernel void findInterestPointsKernel (
                __global char4 * determinant,
                __global Ipoint * ipts,
                __global uint * count,
                const uint pitch,
                const float scale1,
                const float scale2,
                const float step )
{

        int x = get_global_id(0);
        int y = get_global_id(1);

        uc4 dn[3][3];
        dn[1][1].c4 = determinant[y*pitch+x]; // [1]

        // maximum
        char m = dn[1][1].c4.y;
        float scale;

        if (abs(dn[1][1].c4.x) > abs(m) || abs(dn[1][1].c4.w) > abs(m)) // [2]
                return;

        if (abs(dn[1][1].c4.y) > abs(dn[1][1].c4.z)) {
                m = dn[1][1].c4.y;
                scale = scale1;
        } else {
                m = dn[1][1].c4.z;
                scale = scale2;
        }

        // scale index in neighborhood
        int s = scale == scale1 ? 1 : 2;

        if ( abs(m) > 3)
```

```
{
        // check neighborhood
        char mn = 0;

        for (int yn = 0; yn < 3; yn++) { // [3]
        for (int xn = 0; xn < 3; xn++) {
                dn[yn][xn].c4 = determinant[(y+yn-1)*pitch+(x+xn-1)];
                for (int sn = 0; sn < 3; sn++) {
                        mn = max(abs(mn), abs(dn[yn][xn].ac[sn+(s-1)]));
                }
                if (mn > abs(m)) // [4]
                                return;
        }
        }
// ...
```

If the function does not return until the last iteration a local maximum is detected. Up to this point the position of the maximum is only know to a precision equal to the step size of the octave. In this example this would be 2 pixels in x- and y-direction and a difference in scale of $1.2\frac{12-9}{9} = 0.4$. To retrieve its exact position is necessary to interpolate the position as discussed in the next section.

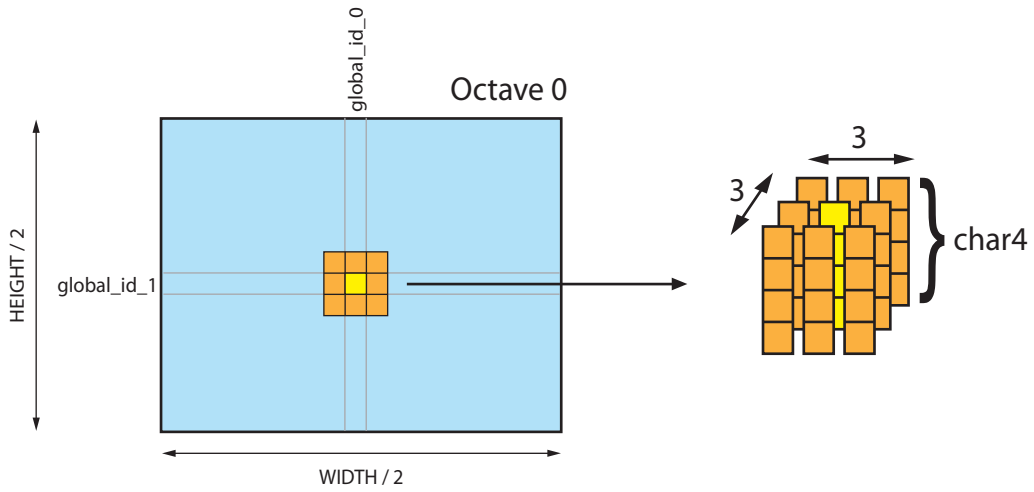


Figure 5.10: Calculation of the filter responses of the four filters in the first octave.

5.4.2 Localization

One problem of the SIMD approach is that all kernels are running virtually at the same time and thus it is only possible to share data among the instances in the same work group. Sometimes however it is necessary to synchronize information across all instances of the kernel which is the case here. The problem arises from the fact

that the non-maximum suppression cannot be executed over the output but over the input array whose size depends on the array holding the determinants of the Hessians for the different octaves. Thus one cannot know the number of interest points that will be detected in advance and allocating memory for every instance with enough space to store the detected interest point would result in a huge array that will in any case only sparsely be filled.

To overcome this problem a buffer for a pre-defined number of interest points (NUM_IPTS) is allocated. The number of detected interest points depends on the size of the image, the size of the used filters and number of octaves as well as the detection threshold. Therefore NUM_IPTS has to be chosen large enough to hold all detected interest points. For an image size of 720×480 and normal image contents this number will usually not exceed some 200 points.

Listing 5.8: Source Code of findInterestPointsKernel (Part 2)

```
// ...
        if (mn == abs(m)) {
                uint index = atom_inc(count);
                if (index < NUM_IPTS) {
                        float dx, dy, ds;
                        interpolate(&dx, &dy, &ds, dn, s);

                        ipts[index].x = (x + dx) * step;
                        ipts[index].y = (y + dy) * step;
                        ipts[index].sign = m > 0 ? 1 : -1;
                        ipts[index].scale = scale + (scale2-scale1) * ds;
                }
        }

    }

}
```

The global synchronization among the kernels is achieved by using a global index and an atomic_inc() function inside the kernels. This allows for the global assignment of an unique index for every detected interest point since the access to the global index happens in a synchronized fashion. Additionally this value can be used to adjust both the threshold and maximum number of allowed interest points dynamically.

After a local maximum has been detected this index is incremented by 1 and stored in a local variable. Then the offset between the position at which the interest point was detected and its exact position are computed by interpolate(). This function takes a 2D array of char4's as an argument that contains the neighborhood of the detected interest point which has been preserved from the previous step and the index of the scale that is needed to localize the maximum inside the central char4

without searching it again since the array also contains one scale layer that isn't needed for the computation. The variables for the return values that contain the offsets in x-, y- and scale-direction are passed in as pointers.

In the last step the offsets are added to the coordinates of the detected points and the sum is multiplied with the step size i.e. the sampling interval of the corresponding octave (2 for the first octave, 4 for the second and 8 for the third). The slope of the scales can be calculated from the difference of scale1 and scale2 passed in as arguments.

Figure 5.11 shows the result of the processing steps discussed so far. Light blobs on dark ground are indicated by red circles and dark blobs on light ground by blue circles and the centers are marked by a green dot. The diameter of each circle corresponds to the scale at which the interest point was detected.

Figure 5.11: Dark (blue circles) and light (red circles) interest points detected at different scales (diameter).

The two images in figure 5.12 show the top right region of the image with the detected interest point at their original position and after the interpolation step. One can clearly see how the circle around the two light bulbs that form one blob was increased in diameter and the center was slightly shifted towards the lower left corner

of the image. The correction resulting from the interpolation is generally bigger for large interest points since they are detected at the highest octave where only every eighth point is sampled. Also the difference between the scales becomes increasingly bigger towards the higher octaves.

(a) detected position and scale (b) interpolated position and scale

Figure 5.12: Interest points before (left) and after (right) interpolation.

5.5 Construction of the Descriptors

The construction of the descriptors is performed by the kernel buildImageDescriptorsKernel(). In order to achieve maximum performance in terms of bandwidth the extraction of every descriptor is split up into 16 distinct subregions as depicted in figure 3.10. Every subregion can be processed independently by one instance of the kernel. Thus every half-warp computes an entire descriptor for a given interest point. To further improve the bandwidth efficiency it is necessary to group at least two blocks of 16 work-items into one work-group so every group spans at least one warp. A local array of floats is used to cache the intermediate results which is important for the normalization of the descriptor vector as will be seen later. The local array of floats has dimension local_size_0 $\times$ descriptor_dimension. Figure 5.13 shows the memory layout used for buildImageDescriptorsKernel().

In a first step the location of the work-item inside the global an local context is determined. The global_id_0 corresponds to the index of the interest point to be processed and local_id_1 is used to assign the 16 sub-regions of the descriptor in row-major order to one work-item in a work-group. The local index of the descriptor i.e. the index inside the work-group is given by local_id_0.

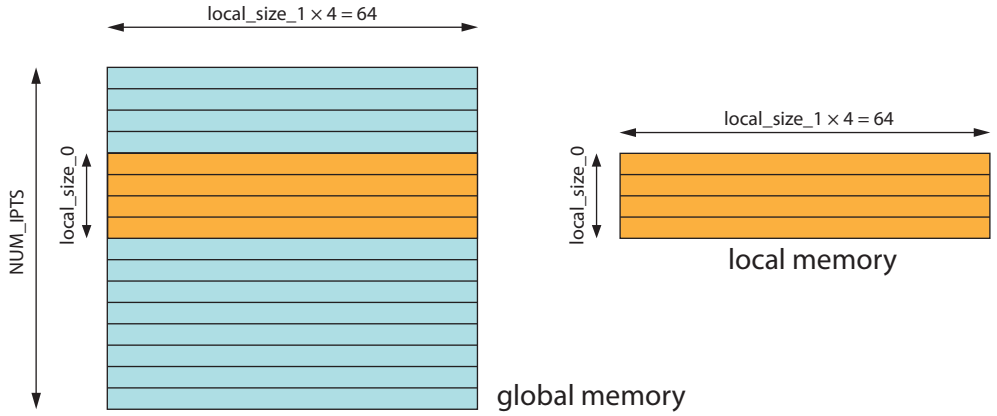


Figure 5.13: Global and local memory used for the computation of the descriptors.

After the coordinates of the interest point have been read from global memory [3] the four sums for the subregion are computed. Two nested loops are used to iterate over this region in y- and x-direction [4]. Inside the loop the global coordinates (x,y) are then calculated for all 25 points inside that subregion [5]. At every position given by (x,y) the Haar wavelet responses in x- and y-direction are calculated and weighted with a Gaussian centered at the location of the interest point [7] and finally accumulated in the corresponding sums for the responses and absolute values of the responses [8].

After the loop has returned the four sums are written to local memory again weighted with a Gaussian. This step is followed by a local barrier to assure that all work-items in the group have finished their computations and writing to local memory.

Listing 5.9: Source Code of buildImageDescriptorsKernel (Part 1)

```
__kernel void buildImageDescriptorsKernel( __read_only image2d_t integral,
                                __global Ipoint * ipts, __global float * descriptors,
                                __local float * cache, __local float * lengths)
{
        int index = get_global_id(0); // number of ipt [1]
        int lid0 = get_local_id(0); // index of descriptor in group
        int lid1 = get_local_id(1); // local index

        int xo = lid1 % 4 - 2; // [2]
        int yo = lid1 / 4 - 2;

        // read coordinates of interest point [3]
        int column = convert_int_sat_rte(ipts[index].x);
        int row = convert_int_sat_rte(ipts[index].y);
        float scale = ipts[index].scale;
        int s = convert_int_sat_rte(scale); // rounded scale
```

```
float dx = 0.0f;
float dy = 0.0f;
float mdx = 0.0f;
float mdy = 0.0f;

// iterate over the 5x5 inner boxes [4]
for (int yi = 0; yi < 5; yi++) {
for (int xi = 0; xi < 5; xi++) {
        int x = column + xo*(5.f*scale) + (xi+0.5f)*scale; [5]
        int y = row    +         yo*(5.f*scale) + (yi+0.5f)*scale;

        // get the gaussian weighted x and y responses [6]
        float rx = imageHaarX(integral, x, y, s) / 255.f;
        float ry = imageHaarY(integral, x, y, s) / 255.f;

        float gx1 = xo*5+xi;
        float gy1 = yo*5+yi;

        float gauss1 = exp(-(gx1*gx1+gy1*gy1) / 200.f); // [7]

        dx  += rx * gauss1; // [8]
        dy  += ry * gauss1;
        mdx += fabs(rx) * gauss1;
        mdy += fabs(ry) * gauss1;
}
}

float gx2 = xo + 0.5f;
float gy2 = yo + 0.5f;

float gauss2 = exp(-(gx2*gx2+gy2*gy2) / 4.5f);

int count = lid0 * 64 + lid1 * 4;

cache[count++] =  dx * gauss2; [9]
cache[count++] =  dy * gauss2;
cache[count++] = mdx * gauss2;
cache[count++] = mdy * gauss2;

// synchronize
barrier(CLK_LOCAL_MEM_FENCE);

// first kernels calculate lengths [10]
if (lid1 == 0) {
        float l = 0.f;
        for (int i = lid0*64; i < (lid0+1)*64; i++) l += cache[i]*cache[i];
        lengths[lid0] = sqrt(l);
}

// synchronize
barrier(CLK_LOCAL_MEM_FENCE);
```

```
        for (int i = 0; i < 4; i++)
                descriptors[index*64 + i*16+lid1] = cache[lid0*64 + i*16+lid1] / lengths[lid0]; //
}
```

For the normalization of the descriptor vector it is necessary to determine its euclidian length. Since this value depends on all the values in the vector it is most efficiently calculated by the first work-item assigned to each descriptor [10]. The concurrent computation in every work-item would still require local synchronization but cause bank conflicts because all work-items would try to access the same registers at the same time. This step is again followed by a barrier to assure local synchronization.

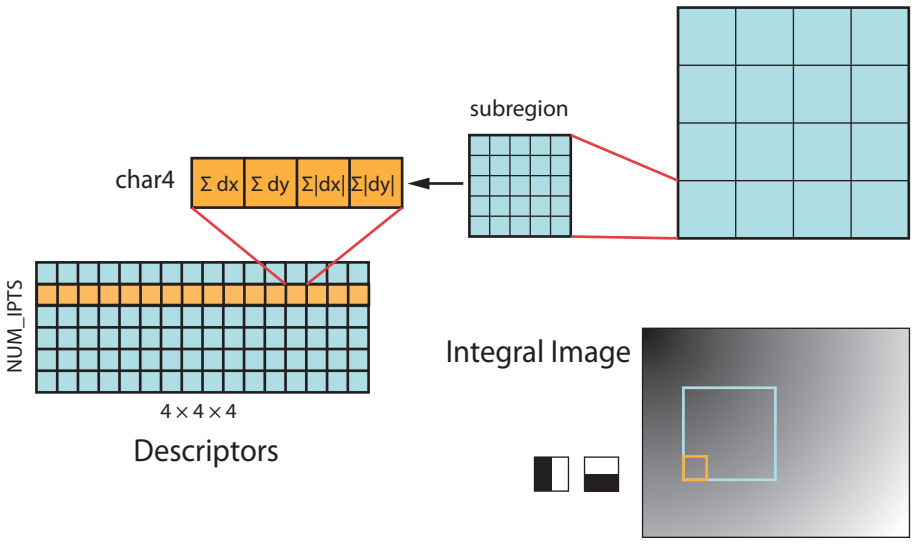


Figure 5.14: Calculation of the filter responses of the four filters in the first octave.

In a last loop every work item divides its four previously calculated sums by the length of the corresponding descriptor vector and finally writes the results to global memory [11]. Figure 5.14 visualizes the process of extracting the descriptor using the Haar wavelet responses calculated from the integral image where every work-item is responsible for one subregion.

5.6 Matching

To find the correspondences between the detected interest points in two images of the same scene it is necessary to define a measure of distance between the descriptors

of each interest point in order to select a best match. This is achieved by calculating the sum of squared distances between every two descriptors. The best match for a given interest point in one image is the interest point in the other image for which the distance is minimal.

5.7 Sum of Squared Distances

By writing every descriptor as a 64-vector $\mathbf{f}$ the feature in the current image and the descriptors in a database used for the comparison as $\mathbf{d}$ the sum of squared distances is calculated as

$$\mathrm{SSD} = \sum_{i=0}^{64} f_i d_i \tag{5.2}$$

where the subscript i denotes the element index. This database can contain any number of entries originating from images taken of the same scene e.g. previous images in a video sequence or images taken from different angles. To obtain the distances between every two vectors in the current image and the database the vectors $\mathbf{f}$ and $\mathbf{d}$ are stacked to form the matrices $\mathbf{F}$ and $\mathbf{D}$ respectively. The elements $c_{i,j}$ of the correlation matrix $\mathbf{C}$ are determined as

$$c_{i,j} = \sum_{k=0}^{64} f_{i,k} d_{j,k}. \tag{5.3}$$

This matrix has dimension $m \times n$ where m is the number of features in $\mathbf{F}$ and m the number of entries in the database-matrix $\mathbf{D}$. Figure 5.15 shows the memory layout used by ssd64Kernel(). Note that the memory holding the database is also aligned in row-major order so it can be reused for the next matching step without reordering when moving on to the next frame of a sequence. The transpose $\mathbf{D}^T$ was chosen for better visualization.

For every element in $\mathbf{C}$ there exists one instance of ssd64Kernel(). This kernel calculates the sum of squared distances for the corresponding element in a straight forward manner according to formula 5.3. After the global position has been determined [1] a for-loop iterates over the two vectors $\mathbf{f}$ and $\mathbf{d}$ and accumulates the sum of the squared distance for every dimension [2]. Finally this value is written back to global memory [3].

Listing 5.10: Source Code of ssd64Kernel()

```
__kernel void ssd64Kernel (
          __global float * C,
          __global float * A,
```

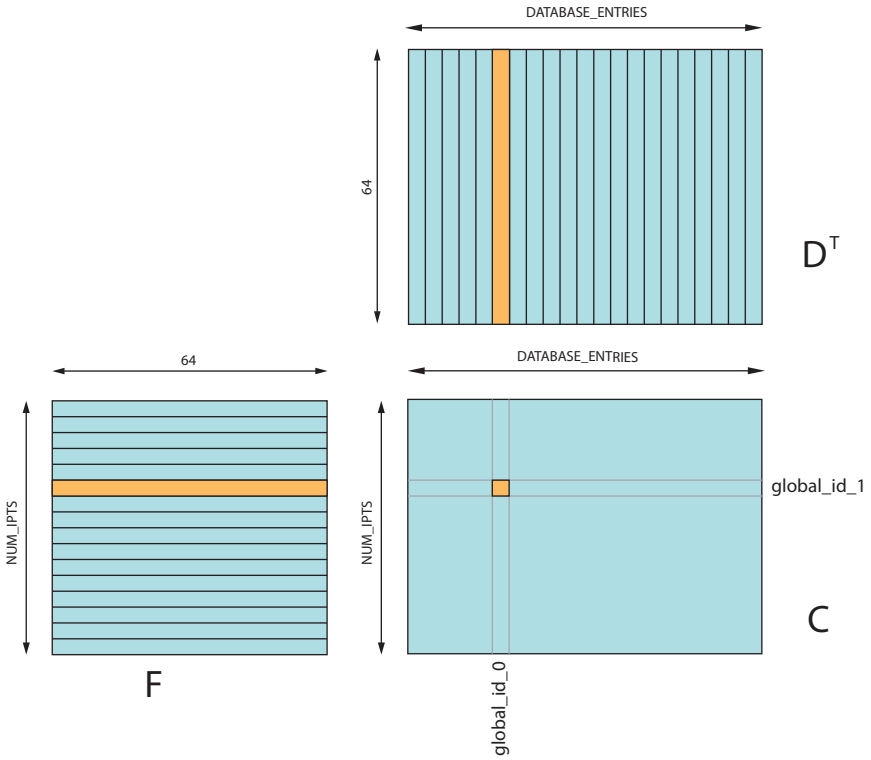


Figure 5.15: Memory layout for the computation of the correlation matrix.

```
          __global float * B
          )
{
        int row = get_global_id(1); // [1]
        int col = get_global_id(0);

        float sum = 0.f;

        for (int i = 0; i < 64; i++) { // [2]
                float d = A[row * 64 + i] - B[col * 64 + i];
                sum += d * d;
        }

        C[row * NUM_IPTS + col] = sum;  // [3]
}
```

To retrieve the indices that map every descriptor in **F** to the best match in **D** the kernel findMatchesKernel() is used. Every work-item is responsible for finding one match resulting in NUM_IPTS instances - one for every series of distances in the correlation matrix. Figure 5.16 visualizes the process for one work-item.

After determining the row in **C** to process the sign of the feature corresponding to the current row is read from global memory [1]. Then a for-loop iterates over all

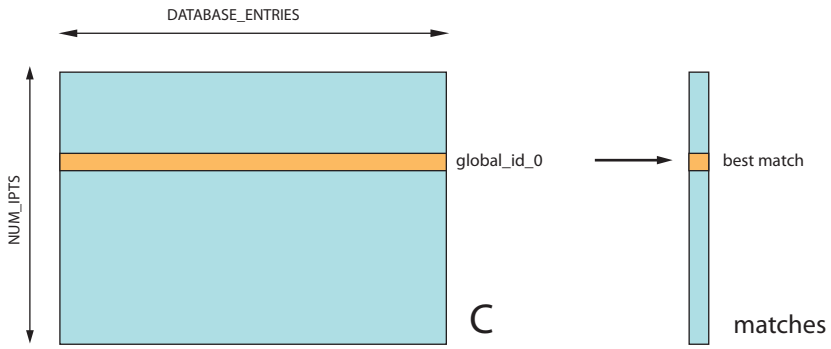


Figure 5.16: Every work-item searches one line of **C** for a suitable match.

interest points in the database [2]. If the interest point passes the sign test (see section 3.5) the distance between the current vector and the vector in the database is evaluated [3]. The lowest and second lowest distances are stored for a final comparison in [4]. If the ratio between the best and second best match is lower than a pre-defined threshold the index of the best match is stored in the vector matches. Since the indices can only be in the range $[0; \text{NUM\_IPTS}]$ a negative value indicates that no suitable match could be found in the database.

Listing 5.11: Source Code of findMatchesKernel()

```
__kernel void findMatchesKernel (
        __global const Ipoint * ipts,
        __global const Ipoint * cand,
        __global const float * C,
        __global int * matches )
{

        int row = get_global_id(0);

        int mi = -1; // match index

        float d1 = FLT_MAX;
        float d2 = FLT_MAX;

        int sign = ipts[row].sign; // [1]

        for (int i = 0; i < NUM_IPTS; i++) { // [2]

                if (sign * cand[i].sign > 0) // [3]
                {
                        float dist = C[row * NUM_IPTS + i];
                         // if this feature matches better than current best
                        if(dist < d1) {
                                d2 = d1;
```

```
                                d1 = dist;
                                mi = i;
                        }
                        // this feature matches better than second best
                        else if(dist<d2) {
                                d2 = dist;
                        }
                }
        }

        // If match has a d1:d2 ratio < 0.65 ipoints are a match [4]
        matches[row] = d1/d2 < 0.65f ? mi : -1;

}
```

Chapter 6

Evaluation Framework

The framework used to develop and evaluate the OpenCL code is comprised of three components. The "Realtime Object Detector" is a simple, platform independent and extendable application that allows the presentation and testing of image processing functions that are loaded as plug-ins. The ComputeEngine is a wrapper class that facilitates the handling of the OpenCL API. Finally the framework contains a number of unit and integration tests for the different kernels.

6.1 Used Libraries

6.1.1 OpenCV

OpenCV (Open Source Computer Vision) is a C library providing functions for real time computer vision originally developed by Intel. It is released under a BSD license and thus free for both academic and commercial use. It builds on all major platforms i.e. Windows, MacOS X and Linux and provides over 500 algorithms related to computer vision and image processing.

For this framework OpenCV was chosen because it provides a convenient and platform independent way of loading and connecting to all kinds of sources such as image files, videos and cameras. On the MacOS X platform can be built as a framework including all headers and library files which allows for an easy development.

OpenCV was chosen for this framework because it provides a convenient and platform independent way of loading and connecting various sources such as image files, videos and cameras. Also, it can be built on the MacOS X platform as a private framework[1] that includes all headers and shared libraries, allowing for easy devel-

[1] On the Mac OS X platform "framework" refers to a "*hierarchical directory that encapsulates shared resources, such as a dynamic shared library, [...] header files, and reference documentation in a single package*" [Inc06]

opment.

6.1.2 Qt

Qt is an application development framework that was originally released by Trolltech and is now maintained by Nokia's Qt Development Frameworks division. The library is distributed under the GNU Lesser General Public License and is free for non-commercial use. In addition to the GUI features it provides functions for XML parsing, thread management, network support and a unified cross-platform API for file handling.

As of version 4.1 Qt features a comprehensive unit-testing framework that provides the functionality and infrastructure to perform component based tests on both GUI and non-GUI components. Another important aspect is the support of a plug-in mechanism that allows for runtime loading of external modules without the limitations of a C-based API.

6.2 ROD - Realtime Object Detector

ROD serves as a versatile and extendible demonstration application that allows for real-time processing and viewing of all kinds of image data such as images, videos and frames taken from an attached camera. Figure 6.1 shows a screenshot of the application processing input from a camera.

Figure 6.2 gives an overview of the architecture with its main class MainWindow. After the Source and Processor plug-in have been loaded they are attached to the processing thread class controlling the execution. Once the loading process is finished the run-method of the ProcessingThread is started.

For every time step, the ProcessingThread loads one frame from the Source and passes it on to the ProcessorPlugin. The ProcessorPlugin performs its computations on the input image, optionally previous frames and external data, and eventually returns the result.

This result can be an augmented version of the input as shown in figure 6.1 where a number of detected interest points have been marked or any intermediate result. The result doesn't have to be the same format or size as the input image will automatically be converted and scaled to match the size of the (resizable) ImageView component. This is especially useful when debugging a ProcessorPlugin (see Section 6.2.2).

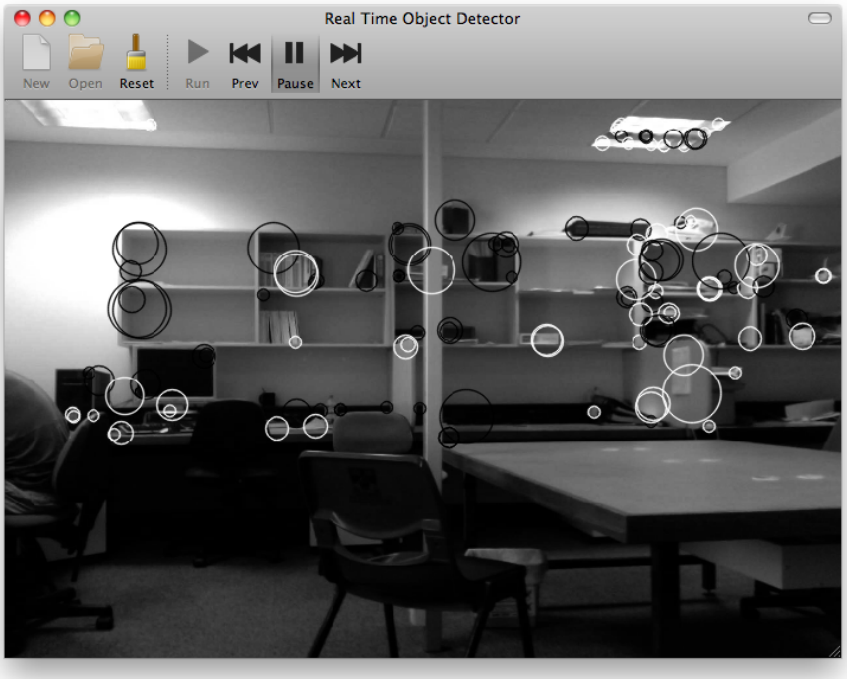


Figure 6.1: The ROD Application processing images from a camera.

The communication between the GUI and the ProcessingThread controlling the execution is handled using the signal and slot mechanism of the Qt framework that provides type and thread-safe means of communication between the components. All signals are connected to the corresponding slots of the receiver components during the loading process.

In this case most signals are emitted by the controls pictured in figure 6.2. Every time the user clicks a button, a signal is sent to the ProcessingThread. The ProcessingThread can then execute the appropriate action e.g. pause the run-loop or skip to the next frame. After a frame is processed by the ProcessorPlugin, a signal is emitted containing the result. This signal is connected to the ImageView component that displays the image.

Since all communication between the threads in the application is handled through signals and slots, the GUI stays responsive at all times. A finite state machine in the MainWindow class manages the state of the buttons and menu entries such that they are only enabled when the corresponding operation is available.

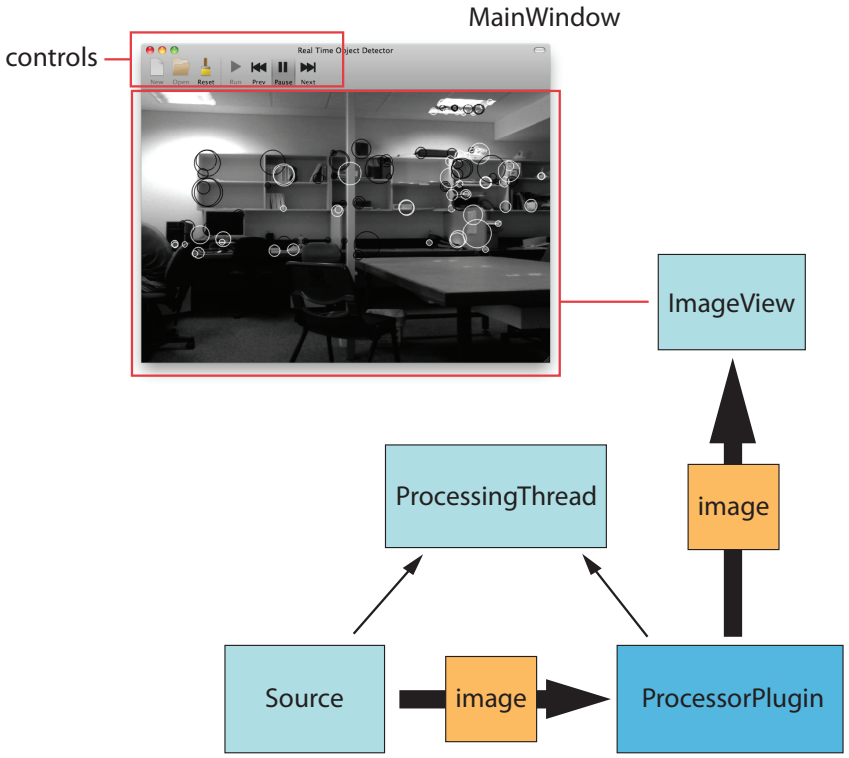


Figure 6.2: Architecture of the ROD Application

6.2.1 Sources

There are currently three types of sources that can be used to provide input images to the ProcessorPlugin: ImageSource to load static images from files, VideoSource for video files and CameraSource to grab frames from attached cameras. The number of supported formats depends on the codecs installed on the system. On MacOS X all common image and video formats as well as the built-in iSight cameras in MacBooks are supported.

Listing 6.1: Interface for the Sources

```
class Source : public QObject
{
public:
    virtual const IplImage * getIplImage() = 0;
    virtual int getWidth() = 0;
    virtual int getHeight() = 0;
    virtual double getFPS() = 0;

public slots:
    virtual void capture() {}
```

```
    virtual void skip(int numFrames) {}
};
```

All sources have to implement the interface presented in listing 6.1 through which the ProcessingThread and the ProcessingPlugin can retrieve information about the source e.g. the dimensions of the input image and the number of frames per second. The IplImage class serves as a container for the image data when passing it on from the source to the ProcessingThread. The operations inside the source are controlled via two slots: one for capturing frames and one for skipping frames. However calling theses slots may have no result in the case of images.

6.2.2 Processor Plug-Ins

The processing of the input images is done by the ProcessorPlugins. These plug-ins are compiled separately as a shared library and loaded during runtime by the ROD application. There is only one method that needs to be implemented as shown in listing 6.2.

Listing 6.2: Interface implemented by the processor plug-in

```
class ProcessorInterface
{

public:
    virtual const IplImage * process(const IplImage *input) = 0;

};
```

The process() method is called for every frame by the processing thread and is given the input image from the source as an input argument. After the method returns the result, which may be an augmented version of the input as depicted in Figure 6.1, the ProcessingThread passes it on as a signal that is received by the ImageView component.

6.2.3 Configuration Files

To a allow the user to create and store different combinations of sources and ProcessorPlugins the configuration is loaded from a simple XML-file that contains the source type and path as well as the path to the shared library that contains the plug-in. Listing 6.3 shows a sample configuration file with a video source.

Listing 6.3: Example configuration file using a video source

```
<!DOCTYPE Configuration>
```

```
<Configuration>
    <Source type="video" file="../videos/stan-f1.avi"/>
    <Processor file="../processors/libOpenCLProcessor.dylib"/>
</Configuration>
```

6.3 Compute Engine

The compute engine is used internally by the ProcessorPlugin and is essentially a wrapper for the OpenCL API that allows for more efficient coding and error handling by providing an additional layer of abstraction over the purely C-based API of OpenCL and is based on sample from [Inc09a].

The compute engine class is used as a singleton that manages the whole interaction between the host application code and the compute-device.

The typical use-case is a static instance of the ComputeEngine class. After the compute engine is connected to a compute device, the program is compiled and linked and the kernel-objects as described in 4.5. After the memory buffers are created using the provided methods of ComputeEngine and the initial data has been loaded to the input buffers the kernels can be executed. Finally the results are read back to the host application.

The major advantage of this wrapper-approach is the fact that it encapsulated the whole error handling process which would otherwise inflate the code dramatically and make it unreadable. This allows for streamlined coding, particularly during the development and debugging process.

To clarify this problem consider the following example: When a memory buffer object is created directly using the OpenCL function

Listing 6.4: Creating a buffer using the OpenCL API directly

```
cl_mem clCreateBuffer ( cl_context context,
                        cl_mem_flags flags,
                        size_t size,
                        void *host_ptr,
                        cl_int *errcode_ret )
```

the pointer to the memory object is returned and the error code CL_SUCCESS is written to an integer pointed to by errcode_ret. However if something goes wrong during this operation one of the following codes will be returned:
CL_INVALID_CONTEXT, CL_INVALID_VALUE, CL_INVALID_BUFFER_SIZE, CL_DEVICE_MAX_MEM_ALLOC_SIZE, CL_INVALID_HOST_PTR, CL_MEM_USE_HOST_PTR, CL_MEM_OBJECT_ALLOCATION_FAILURE

or CL_OUT_OF_HOST_MEMORY.

If this error code is not evaluated the application will crash leaving the user with no idea as to what went wrong. By using the corresponding method

Listing 6.5: Creating a buffer using the ComputeEngine

```
bool createBuffer( const char* acMemObjName,
                   MemFlags eMemFlags,
                   size_t kBytes )
```

of the ComputeEngine, a message is printed giving a reason and the name of the object in case the buffer-object cannot be created.

6.4 Unit-Tests

Unit tests are a new approach to assure the quality of any software product and has become popular with the concept of extreme programming that heavily relies on automated software test. A unit is the smallest testable component of a system. The idea behind this approach is to provide a controlled environment for the component to be able predict the result returned by the unit under test.

For this framework the units are the kernels that perform the calculations on the GPU. Since the kernel code cannot run directly on the CPU (unlike any other component in the framework such as e.g. the CameraSource) a common base class OpenCLTest is provided that handles all the setup required to compile and run the kernels on the graphics board.

Every time a test is executed, the function initTestCase() is called which connects the compute engine and allocates memory for input and output. It also loads all the data required for the test such as images and pre-calculated memory images as well as reference data. Listing 6.6 show the declaration of IntegrateXTest that derives from OpenCLTest.

Listing 6.6: Example configuration file using a video source

```
class IntegrateXTest : public OpenCLTest
{
Q_OBJECT

private slots:
        void initTestCase();
        void testIntegrateX();
        void cleanupTestCase();

};
```

During the test-phase the Qt test library (QtTestLib) calls all private slots of the test class starting with initTestCase(). A short message is printed after the return of every test function that indicates wether the test succeeded or not. Finally cleanupTestCase() is called which disconnects the ComputeEngine and releases the previously allocated memory before the next test is started.

Another important aspect in this context is integration testing. In order to assert that all components work together as expected there exists a test class that simulates the whole process from the input image to the feature extraction and matching. This is especially useful when changing the kernel code as this allows for convenient testing after every step (regression test).

Chapter 7

Results

This chapter is divided into two parts. Section 7.1 provides detailed timing results of the proposed implementation for a number of common image resolutions. In Section 7.2 the performance of the algorithm is evaluated and compared to the OpenSURF reference implementation running on the CPU.

Component	Description
CPU	Intel Core 2 Duo 2.4 GHz, L2-Cache 4 MB
Memory	4 GB DDR2 SDRAM
Operating System	Mac OS X 10.6.2
GPU	GeForce 8600M GT, 256 MB
Bus	PCIe x16

Table 7.1: Specifications of the MacBook Pro used for the evaluation.

All measurements where taken on a Apple MacBook Pro running MacOS X "Snow Leopard". The detailed specifications are given in table 7.1.

7.1 Profiling

This section lists the detailed timing results that where collected using the provided testing-framework and OpenCL profiling information from the GPU. The image sizes were chosen from a range that covers all common applications from small built-in cameras on embedded platforms like the the EyeBot [Bra99] to high resolution webcams as found in modern laptops. The number of pixels (image width $\times$ image height) grows almost linearly across the chosen resolutions which makes it easier to interpret the timing charts as the computational effort of most operations grows linearly with the number of pixels to process.

7.1.1 Total Execution Times

To give an overview of the total execution time figure 7.1 shows the accumulated kernel execution and memory copy times for one image at the different resolutions. For comparison the corresponding execution times of the reference implementation running on the CPU are included in figure 7.2. In both cases the execution times grow linearly with the number of processed pixels. The speed-up by the GPU implementation is more than an order of magnitude.

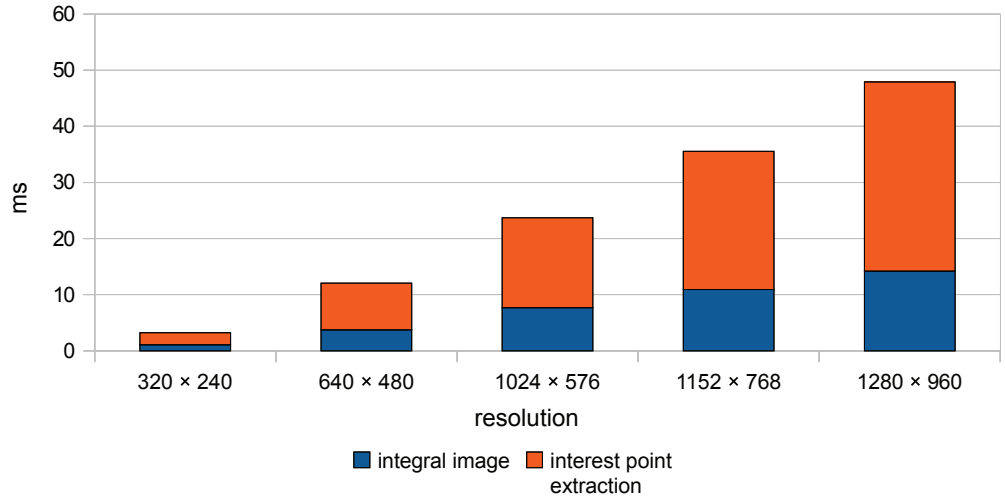


Figure 7.1: Total execution times on the GPU

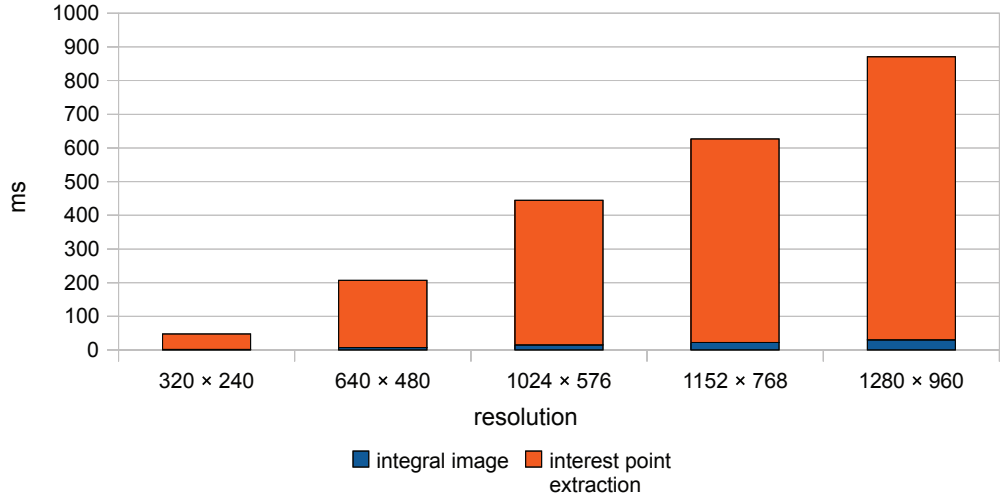


Figure 7.2: Total execution times on the CPU

7.1.2 Integral Image

Figure 7.3 and 7.2 show the execution times for the calculation of the integral image i.e. the integration in x- and y-direction. Even though the times for writing the initial image to the compute-devices memory and copying the results to an image2d_t are not part of the actual calculation these times have been included into this step to make it more comparable to other implementations. Again the required time grows linear with the size of the image.

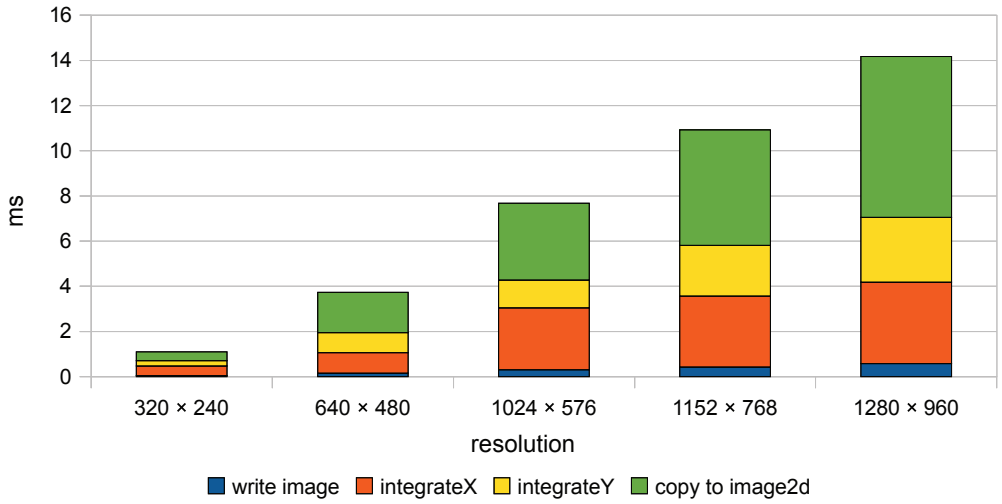


Figure 7.3: Execution times for the computation of the integral image on the GPU

resolution	write image	integrateX	integrateY	copy to image2d
320 × 240	0.0528	0.4244	0.2293	0.3947
640 × 480	0.1607	0.9070	0.8808	1.7820
1024 × 576	0.3038	2.7465	1.2231	3.4082
1152 × 768	0.4247	3.1381	2.2509	5.1077
1280 × 960	0.5852	3.5918	2.8732	7.1254

Table 7.2: Execution times for the computation of the integral image

7.1.3 Calculation of the Determinants of the Hessians

For the calculation of the determinants of the Hessians three kernels are launched - one for every octave. The global work-size for each kernel is one quarter of the work-size of its predecessor starting with a global work-size of a quarter of the image size for octave0. Figures 7.4 and 7.3 show the execution times for the three kernels.

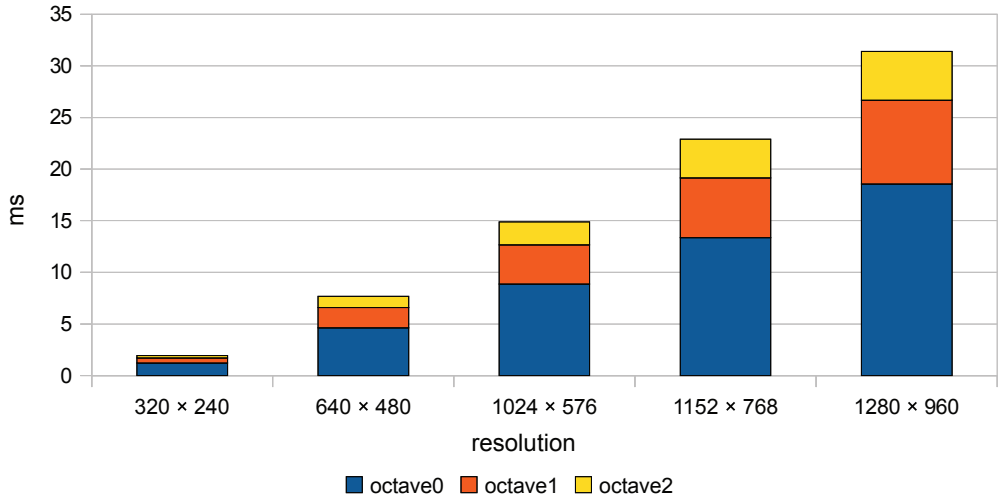


Figure 7.4: Execution times for the computation of the Determinants of the Hessians on the GPU

resolution	octave0	octave1	octave2
320 × 240	1.2209	0.4996	0.2069
640 × 480	4.6244	1.9720	1.0853
1024 × 576	8.8618	3.8055	2.2182
1152 × 768	13.3695	5.7793	3.7546
1280 × 960	18.5642	8.0873	4.7421

Table 7.3: Execution times for the computation of the Determinants of the Hessians on the GPU

7.1.4 Interest Point Detection and Localization

Like in the previous step one kernel is used to detect and interpolate the interest points in each octave. Figures 7.5 and 7.4 show the execution times for the three kernels. These times may however vary slightly depending on the number of local maxima in the determinants as the interpolation and storage of the interest points require additional memory accesses.

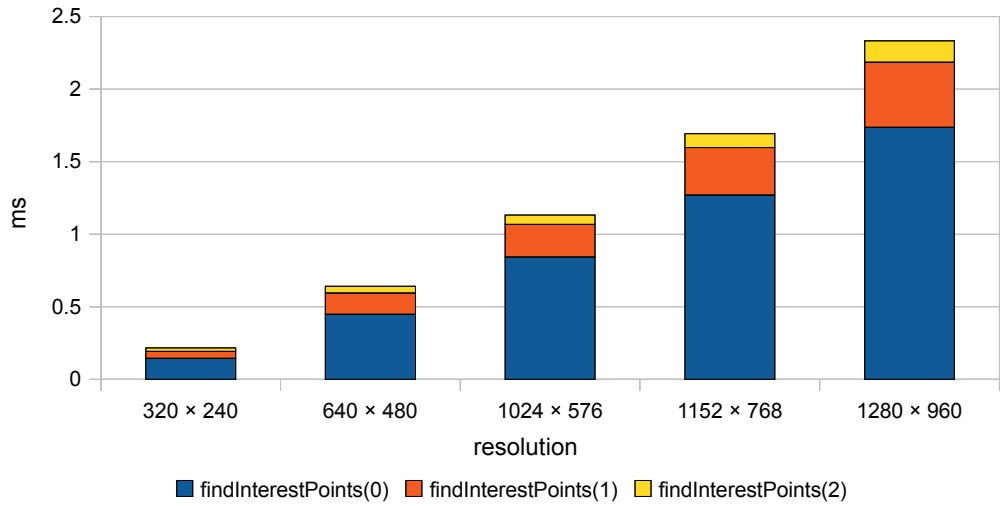


Figure 7.5: Execution times for the interest point detection and localization on the GPU

resolution	octave0	octave1	octave2
320 × 240	0.1456	0.0473	0.0231
640 × 480	0.4492	0.1456	0.0473
1024 × 576	0.8430	0.2240	0.0648
1152 × 768	1.2692	0.3275	0.0967
1280 × 960	1.7366	0.4492	0.1456

Table 7.4: Execution times for the interest point detection and localization on the GPU

7.1.5 Interest Point Description

The following chart and table show the times required to extract the descriptors for 32, 64, 128, 256 and 512 interest points from the same image. Since every descriptor is calculated by a constant number of work-items (16 per interest point) the execution time for this step scales linearly with the number of interest points.

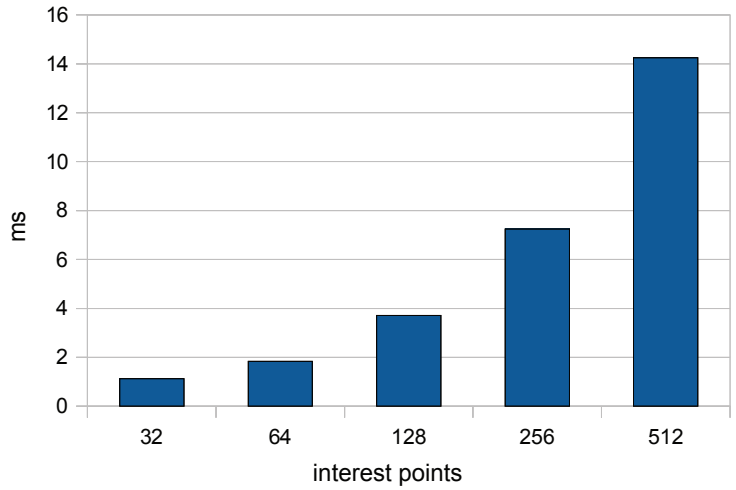


Figure 7.6: Execution times for the interest point description

Interest Points	Kernel Time [ms]
32	1.1227
64	1.8387
128	3.7107
256	7.2492
512	14.2471

Table 7.5: Execution times for the interest point description

7.1.6 Interest Point Matching

The dimensions of the correlation matrix depend on the number of descriptors from the current image and the size of the descriptor database. A common situation is a database that holds only the descriptors from a different image of the same scene and thus has the same size which results in a square correlation matrix.

Since every element of the correlation matrix is calculated by one work-item the execution time scales linearly with the number of its elements as can be seen in figure 7.7 and table 7.6. The time necessary to find the best match in the correlation matrix is very small compared to time for its calculation and is not listed.

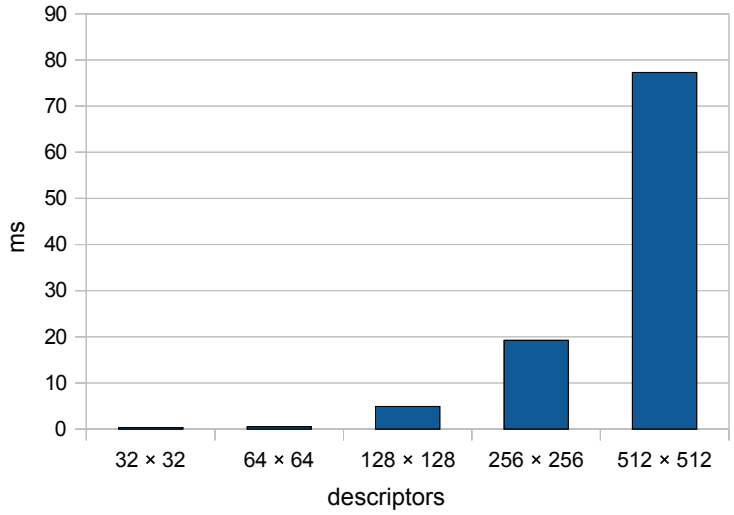


Figure 7.7: Execution times for the computation of the correlation matrix

Interest Points	Kernel Time [ms]
32 × 32	0.3102
64 × 64	0.5000
128 × 128	4.8558
256 × 256	19.2198
512 × 512	77.3037

Table 7.6: Execution times for the computation of the correlation matrix

7.2 Performance Evaluation

To evaluate the performance the proposed implementation has been tested using the datasets provided by Mikolajczyk [MS05]. The datasets contain a sequence of images taken from the same scene and the corresponding homography matrices that map the points in the first image to every other image in the dataset which allows to determine the correctness of the point correspondences found by an algorithm.

To evaluate the algorithm the point correspondences $[P_1, P_2]$ returned by the matching stage are considered where P_1 is a $m \times 3$ matrix with m representing the number of matches. Every line in P_1 and P_2 contains the coordinates of one interest point in the corresponding image and the last column is 1 $(x, y, z = 1)$. By multiplying P_1 with the transpose of the homography matrix H^T we get a matrix of the same size as P_2,

$$P_{2,ref} = P_1 \, H^T = \begin{bmatrix} (x_1, & y_1, & 1) \\ (x_2, & y_2, & 1) \\ & \vdots & \\ (x_m, & y_m, & 1) \end{bmatrix} \cdot H^T \tag{7.1}$$

that contains the coordinates of the correct correspondences for the points in P_1 in the other image after normalizing the coordinates by dividing every coordinate by the last column such that $\mathbf{x}_i = (\frac{x_i}{z_i}, \frac{y_i}{z_i}, 1)$. A point in $P_{2,ref}$ is considered a correspondence if the distance to the same point in the reference matrix $P_{2,ref}$ is lower than 5 pixels in x- and y-direction. The underlying theory and methods are explained in great detail in [HZ03].

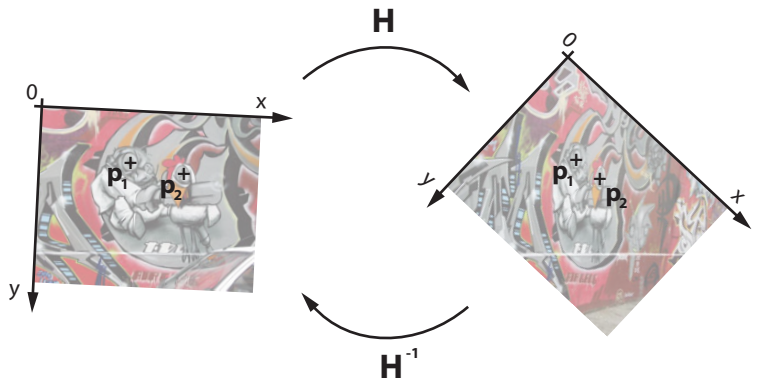
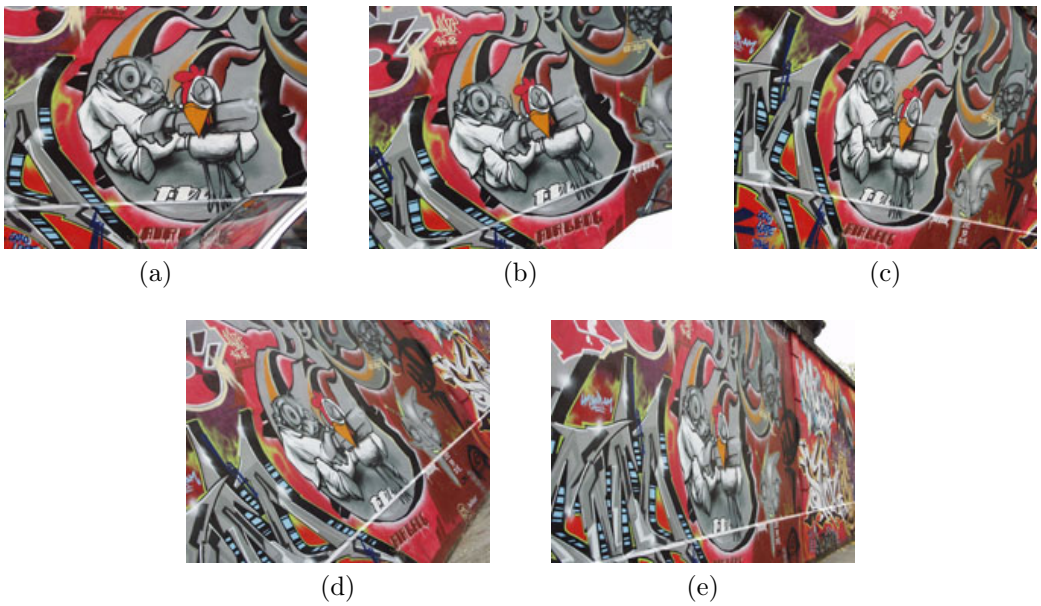


Figure 7.8: The homography transformation H maps points p_1 and p_2 between two images of the same scene.

Figure 7.9 shows the five images of the "graffiti" sequence. The first image in every sequence serves as a reference whose interest points are matched against the interest

points of image one through five.



(a) (b) (c) (d) (e)

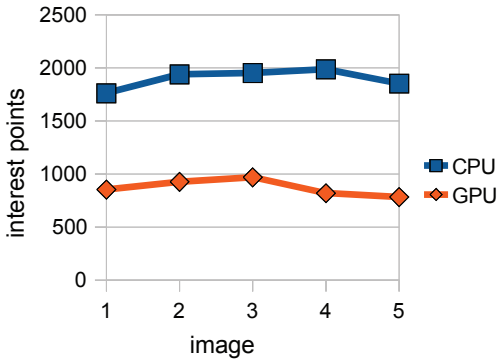
Figure 7.9: The graffiti image sequence

7.2.1 Number of Interest Points

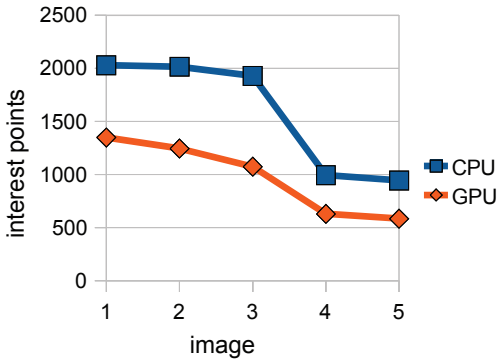
Table 7.7 lists the number of extracted interest points for the OpenCL (GPU) and OpenSURF (CPU) implementations. The results clearly show how the precision used to compute and store the blob responses affects the number of detected interest points. The CPU implementation uses 64 bit floating point whereas the GPU implementation uses only 8 bit per sample which results in a 8 times smaller memory footprint allowing for bigger image sizes and faster processing. This is especially important due to the memory and bandwidth limitations on the graphics board. On the other hand the lowered accuracy leads to reduction of the number of detected interest points by a factor of approximately two.

sequence	1	2	3	4	5
Graffiti	1761/853	1938/ 926	1950/ 969	1987/819	1852/783
Boat	2028/1349	2015/1244	1929/1073	993/629	944/584
UBC	1200/636	1197/630	1218/624	1183/630	1227/677
Wall	1946/466	1902/462	1791/438	1872/508	1893/530
Leuven	963/395	776/312	598/254	490/199	391/160

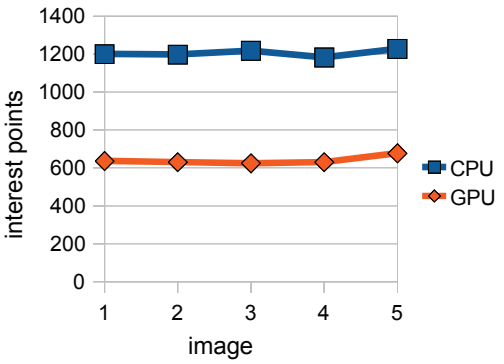
Table 7.7: Number of interest points for the different sequences (CPU/GPU)



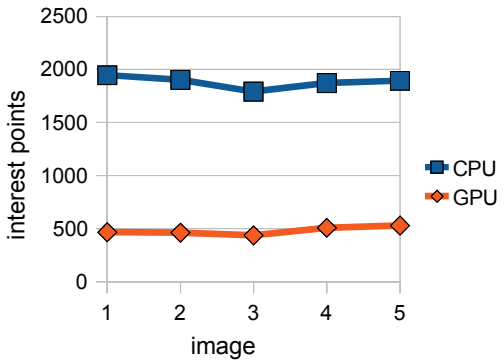
(a) Graffiti



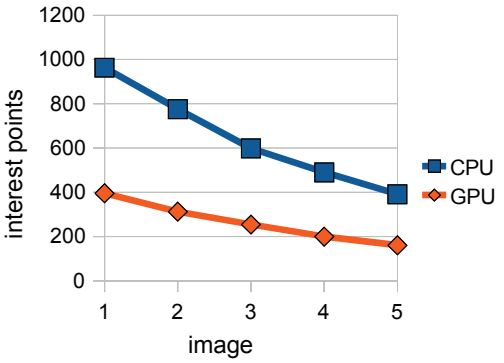
(b) Boat



(c) UBC



(d) Wall



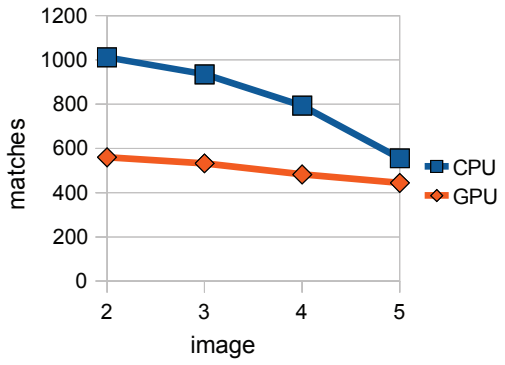
(e) Leuven

Figure 7.10: Number of extracted interest points for the different implementations

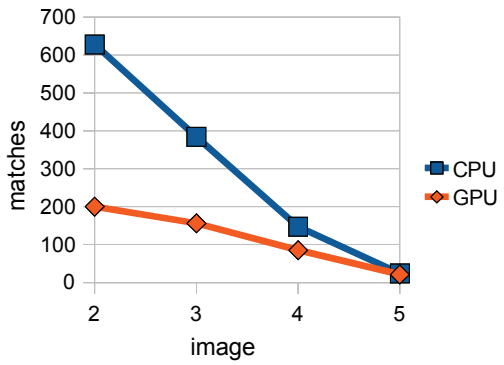
7.2.2 Interest Point Matching

The results of the matching process are given in table 7.8. The graffiti and boat sequences are only included for the sake of completeness. Both sequences feature large changes in rotation (as can be seen in figure 7.9) and are intended for the evaluation of rotation invariant descriptors whereas both implementations compared here run in "upright" mode i.e. it is assumed that there is less than 15 degrees of rotation.

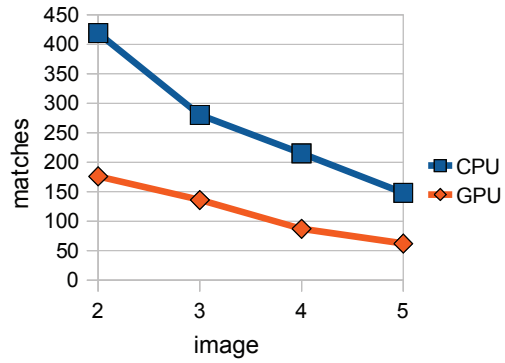
The larger number of correspondences for the first pairs (images 1 and 2) using the CPU implementation corresponds to the larger number of available candidates. The number of matches converges towards the end of the sequence. This is due to the fact that the (fewer) interest points detected by the GPU implementation tend to be stronger and thus can be matched more reliably.



(a) UBC



(b) Wall



(c) Leuven

Figure 7.11: Number of correspondences for the different implementations

sequence	2	3	4	5
Graffiti	13/91	15/51	1/0	2/2
Boat	151/171	0/0	0/0	103/9
UBC	1011/560	935/532	793/482	555/444
Wall	628/200	384/156	147/85	24/21
Leuven	419/176	280/136	215/87	148/62

Table 7.8: Number of correspondences for the different sequences (CPU/GPU)

Chapter 8

Conclusion

An implementation of the SURF algorithm has been proposed that runs entirely on the GPU. This implementation is accompanied by a framework that facilitates the development and testing of the kernel code. All components are independent of any platform specific libraries or APIs and all used libraries are multi-platform and free for non-commercial use. This makes all components usable in future projects without restrictions.

When writing OpenCL programs the challenge starts with the compilation of the kernel code. Since the kernel program containing all kernel and auxiliary functions is presented to the OpenCL compiler as a string during runtime it is not possible to compile the kernel code beforehand. The current version of the compiler does not provide helpful information when it fails to translate the source code into machine instructions which makes it harder to build bigger blocks of code or make big changes to the code at a time without compiling it in between.

Another big issue is the debugging of the kernel code that has taken up a lot of time during the development. As the memory used during the computations resides completely on the graphics board there are no means to use breakpoints and directly view the data during runtime. Further there is no means of generating and displaying messages from within the kernels. The printf() function can only be used when running the kernels on the CPU which does not support the use of shared memory and thus renders it useless for debugging of all kernels that rely on shared memory. The only way to get information about the data used during runtime is to use the output over which the kernels are executed to retrieve information.

As there are only very few metrics that help designing efficient kernel code it involves a lot of trial-and-error to find the best possible solution. There are a lot of parameters such as the number of used registers, the amount of shared memory per work-group and last but not least the size of the work-groups that can be adjusted. Sometimes small changes to the code or execution sizes can increase or decrease

the runtime of a kernel by up to an order of magnitude. Moreover changes to the memory layout in favor of reduced runtime can make it necessary to rewrite entire sections of both host and GPU code.

The results in Chapter 7 clearly show that a significant increase in speed of about one order of magnitude can be achieved when running the interest point detection and matching on the GPU compared to the reference implementation running on the CPU. Sacrificing the high accuracy for the determinant of the Hessians leads to a reduction in detected interest points of up to 50 percent. However since this affects mostly the weak interest points in the image and thus makes the matching more reliable for sequences that feature bigger changes in viewpoint or lighting conditions since the stronger interest points are generally easier to find in the corresponding images.

A future version running on GPUs that can process double precision floating point values and provide higher memory bandwidth could exploit the improved architecture to increase accuracy and match or even outperform the reference implementation. Using double precision when calculating, storing and comparing the descriptors should also slightly improve the matching performance. Another point that is promising a speed-up for the non-maximum suppression is the use of mip-maps to store the image pyramids through the OpenGL language bindings provided by the OpenCL language that allow for efficient sharing of image buffers on the graphics card. Another useful extension would be the orientation assignment as proposed by the SURF paper to make the description and matching process invariant rotation larger than ± 15 degrees.

List of Figures

References

[BL02] M. Brown and D.G. Lowe. Invariant features from interest point groups. *British Machine Vision Conference, Cardiff, Wales*, 2002.

[BL07] M. Brown and D.G. Lowe. Automatic panoramic image stitching using invariant features. *International Journal of Computer Vision*, 74(1):59–73, 2007.

[Bra99] T. Braeunl. Eyebot: A family of autonomous mobile robots. In *Proceedings of the 6th International Conference on Neural Information Prococessing (ICONIP99), 645649a*, 1999.

[BS08] Pavel Babenko and Mubarak Shah. Mingpu: a minimum gpu library for computer vision. *Springer-Verlag*, 2008.

[BT95] I.T.U.R.R. BT. 601-5: "Studio encoding parameters of digital television for standard 4: 3 and wide-screen 16: 9 aspect ratios.". *ITU, Geneva, Switzerland*, 1995.

[BTG06] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. *European Conference on Computer Vision*, 2006.

[Cor09a] NVIDIA Corporation. Nvidia opencl best practices guide, 2009. http:/www.nvidia.com/.

[Cor09b] NVIDIA Corporation. Opencl programming guide for the cuda architecture, 2009. http:/www.nvidia.com/.

[Cro84] F.C. Crow. Summed-area tables for texture mapping. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 207–212. ACM New York, NY, USA, 1984.

[Eva09] Christopher Evans. Notes on the opensurf library. Technical Report CSTR-09-001, University of Bristol, January 2009.

[GVL96] G.H. Golub and C.F. Van Loan. *Matrix computations*. Johns Hopkins Univ Pr, 1996.

[Haa10] A. Haar. Zur theorie der orthogonalen funktionensysteme. *Mathematische Annalen*, 69(3):331–371, 1910.

[Har07] Tom's Hardware. Intel releases 45nm processor details, 2007. http://www.tomshardware.com/.

[HZ03] R. Hartley and A. Zisserman. *Multiple view geometry in computer vision*. Cambridge Univ Pr, 2003.

[Inc06] Apple Inc. Macos x reference library, 2006. http://developer.apple.com/mac/library/documentation/MacOSX/Conceptual/BPFrameworks/Concepts/WhatAreFrameworks.html.

[Inc08] Apple Inc. Apple previews mac os x snow leopard to developers, 2008. http://www.apple.com/pr/library/2008/06/09snowleopard.html.

[Inc09a] Apple Inc. Mac OS X Reference Library, 2009. http://developer.apple.com/mac/library/.

[Inc09b] Apple Inc. OpenCL Programming Guide for Mac OS X, 2009. http://developer.apple.com/.

[Lat02] C.A. Lattner. LLVM: An infrastructure for multi-stage optimization. *Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec*, 2002.

[Low99] D.G. Lowe. Object recognition from local scale-invariant features. In *International Conference on Computer Vision*, volume 2, pages 1150–1157, 1999.

[Low04] D.G. Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[Mat10] Wolfram MathWorld. Matrix invers. http://mathworld.wolfram.com/MatrixInverse.html, 2010.

[MS04] K. Mikolajczyk and C. Schmid. Scale & affine invariant interest point detectors. *International Journal of Computer Vision*, 60(1):63–86, 2004.

[MS05] K. Mikolajczyk and C. Schmid. A performance evaluation of local descriptors. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1615–1630, 2005.

[Mun08] A. Munshi. The OpenCL Specification. *Khronos OpenCL Working Group*, 2008.

[SW09] M. Schweitzer and H.-J. Wuensche. Efficient keypoint matching for robot vision using gpus. In *Proceedings of the 12th International Conference on Computer Vision (ICCV09)*, 2009.

[Tru08] TrustedReviews. nvidia geforce gtx 280 review, 2008. http://www.trustedreviews.com/.

[VJ01] Paul Viola and Michael Jones. *Rapid object detection using a boosted cascade of simple features.* cvpr, 2001.

[Wit83] A. Witkin. Scale-space filtering. In *International Conference on Computer Vision*, volume 2, pages 1019–1021, 1983.