Real-time execution of image change detection

van Lint, R.H.

Award date:
2012

Master project Embedded Systems - 5T515

Real-time execution of image change detection

Hugo van Lint

Supervisor:
prof. dr. ir. P.H.N. de With (Eindhoven University of Technology)
Tutor:
dr. ir. E.G.T Jaspers (ViNotion)
Period:
April 2011 - January 2012
Student id:
0574437

Contents

Introduction

State-of-the-art video analysis systems feature multiple complex processing steps and operate on high resolution images. Intensive computation power is needed for real-time execution. In this project an image change detection application is mapped to a heterogeneous multicore CPU/GPU platform. It is investigated what hardware configuration is required to execute the application in real-time.

For optimal execution, i.e. minimum execution time, a choice has to be made which parts of the application to execute on the CPU and which on the GPU. The difference between CPU and GPU hardware architecture styles decreases. Historically CPUs were designed for low latency and GPUs for high throughput. In CPU architectures the trend has clearly turned towards multicore execution. On the other hand, GPU architectures have been optimized for general purpose execution. As a result, it becomes less obvious what processor type best suits an application.

Image processing algorithms mostly operate on large data sets, e.g. a full-size image or a set of image features. Therefore most algorithms are very well suited to be implemented on a GPU, which is designed for highly data-parallel applications. Many research project have shown that a significant speedup is possible for various image processing algorithms [PXW10], [CVG08], [ZCW10]. But not every algorithm is equally suited for GPU implementation, since the nature of the algorithm and the GPU architecture are uncooperative.

CHAPTER 1

Problem description

1.1 Overview of change detection application

The operation of image change detection is to move through an outdoor area while a camera records the journey. After some time, more or less the same journey is undertaken, and during the second recording the scene is compared to the first one in real-time. The system should reveal changes to the landscape, scenery, road and road conditions. A block diagram of the algorithmic steps in the application is given in Figure 1.1. The blocks have the following functionality:

Feature extraction Feature points and descriptions of the immediate neighborhood of feature points are extracted for every video frame. Features describe distinctive points in an image and are used to find correspondences between images.

Video stabilization To create a smooth video, video stabilization is performed to remove the effects of bumps and judder of the vehicle. Video stabilization utilizes the position of features in subsequent video frames to find displacements between frames. Abrupt displacements are smoothed.
The stabilized video frame data and corresponding feature data are stored in a database for comparison during a later acquisition.

Temporal synchronization During video capturing, temporal synchronization compares the features of a current video frame to the features in the database to find the matching video frame that was acquired at the same location.

Spatial alignment Since video frames are captured at discrete-time intervals and the vehicle is not driving at exactly the same position, the current video frames are not exactly aligned with the frames of the previously acquired video. Therefore, every current

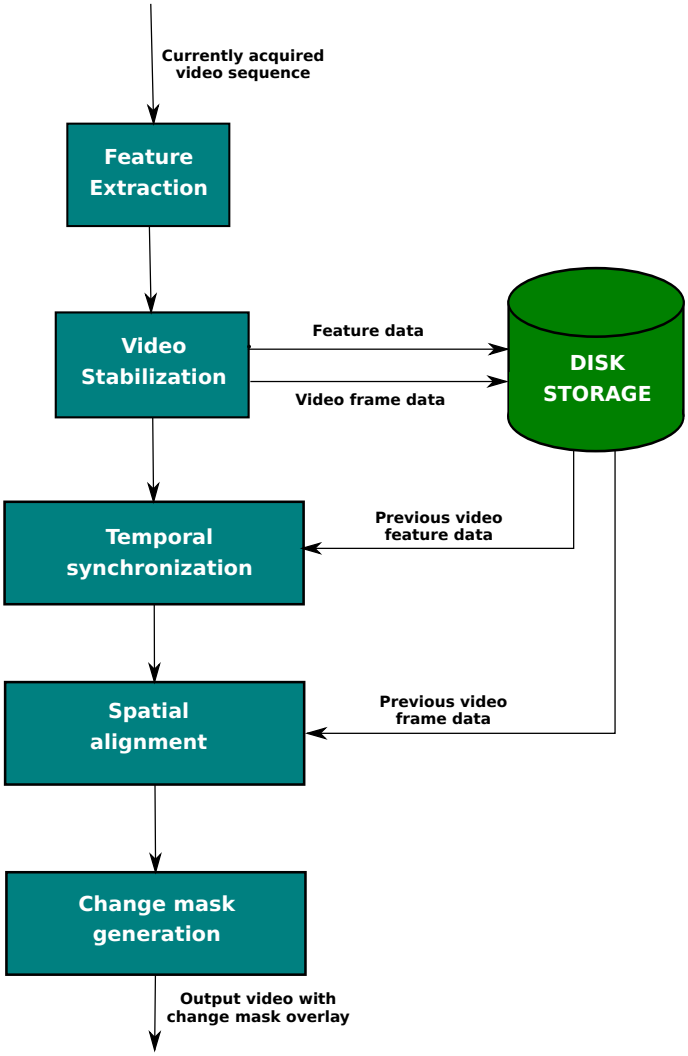


Figure 1.1: Change detection block diagram

video frame is spatially aligned to the matched previous video frame using a projective transform.

Change mask generation After alignment, the differences between the current video frame and the matched previous video frame are evaluated. The result is a binary change mask overlay.

1.2 Requirements

Before the problem statement is given, a number of requirements on the hardware platform of the change detection system are formulated.

Mobility The change detection system is operated in a vehicle, leading to specific constraints on size and power consumption. The system should function for a vehicle speed up to

20 km/h. The hardware platform must be portable from one vehicle to another for the same type of vehicle class.

Real-time execution The output of change detection should be presented to the user in real-time. The change detection result is not required to be updated for every output video frame. Instead, the result is required to be updated for every fifth output video frame, assuming a video camera with 25 fps output. That means the change detection output must be updated at a frame rate of at least 5 Hz.

Programmability Since the algorithms and the camera platform are new, flexibility is key and adjustments should be possible until the last moment. This leads to the requirement of a fully programmable platform.

Power consumption Depending on the vehicle a special additional generator or battery system has to be mounted in the vehicle. For the system prototype, the operation interval should be in the order of 1 hour.

1.3 Problem statement

The change detection application is an application with real-time requirements that must be mapped to a programmable platform. Given an initial C++ implementation, a mapping of application functions to a hardware platform consisting of a set of CPUs/GPUs should be found, such that the requirements are met. The mapping is performed for input video with a resolution of 640x420px (SD) and video of a resolution of 1920x1080px (HD).

To see how optimally the initial implementation is executed on the CPU, an execution time profile of application functions is created. The source code of the functions with the biggest contribution to the total execution time are analyzed, to assess how optimally they are executed and how they could be optimized. On the other hand, alternative image processing algorithms are investigated that are more computationally efficient than the initially implemented ones.

After an execution time profile on the CPU has been created and optimization opportunities are exploited, the extent to which GPU(s) can speedup execution is assessed. The execution time of OpenCV GPU implementations of the heaviest change detection algorithms is measured and besides, a new GPU implementation is created of an algorithm that has not been ported to the GPU yet.

Based on the execution time of the improved CPU functions and the execution time of GPU implementations, an advice is formulated on how to map the application to a set of CPUs/GPUs such that the change detection requirements are met.

CHAPTER 2

CPU/GPU architecture comparison

2.1 Introduction to CPU/GPU platform

In Figure 2.1 an overview is given of the experimental hardware platform. The hardware platform consists of a 4-core Intel Xeon W3550 processor equipped with 8GB RAM main memory. The graphics card is an NVIDIA GeForce GTX480 with a GPU that features 480 processing units and 1536 MB of device memory.

To understand the fundamental differences between a Central Processing Unit (CPU) and a Graphics Processing Unit (GPU), the architecture concepts of both processor are described in the following sections.

2.2 CPU architectures

Nearly all general purpose CPUs are *superscalar* designs today. Superscalar aims at *low latency* execution by techniques like branch prediction, speculative execution and out-of-order execution. These techniques prevent hazards that cause latencies in the pipeline. CPUs feature relatively large on-chip caches to store data that is access multiple times close to the processor to reduce memory access latency.

The current trend in the general purpose CPU technology is to increase performance by increasing the number of processor cores on a chip. Some years ago manufacturers of desktop processors achieved performance growth by improving their uniprocessor architectures. But that trend shifted to *multiprocessor* architectures, as clock speeds limits were being reached and the gap between processor and memory performance grew.

In a multiprocessor architecture each processor executes its own instruction stream. In most cases, each processor executes different processes. A process is defined as a segment of code that runs independently [HP03]. Tasks may be split into multiple processes. In this case the processes are called *threads*. Since the parallelism in this situation is contained in

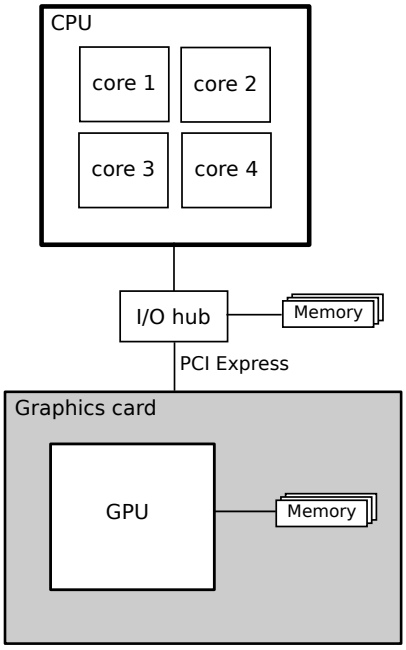


Figure 2.1: CPU-GPU hardware platform

the threads, it is called *thread-level parallelism* or *multithreading*.

2.3 GPU architectures

GPUs have evolved according to the needs of real-time computer graphics. This is a problem that is highly data-parallel. It is assumed that programs have abundant inherent parallelism. Therefore *throughput* instead of latency is the key measure of performance. This has leaded to the following architecture characteristics [GK10]:

- Many simple processing cores
- SIMD execution
- Extensive hardware multithreading to hide memory latency
- Relatively small on-chip memory

The increasingly high transistor density in semiconductor technology makes it possible to put many processing units on a chip. Multicore CPUs feature strategies like multiple out-of-order execution and branch prediction. The cores of multicore CPUs are relatively large due to hardware required to implement these strategies. GPUs typically don't apply these strategies that are aimed at low latency. This results in simpler and smaller processing units. This allows for more processing units on a chip and in result a high maximum *throughput*.

SIMD is adopted in GPUs because it increases the amount of chip area that can be used for functional units. An instruction only has to be decoded once while processing multiple

data values. This clearly requires less control logic compared to the case that an instruction is decoded for every computation. But the SIMD architecture style can only deliver peak performance with homogeneous tasks following the same execution trace. A program must offer sufficient amounts of uniform work to be effectively accelerated.

GPUs feature explicit hardware multithreading to allow fast switching between threads. An important benefit of multithreading is the ability to tolerate latency. If a thread is prevented from running because it waits for an event, e.g. a memory fetch, a multithreaded system can run another unblocked thread. This way latency is tolerated to increase the utilization of the processor.

GPUs don't provide an extensive caching system for load/store operations on external memory. This saves substantial chip area that can be spend on extra processing units. In the design of GPUs the assumption is made that there is abundant parallel work available to hide the latency of off-chip memory accesses.

CHAPTER 3

Application profiling and CPU performance improvements

To see how optimal the change detection application executes on the CPU and to investigate if real-time processing is feasible, an execution time profile of the application functions was created. The functions in which most time is spent were investigated, to see if they can be optimized. The application was tested with real-world video of an outside area, with both 640x420 px (SD) and 1920x1080 px (HD) resolution.

3.1 Measurement method

The execution time on the CPU was measured by the Intel VTune Amplifier tool. The time was measured by statistical profiling, in which samples of the running process are collected together with the function call stack. The performance metric used is the average execution time per frame. The average execution time per frame is obtained by measuring the total execution time for a video and dividing by the number of video frames.

3.2 Initial application profile

To get an overview of the execution time distribution in the application, an execution time profile per function category was created. The set of categories consists of the application blocks stated in Section 1.1 and the *overhead* category. The *overhead* category includes data operations that are not executed inside image processing algorithms, e.g. loading and storing video data from disk and conversion of video from one format into another.

3.2.1 Overhead

Table 3.1 lists the profile of the functions in the *overhead* category. First the purpose of the functions will be explained.

Function	t_{fr} (ms)		t_{rel} (%)	
	SD	HD	SD	HD
Vi::VideoInputVideoFile::read	28.4	164	2.68	1.22
Vi::VideoInputVideoFile::seek	5.41	22.9	0.511	0.170
Vi::VideoOutputVideoFile::write	8.91	79.1	0.841	0.558
Vi::Array2D<unsigned char>::fill	4.16	45.6	0.393	0.339
Vi::Array2D<unsigned char>::operator=	0.801	14.6	0.0756	0.109
ChangeDetection::Vi2OpenCV	10.3	104	0.976	0.773
ChangeDetection::OpenCV2Vi	6.54	66.3	0.617	0.492
ChangeDetection::cropBorders	2.91	27.3	0.275	0.203
ChangeDetection::drawOverlayChanges	1.31	9.06	0.124	0.0673
Vi::ImageStabilizationCIED::Vi2OpenCV	14.1	148	1.33	1.10
Vi::ImageStabilizationCIED::OpenCV2Vi	8.48	85.3	0.800	0.634
Vi::FeatureSIFT::load	430	5388	40.6	40.0
boost::archive::detail::interface_iarchive	132	1769	12.5	13.1
std::locale::id::_M_id	4.77	78.5	0.450	0.583
Total	659	8002	62.2	59.5

Table 3.1: Initial profile of *overhead* functions for SD and HD video. Column two shows the average execution time per frame and column three the relative execution time with respect to the execution time of the total application.

`Vi::VideoInputVideoFile::read` reads frames from a video file. By `Vi::VideoInputVideoFile::seek` the video is advanced by a number of frames. `Vi::VideoOutputVideoFile::write` writes video data to a video file on disk. `Vi:Array2D<unsigned char>::fill` is used to initialize frames of the output video. With `Vi:Array2D<unsigned char>::operator=` image data is copied from one variable into another. Both in the `ChangeDetection` class and the `Vi::ImageStabilizationCIED` class images are converted from ViNotion to OpenCV image format and vice versa by the `Vi2OpenCV` and `OpenCV2Vi` functions. `ChangeDetection::cropBorders` cuts the borders from the images after spatial alignment, because there is a high probability of distortions in edges of images. An image overlay showing the detected changes is drawn by `ChangeDetection::drawOverlayChanges`. In the change detection application video data of a previous acquisition and the corresponding extracted features are loaded from disk. `Vi::FeatureSIFT::load` loads the previously stored feature data from disk to compare it to the feature data of the current acquisition. In the initial application version the features are loaded/stored in text format. That implies that the binary feature data has to be converted into text. This is performed by `boost::archive::detail::interface_iarchive`. During conversion `std::locale::id::_M_id` retrieves the user locale: the country and language settings.

A very large part of the execution time is spent in *overhead* functions. In particular loading feature data takes much time. Loading the features takes in total 567 ms for SD video and 7236 s for HD video. This is a big bottleneck for real-time execution. `Vi::FeatureSIFT::load`

Function	t_{fr} (ms)		t_{rel} (%)	
	SD	HD	SD	HD
`Vi::FeatureDetectionSIFT::process`	279	3412	26.3	25.4
`Vi::FeatureDetectionSIFT::isExtremum`	0.417	4.63	0.0393	0.0344
`Vi::FeatureDetectionSIFT::computeGradientOrientation`	0.353	4.93	0.0330	0.0366
`Vi::FeatureDetectionSIFT::pixval32f`	9.03	109	0.852	0.813
Total	289	3531	27.3	26.2

Table 3.2: Initial profile of *feature extraction* functions

and `boost::archive::detail::interface_iarchive` must be further investigated to see if the procedure of loading and storing the features data can be optimized.
In the application two image processing libraries are used: the ViNotion library and the OpenCV library. These libraries use a different image format. Therefore video frames must be converted to the appropriate image format first before they can be processed. In this application version frames are twice converted from ViNotion format to OpenCV format and vice versa. For SD resolution this takes 39.4 ms and for HD resolution 404 ms. This means that image conversion is a significant performance bottleneck too.
Reading from the input video file takes 28.4 ms and 164 ms for SD and HD resolution respectively. This includes both reading from the current video file and reading from the previous video file. But reading from the previous video file takes about 20 times more time than reading from the current video file. The reading procedure is equal, so that means more video frames are read from the previous video file. It is caused by the synchronization of the current and previous video. Every frame from the current video is compared to 30 frames of the previous video to find the corresponding frame. Frames are compared by only checking the similarity of image features. Therefore the actual image data is not needed during synchronization, so the time spent in reading video frames can be reduced.

3.2.2 Feature extraction

Feature extraction is performed by the Scale-Invariant Feature Transform (SIFT) algorithm [Low04]. The algorithm performs both feature detection and feature description. The execution time profile of this category is shown in Table 3.2. `process` is the main function that calls the subfunctions needed to perform the SIFT algorithm. When looping through all images of the scale-space pyramid, `isExtremum` checks if an extremum is detected with respect to scale and space. `computeGradientOrientation` computes the orientation and the magnitude of gradients, while `pixval32f` casts image pixel values into the 32-bit floating point format.

About a quarter of the processing time is spent in feature extraction, 289 ms for SD and 3531 ms for HD video. To be able to extract features in real-time, this execution time must be drastically decreased. In the initial application version, features extraction is executed twice per frame. For video stabilization of the current video, features of two consecutive frames are needed. Extracted features of the current video are not stored after a frame has been processed. Therefore features must be recomputed. With a simple code modification the features of a frame can be stored so feature extraction is only performed once.

Function	t_{fr} (ms)		t_{rel} (%)	
	SD	HD	SD	HD
`Vi::ImageStabilizationCIED::imageStabMainProcess`	1.25	15.2	0.118	0.113
`Vi::FeatureMatching::process`	3.46	77.5	0.327	0.576
`cv::findHomography`	1.92	8.00	0.181	0.0592
`cv::warpPerspective`	5.94	59.8	0.560	0.444
Total	12.6	161	1.19	1.19

Table 3.3: Initial profile of *video stabilization* functions

Even if features are extracted once per frame, the execution time of SIFT feature extraction is too large. Therefore an alternative method will be investigated in Section 3.4.

3.2.3 Video stabilization

Table 3.3 lists the execution times of the *video stabilization* functions. The main function for video stabilization is `Vi::ImageStabilizationCIED::imageStabMainProcess`. It loads current video frames and feature data at time t and time $t-1$ and calls sub-functions to stabilize abrupt movements in the video. In a region of interest around the horizon `Vi::FeatureMatching::process`[1] compares features of frame t and frame $t-1$ to find features that correspond. After the matching features have been found, `cv::findHomography` finds the perspective transform between the two frames. By `cv::warpPerspective` the perspective transform is applied to frame t.

Video stabilization takes 1.19 % of the total execution time for both SD and HD video. This means that this function category has a minor contribution to the total execution time. The greatest part of the execution time is spent in feature matching and image warping. Although the functions take relatively little time, the execution time of both functions must be reduced to enable real-time processing.
It is remarkable that matching takes relatively more time for HD video than for SD video. This is caused by a higher number of features in the HD case. Therefore it is worth investigating both the feature matching procedure and required number of features for matching.

3.2.4 Temporal synchronization

In Table 3.4 the execution times of the *temporal synchronization* functions are stated. Synchronization is performed by comparing the features of the current frame at time t to features from frames in a window $[t-15, t+14]$. `Vi::FeatureMatching::process` matches features of two different frames and returns the closest match for every feature of the current frame. Thereafter, a criterium is applied to select the previous video frame that resembles the current video frame best.
To match features efficiently, the randomized kd-tree algorithm from the Fast Library for Approximate Nearest Neighbors (FLANN) is adopted [ML09]. In the `cv::flann::Index`

[1]The execution time of feature matching sub-functions are included in this function's execution time

Function	t_{fr} (ms)		t_{rel} (%)	
	SD	HD	SD	HD
`Vi::FeatureMatching::process`	4.25	480	0.401	3.57
`cv::flann::Index`	8.46	114	0.798	0.848
`cv::flann::Index::knnSearch`	30.8	566	2.91	4.21
Total	43.48	1160	4.10	8.62

Table 3.4: Initial profile of *temporal synchronization* functions

Function	t_{fr} (ms)		t_{rel} (%)	
	SD	HD	SD	HD
`cv::findHomography`	0.481	6.23	0.0454	0.0463
`cv::warpPerspective`	4.58	44.8	0.432	0.333
Total	5.06	51.0	0.477	0.379

Table 3.5: Initial profile of *spatial alignment* functions

constructor the kd-tree is built out of the features from a frame in the previously acquired video. The search for the nearest feature neighbor is performed in `cv::flann::Index::knnSearch`.

Temporal synchronization functions have a major impact on the total execution time. For SD video 4.10 % is spent in this category and for HD video 8.62 %. Like in *video stabilization*, the feature matching functions take relatively longer for HD resolution. This extra time is spent in `Vi::FeatureMatching::process` and `cv::flann::Index::knnSearch`.
In `Vi::FeatureMatching::process` the relative execution time (t_{rel}) is 0.401% for SD resolution, while it is 3.57% for HD resolution. This function only initiates the feature matching process, so it is not expected that the execution time for HD resolution is this high. The source code of the function must be analyzed to see where the time is spent.
`cv::flann::Index::knnSearch` is called for every feature in the current video frame. So the execution time depends on the number of retrieved features. As was stated in 3.2.3, it is worth investigating the number of features required for HD video input to see if it can be reduced.

3.2.5 Spatial alignment

The spatial alignment transforms frames from the currently acquired video so the perspective is equal to the perspective of the corresponding frame in the previously acquired video. Like in *video stabilization*, `cv::findHomography` finds the perspective transform between the two frames. `cv::warpPerspective` applies the transforms to the current frame. Table 3.5 lists the execution times of both functions. *Spatial alignment* has a relatively small impact on the application performance. The functions in this category scale well as resolution increases, because the relative execution time does not increase from SD to HD resolution.

Function	t_{fr} (ms)		t_{rel} (%)	
	SD	HD	SD	HD
`ImageDifference::changeVectorAnalysis`	0.769	10.3	0.0726	0.0765
`ImageDifference::buildDifferenceImage`	2.44	30.0	0.230	0.223
`AdaptiveThresholding::process`	2.76	19.0	0.260	0.141
`Block::buildFrequencyAppearance`	5.16	75.6	0.487	0.562
`AdaptiveThresholding::thresholdImage`	1.76	21.0	0.166	0.156
`Vi::ApplyMRF`	37.8	398.4	3.57	2.96
Total	50.7	554	4.78	4.12

Table 3.6: Initial profile of *change mask generation* functions

3.2.6 Change mask generation

After the video is temporally synchronized and spatially aligned, the image changes are detected in three steps. First the image difference is computed by Change Vector Analysis according to Equation (3.1), where $p^i(x, y)$ is the pixel channel value at position (x, y) for channels $i = \{R, G, B\}$. This difference is computed in `ImageDifference::changeVectorAnalysis`. In `ImageDifference::buildDifferenceImage` the differences are mapped to a [0-255] range (a gray-scale image).

$$D(x, y) = \left(\sum_i (p_1^i(x, y) - p_2^i(x, y))^2 \right)^{\frac{1}{2}} \tag{3.1}$$

In the second step `AdaptiveThresholding::process` creates a binary change mask by thresholding the difference image. To this end, the adaptive block-thresholding method by Su and Amer is adopted [SA06]. In the adaptive thresholding process, an image is divided into blocks and for every block a threshold is computed, based on scatter of regions of change (ROC). The global threshold is computed by averaging the thresholds of all blocks. To determine if an image block contains ROC, the first moment m_k of the histogram of each image block is computed, as defined in equation 3.2.

$$m_k = \sum_{i=1}^{g_{max}} i \cdot F_i, \tag{3.2}$$

where F_i is the frequency of gray-levels $i = \{1, 2, \ldots, g_{max}\}$ and g_{max} is equal to the maximum gray level. The histogram of each block is computed in `Block::buildFrequencyAppearance`. `AdaptiveThresholding::thresholdImage` applies the global thresholding to the input difference image.

Third, in `Vi::ApplyMRF` a method based on Markov Random Fields (MRFs) is applied to make the change mask more consistent and to reject artifacts from misalignment and image noise. Pixels are grouped by minimization of energy function

$$E(f) = \sum_{p \in \mathcal{P}} D_p(f_p) + \sum_{p,q \in \mathcal{N}} V_{p,q}(f_p, f_q), \tag{3.3}$$

where $\mathcal{N} \subset \mathcal{P} \times \mathcal{P}$ is the neighborhood on pixels [KZ04]. $D_p(f_p)$ measures the cost of assigning label f_p to pixel p and $V_{p,q}(f_p, f_q)$ measures the cost of assigning labels f_p, f_q to adjacent pixels p, q. The latter term imposes spatial smoothness at the border of objects. In this application the set of labels is equal to the binary set $\{0, 255\}$, where 255 denotes a change and 0 denotes no change. Because the labeling is binary, minimizing the energy function can be performed efficiently using graph cuts.

Looking at the *change mask generation* execution times, specifically `ImageDifference::buildDifferenceImage`, `Block::buildFrequencyAppearance` and `Vi::AplyMRF` are computationally expensive functions.

3.2.7 Execution time profile of total application

Table 3.7 gives an overview of the total execution time and the relative execution time per function category. Figure 3.1 shows a pie chart of the relative execution times.

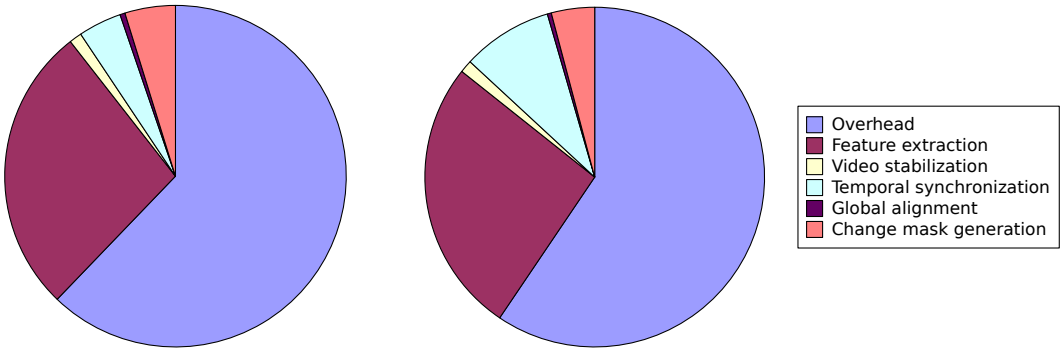
For SD resolution, 1059 ms are required to compute the change detection result. However, to enable real-time change detection a frame rate of 5 Hz is required which implies a maximum execution time of 200 ms per frame. The application must be speeded up by a factor of 5.30 to meet the real-time requirement.
For HD resolution, 13459 ms are needed for every output frame so real-time change detection is infeasible with the current implementation. A 67.3× speedup is required to enable real-time processing.

Looking at the relative execution times, the biggest part of the execution time is spent in *overhead*. For SD resolution 62.2% is spent in this category and for HD resolution 59.5%. The second biggest part of the execution time is spent in *feature extraction*: 27.3% for SD and 26.2% for HD resolution. Because a big part of the time is spent in these categories, it is expected that a significant speedup is possible if the functions of these categories are optimized.

The analysis of the application functions revealed a number of opportunities to speedup the performance. In *overhead*, loading feature data from disk takes a considerable amount of time; approximately half of the application's execution time is spent during this operation. Image conversion is performed twice, because two different function libraries are used. Image conversion should be removed in the end. Besides, redundant frame load operations are performed.
The *feature extraction* algorithm employed is a computationally heavy function, which takes about a quarter of total execution time. Another algorithm should be investigated to see if this computation time can be reduced.
Both in *video stabilization* and *temporal synchronization* image features are used to find correspondences between images. Computation time depends on the number of retrieved features, so it is worth experimenting how many features are needed for a good change detection performance. Besides, the feature matching procedure could be speeded up.
The MRF function in the *change mask generation* category is a computationally heavy function that should be accelerated to enable real-time processing.

Category	t_{fr} (ms)		t_{rel} (%)	
	SD	HD	SD	HD
Overhead	659	8002	62.2	59.5
Feature extraction	289	3531	27.3	26.2
Video stabilization	12.6	160	1.19	1.19
Temporal synchronization	43.5	1160	4.10	8.62
Spatial alignment	5.06	51.0	0.477	0.379
Change mask generation	50.7	554	4.78	4.12
Total	1059	13459	100	100

Table 3.7: Execution time profile of total application per function category.



(a) SD resolution (b) HD resolution

Figure 3.1: Pie chart of the relative execution time per function category.

3.3 Algorithm implementation optimizations

The profile of application functions revealed a number of opportunities to speed up execution by optimizing the implementation of application functions. In this section it is described how application functions were optimized.

3.3.1 Loading previous video frame data

In temporal synchronization 30 previous video frames are loaded for every current video frame, as described in Section 3.2.1. But only the *feature data* of 30 previous video frames is required to compare video frames. Only the best matching previous video frame needs to be loaded to be able to perform spatial alignment. Therefore the application code is adjusted so the index of the best matching previous video frame is saved and after temporal synchronization the frame data of the best matching frame is loaded.

Table 3.8 shows the functions for which execution time reduced after optimization. Now per current video frame only 2 frames are loaded from disk instead of 31. The time spent in `Vi::VideoInputVideoFile::read` reduced by a factor of 12.5 for SD video and 11.8 for HD

Function	t_{fr} (ms)		t'_{fr} (ms)		Δt_{fr} (ms)	
	SD	HD	SD	HD	SD	HD
`Vi::VideoInputVideoFile::read`	28.4	164	2.27	13.91	26.2	150
`Vi::Array2D<unsigned char>::operator=`	0.801	14.6	0.0321	0.949	0.769	13.7

Table 3.8: Speeded up *overhead* functions when only loading matched frame data from disk. Column two states the execution time spent in the function in the initial application version, column three shows the execution time after optimization and column four shows the absolute decrease of execution time.

video. If the execution time of loading a frame would be constant, the reduction factor would be $\frac{31}{2} = 15.5$. So the frame loading time is not constant. The hard drive in the experimental computer system has a 16 MB cache. Presumably part of the frame data can be loaded from cache so the execution time reduces. As a result the time required for loading a frame is not constant.
As a side effect the time spent in `Vi::Array2D<unsigned char>::operator=` reduced. In the initial application version the intermediate best matched frame data was copied into an auxiliary variable. Now this frame data copy is omitted.

3.3.2 Loading feature data

When the overhead functions where analyzed, it turned out that an excessive amount of time is spent in loading previously acquired feature data from disk. Next `Vi::FeatureSIFT::load` will be analyzed in more detail to see how this is caused.
Features are structured as C++ objects in the application. Feature object data consists of:

- (x,y)-coordinate
- scale of the feature
- orientation of the feature
- descriptor length
- descriptor vector

When storing features, the feature objects are serialized first by a Boost library function and subsequently the serialized data is stored into a text file on disk. When features are loaded the text file is read and the serialized data is restored into a feature object. The excessive amount of processing time is spent in serialization of the descriptor vector. During serialization every descriptor element (up to 128) is serialized separately and for every element a unique string is created that entitles the index. Serializing the elements separately is unnecessary, because by the Boost library the descriptor vector can be serialized as a whole.
Another optimization can be made. If the feature data is stored into text files, data must be converted into string representation. It is more efficient to store into a binary file.

Function	t_{fr} (ms)		t'_{fr} (ms)		Δt_{fr} (ms)	
	SD	HD	SD	HD	SD	HD
`Vi::FeatureSIFT::load`	430	5388	11.7	145	418	5242
`boost::archive::detail::interface_iarchive`	132	1769	0	0	132	1769
`std::locale::id::_M_id`	4.77	78.5	0	0	4.77	78.5

Table 3.9: Speeded up *overhead* functions after optimized feature loading

Both optimizations were applied to the application code and the execution time of the functions was measured. The functions that were speeded up after the optimizations are shown in Table 3.9. The optimizations have a big impact on the performance. For SD video the average execution time per frame decreased by 555 ms and for HD video it decreased by 7090 ms. A great amount of *overhead* execution time was saved. Because data is not stored into text format anymore, `boost::archive::detail::interface_iarchive` and `std::locale::id::_M_id` are not executed after optimization.
`Vi::FeatureSIFT::load` decreased to 11.7 ms per frame for SD and to 145 ms per frame for HD video. Still loading feature data of HD video frames takes too long for real-time video processing. In 3.3.3 the effect of a reduced number of retrieved features on the feature loading time will be investigated. Nonetheless, it is worth investigating alternative methods of storing the feature data, e.g. a database.

3.3.3 Feature matching

In both video stabilization and temporal synchronization feature matching is performed. In video stabilization features in two consecutive frames are compared to find the median displacement. The displacement is compensated to correct for abrupt camera motions. In temporal synchronization the features of a currently acquired video are compared to features of a previously acquired video to find matching frames.

In Section 3.2.4 it was shown that matching HD video features takes relatively long compared to matching SD video video features. Especially the relative execution time spent in `Vi::FeatureMatching::process` is much greater for HD video input. When the source code of this function is investigated, it appears that much time is spent in cycling through a list of previous video frame features to access the list element with a specific index number. Because the features are stored in a *list* data structure, random element access is not possible. Storing the features in a *vector* data structure solves this problem, because it does support random access.

To see what the influence of the number of features on the feature matching execution time is, the number of retrieved image features in HD video is varied. To this end the SIFT starting octave is increased. This way the small scale features are not detected and therefore the total number of retrieved features decreases. The resulting feature matching execution time is measured. Table 3.10 states the result of the experiment. In Figure 3.2 the relation between the number of input features and the feature matching execution time is shown. The chart shows a linear relation between the execution time and the number of input features.

starting octave	#features	t_{fr} (ms)
0	2255	36.6
1	954	13.1
2	243	2.80
3	56	0.656

Table 3.10: Execution time required to compare the features of two HD video frames. In the first column the SIFT starting octave is stated. Column two lists the average number of retrieved features in a frame and column three lists the time required to match the features of two consecutive video frames.

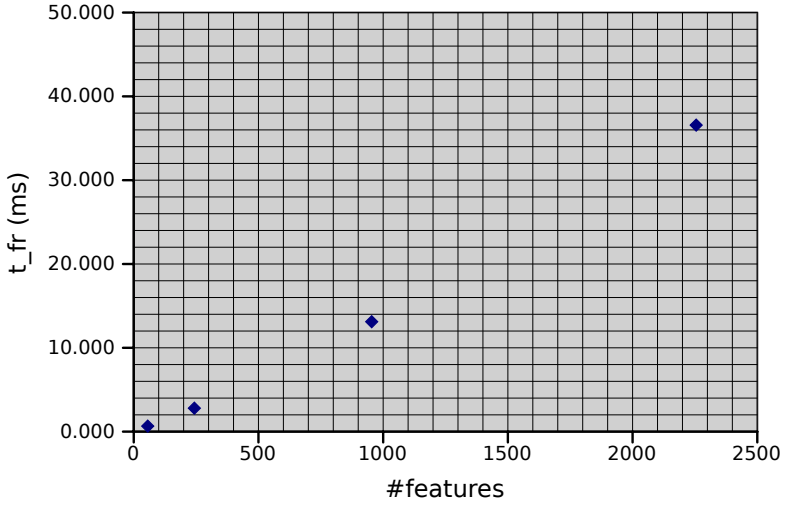


Figure 3.2: The relation of the number of features on feature matching execution time. The horizontal axis shows the number of input features. The vertical axis shows the average execution time required to match the features of two frames (t_{fr}).

The change detection application will be used in an outdoor environment. Outdoors many similar small-scale objects occur, e.g. small rocks, grass and leaves. Small-scale features will be detected in these objects' regions. But in the context of change detection, we are not interested in these features because it is highly likely that these objects will change in the period between acquisitions. Therefore the SIFT `starting octave` is increased to prevent detection of these small-scale features. After experiments on a number of test videos, we found out that for HD video an increase of the `starting octave` to 1 results in an optimal change detection quality.

The feature matching optimizations described before were adopted in the change detection application. The functions in which less execution time is spent after the optimizations, are listed in Table 3.11. Replacing the *list* data structure by a *vector* data structure reduces the time spent in `Vi::FeatureMatching::process` significantly. In *temporal synchronization* the time in this function decreased by 2.10 ms per frame for SD video and by 457 ms per frame for HD video.

Category	Function	t_{fr} (ms)		t'_{fr} (ms)		Δt_{fr} (ms)	
		SD	HD	SD	HD	SD	HD
Overhead	`Vi::FeatureSIFT::load`	11.7	145	11.7	80.6	-	64.4
Temporal sync.	`Vi::FeatureMatching::process`	4.25	480	2.15	22.5	2.10	457
	`cv::flann::Index`	8.46	114	8.46	69.4	-	44.7
	`cv::flann::Index::knnSearch`	30.8	566	30.8	169	-	397
Video stabilization	`Vi::FeatureMatching::process`	3.46	77.5	3.04	13.7	0.42	63.8
	`cv::findHomography`	1.92	7.97	1.92	3.91	-	4.07

Table 3.11: Speeded up functions after feature matching optimization

The reduction of the number of retrieved features in HD video impacts both functions in *overhead*, *temporal synchronization* and *video stabilization*. In *overhead* `Vi::FeatureSIFT::load` takes less time because a smaller number of features are loaded from disk. In *temporal synchronization* both the time for building the feature matching tree (`cv::flann::Index`) and traversing the tree (`cv::flann::Index::knnSearch`) reduces significantly. The reduced number of features influences the time spent in *video stabilization* as well. The total feature matching time (`Vi::FeatureMatching::process`) reduces from 77.5 ms to 13.7 ms. The effect of the replacement of the *list* data structure is included. The time needed to compute the homography matrix reduced from 7.97 ms to 3.91 ms.

3.3.4 Adaptive thresholding

In *change mask generation* an adaptive block-thresholding algorithm generates a binary change mask by thresholding the difference image [SA06]. This algorithm requires a histogram of every image block. Histograms are computed in the `Block::buildFrequencyAppearance` function. In the initial application version the histograms were calculated by the following procedure.
The histogram is stored in a 2D-array, called `mFrequencyAppearance`. In each row of the array a vector of two elements is stored. Each vector holds a gray-level and the number of occurrences. For each pixel of a block it is checked if a vector with the pixel gray-level already exists in `mFrequencyAppearance`. If it exists the number of occurrences is increased by 1. If it does not exist, a new vector is created and the number of occurrences is set to 1.
The histogram can be computed much more efficiently. The histogram can be stored in a simple 1-D array. The number of gray-levels is equal to 256. Because the number of gray-levels is known, an array of a size equal to the number of gray-levels can be allocated up front. Assume that the index of the array is equal to the gray-level values. Then only the number of occurrences needs to be stored in the array elements. Before processing the pixels, all elements are initialized to 0. For every block pixel, simply increase the array element g_i by 1, where g_i is equal to the gray-level of the pixel.

Another optimization can be performed, in `AdaptiveThresholding::thresholdImage`. In this function image pixels in the output image are set to either 0 or 255 by thresholding. Before thresholding the output image is initialized first by setting all pixels to 0. This is unnecessary, because all pixels are set later by the thresholding procedure. Therefore the

Function	t_{fr} (ms)		t'_{fr} (ms)		Δt_{fr} (ms)	
	SD	HD	SD	HD	SD	HD
`AdaptiveThresholding::process`	2.76	19.0	2.78	18.6	-0.03	0.4
`Block::buildFrequencyAppearance`	5.16	75.6	0.321	2.75	4.84	72.8
`AdaptiveThresholding::thresholdImage`	1.76	21.0	0.385	5.79	1.38	15.2

Table 3.12: Speeded up *change mask generation* functions after adaptive thresholding optimizations

initialization can be omitted.

The result of the optimizations is stated in Table 3.12. Both the performance of `Block::buildFrequencyAppearance` and `Adaptivethresholding::thresholdImage` was improved significantly. The procedure in `Block::buildFrequencyAppearance` was simplified. Instead of a double for-loop with a number of if-statements, now only one for-loop without any if-statements is needed. Many branches are saved, resulting in much better performance. 4.84 ms per frame was saved for SD video and 72.8 ms for HD video.
In `AdaptiveThresholding::thresholdImage` a large part of the original execution time was saved, because initialization was omitted. Therefore traversing all image pixels and setting them to 0 is not needed anymore.

3.4 Algorithm modifications: efficient feature computation

Currently the Scale-Invariant Feature Transform (SIFT) algorithm is employed for feature detection and description. Because the execution time needed to extract SIFT features is a bottleneck for real-time change detection, a more computationally efficient feature detector and feature descriptor algorithm were investigated.

3.4.1 Alternative local feature detector

A local feature is an image pattern that differs from its immediate neighborhood. A *feature detector* finds the position and size of features in an image.

In the SIFT algorithm, features are detected by the following procedure [Low04]. First a scale-space pyramid representation of the image is created by smoothing the image multiple times by convolution with a Gaussian kernel. A higher degree of blurring resembles a higher scale. Features are detected by finding local intensity minima and maxima of the Laplacian of the scale-space. The Laplacian of the Gaussian operator is approximated in terms of the Difference-of-Gaussians function. Differences of Gaussians are computed by subtracting pairs of smoothed images.

STAR detector

An alternative feature detector is the *STAR* feature detection algorithm. STAR is based on the Center Surround Extremas algorithm (CenSurE), introduced in [AK08]. Like SIFT, image feature are detected based on extrema of the Laplacian across scale and location. Instead of

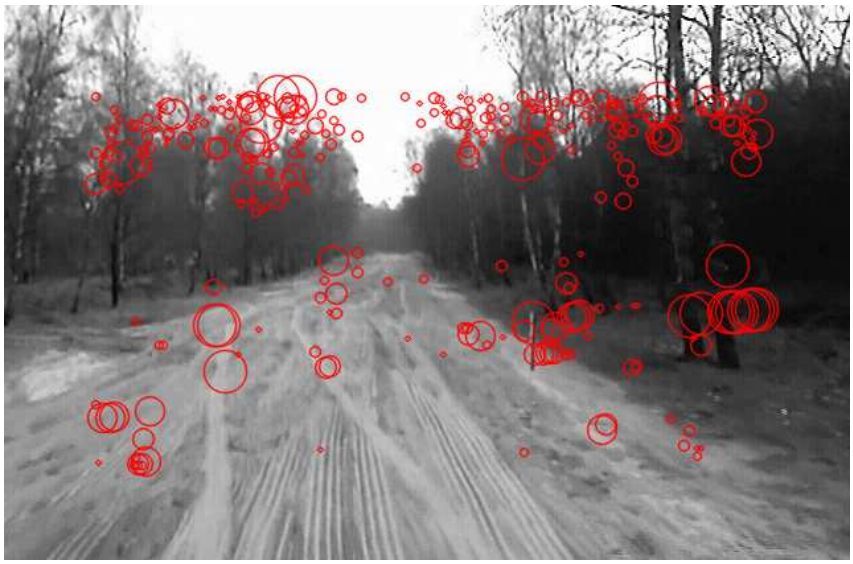


Figure 3.3: Detected STAR features in off-road recording at SD resolution.

the Difference-of-Gaussians approximation of SIFT, the Laplacian is estimated by a simple bi-level center-surround filter. In bi-level filtering image values are multiplied by either 1 or -1. In STAR the bi-level filter has an 8-end STAR-shaped. The STAR-shape is a trade-off between repeatability and computational efficiency. It provides better symmetry than simple boxes, while the filter response can be computed efficiently by integral images. To find extrema across scale, the response of multiple bi-level filter sizes is computed. In STAR filter responses are computed at full image resolution, without downscaling the image.

The detected features in one SD video frame are shown in Figure 3.3. A disadvantage of the OpenCV STAR implementation is that no features are detected in the border area of the image. The border area is omitted, because a minimum area is required to compute the response of the maximum filter size. Before it can be applied to the change detection system, detection of features in the borders should be added.

SURF detector

A second alternative is the Speeded-Up Robust Features (SURF) feature detection algorithm. SURF is an computationally efficient algorithm that detects scale- and rotation-invariant feature points. Before detection an integral image is computed, that allows computation of the sum of pixels in a rectangular image area in $O(1)$ time. Feature detection is performed by computation of the determinant of the Hessian matrix. Second order Gaussian derivatives of the Hessian matrix are approximated by the use of box filters that can be evaluated efficiently using the integral image. Box filter responses are evaluated with different filter kernel sizes, to be able to detect features at multiple scales. After responses are computed for all scales, local minima and maxima over scale and space are detected.

Performance comparison

In [BJJ+11] the SIFT, SURF and STAR feature detection algorithms were compared by testing the repeatability of the detected features in two outside recorded videos. A feature is considered repeatable if it is detected in both the current frame and the subsequent frame.

It was concluded that both STAR and SURF have a better repeatability compared to SIFT in the change detection test sequences. STAR shows a repeatability of 64% for the on-road video and 75% for the off-road video. The repeatability of SURF is slightly better, it is equal to 72% for the on-road video and 78% for the off-road one. The repeatability of SIFT is only 50% for the on-road video and 52% for the off-road one.

3.4.2 Alternative local feature descriptor

After features have been detected, a local feature descriptor describes the salient region around the keypoint. It describes the region by creating a list of properties of the neighborhood.

In the SIFT descriptor algorithm, descriptors are computed for every detected feature in the following way. First the gradient magnitude and orientation of pixels in the region around the feature is calculated. Next the gradients are accumulated into an orientation histogram, covering the 360 degree range of orientations. For rotation invariance, the coordinates of the descriptor and the gradient orientations are rotated relative to the dominant gradient orientation. The descriptor vector consists of all the values of all the orientation histogram entries. The standard descriptor configuration is a 4x4 8-bin histogram array, where each histogram covers a 4x4 px subregion. This results in a 128-dimensional descriptor vector.

SURF descriptor

A more efficient feature descriptor is the SURF descriptor algorithm. The feature descriptor computation starts by assigning a reproducible orientation to every feature point to provide rotation invariance. To this end, Haar wavelet responses are computed in the x- and y-direction within the neighborhood of the feature point. Both the size of the neighborhood, the sampling step and the size of the wavelets depend on the scale of the feature. Based on the wavelet responses, the dominant orientation is estimated by calculating the sum of all responses within a sliding orientation window.
Next the descriptor is calculated in a square region centered around the interest point and oriented along the previously assigned orientation. The method is illustrated in Figure 3.4. The region is split-up into smaller 4x4 grid of square sub-regions and for every sub-region the Haar wavelet response d_x in the x-direction and d_y in the y-direction are computed. The descriptor vector consists of the sums $\sum d_x, \sum dy, \sum |dx|, \sum |dy|$ of every sub-region, yielding a 64-dimensional vector. If the sums are computed separately for $d_y < 0$ and $d_y > 0$, an extended 128-dimensional vector results that is more distinctive.

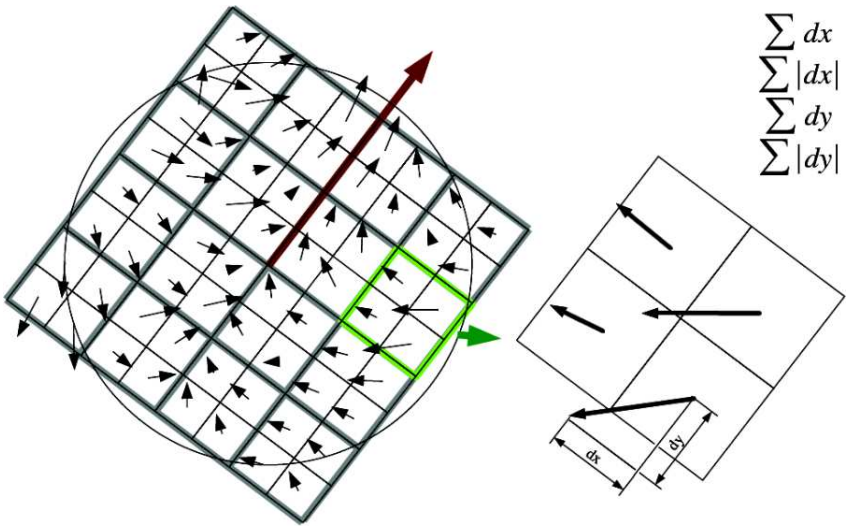


Figure 3.4: Illustration of SURF feature description method. The region grid around the feature point is created. It is split up in 4x4 sub-regions and for every region Haar wavelet responses d_x and d_y are computed. Detected STAR features in off-road recording at SD resolution. The 2x2 sub-division of a sub-region corresponds to the fields of the descriptor.

3.4.3 Computational performance of OpenCV STAR and SURF functions

The execution time of the OpenCV STAR detector and the OpenCV SURF detector/descriptor was measured for both SD and HD resolution. The STAR parameters were set to $\{45, 15, 10, 8, 5\}$ for SD video and to $\{90, 30, 10, 8, 5\}$ for HD video. The SURF detector parameters were set to $\{550, 4, 2, true, false\}$ for both SD and HD resolution. The SURF descriptor length was so set to 128, equal to the standard SIFT descriptor length. For both STAR and SURF multithreading was enabled.

The performance of STAR is stated in Table 3.13 and the performance of SURF in Table 3.14. The execution time of the STAR detector is smaller than the execution time needed for the SURF detector, for both SD and HD resolution. However, image borders are skipped in the OpenCV STAR feature detector while OpenCV SURF does detect features in borders. Therefore the execution time of STAR would be bigger if borders are included. Still, the STAR detector will have a better computational performance, because the borders cover only a minor part of the image area. SURF feature description only takes 2.48 ms per frame for SD resolution and 8.96 ms for HD resolution.

A combination of the STAR detector and SURF descriptor will result in the best computational performance. The time needed for feature extraction using these algorithms is equal to 13.2 ms for SD and 117 for HD resolution. When the SIFT algorithm is replaced by the STAR/SURF combination, the execution time decreases by an order of magnitude, by 22× for SD and 30× for HD.

Function	t_{fr} (ms)	
	SD	HD
`cv::StarDetector`	10.7	108

Table 3.13: Average execution time per frame of OpenCV STAR feature detector

Function	t_{fr} (ms)	
	SD	HD
`cv::SurfDetector`	23.7	263
`cv::SurfDescriptor`	2.48	8.96
Total	26.2	272

Table 3.14: Average execution time per frame of OpenCV SURF feature detector and descriptor

3.5 Further optimizations to be implemented

The optimization reported in Section 3.3 and 3.4 were implemented and tested. In this section two more optimizations are described that were not implemented, but are expected to speed up the application significantly.

3.5.1 Loading feature data

In the *overhead* category `Vi::FeatureSIFT::load` loads feature data of 30 previous video frames from disk for every current video frame. The feature data serves as input to the temporal synchronization. In the optimized application 11.7/145 ms is spent in reading feature data for SD/HD video. Reading data of 30 previous video frames for every current video frame is unnecessary, because the previous video frames overlap. This means that feature data of every frame of the previous video is read 30 times in the worst case.

If a buffer is created in main memory, feature data that has been read from disk before does not have to be read again. Only data of new frames needed for temporal synchronization has to be read from disk. In the best case only data of one previous video frame is read for every current video frame.
A *circular buffer* is an efficient way of storing data in main memory in a First In, First Out (FIFO) fashion. Memory is only allocated when the buffer is created or resized explicitly. When the buffer is full, the oldest data is overwritten by new data. It is an efficient data structure because no time-consuming memory reallocation is needed.
In Figure 3.5 the use of the circular buffer is illustrated. Three pointers are used for buffer administration. `begin_ptr` points to the beginning of the buffer: at this position the oldest data resides. `end_ptr` points to the position where new data elements are written into the buffer. The third pointer, `read_ptr`, points to the buffer position from which the previous video feature data is read. The figure illustrates the buffer state where the buffer is not yet full. It is filled with consecutive previous video data from `begin_ptr` to `end_ptr`. For temporal synchronization 30 frames from the previous video are needed. The green area contains the

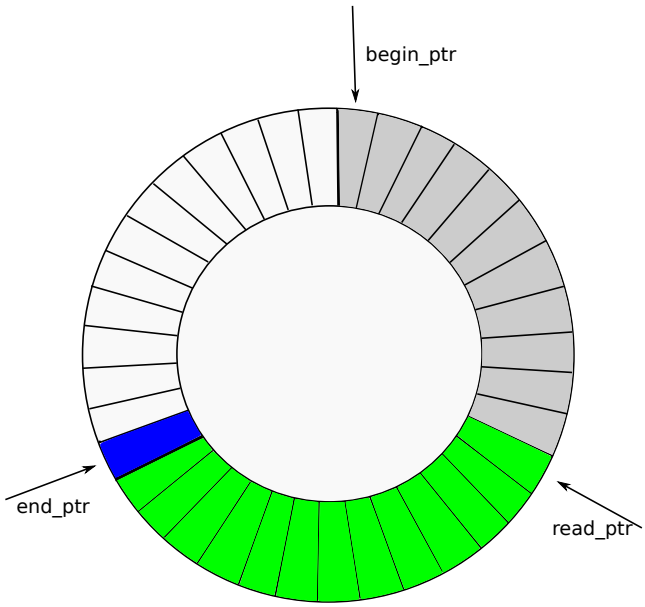


Figure 3.5: Circular frame buffer. Green area: data is directly read from the buffer. Blue area: data is not present in the buffer. Data is loaded from disk and stored into the buffer before it can be accessed.

part of the required feature data that was stored into the buffer before. It is read from the buffer and therefore does not have to be loaded from disk. The data that is not in the buffer is loaded from disk and stored in the buffer at `end_ptr`, indicated by the blue area.

If the circular buffer is implemented, it is expected that `Vi::FeatureSIFT::load` will be speeded up by 30×, because only the feature data of 1 previous frame instead of 30 frames needs to be read from disk for every current frame.

3.5.2 Image conversion

Video frames are converted to OpenCV image format twice in the application, because OpenCV library functions cannot be directly applied to the ViNotion image format. This conversion takes quite much processing time. Therefore the change detection application needs to be adjusted such that all functions work with ViNotion image format. If only one image format is used, no execution time is spent in conversions anymore.

3.5.3 Expected speedup

Table 3.15 states the expected execution time per frame $\widehat{t_{fr}}$ of *overhead* functions with respect to the optimized application version reported in Section 3.3 (t'_{fr}). The time spent in overhead will be reduced by 52.4/474 ms for SD/HD resolution. As a result, the total change detection execution time is expected to decrease to 138/1234 ms for SD/HD video.

Function	t'_{fr} (ms)		$\widehat{t_{fr}}$ (ms)		Δt_{fr} (ms)	
	SD	HD	SD	HD	SD	HD
`Vi::FeatureSIFT::load`	11.7	80.6	0.390	2.69	11.3	77.9
`ChangeDetection::Vi2OpenCV`	9.57	93.9	0	0	9.57	93.9
`ChangeDetection::OpenCV2Vi`	7.14	67.4	0	0	7.14	67.4
`Vi::ImageStabilizationCIED::Vi2OpenCV`	15.6	148	0	0	15.6	148
`Vi::ImageStabilizationCIED::OpenCV2Vi`	8.83	86.4	0	0	8.83	86.4

Table 3.15: Expected speed up of *overhead* functions after applying circular buffer and removing image conversion.

Category	t_{fr} (ms)		$\widehat{t_{fr}}$ (ms)		$\widehat{S}$	
	SD	HD	SD	HD	SD	HD
Overhead	661	8002	23.7	212	27.8	37.8
Feature extraction	289	3531	13.2	117	21.9	30.1
Video stabilization	12.6	161	12.1	92.6	1.03	1.73
Temporal synchronization	43.5	1160	39.2	261	1.11	4.44
Spatial alignment	5.06	51.0	5.06	51.0	1.00	1.00
Change mask generation	50.5	554	44.5	466	1.14	1.19
Total	1089	13459	138	1199	7.69	11.2

Table 3.16: Application profile after performance improvements. Column two lists the average execution time per frame in the initial application version, column three lists the expected average execution time per frame after improvements and column four shows the expected relative speedup.

3.6 Execution time profile of speeded up application

Table 3.16 states the execution time per function category of the speeded up application. The speedup includes the speedup of implementation optimizations described in Section 3.3, the alternative feature detection algorithm described in Section 3.4 and the estimated speedup after the optimizations described in Section 3.5 have been implemented.

The total change detection execution time was significantly reduced. The execution time was speeded up by a factor of 7.69 for SD resolution and a factor of 11.2 for HD resolution. Performance was mainly improved by modifications in the *overhead* and *feature extraction* category. In these categories most execution time was spent relatively in the initial application version. The time spent in *overhead* functions was reduced by a factor of 27.8×/37.8× for SD/HD resolution, due to a drastic decrease of time spent in feature and video frame data from disk. If the SIFT feature extraction algorithm is replaced by a combination of the computationally efficient STAR feature detector and SURF feature descriptor, the time spent in feature extraction is speeded up 21.9×/30.1× for SD/HD resolution.

The total time required to perform change detection on an SD resolution frame, decreased to 138 ms. This implies that SD change detection can be performed at a frame rate of 7.25 Hz. A minimum frame rate of 5 Hz is required to enable real-time processing, which is feasible after the aforementioned performance improvements.
For HD resolution, the execution time time per frame decreased to 1199 ms. This implies that real-time processing is not feasible after these performance improvements.

3.7 Conclusions

An execution time profile of the application functions was created to see if real-time processing is feasible and to investigate in which parts of the application most time is spent. In the initial application version, in total 1089/13459 ms was needed to process a frame of SD/HD resolution on the CPU of the experimental platform. Therefore, the frame rate of 5 Hz required for real-time processing was infeasible for either SD or HD resolution.
The profile showed that more than half of the total execution time was spent in *overhead* functions and that approximately a quarter of the total time was spent in *feature extraction*.

A detailed analysis of the implementation of application functions showed opportunities to speed up the performance. In *overhead* more than half of the total change detection processing time was spent in loading previously acquired feature data from disk. We found out that during object serialization, every descriptor element was serialized separately. Moreover, feature data was stored into text format which implies numbers must be converted into strings. By serializing the descriptor at once and storing feature data into binary format, the time needed to load feature data was decreased by 48.4× for SD and by 49.9× for HD video. Besides, we found out that part of the previous video frame loads were redundant. These were removed.
In *temporal synchronization* a *list* data structure was replaced by a *vector*, because the list was iterated for random access. Opposed to a *list*, a *vector* does support random access and therefore no iteration is required.

By reduction of the number of retrieved features for HD resolution video, the time spent in *overhead*, *temporal synchronization* as well as *video stabilization* was reduced.
In *change mask generation* the performance of the adaptive thresholding algorithm was increased by simplifying the data structure employed during histogram computation.
Two optimizations opportunities were found that have not been implemented yet. Introduction of a circular buffer will further reduce the time needed for feature data loading and when one instead of two image formats are used, image conversions can be removed.

To reduce the time spent in *feature extraction*, an alternative feature extraction algorithm was investigated. A combination of the OpenCV STAR detector and OpenCV SURF descriptor resulted in a reduction of execution time by 21.9×/30.1× for SD/HD video, while the feature repeatability increased.

Due to the described performance optimizations, the total execution time required per frame reduces from 1089 to 138 ms for SD resolution and from 13459 to 1199 ms for HD resolution. This implies the performance was increased by 7.9× and 11× for SD/HD resolution. As a result of the described performance improvements, real-time SD change detection on one 1 CPU is enabled, as 138 ms per frame implies a frame rate of 7.25 Hz. For HD resolution real-time change detection is not feasible on 1 CPU after the described performance improvements.

Performance was increased considerably, but can be further increased by applying multithreading and SIMD instructions. In the OpenCV STAR detector and SURF descriptor multithreading and SIMD instructions are already applied, but they are not in the other application functions. However, it depends on the structure of an algorithm if it is possible to apply multithreading and SIMD instructions without algorithmic modifications.
Because of the extent in which execution time must be decreased to enable real-time HD change detection, we focus on GPU implementations instead of further CPU optimizations.

CHAPTER 4

Accelerating STAR by GPU integral image computation

In the STAR feature detector algorithm, presented in Section 3.4, integral images are employed for efficient computation of the bi-level filter response. The algorithm employs a STAR-shaped filter kernel, consisting of both an upright and a 45° tilted (rotated) square. The sum of pixels in the filter area can be quickly computed using the *upright* and *tilted* integral image.

An efficient GPU implementation of upright and tilted integral image computation was created that can be used to accelerate STAR feature detection. This implementation is a first step towards a full STAR GPU implementation.

4.1 Upright integral image

In 1984 *summed-area tables* were introduced in the field of computer graphics [Cro84]. Viola and Jones first introduced the summed-area table concept in image analysis to speed up object detection [VJ01]. They renamed the concept to *integral image*.

An integral image is an intermediate representation of an image that allows very efficient computation of the sum of pixels in an upright rectangular area. Equation (4.1) is used to calculate the integral image $I(x, y)$, where $i(x, y)$ is the input image. The value of $I(x, y)$ is equal to the sum of all input image pixel values to the left and above (x, y). This is illustrated in Figure 4.1.

$$I(x, y) = \sum_{x' \leq x, y' \leq y} i(x', y') \tag{4.1}$$

To compute the sum of a rectangular area of an image, only four integral image lookups are needed. This is illustrated in Figure 4.2. The sum of pixels in area D can be computed by $I_4 - I_2 - I_3 + I_1$, where I_l is the integral image value at location l.

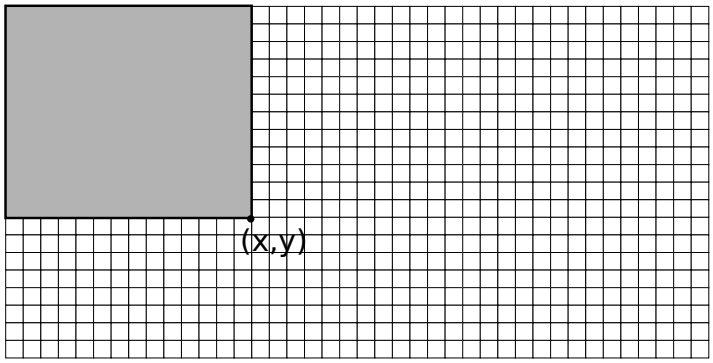


Figure 4.1: The integral image value at position (x, y) is equal to the sum of all pixel values to left and above (x, y)

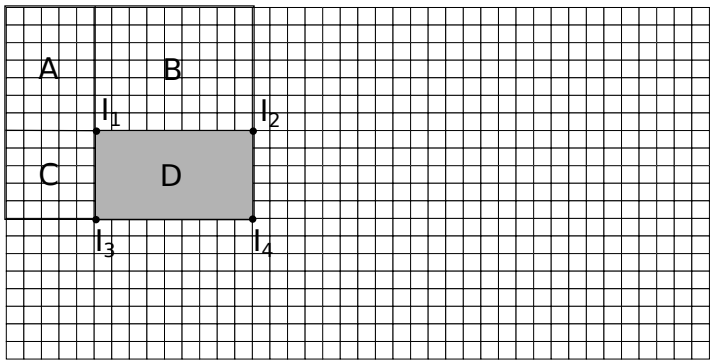


Figure 4.2: The sum of pixels in area D of the original image can be computed with integral image values by $I_4 - I_2 - I_3 + I_1$.

4.2 Tilted integral image

To compute the tilted integral image, Equation (4.2) is evaluated. The tilted integral value at position (x, y) is equal to the sum of pixel values in a rectangular area above (x, y) that is rotated clockwise by 45°.

$$T(x, y) = \sum_{y'<y, |x'-x+1| \leq y-y'-1} i(x', y') \tag{4.2}$$

After the tilted integral image has been computed, the sum of pixels in a tilted rectangular area can be quickly computed. The sum of pixels can be computed with four tilted integral image values, like the upright integral images. This is illustrated in Figure 4.3.

4.3 Parallel computation

Data-parallel algorithms are needed to compute the upright and integral images efficiently on the GPU. The GPU is fully utilized only if abundant parallelism is available. First the *parallel prefix sum* algorithm, that is employed in calculation of both the upright and tilted integral

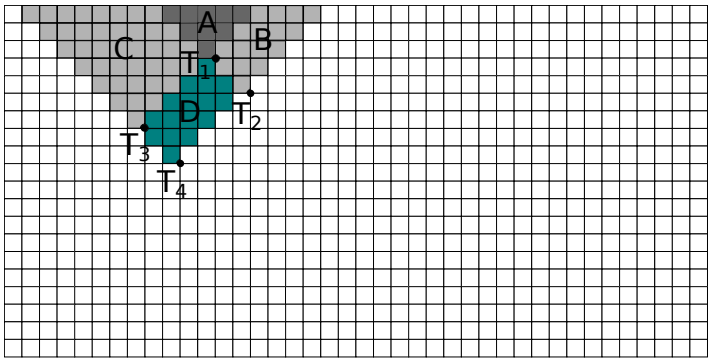


Figure 4.3: Like the upright integral image, the sum of pixels in area D of the original image can be computed with tilted integral image values by $T_4 - T_2 - T_3 + T_1$.

image, will be presented. Next parallel algorithms are stated that calculate the upright and tilted integral image.

4.3.1 Parallel prefix sum

The prefix sum, or *scan*, takes a binary associative operator $\oplus$ and an array of n elements and returns the array

$$[a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1})] \tag{4.3}$$

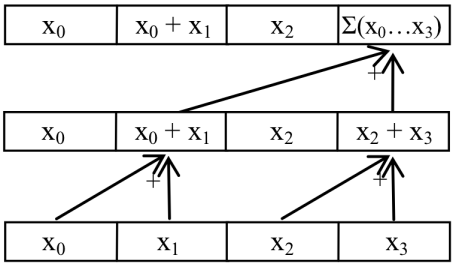
If $\oplus$ is addition, the *inclusive scan* operation is obtained. If the resulting array is shifted to the right by one element and the identity is inserted at the beginning, *exclusive scan* results, which returns

$$[0, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})] \tag{4.4}$$

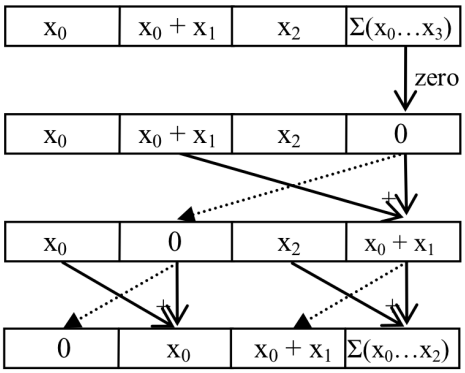
The exclusive scan can be efficiently computed in parallel by the *parallel prefix sum* algorithm presented in [HSO07]. The algorithm uses the *balanced binary tree* pattern to compute the exclusive scan in $O(n)$ time. The parallel prefix sum performs computations on an array in-place. No tree data structure is built, the tree branches correspond to the position of the array elements that are processed. The algorithm operates in two phases: the *reduce* phase and the *down-sweep* phase.

Figure 4.4 shows how the two phases of the algorithm are executed. In the *reduce* phase the tree is traversed from leaves to root while the partial sum at each tree node is computed. In the *down-sweep* phase the tree is traversed back from root to leaves. The partial sums are used to build the exclusive scan result in-place. In the first step of this phase the last element is set to zero, because the total sum is not included in the exclusive scan result. At each level of the tree, add and swap operations are computed in parallel.

Because of the balanced binary tree structure required for work-efficient computation, the algorithm requires the size of an input array to be a power of two. The parallel prefix sum algorithm can be applied on arrays with a size not equal to a power of two, if the array size is increased to a power of two and the new elements are set to zero (zero-padding). This implies



(a) In the reduce phase the tree is traversed from leaves to root, while the sum of two array elements is computed at every tree node.



(b) In the down-sweep phase the tree is traversed from root to leaves. First, the last element is set to zero because of the *exclusive* scan. At each tree node, two elements are swapped and the sum of the elements is computed. Finally, the leaves of the tree contain the exclusive scan.

Figure 4.4: The two phases of the parallel prefix sum algorithm, illustrated for a 4-dimensional array. The algorithm uses a binary tree pattern to compute the result in $O(n)$ time. At each level of the tree, sum and swap operations are computed in parallel.

that extra work is performed on the zero-padded part of the array.
The amount of extra work can be reduced by splitting the input array in a set of arrays blocks with sizes equal to a power of two. First the exclusive scan algorithm is performed on every array block separately. Next the full scan can be obtained in the following way. To be able to compute the full integral, the sum of all elements of every array block is stored into an auxiliary array I_{sum}. Afterwards, the exclusive scan of I_{sum} is calculated and $I_{sum}[i]$ is added to array block $i + 1$. This procedure is explained in Figure 4.5.

4.3.2 Upright integral image

As described in [BHM10], the upright integral image can be computed in $O(n)$ time with the parallel prefix sum algorithm and transposition of the image. The upright integral image is computed in the following steps:

1. Parallel prefix sum of every image row
2. Transpose image
3. Parallel prefix sum of every transposed image row
4. Transpose image

First the exclusive scan of all image rows is computed by the parallel prefix sum algorithm. Image rows can be processed concurrently, because rows are independent. That means there are two dimensions of parallelism: parallelism inside an image row parallel prefix sum and

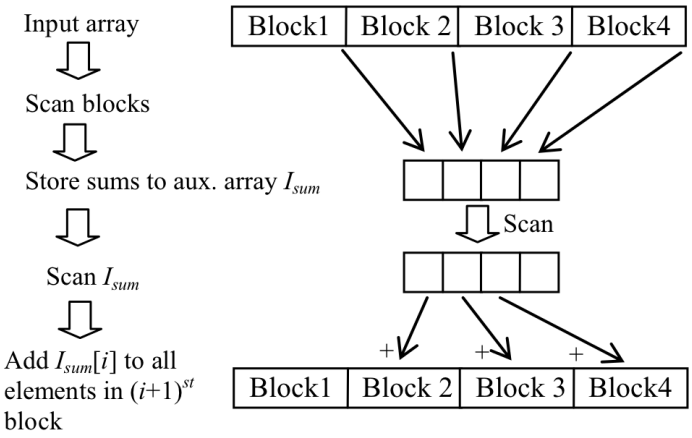


Figure 4.5: Example of exclusive scan on an input array that is first split into four blocks.

parallelism over image rows. Second, the image is transposed and third the exclusive scan of the transposed image rows is computed. The transposed image rows are equal to the original image columns. Finally, the upright integral image is obtained after the image is transposed a second time.

4.3.3 Tilted integral image

The tilted integral image is the 45° rotated version of upright integral image. But calculation is not as simple as calculation of the upright integral image. It would be straightforward to rotate the scan direction of the upright integral image by 45°, obtaining the scan of image diagonals. If a diagonal scan is performed from left-top to right-bottom and subsequently from right-top to left-bottom, the result is not equal to the tilted integral image, as illustrated in Figure 4.6.
The result $D(x, y)$ consists of array elements that contain the sum of odd diagonals and elements that contain the sum of even diagonals, in a checkerboard pattern. Because of this checkerboard pattern, the tilted integral image value at this position can be calculated by

$$T(x, y) = D(x, y) + D(x, y - 1). \tag{4.5}$$

However, Equation (4.5) does not hold for tilted integral values in a triangular area in the right-bottom of the tilted integral image, as illustrated in Figure 4.7a. Values are incorrect below the dashed line. In this area $T(x, y)$ excludes the sum of pixel values of a triangular area in the right-top of the input image. These pixel values are excluded, because they are not propagated in the diagonal scan in right to left direction.
To solve this problem, the sum of pixel values in the right-top triangular area are added. After the top-left to right-bottom diagonal scan, this sum can be quickly computed by performing a vertical exclusive scan of the last column. This is shown in Figure 4.7b, that illustrates the missing area of Figure 4.7a.

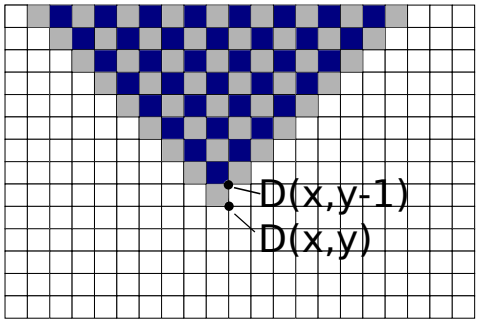
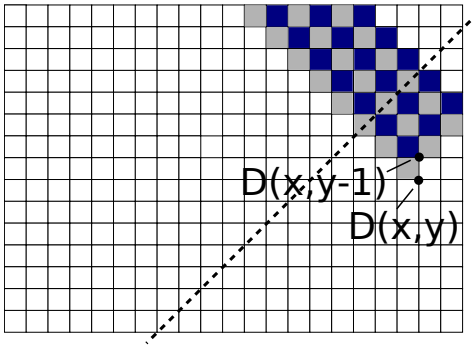
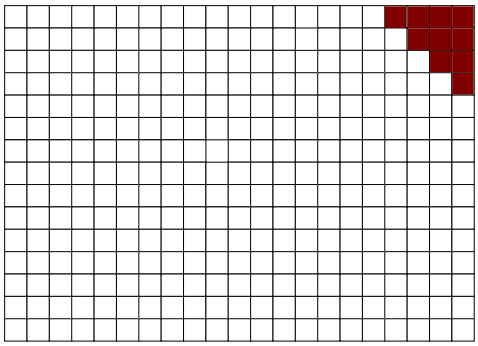


Figure 4.6: Result of a diagonal scan from left-top to right-bottom and next a diagonal scan from right-top to left-bottom. $D(x, y)$ is equal to the sum of pixels in odd diagonals in the tilted rectangle above (x, y). $D(x, y-1)$ equals the sum of pixels of even diagonals in the tilted rectangle. $T(x, y)$ is equal to the sum of $D(x, y)$ and $D(x, y-1)$.



(a) Result of diagonal scan in two directions at the right border of the image. $D(x, y) + D(x, y-1)$ excludes the pixels in the right-top triangular area.



(b) The missing triangular area can be computed by a vertical exclusive scan of the last column, after the left to right diagonal scan.

Figure 4.7: To calculate the tilted integral image values below the dashed line, the missing sum of pixels in the right-top triangular area must be added.

When the previously described calculation are combined, the full tilted integral image can be calculated in the following 5 steps:

1. Exclusive diagonal scan in left-top to right-bottom direction.
2. Exclusive scan of last column, save the result to an auxiliary array.
3. Inclusive diagonal scan in right-top to left-bottom direction.
4. Add to all elements the pixel value above.
5. Add the missing right-top triangle area.

Note the *inclusive scan* in step 3. An inclusive instead of an exclusive scan is performed, because in step 1 the first elements of rows and columns of the tilted integral image are already set to zero. Another exclusive scan would result in loss of a row of the tilted integral

image. The inclusive scan can be computed by a minor modification of down-sweep phase the parallel prefix sum algorithm.

4.4 GPU implementation

The parallel upright and tilted integral image algorithms were implemented in CUDA and tested on an NVIDIA GTX 480 GPU. The algorithms were optimized for an image resolution of 1920x1080 px. However, the implementation can process images up to a resolution of 4096x4096px if the thread block sizes are adjusted.

4.4.1 Upright integral image

The upright integral image algorithm was implemented in the following way.

For computation of the parallel prefix sum of a 1-D array, the implementation of [HSO07] was adopted. The computation on the 1-D array is performed by the threads in one thread block. Each thread computes the partial sum of two array elements. The parallel prefix sum is performed in shared memory and shared memory bank conflicts are avoided by padding the input arrays. i/n_{banks} padding is added to the shared memory indices, where i is the index number and n_banks is the number of shared memory banks. This requires that the size of shared memory array should be increased to $width + width/n_{banks}$ for the horizontal scan and $height + height/n_{banks}$ for the vertical scan.
For every image row/column the exclusive scan is computed independently. The algorithm requires input arrays that have a size equal to a power of two. For efficiency, rows/columns are split into two parts and each part is processed by one a separate CUDA kernel. The first kernel computes the regular exclusive scan. The second kernel computes the exclusive scan of the second row/column part and adds the total sum of the first row/column part to all elements. This procedure was explained in Section 4.3.1.
We founded out that the GPU performs best if the thread block size is equal to *512*. That means *1024* elements are scanned per thread block. 1920x1080px image rows are processed by two blocks of 512 threads. Image columns are processed by one block of 512 threads and one block of 32 threads.

The transpose of the image is computed using the CUDA SDK's transpose kernel that provides coalesced device memory access and avoids bank conflicts. The image is transposed in 16x16 thread blocks. Memory access is coalesced because the image data is divided into 16x16 blocks and transferred to shared memory before performing the actual transposition. Uncoalesced access to shared memory does not hurt performance as long as there are no memory bank conflicts. Bank conflicts are avoided by increasing the width of shared memory data blocks by 1. This prevents threads in a warp to try to access the same memory bank when storing block columns.

4.4.2 Tilted integral image

For computation of the tilted integral image, the parallel prefix sum code of the upright integral image implementation is reused. In the GPU implementation the tilted integral image is computed as follows.

1. To compute the exclusive scan of diagonals in left-top to right-bottom direction, image diagonals instead of rows/columns are passed to the parallel prefix sum function. The results are computed in parallel, where each thread block processes a diagonal.

2. After the first step, the parallel prefix sum function is used to compute the exclusive scan of the last column of the result of the first step. The result is stored into an auxiliary array.

3. The inclusive scan of right-top to left-bottom diagonals is computed by passing the diagonal arrays to the parallel inclusive scan function.

4. Next to every array element the array element directly above is added. The operation is performed by 16x16 thread blocks, in device memory.

5. Finally, the result of step 2 of the algorithm is added to the result of step 4.

Image diagonals have varying sizes. However, the parallel prefix sum algorithm requires input arrays with a size equal to a power of two. Moreover, CUDA thread blocks have constant size within a kernel function. This implies that the diagonal lengths must be increased to the size of the longest image diagonal, rounded up to a power of two. This means that a huge amount of redundant computations are performed.
Most of the redundant computations are saved by skipping threads that process data that falls outside the image. An extra variable is passed to the parallel prefix sum algorithm that is equal to the size of the diagonal, rounded up to a power of two. The threads that process diagonal array elements with an index greater than the variable value are skipped, as shown in Figure 4.8.

The diagonal scan of a 1920x1080 image is computed in two parts. First the diagonal scan of the upper part of the image is computed. Thread blocks have 512 threads for optimal performance, so up to 1024 elements are processed. Next the diagonal scan of the lower part is computed by thread blocks of 32 threads and the sum of all upper diagonal elements is added.

4.5 Performance evaluation

The GPU implementation of the upright and tilted integral is evaluated by comparing the execution time to the execution time of an optimized CPU implementation for 1920x1080 px images. The CPU implementation is the integral image function of the OpenCV STAR detector. In this function both the upright and tilted integral are computed in the same loop.

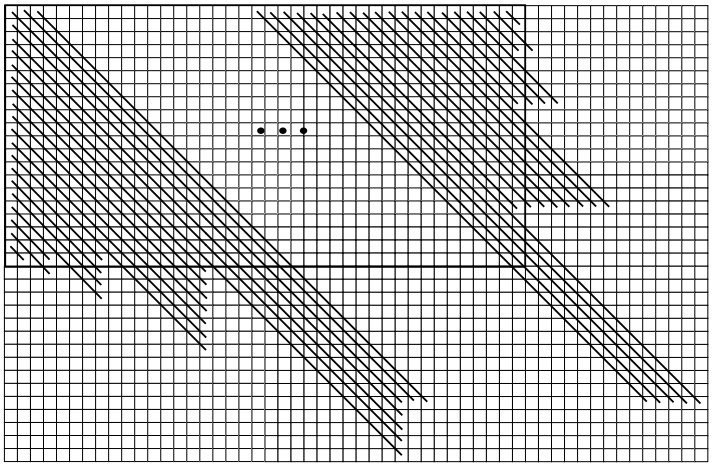


Figure 4.8: To save redundant work, threads are skipped that process diagonal elements with an index that is greater than the diagonal size rounded up by a power of two. The diagonal lines illustrate which elements are not skipped.

Function	$t_{fr,CPU}$ (ms)
`cvComputeIntegralImages`	18.1

Table 4.1: Execution time needed to compute upright and tilted integral image for 1920x1080 px image on CPU.

Efficient sequential computation is performed by dynamic programming techniques where previously computed results are used to compute the current value quickly [LM02].

The execution time of the CPU integral function function is stated in Table 4.1. The time needed to execute the GPU upright and tilted integral functions is stated in Table 4.2.
The GPU speeds up execution significantly. The total execution time on the GPU is 2.9× smaller than the execution time on the CPU. Computation of the tilted integral image takes longer than the upright integral image: 65% of the total time is spent in tilted integral image computation. This is due to less efficient global memory access and the 0-padding that is needed to increase diagonal sizes to a power of two. Global memory access is less efficient for a diagonal scan, because data access is non-aligned opposed to the aligned data access of a horizontal scan.

4.6 Conclusions

An efficient GPU implementation of upright and 45° rotated (tilted) integral image computation was created, that will accelerate STAR feature detection. These implementations are a first step towards a full STAR GPU implementation. The GPU functions were optimized for HD image resolution. However, the implementation can process images up to a resolution of 4096x4096px by increasing the GPU thread block sizes.

For upright integral image computation, the approach of [BHM10] was adopted. The

Function	$t_{fr,GPU}$ (ms)
`uprightIntegral`	2.16
`tiltedIntegral`	4.04
Total	6.20

Table 4.2: Execution time needed to compute upright and tilted integral image for 1920x1080 px image on GPU.

integral is computed by employing the work-efficient *parallel prefix sum algorithm*, first for image rows and subsequently for image columns. The algorithm is implemented efficiently on the GPU by providing global memory coalescence and avoiding shared memory bank conflicts. A drawback of the prefix sum algorithm is the requirement for an input array that is equal to a power of two. To avoid the need to round the number columns to 2048 for HD resolution, image columns were split into two sets with power of two size: the first set consists of 1024 rows and the second set of 64 rows. After the two image parts have been processed separately, the sum of all elements of the first set is added to the elements of the second set.

A new parallel algorithm was created to efficiently compute the tilted integral image. The algorithm is based on computation of the parallel prefix sum of image diagonals in two directions. In the implementation of the algorithm, shared memory bank conflicts are avoided. However, coalesced global memory access can not be provided due to the diagonal access pattern.
The requirement of power of two array sizes and the constraint that CUDA thread blocks must have constant size implies that many redundant computations are performed when diagonals are processed. Most of the redundant computations were saved by skipping threads that process data that falls outside the image. To this end an extra variable is passed to the parallel prefix sum algorithm that is equal to the size of the diagonal, rounded up to a power of two. To avoid rounding the number columns to 2048 for HD images, the same procedure as performed for the upright integral image is carried out.

The performance of the GPU implementations for HD resolution input was measured and compared to the efficient integral image function used in the OpenCV STAR CPU implementation. In OpenCV both the upright and tilted integral images are computed in one iteration over the image. Compared to the CPU implementation, computation of both integral images is accelerated 2.9× by the GPU.

CHAPTER 5

Performance of existing GPU implementations

After the performance optimizations described in Chapter 3, real-time change detection for HD images was still not feasible. Profiling of the application functions showed in which functions most time was spent. In order to assess the extent to which a GPU can speed up change detection, existing GPU implementations of the algorithms that take the most processing time on the CPU were investigated. The execution time of OpenCV GPU feature extraction, perspective transformation and feature matching was measured. Besides, an alternative GPU implementation of MRF filtering algorithm was investigated, that was proposed in [VN08].

5.1 Measurement method

The GPU functions were executed for 1920x1080 px (HD) video of an outside, off-road scene. 138 frames were processed and the average execution time per frame was measured. The NVIDIA Visual Profiler tool was used to measure the time spent in the GPU. The time needed to copy data from the host to the GPU device memory and back is included in the results.
The execution time of the GPU implementations was compared to the execution time of the optimized CPU functions described in Section 3.3 and 3.4.

5.2 Feature extraction

In Section 3.4.3 the CPU execution time of a combination of the STAR feature detector and the SURF feature descriptor was stated. For HD video it takes 118 ms per frame to extract features on the CPU, which is too long for real-time processing. Therefore the OpenCV GPU implementation of the SURF feature detector and descriptor was evaluated to see if a GPU can speed up feature extraction.

Function	$t_{fr,CPU}$ (ms)	$t_{fr,GPU}$ (ms)	S
`cv::SURF`	272	40.4	6.73

Table 5.1: Execution time of OpenCV SURF on CPU and GPU. Column two shows the average execution time per frame on the CPU ($t_{fr,CPU}$), column three shows the average execution time per frame on the GPU ($t_{fr,GPU}$) and column four states the speedup of $t_{fr,GPU}$ with respect to $t_{fr,CPU}$ (S).

Function	$t_{fr,CPU}$ (ms)	$t_{fr,GPU}$ (ms)	S
`cv::warpPerspective`	44.8	1.68	26.7

Table 5.2: Execution time of OpenCV perspective transformation on CPU and GPU

In Table 5.1 the average execution time per frame of the SURF feature detector/descriptor on the CPU ($t_{fr,CPU}$) and the GPU ($t_{fr,GPU}$) is given. SURF takes 40.4 ms on the GPU, while it takes 272 ms on the CPU. This corresponds to a speedup of 6.73×. Compared to the CPU STAR/SURF combination, it takes 2.90× less time.

5.3 Perspective transformation

In the change detection algorithm a perspective transformation is performed in both video stabilization and spatial alignment. This is carried out by the OpenCV `warpPerspective` function. OpenCV offers both a CPU and GPU function. The execution time of both functions was measured to see how much the GPU speeds up computation.

Table 5.2 shows the result of the measurements. The GPU implementation takes much less time than the CPU implementation, it is 26.7× faster.

5.4 Feature matching

Feature matching is performed in video stabilization and temporal synchronization. In the current change detection application version, the best match for a feature is computed by first creating a search tree of features and subsequently searching for the best match. This is efficient on the CPU, because a feature is compared to only part of a set of other features. Therefore many iterations are saved.

On a GPU creation of a search tree is not efficient, because it is a inherently sequential procedure. A function is executed efficiently on a GPU only if it can process many independent data elements in parallel. Therefore the most efficient way of performing feature matching on the GPU is to do *brute force* matching. Brute force means that a feature in one set is compared to *every* feature in another set. These comparisons are independent, as a result of which the comparisons can be performed in parallel on the GPU. On the CPU the OpenCV FLANN kd-tree method is used. On the GPU the OpenCV `BruteForceMatcher_GPU` function is adopted.

Function	$t_{fr,CPU}$ (ms)	$t_{fr,GPU}$ (ms)	S
`Vi::FeatureMatching`	8.70	1.03	8.45

Table 5.3: Execution time of matching the features of two video frames on CPU and GPU

Function	$t_{fr,CPU}$ (ms)	$\widehat{t_{fr,GPU}}$ (ms)	$\widehat{S}$
`Vi::ApplyMRF`	398	82.0	4.86

Table 5.4: Execution time on CPU and estimated execution time on GPU of Markov Random Field filtering

The execution time needed to match the features of two video frames on the CPU and GPU is stated in Table 5.3. On the CPU the time needed for matching is already small, but on the GPU it takes even less time. On the CPU it takes 8.70 ms and on the GPU 1.03 ms, which corresponds to a speed up of 8.45×.

5.5 Markov random field filtering

In the change mask generation block of the application, Markov Random Field (MRF) filtering is applied to make the change mask consistent and filter out noise. This performed by creating a Markov Random Field (MRF) over the image and minimizing an energy function defined over the MRF. This is a powerful yet computationally expensive algorithm. It would be advantageous to speed up the computation of the algorithm by a GPU.

We have found an efficient GPU implementation for MRF energy minimization by graph cuts in [VN08]. The authors apply the energy minimization to image segmentation, but the implementation is equally suited for MRF filtering. We have not yet been able to test the GPU implementation in the change detection context, but the performance numbers stated in [VN08] predict the possible GPU speedup in the change detection application.

The authors tested their implementation on three real-world test images and compared to a reference CPU implementation. The speedup of the implementation is equal to 4.86× on average for the three test images. The authors measured the GPU performance on a NVIDIA GTX 280. Our experiments showed that the CPU performance reported by the authors is comparable to the CPU performance of our change detection MRF function. Because the CPU performance is comparable and the authors have tested their implementation on a recent GPU, it may be assumed that a 4.86× speedup is feasible in the change detection application.

Table 5.4 shows the execution time per frame of the change detection CPU MRF function ($t_{fr,CPU}$) and the estimated execution time on a GPU ($\widehat{t_{fr,GPU}}$) for HD resolution. The GPU execution time was estimated by dividing the CPU time by the estimated speedup of 4.86×.

5.6 Conclusions

Every GPU implementation that was tested will speed up the change detection's computational performance. Table 5.5 shows an overview of the speedup of the GPU with respect

Function	$t_{fr,CPU}$ (ms)	$t_{fr,GPU}$ (ms)	S
`cv::SURF`	272	40.4	6.73
`cv::warpPerspective`	44.8	1.68	26.7
`Vi::FeatureMatching`	8.70	1.03	8.45
`Vi::ApplyMRF`	398	82.0	4.86

Table 5.5: Overview of the execution time per HD frame of the tested CPU/GPU implementations and the speedup of GPU with respect to CPU.

to the CPU for the tested application funcitions. The OpenCV SURF GPU function speeds up feature extraction by 6.73×, but it is 2.90× faster than the combination of STAR detection/SURF description on the CPU. Perspective transformation is speeded up by 26.7× if the OpenCV `warpPerspective` GPU implementation is adopted. A 8.45× speedup is reached in feature matching if OpenCV's `BruteForce- Matcher_GPU` is adopted. MRF filtering, the most expensive image processing algorithm in the application, is estimated to be 4.86× faster with the GPU implementation of [VN08].

CHAPTER 6

Hardware platform advice for real-time change detection

In this chapter an advice will be given on the hardware configuration needed for real-time change detection on SD and HD resolution video. The results of the CPU performance optimizations described in Chapter 3 and the measurements of GPU implementations described in Chapter 5 will be used to decide which algorithms are mapped to a CPU and which ones to a GPU.

6.1 SD resolution

An overview of the estimated CPU execution time for SD resolution is given in Table 6.1. Assuming that all optimizations mentioned in Chapter 3 are implemented, the time needed for change detection on SD resolution video is estimated to be 138 ms per frame. This corresponds to a frame rate of 7.25 Hz. As a minimum frame rate of 5 Hz is required for real-time change detection, real-time processing of SD video on one CPU only is feasible.

Category	$\widehat{t_{fr}}$ (ms)
Overhead	23.7
Feature extraction	13.2
Video stabilization	12.1
Temporal synchronization	39.2
Spatial alignment	5.06
Change mask generation	44.5
Total	138

Table 6.1: Overview of the estimated execution time per SD frame on the CPU after the optimizations described in Chapter 3 have been applied.

6.2 HD resolution

After all CPU optimizations have been implemented, 1199 ms per HD resolution frame are needed if only one CPU is used, as was stated in Section 3.5. Even if multithreading is applied to all image processing algorithms, real-time change detection is not feasible. Multithreading of a function on a 4-core CPU ideally results in a speedup of 4×, while a 6× speedup of the total time is required for a frame rate of 5 Hz. Therefore GPU implementations of the most time consuming functions were considered in Chapter 5.

Table 6.2 states the application profile for HD resolution, assuming that the GPU implementations discussed in Chapter 5 are adopted. In total 460 ms per frame are needed with this set of functions, which corresponds to a maximum frame rate of 2.2 Hz. Real-time change detection is infeasible for this set of functions, as a frame rate of 5 Hz is required. However, the CPU and the GPU can process frames in parallel when a frame delay is introduced, which is known as the *pipelining* concept. In the next section it is described to which extent hardware pipelining can increase the change detection frame rate.

6.2.1 Increasing frame rate by pipelining

A well-known method to increase the throughput of a system is the introduction of *pipelining*. In a synchronous pipelined system a number of processing units is connected in series with buffers in between. Every processing unit performs a part of the computation, such that data is processed in a time-sliced fashion. Every processing unit processes one data item during a time-slice, in parallel. In this way a latency is introduced, while throughput is increased.

The *Throughput* of the pipelined system is determined by Equation (6.1), where t_{PU_i} is the latency of computations performed in processing unit i. The minimum length of a time slice is dictated by the processing unit in which the most time is spent. The throughput corresponds to the reciprocal of the maximum latency.

$$Throughput = \frac{1}{max(t_{PU_1}, t_{PU_2}, \ldots, t_{PU_N})} \tag{6.1}$$

6.2.2 Multiple CPUs

First, the maximum throughput of a pipelined change detection system consisting of only CPUs and no GPUs is deducted. For maximum throughput, the change detection functions are distributed over the CPUs, such that the time spent in the CPUs is *balanced*. This way the maximum latency is minimum.

If the number of CPUs is increased, the maximum throughput is reached when the system consists of 4 CPUs. This is illustrated by Table 6.3, in which a balanced distribution of application functions over 4 CPUs is listed. The maximum latency is spent in CPU4, which executes `Vi::ApplyMRF`. If the number of CPUs would be further increased, the throughput of the total system is *not* increased, because `Vi::ApplyMRF` cannot be split up. This function limits the throughput of a hardware platform consisting of only CPUs to $\frac{1}{0.398} = 2.51$ Hz. Therefore, real-time change detection on HD video is not feasible on a hardware platform consisting of only CPUs.

Category	Function	Device	$\widehat{t_{fr}}$ (ms)
Overhead	`Vi::VideoInputVideoFile::read`	CPU	13.9
	`Vi::VideoInputVideoFile::seek`		41.3
	`Vi::VideoOutputVideoFile::write`		73.8
	`Vi::Array2D<unsigned char>::fill`		43.0
	`Vi::Array2D<unsigned char>::operator=`		0.95
	`ChangeDetection::cropBorders`		27.5
	`ChangeDetection::drawOverlayChanges`		8.51
	`Vi::Feature::load`		2.69
Feature extraction	`cv::SURF`	GPU	40.4
Video stabilization	`Vi::ImageStabilizationCIED::imageStabMainProcess`	CPU	15.2
	`cv::BruteForceMatcher_GPU`	GPU	1.03
	`cv::findHomography`	CPU	3.91
	`cv::warpPerspective`	GPU	1.68
Temporal synchronization	`cv::BruteForceMatcher_GPU`	GPU	30.9
Spatial alignment	`cv::findHomography`	CPU	6.23
	`cv::warpPerspective`	GPU	1.68
Change mask generation	`ImageDifference::buildDifferenceImage`	CPU	30.0
	`ImageDifference::changeVectorAnalysis`	CPU	10.3
	`AdaptiveThresholding::process`	CPU	18.6
	`Block::buildFrequencyAppearance`	CPU	2.75
	`AdaptiveThresholding::thresholdImage`	CPU	5.79
	`Vi::ApplyMRF`	GPU	79.6
Total			460

Table 6.2: Overview of execution time for HD video when GPU implementations described in Chapter 5 are adopted.

Category	Device	$\widehat{t_{fr}}$ (ms)
Overhead	CPU1	212
Feature extraction	CPU1	117
Subtotal		329
Video stabilization	CPU2	92.6
Temporal synchronization	CPU2	261
Subtotal		354
Spatial alignment	CPU3	51.0
Change mask generation excl. `Vi::ApplyMRF`	CPU3	67.5
Subtotal		119
`Vi::ApplyMRF`	CPU4	398

Table 6.3: Distribution of application functions over 4 CPUs.

Category	Device	$\widehat{t_{fr}}$ (ms)
Overhead	CPU	212
Feature extraction	GPU	40.4
Video stabilization	GPU	21.8
Temporal synchronization	GPU	30.9
Spatial alignment	GPU	7.91
Change mask generation	GPU	149
Subtotal		250

Table 6.4: Distribution of application functions over 1 CPU and 1 GPU.

6.2.3 1 CPU and 1 GPU

GPU(s) need to be added to the system to improve the change detection frame rate. A pipelined hardware platform consisting of 1 CPU and 1 GPU is investigated. Table 6.4 lists the optimal distribution of application functions over processing units for this configuration and the corresponding executing time. For functions that are executed on the GPU of which no implementation is available yet, it is assumed that the execution time on the GPU is equal to the execution time on the CPU. *Overhead* functions are executed on the CPU, while the other functions are executed on the GPU.

For this distribution, the latency of CPU functions is equal to 212 ms and the latency of GPU functions is equal to 250 ms. As a result, the throughput of the system is equal to 4.00 Hz.

6.2.4 1 CPU and 2 GPUs

Because a frame rate of 5 Hz is required, it is investigated to what extent the throughput can be increased if one extra GPU is added to the system. Table 6.5 shows the distribution of application functions over processing units for which throughput is maximum. The latency of computations in the CPU is the maximum latency, so the throughput is equal to $\frac{1}{0.212} = 4.72$ Hz.

Even if extra GPUs are added, the frame rate is too small to meet the real-time requirement of 5 Hz. The frame rate is limited by the execution time spent in *overhead* functions. Real-time change detection is feasible if the execution time spent in *overhead* functions is reduced, such that it is smaller than 200 ms. It is expected that the execution time spent in this category can be reduced sufficiently by creating a separate thread in which `Vi::VideoOutputVideoFile::write` is executed, which performs encoding and writes video frames to disk.

6.2.5 2 CPUs and 1 GPU

An alternative to increasing the number of GPUs is increasing the number of CPUs. The maximum throughput for a hardware platform with 2 CPUs and 1 GPU is investigated. Table 6.6 shows the distribution of application functions over processing units for which

Category	Device	$\widehat{t_{fr}}$ (ms)
Overhead	CPU	212
Feature extraction	GPU1	40.4
Video stabilization	GPU1	21.8
Temporal synchronization	GPU1	30.9
Spatial alignment	GPU1	7.91
Subtotal		101
Change mask generation	GPU2	149

Table 6.5: Distribution of application functions over 1 CPU and 2 GPUs.

throughput is maximum. As the throughput is limited by the time spent in the GPU, it is equal to 4.76 Hz.

Real-time change detection is infeasible on 2 CPUs and 1 GPU given the current set of implementations. However, the execution time of functions of which no GPU implementation is available was estimated conservatively. It was estimated that the execution time of a GPU implementation is equal to the execution time of the CPU implementation. As was shown in Chapter 5, most image processing algorithms are speeded up by a GPU. It is likely that the GPU will also speedup the not yet implemented functions. Therefore, it is expected that the execution time spent in the GPU will be smaller than 200 ms and as a result, real-time change detection will be feasible.

Function/Category	Device	$\widehat{t_{fr}}$ (ms)
Overhead excl. `Vi::VideoOutputVideoFile::write`	CPU1	138
`Vi::VideoOutputVideoFile::write`	CPU2	73.8
Feature extraction	CPU2	117
Subtotal		190.9
Video stabilization	GPU	21.8
Temporal synchronization	GPU	30.9
Spatial alignment	GPU	7.91
Change mask generation	GPU	149
Subtotal		210

Table 6.6: Distribution of application functions over 2 CPUs and 1 GPU.

6.3 Conclusions

Change detection on SD resolution can be performed with a frame rate of 7.25 Hz on the CPU of the experimental platform, as was also shown in Chapter 3.

Real-time change detection on HD resolution is infeasible on only 1 CPU, even after optimizations. If the GPU implementations described in Chapter 5 are adopted, the execu-

tion time decreases to 460 ms per frame. To improve performance, pipelined execution was proposed. Pipelining increases latency while throughput improves.

First it was investigated if real-time HD change detection is feasible if the number CPUs is increased. Real-time change detection is not feasible without GPU(s), because the MRF filtering function is a bottleneck.
Pipelined execution on 1 CPU and 1 GPU increases the throughput of the system to 4.00 Hz. If the number of GPUs is increased to 2, a frame rate of 4.72 Hz is feasible. On the other hand a configuration of 2 CPUs and 1 GPU enables a frame rate of 4.76 Hz. For both configurations, the difference between the minimum frame rate required for real-time processing and the actual frame rate is small. It is expected that real-time processing is feasible for both configurations after minor optimizations.
Regarding energy consumption, the configuration consisting of 2 CPUs and 1 GPU is preferred over 1 CPU and 2 GPUs, as the Intel Xeon W3550 CPU has a TDP of 130W, while the NVIDIA GeForce GTX480 has a TDP of 250W.

CHAPTER 7

Conclusions

In this project an image change detection application was mapped to a hardware platform consisting of CPU/GPU processing units, such that it can be executed at a frame rate of at least 5 Hz. The mapping was performed for input video with a resolution of 640x420px (SD) and video of a resolution of 1920x1080px (HD).

First, the performance of the initial application implementation was measured to see how efficiently the application is executed. A profile of application functions was created to get an overview of the application parts in which most time is spent. The profile showed that more than half of the total execution time was spent in functions of the *overhead* category and that approximately a quarter of the total time was spent in *feature extraction*.
In the initial application version 1089/13459 ms was needed in total to process a frame of SD/HD resolution on the CPU of the experimental platform.

A number of optimization opportunities were found by which performance could be improved. In *overhead* more than half of the total change detection processing time was spent in loading previously acquired feature data from disk. By removing redundant serialization operations and storing data into binary format instead of text format, the time needed to load feature data was decreased by 48.4× for SD and by 49.9× for HD video.
In *temporal synchronization* a *list* data structure was replaced by a *vector* data structure to prevent iteration through a list for random element access.
By reduction of the number of retrieved features for HD resolution video, the time spent in *overhead*, *temporal synchronization* as well as *video stabilization* was reduced.
In *change mask generation* the performance of the adaptive thresholding algorithm was increased by simplifying the data structure employed during histogram computation.
Two optimizations opportunities were found that have not been implemented yet. Introduction of a circular buffer will further reduce the time needed for feature data loading. Second, execution time will be saved if one image format is adopted instead of two different ones.

To reduce the time spent in *feature extraction*, a more efficient feature extraction algo-

rithm was proposed. A combination of the STAR detector and SURF descriptor results in a reduction of execution time by 21.9×/30.1× for SD/HD video, while the feature repeatability increases.

The proposed performance improvements will reduce the total execution time required per frame from 1089 to 138 ms for SD resolution and from 13459 to 1199 ms for HD resolution. This means the performance will be increased 7.9×/11× for SD/HD resolution respectively. As a result, real-time SD change detection on one 1 CPU is enabled, as 138 ms per frame enables operation at a frame rate of 7.25 Hz. For HD resolution real-time change detection is not feasible on 1 CPU after the described performance improvements.

An efficient GPU implementation of upright and 45° rotated (tilted) integral image computation was created, that will accelerate STAR feature detection. This implementations is a first step towards a full STAR GPU implementation.
For upright integral image computation, the approach of [BHM10] was adopted. Redundant operations due to the requirement for power of two sized arrays were avoided by splitting the image in two parts and processing the parts separately.
A new parallel algorithm was created to efficiently compute the tilted integral image. The algorithm is based on computation of the parallel prefix sum of image diagonals in two directions. Redundant operations because of the varying diagonal size and the need for power of two sized arrays were avoided by skipping threads that process data that falls outside the image.
Compared to the integral image function of the OpenCV STAR CPU implementation, computation of both the upright and tilted integral image is accelerated 2.9× by the GPU.

Because HD change detection was not feasible after the proposed CPU performance improvements, the extent to which a GPU can speedup execution was assessed. The OpenCV SURF GPU function speeds up feature extraction by 2.90× with respect to the STAR detection/SURF description combination on the CPU. Perspective transformation is speeded up by 26.7× if the OpenCV `warpPerspective` GPU implementation is adopted. A 8.45× speedup is reached in feature matching if OpenCV's `BruteForceMatcher_GPU` is adopted. MRF filtering, the most expensive image processing algorithm in the application, is estimated to be 4.86× faster with the GPU implementation of [VN08].

Based on the execution time of the improved CPU functions and the execution time of GPU implementations, an advice was formulated on how to map the application to a set of CPUs/GPUs such that the change detection requirements are met.
SD change detection can be performed in real-time on 1 Intel Xeon W3550 CPU.
Real-time HD change detection is infeasible on only 1 CPU, even after optimizations. To improve the frame rate, pipelined execution was proposed. First it was investigated if change detection is feasible if the application is executed on only CPUs. This is not feasible, because the MRF filtering function is a bottleneck. Pipelined execution on 1 CPU and 1 GPU increases the throughput of the system to 4.00 Hz. If the number of GPUs is increased to 2, a frame rate of 4.72 Hz is feasible. On the other hand, a configuration of 2 CPUs and 1 GPU enables a frame rate of 4.76 Hz. For both configurations, the difference between the minimum frame rate required for real-time processing and the actual frame rate is small. It is expected that

real-time processing is feasible for both configurations after minor optimizations.
For HD resolution, a configuration of 2 CPUs and 1 GPU meets the change detection system requirements best, because energy consumption is smaller than a configuration consisting of 1 CPU and 2 GPUs, while the output frame rate is approximately equal.

Bibliography

[AK08] Motilal Agrawal and Kurt Konolige. CenSurE: Center Surround Extremas for realtime feature detection and matching. In *ECCV*, 2008.

[BHM10] B. Bilgic, B.K.P. Horn, and I. Masaki. Efficient integral image computation on the GPU. In *Intelligent Vehicles Symposium (IV), 2010 IEEE*, pages 528 –533, june 2010.

[BJJ+11] Allessandro Becciu, Harro Jacobs, Egbert Jaspers, Hugo van Lint, Alessandro Rossi, Peter H.N. de With, and Dennis van de Wouw. D2.3 C-IED change detection system progress report. Technical report, ViNotion B.V., CycloMedia Technology B.V., Eindhoven University of Technology SPS-VCA, 2011.

[Cro84] Franklin C. Crow. Summed-area tables for texture mapping. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '84, pages 207–212, New York, NY, USA, 1984. ACM.

[CVG08] N. Cornelis and L. Van Gool. Fast scale invariant feature detection and matching on programmable graphics hardware. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, pages 1 –8, june 2008.

[GK10] Michael Garland and David B. Kirk. Understanding throughput-oriented architectures. *Commun. ACM*, 53:58–66, November 2010.

[HP03] John L. Hennesy and David A. Patterson. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann, 3th edition, 2003.

[HSO07] M. Harris, S. Sengupta, and J.D. Owens. Parallel Prefix Sum (Scan) with CUDA. In *GPU Gems 3*, chapter 39. Addison-Wesley, 2007.

[KZ04] Vladimir Kolmogorov and Ramin Zabih. What energy functions can be minimized via graph cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26:65–81, 2004.

[LM02] R Lienhart and J Maydt. An extended set of Haar-like features for rapid object detection. In *Proceedings International Conference on Image Processing*, volume 1, pages 900–903. IEEE, 2002.

[Low04] David G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60:91–110, November 2004.

[ML09] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *International Conference on Computer Vision Theory and Application (VISSAPP'09)*, pages 331–340. INSTICC Press, 2009.

[PXW10] M. Poremba, Yuan Xie, and M. Wolf. Accelerating adaptive background subtraction with GPU and CBEA architecture. In *Signal Processing Systems (SIPS), 2010 IEEE Workshop on*, pages 305 –310, oct. 2010.

[SA06] C. Su and A. Amer. A real-time adaptive thresholding for video change detection. In *Image Processing, 2006 IEEE International Conference on*, pages 157 –160, oct. 2006.

[VJ01] Paul Viola and Michael Jones. Robust real-time object detection. In *International Journal of Computer Vision*, 2001.

[VN08] V. Vineet and P.J. Narayanan. Cuda cuts: Fast graph cuts on the gpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, pages 1 –8, june 2008.

[ZCW10] Nan Zhang, Yun-Shan Chen, and Jian-Li Wang. Image parallel processing based on GPU. In *Advanced Computer Control (ICACC), 2010 2nd International Conference on*, volume 3, pages 367 –370, march 2010.