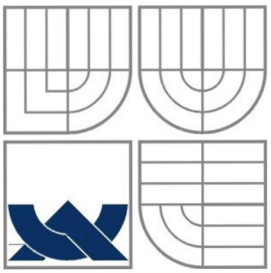
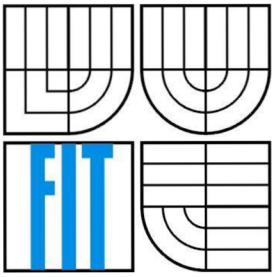


BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



FACULTY OF INFORMATION TECHNOLOGY

DEPARTMENT OF INTELLIGENT SYSTEMS

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

HARDWARE ACCELERATED DIGITAL IMAGE STABILIZATION IN A VIDEO STREAM

STABILIZACE OBRAZU POMOCÍ HARDWAROVÉ AKCELERACE

SEMESTER THESIS

SEMESTRÁLNÍ PRÁCE

AUTHOR
AUTOR PRÁCE

Bc. DÁVID PACURA

SUPERVISOR

VEDOUCÍ PRÁCE

doc. Ing., Dipl.-Ing. MARTIN DRAHANSKÝ, Ph.D.

BRNO 2016

Brno University of Technology - Faculty of Information Technology

Department of Intelligent Systems | Academic year 2015/2016

Master Thesis Specification

For: **Pacura Dávid, Bc.**
Branch of study: Bioinformatics and biocomputing
Title: **Hardware Accelerated Digital Image Stabilization in a Video Stream**
Category: Image Processing

Instructions for project work:

1. Study the available literature concerning digital image stabilization in a video stream. Also familiarize yourself with the available GPGPU architecture in the workplace.
2. Design suitable parallel algorithm for GPGPU digital image stabilization in a video stream.
3. Implement the proposed algorithm.
4. Create suitable video sequences in order to test the implemented algorithm. Also prepare the system for real time testing.
5. Perform experiments with data in a real time. Summarize and discuss achieved results.

Basic references:

- KIR Burcu, KURT Meltem, URHAN Ouzhan. Local Binary Pattern Based Fast Digital Image Stabilization. *Signal Processing Letters, IEEE*, 2015, 22.3: 341-345.
- LI Gang. FPGA implementation of real-time digital image stabilization. In: *Selected Proceedings of the Photoelectronic Technology Committee Conferences held July-December 2013*. International Society for Optics and Photonics, 2014. p. 91421U-91421U-7.
- DRAHANSKÝ Martin, ORSÁG Filip a HANÁČEK Petr. Accelerometer Based Digital Video Stabilization for General Security Surveillance Systems. *International Journal of Security and Its Applications*. Daejeon: Středisko pro podporu vědy a technického výzkumu, 2010, Vol. 2010, No. 1, p. 10. ISSN 1738-9976.

Requirements for the semestral defense:
Items 1 and 2.

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

The Master Thesis must define its purpose, describe a current state of the art, introduce the theoretical and technical background relevant to the problems solved, and specify what parts have been used from earlier projects or have been taken over from other sources.

Each student will hand-in printed as well as electronic versions of the technical report, an electronic version of the complete program documentation, program source files, and a functional hardware prototype sample if desired. The information in electronic form will be stored on a standard non-rewritable medium (CD-R, DVD-R, etc.) in formats common at the FIT. In order to allow regular handling, the medium will be securely attached to the printed report.

Supervisor: **Drahanský Martin, doc. Ing., Dipl.-Ing., Ph.D.**, DITS FIT BUT
Beginning of work: November 1, 2015
Date of delivery: May 25, 2016

L.S.

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

Petr Hanáček
Associate Professor and Head of Department

Abstract

The aim of this thesis is to propose a new method for digital image stabilization in video stream by exploiting computing power of GPGPU. This unit enables a real time stabilization of high resolution digital video sequences, which is important for further post-processing in computer vision and/or military applications. In order to compare available architectures for GPGPU programming, the proposed algorithm is implemented in three major frameworks. Results are then compared and discussed.

Abstrakt

Cílem této práce je návrh nové techniky pro stabilizaci obrazu za pomoci hardwarové akcelerace prostřednictvím GPGPU. Využití této techniky umožnuje stabilizaci videosekvencí v reálném čase i pro video ve vysokém rozlišení. Toho je zapotřebí pro ulehčení dalšího zpracování v počítačovém vidění nebo v armádních aplikacích. Z důvodu existence vícerých programovacích modelů pro GPGPU je navrhnutý stabilizační algoritmus implementován ve třech nejpoužívanějších z nich. Jejich výkon a výsledky jsou následně porovnány a diskutovány.

Keywords

Digital image stabilization, hardware acceleration, GPGPU, GPU, CUDA, OpenCL, C++ AMP, kernel, parallelization

Klíčová slova

Digitální stabilizace obrazu, hardwarová akcelerace, GPGPU, GPU, CUDA, OPENCL, C++ AMP, kernel, paralelizace

Citace

PACURA, Dávid: Hardware Accelerated Digital Image Stabilization, Master's thesis, Brno, 2016, Vysoké učení technické v Brně, Fakulta Informačních technologií. Vedoucí práce Drahanský Martin.

Hardware Accelerated Digital Image Stabilization in a Video Stream

Prohlášení

„Prohlašuji, že jsem svou diplomovou práci na téma Stabilizace obrazu pomocí hardwarové akcelerace vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.
Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních, a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.“

……
Dávid Pacura
24.5.2016

Table of Contents

List of Figures

1 INTRODUCTION

Over the past two decades, a rapid development in a field of electronic technology brought, among other things, an increase of use of digital video cameras. This was allowed by high miniaturization followed by low prices, and as result, usage of analog video cameras was edged out. Nowadays, as a consequence, a wide variety of use cases where digital video camera is used exists (e.g. movies production, home video recording, video surveillance, reconnaissance, motion detection, target tracing, automatic recognition, evaluation and verification). However, in many cases, camera device is hand held or mounted on moving objects (e.g. ships, vehicles) or on high poles and towers, where object movement or gusting wind causes camera shaking. In a lot of these cases, high resolution, high frames count per second and steady image without parasitic effects like shake, jitter and blur is required. This is due to requirements for successful post-processing like target tracking or movement detection [1]. Another important aspect is video compression ratio – a better compression can be achieved with stable image with low changes than with shaky video with lots of changes between subsequent frames [2].

However, there is often limited space, resources or both to fulfill these requirements – usage of hardware stabilization is in many cases highly restricted or even impossible. In this case, a digital image stabilization can be used. This enables a use of smaller video cameras, but it requires high computing power for post-processing. Often, real time processing of data is also required, which increases requirements for computing power even more, because of not only video stabilization, but also additional required steps in process must be resolved almost intermediately. This can be achieved by specialized hardware like FPGA (field programmable gate array) or using GPGPU (general purpose graphics processing unit). Both approaches have their application and even that the energy efficiency of FPGA is generally better, their performance is comparable or even worse than of GPU [3]. However this thesis will focus only on GPGPU because of its easy availability and relatively low price. This is a result of the development of CPUs in the last years, where Intel's versions all integrate relatively powerful graphic processor on the same printed circuit board (PCB) [4] and AMD offers APUs – CPUs fused with GPUs [5].

2 TERMS AND TAXONOMY

In order to fully understand principles and algorithms used in digital image stabilization, used terms and taxonomy are introduced first.

2.1 Causes of image blur

One of the biggest problems of image capture is blur. It can originate from two phenomena – shaking of the capture device, wrong focal distance of lenses, or both. However, while focal distance can be fixed relatively easily, shake compensation is much harder – generally, it is a slight random movement of camera in some direction. Its causes are various – from hand shaking through movable platform shaking (moving vehicles) to fixed platform shaking (high towers and gusting wind).

A special category are telephoto shots, where only narrow field of view is used – high zoom magnifies even the slightest jitter and captured sequence becomes unstable. In case of photographs, this can be compensated by high shutter speed and bigger aperture. However, this is not always possible due to combination of various reasons (limited resources or space, distance of object to capture, or light conditions).

In case of video capture, even usage of this countermeasures does still produce a resulting video sequence, where although object of interest is not blurred, it is also not still. Luckily, a way to compensate these parasitic effects exists and it is called image stabilization. A multiple approaches to this problem exist, each based on different principle, but with one ultimate objective – to produce shake-free video sequence [6].

2.2 Physiological tremor

Tremor is an involuntary oscillatory movement of body parts directly generated by muscles during their activities when they contract and relax repeatedly. It is common physiological phenomenon present in all human beings [7]. It is independent of age and not clearly visible to the naked eye. However, it depends on the muscles capability to maintain certain position against the force of gravity (standing up and holding camera with outstretched arms produces higher physiological tremor in arm muscles than holding camera with arms supported by a stable object). The consequences of this phenomena are shaking image in video sequence captured by handheld devices and/or blurring effect of objects.

In order to bring adequate countermeasures for tremor introduced to the video by user hand-holding the camera, statistical modeling of its effects is required. An acquisition campaign [8] identified these vibrations as an oscillating signal with normal distribution and amplitude less than 0.5 degrees and typical frequency between 0 Hz and 20 Hz (see Figure 2.1). Another study [9] shows, that 99% of shake motion is below 10 Hz and amplitude less than 0.75 degrees. Both studies agree, that translation itself can be

neglected, as it would require movement of 10 cm in each of axes to significantly affect the viewing pose. This is rather unlikely in case of handshake.

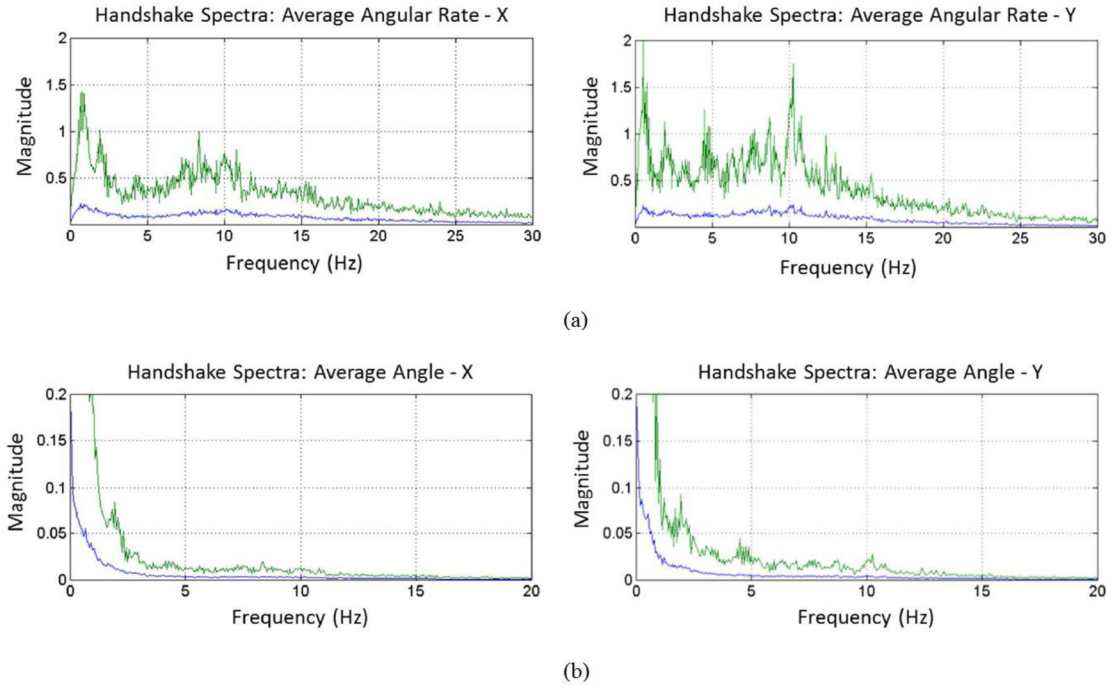


Figure 2.1: Mean (blue) and +3σ (green) bound on hand tremor spectra: a) angular rates measured on x and y axis; b) angles measured on x and y axis [8].

As can be seen from Figure 2.1, the amplitude of 0.5 degrees is computed with standard deviation of 3, which means that the 99.7 % of values lies within this interval.

2.3 Effects of stabilization on resulting video

Stabilized video has one more advantage beside those discussed before – smaller resulting size. This is a result of compression algorithms used in video codecs, because of huge space requirements of uncompressed video signal: uncompressed color (8-bit RGB) FullHD video (1,920 × 1,080 pixels) with 30 frames per second would require 625 gigabytes of space to store one hour of recording. Therefore, various standards for video compression exist, which reduces space requirements by various techniques of removing duplicate information – e.g. Xvid [10], MPEG-4 [11] or H.264 [12].

Among other techniques, a commonly used one is inter-frame prediction. It is used to reduce amount of duplicate information between frames by recycling data – frames are divided into macroblocks and future frames can refer to these. A relation exists between number of macroblock reuse and resulting video output size – the more similar subsequent frames are, the more prediction can be used and less space is needed to store the information. Therefore, video stabilization can significantly reduce resulting video size by making subsequent frames more similar [13].

2.4 Video stabilization quality evaluation

The important part of video stabilization is the evaluation of its quality. For this reason, multiple approaches exists. Yet, all of them have some issues. In this section, the ones that will be used in this work during testing are presented.

2.4.1 Visual evaluation

The visual comparison is one of the most common quality evaluation types, as it enables direct assessment by the visual comparison. However, it is not objective and does not enable the absolute evaluation of video quality, but only relative comparison (e.g. better, worse) between different video sequences. Also, the different people may have different preferences, which yields another uncertainty to the evaluation process.

2.4.2 Inter-frame Transform Fidelity

Another option is the use of commonly used metrics – the Inter-frame Transform Fidelity (ITF) [14] – it is the modification of the Peak Signal to Noise Ratio (PSNR) metrics used to compare the similarity of two images. It is defined as follows:

$$ITF = \frac{1}{N_f - 1} \sum_{k=1}^{N_{f-1}} PSNR(k) \quad \text{[dB]} \tag{2.1}$$

where N_f is the number of frames in video sequence and

$$PSNR(k) = 10log_{10} \frac{Ip_{max}}{MSE(k)} \quad \text{[dB]} \tag{2.2}$$

is the peak signal to noise ratio between two consecutive frames, where

$$MSE(k) = \frac{1}{M \times N} \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} \|F_k(i,j) - F_{k-1}(i,j)\|^2 \tag{2.3}$$

is the mean square error between monochromatic images with dimensions of $M \times N$, Ip_{max} is the maximum possible pixel intensity in the frame and F_k is the k-th frame from sequence.

Yet, even this quality evaluation metrics is not optimal, as it does not respond well to zoom changes or presence of multiple moving objects in image. Therefore, while it responds very well to the videos with mostly static objects present in scene, it will yield very similar results for both original and stabilized videos containing rapid zoom changes or captured with moving camera (e.g. car mounted). For this case, another evaluation is required.

2.4.3 Camera path monitoring

The next possibility for evaluation is the comparison of the camera path before and after video stabilization. In its basic form, where the assessment of smooth camera motion is evaluated visually only by humans, the same problem as with visual evaluation itself (section 2.4.1) – inability to specify the exact quality value – exist. However, the smoothness can be evaluated also by algorithmic approaches.

For fully objective evaluation, it is required, that for the same video sequence, the global motion vectors are created for both original and stabilized video sequence, ideally by some commonly used algorithm for video stabilization with good displacement of frames evaluation.

One of possibilities for this evaluation is the use of the frequency analysis – the sequence of global motion vectors is transformed using discrete Fourier transform (DFT) for both original and stabilized video. Then, the presence of different frequencies and their magnitude is evaluated. In the ideal case, the stabilized video should contain only low frequencies up to several Hertz (Hz), as the intended camera motion is slow and smooth (composed from low frequencies) while the jitter causes rapid changes of camera pose, therefore higher frequencies are present in signal.

However, this approach has, similarly to the previous two, issues: only the frequencies up to the half of the sampling video frequency can be detected. This is the rule of the sampling theorem. Therefore, for the input video with 24 frames per second (FPS), only frequencies up to the 12 Hz can be detected.

3 IMAGE STABILIZATION SYSTEMS CLASSIFICATION

The purpose of video stabilization is to remove unwanted motion in video feed. To achieve this, a multiple principles can be used. In literature, a total of four types of stabilization is recognized. They are all discussed in this chapter. However, in real scenarios, their combination is often used resulting into a hybrid methods empowering best of used principles.

3.1 Mechanical image stabilizer

A mechanical principle of image stabilization (MIS) is the oldest method. Unlike other methods, this one does require additional equipment besides camera. The simplest example is tripod – a portable tree-legged frame or monopod – also a portable, but one-legged frame, both used as a platform to maintain camera in stable position. However, it was not invented with analog video itself. Disadvantage of this approach is impossibility to stabilize image on a moving platform. Besides these solutions, a more advanced option is Steadicam [15]. It was invented in 1973 by commercial director Garrett Brown. It consists of three major elements:

- An articulated, iso-elastic arm.
- A specialized sled for camera equipment holding.
- A supportive vest.

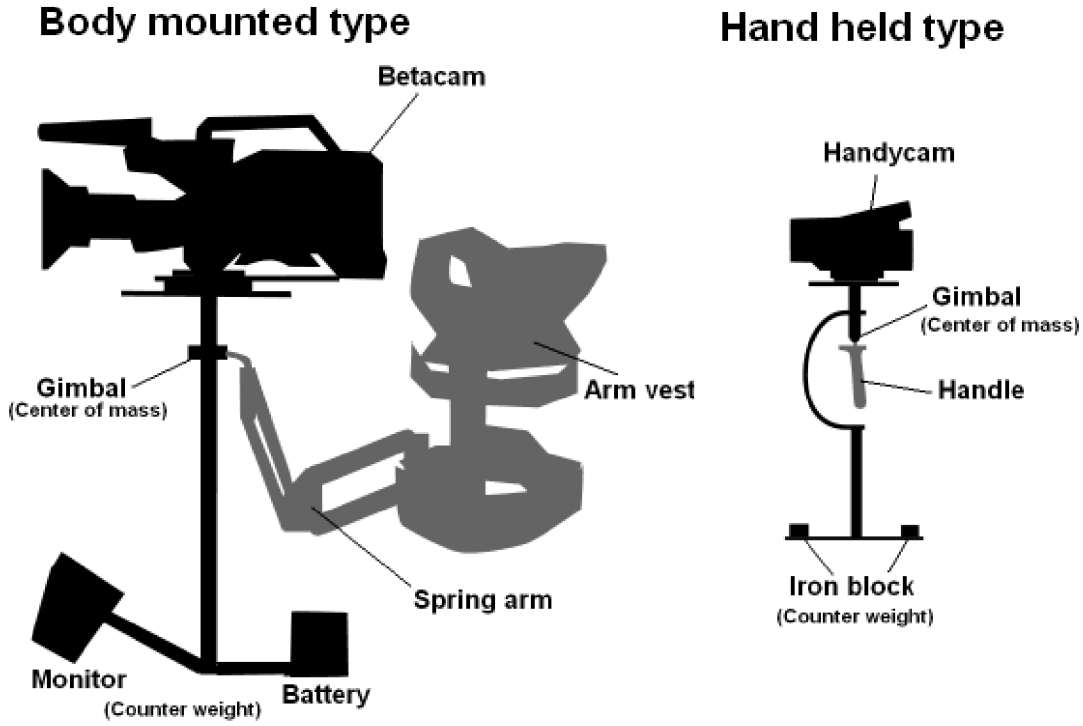


Figure 3.1: Principle of Steadicam [16].

Also, a slightly modified hand-held version exists. It is used for lighter cameras and consists only of two parts – the supportive vest is no longer required.

Steadicam utilizes only a simple dynamic characteristics of balanced object and does not have any active control mechanism. The main part – the sled for a camera is divided into two parts. On one end, a platform for the camera is placed, on the other end, a counter weight is required. Based on the camera type and other requirements, another equipment can be mounted (e.g. additional battery or monitor) instead.

Iso-elastic arm is connected to sled in its center of mass with 3-axis gimbal. The iso-elastic arm is composed of spring and two links that enables smooth movement. Arm then acts as a shock absorber – with movement of operator, a part of arm attached to west moves, however spring system in arm responds and effectively cancels sharp jolts. This enables cancelation of any rotations of pitch, yaw and roll that results in shaking video and at the same time enables operator smooth positioning [15] [16].

3.2 Optical image stabilizer

Optical image stabilization (OIS), also called lens-shift stabilization, controls the optical path between the target and the image sensor by moving mechanical parts of the camera itself. For this, actuators for path correction and sensors for position following are needed. This raises the complexity, thus increasing dimensions and cost of the camera module. However, this principle can be still more space-efficient than electronic stabilization methods. Especially in case of small image sensors, like adaptive Liquid Lens (LL) or Shape Memory Alloy (SMA), where small lenses are required. However, other principles are also applicable – piezo-electric motor or Voice Coil Motor (VCM) [8], therefore it can be used also in bigger electronics like compact cameras and Digital Single-Lens Reflex cameras (DSLRs).

Practically, two principles exist [8]:

- Barrel Shift – image sensor is in fixed position and lenses move with translation movement.
- Camera Tilt – image sensor is integrated with lenses and their movement is angular.

In case of position sensors for detection of lens movements, also two principles exist:

- Hall sensors – appropriate for barrel shift.
- Photo sensors – appropriate for camera tilt.

An electronic circuit implementing OIS is therefore composed of four parts:

- MEMS gyroscope – for detection of movements and vibrations inflicted on the system in the horizontal and vertical axes.
- Hall sensors – for detection of lens movements within camera module.
- Driver – for piloting the camera module into right position.
- Microcontroller – for executing the control algorithm.

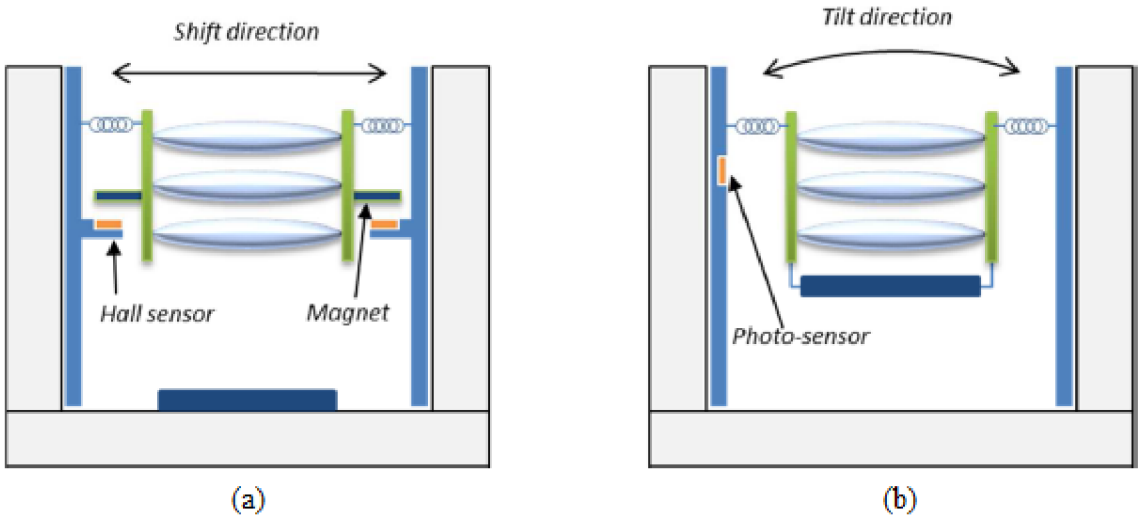


Figure 3.2: Schematics of Barrel shift sensor with Hall sensor (a) and Camera Tilt with Photo sensor (b) [8].

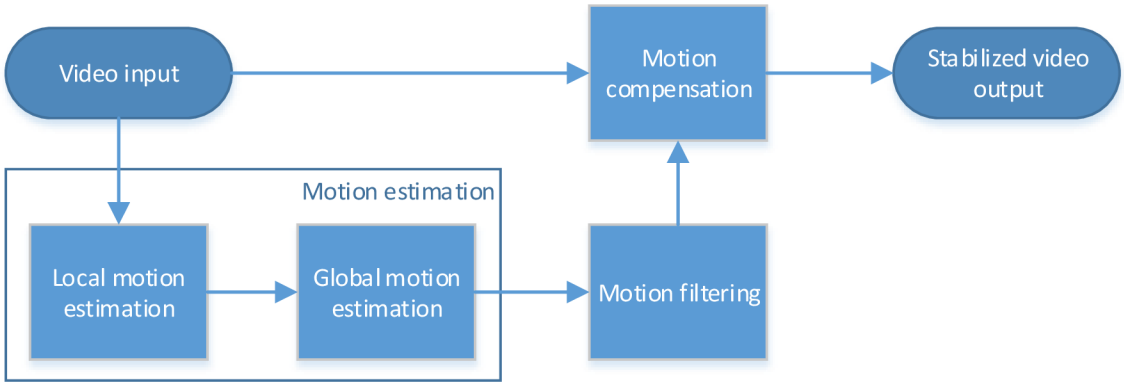
3.3 Electronic image stabilizer

Electronic image stabilizer (EIS), also called sensor-shift stabilization or in-camera stabilization, controls position of the image sensor itself. In principle, it is the same as OIS – actuators use MEMS gyroscope to detect movements and compensate it with actuators by moving the image sensor into the center of lenses' optical path. However, actuators are in this case actually stepper motors (because of size of part required to move). As a result, this solution is bigger than OIS [17]. A performance comparison [18] shows, that this solution is slightly worse, however, it has its application – OIS cannot be used in all cases (movement of bigger and relatively heavy lens systems is rather difficult).

3.4 Digital image stabilizer

Digital image stabilization (DIS) is in contrast to the previous methods, as it does not require any special hardware and is implemented purely by software. Therefore, it is the best solution in terms of size and resources, as it has no hardware requirements for image sensor. However, this principle has relatively high computational demands, as number of operations for a set of frames rises exponentially with their resolution ($\mathbf{O}(n^2)$ or worse).

DIS methods have three implementation steps: motion estimation, motion filtering and motion compensation (see Figure 3.3). Their importance and function will be discussed below.



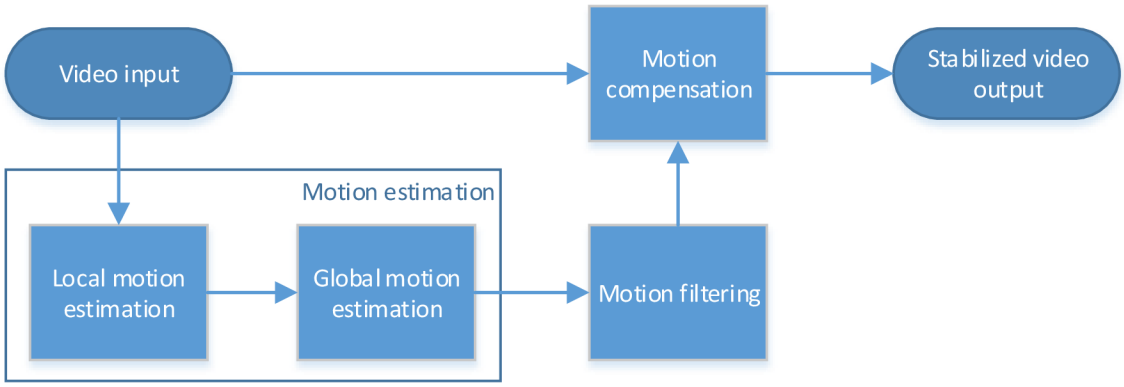


Figure 3.3: Processing flow of DIS [2].

3.4.1 Motion estimation

The first step of digital video stabilization is to determine how the camera is moving. This can be done by comparison of subsequent video frames. Because of a possibility of multiple movements in a scene (e.g. shaking and panning of the camera together with multiple movements in the scene), a two-step motion estimation must be performed in order to determine the movement of the camera itself. Firstly, the current frame is divided into regular grid (in order to reduce computational cost), where each grid contains equal number of pixels (typically 8×8 or 16×16 pixels). Then, local motion vectors (LVM) are computed for each block by comparing blocks from current frame with corresponding blocks in previous frame. Search for the corresponding blocks is done by searching block's neighborhood in previous frame. The global motion vector (GMV) representing the estimated movement of the camera between subsequent frames is then obtained as a combination of local motion vectors [2]. Because of big search space and high number of compare operations, motion estimation (ME) is the most computationally expensive part of algorithm. It is also important to note that correct motion estimation is vital, as any error introduced in this step may affect motion compensation step [19].

3.4.2 Motion filtering

Even though global motion estimation vector calculated in previous step represents only overall camera motion, it still contains both wanted (movement of the camera that should remain after stabilization; i.e. intended movements like panning, zooming and tilting) and unwanted motion (shake and jitter). In order to effectively stabilize video, these two types of movement must be separated and motion compensation must be based on the unwanted motion.

To enable separation of these two types of motion, their characteristics must be compared. The wanted motion is relatively slow and smooth, continuing over multiple frames in similar directions, thus producing lower frequencies. By contrast, unwanted motion is represented by quicker and random changes between frames, therefore produces higher frequencies. However, the specific border between these two types of movement depends on the situation of the camera (e.g. parasitic shaking from handheld camera during walking has different characteristic from jittering caused by a camera mounted on high pole by gusting wind) [20].

3.4.3 Motion compensation

Standard motion compensation (MC) works by cropping an image from the sensor by using a movable window with smaller resolution. This window is then moved between images in order to minimize the difference between the current and the previous frame [6]. Because of this, effective resolution of the stabilized video is lower than that of the sensor. Therefore, a maximal resolution of window (effective output area) in each axis can be calculated:

$$EOA = SA - UA \qquad [px] \qquad (3.1)$$

where EOA is effective output area, SA is sensor area and UA is unsafe area.

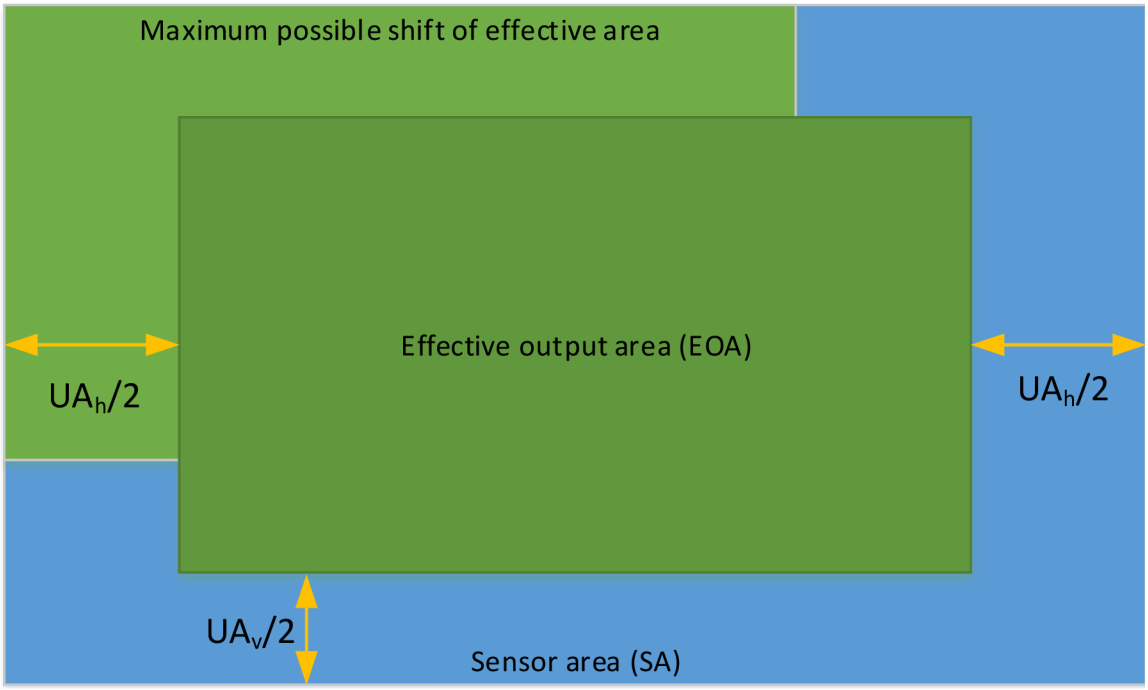


Figure 3.4: Digital stabilization principle.

The unsafe area (UA) for both horizontal and vertical axes is computed as:

$$UA = \frac{res}{AOV} * \alpha \qquad [\frac{px}{°}] \qquad (3.2)$$

where UA $[\frac{px}{°}]$ is unsafe area in pixels for one angular change of one degree, res $[px]$ is actual resolution in dimension (horizontal or vertical), AOV [°] is actual angle of view and α is maximum absolute difference of angles.

In case of video camera held by a healthy human focused on holding camera still, a typical angular change is 0.5 degrees in each direction (see Chapter 2.2 Physiological tremor), therefore $\alpha = 1°$.

An actual angle of view (AOV) can be computed from the focal length of lenses and size of the chip. Then, based on AOV, chip resolution and maximum possible angle of change, a number of unsafe pixels can be computed. In case of rectilinear (no-spatially-distorted) lenses, field of view is:

$$AOV = 2\ arctan\frac{l}{2f} \qquad [°] \qquad (3.3)$$

where AOV [°] is the actual angle of view, l $[mm]$ is the length of sensor in dimension (horizontal or vertical) we compute AOV and $f\,[mm]$ is equivalent to F $[mm]$ if actual focus is in infinity.

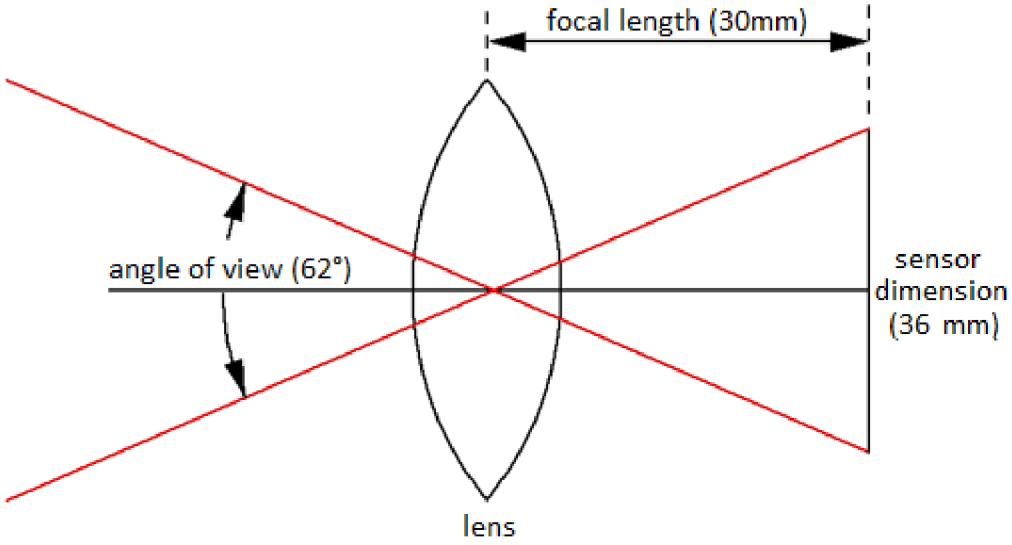


Figure 3.5: A schema of relationship between AOV, F and sensor dimension.

Relationship between focal length and percentage of chip area considered as unsafe is linear (see Figure 3.6). However, angle of view decreases exponentially with F. This means that while with lower values of F, magnification rises rapidly, with high values of F, changes in magnification become negligible (change in magnification of 35 mm film equivalent between $F = 600\ mm$ and $F = 700\ mm$ is only 1.17 ×, while the unsafe area rises from 29 % to 34 %). As a result, with large magnification, a digital stabilization becomes impossible, as it would require effective output area to be unusable in practical applications (e.g. stabilization of FullHD video $(1{,}920 \times 1{,}080\ px)$ video at $F = 600\ mm$ would produce only output video resolution of $1{,}361 \times 609\ px$, which is less than HD resolution $(1{,}280 \times 720\ px)$).

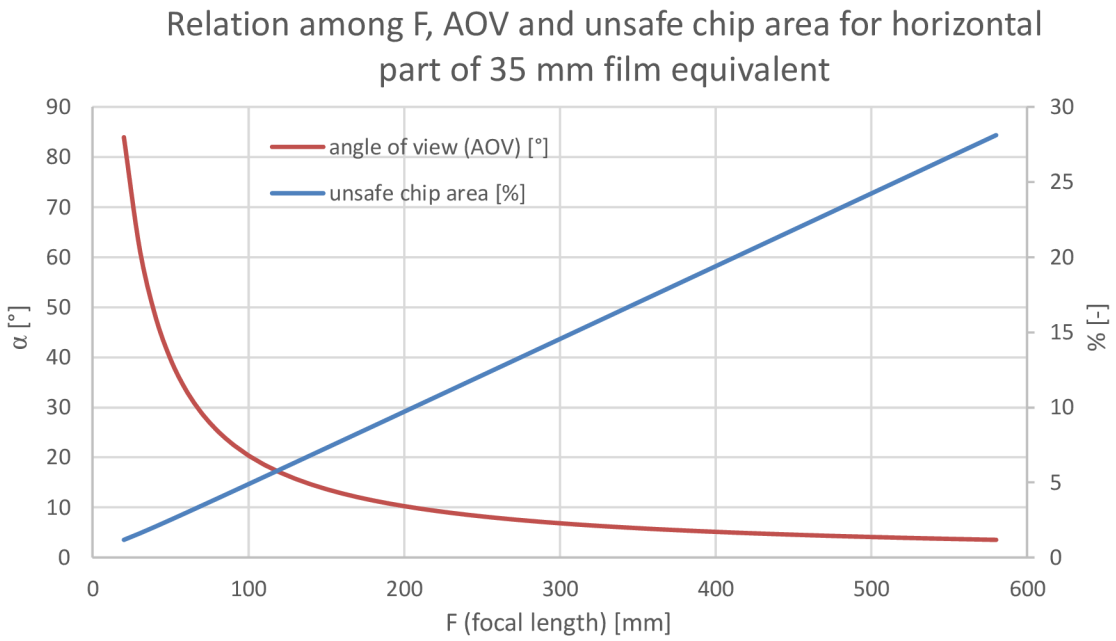


Figure 3.6: Relationship between angle of view and unsafe chip area with focal length of horizontal part of 35 mm film equivalent.

With increase of resolution of sensor area, a need for higher computing power arises. While exact requirements for number of computations depend solely on implemented method, a search space rises with power of 2 ($\mathbf{O}(n^2)$). Therefore various heuristic methods exist in order to reduce required computational power while keeping performance the same.

3.4.4 Enhanced motion compensation

Beside standard methods for motion compensation (MC), enhanced versions also exist. They do not crop video immediately by using moving window. Instead, they firstly compute positions change for each frame and then compute window dimensions that are capable of covering entire scene. Also, frames do not necessary have to be replaced entirely – only overlapping part of new frame overwrites old frame and remaining data is preserved. This approach is called mosaicking [21] and enables maximal utilization of available visual information – after computation of shifts of individual frames, a final effective resolution can be easily computed, enabling higher output than standard motion compensation (even in case of entirely replaced frames, as final window is no longer dependent on predicted statistics, but rather actual statistic of video). Advantage of this method is resistance to occasional peak changes that would cause black frame or shifted frame in standard MC. However, a problem with combining old and new frames arises, when changes in perspective or focus are present, a change outside current camera's angle of view occurs or stabilization is not perfect, as this phenomena produces disruptive transitions which are disturbing for humans.

3.5 Chapter summary

With performance comparison between MIS, OIS and EIS [18], the best one is OIS, with EIS right behind and the last one is MIS. As of digital image stabilization, it can be hardly compared to the previous methods, because while they are used before image capture (and therefore influencing image parameters like shutter speed and/or aperture), DIS is used in post-processing (and cannot influence image parameters). However, it can be assumed, that its performance depends on sharpness of the image – stable image can be achieved, however blurry image cannot be fully restored to its sharp original.

Another limitation is resolution penalty of DIS when compared to other methods. However, in case of video capture, this is not as big problem as in photography, because of lower requirements on resolution. Because of this, DIS can be used as a secondary stabilization method in order to compensate shake, which was not possible to filter out through other methods. However, it is important to note that digital image stabilization reduces output resolution, therefore its ability to stabilize input video sequence is limited by amplitude of jitter. This problem is significant mostly in bigger magnification, where low angle of view is present and even slight angle changes cause significant image shift. Mosaicking is a solution to this problem, but this approach brings new problems, that can outweigh its advantages. Therefore, a detailed analysis of the context must be performed prior to actual video stabilization.

4 GPGPU ACCELERATION PLATFORMS

General purpose graphic processing unit (GPGPU) is phenomenon of the last decade. The beginnings reach back to the 2000, when demonstration of abstraction of GPU as a SIMD (single instruction, multiple data) processor through 3D graphics API OpenGL took place. However, it was not competitive to CPU, because even as simple task as adding two vectors required about 20 lines of code [22] and raw processing power of GPUs did not exceed those of CPUs until year 2003 (Intel Pentium 4 Northwood @ 2.4 GHz with 9.6 GFLOPS vs. NVIDIA top GPU GeForce FX 5800 with 12 GFLOPS). After that, programmers realized that graphics processors have much higher raw performance grow through new generations than CPUs and are suitable for computations even though its usage is limited. Therefore, their interest in general purpose computation through specialized units in GPUs had risen significantly. As a consequence, in 2003, a research group from Stanford created ISO C99-like language called BrookGPU that provided a more convenient way for graphic cards programming. However it was still not possible to overcome some hardware limitations, like no elements indexing and limited data types – organized into triples (for RGB) and foursomes (for RGBA) [23]. All this changed with the release of new DirectX API version 10 – two biggest GPU companies brought highly programmable GPUs together with support for new APIs (AMD's FireStream [24] and NVIDIA's CUDA [25]) for general purpose computing. However, GPU is still an accelerator connected through peripheral component interconnect express bus (PCI Express) and requires host processor (CPU) to schedule work.

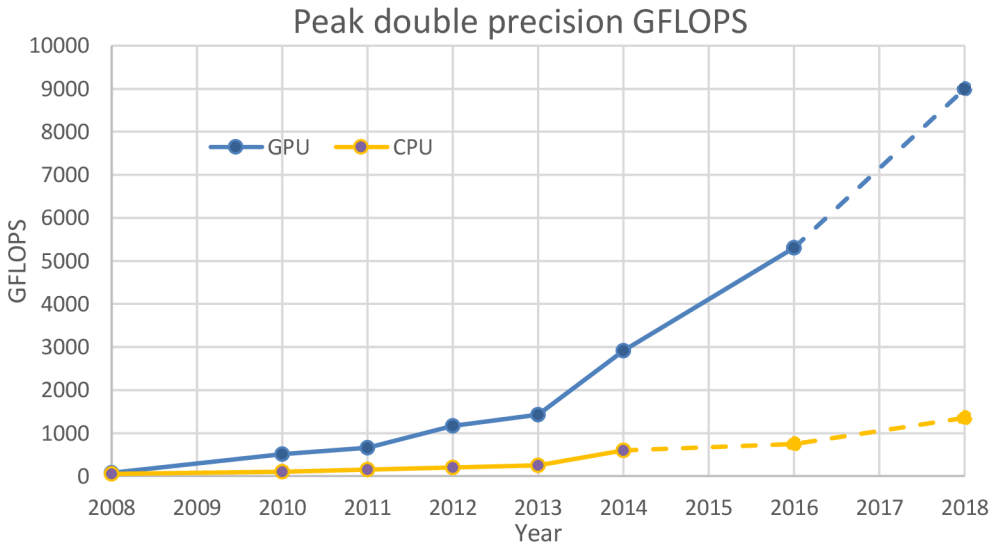


Figure 4.1: Development and prediction of CPU and GPU computing power.

The reason behind high performance of GPU is high number of simple processing units. However, because of its simplicity, GPUs do not outperform CPU in all tasks. Generally, computer architectures can be classified by Flynn's taxonomy [26]:

- SISD – single instruction stream, single data stream.
- SIMD – single instruction stream, multiple data streams.

- MISD – multiple instruction streams, single data stream.
- MIMD – multiple instruction streams, multiple data streams.

Based on this division, single core CPU is SISD architecture, multicore CPU is MIMD architecture. However, GPU is both SIMD and MIMD – SIMD in case of data-based parallelism on compute unit's level and MIMD in case of task based parallelism on compute units group's level.

Use of GPGPU brings a huge advantage in comparison to traditional CPU processing in terms of speed. This is due to use of specialized hardware designed for massive parallelization. This makes GPGPU ideal for video processing, because it enables parallelization of these operations, thus significantly speedup of processing time. In case of video stabilization, a set of operations is performed repeatedly on all pixels of each image. However, in some cases, especially on hardware of the first generation, there is a problem with IEEE 754 compliance, where the precision of some operations is slightly less than required by this standard [27]. Another issue is performance penalty of double precision computations – while single precision is fast, double precision is typically two to four times slower. Therefore, it is important have this in mind during design and development of applications for GPU, as development costs could possibly outweigh final speedup.

However, also other issues exist – beside slow access to GPU memory (about 700 clock cycles), due to current hardware design, branching sensitivity exists. This is a hardware limitation (compromise between speed and universality). Threads are therefore organized into groups of 32 in case of NVIDIA [27] and 64 in case of AMD [28]. As a result, threads in the same groups should all take the same branch, otherwise performance penalty will occur – threads will be required to compute all possible branches (serialize different execution paths), but only results of valid ones will be stored [27] [28].

Currently, there are three main programming frameworks for GPGPU available: OpenCL, CUDA and C++ AMP. All of them have some pros and cons and will be discussed below.

4.1 OpenCL

OpenCL [28] is an open source framework currently developed by Khronos group (initial creator of framework is Apple). Khronos group is a consortium created by the biggest world technology companies (AMD, ARM, Intel, IBM, NVIDIA, Apple and others.) in order to define standards. OpenCL is a standard for general purpose parallel programing across CPUs, GPUs and other processors. Its creation was a reaction to proprietary CUDA framework by NVIDIA. OpenCL framework utilizes a subset of ISO C99 and adds additional extensions in order to support parallelization. It can be also used on handheld and embedded devices. Efficient interoperation with other Khrono's products like OpenGL is also an asset. Typical OpenCL device consists of multiple compute units. Functions that are executed on OpenCL device are called kernels [28].

First version 1.0 was released in August 2009 [29]. However, it had a lot of drawbacks. Therefore, shortly after that, next version 1.1 was released in June 2010 [30], which added multiple features (e.g. new data types, operations on regions, advanced events). In November 2011, version 1.2 was released [31]. It brought ability to compile OpenCL code into external libraries and possibility to use specialized hardware features in kernels. As of November 2013, the latest stable version is 2.0, which added support for some of the ISO C++11 features and nested parallelism [32]. Version 2.1 is currently in development [33], one of biggest changes is replacement of OpenCL C with ISO C++14 subset. However, only part of companies in Khronos group support the latest standard version (e.g. NVIDIA supports OpenCL 1.2 as of April 2015 [34], with more than one year delay, because of its primary focus on CUDA, whereas AMD already supports version 2.0 from September 2014 [35]), which degrades its advantage in multiplatform usage.

Standard defines APIs only for C and C++, but third-party APIs for Python, Java and .NET also exist. Also, commonly used libraries are freely available [36]:

- *clBLAS* – for basic linear algebra subroutines (implementation of BLAS specification).
- *clSparse* – for matrix and vector operations.
- *clRNG* – for high performance random number generation.
- *clFFT* – for fast Fourier transformations.

Also, an extension for web – WebCL exists from March 2014 [37]. It is a JavaScrip binding to OpenCL and enables performing a complicated calculations on host device. No plugins on host device are required, only a compatible browser.

Programming in OpenCL consists of two parts. First, one must use API functions to initialize compute unit and schedule work. In the second part, a standalone file with extension .cl must be created. This file contains all functions that will be executed on OpenCL device. These functions are called in actual source code of application through OpenCL API, where actual function to be executed is send as a string in appropriate function call [28].

```
1. #pragma OPENCL EXTENSION cl_khr_byte_addressable_store : enable
2. __kernel void vectorAdd(__global float *A, __global float *B, __global
   float *S, __const unsigned int n) {
3.     size_t id = get_global_id(0);
4.     if (i < n)
5.         S[id] = A[id] + B[id];
6. }
7.
8. kernel.setArg(0, a);
9. kernel.setArg(1, b);
10. kernel.setArg(2, s);
11. kernel.setArg(3, sizeof(size_t), &n);
12. queue.enqueueWriteBuffer(a, CL_TRUE, 0, n, A, 0, 0);
13. queue.enqueueWriteBuffer(b, CL_TRUE, 0, n, B, 0, 0);
14. queue.enqueueNDRangeKernel(kernel, cl::NullRange, cl::NDRange(n),
    cl::NDRange(1, 1), NULL, &event);
15. queue.enqueueReadBuffer(s, CL_TRUE,  0, n, S);
```

Figure 4.2: Vector addition in OpenCL.

4.2 CUDA

CUDA is a proprietary framework developed and supported by company NVIDIA which specializes in gaming graphics cards development. It works only with NVIDIA GPUs. This is both an advantage (faster development cycle is possible) and a disadvantage (GPUs from other vendors are not supported). Because of this, NVIDIA focuses primarily on CUDA support and OpenCL is secondary.

The development has begun together with a GPU generation called Tesla. This GPU architecture was a revolutionary, as it implemented support for Direct3D 10, which required a new type of computing units – geometry shader for complex geometry generation on graphics hardware. With this, a total of three types of compute units was required to be present on GPU (after vertex shader and pixel shader) [38]. However, the problem with using these specialized units was that, depending on a rendered scene, some of these units could be used heavily, while others could be used only partially or not at all. This was a huge performance bottleneck. Also, adding new type of compute units would not bring performance gain, but enlarge the area of the chip itself instead. Therefore, NVIDIA's reacted with introduction of new universal units, which could be used as each of these required units. Its allocation became automatic and the bottleneck was removed. At the same time, engineers in NVIDIA realized that this programmability of units can be further utilized, and brought an idea of general purpose computing on GPUs [22]. CUDA was introduced together with these new graphic cards in November 2006. However, public availability was delayed until February 2007 [25], when also graphic cards optimized for GPGPU computations were released.

From its initial release, multiple versions have been released, with support of new features every time. Most of the releases took place shortly after introduction of new GPU architectures, which also meant bigger changes. Main features added over time are: shared memory, unified memory, libraries for GPU code, GPU code precompilation, new

work spawning from within GPU code or C++11 features like lambdas or auto type specifiers. Currently, version 7.5 has been available since September 2015. All versions are backward compatible. Forward compatibility is guaranteed on binary level. However, the new features are often limited to new hardware [39] [40] [41] [42].

CUDA exists in two programing languages:

- CUDA C/C++ based on subset of ISO C11 and ISO C++11
- CUDA Fortran based on ISO Fortran

For each, a separate compiler is provided. Also, third-party wrappers for other languages exist (e.g. Python, Perl, Java, Ruby, .NET, R, MATLAB) [43].

A big advantage of CUDA over other GPGPU frameworks is the availability infrastructure – a great collection of powerful tools and libraries exists [43]. One example is an advanced IDE with a debugger and a profiler or a powerful set of plugins for Visual Studio. Another example are official free libraries, which contain additional tools beside OpenCL's equivalents. Examples of libraries available for CUDA over OpenCL are:

- *NPP* – a library focused on image and video processing. It contains hundreds of signal processing primitives, therefore prevents unnecessary copying of results from device memory to host memory. It is extensively used by CUDA version of OpenCV.
- *Thrust* – library similar to C++ Standard Template Library (*STL*). It automatically chooses between CPU and GPU code execution at compile time.

These libraries do not only simplify development of fast parallel code, but also reduce some overhead that arises when users choose to implement this functionality manually.

CUDA, as opposed to OpenCL, does not require separate files for GPU execution. CUDA is also ahead of OpenCL by supporting not only direct access to other CUDA devices memory on local machine through PCI-e lanes, but also a remote access between GPUs through network which makes it suitable for use in servers. The unified memory model is also an advantage – it reduces complexity of code by viewing both PC's RAM and GPU's RAM as a whole, thus using a single pointer in both CPU and GPU. Also, like OpenCL, ARM platform is also supported. Dynamic allocation of memory and assembly-level optimizations are a must [41] [42].

```
1. __global__ void vectorAdd(float *A, float *B, float *C, int n) {
2.     int i = blockDim.x * blockIdx.x + threadIdx.x;
3.     if (i < n)
4.         C[i] = A[i] + B[i];
5. }
6.
7. vectorAdd <<< blocksPerGrid, threadsPerBlock >>>(A, B, S, n);
8. cudaDeviceSynchronize();
```

Figure 4.3: Vector addition in CUDA.

4.3 C++ AMP

C++ AMP is a compiler and programming model extension to C++ that provides an easy way to write programs that execute on data-parallel hardware, such as graphics cards [44]. It is developed and maintained by Microsoft, and its specification is open. First version 1.0 released in August 2012 was implemented on DirectX 11. However, this tight coupling disallowed usage outside of Windows environment. Therefore, in December 2013, a new version 1.2 was released, which removed those dependencies and made the library multiplatform. Due to its open specification, C++ AMP was developed with comments and suggestion from AMD and NVIDIA. In some cases, it is used as a basis in AMD's libraries for OpenCL. Also, a compiler from C++ AMP to OpenCL has been available from November 2013 [45].

```
1. array_view<const float, 1> a(n, A);
2. array_view<const float, 1> b(n, B);
3. array_view<float, 1> sum(n, S);
4. sum.discard_data();
5.
6. parallel_for_each(sum.extent, [=](index<1> i) restrict(amp) {
7.     sum[i] = a[i] + b[i];
8. });
```

Figure 4.4: Vector addition in C++ AMP.

C++ AMP adds new keyword *restrict*, whose purpose is to enable usage of certain functions or types only on specific type of hardware (CPU or GPU). Usage of restriction for GPU enables static code checking against unsupported types (e.g. char, long long or long double). This is due to limited type support on current GPU hardware. The group of supported types is referred to as an *amp-compatible*). Also, an intermediate representation of floating point expression shall not use higher precision than the operands demand (e.g. sum of two floats cannot be performed using double precision and then converted back to float). In case of compound types, pointers to pointers are not supported. Additionally, local variables can be stored in registers or in memory shared by thread group, referred as *tile_static*. Further, a math library is also a part of the specification [44]. Also, another libraries, with the same functionality like those for OpenCL and CUDA are available: *STL-style* algorithms, *RNG*, *FFT*, *BLAS* and *Linear algebra*.

The debugging and the profiling, the same tools that are available in Visual Studio for C++ development are also available for C++ AMP.

4.4 Chapter summary

Choice of the right platform depends on many parameters. For real production use, CUDA seems to outweigh OpenCL in these of criteria: simple integration into C++ projects, advanced debugger and profiler, high range of optimized libraries, quicker feature availability (in some instances two years before AMD in case of tools an libraries, or before OpenCL in case of language features) and extensive support for developers.

Ability to mix both device code and host code in the same source code file is also an advantage. However, main disadvantage is limitation to NVIDIA graphic cards in case of PCs and lower support for mobile devices. On the other hand, OpenCL provides a wider range of platforms (all PC graphics cards, range of mobile and low power GPUs and CPU architectures, including Intel's x86, IBM's Power and ARM) and standardized API.

The third and the youngest API – C++ AMP is somewhere in-between CUDA and OpenCL. From one point of view, its look is closer to ANSI C++11 and both device and host code are mixed. Also, static code checking and availability of debugging tools in Visual Studio eases work for developers and does not require them to learn new, radically different languages for GPGPU. On the other hand, there are only handful of libraries for C++ AMP and developer community is significantly lower. Also, like CUDA, amount of supported hardware is nowhere near OpenCL.

A code sample for the same operation (vector addition) was shown for all three frameworks, in order to better illustrate differences between them. A full code for vector addition and comparison to CPU parallel code can be found APPENDIX A: OpenCL, CUDA and C++ AMP in comparison with C++ OpenMP.

5 ALGORITHM DESIGN

In order to implement a suitable stabilization algorithm, the worst case scenario of input video and properties must be defined first:

- Shaking up to the frequency of 20 Hz
- Shaking up to 0.5° variation
- Focal length up to $F = 200\ mm$
- FullHD input resolution
- Fast processing

The choice of $F = 200\ mm$ is the result of the focal length commonly available in consumer compact cameras. While nowadays lot of ultra-zooms is available with focal lengths up to $F = 2000\ mm$, the video stabilization would not be possible at this focal length due to the relation shown in Figure 3.6. Therefore, the more common value that covers the most used focal lengths was chosen (besides the compact cameras also the DLSRs where only smaller focal lengths are commonly available and mobile devices with typical $F = 28\ mm$.

5.1 Preprocessing

In order to lower computing power requirements, a preprocessing step is introduced. Its goal is to retrieve areas of concern and prepare them for the motion estimation phase.

5.1.1 Areas of concern selection

We decided to split each frame into eight areas of concern (searching sub-windows) around its edges. The center rectangle is not considered, as typically an object of interest is present in the central part of each frame. This is optimization of both required computational power and to improve estimation of local and global motion vectors, as object of interest can perform movement independent of camera's movement. If not considered, this would bring unwanted error to global motion vector. They have rectangular shape proportional to the resolution of input frame and are equally distributed (see Figure 5.1). Because of the nature of hand tremor (see section 2.2) and assuming that the input video has at least 24 frames per second, the distance from the frame's edge can be 5 % of resolution or less even for high focal lengths (e.g. $F = 200mm$).

Then, matching sub-windows (MSWs) sizes must be small enough to bring significant savings in required power and big enough to have sufficient amount of details for searching in searching sub-window (SSW) of the previous frame. This values can be computed from the equations 3.1, 3.2 and 3.3: for the 0.5 degrees variation, the search windows size should be 4.9 % of resolution for each axis. Because the change can be for both directions, the double of this value is required (9.8 %). Therefore, a value of 10 % of frame resolution was chosen for size of SSW. This together with 5 % distance from

the edges leaves 60 % of unused space (30 % of resolution for each space between SSWs). However, because the proposed method is not developed exclusively for stabilization of videos from hand-held devices, this dimensions can be adjusted based on different usage scenarios in order to better fit required application. This adjustment results into smaller error and/or to lower required computer power (e.g. in the video shot from fixed camera inside car, the shaking occurs mostly in y-axis and therefore width of SSWs can be reduced).

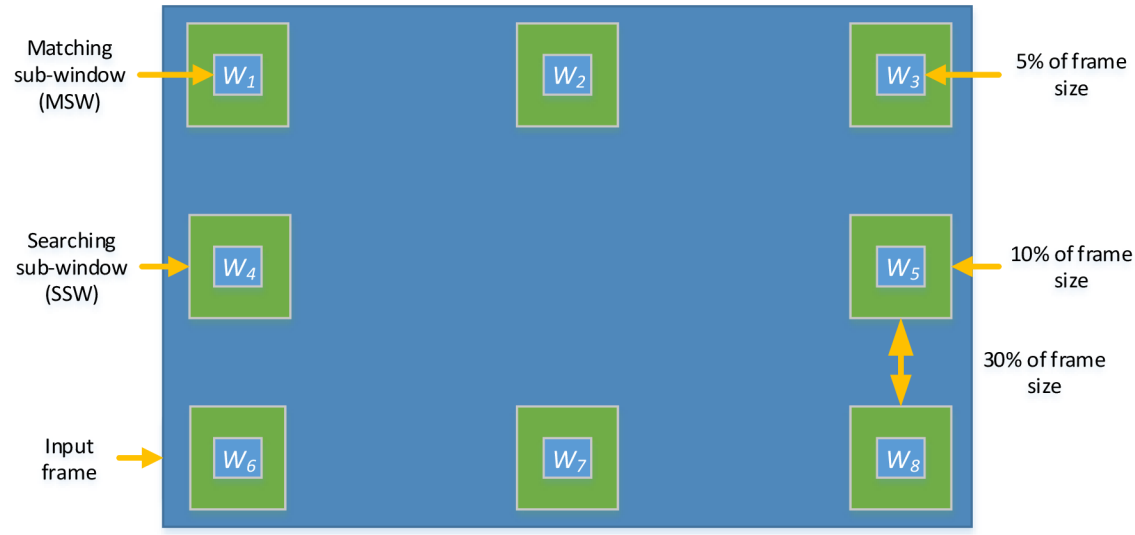


Figure 5.1: Areas of concern selection.

As for the matching sub-windows size, the same as for the distance from frame's edge applies. This results in a half of dimensions of search sub-windows.

However, in order to speed up GPU computation, a coalesced access to GPU's memory must be ensured and unnecessary branching prevented. Therefore, matching sub-windows' dimensions must be rounded to the multiple of 32. This also applies to searching sub-windows: their dimensions must be updated to the double of those of matching sub-windows. The final values used for stabilization on GPU are in Table 5.1.

Table 5.1: Input video resolutions and resulting SSW and MSW sizes.

Input resolution [px]	SSW size [px]	MSW size [px]
1920×1080	192×128	96×64
1280×720	128×96	64×48

After the previous step, eight areas of concern exist. However, adjustments must be made in order to further lower the computation requirements. For color images, the first step is conversion to grayscale. Then binarization process follows: each area is converted from full-bit frame into binary image by local binary pattern (LBP), which enables template matching by simple XOR operation. However, conversion to binary image itself is tricky – a high level of detail must be preserved after binarization step (traditional methods tend to convert similar colors into the same binary value and therefore omit edges).

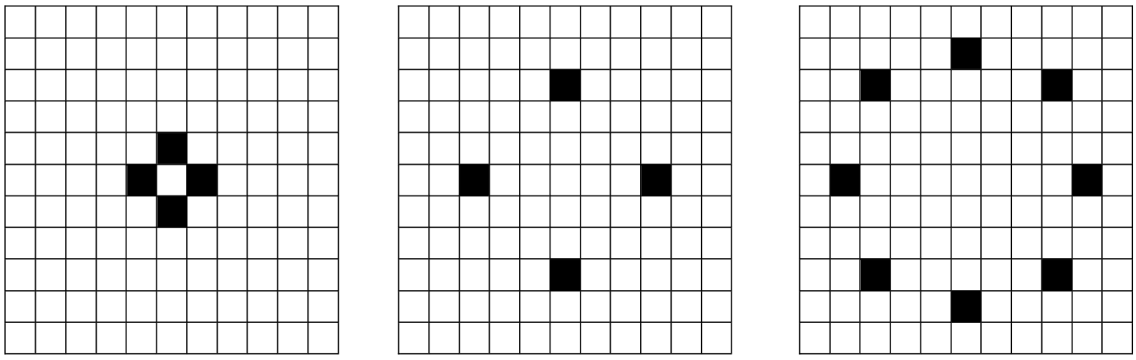


Figure 5.2: Different local binary patterns configurations. a) $LBP_{(4,1)}$ b) $LBP_{(4,3)}$ c) $LBP_{(8,4)}$

Therefore, enhanced LBP binarization proposed by [19] is used: each pixel of input image is compared against P equally spaced reference pixels (points) forming a circle of a radius R. Output value for each pixel of the output image is then computed as:

$$B_{(P,R)}(i,j) = \begin{cases} 1 & if \sum_{p=0}^{P-1} sign\big(I(p) - I(i,j)\big) \geq \lfloor P/2 \rfloor \\ 0 & otherwise \end{cases} \tag{5.1}$$

and

$$sign(x) = \begin{cases} 1 & x \geq 0 \\ 0 & otherwise \end{cases} \tag{5.2}$$

where P is the count of reference points, R is their radius, (i,j) is the coordinate of currently processed pixel, p is the coordinate of current reference point, I is the function returning image's intensity value for given coordinate and $\lfloor x \rfloor$ denotes the largest integer not greater than x.

This approach reduces the maximum number of comparisons and additions to obtain pixel value to P.

The proposed algorithm uses the $LBP_{4,2}$ configuration, as it is significantly faster than configurations with bigger number of reference pixels. Also, the empirical testing has shown, that this configuration yields more accurate binary representation of edges and therefore enables better local motion estimation.

5.2 Local motion estimation

After the preprocessing step, local motion estimation follows. Firstly, matching sub-windows are extracted from the binarized areas. After that, comparison of all MSWs of current frame with the corresponding SSWs of previous frame is performed by computing number of non-matching points (NNMP) [19] criteria for each possible displacement:

$$NNMP(dx,dy) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \{B^t(i,j) \oplus B^{t-1}(i+dx, j+dy)\} \tag{5.3}$$

and

$$-s \leq (dx, dy) \leq s \tag{5.4}$$

where (dx, dy) is the candidate displacement of the matching sub-window in the searching sub-window, N is the MSW's dimension ($N \times N$), B^t is MSW of current frame, B^{t-1} denotes SSW of previous frame, $\oplus$ represents Boolean operation XOR and s is half of the difference of the matching sub-windows and searching sub-windows dimensions.

This results into eight matrices of NNMP values, where each value's index denotes (dx, dy). From each list, the eight lowest values are taken and their coordinates become the local motion vectors (LMVs). This is an improvement suggested by [1] in order to enable stabilization of frames without clear edges (e.g. desert, sea, snow). This gives in total of 64 LMVs vectors.

5.3 Global motion estimation and filtering

Global motion vector (GMV) is computed for each frame as a median of all axes of 64 best LMVs:

$$GMV_a = median(LMVs_a), \qquad a = \{x, y\} \tag{5.5}$$

where a denotes the axes of LMV, $LMVs$ is set of selected local motion vectors and $median(x)$ is the function selecting median value

This filtration effectively removes LMVs, into which an error was introduced by object of interest movement extended into the searching sub-windows. Also, it automatically account for camera movement in z axis (e.g. camera mounted on moving vehicle) and no additional filtering is needed.

5.4 Motion compensation

The last step in digital image stabilization is the movement of image frames into final form of stabilized output. Because one of the requirements is possibility of real-time usage, a standard form of DIS is chosen. Therefore, the final resolution of video output is defined by the size of moving windows which is estimated according to equation 3.1.

Yet, this approach is able to achieve only stabilization in fixed position – it is unable to track intended movement of the camera. Therefore, the filtration of intended motion must be introduced. For this tasks, multiple possibilities exists: low pas filtering methods (e.g. fuzzy filtering or Gaussian weighting). Another example is the Particle filter, which is suitable for filtering of non-linear motion of camera, but relatively slow and not suitable for real-time processing. Therefore, we choose to use the Kalman filter, which is optimal filter in the minimum variance sense. Its advantage is the speed, which is suitable for real-time processing, but it is sensitive to parameter values.

6 IMPLEMENTATION

The designed algorithm was implemented using C++11 and OpenCV [46] library in version 3.1 under Windows environment (compilation of code under other environments is also possible). The project is created in Visual Studio 2013 [47]. The reason behind this choice is the Visual Studio's support for C++ AMP and also the great integration of Visual Studio and NVIDIA's profiler and debugger. The main application is created in single project resulting into single executable and requires the OpenCV's modules (enclosed at the attachment DVD). The application itself is command-line based and does not contain graphical user interface. For the configuration, following arguments can be specified:

- `-ocl` – enables the video stabilization using OpenCL.
- `-cuda` – enables the video stabilization using CUDA.
- `-amp` – enables the video stabilization using C++ AMP.
- `-cpu` – enables the video stabilization using CPU.
- `-write` – writes stabilized video into output file.
- `-show` – shows stabilized video during processing.
- `-input filename` – sets the input video file.
- `-limit n` – sets the number of frames to process.
- `-h` – shows help.

However, due to the performance reasons, the specification of the SSW size is not possible at runtime. This is due to the better optimization of code, when the SSW and MSW sizes are known at runtime (performance penalty is about 10 %). Therefore, windows sizes are set as a macros and after change, the recompilation is required. The enclosed DVD contains all of the binaries used for testing (binary's name contains SSW size).

The compliance of all implementations in terms of identical results for the same input was ensured by creating reference version for execution on CPU. Then, each GPGPU version results for transition of each frame were compared with this reference implementation.

Also, all GPGPU versions were created with support for asynchronous data copy to and from device. This (if supported), causes that data can be copied during kernel execution, that further improves the execution speed, as GPU does not have to wait for new data to arrive. Further, the each area of concern is processed on separate stream. Similarly to data copy, this can speed up the execution process as hardware can launch next scheduled kernels if the current kernel is not demanding enough (if supported by both hardware and software).

Because the part of this thesis's aim is to compare the suitability of available GPGPU architectures for video stabilization, the following subsections will contain both the highlights and downs encountered during the application development and author's opinions. It is important to note, that the author has previous experience with both CUDA

and OpenCL and therefore knows principles of programming massively parallel processors.

6.1 CPU implementation

The CPU version was created in order to have reference point for comparison in terms of both performance and identity of results. Therefore, this implementation was created first and was used as a reference for GPU implementations. The performance was optimized using preallocated data structures. This reduces the overhead created by repeated allocation and deallocation of memory. Also, whenever possible, the OpenCV optimized methods were used. However, no further optimizations on instruction level were introduced.

Further, the parallelized version using OpenMP [48] was created. It enables work splitting between both different cores and processors and therefore the maximal usage of CPU resources. This can be considered as a good measure of the worth of GPU usage against the fully utilized CPU.

6.2 CUDA implementation

The OpenCV contains support for CUDA execution in large portion of methods available for CPU execution. However, CUDA is not used by default and requires programmer to explicitly state calls for CUDA powered methods. This methods resides in separate assemblies that are optional during build of OpenCV's binaries.

For CUDA, a small framework in OpenCV exists. It is encapsulated in the `"GpuMat"` class, very similar to the standard `"Mat"` class. However, it can be used only on GPU and enables simple memory allocation and data copy from `"Mat"` data structure allocated for CPU without any knowledge how to allocate memory and transfer data in CUDA. The advantage of this framework is quick creation of subimage in the same manner as in case of standard `"Mat"` class. Also, because the CUDA supports C++11, similar wrapper inside kernels is available for data access. This enables to directly access pixel of image specified by x and y positions and programmer does not have to worry about the conversion from required points to indices of one dimensional array. Yet, the pointer to the data is still available and can be used. The wrapper also provides the usable size of image and the size of the row for manual indices computation.

In the implementation itself, the similar preallocation as in case of CPU implementation was introduced. However, the memory had to be preallocated on both CPU and GPU. For this case, the so-called `"pinned memory"` available in the Thrust library is used. This improves the transfer speeds between CPU's RAM and GPU's RAM by a factor of 2 by preventing CPU allocated memory to be swapped out.

The LBP kernel was implemented in two versions: one using only global memory acces and second one, where the used data were firstly copied into shared memory and the the computation was made. This reduced the count of global memory reads from 5

down to 2. However, this approach requires synchronization using the memory barrier. The perliminary tesing has shown, that the version with only global data acces is about twice as fast as the version with data copy in shared memory. Therefore, the global data access version will be used in further tests.

Because of the perliminary performance testing of LBP kernel, the NNMP kernel was implemented only with global data access. The shared memory was then used for local storage of intermediate NNMP values and the reduction in order to get the final NNMP values together with local motion vectors.

The sorting of resulting NNMP values is required prior to the selection of best local motion vectors. The sorting of NNMP values for each area of concern was done also by GPU, as the data transfer of only 48 bytes is required if the data are sorted (data are stored in triples of `"short"` data type – 16 bytes long, 8 best vectors is required), while CPU sorting (even it is more effective) would require the transfer of six times more data that is the size of single MSW, which would introduce further overhead. In this case, the Thrust library's Radix-sort is used.

The Thrust library [49] is the parallel algorithms library for CUDA, which resembles the C++ Standard Template Library (STL). It implements lot of functions available in STL equal in terms of functionality (sorting, reductions, transformations, iterators). In lot of functions, the direct data copy from/to STL's `"vector"` class is available.

The kernels for RGB to grayscale conversions and LBP binarization are scheduled in a way that each pixel is processed by single thread. Yet, they are organized into smaller groups for more effective access into global memory in order to prevent bank conflicts (multiple threads accessing single memory bank or alternating access to banks) within warp (otherwise, the serialization of access would be introduced). The size this groups is computed automatically by querying the hardware property returning max number of threads that can be executed in single group (in most cases, the value is 1024) and by using the NVIDIA's occupancy calculator, that is able to suggest the best work group size based on the used hardware and the kernel itself (the number of used registers and size of used shared memory).

The kernel computing the NNMP values uses similarly to other kernels maximal possible count of work items in single group. For performance reasons, each thread computes the small part of NNMP for the corresponding translation (the computation of the whole NNMP on single thread would be enormously long and would result into big number unused threads). Because the size of MSW exceeds the typical number of work items in single group (1024) even for small sizes, the optimization is introduced, where number of work items is reduced by factor of 32. Then, each thread computes the NNMP values for 32 pixels of correlation. However, in order to get the resulting NNMP value, parallel reduction is required. This results into small overhead, when only portion of threads is used for computation. In order to improve performance, the reduction is implemented in a way, that all threads in single warp performs the same instructions. Yet,

reduction itself requires the explicit synchronization of threads which results into another delay in execution.

While the resulting code itself is pretty simple, the way to get to know how to create optimized application from available libraries is pretty hard, as the available documentation contains only basics of how to get the library running. This enables to create working video stabilization, but the performance gain over CPU implementation cannot be considered as good. The more advanced functions have only autogenerated documentation that does not say anything about the provided methods. Therefore, the experimentation and questions browsing on specialized sites like Stack Overflow [50] is required. This apply for both the OpenCV and Thrust. On the contrary, the documentation for the CUDA itself is great and covers all features available in the framework. Yet, the problem with compiler was encountered, where the MAD instruction was not correctly recognized, which resulted into degraded performance. Therefore, this instruction was forced using the PTX assembly language [51]. This also apply for OpenCV wrapper to access image pixel of specified position – for this reason, wrapper was not used and conversion from points to indices was computed separately.

6.3 OpenCL implementation

The OpenCV's support for OpenCL is from the version 3 implicit and only requires to call function to enable it and to use the `"UMat"` class (universal `"Mat"`), which is similar to standard `"Mat"`. Then, all functions on `"UMat"` that have OpenCL version will run on GPU if possible. No additional code changes are required. Unlike `"GpuMat"` for CUDA, the `"UMat"` can be processed on both CPU and GPU (the data copy is automatic). Similarly to `"GpuMat"`, most of the functionality of standard `"Mat"` class is available.

However, the OpenCV's documentation does not state how to create and lunch custom OpenCV kernels. While some methods can be found in documentation, the OpenCV source code must be examined for thorough understanding. The OpenCV contains wrappers to OpenCL API that greatly reduces the size of code required for initialization. The most problematic parts were creation of unique command queue for each area of concern and the kernel launch itself: the queues required creation in cycle instead of simple list initialization and the kernel launch had to be called directly through OpenCL API call `"clEnqueueNDRangeKernel"`. However, the use of OpenCV's wrapper was still advantageous, as it enabled both the easy kernel compilation and argument setting (memory allocation). Due to the compatibility reason with NVIDIA (OpenCL support of version 1.2), the OpenCL version 2.0 with precompiled GPU code was not used. The disadvantage of this solution is visibility of source code to the end user in the plain text form. While this can be solved by means of encryption, the compilation as in case of NVIDIA would be more suitable (however, this brings compatibility issues between different device vendors).

Another problem is that the OpenCL 1.2 does not support C++11 features. Therefore, if the support of NVIDIA's cards is required, the C-like language must be

used. This means, that the kernels must be defined in separate file from the code, there is no support for include directives in the GPU code and therefore no wrapper similar to the CUDA version. From this point, the kernels definition requires to explicitly state all input arguments (pointers, data sizes) manually. Therefore, conversion from data structures used for kernel arguments setting on CPU side must be performed. The problem is, that this behavior is nowhere described and can only be deducted from the OpenCV's source code examination. This also means, that no wrapper functions are available for accessing the image pixels by coordinates and their conversion to one dimensional array indices must be computed manually.

The RGB to grayscale, LBP and NNMP algorithms itself are equal to the CUDA implementations, the only difference is in the input parameters, that are not wrapped. This is also true for the size of single work group. However, there is a difference between the NVIDIA and AMD – the typical workgroup size in case of NVIDIA is the same as in for the CUDA (1024), but the AMD has the limit of 256 threads. Therefore, in case of NNMP kernel, the larger portion of computations must be made on single thread.

The next difference from CUDA is the usage of memory barrier for synchronization. In CUDA, the barrier can be used inside divergent branches, the only requirement is that threads inside warp all takes the same execution path. In case of NVIDIA cards, this is also true for OpenCL, despite the fact, that the OpenCL specification itself prohibits this behavior. In case of AMD, the standard is respected and such code is rejected by compiler. The most interesting is, that on NVIDIA's cards, there is measurable performance difference between these two variants.

Similarly to CUDA compiler, also the OpenCL compiler failed to correctly recognize the MAD instruction and it has to be defined explicitly. However, the OpenCL does not support the assembly language snippets insertion into kernels. Instead, some of the instructions are available as an API functions.

The sorting of NNMPs is performed on GPU similarly do CUDA implementation. However, no external library is used, but rather the Bitonic sort algorithm form NVIDIA samples library.

6.4 C++ AMP

In the case of C++ AMP, there is no support for this technology in OpenCV. Therefore, the programmer must take care of all data conversions and copy implicitly. Yet, the amount of required work is significantly smaller when compared with the amount of work required when using both OpenCL and CUDA without OpenCV framework. Therefore, the conversion of data from OpenCV's `"Mat"` structure into array is required. However, there are significant issues – the C++ AMP does only support 32 integer numbers and data needs to be either converted to this format or accessed on device using the bit masking. The first approach brings the performance overhead during the data transfer, as 4 times more data needs to be copied. This can be slightly reduced by usage of texture instead of standard data pointer (the GPUs contains the native support for textures). The

second approach brings the performance overhead when bit masking is performed. The preliminary testing of both methods has shown, that the use of texture memory brings slightly better results and therefore it will be used for the thorough testing.

The next issue is only one command queue – all kernels are issued on single execution queue, where also data copy occurs. Therefore, C++ AMP is unable to execute multiple kernels at once as CUDA.

Yet, there are advantages over both CUDA and OpenCL – the kernels can be launched as anonymous methods inlined into other code. Also, the direct data copy from arrays and vectors to device is supported without the need to manually allocate memory.

The used kernels are basically the same as in case of CUDA and OpenCL version – they are also partitioned into biggest possible groups and the NNMPs computation requires one thread to compute multiple values. The difference comes again with sorting, where the Bitonic sort implementation from Microsoft samples is used.

The C++ AMP has the same requirement for memory barrier as OpenCL – all threads must hit the barrier and therefore it is not possible to use it in divergent code.

7 TESTING VIDEO SEQUENCES

For testing purposes, 11 testing video sequences was created using two cameras – the compact Olympus SZ-31MR and DSLR Nikon D3100. The first one enables creation of FullHD video sequences at the speed of 60 FPS. However, the video suffer from blurred edges (even for HD video). The second one also enables creation of FullHD video, but only at the speed of 24 FPS. The sequences does not suffer from blurred edges, however the higher amount of noise is present. Both cameras use the CMOS chip susceptible to the rolling shutter effect.

The shot video sequences contains different situations: from intentional shake of handheld camera through natural shake to jitter caused by moving vehicle (camera was mounted inside car). For each situation, typically two videos that differ by used camera, focus, focal distance, resolution and FPS exits. Their brief overview is in Table 5.1. The focal distance is recalculated from real focal distance of camera into its 35 mm equivalent.

Table 7.1: Brief overview of testing video sequences.

	Name	Resolution	FPS	Focal distance	Held type	Camera
1	**Car-ride**	1280×720	30	50 mm	mounted	Olympus SZ-31MR
2	**Car-ride-2**	1920×1080	24	50 mm	mounted	Nikon D3100
3	**Car-ride-3**	1280×720	30	50 mm	mounted	Nikon D3100
4	**Car-ride-4**	1280×720	30	50 mm	mounted	Nikon D3100
5	**Jitter**	1920×1080	24	30 mm	hand-held	Nikon D3100
6	**Jitter-2**	1280×720	30	30 mm	hand-held	Nikon D3100
7	**Pan-zoom**	1920×1080	60	200 mm	hand-held	Olympus SZ-31MR
8	**Pan-zoom-2**	1920×1080	24	40 mm	hand-held	Nikon D3100
9	**Tracking**	1920×1080	60	150 mm	hand-held	Olympus SZ-31MR
10	**Walking**	1280×720	30	30 mm	walking hand-held	Olympus SZ-31MR
11	**Walking-2**	1920×1080	24	30 mm	walking hand-held	Nikon D3100

The first four video sequences, as the name suggest, are captured by the camera mounted inside car. The first one, shot by the Olympus compact camera suffers from blurred edges, changing focus (due to the autofocus) and rolling shutter effect. Two kinds of image blur are present: the one caused by the bad focus and the one caused by the high lossy compression of video. The video itself captures the driver's view on the road. During the bigger part of video, the vehicle is moving at the speed of 50 kph with occasional slowdown due to the obstacles present in the path. During the rest of the video,

the vehicle is moving at the speed of 100 kph. The road itself contains lot of potholes and undulations causing the vehicle to swing.

The second video is captured by the Nikon DLSR and does not suffer from neither changing focus nor image blur. However, the rolling shutter effect is more visible. The video captures the in-town driving with big number of hairpin bends at the average speed of 40 kph. Also, a few second scenes with the car in stationary position is present. During this period, the doors are opened and closed multiple times as the driver gets off and back on the vehicle. Therefore, there is slight movement in both axes.

The third video sequence is very similar to the second one, but has lower input resolution, which results into smaller image disturbances.

The fourth video is captured on the cart-way with the average speed of 25 kph. The road is unpaved and contains the high number of potholes which causes rapid shaking in both axes. Further, the bottom part of the image covered with the car's dashboard, which that moves in different way than the exterior. This causes that three SSWs are not usable for image stabilization. Besides that, the car's windshield is dirty, which introduces the fixed defects into the image.

The fifth video shows fixed scene and contains rapid jitter caused by hand tremor. The hand tremor is intentionally amplified to provide bigger translation between frames.

The sixth video sequence is very similar to the previous one. However, this time the jitter is caused by the turning of the ring that sets the shutter speed. Therefore, the frames of the video have different shutter speeds and the presence of the rolling shutter effect changes through the time.

The seventh video sequence contains the pan and zoom effects. It can be characterized by slow intended changes of the camera pose and zooming. The rolling shutter effect is present only during the zooming. The camera his hand-held, therefore high amount of shake is present.

The eight video is very similar to the previous one. However, this time, the intended movement of the camera is fast. Also, the rolling shutter effect is strong even when no zooming is occurring.

The ninth video can be characterized as tracking of objects. The camera is hand-held and zoomed to the $F = 200\ mm$. Similarly to the first video, the rolling shutter effect and image blur are present.

The tenth video sequence captures the walking inside hall. It has bad light conditions and contains high amount of noise. Also, there is only low count of non-distinctive edges present.

The eleventh video captures similarly to the previous one walking from the first person point of view. However, this time there are good light conditions and high amount of edges, as the scene takes place in nature. Yet, high amount of image blur is present and the edges are again not very distinctive. Also, the sake can be characterized as strong, because the person holding the camera walks downhill.

8 RESULTS

The video sequences described in chapter 7 were tested using multiple hardware configurations in order to evaluate the performance of proposed method on different GPU architectures. Also, this enabled better comparison of used GPGPU frameworks, as testing on single GPU could result into distorted results. However, not all of the hardware supports all GPGPU frameworks – the CUDA framework is no available for AMD graphic cards (see chapter 4.2). Therefore, CUDA performance could not be tested in all cases. Next, on the NVIDIA cards, two versions of OpenCL implementation were tested (as described in chapter 6.3). The tests were performed on different sizes of the searching sub-windows and corresponding matching sub-windows (their ratio was preserved). The SSW sizes were chosen that the requirement for fast GPGPU processing is preserved – as the multiples of value 32 for both x and y axis. Further, also quality of proposed video stabilization method was evaluated by both frequency analysis and visual comparison of camera path.

For performance testing, the hardware specified in Table 8.1 and Table 8.2 was used. It contains both fairly old hardware – Q6600 (2007) and new hardware – E5-2683 (2015). Also, the tested GPUs differ not only by their theoretical performance, but also designation – compute (M6000, E5-2683), games (GTX 670, R9 Fury) or both (GTX 560Ti, GTX 980Ti). In case of CPUs, all of the used can be considered as a high-end. Yet, due to the age gap, their performance differs radically. Therefore, their theoretical performance (same for double, single and integer precision) is also given in Table 8.1. The similar apply also to the used GPUs. However, the difference here is bigger, as some used cards cannot be considered as high-end (Radeon 5750, GTX 760). Therefore, their theoretical performance in single precision (the same as used integer precision) is also given in Table 8.2. Yet, this value cannot be viewed as absolute measure of GPU's performance between different GPU architectures, as they differ radically, and some of them do not provide high compute power even though the theoretical performance value is high. This is true for the GTX 670 and GTX 760 (Kepler architecture), which were designed primary for high performance in computer games and the computing was put aside. Therefore, based on the nature of computation, they can outperform the older GTX 560Ti (Fermi architecture) only in some applications.

The performance of individual GPUs depends also on the memory bandwidth between the GPU itself and GPU's RAM, because it defines the speed of how fast the GPU access the global memory data can. Therefore, this value is also given in Table 8.2. The next thing is the speed of the bus, through which the GPUs are connected with the CPU. This affects the data transfer rate between the CPU's RAM and GPU's RAM. In all tested cases, the PCIe x16 v3.0 (Peripheral Component Interconnect Express of version 3.0 with 16 data transfer lanes) was used, providing the data transfer speed of 15.75 GB/s.

Table 8.1: The CPU specification of used hardware configurations.

Configuration number	CPU name	Speed [GHz]	Cores / Threads	Performance [GFLOPS]	RAM [GB]
1	Intel Core 2 Quad Q 6600	3.21	4/4	59	6
2	Intel Core 2 Quad Q 6600	3.21	4/4	59	6
3	Intel Core i5-4670K	3.40	4/8	117	16
4	Intel Core i5-3930K	3.20	6/12	165	64
5	Intel Core i5-3930K	3.20	6/12	165	64
6	2x Intel Xeon E5-2683 v3	2.00	28/56	480	512
7	Intel i5-3770K	4.40	4/8	150	32

Table 8.2: The GPU specification of used hardware configurations.

Configuration	GPU name	Speed [GHz]	Cores	Performance [TFLOPS]	Bandwidth [GB/s]	RAM [GB]
1	AMD Radeon HD 5750	0.90	720	1.30	80	1.00
2	NVIDIA GTX 560Ti-448	0.95	448	1.70	216	1.28
3	NVIDIA GTX 760	1.10	1152	2.20	192	2.00
4	NVIDIA GTX 670	1.15	1344	2.46	192	4.00
5	AMD R9 Fury	1.00	3584	7.53	512	4.00
6	NVIDIA Quadro M6000	1.15	3072	6.07	317	12.00
7	NVIDIA GTX 980Ti	1.30	2816	5.63	384	6.00

In this chapter, both the performance of the proposed method (the execution speed) and the quality of video stabilization will be evaluated.

8.1 Performance comparison

In this section, the proposed algorithm will be benchmarked on the computer configurations described in Table 8.1 and Table 8.2. This test will consist of processing speed measurement of proposed video stabilization. It will be tested with 8 different sizes of SSW, as described in section 5.1.1, resulting into 8 tables with results. Each table contains both the speed measured in frames per second (FPS) and the utilization of GPU. The FPS is the absolute measure of processing speed and defines, whether the video can be stabilized in real-time or not.

The testing was performed on 8 different SSW sizes (see Table 8.3) for two reasons: firstly, the bigger SSW results into more compute operations and can hide the delay introduced by data transfer. Also, in case of fast GPUs, it may still provide high enough processing speeds. Secondly, the testing videos contains not only the hand video sequences shot by hand-held video camera, but also sequences from camera mounted inside moving vehicle and hand-held camera during walking. Therefore, the initial

consideration in section 5.1.1 does not apply for all and some videos may require bigger SSW sizes, while other smaller SSW sizes.

Table 8.3: SSW sizes in pixels used for testing.

128×128	192×128	256×128	192×192	256×192	256×256	384×192	384×256

It is important to note, that the used correlation has the time complexity of $O(n^2)$, therefore doubling the size of SSW means four times more operations. For this reason, it is expected to see slowdown by a factor of 4 for doubled SSW dimensions. Yet, the other factors as the complexity of other subroutines, data transfers speeds and kernel execution times also interferes and the resulting performance impact may be lower.

8.1.1 Profiling of GPGPU implementations

During the development, the preliminary testing has shown that the GPU usage varies for different GPGPU implementations of the same algorithm on the same hardware configuration. Therefore, the detailed analysis was performed: CUDA and standardized OpenCL using NVIDIA NSIGHT profiler for Visual Studio and C++ AMP using Concurrency Visualizer for Visual Studio. The profiling was due to its complexity performed only on the HW configuration 2 and only for the smallest and the biggest SSW (size of $128 \times 128\ px$ and $384 \times 256\ px$ respectively).

The Figure 8.1 shows, that the CUDA implementation executes all kernels of one stabilization cycle in 11.718 ms, which is 85.3 FPS. This is about 3.3 % slower than measured value, but the slowdown is caused by the profiler itself. The figure also shows, that all the kernels are executed on average in 3.374 ms. This would result into the 296.4 FPS. However, there is also the data copy (1.414 ms). Despite that, the framerate of 208.8 FPS would still be possible. The problem is the white space in the timeline (6.930 ms). This is due to the slow CPU processing. If we assume, that this card is used with the more powerful CPU that is able to execute required work during the kernel execution, the framerate of 208.8 FPS would be achieved with usage of 71.0 % (given the data copy would take the same amount of time). However, the GPU is waiting for CPU even between the data copy. If this was also eliminated (1.18 ms), the execution speed of 277.2 FPS and usage of 97.7 % would be achievable (given the 64 context switching operations occurs and one takes 1.2 µs).

The Figure 8.1 also shows that the each area of concern has its own execution stream (as expected). There is also clearly visible concurrent execution of small kernels – LBP and RGB to grayscale partially overlaps (see Figure 8.2 for detailed view). Also, the sorting kernels are overlapping the NNMP kernel.

The Figure 8.2 further shows, that the most time consuming kernel is the NNMP kernel (352 µs), followed by the sorting (85 µs). The RGB to gray conversion (4 µs) and LBP computation (3.5 µs) are almost negligible. It also shows, that even the kernels are already scheduled, the context switching takes 1.2 µs.

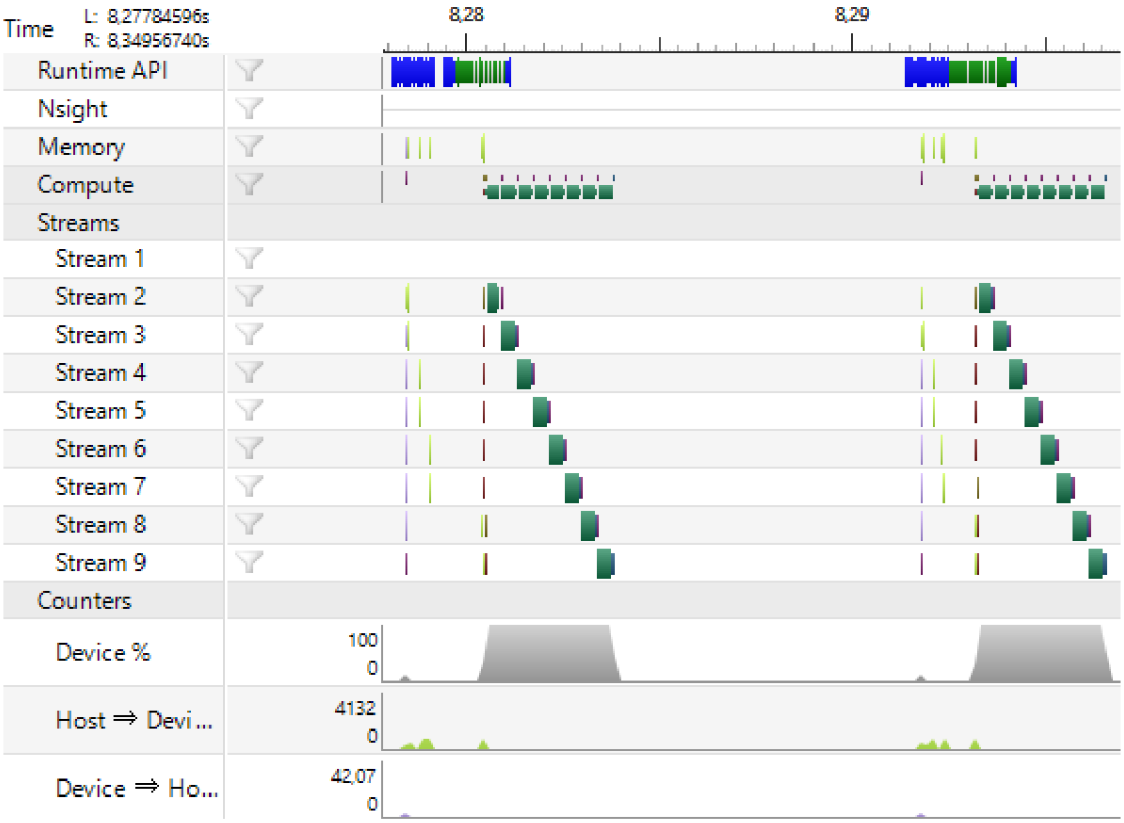


Figure 8.1: Timeline view from NVIDIA CUDA profiler on kernels execution for SSW of size 128×128 px and HW configuration 2 showing the execution order and timing of kernels for two stabilization cycles.

The last unmentioned GPU time consuming operation is data transfer: data copy to the GPU takes 15 µs for 51 kB of data (3 323.4 MB/s) and 1.15 µs from GPU for 48 B of data (39.7 MB/s).

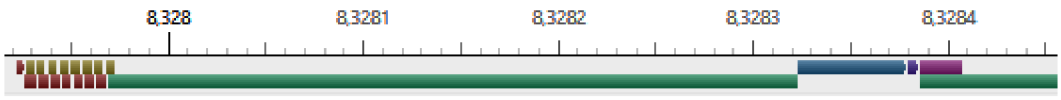


Figure 8.2: Detail of timeline view from Figure 8.1 showing the concurrent execution of kernels: red - rgb2gray kernel, brown - LBP kernel, green - NNMP kernel, blue and purple - sort kernels.

The Figure 8.3 shows, that the OpenCL implementation executes all kernels of one stabilization cycle in 13.805 ms, which is 72.4 FPS. This is about 12 % slower than measured value, but the slowdown is caused by the profiler itself. The figure also shows, that all the kernels are executed on average in 4.056 ms. This would result into the 246.6 FPS. However, there is also the data copy (4.119 ms). Despite that, the framerate of 122.3 FPS would still be possible. The problem is the white space in the timeline (5.630 ms). This is due to the slow CPU processing. If we assume, similarly to CUDA implementation, that this card is used with the more powerful CPU that is able to execute required work during the kernel execution, the framerate of 122.3 FPS would be achieved with usage of 49.7 % (given the data copy would take the same amount of time). However, the GPU is waiting for CPU even between the data copy. If this was also eliminated (4.00

ms), the execution speed of 239.5 FPS and usage of 97.0 % would be achievable (given the 88 context switching operations occurs and one takes 1.4 µs).

Figure 8.3: Timeline view from NVIDIA OpenCL profiler on kernels execution for SSW of size 128×128 px and HW configuration 2 showing the execution order and timing of kernels for two stabilization cycles.

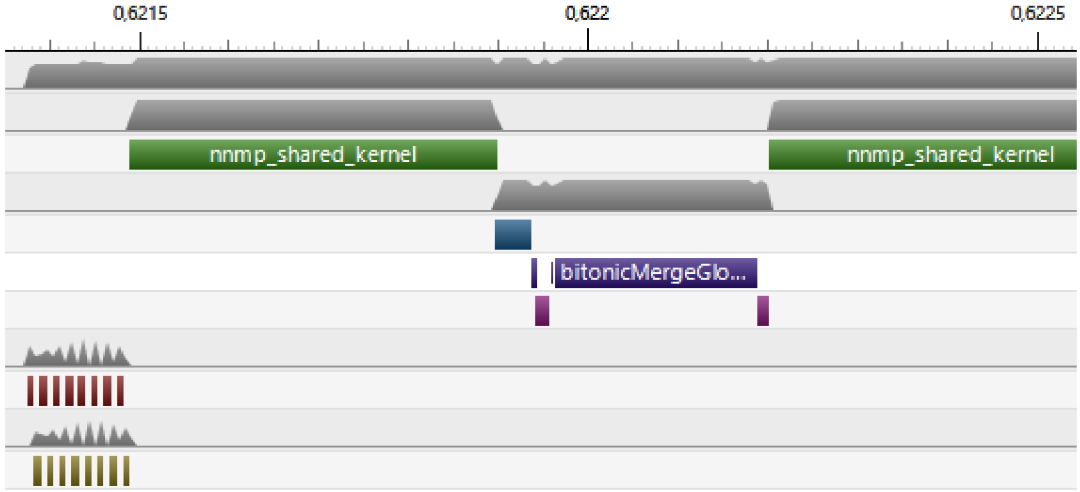


Figure 8.4: Detail of timeline view from Figure 8.3 showing the execution order of kernels: red - rgb2gray kernel, brown - LBP kernel, green - NNMP kernel, gray – GPU utilization, blue and purple - sort kernels.

The Figure 8.4 shows, that the most time consuming kernel is again the NNMP kernel (406 µs), followed by the sorting (76 µs). The RGB to gray conversion (5.8 µs) and LBP computation (5.6 µs) are again almost negligible, but almost twice as time consuming as in case of CUDA. It also shows, that even the kernels are already scheduled,

the context switching takes 1.4 µs. Unlike the CUDA, there is no concurrent execution of kernels present, as the OpenCL 1.2 specification does not support such behavior.

The last time consuming operation on GPU is data transfer: data copy to the GPU takes 14 µs for 52 kB of data (3 627.2 MB/s), which is slightly higher transfer speed than in CUDA. While this difference may seem small, it is the average value of hundreds of values and not the deviation within measure error. The data read using the OpenCV's "`UMat`" rectangle for copying only 48 bytes does not work as expected, and the entire matrix is copied instead. Therefore 24 kB of data is copied within the 7.2 µs (3 255.2 MB/s). However, at this SSW size, it cannot be considered as problem, because the removal of introduced delay would bring performance gain of only 2.8 FPS (to the estimated 239.5 FPS).

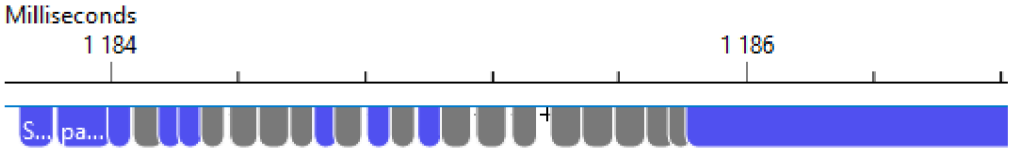


Figure 8.5: Timeline view from Visual Studio Concurrency profiler on kernels execution for SSW of size 128×128 px and HW configuration 2 showing the concurrent execution of kernels: blue – data copy and kernels execution, gray – grouped data copies and kernels, that are too small to distinguish.

The C++ AMP implementation executes all kernels of one stabilization cycle in 31.740 ms, which is the speed of 31.5 FPS. This is about 26 % slower than measured value, but the slowdown is similarly to previous implementations caused by the profiler itself. Because the C++ AMP does not enable usage of multiple execution queues, both data transfers and kernels executed in order without overlap. The problem of this solution is, that the data are transferred synchronously. While the framework officially supports asynchronous data copy, we were unable to create operational solution. However, despite that, the data copy is almost four times slower than in case of both CUDA and OpenCL – the transfer of 51 kB of data takes 51 µs (976.6 MB/s). Yet, the data copy from device is faster than in case of OpenCL, despite the bigger amount of copied data: 16 kB is copied within 2 µs (7 812.5 MB/s). Further, the kernel execution is slower: the RGB to gray conversion takes 37 µs and LBP kernel 32 µs. This is almost 10 times slower than in case of CUDA. However, the execution speed of NNMP kernel is twice as fast – execution lasts on average 180 µs. The reason behind this may be the usage of texture memory. However, further development and investigation is required to confirm this theory. The last group of kernels is used for sorting. Their execution lasts on average 250 µs, which is again slower than in case of CUDA and OpenCL (3 times). The context switching operation takes 2 µs. However, it occurs 480 times, which results into 960 µs delay introduction. With this values alone, the execution of single stabilization cycle would took 5.376 ms, resulting into theoretical speed of 186.0 FPS. However, there is another problem with the unnecessary data copies – for each NNMPS kernel, 48 kB of data is copied. This would not represent big problem, if it would not take 2.1 ms. Therefore, 16.8

ms in each cycle is for unknown reason spend by data copy back to host. If this issue was removed, the resulting performance would result in doubled execution speed.

However, the GPU waiting can be eliminated even for slower CPU by the bigger SSW size – this would cause the longer execution NNMP kernel. Therefore, also the SSW size of $384 \times 256\,px$ was profiled. As can be seen from Figure 8.6, the data copy duration takes now only small fraction of whole cycle execution time (2.65 ms to be precise). While this is twice the value of the time for SSW size of $128 \times 128\,px$, a lot bigger amount of data is being copied – 295 kB per stream, with duration of 60 µs (4 807 MB/s). The transfer of results is the same – 1.15 µs for 48 B of data (39.7 MB/s). This results into GPU waiting of 2.14 ms in each cycle (the data copy takes 505.4 µs including the context switching operations), which is still almost twice as long as in previous case. Therefore, it can be assumed, that the CPU is still not powerful enough, because the GPU is waiting even the data copying is asynchronous.

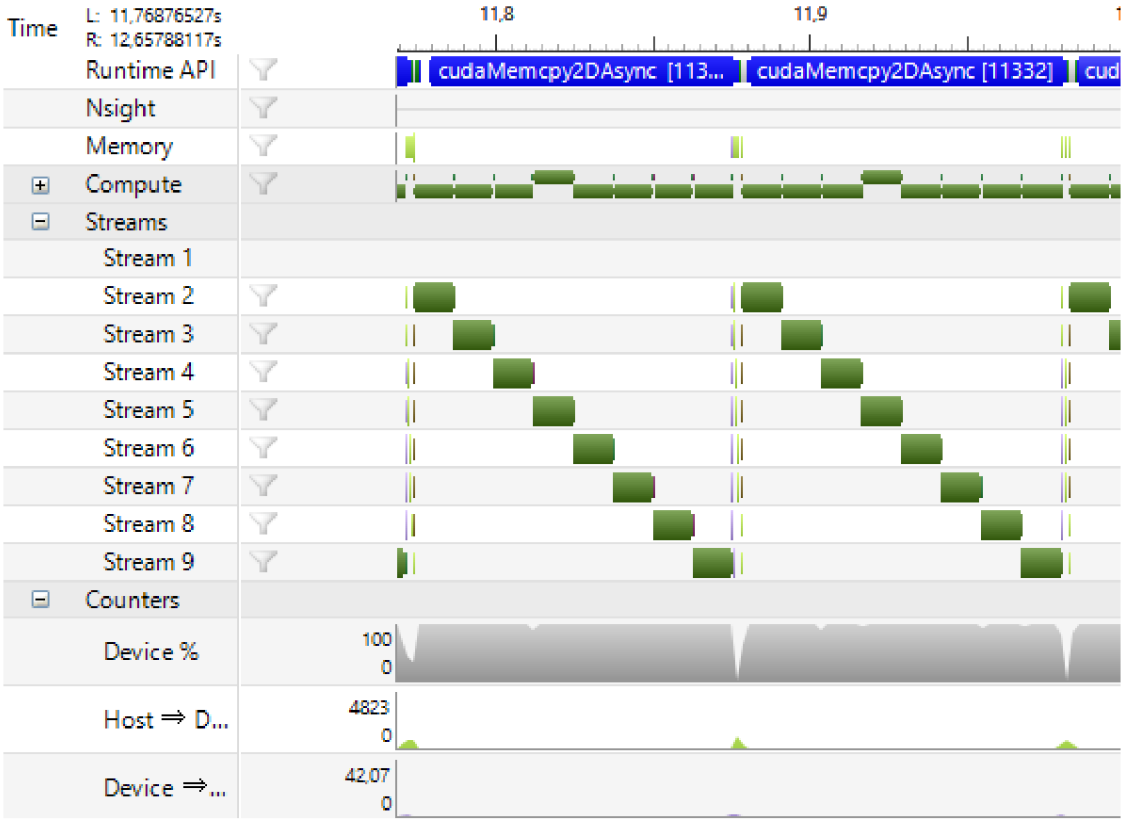


Figure 8.6: Timeline view from NVIDIA CUDA profiler on kernels execution for SSW of size 384×256 px and HW configuration 2 showing the execution order and timing of kernels for two stabilization cycles.

Therefore, the use of bigger SSW resulted into better utilization of GPU (the waiting on CPU reduced from 8.11 ms to 2.14 ms. Yet, there is still some space left for further improvements (e.g. better CPU work scheduling).

The detailed timeline view from Figure 8.7 shows, that the kernels execution is still overlapping, but with significantly smaller part. This is caused by the number of computations required for each kernel, which results into high utilization of GPU.

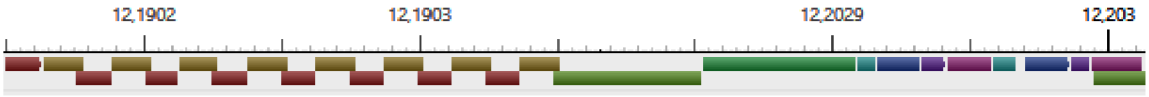


Figure 8.7: Detail of timeline view from Figure 8.6 showing the concurrent execution of kernels: red - rgb2gray kernel, brown - LBP kernel, green - NNMP kernel, other - sort kernels. The NNMP kernel was trimmed for better visualization.

Further, there is also visible bigger count of used sorting kernels – the result of NNMPS is too big to fit into shared memory at once, therefore the sorting is partitioned. The timing of kernels from the longest is as follows: 12.47 ms for NNMP kernel, 1.45 ms for sorting, 13.7 µs for LBP kernel and 11.8 µs for RGB to gray conversion. The whole cycle for this case takes 104.1 ms to execute (9.6 FPS), which is the slowdown of almost 5.9 %. The bigger slowdown is interesting, as we expected it to be smaller, because the lower amount of events is logged by the profiler. Yet, as described before, there is still some room left for improvements, and in theory, the execution speed of 9.8 FPS can be achieved (102 ms per cycle and utilization of 99.8 %) by either faster CPU or different algorithm structure. However, it would still not result into real-time execution and therefore this possible modification can be discarded.

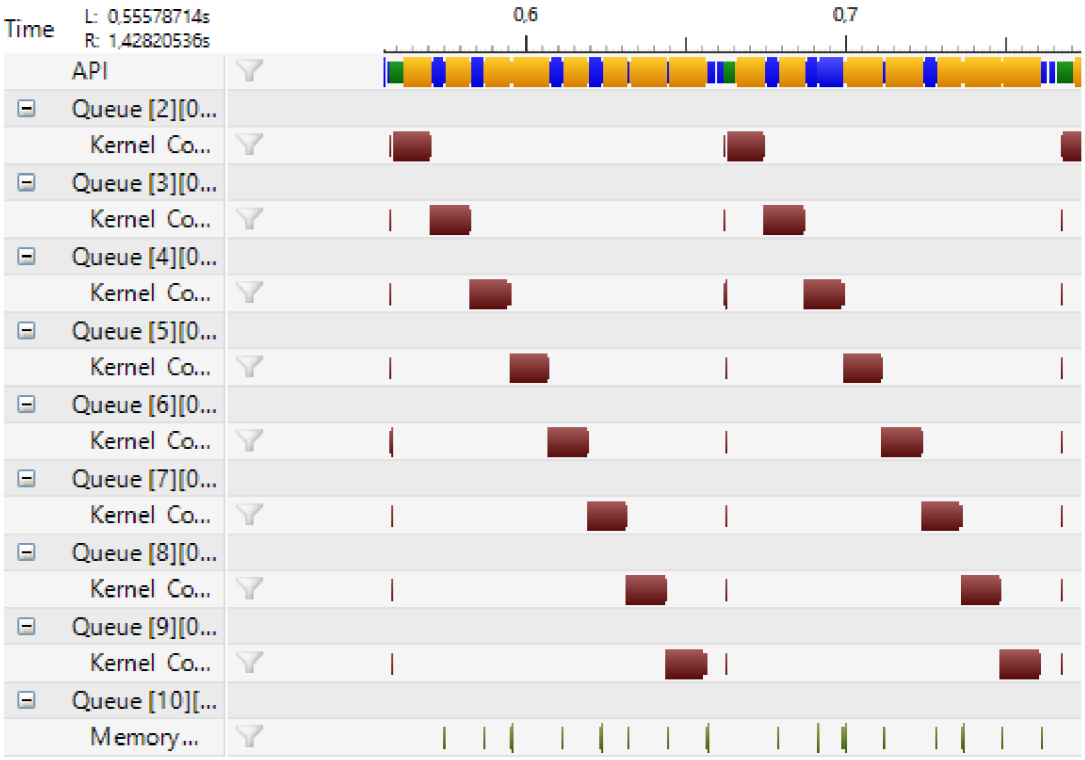


Figure 8.8: Timeline view from NVIDIA OpenCL profiler on kernels execution for SSW of size 384×256 px and HW configuration 2 showing the execution order and timing of kernels for two stabilization cycles.

The similar does apply also for OpenCL (see Figure 8.8) – the gap, where the GPU remains unutilized is greatly reduced when compared to the previous case. Yet, the data transfer takes 6.48 ms, from which the data transfer to device takes 62.2 µs (4 647.3 MB/s) for 296 kB of data (the OpenCL transfers slightly bigger amount of data, as OpenCV adds padding for faster access. For unknown reason, this is not done for CUDA). The request for result data again causes the copy of entire data block – 144 kB during 26.9 µs (5 200.2

MB/s), not only 48 B. Yet, the data copy occurs during the execution of kernels and therefore it is not a problem. However, we were unable to find out, why the data copy in CUDA was not also executed during kernel execution (both implementations use the same CPU algorithm and are designed for this behavior).

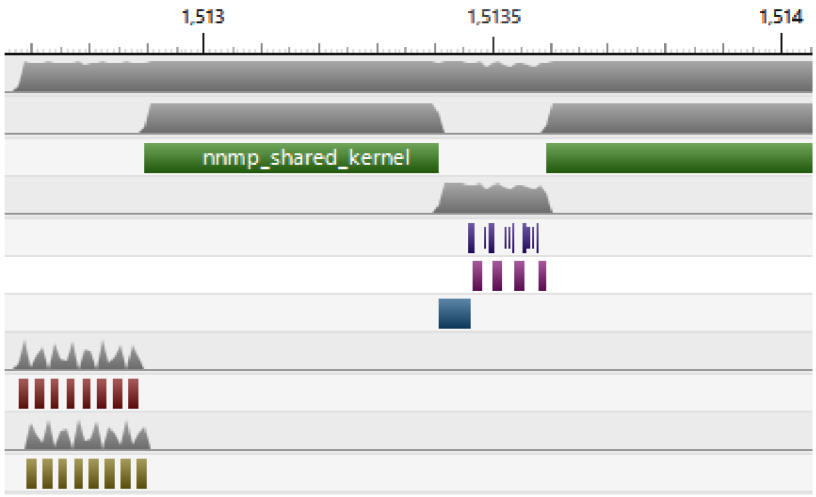


Figure 8.9 Detail of Timeline view from Figure 8.8 showing the execution order of kernels: red - rgb2gray kernel, brown - LBP kernel, green - NNMP kernel, gray – GPU utilization, other - sort kernels. The NNMP kernel was trimmed for better visualization.

The detailed timeline view in Figure 8.9 shows, that similarly to the CUDA version, more kernels is launched for sorting of results resulting into more context switching operations (again 1.4 µs per one context switch). Also, there is still no overlap of kernels which results into degraded performance over CUDA. The one execution cycle takes 108.0 ms (9.25 FPS) which results into performance degradation of 4.1 % caused by the profiler. Here, on the contrary of CUDA, the slowdown behaves as expected (the slowdown is reduced with the lower amount of evets to log). The data copy operation including the GPU waiting takes 9.3 ms, which is much bigger value than in case of CUDA. If the GPU waiting was removed, the execution speed of 9.3 FPS would be achievable, which is similarly to the CUDA implementation negligible. The order of the kernels requiring most execution time is also the same: 12.04 ms for NNMP kernel, 418 µs for sorting, 12.6 µs for LBP kernel and 11.9 µs for RGB to gray conversion. The NNMP kernel is slightly faster than its CUDA equivalent (by 3.5%). However, the sorting kernels are faster by 71.2 %. Yet, further testing is required for final decision if the Bitonic sort used in OpenCL version is faster than Radix sort used in CUDA, as the time consumption of Radix sort can be enlarged by the concurrent execution with NNMP kernel. Also the LBP kernel is slightly faster than its CUDA equivalent. This can be caused by the padding added to the data by OpenCV. Yet, the OpenCL version is still slower by almost 4 % from its CUDA equivalent, due to the CUDA's ability to run multiple kernels concurrently.

The C++ AMP implementation cycle for SSW of size $384 \times 256\ px$ is 417.9 ms long, which is the speed of 2.4 FPS (slowdown of 4.3 %). The slowdown behaves as

expected (lower with longer execution times for kernels). The execution order is due to the restrictions described before the same as for smallest tested SSW size. However, the interesting thing is the length of kernel execution times – the profiler shows, that the NNMP execution is even faster than for smaller SSW (112 µs). Yet, this is not possible, as the kernel is executing 16 times more operations. However, the closer look at other operations shows, that the data copy operation of 432 kB takes 48.67 ms, whether the same data copy operation 48 kB in 2.1 ms, which is 23 times slower. Therefore, it can be assumed, that the information shown in profiler are not the ones for entire kernel execution, but rather its initialization (similarly to the CUDA's Runtime API line in timeline (see Figure 8.6) and OpenCL's API line in timeline (see Figure 8.8). Therefore, all theoretical speed computations based on this information are not valid. Yet, it might be possible to measure the time more accurately, as the possibility to add custom markers into code exists [44]. However, their usage would be out of the scope of this work.

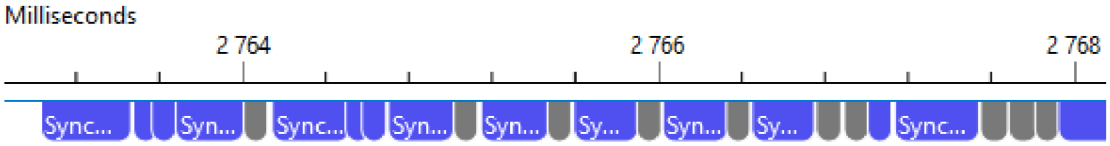


Figure 8.10: Timeline view from Visual Studio Concurrency profiler on kernels execution for SSW of size 384×256 px and HW configuration 2 showing the concurrent execution of kernels: blue – data copy and kernels execution, gray – grouped data copies and kernels, that are too small to distinguish.

These results has shown, that for the small sizes of SSWs, it is important to have fast CPU (otherwise, the GPU must wait). The speed of GPU's access to the RAM is also important (even more for fast GPUs), as the NNMP kernel execution time is similar to the other kernels and their execution time can negatively affect the total execution time, because of the possibly slow access to the global memory. However, this does not apply as much in case of bigger SSWs, as the NNMP computation uses local data copy. Therefore, for optimal results, all these factors (CPU and GPU speed, GPU's memory bandwidth and PCIe transfer rate, input video resolution and SSW size) must be in right balance in order to fully utilize all available resources.

Also, there is the question, of how much accurate are used profilers, especially in time measurement for small data transfers and kernels. Nonetheless, the there is a problem with presented theoretical speeds: the profiler causes slowdown of execution (as has been shown), but we were unable to determine, if the slowdown is caused by the bigger breaks between the executions, the kernel execution times stretching or both. Therefore, the computed theoretical values would likely differ in real scenario and can be used only for the illustration of the performance bottlenecks.

8.1.2 Measured results

The performance testing was done on video sequence "tracking" using average values from the first 500 frames. This ensured, that there is enough samples (stabilization cycles) and also it is fast enough, as the testing of whole video sequence would require great

amount of time. In order to remove influence of kernels compilation, the second run is used in measurement.

The selection of this video does not have any specific reason. The testing on the other video sequences would yield mostly the same results (the smaller resolution reduces the CPU usage and therefore there might be some small performance gain on slower CPUs like Intel Q6600, but both cameras produces the same bitrate when recomputed to the same FPS). Further, the deviations from the average were not measured, as the preliminary testing has shown, that they are smaller than 2 % (which does not matter, because the change can be neglected). The same is also true for the repeatability.

For the rest of this thesis, the boundary for the real-time processing is set to the 24 FPS, as this value is lowest standardized framerate for both video acquisition devices and cinemas. However, as mentioned before, the video stabilization is in computer vision often the only one of the first steps in entire process. Therefore, it is preferable to achieve high processing speed of video stabilization in order to enable further processing in real time. Also, the standards defining the video in 60 FPS exists. For this reason, the second boundary at the 60 FPS is chosen, that enables the real time processing of both most available video format speeds and enough spare time for additional video processing in real time. The measured values between these boundaries are distinguished using different colors, as can be seen in Table 8.4.

Table 8.4: Visual division of FPS into color categories.

0 – 29.9	30 – 59.9	60 +

Table 8.5: Performance comparison of proposed algorithm using different frameworks for SSW size of 128×128 px.

HW configuration		1	2	3	4	5	6	7
OpenCL	FPS	55.0	82.2	164.0	198.8	142.7	78.4	196.2
	Usage	58 %	25 %	75 %	60 %	42 %	38 %	44 %
OpenCL NVIDIA	FPS	-	84.1	164.0	198.8	-	61.0	200.4
	Usage	-	26 %	75 %	60 %	-	10 %	44 %
CUDA	FPS	-	88.2	195.2	224.0	-	160.0	202.2
	Usage	-	28 %	84 %	80 %	-	12 %	44 %
C++ AMP	FPS	21.0	42.5	84.4	109.7	61.2	81.8	114.9
	Usage	49 %	58 %	65 %	52 %	44 %	49 %	44 %
CPU OpenMP	FPS	26.2	26.2	51.8	56.5	56.5	30.3	51.5
	Usage	96 %	96 %	95 %	95 %	98 %	99 %	98 %
CPU	FPS	6.5	6.5	10.9	7.5	7.5	0.9	10.5
	Usage	24 %	24 %	20 %	13 %	13 %	3 %	20 %

As can be seen from Table 8.5, even for the smallest SSW size, the CPU does not have enough compute power to stabilize video in real-time. However, when all the cores are used, the processing speed can be considered to be real-time, and in case of HW configurations 3, 4, 5 and 7 also with some space for further processing. In case of GPU,

the processing speed is far beyond the real-time boundary, except the C++ AMP version, in HW configuration 1, where the execution speed does not enable real-time video stabilization. Further, only in two cases (OpenCL for configuration 1 and C++ AMP for configuration 2) is below the second limit of 60 FPS. From other results, it can be seen, that the CUDA implementation is fastest, closely followed with both OpenCL versions. Then, the third is C++ AMP with performance slightly better than half of CUDA performance. Yet, on the slowest GPU, it is even slower than CPU. Also, this implementation is slower than best of current CPU's even on the more powerful GPU used in HW configuration 2.

The next thing is the utilization of the GPUs. The slowest graphics card (Radeon 5750) is in OpenCL implementation used significantly more than card in configuration 2 (58 % vs 25 %). As the profiling has shown (see section 8.1.1), this is due to the both small performance of GPU and the slow CPU, where the same kernel execution takes much more time in case of C++ AMP and the OpenCL implementation must wait for the CPU provide data. The same is also true for the other tested HW configurations.

Here is also visible the difference of between the standard OpenCL and NVIDIA's modified version with memory barrier in divergent branch – yet, it is true only for the HW configurations 2, 6 and 7 (Fermi and Maxwell architectures designed for computing).

Table 8.6: Performance comparison of proposed algorithm using different frameworks for SSW size of 192×128 px.

HW configuration		**1**	**2**	**3**	**4**	**5**	**6**	**7**
OpenCL	FPS	31.0	82.0	89.7	100.0	142.8	108.0	176.4
	Usage	70 %	60 %	82 %	79 %	62 %	39 %	48 %
OpenCL NVIDIA	FPS	-	82.0	86.6	100.0	-	49.2	176.4
	Usage	-	60 %	82 %	80 %	-	19 %	48 %
CUDA	FPS	-	89.1	95.1	114.0	-	153	192.0
	Usage	-	80 %	84 %	93 %	-	52 %	52 %
C++ AMP	FPS	13.5	26.5	48.1	58.0	34.9	63.3	78.1
	Usage	65 %	70 %	73 %	66 %	70 %	47 %	56 %
CPU OpenMP	FPS	15.1	15.1	31.0	35.4	35.4	20.2	32.0
	Usage	96 %	96 %	97 %	98 %	98 %	99 %	98 %
CPU	FPS	3.8	3.8	6.4	4.7	4.7	0.6	6.5
	Usage	24 %	24 %	20 %	13 %	13 %	3 %	20 %

The Table 8.6 shows, that the CPU was indeed a limitation for all of used configurations: the GPU utilization has risen rapidly when compared with previous SSW size, but the FPS rate remains almost the same for some configurations (2, 5, 6 and 7). Yet, the CPU speed declines roughly 66 %. Therefore, the only the more powerful CPUs are able to stabilize video in real time (3, 4, 5 and 7) when all cores are employed and the others are way below the required limit.

The interesting thing in results is the performance bottleneck of NVIDIA's OpenCL against normal OpenCL for HW configurations 3 and 6. However, we were unable to

investigate this behavior deeply, as the tests were performed on these PCs using remote desktop without ability to install profiler.

Table 8.7: Performance comparison of proposed algorithm using different frameworks for SSW size of 256×128 px.

HW configuration		1	2	3	4	5	6	7
OpenCL	FPS	23.5	65.0	54.3	67.4	104.7	86.0	160.3
	Usage	88 %	75 %	80 %	86 %	52 %	29 %	65 %
OpenCL NVIDIA	FPS	-	70.0	56.4	67.8	-	86.0	162.3
	Usage	-	75 %	82 %	87 %	-	29 %	65 %
CUDA	FPS	-	81.0	62.2	75.1	-	158.0	230.8
	Usage	-	89 %	94 %	95 %	-	58 %	70 %
C++ AMP	FPS	9.4	16.8	28.8	34.1	31.2	40.0	48.1
	Usage	73 %	77 %	78 %	74 %	66 %	55 %	70 %
CPU OpenMP	FPS	9.4	9.4	21.1	21.7	21.7	13.2	21.3
	Usage	96 %	96 %	96 %	98 %	98 %	99 %	98 %
CPU	FPS	2.4	2.4	4.4	2.9	2.9	0.4	4.3
	Usage	24 %	24 %	20 %	13 %	13 %	3 %	20 %

The test results from Table 8.5 shows that even the SSW size has doubled, the resulting speed remains approximately the same for the computation oriented GPUs (2, 5, 6 and 7) and in case of NVIDIA, their version of OpenCL has again better results. The performance of the other GPUs is slowly declining and the utilization has risen for all GPUs, approaching 95 % in some cases. Yet, all GPUs except Radeon 5750 are able to stabilize the video in real-time and most of the even with enough power for other computations. However, the C++ AMP version performance is still slow.

The CPUs speed has again declined (as expected) by roughly 40 % and none of the CPUs is able to stabilize the resulting video in real-time even when all cores are used.

Table 8.8: Performance comparison of proposed algorithm using different frameworks for SSW size of 192×192 px.

HW configuration		1	2	3	4	5	6	7
OpenCL	FPS	17.1	53.0	39.6	47.6	87.7	66.9	111.0
	Usage	82 %	75 %	79 %	83 %	61 %	59 %	57 %
OpenCL NVIDIA	FPS	-	53.0	38.9	48.2	-	67.1	111.0
	Usage	-	74 %	78 %	85 %	-	35 %	57 %
CUDA	FPS	-	64.7	45.2	54.5	-	115.5	175.3
	Usage	-	93 %	91 %	96 %	-	67 %	80 %
C++ AMP	FPS	7.9	14.4	24.9	29.6	35.7	35.7	42.7
	Usage	77 %	82 %	83 %	75 %	96 %	55 %	70 %
CPU OpenMP	FPS	7.2	7.2	17.1	17.1	17.1	11.5	18.0
	Usage	96 %	96 %	95 %	98 %	98 %	99 %	98 %
CPU	FPS	1.8	1.8	3.6	2.3	2.3	0.3	3.7
	Usage	24 %	24 %	20 %	13 %	13 %	3 %	20 %

The SSW size of $192 \times 192\ px$ is demanding enough (see Table 8.8) to fully utilize all the GPUS and the performance drop is observable even for the most powerful ones. Yet, the results are only slightly worse than in case of results in Table 8.5 (performance drop of approximately 15 %). However, the CPU implementations have performance drop of almost 25 %.

Table 8.9: Performance comparison of proposed algorithm using different frameworks for SSW size of 256×192 px.

HW configuration		1	2	3	4	5	6	7
OpenCL	FPS	10.9	34.6	27.9	33.3	90.1	60.0	95.4
	Usage	90 %	83 %	89 %	91 %	65 %	47 %	62 %
OpenCL NVIDIA	FPS	-	34.6	28.5	33.3	-	60.0	95.4
	Usage	-	83 %	90 %	91 %	-	47 %	62 %
CUDA	FPS	-	39.7	30.2	35.2	-	96.5	126.2
	Usage	-	96 %	97 %	97 %	-	71 %	84 %
C++ AMP	FPS	1.9	8.9	17.2	19.2	21.6	25.1	28.1
	Usage	83 %	87 %	88 %	82 %	61 %	67 %	72 %
CPU OpenMP	FPS	4.5	4.5	10.6	10.3	10.3	8.4	11.5
	Usage	96 %	96 %	96 %	98 %	98 %	99 %	99 %
CPU	FPS	1.1	1.1	2.2	1.4	1.4	0.3	2.3
	Usage	24 %	24 %	20 %	13 %	13 %	3 %	20 %

The further increase of the SSW size brings again performance drop for all tested hardware configurations. The GPUs 2, 3 and 4 are still able to process video in the real-time, but there is not much room left for other computations. Also, the C++ AMP versions are no longer usable, except for configurations 6 and 7. These GPUs are still not fully utilized (the used CPUs are slow), yet they are suitable for another tasks except the video stabilization. Also, the CUDA version is still faster than OpenCL.

Table 8.10: Performance comparison of proposed algorithm using different frameworks for SSW size of 256×256 px.

HW configuration		1	2	3	4	5	6	7
OpenCL	FPS	6.9	21.2	16.3	19.6	59.2	40.1	92.7
	Usage	95 %	89 %	97 %	99 %	65 %	45 %	99 %
OpenCL NVIDIA	FPS	-	21.2	16.4	19.6	-	39.9	92.7
	Usage	-	89 %	97 %	99 %	-	52 %	99 %
CUDA	FPS	-	23.4	16.8	19.6	-	60.0	78.4
	Usage	-	97 %	97 %	98 %	-	81 %	89 %
C++ AMP	FPS	3.0	5.4	10.4	12.0	15.0	16.7	19.0
	Usage	90 %	92 %	90 %	87 %	85 %	75 %	73 %
CPU OpenMP	FPS	2.6	2.6	6.3	6.0	6.0	5.2	6.9
	Usage	96 %	96 %	98 %	98 %	98 %	62 %	99 %
CPU	FPS	0.7	0.7	1.3	0.8	0.8	0.2	1.4
	Usage	24 %	24 %	20 %	13 %	13 %	3 %	20 %

The SSW size four times bigger than the initial one is the challenge for all used configurations (see Table 8.10). The CPUs are again slower than in previous cases. However, also the GPUs from configurations 2, 3 and 4 are no longer usable for real-time video stabilization using implemented method. The same apply for the C++ AMP across all GPUs. However, the GPUs 5, 6 and 7 are still suitable for real-time processing, even that the framerate of stabilization is relatively low, because their utilization is only slightly higher than 50 % (CPU is the limitation). Therefore, additional, not CPU intense tasks can be computed without the impact on the stabilization speed.

The interesting thing also visible from the table is the better performance of OpenCL over CUDA for GPU 7 by almost 19 %. However, this behavior could not be examined by profiler for the reasons mentioned before.

Table 8.11: Performance comparison of proposed algorithm using different frameworks for SSW size of 384×192 px.

HW configuration		1	2	3	4	5	6	7
OpenCL	FPS	5.0	16.6	12.7	13.3	44.2	39.2	70.9
	Usage	93 %	91 %	97 %	99 %	71 %	62 %	81 %
OpenCL NVIDIA	FPS	-	16.6	12.8	13.3	-	39.2	70.9
	Usage	-	91 %	97 %	99 %	-	62 %	81 %
CUDA	FPS	-	17.9	13.1	13.3	-	58.0	63.4
	Usage	-	98 %	97 %	99 %	-	81 %	92 %
C++ AMP	FPS	2.4	4.3	8.2	9.4	11.7	14.1	14.9
	Usage	92 %	92 %	95 %	91 %	85 %	65 %	80 %
CPU OpenMP	FPS	2.1	2.1	6.3	5.3	5.3	4.6	6.0
	Usage	96 %	96 %	97 %	98 %	98 %	62 %	99 %
CPU	FPS	0.5	0.5	1.3	0.7	0.7	0.2	1.2
	Usage	24 %	24 %	20 %	13 %	13 %	3 %	20 %

The SSW size increase to $384 \times 192\ px$ does basically make first four HW configurations unusable for real-time stabilization (see Table 8.11). The remaining three configurations are usable only with OpenCL and CUDA. However, the GPUs are still not fully utilized, as the used CPUs does are not able to supply enough of data for processing. Therefore, additional, non CPU intensive tasks can be computed on GPUs.

Also, this case also shows the same interesting behavior as in previous case, where the OpenCL version is faster than CUDA.

The last tested SSW size (Table 8.12) brings another drop in speed of all GPUs. The results are similar to the previous case, as only the last three GPUs are suitable for real-time video stabilization for this SSW size. However, they are still not fully utilized and therefore can be used for additional computations.

The impact of the CPU performance on the used GPU can be seen from the speedup between the last two GPUs: both of them uses the same architecture and even the chip. However, the later disabled two blocks of compute units and the high performance is ensured by the higher working frequency. Yet, the later has significantly bigger

performance in video-stabilization. This is caused by the used CPUs, where their working frequencies are 2.0 GHz and 4.4 GHz respectively. Because the implemented video stabilization method uses only one thread to select the areas of concern and send them to GPU, the frequency has linear impact to raw performance. Therefore, the GPU can be also used by other unused threads for other computations.

Table 8.12: Performance comparison of proposed algorithm using different frameworks for SSW size of 384×256 px.

HW configuration		1	2	3	4	5	6	7
OpenCL	FPS	3.9	9.7	6.7	7.8	30.5	23.5	35.5
	Usage	94 %	93 %	90 %	97 %	79 %	63 %	82 %
OpenCL NVIDIA	FPS	-	9.7	6.7	7.8	-	23.5	35.5
	Usage	-	93 %	97 %	97 %	-	63 %	82 %
CUDA	FPS	-	10.2	6.9	8.2	-	30.3	37.7
	Usage	-	98 %	99 %	99 %	-	85 %	95 %
C++ AMP	FPS	0.6	2.5	5.1	5.9	8.2	7.5	9.8
	Usage	88 %	95 %	95 %	92 %	85 %	74 %	85 %
CPU OpenMP	FPS	1.2	1.2	3.0	2.8	2.8	2.3	3.3
	Usage	96 %	96 %	98 %	98 %	98 %	32 %	99 %
CPU	FPS	0.3	0.3	0.6	0.4	0.4	0.1	0.7
	Usage	24 %	24 %	20 %	13 %	13 %	3 %	20 %

Besides the raw performance values, the actual speedup of GPGPU over CPU is also important. For this reason, two summary charts were created for each configuration (see APPENDIX B: GPGPU speedup over CPU): the one showing the speedup against the sequential CPU version and the one showing the speedup against the CPU parallelized version. The speedup values are scaled to the real area of used SSW in order to better visualize the possible speedup and its course.

The results show, that the speedup depends greatly on the ration of the speeds of used CPU and GPU – if the ratio is big (the CPU is slow), the GPU must wait for CPU to deliver data required for computation. This behavior can be seen for HW configurations 2, 6 and 7 (Figure 9.7, Figure 9.11 and Figure 9.12). However, when the amount of computation reaches the certain level, the GPU is no longer limited and the speedup stabilizes at the approximately same value. The value itself depends on the used CPU, therefore, it vary greatly between used configurations. For this reason, the summary figure with speedup normalized to 100 GFLOPS of CPU performance and 1 TFLOPS of GPU performance was created in order to enable fair comparison of speedups achievable on different GPUs (see Figure 8.11). This clearly shows, that the architectures designed for compute (HW configurations 2, 5, 6 and 7) provides the best performance. In ideal case (e.g. by direct measurement), the data series should have linear shape. However, because the data of the figure are only estimated from other measured values, there are deviations from this expected shape.

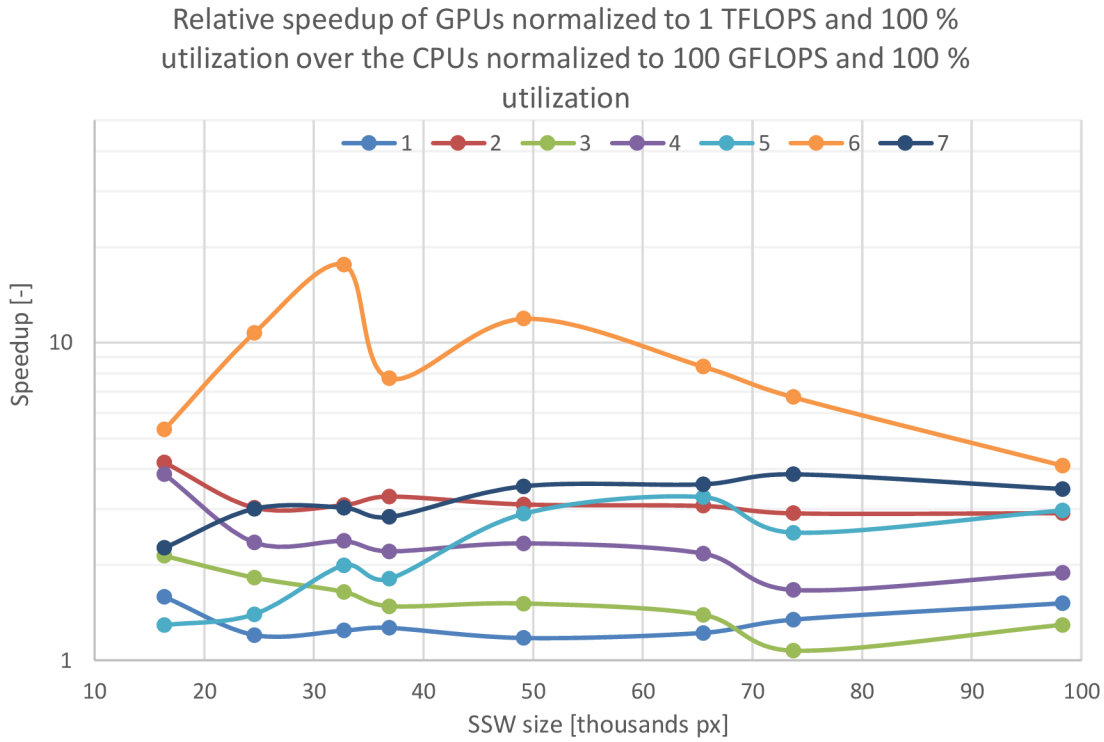


Figure 8.11: The relative speedup of the OpenCL implementation on the all used GPUs normalized to the 1 TFLOPS of performance and 100 % of utilization for all measured SSW sizes over the CPUs normalized to the 100 GFLOPS and 100 % utilization.

The interesting thing is, that when the performance of both GPU and CPU is normalized to the same value of 100 GFLOPS, none of the GPUs does provide better performance per 1 FLOPS except the NVIDIA Quadro M6000 in some cases. Yet, three advantages of GPUs exists: firstly, in lot of cases, they are already present in PC and it would be pity not to use them. Secondly, they provide additional performance and when used, the part of CPU dedicated to other tasks. Lastly, their performance is higher by at least two orders of magnitude with the same power consumption and their price is comparable.

Further, the performance of the tested GPGPU frameworks was also tested. The results can be seen in Figure 8.12. Here, the OpenCL performance is considered to represent 100 %. On all tested hardware configurations, the C++ AMP performance was between 20 % and 60 % of the OpenCL performance. For this reason, the C++ AMP implementation of algorithm is not competitive. The second solution – CUDA has better results and outperforms the OpenCL in most cases. Its performance is mostly about 5 % better, for smaller SSW sizes it is even more. In case of large SSW sizes, the difference drops down to 5 %.

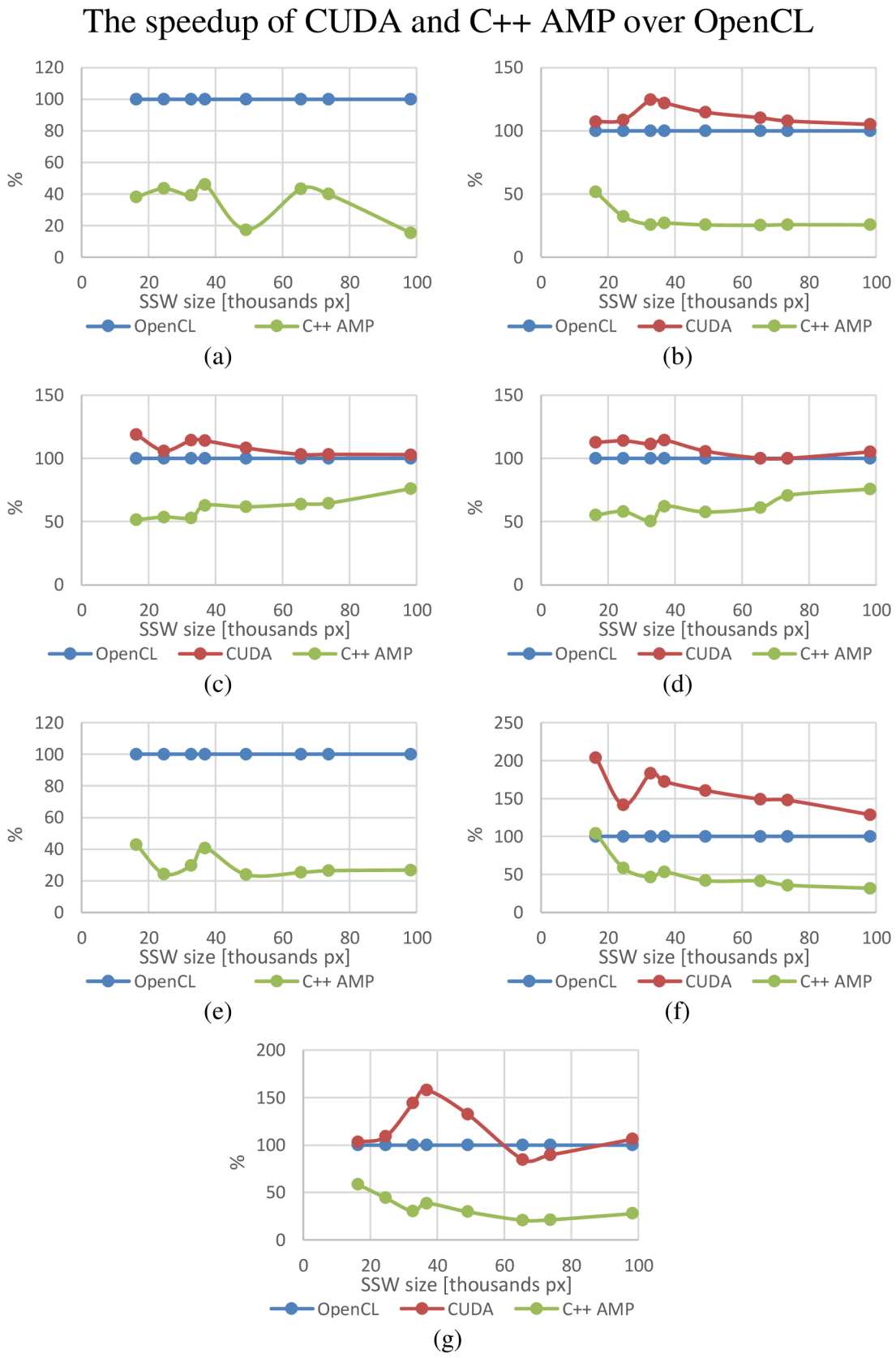


Figure 8.12: The speedup comparison for CUDA and C++ AMP over OpenCL: a) HW configuration 1, b) HW configuration 2, c) HW configuration 3, d) HW configuration 4, e) HW configuration 5, f) HW configuration 6, g) HW configuration 7.

8.2 Video stabilization quality evaluation

For the real-time applications, the quality of video stabilization is equally important as its speed. While multiple evaluation techniques of video stabilization quality exists, they all have some issues (see chapter 2.4). Therefore, the presented real-time video stabilization method was evaluated using multiple approaches.

8.2.1 ITF

The ITF testing was performed only on the central part of size $880 \times 320\ px$. The reason for choosing only the small part of input resolution is possibility of error introduction caused by the black edges that are the result of image stabilization.

Table 8.13: Comparison of ITF values for original and stabilized video.

Video number	1	2	3	4	5	6	7	8	9	10	11
Original ITF [dB]	21.6	29.5	23.4	26.8	29.9	30.3	22.6	23.4	27.7	20.6	19.8
Stabilized ITF [dB]	22.7	29.1	22.8	28.4	31.2	33.4	22.7	25.0	28.2	24.3	23.1
Difference [dB]	1.1	-0.4	-0.6	1.6	1.3	3.1	0.1	1.6	0.5	3.7	3.3

As can be seen from the results of measurement in Table 8.13, the ITF values are indeed very similar for video sequences containing rapid zoom changes. In two of four video sequences capturing the car ride from the driver's viewpoint the ITF metrics get worse with the image stabilization. However, the difference is not big even for the video sequences with the minimum of the movement. This is caused by the strong footprint of the rolling shutter effect in all of the videos and focus changes. Therefore, the quality of video stabilization was evaluated also visually. The subjective opinion of multiple persons is that this metrics does not show the real quality of the stabilization. For all those reasons, the second evaluation using the frequency analysis was performed

8.2.2 Camera path evaluation

Because the ITF test results shows very similar values for videos with the zoom changes, another technique employing frequency analysis to compare the presence of different frequencies was used. While for the full objectivity, the use of some different stabilization method for frame displacement computation would be suitable, we were unable to find such a solution that is used enough and at the same time able to export the displacement values for individual frame pairs. Therefore, the proposed algorithm was used. In order to ensure the correctness of stabilization, the biggest tested SSW size ($384 \times 256\ px$)

was used. This size enables to use correction of up to $96\,px$ for x axis and $64\,px$ for y axis in both directions (half of the MSW size, as the frames can be displaced in both directions). Then, the displacement values were padded to the power of 2 and transformed using DFT. The output of this process are the magnitudes of the frequency components present in the data. The data itself were converted to the absolute values for better visualization and shows only the real part of the transform. They can be found in APPENDIX D: Frequency analysis of camera paths before and after video stabilization.

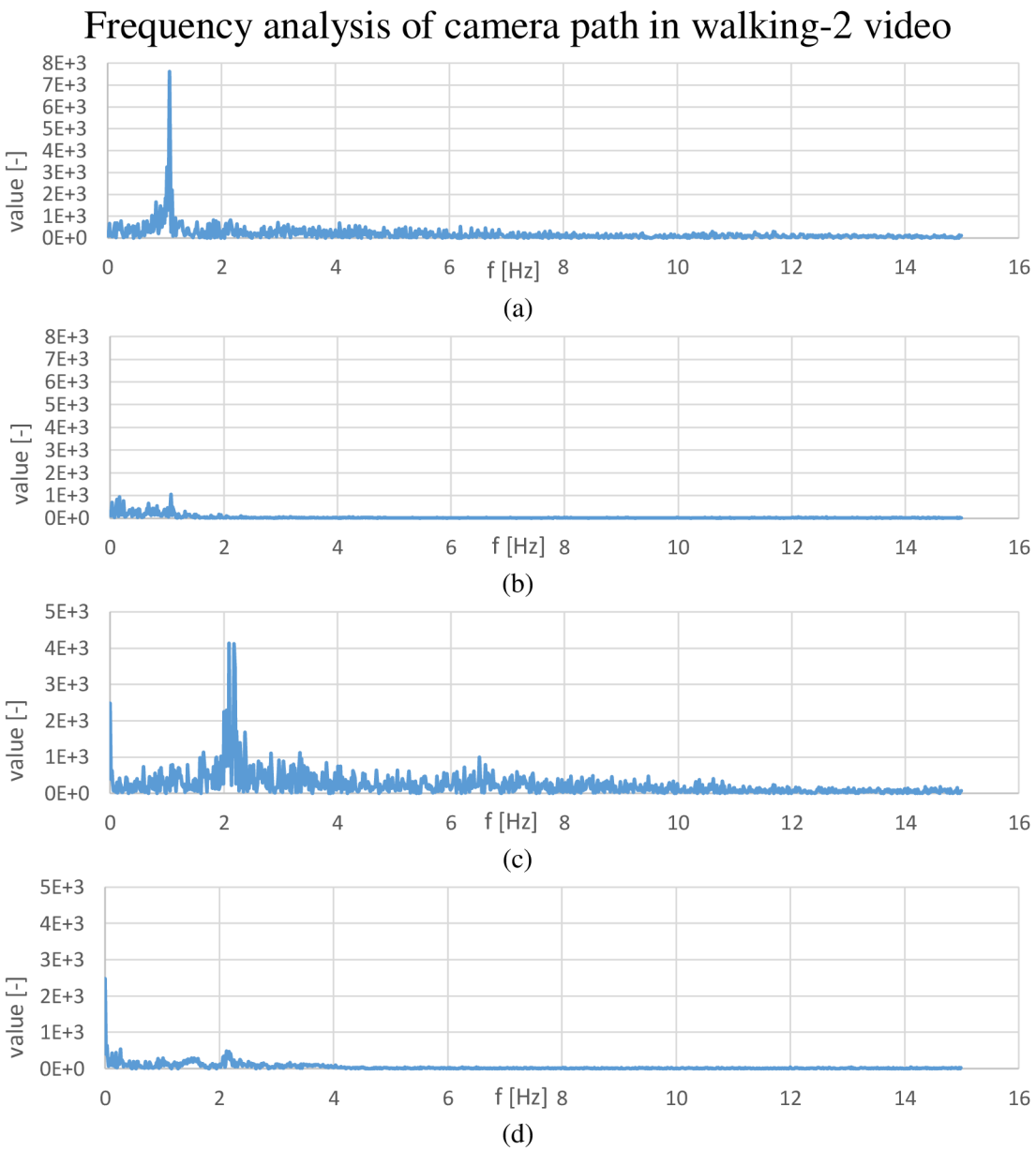


Figure 8.13: The frequency analysis of the walking-2 video sequence: a) x axis before stabilization, b) x axis after stabilization, c) y axis before stabilization, d) y axis after stabilization.

As can be seen from the results, in all of the video sequences, the higher frequencies are present. However, their amount is radically reduced by the stabilization. Also, prior to the stabilization, in some case (e.g. "walking-2" video sequence, Figure 8.13) is the

prevailing frequency of about 1 Hz in x axis and 2 Hz in y axis. The proposed stabilization method removes this prevailing frequencies and the most common frequencies becomes the ones around zero.

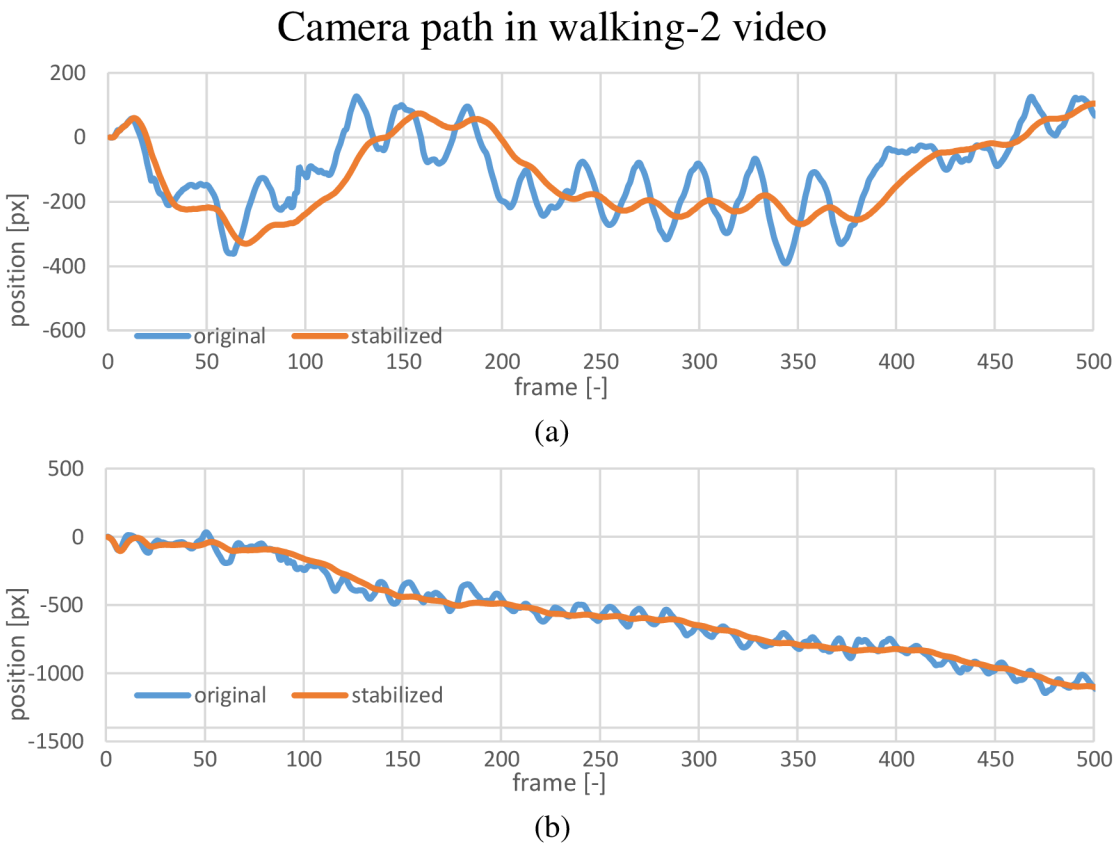


Figure 8.14: Camera path in walking-2 video before and after video stabilization: a) x axis, b) y axis.

For better comparison, the figures containing the absolute position of camera pose against the first frame for the first 500 frames was generated. They can be found in APPENDIX C: Camera paths before and after video stabilization. The Figure 8.14 shows the absolute camera pose before and after the video stabilization of the "walking-2" video sequence. As can be seen, the proposed stabilization smoothens the camera path. However, the used Kalman filtering has several issues: firstly, the response separation of wanted and unwanted motion in first few frames is not perfect, as the camera movement is followed closely. This is the expected behavior, as the Kalman filtering uses prediction of the future state based on the previous states. However, the second and bigger issue is the inability to recognize the sudden intended change in camera position and cannot be resolved by the standard Kalman filter, as it is one of its properties. Therefore, based on the preferences and nature of the video that needs to be stabilized, its parameters can be modified to better suit its application (e. g. in case of walking, the used settings fails to straighten the camera movement in x axis. This can be resolved by smaller sensitivity to the difference of the actual and predicted position, but it will result into bigger delay when sudden intended change of camera position is introduced. From the Figure 8.14a it is clear, that the delay is approximately 7 frames. This results to the threshold of 2.1 Hz.

The bigger frequencies are filtered out, as they are considered to by shake. The lower frequencies are smoothen, but followed with delay. Therefore, the better filtering is required to solve this issues.

8.2.3 Subjective quality evaluation

By subjective visual evaluation, all of the testing video sequences are better than their originals in terms of stability. However, the problem is the presence of rolling shutter effect, where image defects like scaling in vertical axis or wriggle are present in the images. This, when combined with blur, often causes that the resulting video seems unstabilized and only the side by side comparison proves otherwise. Another common problem is that for small focal lengths, the barrel effect is present in the frames and their stabilization creates the weird and disturbing changes in images (e.g. "walking" video sequences). Therefore, the high quality camera with good optics is required to eliminate the presence of these image defects.

Beside these problems, it can be said, that the proposed method overcame our expectations as it is able to deal not only with the image defects but also with the zoom effect (camera movement in z axis) or with the partially covered image with different objects (the "car-ride-4" video has the dashboard in the bottom quarter of the image).

8.3 Comparison of real and estimated SSW size

Based on the output of the stabilization of tested video sequences, the analysis of the required compensation values was performed. The histogram shows, that the distribution of values is not the normal distribution as in case of hand tremor (see section 2.2), but rather peaked at the deviation of zero with the sharp decrease within deviation of few pixels and then followed by slow descent to the extremes (see Figure 8.15).

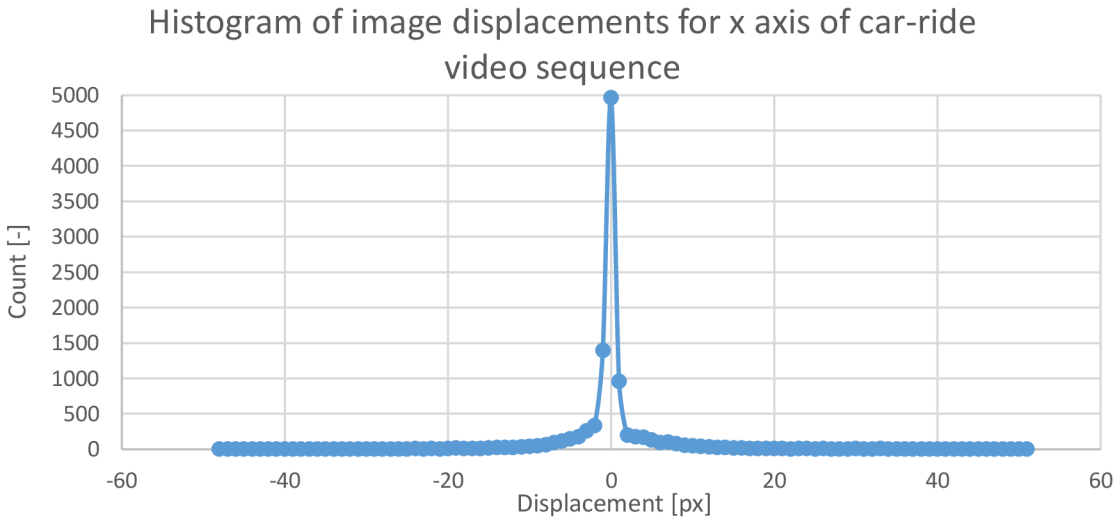


Figure 8.15: Distribution of displacement values in the x axis of the car-ride video sequence.

For this reason, it may be suitable to lower the computation requirements by defining the smaller window size. Because of the sharp descent of counts for bigger displacements, even the reduction of requirements to correctly capture 99 % of translation changes may bring significant speedup. Therefore, for all of tested video sequences, the SSW sizes to capture 99 %, 95 % and 90 % of displacements between consecutive frames were computed (see Table 8.14). The SSW sizes in table are distinguished by different colors based on the ability of the hardware used for testing to stabilize the video with the defined SSW sizes in real-time and perform other computations: green – all hardware, yellow – only HW configurations 5, 6 and 7, red – none of the used hardware.

Table 8.14: Measured sizes of SSW required to compensate specific percentage of consecutive frames displacement.

Video number	1	2	3	4	5	6	7	8	9	10	11
100 % [px]	204 × 236	304 × 212	180 × 72	168 × 88	68 × 108	40 × 52	156 × 140	376 × 192	192 × 180	232 × 196	304 × 232
99 % [px]	72 × 76	232 X 76	144 × 36	72 × 32	60 × 96	36 × 28	64 × 72	360 × 92	100 × 80	164 × 144	140 × 172
95 % [px]	36 × 48	128 × 36	88 × 20	44 × 20	44 × 40	28 × 16	44 × 40	164 × 44	48 × 32	108 × 48	108 × 124
90 % [px]	24 × 36	64 × 24	52 × 12	32 × 16	32 × 32	20 × 12	32 × 24	92 × 32	40 × 20	92 × 28	92 × 108

As can be seen from the results in Table 8.14, even the reduction to capture only 99 % of displacement values brings significant speedup. However, it is important to note, that the SSW sizes shown in this table are not optimized for GPU processing and are computed only for the used testing videos. Therefore, in real scenario, a more thorough testing may be required to set the SSW size accordingly.

The table further shows that the required SSW sizes for the stabilization of the car-ride videos depends greatly on the nature of the roads and the speed of vehicle. In the first "car-ride" video, the changes in y axis are bigger than in x axis, this is due to the high speed of vehicle, when even the small pothole causes the vehicle to swing. In other three video sequences, this effect is not present, as the car is moving slowly, even that the potholes are bigger (average speed of 25 kph over 70 kph in first video). Secondly, there is the great influence of the distance of the objects – the near object changes their position and size more rapidly between the consecutive frames and therefore bigger search window size is required. Therefore, the design of the SSW size in case of vehicle mounted camera must account for its focal distance, the environment (distance of the object from camera) and the expected speed of vehicle. Yet, the "car-ride" video sequence shows, that the proposed method is able to stabilize with the vehicle moving at the top speed of 100 kph without problems.

Next, the “jitter” videos are as expected within the expected tolerance, as described in section 5.1.1. However, this does not apply for the “pan-zoom” video sequences. Yet, this is not due to the shake, but rather the sudden and intended change of camera pose. Therefore, during the making of the decision of how big the SSW should be, the maximal speed of intended camera pose should be also accounted for. The same does apply for the tracking video.

However, in the walking videos, the camera position change is caused by the nature of walking, and therefore it cannot be affected in order to reduce the needed SSW size.

The results in Table 8.14 further shows, that the requirement to correctly compensate 95 % (or 99 %) of consecutive frames displacements enables to greatly lower the computation requirements. This also means, that with bigger SSW sizes, the stabilization of video sequences captured with the big focal distance is also possible, if the actual translation differences are not too big.

9 CONCLUSION AND FUTURE WORK

The aim of this Master's thesis was to introduce reader into the problem of image stabilization and available solutions. Firstly, problem description was presented together with examples of situations, where image stabilization is needed.

In the first chapter, three phenomena related to shaky image were presented together with explanation of their origin, relationship and possible countermeasures. Also the methods used for evaluation of quality of video stabilization were presented.

Second chapter presented four approaches used to stabilize image: mechanical, optical, electronics and digital with aim on DIS. Digital image stabilization was then deeply explained and reader was guided through individual steps of this process. Also the issues caused by changes of camera's angle of view and focal length were introduced together with possible countermeasures. Enhanced version of DIS was also presented. Finally, a brief summary comparing all four principles was provided.

The third chapter introduced the reader into GPGPU frameworks available in the workplace. Three technologies were discussed: OpenCL, CUDA and C++ AMP. For each, brief summary of pros and cons was presented, together with a short example of code. Finally, all three technologies were compared in a brief summary.

The fourth chapter firstly presented requirements for the video stabilization algorithm. Then, based on them, the algorithm for real-time stabilization using GPU was designed. The possibility of real-time processing is achieved by selection of eight areas of concern from the input frame. This both reduces the search space (less computationally expensive) and brings the error correction mechanism, as the required translation of consecutive frames is computed independently for each area. However, another level of error correction is introduced by taking the 8 best motion vectors from each area. This can improve the selection of the global motion vector in case of areas with non-distinctive edges.

The next step in order to achieve fast processing is the reduction of the color space for the correlation. Therefore, the special binarization technique called Local Binary Patterns is used. Its modified version enables creation of truly binary images and unlike in the case of the simple thresholding, the resulting image has edges present even for the very similar areas of image (e.g. sand, snow or sky). Secondly, binary image enables fast correlation using XOR. Then, for the each displacement of correlation template and the search area, this operation returns the number of non-matching points (NNMP), which can be directly used as the criterion for the local motion vector. Therefore, best local motion vectors are those, where NNMP value is lowest. The next step is the computation of global motion vector. This is achieved by selection of median value of local motion vectors for both of axes independently. Last step is the separation of the intended camera motion from the shake itself. For this, the Kalman filtering is used.

The fifth chapter describes the implementation of the proposed algorithm. Besides the specifics of implementation, also the author's subjective impression on the specific technology is mentioned. The designed algorithm was implemented using all three

GPGPU solutions described in this work. Moreover, the reference CPU version was created in both sequential and multithreaded version using OpenCL.

The sixth chapter contains the description of the 11 created testing videos. Besides their detailed description, also table with common properties is presented. The videos contains are captured using two different cameras and in various light conditions and environments. They are divided into multiple categories based on the mounting type (fixed, hand-held) and the resulting video type (walking, jitter, pan and zoom, car ride, tracking).

The seventh chapter contains the test results of the all algorithm implementations in the terms of execution speed and the assessment of the quality of the algorithm itself.

Firstly, the nine hardware configurations used for testing are presented. Their description contains exact specification of the used hardware (working frequency, available memory, theoretical performance). Then, all three GPU implementations are profiled under the one of the used hardware configurations for two different sizes of areas of concern (the smallest one and biggest one used during thorough testing of performance). Here, the assumptions that were done during implementation are verified and discussed. The most important is, that even the algorithms are designed to work asynchronly, the used CPU is too slow to deliver data in time when the smallest areas of concern are used and the GPU must wait. This results into different utilization of GPU over different GPGPU implementations. The second finding is that the CUDA is able to launch different kernels at the same time, if the utilization of GPU is low. This overlap of kernels brings advantage over the OpenCL, as even that the duration of the kernels is slightly longer, the overlap reduces the total execution time. Next, the C++ AMP profiler, on the contrary of the CUDA and OpenCL does not report the actual operations on GPU, but only their scheduling order. Therefore, it is not possible to compare actual execution times and compute the maximal theoretically achievable execution speed. Also, it was found out, that the OpenCL version does transfer all the resulting values of NNMP instead of only required 8 (the CUDA does this correctly).

The second part of this chapter presents the performance results for 8 different sizes of areas of concern for all hardware configurations. Besides the achieved speed measured in frames per seconds (FPS), also the utilization of GPU is measured. The utilization is also measured for both CPU versions.

The results shows that depending on the computing power of the used GPU, even the fastest CPUs are not fast enough to fully utilize the most powerful GPUs. This can be partially overcome by the use of bigger area of concern. The next thing is, that all of the GPUs are able to achieve the real-time processing speed (24 FPS), even when limited by the CPU. The performance of the different GPGPU implementations shows, that the C++ AMP can achieve only about half of the performance of OpenCL. The CUDA implementation is faster than OpenCL by 10 % in for smaller areas of concern and 3 % in case of bigger areas of concern.

Further, the resulting performance of all GPUs and CPUs was normalized to the same theoretical computing power (1 TFLOPS and 100 GFLOPS respectively) and 100

% of utilization in order to compare the performance of the GPGPU architectures themselves. In case of NVIDIA, the results are as expected, where the GPU architecture designed for computing (Fermi and Maxwell) have much higher performance than Kepler designated for computer games. The similar result is in case of AMD, where the new CGN architecture of R9 Fury outperform the old Radeon 5750 by several times. Based on this, it can be concluded, that the NVIDIA's GPUs dedicated for computing have better performance in this algorithm to the unit of computing power than the AMD's. However, this claim may not be valid for all GPU computations.

In the third part, the quality of the video stabilization itself was evaluated. Firstly the Inter-frame Transform Fidelity criterion based on PSNR was used. However, because this methodology is not suitable for the videos containing zoom changes, the resulting values were very similar and in two cases of car-ride videos even worse. Yet, the subjective visual evaluation has shown, that the stabilized video is better than original. Therefore, second evaluation metrics empowering the discrete Fourier transform was used to get the magnitude of the present frequencies. Here, the improvement is clearly visible, as the higher frequencies are greatly reduced. This confirms the visual assessment. Yet, another evaluation was performed by visual comparison of the camera path before and after the video stabilization. This also confirmed that the performance of the method can be considered to be great. However, the proposed method suffers from the delayed reaction of the used Kalman filter for preserving of the intended motion.

The last part of this chapter perform statistical analysis of the displacements measured between frames in all testing videos. It was found out, that the values are not distributed according to the normal distribution, but rather peaked around the zero displacement with rapid decrease within small deviation. Then, the further decrease is slow towards the extremes. This means, that the lowering of requirements from correct compensation of 100 % frames to only 95 % can save significant amount of computations and therefore speedup the entire process.

To summarize, both the development and the testing shows, that the use of CUDA should be preferred over the OpenCL if suitable, because of the easier development and slightly better performance. The C++ AMP cannot be recommended for the image processing operations, as it lacks the support for 8-bit data type and data must be padded to 32 bits. This brings the unnecessary overhead caused by the conversions. Next, the achieved speeds of processing are far beyond the real-time threshold of 24 FPS, especially for smaller areas of concern and therefore this algorithm may be suitable even for mobile devices. It was also proven, than the most powerful GPUs currently available are too fast for the best CPUs for this task and therefore can be used for other computations without performance impact. Further, the proposed method showed great robustness against the different image defects like blur or rolling shutter, but also against movement in z axis. This is also true for the partially covered input frame with the foreign object. However, the problematic part is the Kalman filtering, which has issues with reaction to the sudden, but intended changes of camera pose. Lastly, the proposed algorithm enables the adaptation to the real-life scenarios, as the size of the areas of concerns can be adapted to

match the expected shake in both axes. At the end, it can be concluded, that all of the items in assignment were fulfilled.

In the future work, the designed algorithm can be further improved by replacing the Kalman filtering by some hybrid method. Also, it may be worth to try different ratio of the correlation template and the searching window. Lastly, the optimization of proposed algorithms, especially for the used GPU architectures could bring another performance boost.

References

[1] M. Drahanský, F. Orság and P. Hanáček, "Accelerometer Based Digital Video Stabilization for General Security Surveillance Systems," *International Journal of Security and Its Applications,* vol. 1, no. 1, p. 10, 2010.

[2] G. Li, "FPGA implementation of real-time digital image stabilization," Selected Proceedings of the Photoelectronic Technology Committee Conferences held July-December 2013, 2014.

[3] S. Mittal and J. S. Vetter, "A survey of methods for analyzing and improving GPU energy efficiency," *ACM Comput. Surv. 47,* vol. 2, no. Article 19, p. 23, July 2014.

[4] Intel, "Why OpenCL™?," [Online]. Available: https://software.intel.com/en-us/intel-opencl. [Accessed 2 December 2015].

[5] AMD, "AMD Accelerated Processing Units (APUs)," [Online]. Available: http://www.amd.com/en-us/innovations/software-technologies/apu. [Accessed 1 December 2015].

[6] J.-F. Chen and C.-S. Fuh, "IMAGE STABILIZATION WITH BEST SHOT SELECTOR AND SUPER RESOLUTION RECONSTRUCTION," *18th IPPR Conference on Computer Vision, Graphics and Image Processing,* pp. 1215-1222, August 2005.

[7] K. T. Wyne, "A comprehensive review of tremor," *Journal of the American Academy of Physician Assistants ,* vol. 18, no. 12, pp. 43-50, December 2005.

[8] F. L. Rosa, M. C. Virzì, F. Bonaccorso and M. Branciforte, "Optical Image Stabilization".

[9] E. M. Or and D. Pundik, "Hand Motion and Image Stabilization in Hand-held Devices," *IEEE Transactions on Consumer Electronics,* vol. 53, no. 4, pp. 1508-1512, November 2007.

[10] Xvid, "Questions and Answers," [Online]. Available: https://www.xvid.com/faq/. [Accessed 8 March 2016].

[11] The Moving Picture Experts Group, "MPEG-4," [Online]. Available: http://mpeg.chiariglione.org/standards/mpeg-4. [Accessed 8 March 2016].

[12] ITU, "Joint Video Team," [Online]. Available: http://www.itu.int/en/ITU-T/studygroups/com16/video/Pages/jvt.aspx. [Accessed 8 March 2016].

[13] Q. Cai, L. Song, G. Li and N. Ling, "Lossy and Lossless Intra Coding Performance Evaluation: HEVC, H.264/AVC, JPEG 2000 and JPEG LS," *Signal & Information Processing Association Annual Summit and Conference,* pp. 1-9, December 2012.

[14] J. Xu, H. W. Chang, S. Yang and M. Wang, "Fast feature-based video stabilization without accumulative global motion estimation," *IEEE Transactions on Consumer Electronics,* vol. 3, no. 58, pp. 993-999, 2012.

[15] T. Harris and J. Perrtano, "How Steadicams Work," [Online]. Available: http://entertainment.howstuffworks.com/steadicam.htm. [Accessed 7 November 2015].

[16] ELM ChaN, "Home built Steadicam," October 2007. [Online]. Available: http://elm-chan.org/docs/photo/hhsc_e.html. [Accessed 5 November 2015].

[17] R. Wotiz, June 2012. [Online]. Available: http://circuitcellar.com/wp-content/uploads/2012/06/CC2012050461.pdf. [Accessed 24 November 2015].

[18] F. Mée, "Image Stabilisers: Optical or Mechanical?," October 2012. [Online]. Available: http://www.digitalversus.com/digital-camera/image-stabilisers-optical-mechanical-a1608.html. [Accessed 17 November 2015].

[19] B. Kir, M. Kurt and O. Urhan, "Local Binary Pattern Based Fast Digital Image Stabilization," *Signal Processing Letters,* no. 22.3, pp. 341-345, 2015.

[20] W. J. Freeman, Digital Video Stabilization with Inertial Fusion, Blacksburg, VA: Virginia Polytechnic Institute and State University, 2013, p. 74.

[21] R. B. Inampudi, "Image Mosaicing," *Geoscience and Remote Sensing Symposium Proceedings,* vol. 5, pp. 2363-2365, July 1998.

[22] NVIDIA, "From Brook to CUDA," 2009. [Online]. Available: http://www.nvidia.com/content/GTC/documents/1001_GTC09.pdf. [Accessed 11 November 2015].

[23] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston and P. Hanrahan, "Brook for GPUs: stream computing on graphics hardware," *ACM Transactions on Graphics (TOG),* vol. 23, no. 3, pp. 777-786, August 2004.

[24] AMD, "AMD Delivers First Stream Processor with Double Precision Floating Point Technology," November 2007. [Online]. Available: http://ir.amd.com/phoenix.zhtml?c=74093&p=irol-newsArticle&ID=1074516. [Accessed 10 November 2015].

[25] NVIDIA, "NVIDIA® CUDA™ Unleashes Power of GPU Computing," February 2007. [Online]. Available: http://www.nvidia.com/object/IO_39918.html. [Accessed 12 November 2015].

[26] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on computers,* Vols. C-21, no. 9, pp. 948-960, September 1972.

[27] NVIDIA, "CUDA C Best Practices Guide," September 2015. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide. [Accessed 13 November 2015].

[28] AMD, "AMD Accelerated Parallel Processing - OpenCL Programming Guide," November 2013. [Online]. Available: http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf. [Accessed 2 November 2015].

[29] Apple, "Apple Previews Mac OS X Snow Leopard to Developers," June 2008. [Online]. Available: http://www.apple.com/pr/library/2008/06/09Apple-Previews-Mac-OS-X-Snow-Leopard-to-Developers.html. [Accessed 5 November 2015].

[30] Khronos group, "Khronos Drives Momentum of Parallel Computing Standard with Release of OpenCL 1.1 Specification," June 2010. [Online]. Available: https://www.khronos.org/news/press/khronos-group-releases-opencl-1-1-parallel-computing-standard. [Accessed 11 November 2015].

[31] Khronos group, "Khronos Releases OpenCL 1.2 Specification," November 2011. [Online]. Available: https://www.khronos.org/news/press/khronos-releases-opencl-1.2-specification. [Accessed 11 November 2015].

[32] Khronos group, "Khronos Releases OpenCL 2.0," July 2013. [Online]. Available: https://www.khronos.org/news/press/khronos-releases-opencl-2.0. [Accessed 11 November 2015].

[33] Khronos group, "Khronos Releases OpenCL 2.1 Provisional Specification for Public Review," March 2015. [Online]. Available: https://www.khronos.org/news/press/khronos-releases-opencl-2.1-provisional-specification-for-public-review. [Accessed 11 November 2015].

[34] NVIDIA, "Release 349 Graphics Drivers for Windows, Version 350.12," April 2015. [Online]. Available: http://us.download.nvidia.com/Windows/350.12/350.12-win8-win7-winvista-desktop-release-notes.pdf. [Accessed 14 November 2015].

[35] AMD, "Catalyst 14.41 RC1," September 2014. [Online]. Available: http://support.amd.com/en-us/kb-articles/Pages/OpenCL2-Driver.aspx. [Accessed 14 November 2015].

[36] AMD, "ACL – AMD Compute Libraries," [Online]. Available: http://developer.amd.com/tools-and-sdks/opencl-zone/acl-amd-compute-libraries/. [Accessed 1 November 2015].

[37] Khronos group, "JavaScript bindings to OpenCL brings heterogeneous parallel computing to Web browsers," March 2014. [Online]. Available: https://www.khronos.org/news/press/khronos-releases-webcl-1.0-specification. [Accessed 11 November 2015].

[38] Microsoft, "Pipeline Stages (Direct3D 10)," [Online]. Available: https://msdn.microsoft.com/en-us/library/bb205123(VS.85).aspx. [Accessed 10 November 2015].

[39] NVIDIA, "CUDA 4.0," 2011. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/4_0/CUDA_Toolkit_4.0_Overview.pdf. [Accessed 12 November 2015].

[40] NVIDIA, "CUDA 5.0," 2012. [Online]. Available: http://on-demand.gputechconf.com/gtc/2012/presentations/SS104-CUDA-5-What's-New.pdf. [Accessed 12 November 2015].

[41] NVIDIA, "CUDA 6.0," April 2014. [Online]. Available: http://devblogs.nvidia.com/parallelforall/powerful-new-features-cuda-6/. [Accessed 12 November 2015].

[42] NVIDIA, "CUDA 7.0," January 2015. [Online]. Available: http://devblogs.nvidia.com/parallelforall/cuda-7-release-candidate-feature-overview/. [Accessed 12 November 2015].

[43] NVIDIA, "Tools & Ecosystem," [Online]. Available: https://developer.nvidia.com/cuda-tools-ecosystem. [Accessed 17 November 2015].

[44] Microsoft, "C++ AMP : Language and Programming Model," December 2013. [Online]. Available: http://download.microsoft.com/download/2/2/9/22972859-15C2-4D96-97AE-93344241D56C/CppAMPOpenSpecificationV12.pdf. [Accessed 18 November 2015].

[45] HSA foundation, "Bringing C++AMP Beyond Windows via CLANG and LLVM," November 2013. [Online]. Available: http://www.hsafoundation.com/bringing-camp-beyond-windows-via-clang-llvm/. [Accessed 8 November 2015].

[46] itseez, "OpenCV," [Online]. Available: http://opencv.org/. [Accessed 28 April 2016].

[47] Microsoft, "Visual Studio," [Online]. Available: https://www.visualstudio.com/en-us/downloads/download-visual-studio-vs.aspx. [Accessed 11 May 2016].

[48] OpenMP.org, "The OpenMP® API specification for parallel programming," [Online]. Available: http://openmp.org/wp/. [Accessed 14 May 2016].

[49] J. Hoberock and N. Bell, "What is Thrust?," [Online]. Available: https://thrust.github.io/. [Accessed 14 May 2016].

[50] Stack Exchange, "Stack Overflow," [Online]. Available: http://stackoverflow.com/. [Accessed 14 May 2016].

[51] NVIDIA, "PTX ISA," 1 September 2015. [Online]. Available: http://docs.nvidia.com/cuda/parallel-thread-execution. [Accessed 14 May 2016].

[52] T. Tamasi, "The evolution of computer graphics," NVIDIA, San Jose, 2008.

APPENDIX A: OpenCL, CUDA and C++ AMP in comparison with C++ OpenMP

OpenCL code sample

```
1.  #pragma OPENCL EXTENSION cl_khr_byte_addressable_store : enable
2.  __kernel void vectorAdd(__global float *A, __global float *B, __global
    float *S, __const unsigned int n) {
3.      size_t id = get_global_id(0);
4.      if (i < n)
5.          S[id] = A[id] + B[id];
6.  }
```

Figure 9.1: OpenCL device code for vector addition.

```
1.  #include <utility>
2.  #define __NO_STD_VECTOR // Use cl::vector instead of STL version
3.  #include <CL/cl.hpp>
4.  #include <cstdio>
5.  #include <cstdlib>
6.  #include <fstream>
7.  #include <iostream>
8.  #include <string>
9.  #include <iterator>
10.
11. int main() {
12.     const size_t n = 50000;
13.     float *A = (float *)malloc(n * sizeof(float));
14.     float *B = (float *)malloc(n * sizeof(float));
15.     float *S = (float *)malloc(n * sizeof(float));
16.
17.     for (int i = 0; i < n; ++i) {
18.         A[i] = rand() / (float)RAND_MAX;
19.         B[i] = rand() / (float)RAND_MAX;
20.     }
21.
22.     cl_int err;
23.
24.     cl::vector< cl::Platform > platformList;
25.     cl::Platform::get(&platformList);
26.     std::string platformVendor;
27.     platformList[0].getInfo((cl_platform_info)CL_PLATFORM_VENDOR,
28.         &platformVendor);
29.     cl_context_properties cprops[3] = { CL_CONTEXT_PLATFORM,
30.         (cl_context_properties)(platformList[0])(), 0 };
31.     cl::Context context(CL_DEVICE_TYPE_CPU, cprops, NULL, NULL, &err);
32.     cl::Buffer a(context, CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,n,A, &err);
33.     cl::Buffer b(context, CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,n,B, &err);
34.     cl::Buffer s(context, CL_MEM_WRITE_ONLY, n, NULL, &err);
35.
36.     cl::vector<cl::Device> devices;
37.     devices = context.getInfo<CL_CONTEXT_DEVICES>();
38.
```

```
    std::ifstream file("hello_world.cl");
    std::string prog(std::istreambuf_iterator<char>(file),
        (std::istreambuf_iterator<char>()));
    cl::Program::Sources source(1, std::make_pair(prog.c_str(),
        prog.length() + 1));
    cl::Program program(context, source);
    program.build(devices, "");

    cl::Kernel kernel(program, "vectorAdd", &err);

    cl::CommandQueue queue(context, devices[0], 0, &err);
    cl::Event event;
    kernel.setArg(0, a);
    kernel.setArg(1, b);
    kernel.setArg(2, s);
    kernel.setArg(3, sizeof(size_t), &n);
    queue.enqueueWriteBuffer(a, CL_TRUE, 0, n, A, 0, 0);
    queue.enqueueWriteBuffer(b, CL_TRUE, 0, n, B, 0, 0);
    queue.enqueueNDRangeKernel(kernel, cl::NullRange, cl::NDRange(n),
        cl::NDRange(1, 1), NULL, &event);
    queue.enqueueReadBuffer(s, CL_TRUE,     0, n, S);

    for (unsigned int i = 0; i < n; i++)
        std::cout << static_cast<float>(S[i]);

    free(A); free(B); free(S);
    return 0;
}
```

Figure 9.2: OpenCL host code for vector addition.

CUDA code sample

```
#include <cuda_runtime.h>

__global__ void vectorAdd(float *A, float *B, float *C, int n) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n)
        C[i] = A[i] + B[i];
}

int main(void) {
    size_t n = 50000;
    float *A, *B, *S;
    cudaMallocManaged(&A, n); cudaMallocManaged(&B, n);
    cudaMallocManaged(&S, n);

    for (int i = 0; i < n; ++i)      {
        A[i] = rand() / (float)RAND_MAX;
        B[i] = rand() / (float)RAND_MAX;
    }

    int threadsPerBlock = 256;
    int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;
    vectorAdd <<< blocksPerGrid, threadsPerBlock >>>(A, B, S, n);
    cudaDeviceSynchronize();

```

```
    for (unsigned int i = 0; i < n; i++)
        std::cout << static_cast<float>(S[i]);

    cudaFree(A); cudaFree(B); cudaFree(S);
    return 0;
}

```

Figure 9.3: CUDA code for vector addition.

C++ AMP code sample

```
#include <iostream>
#include <amp.h>

int main() {
    size_t n = 50000;
    float *A = (float *)malloc(n * sizeof(float));
    float *B = (float *)malloc(n * sizeof(float));
    float *S = (float *)malloc(n * sizeof(float));

    for (int i = 0; i < n; ++i) {
        A[i] = rand() / (float)RAND_MAX;
        B[i] = rand() / (float)RAND_MAX;
    }

    concurrency::array_view<const float, 1> a(n, A);
    concurrency::array_view<const float, 1> b(n, B);
    concurrency::array_view<float, 1> sum(n, S);
    sum.discard_data();

    concurrency::parallel_for_each(sum.extent, [=](concurrency::index<1> i)
        restrict(amp) {
        sum[i] = a[i] + b[i];
    });

    for (unsigned int i = 0; i < n; i++)
        std::cout << static_cast<float>(sum[i]);

    free(A); free(B); free(S);
    return 0;
}
```

Figure 9.4: C++ AMP code for vector addition.

C++ OpenMP code sample

```
#include <iostream>

int main() {
   size_t n = 50000;
    float *A = (float *)malloc(n * sizeof(float));
    float *B = (float *)malloc(n * sizeof(float));
    float *S = (float *)malloc(n * sizeof(float));

```

```
1.      for (int i = 0; i < n; ++i) {
2.          A[i] = rand() / (float)RAND_MAX;
3.          B[i] = rand() / (float)RAND_MAX;
4.      }
5.
6.      #pragma omp parallel
7.      for (int i = 0; i < n; i++)
8.          S[i] = A[i] + B[i];
9.
10.     for (unsigned int i = 0; i < n; i++)
11.         std::cout << static_cast<float>(S[i]);
12.
13.     free(A); free(B); free(S);
14.     return 0;
15. }
```

Figure 9.5: C++ OpenMP code for vector addition.

APPENDIX B: GPGPU speedup over CPU

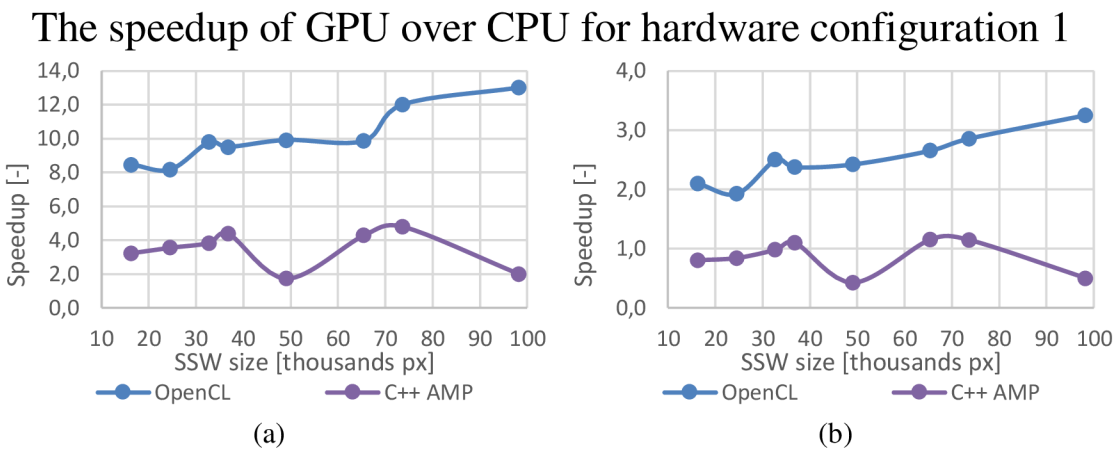


Figure 9.6: The speedup of GPU processing for different sizes of SSW over: a) single threaded CPU processing, b) OpenMP parallelized CPU processing for hardware configuration 1.

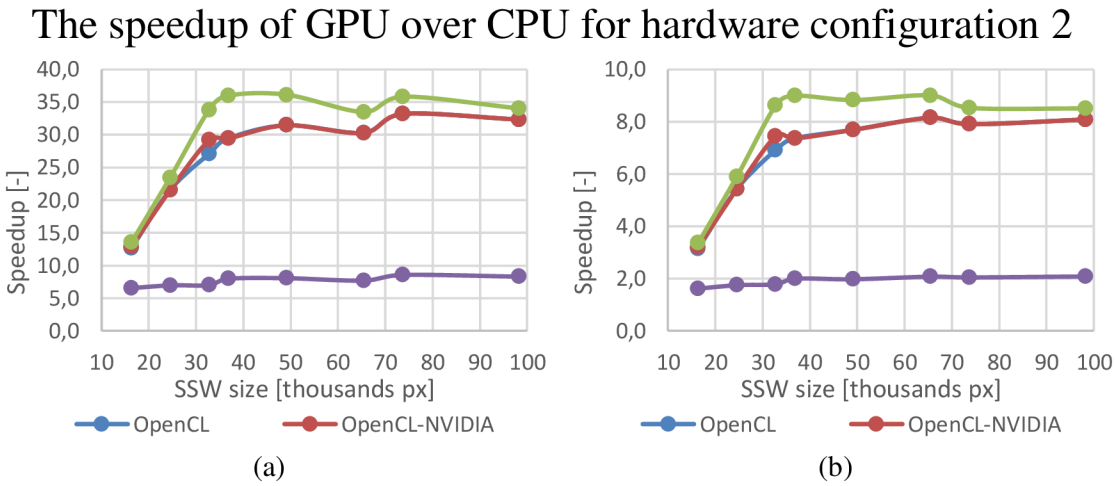


Figure 9.7: The speedup of GPU processing for different sizes of SSW over: a) single threaded CPU processing, b) OpenMP parallelized CPU processing for hardware configuration 2.

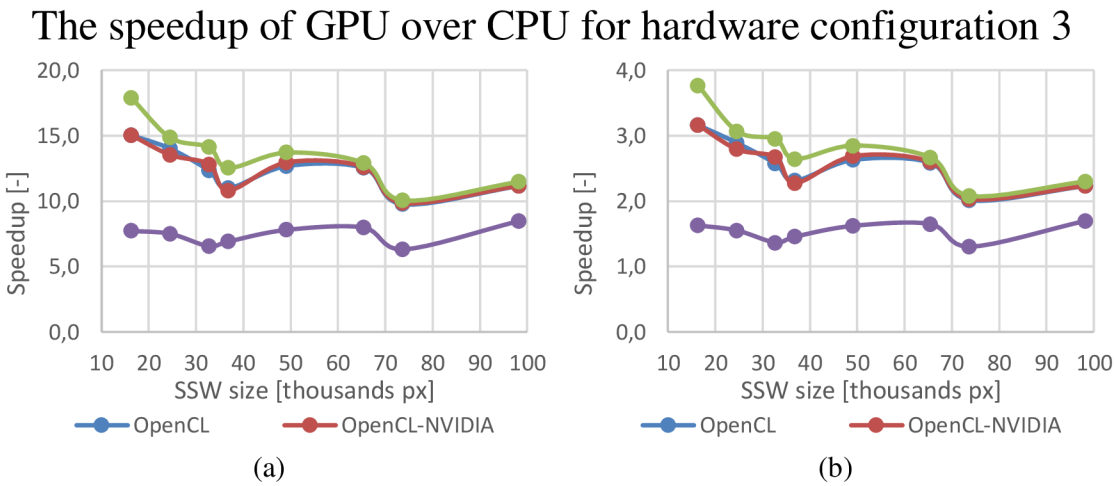


Figure 9.8: The speedup of GPU processing for different sizes of SSW over: a) single threaded CPU processing, b) OpenMP parallelized CPU processing for hardware configuration 3.

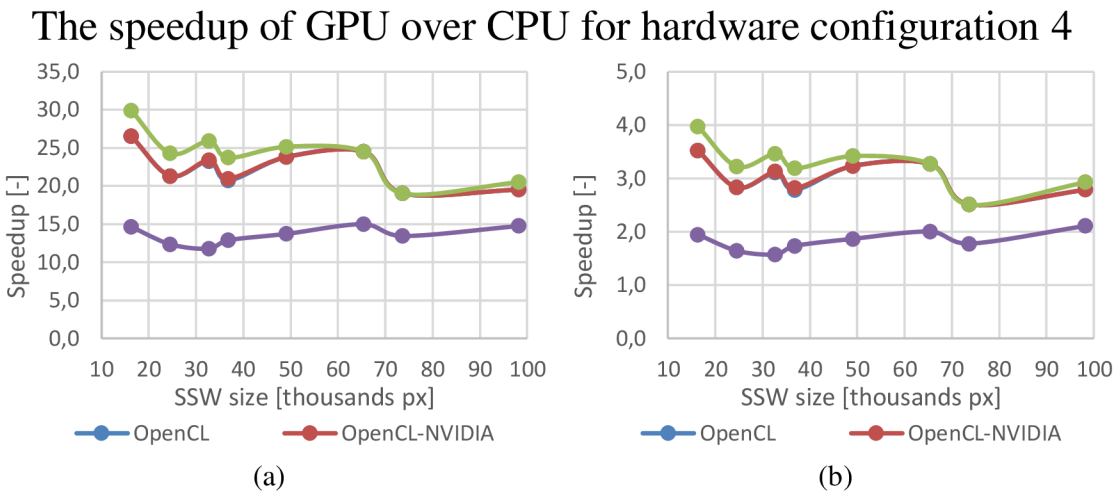


Figure 9.9: The speedup of GPU processing for different sizes of SSW over: a) single threaded CPU processing, b) OpenMP parallelized CPU processing for hardware configuration 4.

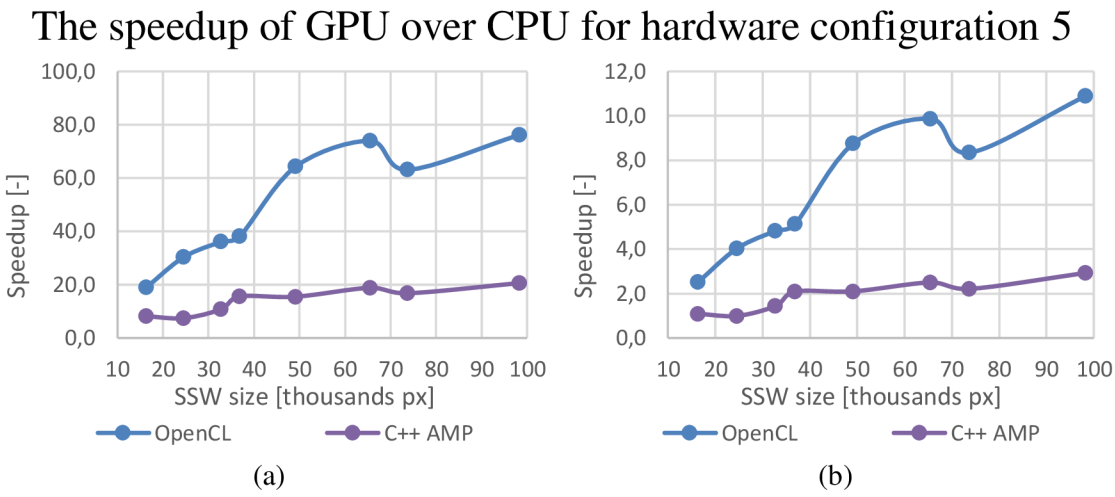


Figure 9.10: The speedup of GPU processing for different sizes of SSW over: a) single threaded CPU processing, b) OpenMP parallelized CPU processing for hardware configuration 5.

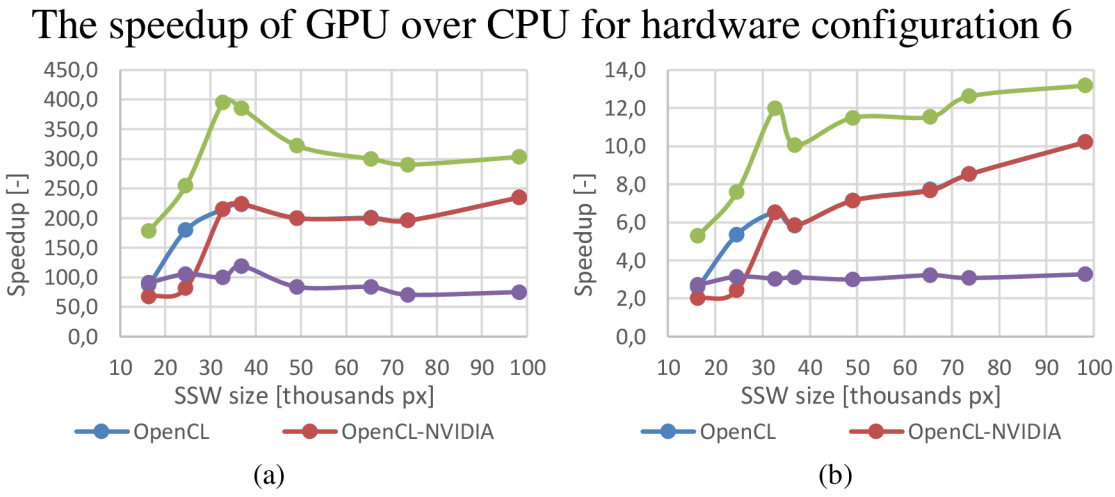


Figure 9.11: The speedup of GPU processing for different sizes of SSW over: a) single threaded CPU processing, b) OpenMP parallelized CPU processing for hardware configuration 6.

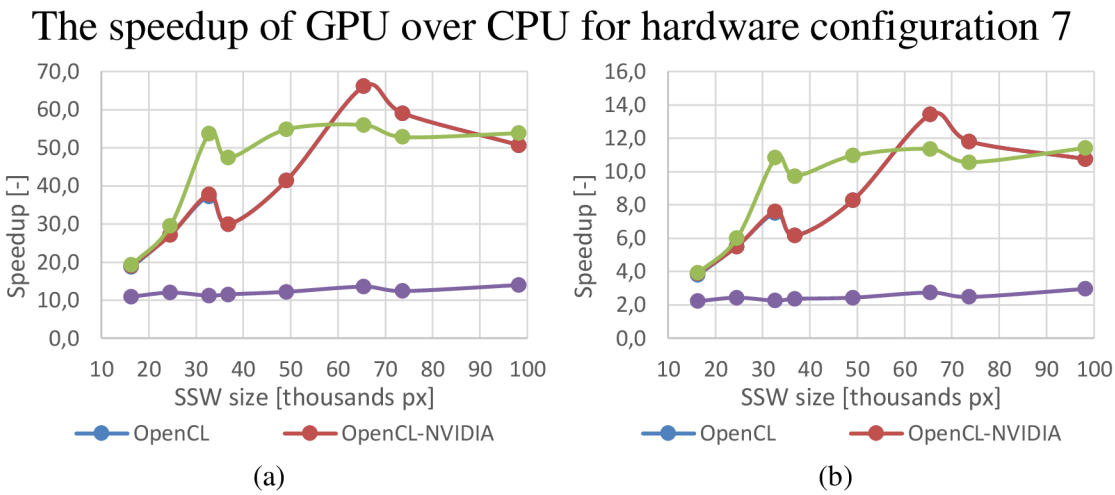


Figure 9.12: The speedup of GPU processing for different sizes of SSW over: a) single threaded CPU processing, b) OpenMP parallelized CPU processing for hardware configuration 7.

APPENDIX C: Camera paths before and after video stabilization

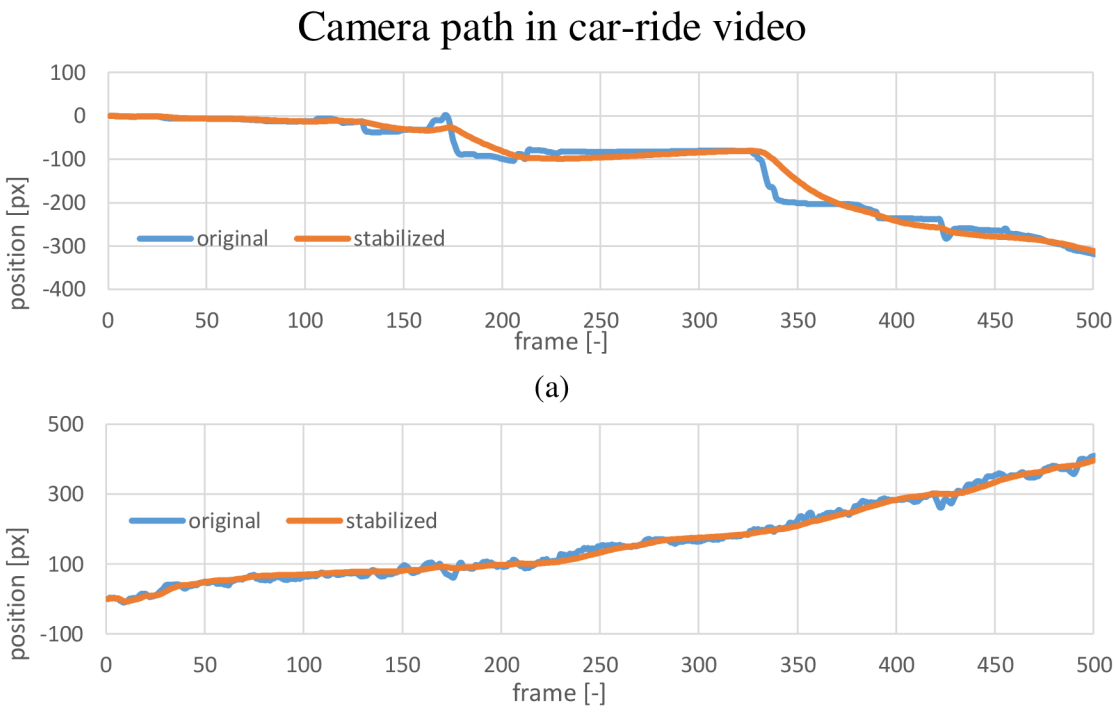


Figure 9.13: Camera path in car-ride video before and after video stabilization: a) x axis, b) y axis.

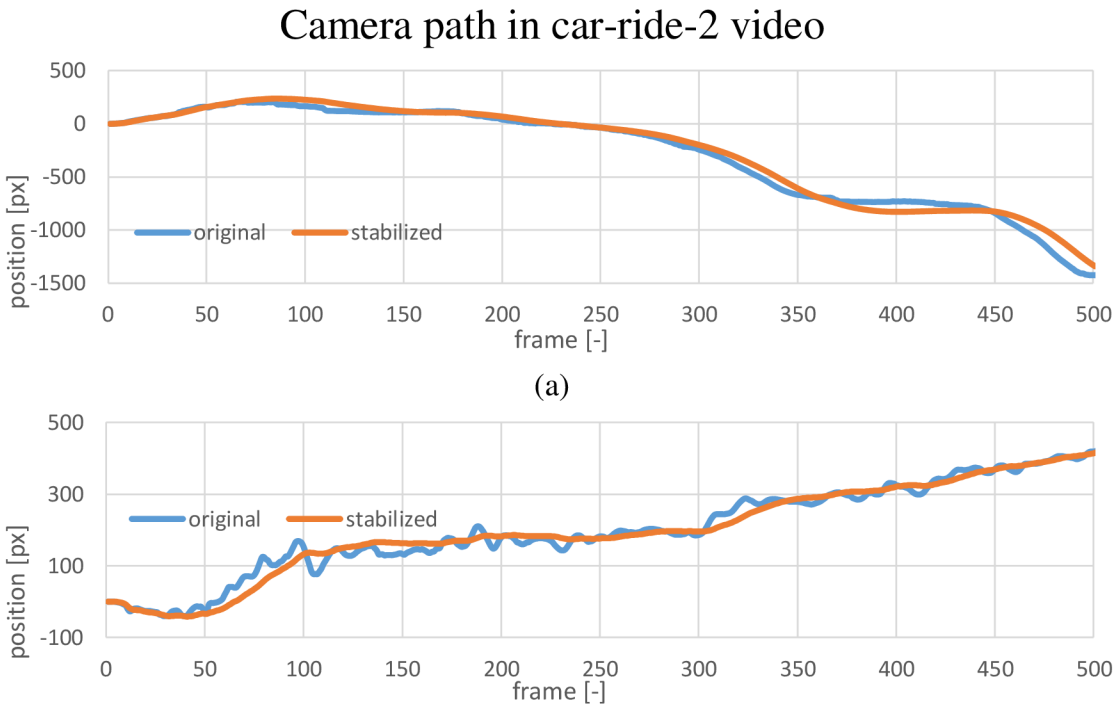


Figure 9.14: Camera path in car-ride-2 video before and after video stabilization: a) x axis, b) y axis.

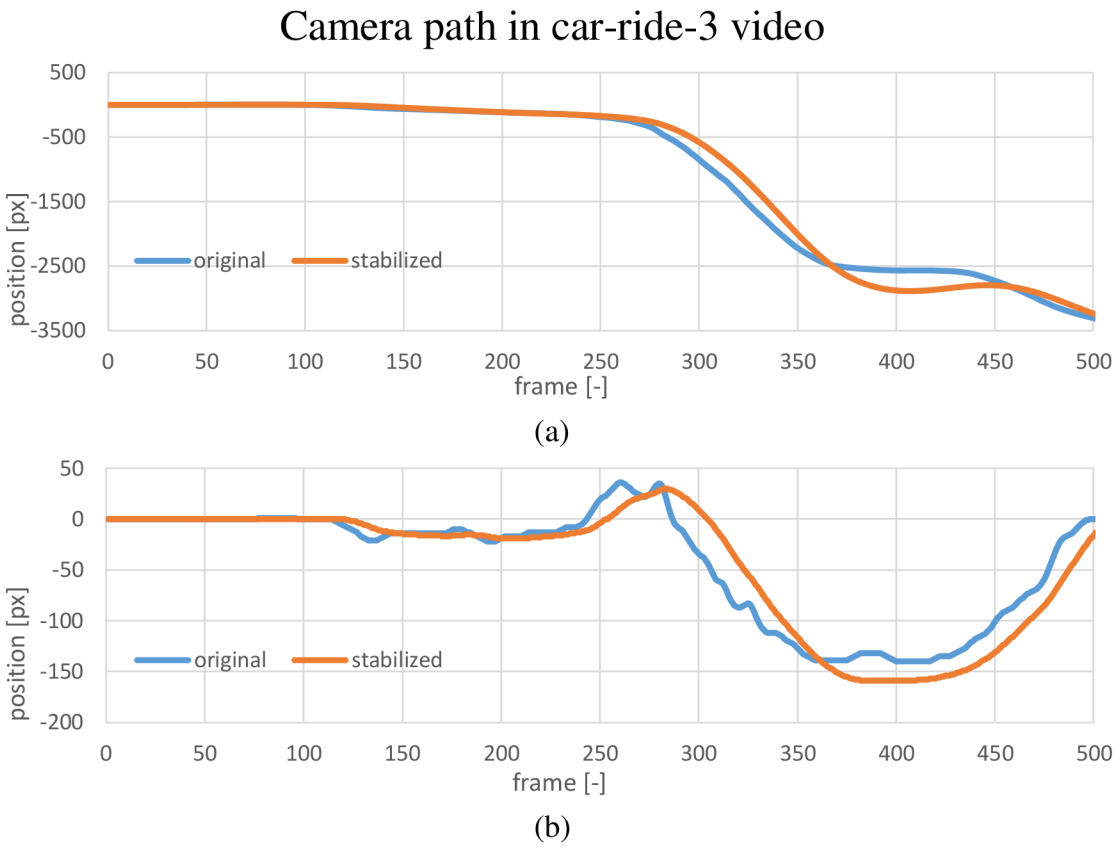


Figure 9.15: Camera path in car-ride-3 video before and after video stabilization: a) x axis, b) y axis.

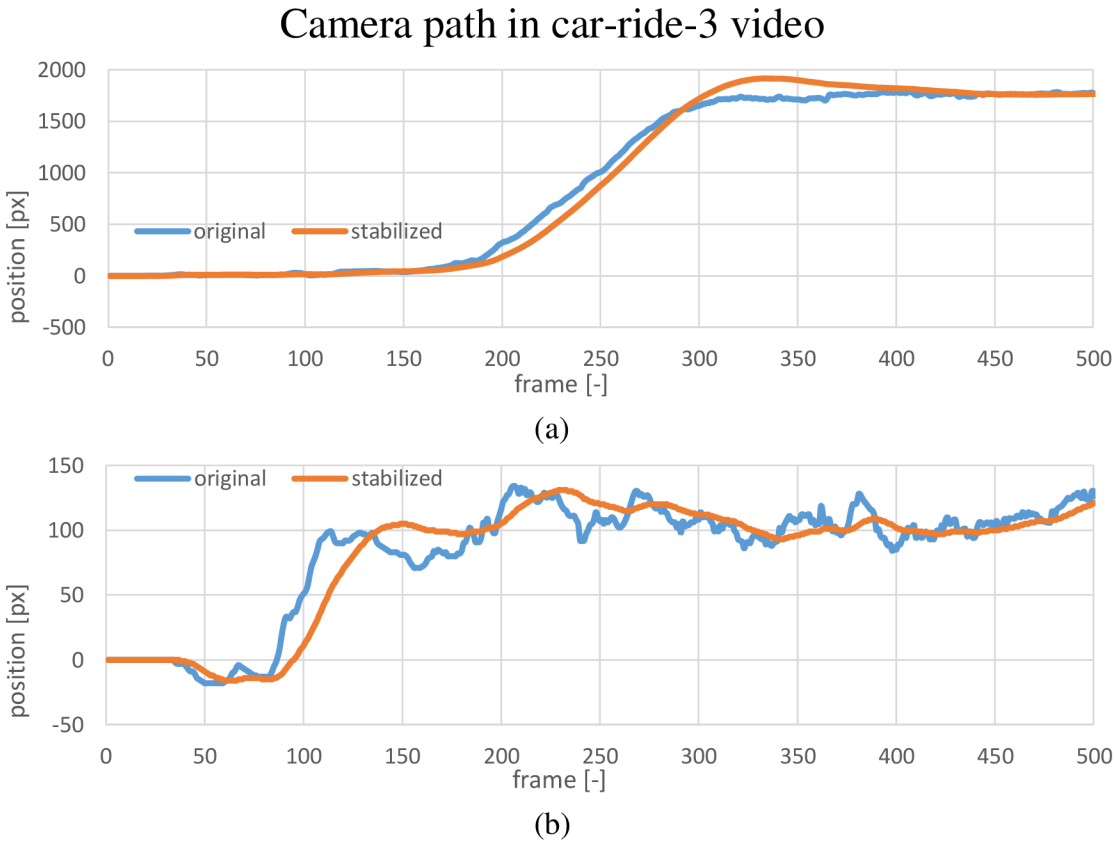


Figure 9.16: Camera path in car-ride-3 video before and after video stabilization: a) x axis, b) y axis.

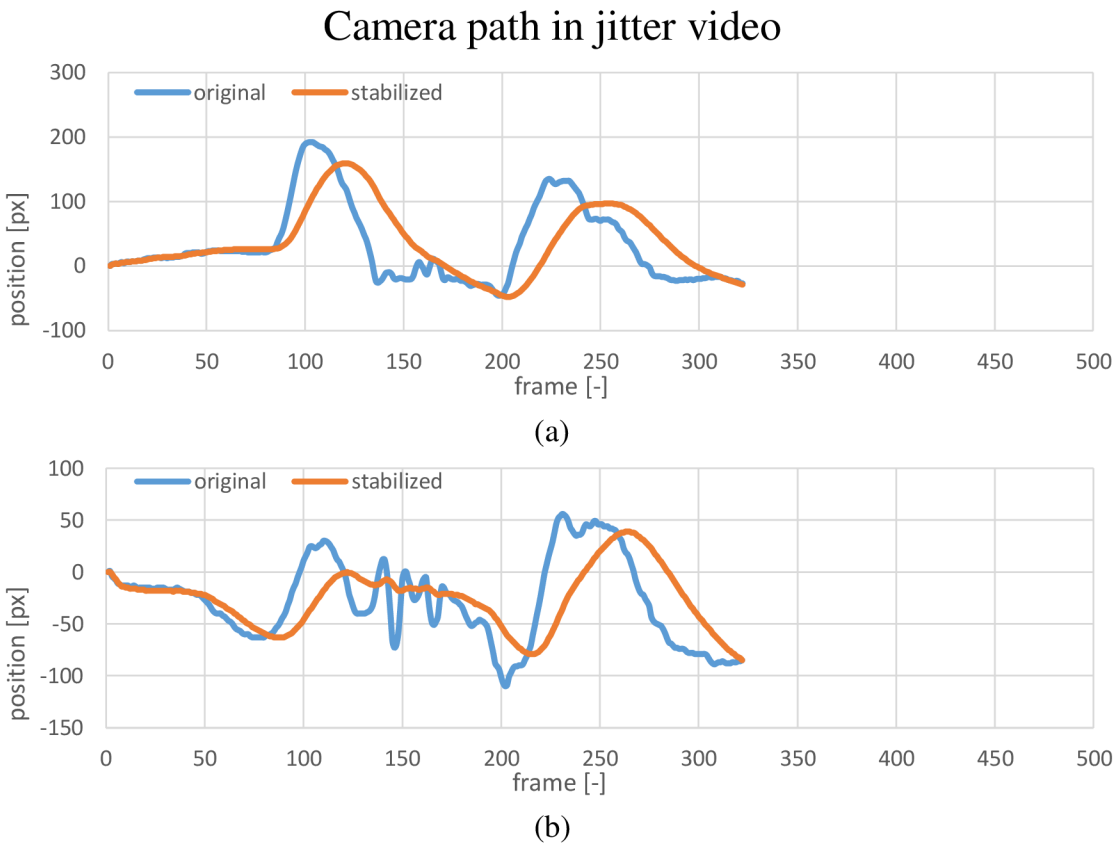


Figure 9.17: Camera path in jitter video before and after video stabilization: a) x axis, b) y axis.

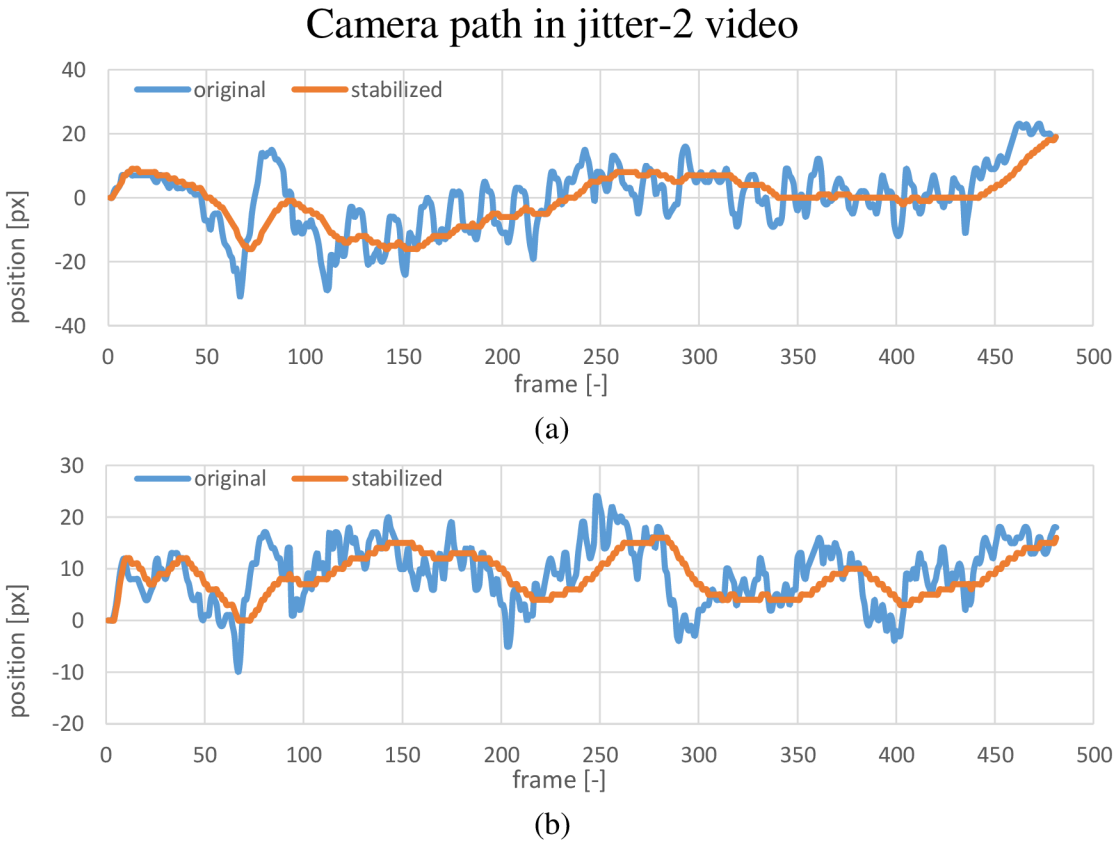
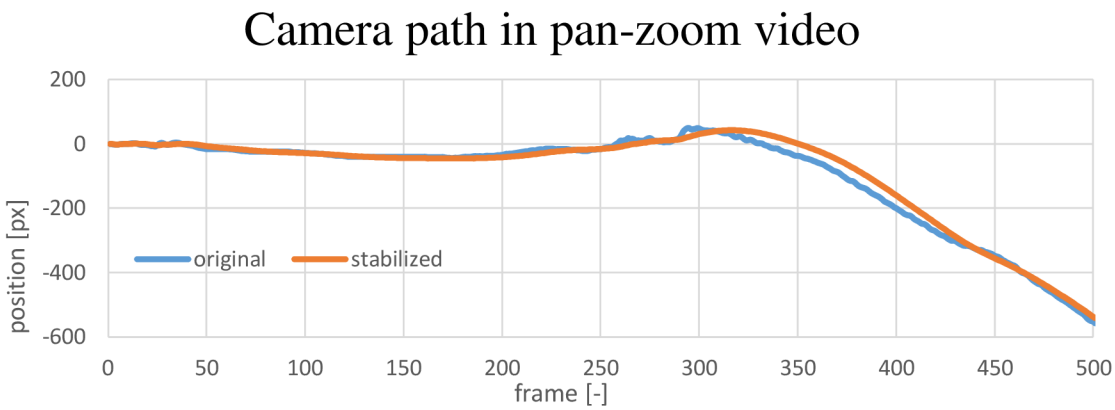
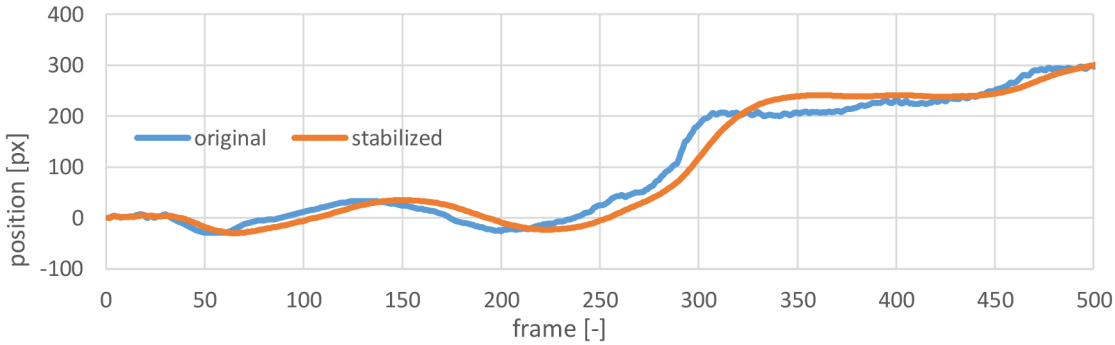


Figure 9.18: Camera path in jitter-2 video before and after video stabilization: a) x axis, b) y axis.

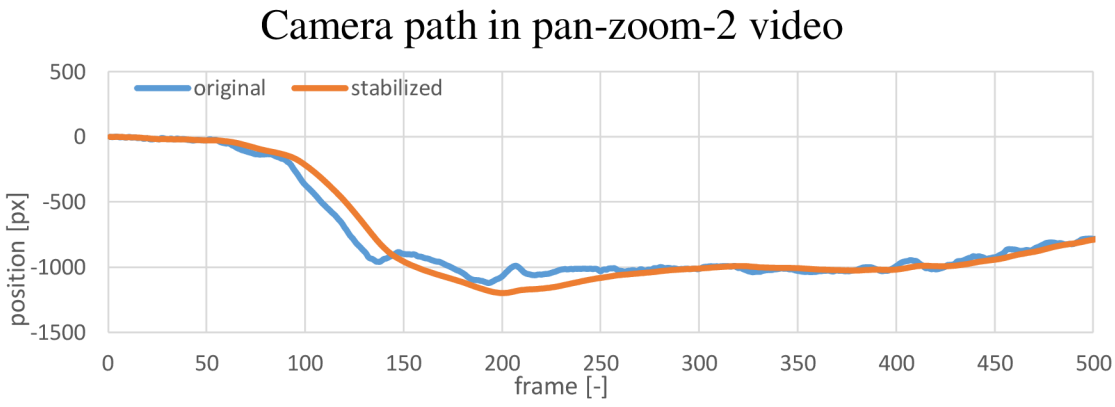


(a)

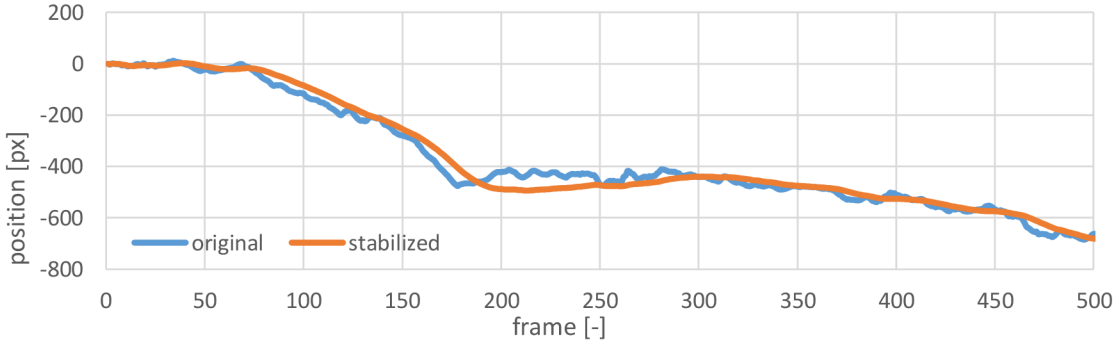


(b)

Figure 9.19: Camera path in pan-zoom video before and after video stabilization: a) x axis, b) y axis.



(a)



(b)

Figure 9.20: : Camera path in pan-zoom-2 video before and after video stabilization: a) x axis, b) y axis.

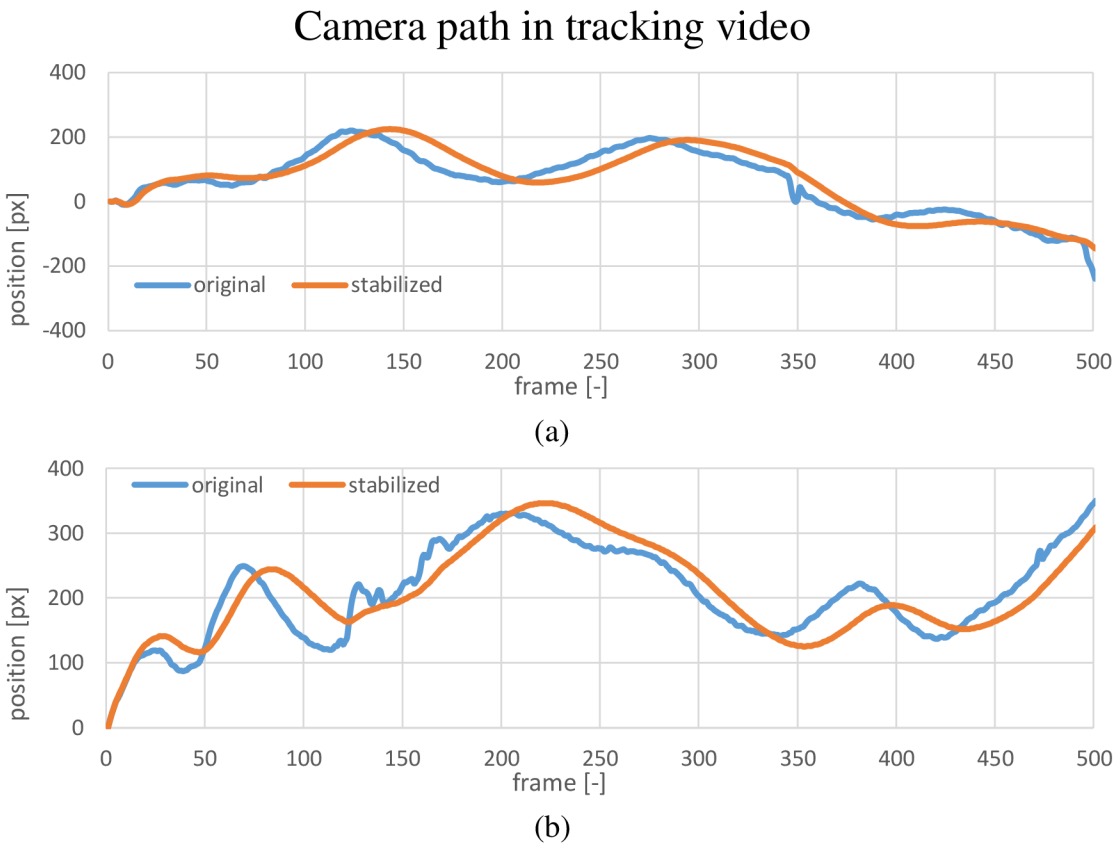


Figure 9.21: Camera path in tracking video before and after video stabilization: a) x axis, b) y axis.

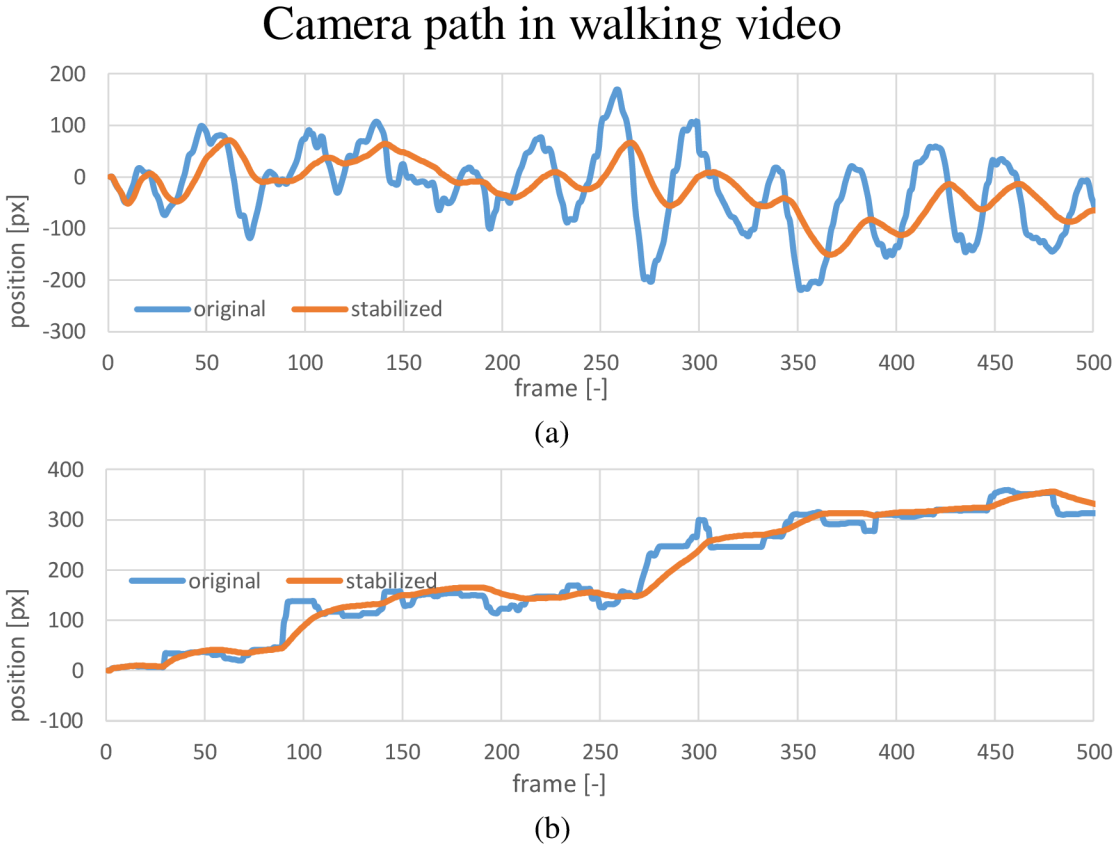
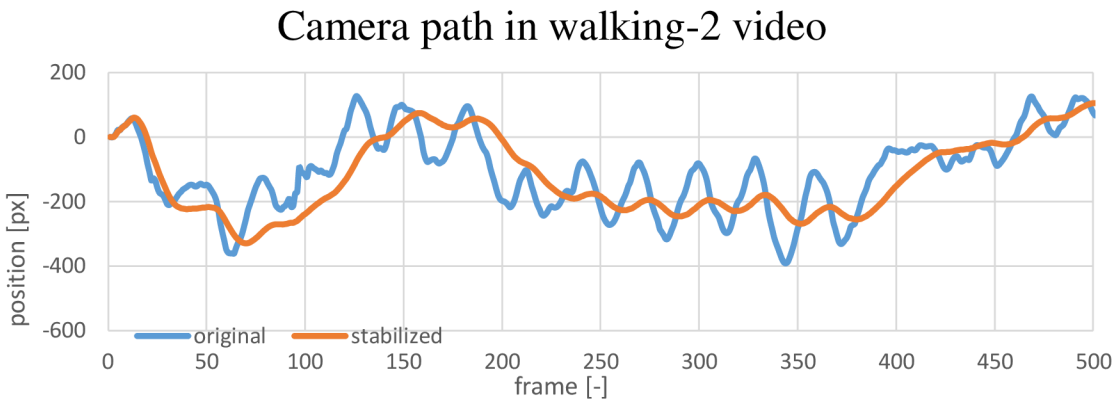
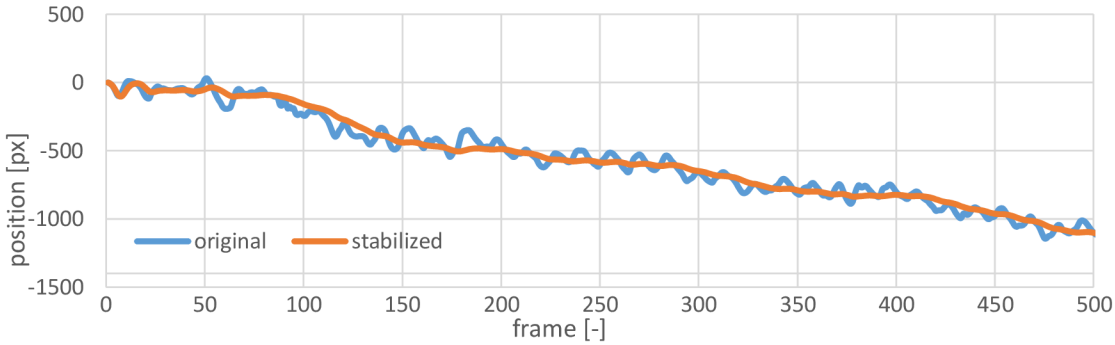


Figure 9.22: Camera path in walking video before and after video stabilization: a) x axis, b) y axis.



(a)



(b)

Figure 9.23: Camera path in walking-2 video before and after video stabilization: a) x axis, b) y axis.

APPENDIX D: Frequency analysis of camera paths before and after video stabilization

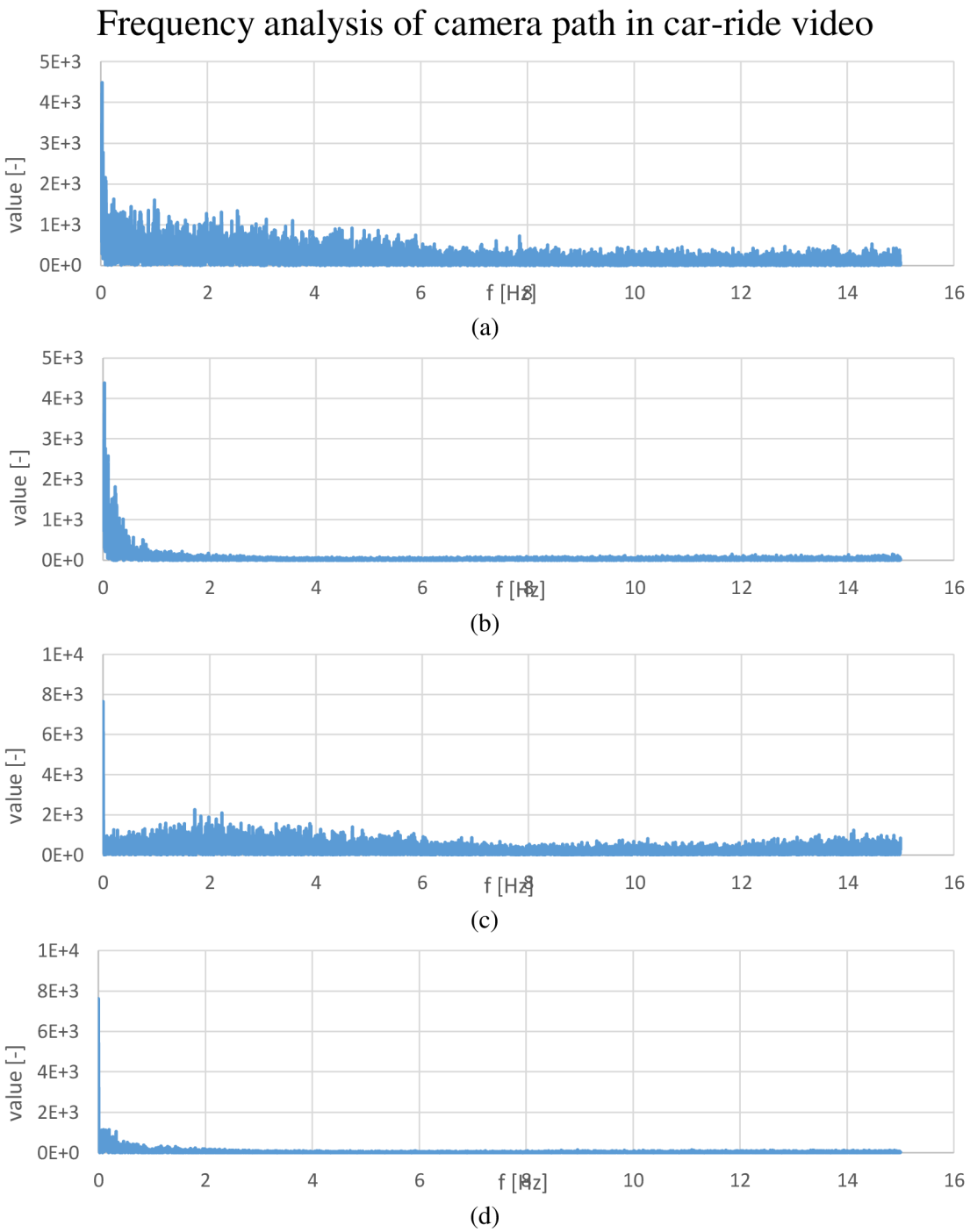


Figure 9.24: Frequency analysis of camera path in car-ride video: a) x axis before stabilization, b) x axis after stabilization, c) y axis before stabilization, d) y axis after stabilization.

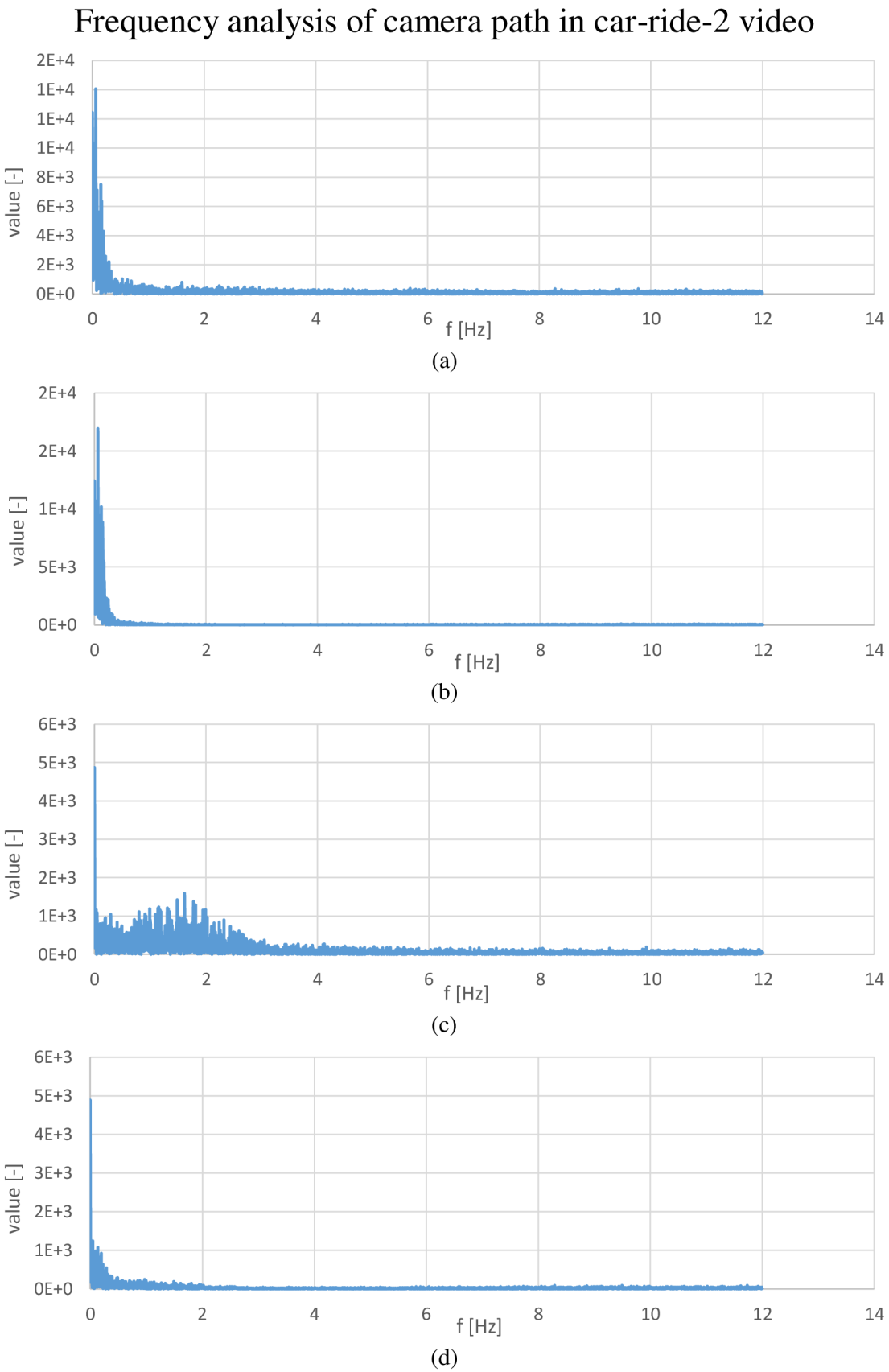


Figure 9.25: Frequency analysis of camera path in car-ride-2 video: a) x axis before stabilization, b) x axis after stabilization, c) y axis before stabilization, d) y axis after stabilization.

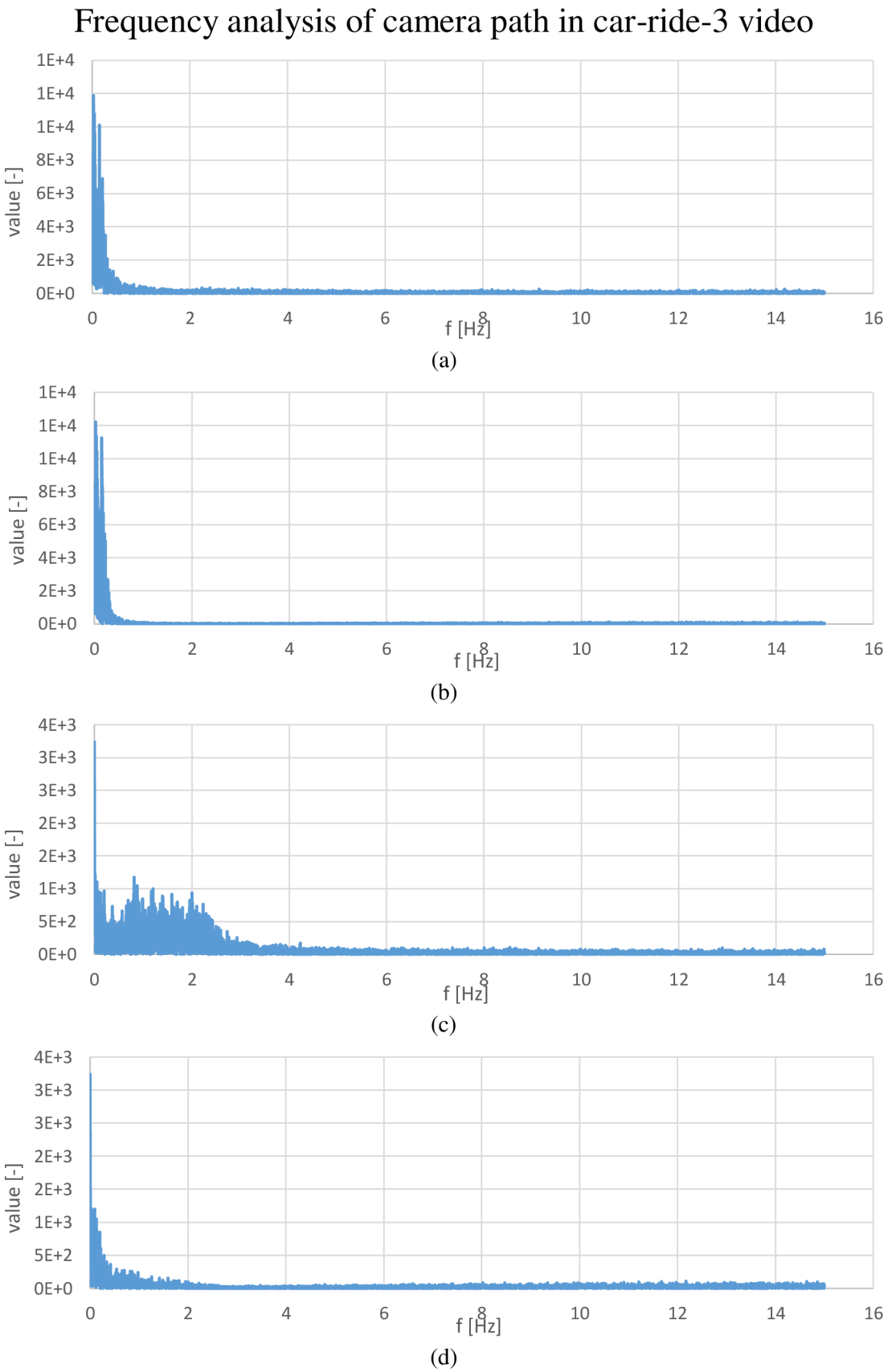


Figure 9.26: Frequency analysis of camera path in car-ride-3 video: a) x axis before stabilization, b) x axis after stabilization, c) y axis before stabilization, d) y axis after stabilization.

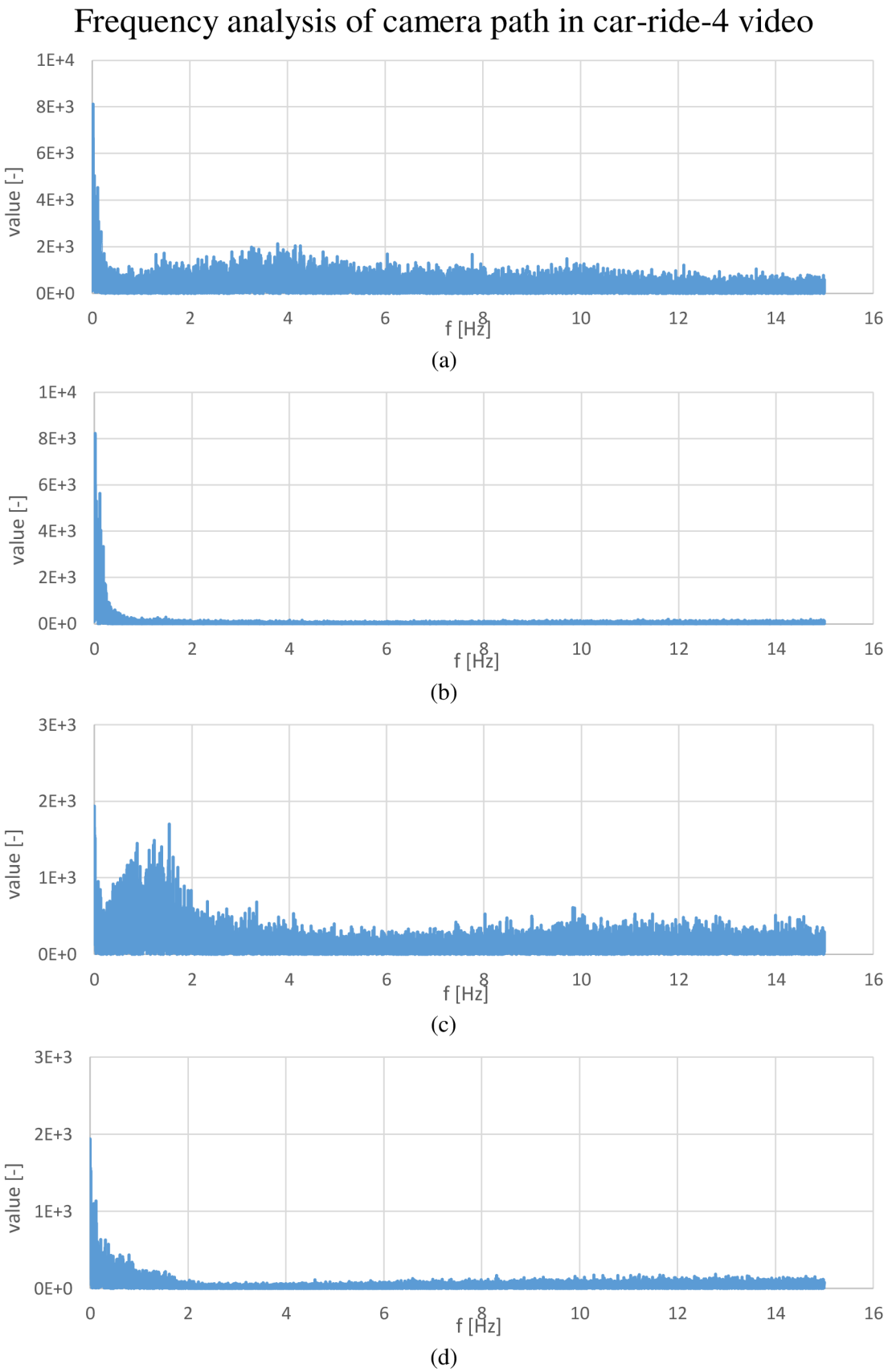


Figure 9.27: Frequency analysis of camera path in car-ride-4 video: a) x axis before stabilization, b) x axis after stabilization, c) y axis before stabilization, d) y axis after stabilization.

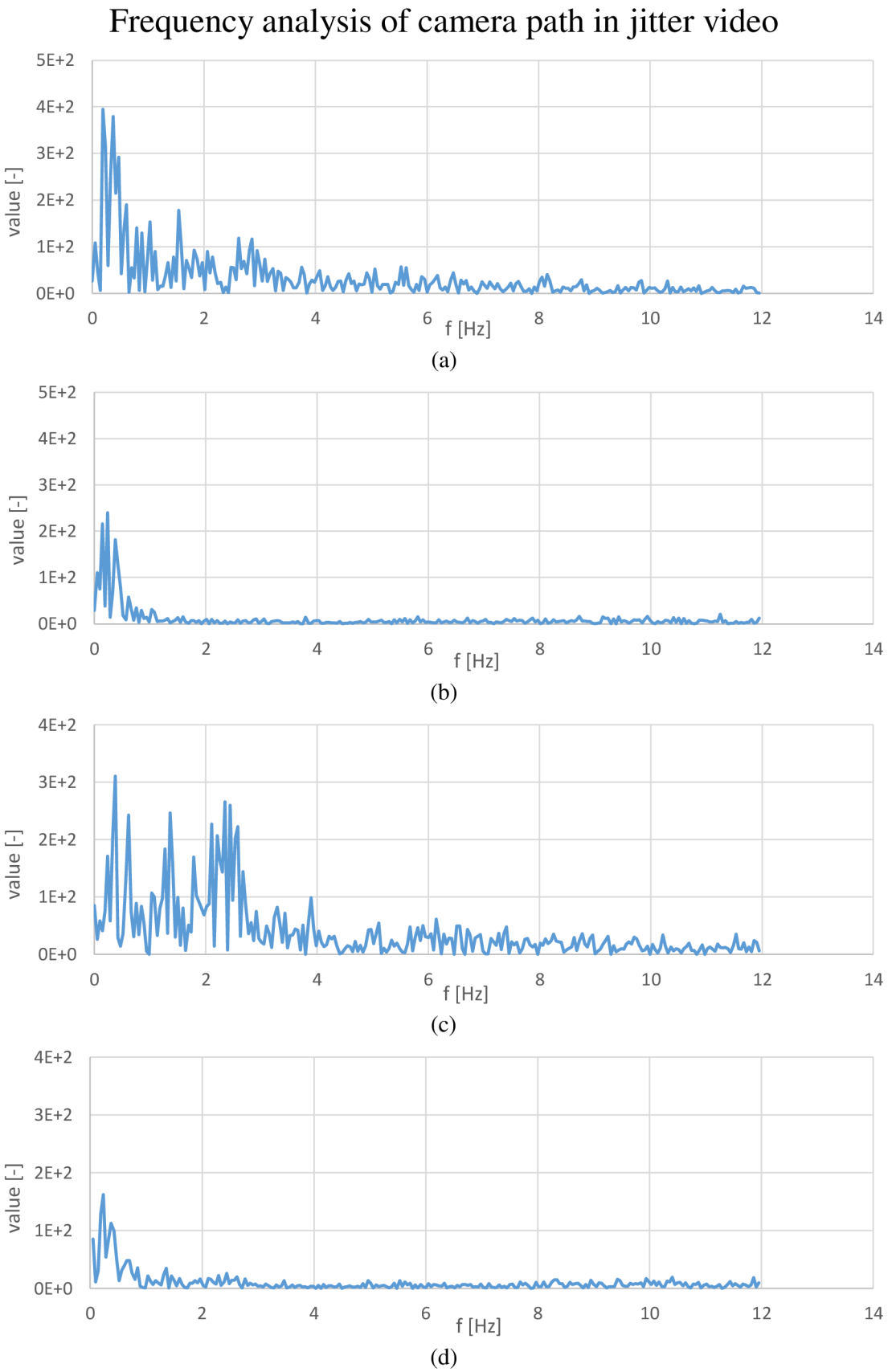


Figure 9.28: Frequency analysis of camera path in jitter video: a) x axis before stabilization, b) x axis after stabilization, c) y axis before stabilization, d) y axis after stabilization.

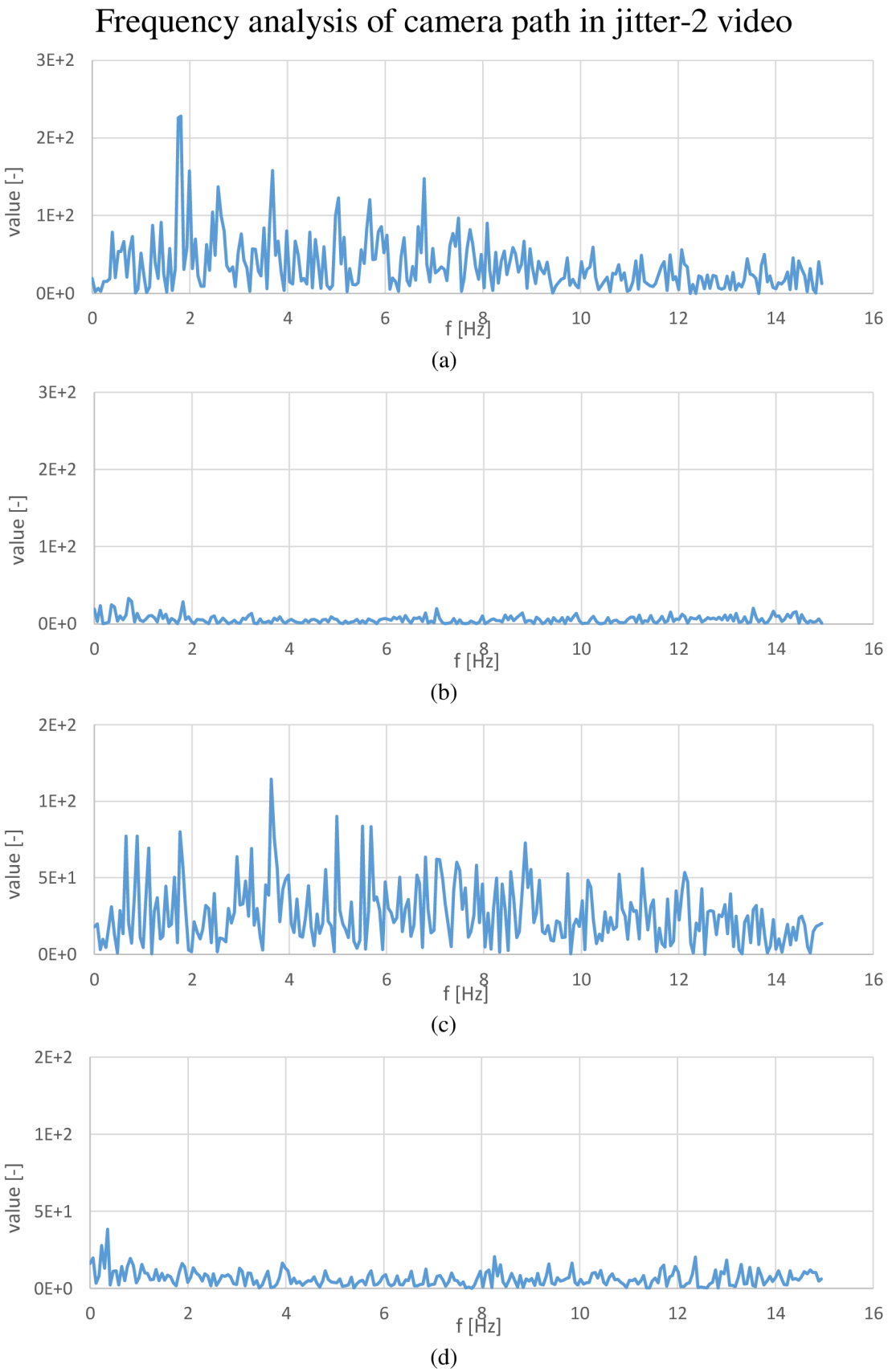


Figure 9.29: Frequency analysis of camera path in jitter-2 video: a) x axis before stabilization, b) x axis after stabilization, c) y axis before stabilization, d) y axis after stabilization.

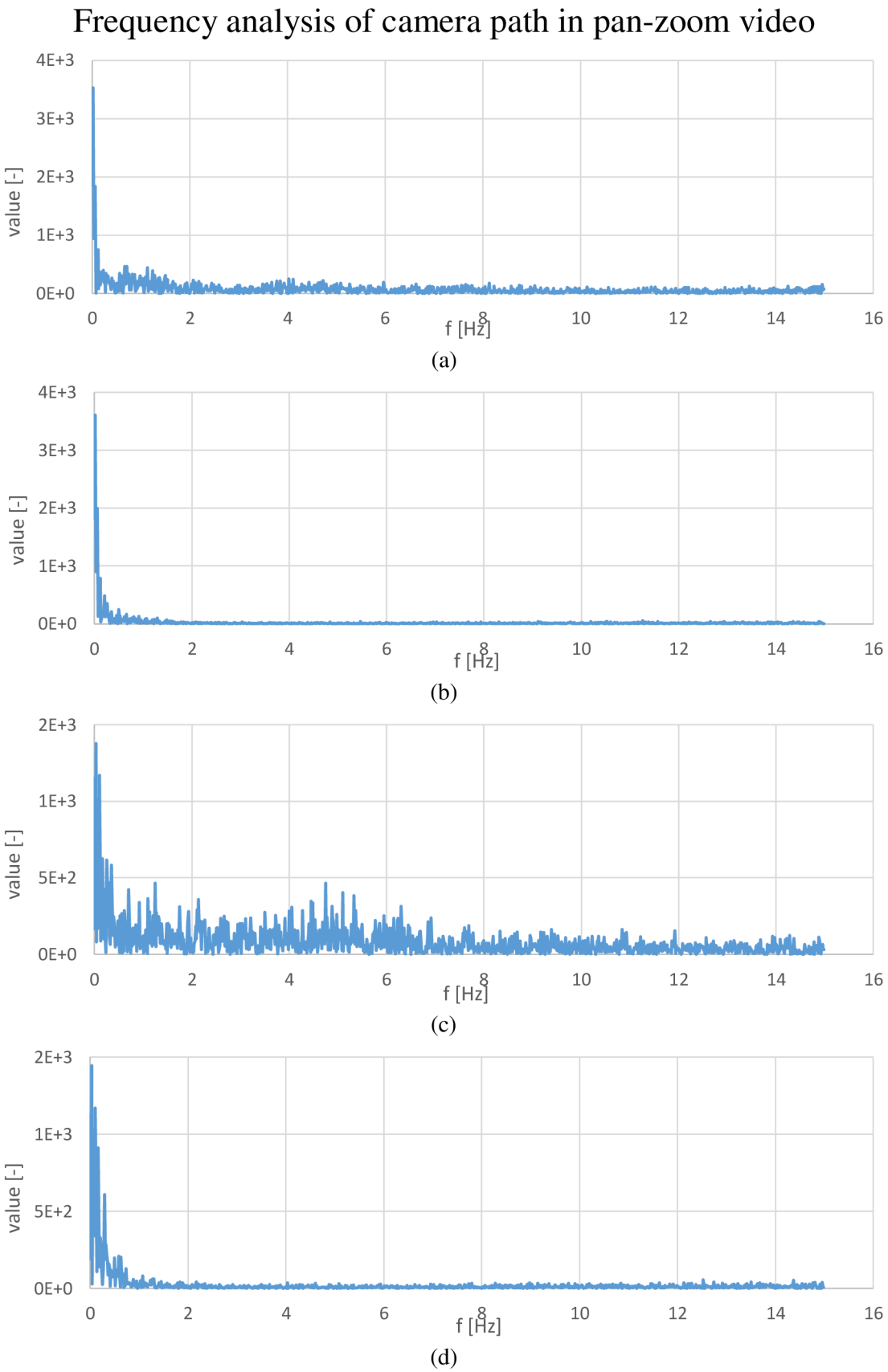


Figure 9.30: Frequency analysis of camera path in pan-zoom video: a) x axis before stabilization, b) x axis after stabilization, c) y axis before stabilization, d) y axis after stabilization.

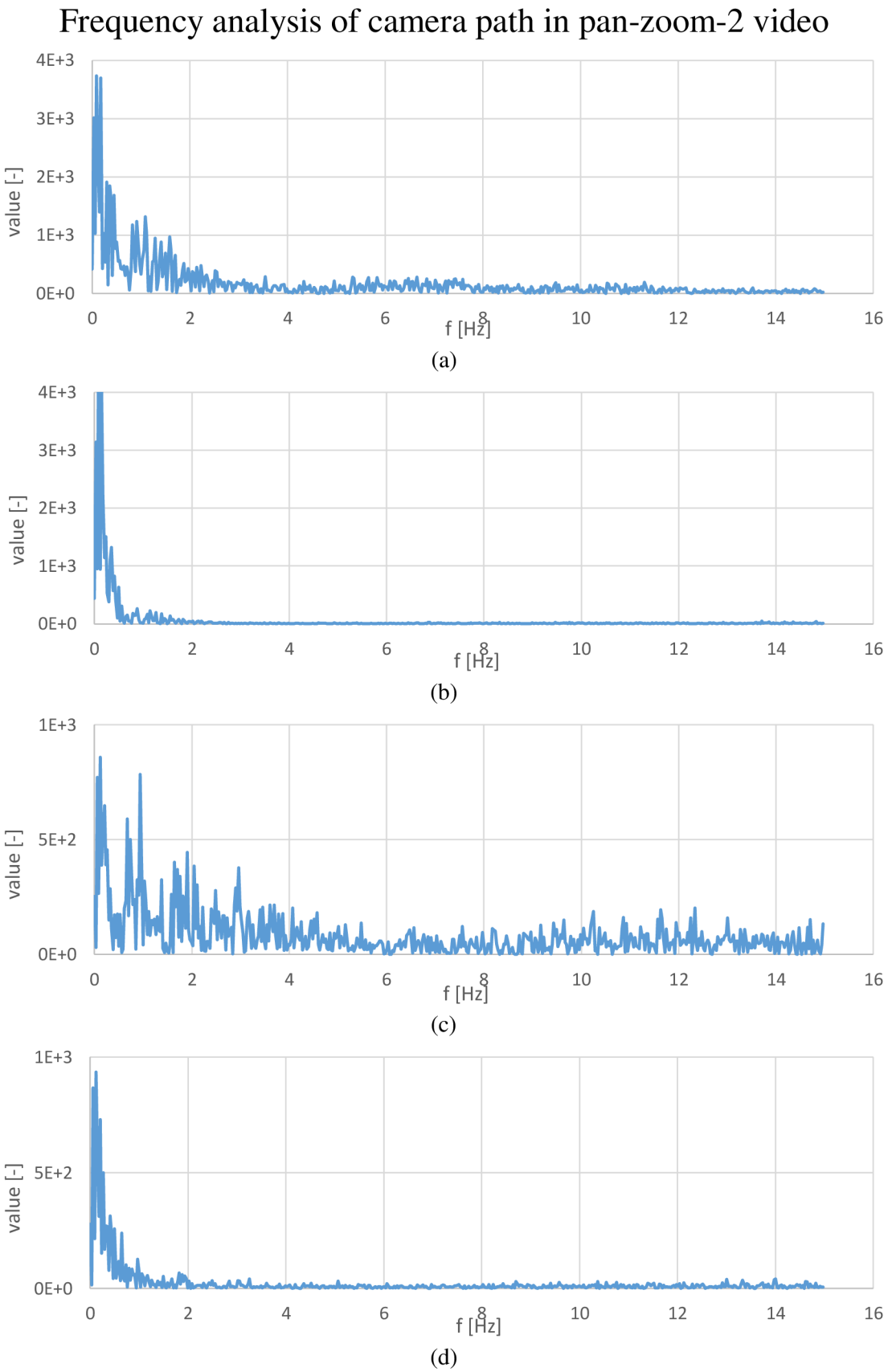


Figure 9.31: Frequency analysis of camera path in pan-zoom-2 video: a) x axis before stabilization, b) x axis after stabilization, c) y axis before stabilization, d) y axis after stabilization.

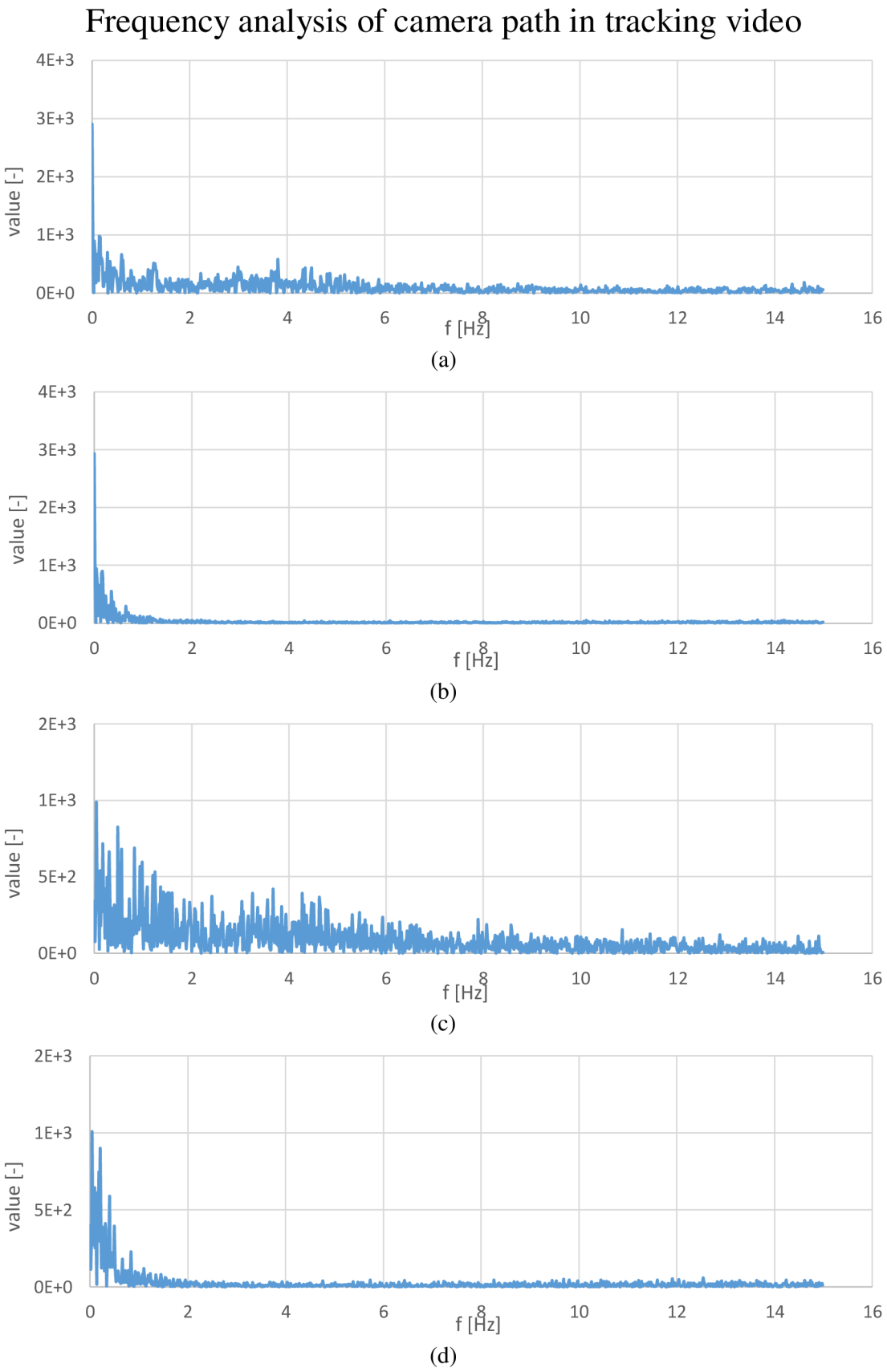


Figure 9.32: Frequency analysis of camera path in tracking video: a) x axis before stabilization, b) x axis after stabilization, c) y axis before stabilization, d) y axis after stabilization.

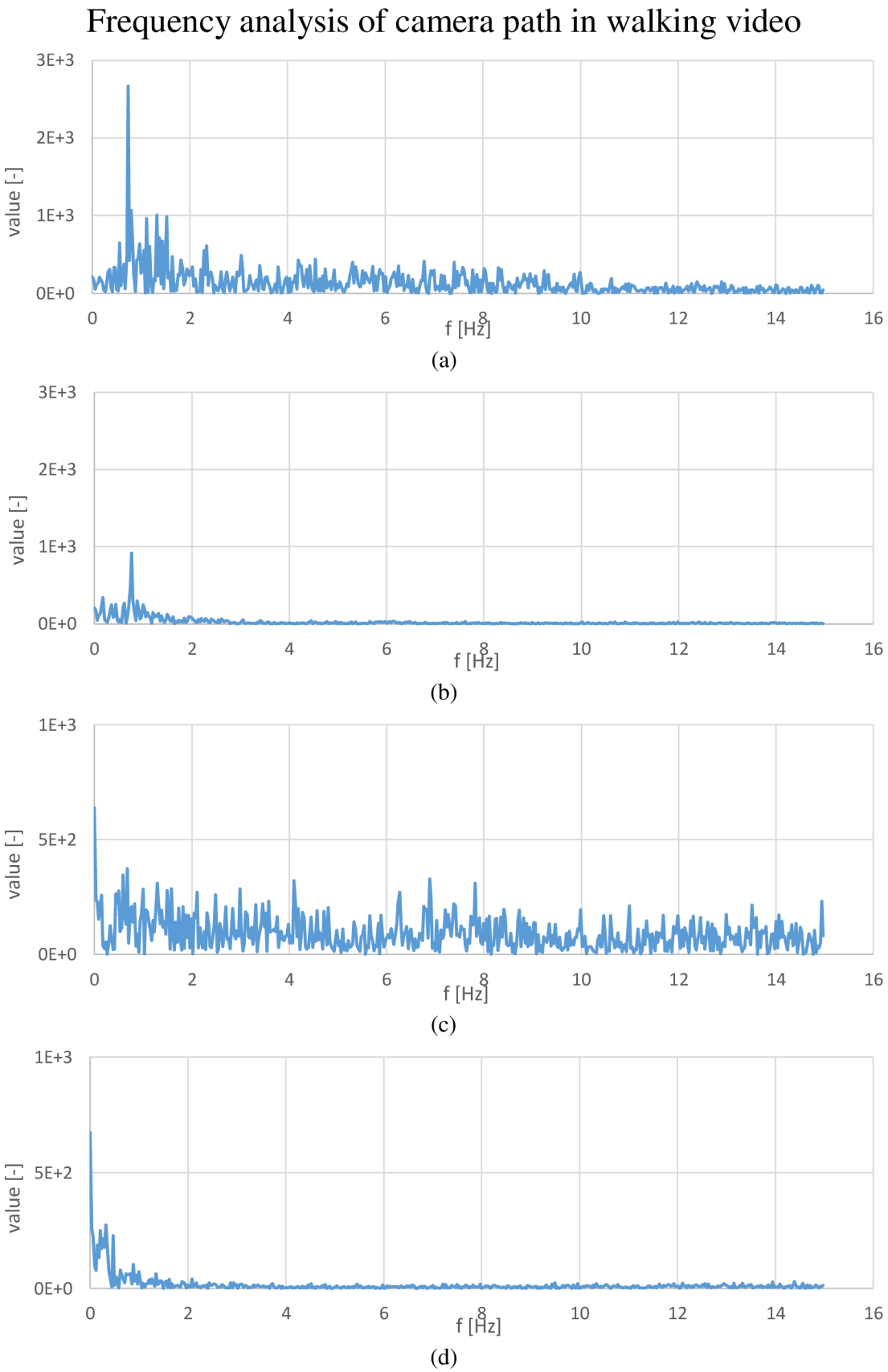


Figure 9.33: Frequency analysis of camera path in walking video: a) x axis before stabilization, b) x axis after stabilization, c) y axis before stabilization, d) y axis after stabilization.

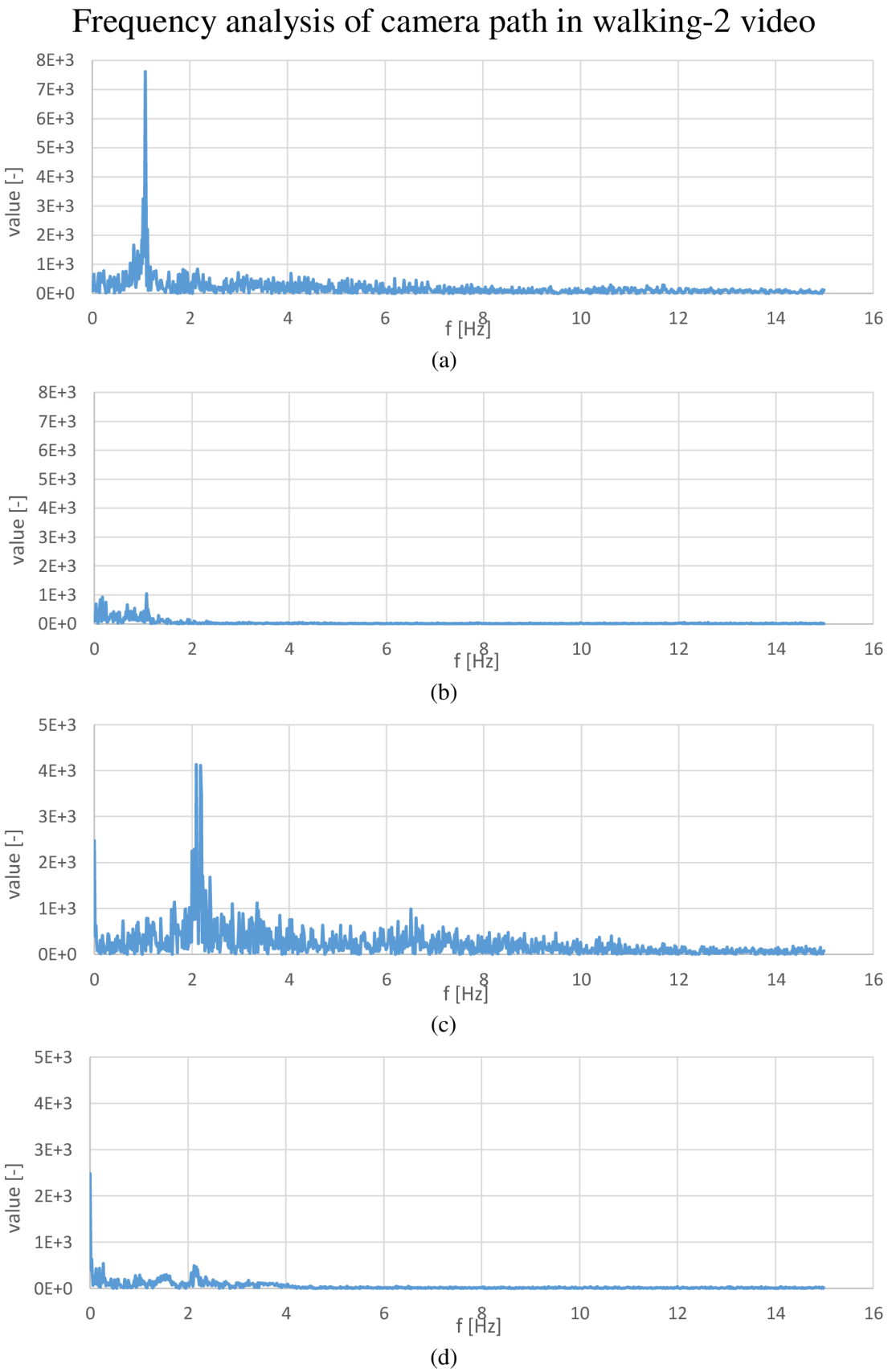


Figure 9.34: Frequency analysis of camera path in walking-2 video: a) x axis before stabilization, b) x axis after stabilization, c) y axis before stabilization, d) y axis after stabilization.