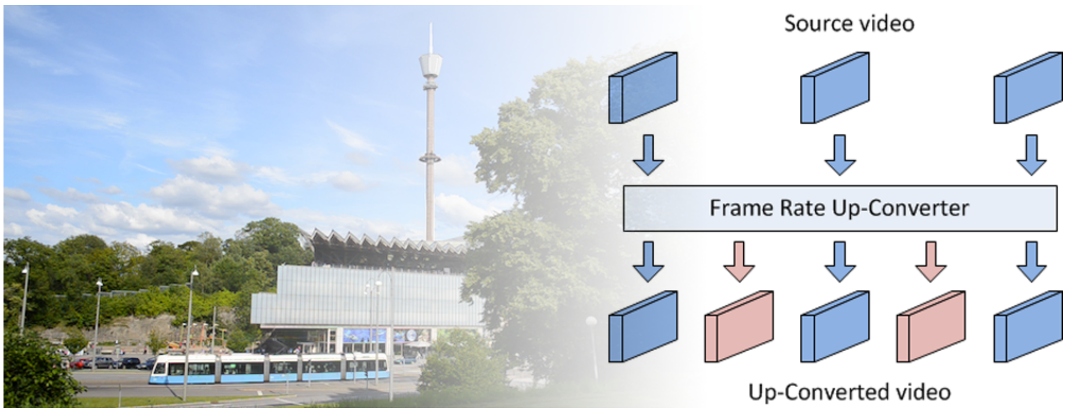
CHALMERS



Frame rate up-conversion of real-time high-definition remote surveillance video

Andreas Isberg
andreas@isberg.se
Master of Science in Software Engineering and Technology

Johan Jostell
johan.jostell@gmail.com
Master of Science in Computer Science: Algorithms, Languages and Logic

Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, April 2012

Frame rate up-conversion of real-time high-definition remote surveillance video

Examiner: Devdatt Dubhashi
Supervisors: Vinay Jethava, Henrik Engström

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover: Frame rate up-conversion (see Chapter 3) of the *Liseberg* video (see Figure 9.2).

Department of Computer Science and Engineering
Göteborg, Sweden April 2012

Abstract

Many surveillance and monitoring systems capture video using static cameras with relatively low frame rate. Low frame rates have benefits such as less storage requirements and less bandwidth used. These issues may be of even greater concern for high-definition video capture and remote surveillance. Also, a reduced frame rate makes the video appear less smooth, causing increased strain for the viewer. It is therefore desirable to increase the frame rate without increasing bandwidth use or changing equipment.

This thesis presents the design and implementation of a tool for performing frame rate up-conversion in real-time. The tool makes use of GPUs and multi-core CPUs found in modern computer systems. Techniques and frameworks such as OpenMP, SSE and OpenCL are utilized to make full use of the systems capabilities.

Frame rate up-conversion is performed in two steps: motion estimation and motion compensation. For motion estimation a number of block matching algorithms were evaluated and implemented. We present in this thesis our version of an exhaustive bidirectional search algorithm for block matching, called Full Bidirectional Search (FBDS). It uses zero motion prejudgement inspired from Adaptive Rood Pattern Search (ARPS). The tool performs bidirectional motion compensation (BDMC) on the GPU using OpenCL.

Testing was performed on three high-definition videos recorded by us, fitting the use scenario of remote surveillance. Results show increased quality compared to simple frame averaging, although with occasional block artefacts. The tool is capable of up-converting the recorded videos by a factor of three in a timespan less than 50 ms. Hence, it is viable for use in a high-definition remote surveillance system.

Sammanfattning

Många övervakningssystem använder statiska kameror med relativt låg bildfrekvens för att spela in video. Låg bildfrekvens har vissa fördelar, t.ex. mindre lagringsutrymme och mindre bandbreddsanvändning för fjärrövervakningssystem. För högupplöst video kan dessa fördelar vara ännu viktigare. Video kan upplevas som ryckig när den är inspelad med en låg bildfrekvens, vilket kan orsaka ökad ansträngning för personen som kollar på videon. Det är därför önskvärt att öka bildfrekvensen utan att öka bandbredden eller byta utrustning.

I detta examensarbete presenteras design och implementering av ett verktyg för att utföra uppkonvertering av bildfrekvens i realtid. Verktyget använder sig av grafikkort och flerkärniga processorer tillgängliga i moderna datorer. Tekniker och ramverk så som OpenMP, SSE och OpenCL används för att utnyttja systemens fulla kapacitet.

Uppkonvertering av video sker i två steg: rörelse-estimering och rörelse-kompensering. Ett flertal blockmatchningsalgoritmer implementerades och utvärderades för rörelse-estimering. Vi presenterar i denna rapport en egenutvecklad version av en fullständig dubbelriktad sökalgoritm för blockmatchning, kallad Full Bidirectional Search (FBDS), med förkontroll av statiska bildområden inspirerat av Adaptive Rood Pattern Search (ARPS). Dubbelriktad rörelse-kompensering (BDMC) utfördes på grafikkortet med hjälp av OpenCL.

Testning utfördes både på tre egeninspelade högupplösta sekvenser som passar användningsområdet för fjärrstyrd övervakning. Resultaten visar att videokvaliteten ökar jämfört med enklare tekniker, även om vissa blockartefakter förekommer. Verktyget klarar uppkonvertering med faktor tre på de egeninspelade videosekvenserna, inom en tidsrymd kortare än 50 ms. Detta visar att verktyget lämpar sig för användning inom fjärrövervakningssytem med högupplöst video.

Acknowledgments

We would like to thank our supervisors Vinay Jethava and Henrik Engström for support and advice during the writing of this thesis. Vinay has been a great help in sifting through the relevant literature and providing insightful comments during the development. Henrik's expertise in x86 assembly has been a large contribution in our optimization efforts. We would also like to thank Erik Levin who was kind enough to let us borrow his camera for filming test videos. Finally we would like to thank our friends and families for all their support.

Johan and Andreas

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Frame rate conversion

Many surveillance and monitoring videos are captured using static cameras with relatively low frame rate. The low frame rate of the videos is economic both in terms of the monitoring equipment and storage requirement. Other factors may influence the necessity of keeping a low frame rate, such as limited transmission bandwidth for a remote surveillance solution. Storage space may be an issue for high-definition (HD) resolution video capture. However, this poses a challenge in terms for the manual inspection of such video feeds, since low frame rate videos frequently consist of non-smooth motion artefacts. This can lead to increased strain for the viewer. Therefore, a key challenge towards improving the user experience lies in successfully obtaining a smoother video by increasing the frame rate.

In this thesis we present a solution aimed at removing these issues for remote surveillance HD video feeds, by a technique known as frame rate up-conversion.

1.2 Thesis contributions

The main contributions of this thesis are:

- The design and implementation of a frame rate up-conversion tool (FRUCT), specialized for use with remote surveillance video.
- A comparative study of some well-known, published unidirectional block matching algorithms (BMAs) and their suitability for use with frame rate up-conversion of remote surveillance video.
- Research and data on the topic of how Open Computing Language (OpenCL) can be used for GPU-acceleration of frame rate up-conversion.

- Full Bidirectional Search (FBDS), our version of an exhaustive search BMA, using ideas from Adaptive Rood Pattern Search (ARPS) for early termination of search in static areas of video.

1.3 Disposition

The thesis is organized as follows:

- Chapter 2 presents the problem description and goals, as well as the scope and delimitations.
- Chapter 3 describes the theory behind both naive and more advanced ways of performing frame rate up-conversion. A short summary of some of the existing research in the area is also given.
- Chapter 4 introduces a number of motion estimation algorithms used in this thesis.
- Chapter 5 gives details on the motion compensation algorithms used in this thesis.
- Chapter 6 familiarizes the reader with the terminology and processing flow of OpenCL.
- Chapter 7 contains information about evaluation methods by which the result can be measured.
- Chapter 8 shows a detailed look on our implementation of a frame rate up-converter.
- Chapter 9 displays the results after frame rate up-conversion.
- Chapter 10 sums up the thesis with conclusions and notes on possible future work.

2. Problem description

2.1 System specifications

The focus of this thesis is the design and implementation of a frame rate up-conversion tool (FRUCT). The up-converter will be part of a larger system, as can be seen from Figure 2.1. It will work as a "middle-man" between the decoder and renderer software.

Input to the FRUCT is raw 1080p video at 20 frames per second (FPS) corresponding to the intended remote surveillance scenario. Output should be up-converted video in the same format at 60 FPS. In other words: two interpolated frames should be inserted for every original frame. Performance and quality should be high enough to be successfully applied in real-time.

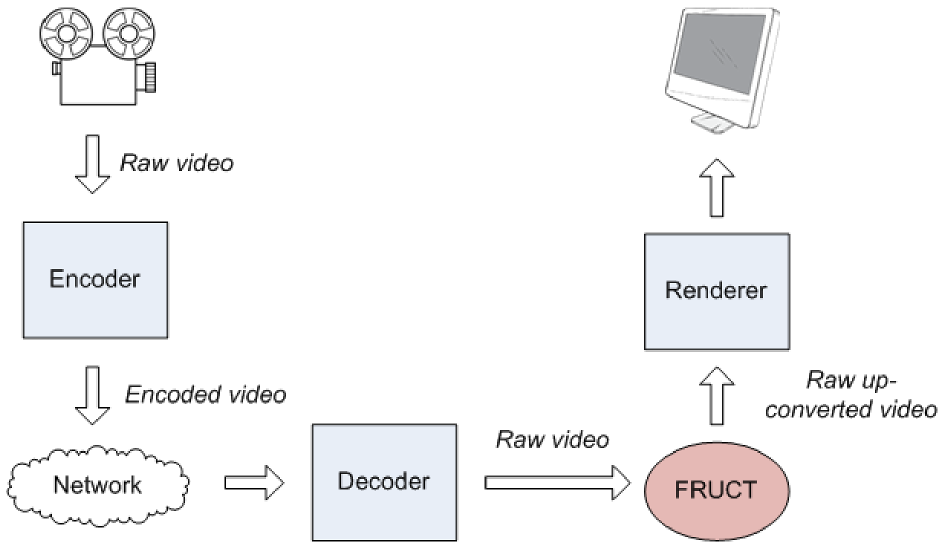


Figure 2.1: Video path from source to screen with the FRUCT.

The area of application, remote surveillance with the settings described above, adds a number of items to the specification. These may not be as important for a general-purpose implementation of a FRUCT.

2.1.1 Software-based implementation

The FRUCT should be implemented as software, not as hardware. Many hardware-based solutions exist today, especially for use in real-time and for high-definition (HD) video, e.g. by Lee & Nguyen (2010). Software-based solutions are however easier to maintain and can more easily interact with other parts of a software-based remote surveillance system.

2.1.2 Definition of real-time

The FRUCT should be capable of up-converting a received video stream from 20 FPS to 60 FPS (an up-conversion factor of 3) in real-time. For true real-time behaviour, frames constructed by the FRUCT must be completed within 1/60 seconds, roughly 16.7 ms, for this to be fulfilled.

The real-time demand in the case of remote surveillance also imposes a restriction on the delay between a video frame being captured in the camera and the the frame being displayed on-screen. This delay should ideally be kept as low as possible, and in this specific case no more than one (1) second. The variable transmission delay poses additional difficulty to put a hard limit on the maximum delay the FRUCT could introduce in the system. The FRUCT may require some initial buffering or setup routines, introducing a delay before the first received frame is written to the output channel.

A solution involving a relaxation on the real-time demand is to produce frames to be placed in an output buffer with a predetermined size, typically equal to the up-conversion factor. For a buffer size of n and frame rate f, the time limit is then equal to $\frac{n}{f}$, which for $n = 3$ and $f = 60$ calculates to 50 ms. In other words, we have 50 ms to fill the buffer with three frames where one is an original frame and two are constructed by the FRUCT. Since the original frame can be copied directly to the output buffer, most of the 50 ms can in practice be used for construction of the two new frames.

2.1.3 High-definition

Video received by the FRUCT will be in HD, 1080p. Video capture at such high resolutions produce large amounts of data to be encoded, transmitted and decoded. A majority of the FRUCTs described in existing literature have primarily been tested on low-resolution video formats with less focus on real-time capability.

2.1.4 Static scenes

The intended usage scenario for the FRUCT is stationary cameras and mostly static outdoor scenery. A more general purpose FRUCT must be able to handle video motion as well as camera motion (panning, tilting, etc.), which in the latter case is known as global motion. Also, cuts and scene changes also never happen for continously filming stationary cameras. There is however the case of events such as sudden changes in cloud coverage or weather which may give effects to a large portion of the visible area, which have to be taken into account.

2.1.5 Remote objects

Objects found in video from the intended usage scenario are a sizeable distance from the camera; from ten to a hundred meters up to several hundred meters. The distance means that they will appear on a relatively small area of the video frame. This opens up for both possibilities and problems: the motion of these objects from frame to frame in terms of pixels on the screen will be fairly small. However, depending on which method used for motion estimation (ME) there might be problems finding true motion of the image areas covered by the objects.

2.1.6 High reliability

For the video to be a reliable source of information, especially vital if it is to be used for security-critical surveillance, the processed video should stay as close to the original as possible. The frames used for interpolation are not to be manipulated in any way, and shall be shown "as-is". Visible artefacts in the video should preferably be kept to a minimum, more so in regions of interest.

2.2 Scope and delimitations

In order to keep the focus on areas of importance, the following decisions were made:

- The implementation should not have to be a complete product, a working prototype would suffice.
- The prototype should be implemented in software, not hardware. Hardware solutions have been proven to be successful. However, a well implemented software solution should still be able to produce good results, with the benefits of being more easily implemented and maintained.
- The prototype should focus on high performance and quality for small moving objects (far away, thus having small motion), rather than large objects close to the camera. This focus is due to the intended use for remote surveillance (remote in both senses; the video streaming to a remote location and objects under surveillance being a sizeable distance from the cameras).
- The prototype should not handle any encoding, decoding or rendering except for possibly obtaining motion vectors from the H.264/MPEG-4 AVC (H.264) bitstream.
- The prototype should only be used with video from stationary cameras, as opposed to a solution capable of up-converting any kind of video.
- The prototype should work with 1080p video resolution (or lower), in Y'CbCr 4:2:0 color space (raw, uncompressed video).

3. Frame rate up-conversion

Increasing the frame rate, i.e. up-conversion, is done to enhance the visual experience of video with low frame rate. Video at 30 frames per second (FPS) is clearly smoother than video at 20 FPS, especially for video with much motion. Up-conversion is performed by adding new frames to a video from existing ones, as Figure 3.1 illustrates. Much research has been put into doing this in a fast and efficient manner.

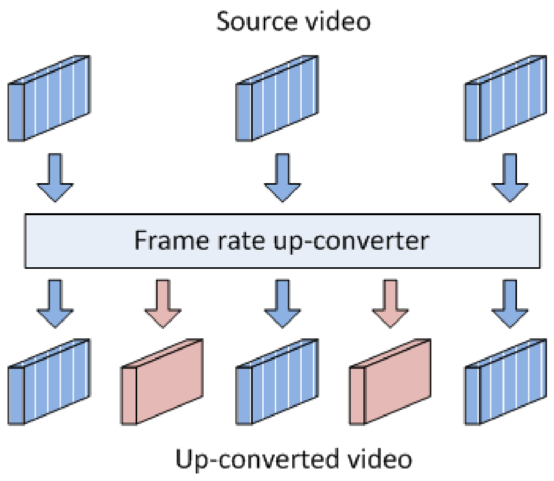


Figure 3.1: The frame rate up-converter illustrated as a black box. Input is a source video and output is an up-converted video.

Existing research in the area falls under two broad categories of solutions: hardware-based and software-based. Hardware-based solutions using Field-Programmable Gate Arrays (FPGAs) perform well for real-time 1080p video (Lee & Nguyen 2010). Such solutions are however beyond the scope of this thesis.

3.1 Frame repetition

The simplest way of increasing the frame rate is by repeating every frame F k (i.e. up-conversion factor) number of times. Equation 3.1 below describes frame repetition (FR), where $1 \leq j \leq k$, $j \in \mathbb{Z}^+$ and t being a time index.

$$F_{t-\frac{j}{k}} = F_{t-1} \tag{3.1}$$

For example, $F_{t-\frac{1}{2}}$ is the interpolated frame between frames F_{t-1} and F_t. This method is extremely fast but gives a poor visual experience.

3.2 Frame averaging

A slightly better way to increase the frame rate is by interpolating all pixel values for a new frame from two neighbouring frames. Figure 3.2 below illustrates frame averaging (FA).

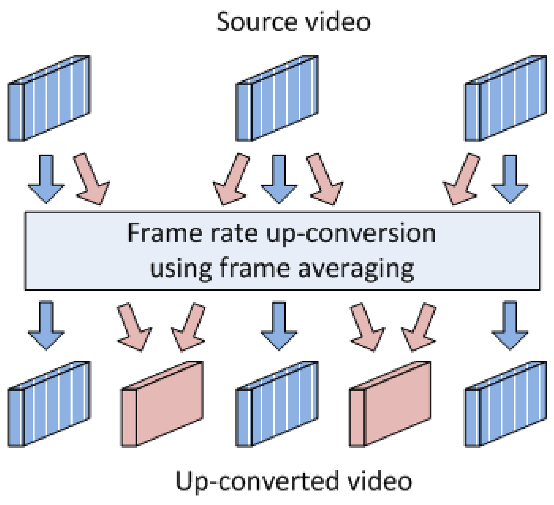


Figure 3.2: Up-conversion using FA. Every new frame (non-striped) is averaged by the two neighbouring frames (striped).

Let $P_t(x, y)$ be the value of a pixel with coordinates (x, y) in frame f_t, t being a time index. Then for all (x, y) inside the frame boundaries of f_t, with

$1 \leq j < k$ and $j \in \mathbb{Z}^+$, the pixel values for an intermediate interpolated frame $f_{t-\frac{j}{k}}$ is given by Equation 3.2.

$$P_{t-\frac{j}{k}}(x,y) = \left(\frac{j}{k}\right) P_{t-1}(x,y) + \left(1 - \frac{j}{k}\right) P_t(x,y) \tag{3.2}$$

This method is very fast, and gives a smoother visual experience than FR. It is however not suitable for high-motion video, as seen in Figure 3.3. Smoothness is gained at the loss of sharpness.

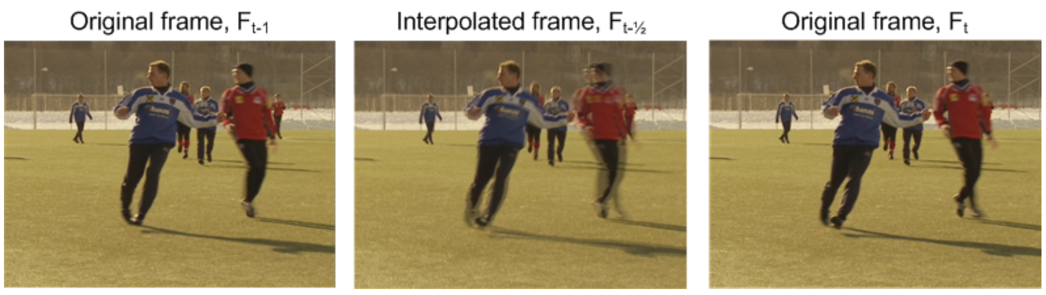


Figure 3.3: Up-converting a high-motion video using frame averaging. This video is the *Soccer* standard benchmarking video[1].

3.3 Advanced methods

The main challenge in frame rate up-conversion is how to efficiently and correctly perform motion estimation (ME) and produce the intermediate frames using motion compensation (MC). Motion-compensated frame interpolation is a term often used to describe this process.

There are many existing algorithms for motion estimation. Approximating the optical flow, motion of individual pixels from one frame to another, is one way. However, it has traditionally not been used in video encoding for two purposes. It is a relatively slow method, and encoding the motion of individual pixels would require too much data to be feasible. Instead, methods based on the movements of rectangular blocks of pixels are often used. Since block matching is the de facto most-used family of motion estimation algorithms in use, fast algorithms have been developed during the years. Some

[1]Standard videos can be found at http://media.xiph.org/video/derf/

of the block matching algorithms (BMAs), of which a few were selected for implementation in this thesis, are presented in Chapter 4.

Once the video motion has been estimated other algorithms may be used to utilize the so-called motion vectors of each block to produce a sharp intermediate frame. Algorithms for this are presented in Chapter 5. In the following section we give a brief overview of the research that has been done in the area of frame rate up-conversion.

3.4 Existing research

ME is often done with various BMAs, to reduce computational complexity. Some form of BMA is performed in encoders for all video coding standards, e.g. H.264/MPEG-4 AVC (H.264) (Richardson 2010). The resulting motion vectors can be extracted and decoded from the bitstream at the receiving end to reduce or remove the need for ME when performing frame rate up-conversion (Chen et al. 1998, Sasai et al. 2004). In order to detect and account for abrupt illumination changes, which can cause faulty motion vectors during BMA, histogram-based methods have been developed (Thaipanich et al. 2009). By using an increased temporal window in BMA the accuracy of ME can be increased (Kang et al. 2008), though at the price of processing time.

A very popular post-processing step on the Motion Vector Field (MVF) obtained from ME is median filtering (Zhai et al. 2005, Choi et al. 2006, Gan et al. 2007, Luessi & Katsaggelos 2009). This step can also be performed on hardware (Tasdizen & Hamzaoglu 2010). The filtering "smooths" the MVF, removing outliers. The problem of overlapping motion-compensated blocks is often solved by employing Overlapped Block Motion Compensation (OBMC) techniques (Lee et al. 2003, Zhai et al. 2005, Choi et al. 2006). An alternative method of handling occlusion is for example using the divergence of the MVF (Hong 2009).

4. Motion estimation

There are several different ways to estimate motion between frames, and many models which may be suitable to describe this motion. The model most commonly used is that of linear motion with constant acceleration. This is a reasonable assumption provided the frame rate is not too low, since the frame-to-frame coherence is usually high.

Section 4.1 below presents a way to perform motion estimation (ME) by using a block matching algorithm (BMA). Image blocks in temporally adjacent frames are matched, as illustrated in Figure 4.1, using a block similarity function (see Section 4.2). Section 4.3 presents another way to estimate motion by using pre-calculated motion vectors from encoded bitstreams.

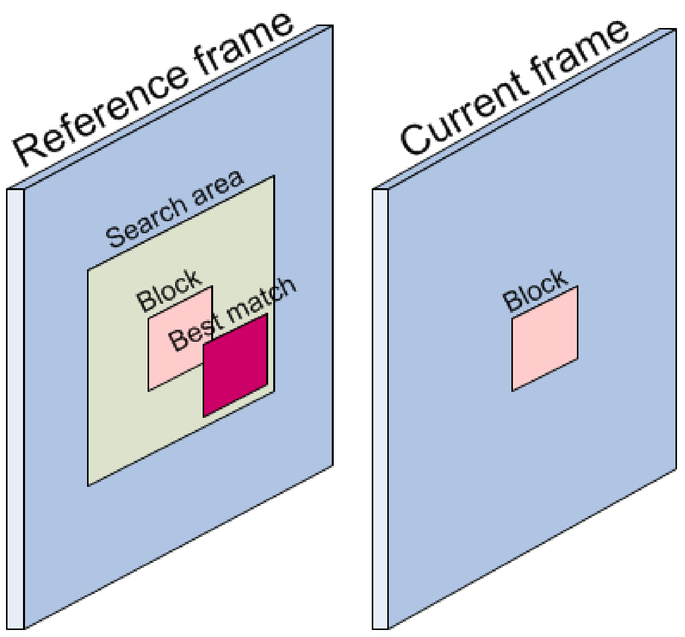


Figure 4.1: Block matching between a block in the current frame against blocks in the reference frame. The best match is found in the lower-right corner. The search area is used to limit the amount of blocks matched.

4.1 Block matching algorithms

In block matching algorithms (BMAs) a frame is divided into a set, B, of non-overlapping rectangular image blocks. An individual block $b \in B$ is matched against blocks in a temporally adjacent reference frame. The best match found for a block b using some appropriate similarity function is used to form a motion vector for the block. The motion vectors are used in motion compensation (MC) described in Chapter 5.

The most common block sizes used are 4x4, 8x8 and 16x16 pixels. This is because they are common divisors to the most well-known resolutions, such as Common Intermediate Format (CIF) and 720p. 1080p is however not divisible by 16, and this must be taken into consideration during implementation.

Matching blocks can be computationally expensive, and different BMAs try to reduce the number of block matching operations required to find the best match. All BMAs presented below, except for Adaptive Rood Pattern Search (ARPS) (Nie & Ma 2002), are fully parallelizable as all blocks within a frame are independent from each other.

α	Length of rood arm
b	An image block
b_H	Height of block in pixels
b_W	Width of block in pixels
H	Search area height
MV_x	Horizontal component of a block motion vector
MV_y	Vertical component of a block motion vector
n	Total number of blocks matched for one block
s_i	Number of block mathing operations in step i of an algorithm
t	Time index
W	Search area width
$\mathbb{Z}^+$	Positive integers (1, 2, 3, ...,)

Table 4.1: List of notations used in this chapter.

4.1.1 Full search

Full Search (FS) involves finding a best match for a reference block against all possible block locations within a specified search window. The number of blocks matched, n, for blocks of width b_W and height b_H, and search window width W and height H (where $W \geq b_W$ and $H \geq b_H$), can be calculated using Equation 4.1 below.

$$n = (W - b_W + 1) * (H - b_H + 1) \tag{4.1}$$

For a typical setting of $b_W, b_H = 16$ and $W, H = 30$ a total of 225 blocks are matched. Matching a block against an entire frame is possible, but is not practically feasible, especially for high frame resolutions. For example, matching a block against an entire frame using $b_W, b_H = 16$ and $W = 1920$ and $H = 1080$, a total of 2028825 block matching operations would be required.

4.1.2 Three Step Search

One of the very first BMAs was the Three Step Search (TSS) (Koga et al. 1981). This algorithm has influenced many other algorithms, such as New Three Step Search (NTSS) and Four Step Search (4SS). TSS reduces the number of search operations, compared to FS, by iteratively locating a best match in three steps.

- **Step one**
  Match all positions, including the center position, at a distance of four pixels away from the center. This step matches nine blocks, i.e. $s_1 = 9$.
- **Step two**
  Center the search position around the best match found. Match all positions at a distance of two pixels away from the center. This step matches eight blocks, i.e. $s_2 = 8$.
- **Step three**
  Center the search position around the best match found. Match all positions at a distance of one pixel away from the center. This step matches eight blocks, i.e. $s_2 = 8$. The best match found in this step is the match returned by the algorithm.

This algorithm always perform 25 block matching operations (Equation 4.2). Figure 4.2 below illustrates an example run of TSS.

$$n = s_1 + s_2 + s_3 = 25 \tag{4.2}$$

TSS reduces the number of blocks matched compared to full search. If $b_W, b_H = 16$ and $W, H = 30$ is used for full search, TSS performs nine times better.

Barjatya (2004) argues that one disadvantage of TSS is that it can miss small motion due to the search pattern.

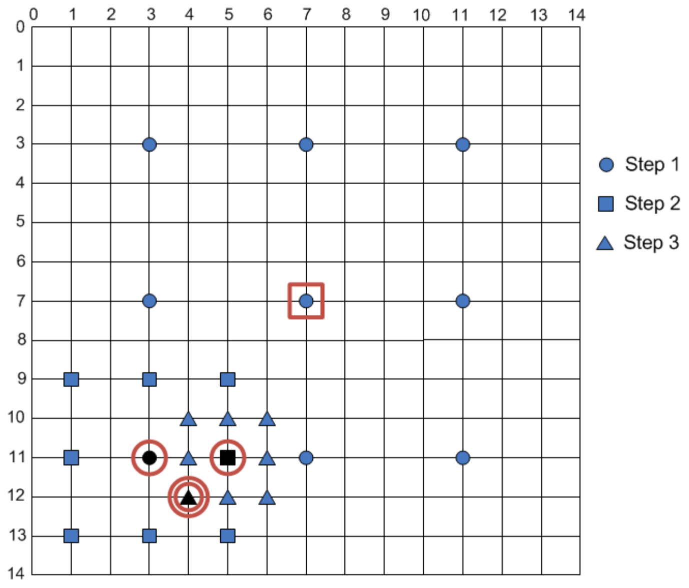


Figure 4.2: Illustration of the TSS algorithm. The starting center position is indicated by a surrounding square. The circles indicate the best matches found in each step. The double circle indicates the best match found by the algorithm. TSS always perform 25 block matching operations.

4.1.3 New Three Step Search

Li et al. (1994) suggested an improved version of TSS, called New Three Step Search (NTSS). Where TSS can have problems of finding small motion, NTSS tackles this using a center-biased search pattern. In contrast to TSS it has two different early termination conditions, where searching can be stopped early to reduce the total number of blocks matched.

- **Step one**
  Match all positions, including the center position, at a distance of one and four pixels away from the center. This step matches 17 blocks, i.e. $s_1 = 17$. If the best match is found in the center, return this as the best match found.

- **Steps two** and **three**
  If the best match in s_1 is found at a distance of:

  - **One** pixel from the center. Then center the search position around the best match found. Match all positions not matched in the first step, at a distance of one pixel away from the center. This step matches either three or five blocks. The best match found at this point is the best matching block.
  - **Four** pixels from the center. Then follow steps two and three from TSS. The best match found in step three is the match returned by NTSS.

  Step two matches three, five or eight blocks, i.e. $s_2 \in \{3, 5, 8\}$. Step three can be skipped, or matches eight blocks, i.e. $s_3 \in \{0, 8\}$.

The number of blocks matched, n, are the sum of all blocks matched in the steps above. This can vary from 17 to 33 matches (Equation 4.3). Figure 4.3 below illustrates an example run of NTSS.

$$n = s_1 + s_2 + s_3, \text{ where } n \in \{17, 20, 22, 30, 32, 33\} \tag{4.3}$$

Li et al. (1994), Barjatya (2004) have presented results showing that NTSS performs slightly better than TSS.

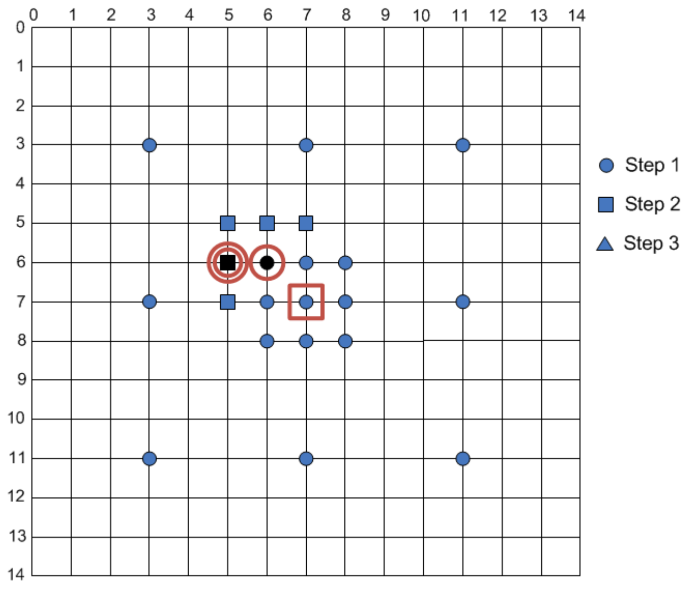


Figure 4.3: Illustration of the NTSS algorithm. The third step was skipped as the best match found in the first step was 1 pixel away from the center. The starting center position is indicated by a surrounding square. The circles indicate the best matches found in each step. The double circle indicates the best match found by the algorithm. A total of 22 blocks were matched in this example.

4.1.4 Four Step Search

The Four Step Search (4SS) algorithm (Po & Ma 1996) is based on TSS and NTSS. Like TSS and NTSS it uses a center biased search pattern and has early termination conditions in the two first steps.

- **Step one**
  Match all positions, including the center position, at a distance of two pixels away from the center. This step matches 9 blocks, i.e. $s_1 = 9$. If the best match is found in the center, skip to step four.
- **Step two**
  Center the search position around the best match found. Match all positions not matched in the first step at a distance of two pixels away from the center. This step is either skipped, or matches three or five blocks, i.e. $s_2 \in \{0, 3, 5\}$. If the best match is found in the center, skip to step four.
- **Step three**

This step is exactly the same as step two. It is either skipped, or matches three or five blocks, i.e. $s_3 \in \{0, 3, 5\}$.

- **Step four**
  Center the search position around the best match found. Match all positions at a distance of one pixel away from the center. This step matches eight blocks, i.e. $s_4 = 8$. The best match found in this step is the match returned by the algorithm.

The number of blocks matched, n, are the sum of all blocks matched in the four steps. This can vary from 17 to 27 matches (Equation 4.4). Figure 4.4 below illustrates an example run of 4SS where 25 blocks are matched.

$$n = s_1 + s_2 + s_3 + s_4, \text{ where } n \in \{17, 20, 22, 23, 25, 27\} \tag{4.4}$$

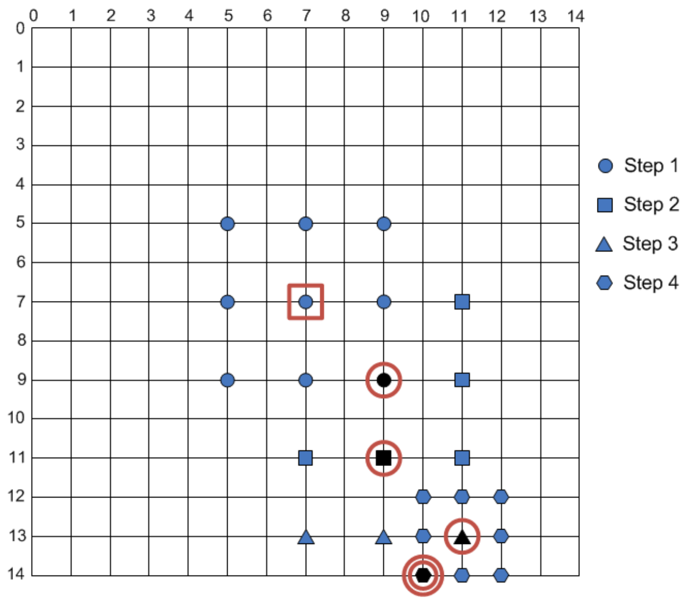


Figure 4.4: Illustration of the 4SS algorithm. The starting center position is indicated by a surrounding square. The circles indicate the best found matches in each step. The double circle indicates the best match found by the algorithm. A total of 25 blocks were matched in this example.

Po & Ma (1996), Barjatya (2004) show that 4SS reduces the number of blocks matched, compared to TSS, by 20-30%, depending on the amount of motion in the video. 4SS can in the worst case be 8% slower than TSS.

4.1.5 Diamond Search

Diamond Search (DS) (Zhu & Ma 1997) is influenced by 4SS but uses a diamond shape search pattern instead. It has three steps, where the second step can be repeated any number of times.

- **Step one**
  Start by matching the center position, the positions two pixels away in the horizontal and vertical directions, and the positions one pixel away in the diagonal directions. This step matches nine blocks, i.e. $s_1 = 9$. If the best match is found in the center, skip to step three.
- **Step two**
  Center the search position around the best match found. Match all positions not matched in the first step in a diamond shaped pattern. This step is either skipped, or matches three or five blocks, i.e. $s_2 \in \{0, 3, 5\}$. Repeat this step until the best match is found in the center.
- **Step three**
  Center the search position around the best match found. Match all positions one pixel away in the horizontal and vertical directions. This step matches four blocks, i.e. $s_3 = 4$. The best match found in this step is the match returned by the algorithm.

The number of blocks matched, n, are the sum of all blocks matched in the three steps (Equation 4.5). Figure 4.5 below illustrates an example run of DS where 21 blocks are matched.

$$\begin{aligned} n = s_1 + k * s_2 + s_3, \text{ where } n \in \{13, 16, 18, 19, 21, 22, 23 \ldots\}, \\ k \in \{0, 1, 2, \ldots\} \end{aligned} \tag{4.5}$$

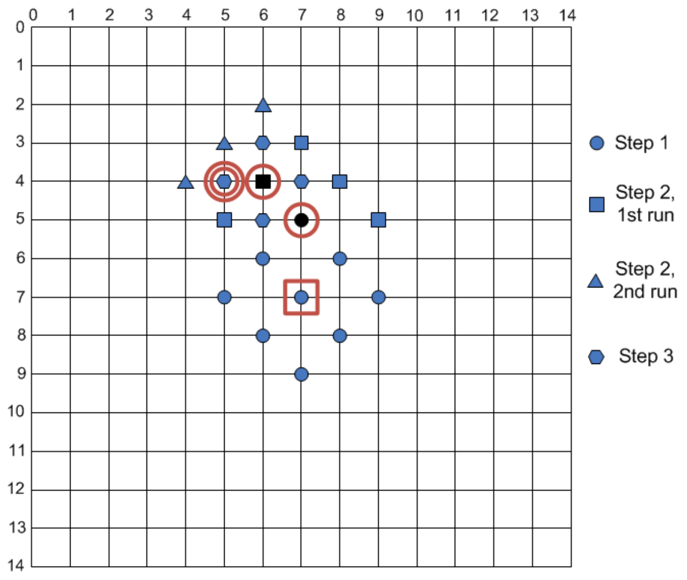


Figure 4.5: Illustration of the DS algorithm. The starting center position is indicated by a surrounding square. The circles indicate the best matches found in each step. Notice that the same position was found as the best match for both the first and second run of step two. The double circle indicates the best match found by the algorithm. A total of 21 blocks were matched in this example.

Barjatya (2004) shows that DS slightly reduces the number of blocks matched compared to 4SS.

4.1.6 Adaptive Rood Pattern Search

The Adaptive Rood Pattern Search (ARPS) (Nie & Ma 2002) is named after its rood-like pattern. Every block uses the motion vector of their left neighbouring block to set the length of the rood arm α. ARPS is thus only parallelizable in the vertical direction as the blocks are horizontally dependent. This dependency gives the algorithm the potential of accurately finding large motion between frames, since it assumes that motion in general is coherent in a frame.

ARPS uses zero-motion prejudgement for early termination of video with static content, which can significantly reduce the total number of blocks matched for a frame.

Three steps are used to match blocks, where the last step is repeated until the center position is the best match.

- **Step one**
  Perform zero-motion prejudgement; match the block against the block at the same position in the reference frame. If the difference is below a threshold τ, stop searching and return this match. This step matches one block, i.e. $s_1 = 1$.

- **Step two**
  If the block is the left-most block, set $\alpha = 2$. Otherwise use the largest component, MV_x or MV_y, of the motion vector of the left neighbouring block, i.e. $\alpha = max(|MV_x|, |MV_y|)$. Letting (x, y) denote the current center of search, all positions $(x \pm \alpha, y \pm \alpha)$ and $(x + MV_x, y + MV_y)$ are matched. This step is either skipped, or matches five blocks, i.e. $s_2 \in \{0, 5\}$.

- **Step three**
  Center the search position around the best match found. Match all positions, not matched in the first or second step, at a distance of one pixel away from the center in the horizontal and vertical directions. Iterate this step until the best match is the center position. This position is returned as the best match by the algorithm. This step is either skipped, or matches two to four blocks, i.e. $s_3 \in \{0, 2, 3, 4\}$.

The number of blocks matched, n, are the sum of all blocks matched in the three steps (Equation 4.6). Notice that step three can be repeated k number of times. Figure 4.6 below illustrates an example run of ARPS where step three is iterated three times. A total of 15 blocks are matched in this example.

$$\begin{aligned} n = s_1 + s_2 + k * s_3, \text{ where } & n \in \{1, 8, 9, 10, \ldots\}, \\ & k \in \{0, 1, 2, \ldots\} \end{aligned} \tag{4.6}$$

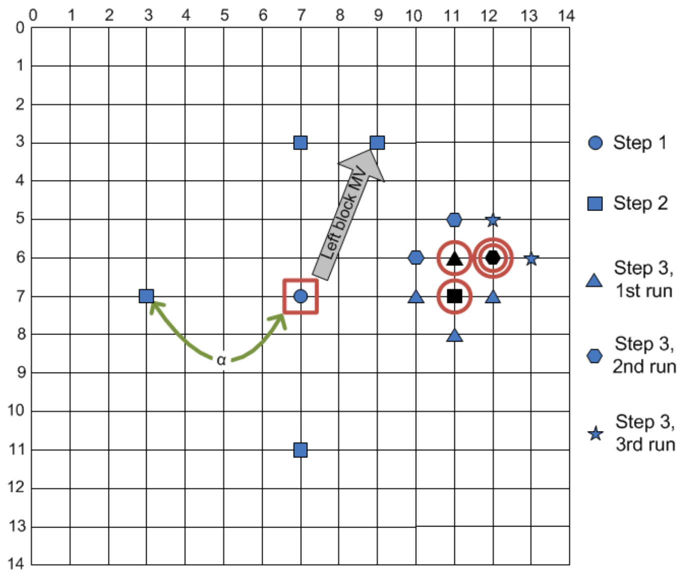


Figure 4.6: Illustration of the ARPS algorithm. The starting center position is indicated by a surrounding square. The rood arm α is illustrated by the thin line. The circles indicate the best found matches in each step. The double circle indicates the best match found by the algorithm. A total of 15 blocks were matched in this example.

Nie & Ma (2002), Barjatya (2004) show that ARPS reduces the number of blocks matched, compared to DS, by a factor of two.

4.1.7 Full Bidirectional Search

Bidirectional search, where bidirectional and bilateral are sometimes used interchangeably, is a family of BMAs which in theory should be more suited for use together with bidirectional motion compensation (BDMC). The searching procedure is changed compared to e.g. 4SS and ARPS by not keeping a position in one frame constant while searching for a match in another frame. Instead, positions in both frames are allowed to vary. The idea is not to estimate where one block of the frame might have moved to or from in the reference frame, but which surrounding parts may have moved *through* the current block. This corresponds well to the behaviour of BDMC, since the motion vectors for the block, as used by BDMC, can be said to originate from an unfilled block in an interpolated intermediate frame. Variations on the bidirectional search is used by e.g. Zhai et al. (2005) and Kang et al. (2008).

We present in this thesis Full Bidirectional Search (FBDS), our simple version of an exhaustive bidirectional search algorithm. It uses the zero motion prejudgement found in ARPS for early termination, which for the low to moderate motion found in the remote surveillance use scenario should prove highly beneficial. The search area ranges from -4 to +4 in horizontal and vertical directions, for a total of 81 positions searched (82 including the early zero motion check). The algorithm rests heavily on the assumption of linear motion. The offset in search position is negated for one of the frames; e.g. a position with offset of minus one in both horizontal and vertical directions compared to the block's position is matched against a position with offset plus one in horizontal and vertical directions in the other frame.

The number of positions searched may seem large compared to e.g. 4SS and ARPS. However, zero motion prejudgement, aggressive optimization from the compiler and improved scheduling when using OpenMP can reduce this issue.

Figure 4.7 shows how the two steps of FBDS work. In step one, the center position in both frames is matched and checked against the zero motion threshold. Step two (s_2) involves 81 matches, starting at the top left (-4, -4) block in frame A matched to the bottom right (+4, +4) block in frame B. The last matching operation in this step is between the bottom right (+4, +4) block in frame A and the top left (-4, -4) block in frame B.

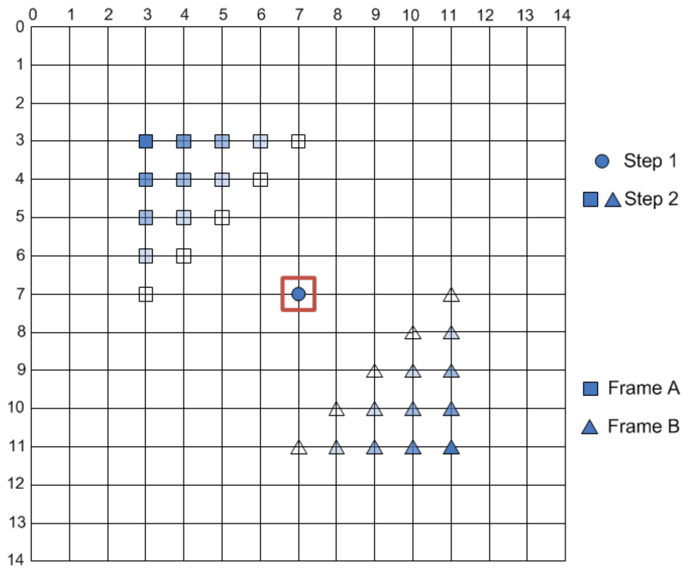


Figure 4.7: Illustration of the FBDS algorithm. The starting center position is indicated by a surrounding square. Step 2 involves 81 additional matches, starting at the top left block for frame A and the bottom right block in frame B. The last match is between the bottom right block in frame A and the top left block in frame B. A total of 82 blocks were matched in this example.

4.2 Block similarity functions

There are a number of different techniques to measure the similarity between two blocks (denoted by b_A and b_B in the following sections). All pixels in the blocks are compared in pairs and a total value indicating the similarity is computed. A low value indicates higher similiarity. As the pixels are independent from each other in each frame, the techniques are parallelizable and can be implemented in an efficient manner.

4.2.1 Sum of Absolute Differences

The Sum of Absolute Differences (SAD) summarizes the absolute difference pixel values (Bovik 2009, Marzat & Ducrot 2009), as described in Equation 4.7. SAD is sometimes referred to as Sum of Absolute Errors (SAE).

$$SAD(b_A, b_B) = \sum_{i=0}^{b_W-1} \sum_{j=0}^{b_H-1} |b_{A_{ij}} - b_{B_{ij}}| \tag{4.7}$$

This technique is widely adopted (Cetin & Hamzaoglu 2010, Lee & Nguyen 2009, Luessi & Katsaggelos 2009), as it is computationally cheaper than many other block similarity functions. Streaming SIMD Extensions (SSE) exploits the inherent parallelism by having special instructions for computing the SAD for up to 16 pixels at a time (see Section 8.2.1).

4.2.2 Mean Absolute Difference

Mean Absolute Difference (MAD) computes the mean of SAD (Barjatya 2004), as described in Equation 4.8. MAD is sometimes referred to as Mean Absolute Error (MAE).

$$MAD(b_A, b_B) = \frac{1}{b_W * b_H} SAD(b_A, b_B) \tag{4.8}$$

MAD is computationally more expensive than SAD and an implementation requires floating point precision.

4.2.3 Sum of Squared Differences

Sum of Squared Differences (SSD) summarizes the squared difference pixel values (Marzat & Ducrot 2009), as described in Equation 4.9.

$$SSD(b_A, b_B) = \sum_{i=0}^{b_W-1} \sum_{j=0}^{b_H-1} (b_{A_{ij}} - b_{B_{ij}})^2 \tag{4.9}$$

4.2.4 Mean Squared Error

Mean Squared Error (MSE) computes the mean of SSD (Bovik 2009), as described in Equation 4.10.

$$MSE(b_A, b_B) = \frac{1}{b_W * b_H} SSD(b_A, b_B) \tag{4.10}$$

MSE is computationally more expensive than SSD and an implementation requires floating point precision. This function is used to compute the peak signal-to-noise ratio (PSNR) between two frames, which is described in Section 7.1.1.

4.3 Bitstream motion vectors

Some form of block matching is performed in encoders for all video coding standards. The resulting motion vectors can be extracted and decoded from the bitstream at the receiving end to reduce or remove the need for ME when performing frame rate up-conversion. This technique has successfully been used by e.g. Chen et al. (1998) and Sasai et al. (2004). Depending on the settings and runtime decisions on the encoder side the motion vectors in the bitstream may or may not be describing true motion in the video. They can in some cases be missing entirely (Huang & Nguyen 2008). If bitstream motion vectors are to be used some sort of judging of the quality of the received motion vectors must be incorporated into the algorithm, as done by e.g. Luessi & Katsaggelos (2009).

5. Motion compensation

5.1 Direct motion compensation

The most simple form of motion compensation (MC) is direct motion compensation (DMC), also referred to as unidirectional motion compensation. In a new interpolated frame, blocks are filled with data by moving blocks in the reference frame along their motion vectors (obtained in motion estimation (ME)). This can be done in either the forward or backward direction. Figure 5.1 below illustrates DMC in the forward direction.

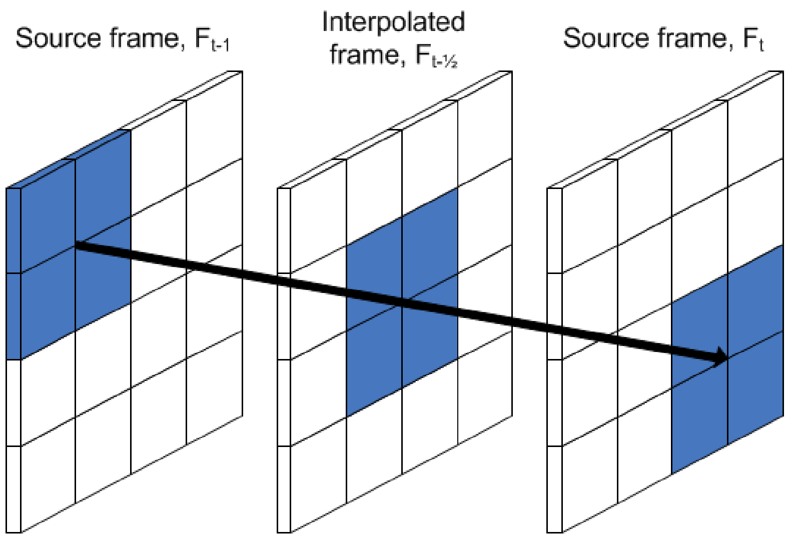


Figure 5.1: Illustration of DMC in the forward direction. Every block in F_{t-1} is moved along its motion vector.

The problem with DMC is that all pixels might not be filled (Huang & Nguyen 2008, Luessi & Katsaggelos 2009). These "holes" are illustrated in Figure 5.2 below. Various methods exist to fill these holes, such as video inpainting (Criminisi et al. 2004, Wexler et al. 2004) and texture synthesis (Ashikhmin 2001).

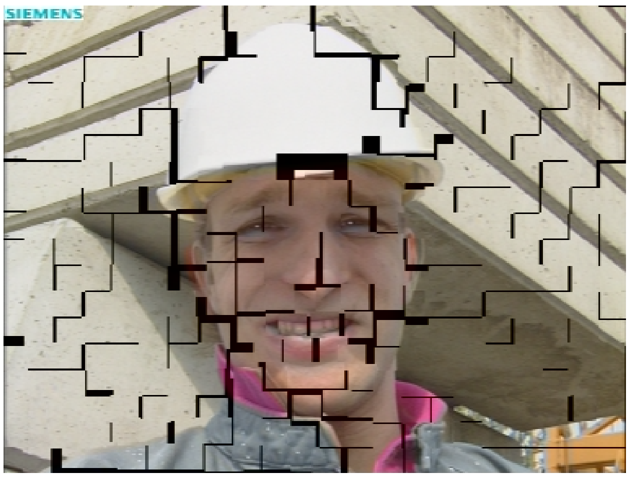


Figure 5.2: Unfilled pixels when using direct motion compensation. This frame is from the *Foreman* standard benchmarking video.

5.2 Bidirectional motion compensation

Another way of performing motion compensation is bidirectional motion compensation (BDMC). For every block in a new frame, the motion vector found for the same block in the reference frame is used to interpolate in both the backward and forward directions. This method leaves no unfilled pixels, like DMC, as the method iterates over all blocks in the new frame instead of blocks from the reference frame. Figure 5.3 below illustrates BDMC.

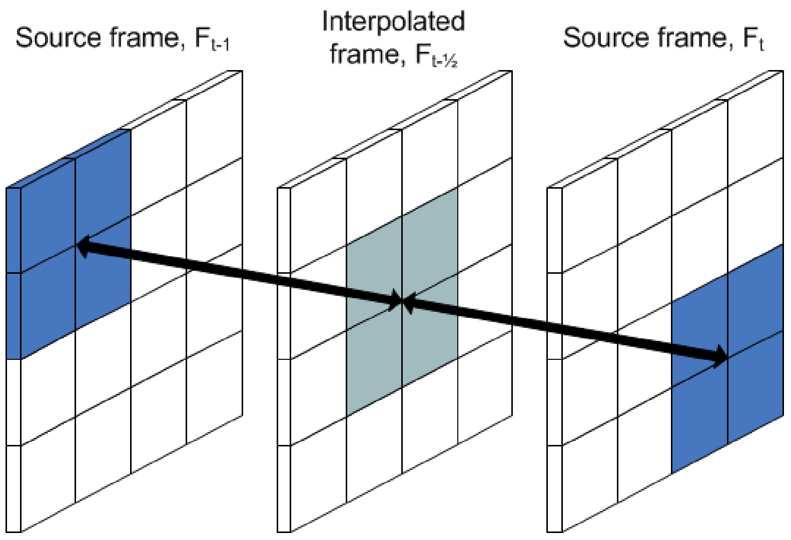


Figure 5.3: BDMC illustrated. For every block in the interpolated frame $F_{t-\frac{1}{2}}$, the motion vector found for the same block in the reference frame F_{t-1} is used to interpolate in both the backward and forward directions.

Let $P_t(x, y)$ be the value of a pixel with coordinates (x, y) in frame f_t, t being a time index. Then for all (x, y) inside the boundaries of f_t, with $1 \leq j < k$ and $j \in \mathbb{Z}^+$, the pixel values for an intermediate interpolated frame $f_{t-\frac{j}{k}}$ is given by Equation 5.1 below.

$$\begin{aligned} P_{t-\frac{j}{k}}(x, y) = \frac{1}{2}P_{t-1}\left(x - (1 - \frac{j}{k}) * MV_x, y - (1 - \frac{j}{k}) * MV_y)\right) \\ + \frac{1}{2}P_t\left(x + (\frac{j}{k}) * MV_x, y + (\frac{j}{k}) * MV_y)\right) \end{aligned} \tag{5.1}$$

6. OpenCL

The growing interest in parallel processing solutions has lead to the emergence of multi-core CPUs as well as GPGPU, general purpose computing on graphics processing units. The GPUs of today contain hundreds up to a thousand stream processing cores, a good match for the highly parallelizable algorithms in this thesis. Efforts to utilize this processing power include frameworks and APIs such as CUDA from Nvidia, OpenCL and Microsoft's DirectCompute.

Open Computing Language (OpenCL) was chosen for the work of this thesis on the merits of having a wider industry support. CUDA is only supported by Nvidia, and DirectCompute is part of Microsoft's DirectX frameworks. We made use of OpenCL for GPU-acceleration of the motion compensation part of the code, taking advantage of on-device specialized hardware for bilinear interpolation. For a sufficiently high-end GPU it also proved to be faster than a CPU-based implementation of motion compensation. We used the first version of the standard, OpenCL 1.0. A later version, OpenCL 1.1, is available but does not contain any substantial changes relevant for our needs, and at the time of writing required beta-version drivers for some platforms.

OpenCL is an open industry standard. It was developed for allowing general purpose code to run on a wide range of processor types, such as CPUs, GPUs or Digital Signal Processing (DSP) units. The specification is defined by the Khronos OpenCL Working Group, whose mother organisation The Khronos Group is also responsible for the development of OpenGL (hence the naming of OpenCL). The standard is supported by companies such as Nvidia, AMD, IBM, Apple and others. OpenCL consists of a language, OpenCL C (based on the C99 standard), and a set of APIs for controlling the execution of the code written in this language.

In the following sections we present a brief overview of the terminology and processing flow when working with OpenCL. For a more detailed overview, please refer to the OpenCL specification (Khronos OpenCL Working Group 2008).

6.1 Processing flow

Setting up OpenCL to run code on a GPU typically requires the following steps:

1. Acquire a *platform ID* and *device ID*.
2. Create a compute *context*.
3. Create a *command queue*.
4. Create a compute *program*.
5. Build the compute program.
6. Create a compute *kernel* in the compute program.
7. Allocate *buffers* on the device (if buffers are required by the kernel).

Once these steps have been completed some additional steps are required to actually run the code and obtain the results:

8. Set the *global* and *local* work sizes. In many cases this only needs to be done once.
9. Transfer data from the *host* into device buffers (if necessary).
10. Set the kernel *arguments*. Some arguments may only have to be set once.
11. Execute the kernel on the device.
12. Transfer result from device back to host.

6.2 Terminology

A *platform ID* contains the *host* and a collection of *devices*. The host is the part of the system from where the calls to the OpenCL API are made. There can be many OpenCL implementations available on the system. One example of this would be if implementations from both AMD and Nvidia were installed. A choice would then have to be made which of these to use.

Devices consists of one or more *compute units*, which in turn consists of *processing elements*. A device can be a CPU, a Graphics Processing Unit (GPU) or other type of device (referred to as "dedicated OpenCL accelerators" in the specification). For a multi-core CPU the corresponding OpenCL representation would be a device with a single compute unit and one processing element per core. If a system contains more than one CPU these would be available as separate compute units.

A *context* is the environment within which the code executes. This environment includes a set of devices together with information about the memory properties of these devices. It also contains *command queues* associated to the devices. The command queue holds *commands* enqueued for a specific device.

Kernels, functions written in the OpenCL C language, are grouped together in *programs* which are either built for specific devices during runtime or loaded from pre-built binaries. To execute a kernel in a successfully built program, the kernel *arguments* must first be set. These arguments may e.g. be primitive types such as integer values, *handles* to *buffers* allocated on the device, or texture sampler objects.

The size and dimensions of the data to be processed must be specified by setting the *global* and *local* work sizes. Data representations may be one-, two- or three-dimensional. The global work size determines the total number of *work items* needed to process all the data, and specifying a local work size gives the option to partition the data.

Several instances of the kernel is executed in parallel on the device, each instance being a work-item. The work-items are grouped into *work groups*, whose sizes depends on the specified local work sizes. Partitioning the data into smaller chunks e.g. enables sharing of data common to all work-items in a work group. Efficient sharing may optimize memory operations, leading to much higher performance.

Each work group execute on a single compute unit, and each work item can be distinguished from the others by its so-called *global* and *local ID*s. The global ID is the coordinate in the data being processed, coordinates which may be one-, two- or three-dimensional depending on the type of data. If e.g. a two-dimensional image is being processed, the global ID of a work item might map to a specific pixel coordinate in the image. The local ID determines the coordinate of the work item within its work group.

If on-device buffers are used by a kernel these must be filled with data by enqueuing data transfer *commands* to a *command queue* associated with the device. Commands may be issued to be *blocking*, where the next statement is not executed before the transfer is complete, or *non-blocking*, resuming execution while allowing the transfer to happen "in the background". When the arguments and work sizes have been set, and buffers filled with data, the kernel can be enqueued for execution by adding an execution command to a command queue. This execution command can also be said to be non-blocking.

Output from kernels often come in the form of a device buffer filled with modified data. This data can be copied back from the device, also either blocking or non-blocking, by issuing a buffer-reading command to the command queue. The command can be added to the queue right after the kernel has been queued for execution, since the transfer will not initiate before the execution is complete (unless the command queue has been specified to allow out-of-order execution). Once data has been transferred back from the device it may be written to file or used in other ways.

7. Evaluation methods

There are two different performance aspects to consider when evaluating a frame rate up-conversion tool (FRUCT): algorithm runtimes and image quality.

The runtime depends on how fast the motion estimation (ME) and motion compensation (MC) is performed. This aspect is particularly interesting in a time-constrained setting, e.g. real-time environments.

The image quality can be evaluated with both objective and subjective methods; objective methods are based on mathematical formulas, whereas subjective methods are based on how humans perceive the quality. A number of objective and subjective methods are presented below along with the choice of methods for this thesis.

7.1 Objective quality evaluation

Objective quality evaluation can be categorised into full-reference (source frame exists), reduced-reference (some part of a source frame exists) and no-reference (no source frame exists) quality estimation.

To estimate the quality of a new frame without, or with a reduced, source frame is a difficult and challenging task. Nevertheless, there have been successfull results; Woodard & Carley-Spencer (2006) uses no-reference objective measurement to automate screening for structural magnetic resonance images, and Wang et al. (2002) uses no-reference objective measurement to assess quality of JPEG images.

In full-reference evaluation, new frames are compared with source frames to determine the quality of the new frames. Frames are dropped at a factor k from a source video to produce a down-converted video. E.g. a source video at 60 frames per second (FPS) can be down-converted with $k = 3$ to a new video at 20 FPS. The frame rate up-conversion tool then adds new frames, with the same factor k, to produce a new video at 60 FPS.

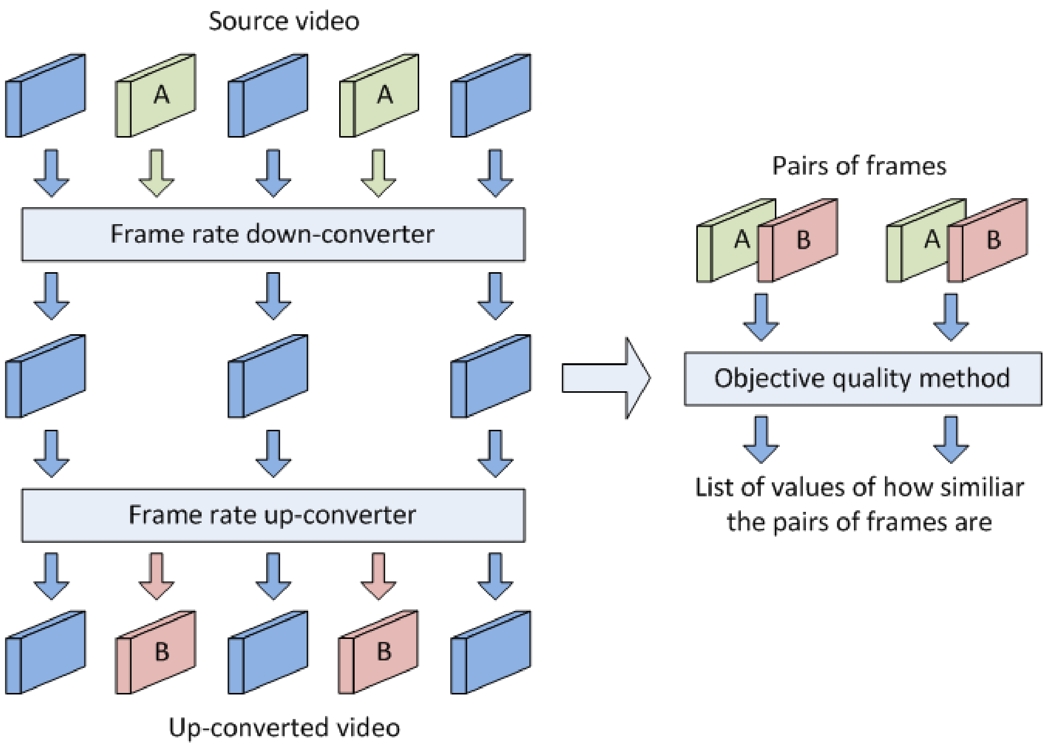


Figure 7.1: Full-reference objective quality evaluation illustrated. A source video is first down-converted and then up-converted. The source frames (marked as A) and the up-converted frames (marked as B) are compared pair-wise using an objective quality evaluation method. The output indicates how similar the pairs of frames are.

7.1.1 Peak signal-to-noise ratio

Richardson (2010) describes peak signal-to-noise ratio (PSNR) as a fast and easy full-reference metric to compare the luma between two frames. PSNR gives a logarithmic value in decibel, as shown in Equation 7.1. MSE is the Mean Squared Error (MSE) (described in section 4.2.4) and m is the number of bits needed to sample one pixel. If two frames are identical the MSE will be zero and the $PSNR_{dB}$ undefined.

$$PSNR_{dB} = 10 \log_{10} \frac{(2^m - 1)^2}{MSE} \tag{7.1}$$

PSNR does not always correspond well with subjective quality evaluation (Richardson 2010, Wang & Bovik 2009). PSNR is, despite this, the de-facto standard for objective quality evaluation (Oelbaum et al. 2007), mainly because it is very easy to implement and fast compared to more complex methods.

7.1.2 Structural similarity

Structural Similarity (SSIM) (Wang & Bovik 2002, Wang et al. 2004) is another full-reference objective quality evaluation metric. Like PSNR, SSIM compares the luma data a pair of frames, but it also takes differences in contrast and structure into consideration.

SSIM gives as result a normalized value in the range -1.0 to 1.0 indicating how similar the frames are. A value of 1.0 is returned only if the frames are identical. Equation 7.2 describes SSIM where x and y are the two frames, μ_x and μ_y are the estimated mean luma of x and y, σ_x and σ_y are the estimated contrasts of x and y, and σ_{xy} the covariance of σ_x and σ_y. C_1 and C_2 are constants used to avoid instability in the function.

$$SSIM(x,y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)} \tag{7.2}$$

The comparison performed by SSIM corresponds better with subjective quality evaluation, but is more complex and harder to implement than PSNR.

7.1.3 Other methods

Much research effort has been put into developing objective quality evaluation methods that corresponds well with subjective quality evaluation. Here are some of the other methods presented in recent years:

- PSNR+ an extension of PSNR (Oelbaum et al. 2007).
- Predicted Mean Opinion Score (MOSp) (Bhat et al. 2009).
- Suthaharan et al. (2005) proposed a metric based on Just Noticeable Difference (JND).

These methods have high reported correlation with subjective quality evaluation methods, up to 70-90% (Richardson 2010).

7.2 Subjective quality evaluation

Subjective quality evaluation is based on how humans perceive the quality of images and video. Richardson (2010) argues that there are several factors that may influence how the quality is perceived, e.g. the tester's state of mind and the test environment. Also, humans tend to be sensitive to spatial and temporal fidelity, i.e. how clearly different parts of the video can be seen and how smooth it is.

In order to ensure that the results obtained from a subjective quality evaluation are reliable, a large pool of testers are needed. This is because testers, as they grade an increasing number of videos, often learn to look for artefacts, which may effect the results (Richardson 2010). The results, obtained from all testers, are normalized (Bhat et al. 2009), to a Mean Opinion Score (MOS), that "indicates the relative quality of the impaired and reference sequences" (Richardson 2010, p.20).

Three different subjective quality evaluation methods are presented below.

7.2.1 Double-stimulus impairment scale

The ITU Radiocommunication Sector (ITU-R) describes different methods for assessing the quality of images and videos in BT.500-11 (ITU Radiocommunication Sector 2002).

One of these methods is Double-Stimulus Impairment Scale (DSIS), where a tester is presented with pairs of videos, one after another, knowing that the first video is the reference video and the second one is impaired. The tester then grades the second video, having the reference video in mind, on a five-step scale ranging from *very annoying* to *imperceptible.*

7.2.2 Double-stimulus continuous quality-scale

Double-Stimulus Continuous Quality-Scale (DSCQS) is another method described in BT.500-11. A tester is presented with pairs of videos, where the videos are shown simultaneously. The tester grades both videos, without knowing which is the reference video or the impaired video, on a five step scale ranging from *bad* to *excellent.* The order should be randomised for every pair of videos. This method is widely used according to Richardson (2010).

7.2.3 Subjective assessment method for video quality

Kozamernik et al. (2005) have suggested a method for subjective quality assessment called Subjective Assessment Method for Video Quality (SAMVIQ). The tester is presented with a reference video and a set of impaired videos. The tester can watch the videos in any order, and grades the impaired videos on a scale from 0 to 100. SAMVIQ gives results that are comparable with DSIS and DSCQS (Huynh-Thu et al. 2007, Tominaga et al. 2010), and the authors argue that "SAMVIQ is simpler, faster and more user-friendly than traditional subjective evaluation methods".

7.3 Choice of quality evaluation methods

PSNR and SSIM were chosen to evaluate the quality of the frame rate up-conversion tool. PSNR was chosen for implementation, even though it might not correspond well with subjective methods, because of its simplicity and its wide useage in the field. SSIM was chosen to complement the results from PSNR. The third-party software MSU Video Quality Measurement Tool[2] was used for this.

None of the subjective quality methods were chosen. Instead was the up-converted videos evaluated by ourselves in an ad-hoc manner.

[2]http://compression.ru/video/quality_measure/video_measurement_tool_en.html

8. Implementation

The frame rate up-conversion tool (FRUCT) was implemented in C++, where some parts critical for performance were hand-coded in assembly. Open Computing Language (OpenCL) was used for the motion compensation (MC) part. OpenMP compiler pragmas were used for all major parallelizable loops in the code to ensure all available CPU cores could be utilized. frame repetition (FR) and frame averaging (FA) were both implemented to serve as a basis for comparison against the more advanced algorithms. The general up-conversion procedure can be described with the pseudo-code in Figure 8.1.

```
FUNCTION UpConvert (
    input  : FILE or STREAM,
    output : FILE or STREAM,
    factor : INTEGER)
{
    prev, next, new := 0, 1, 2;
    frames := BUFFER[3];
    mvectors := BUFFER;

    frames[prev] <- ReadFrame(input);

    WHILE (input contains frames):
        frames[next] <- ReadFrame(input);
        mvectors <- EstimateMotion(
            frames[prev], frames[next]);
        FOR i := 1 TO factor:
            frames[new] <- CompensateMotion(
                frames[prev], frames[next],
                mvectors, i);
            WriteFrame(frames[new], output);
        SWAP(prev, next);
}
```

Figure 8.1: Up-conversion pseudo code

8.1 Naive methods

The FR implementation is trivial; simply reading one frame at a time from the source file and writing it to the output file k number of times, where k is the up-conversion factor.

FA was implemented in two versions: one CPU-based and one OpenCL-based. The OpenCL-based FA was an exploration of how Graphics Processing Unit (GPU) acceleration could be used for the motion estimation (ME)- and MC algorithms in the FRUCT.

8.2 Motion estimation

ME was performed on luma from the raw video data. No extraction of motion vectors from the bitstream was made, for the reasons outlined in Section 4.3. By not using bitstream motion vectors the FRUCT is also not tied to a specific video codec and can be used together with all contemporary and future codecs.

A drawback is however the loss of sub-pixel motion vector precision; in H.264/MPEG-4 AVC (H.264) motion vectors have up to quarter-pixel precision, which means motion vectors are not limited to the integer values produced by standard implementations of most ME algorithms described in this thesis. It would be possible to add sub-pixel accuracy to these algorithms, but at a cost of additional processing time which would probably put the goal of real-time capability at risk.

Of the block matching algorithms (BMAs) mentioned in section 4.1 Full Search (FS), Four Step Search (4SS), Adaptive Rood Pattern Search (ARPS) and Full Bidirectional Search (FBDS) were implemented. The choice of which algorithms to use was based on reported performance, both in terms of quality and runtime (Barjatya 2004). For 4SS the check for redundant search positions was not implemented. In order for the block-matching to be feasible for real-time operation two main parts of any BMA should be the primary focus of optimization efforts. First, the number of potential positions searched should be kept as low as possible without degrading quality too much (a necessary trade-off). Second, the operation of matching two blocks against each other should be as efficient as at all possible, since this part of the code will be run intensively. Details of the techniques implemented for increased performance and quality is given in the following sections.

8.2.1 SAD assembly optimization

By choosing an efficient algorithm the number of matching operations can be minimized, at a reasonable cost of small ME errors and lost accuracy. However, an optimized similarity function (see Section 4.2) can also greatly reduce time spent. The Sum of Absolute Differences (SAD) function, due to its wide use in video encoding, has the benefit of having specialized CPU instructions available.

In the work of this thesis the Streaming SIMD Extensions (SSE) instruction PSADBW (Intel Corporation 2011) was used, where PSAD stands for Packed Sum of Absolute Differences. This allowed calculation of absolute differences of 8 pairs of 8-bit values in a single instruction. In other words: if the block size is set to 8x8 an entire block row of 8 values can be processed for each PSADBW instruction. This means that the total number of calls to PSADBW to compute the SAD for a 8x8 block was 8 and for a 4x4 block only 2. Note that only 64-bit registers were used for PSADBW, not 128-bit, since we during implementation were unaware of the possibility of using these registers for this instruction. Using 128-bit instead of 64-bit registers could speed up execution of SAD calculation for a block by a factor of two.

8.2.2 Overlapped block motion estimation

Using a smaller block size means there is less data to use when matching blocks, e.g. 4x4 blocks contain 16 times less data than 16x16 blocks. This can lead to increased sensitivity to image noise and in effect faulty motion vectors. To reduce this issue when the block size is smaller than 16x16 we used enlarged blocks in the BMAs, a technique used by e.g. Zhai et al. (2005). Each 4x4 or 8x8 block is temporarily enlarged to 16x16 by including surrounding pixels in each direction, and then matched against a similarly enlarged block in the reference frame.

8.2.3 Zero motion prejudgement

In many types of video, especially the kind of mostly-static video this FRUCT implementation is intended for, there is a large frame-to-frame coherence. The high level of coherence can be found by checking the SAD of a macroblock at position (i, j) in both frames (i.e. no offset in search position) against a threshold value. If the SAD is below the threshold value the block is said to have no motion and no further ME is performed for this block. This technique is taken from ARPS (see Section 4.1.6). We have incorporated this technique into all ME algorithms implemented for this thesis, except for FS. The threshold value was set to 1024 for a 16x16 block.

8.3 Motion compensation

Direct motion compensation (DMC) was the first MC algorithm to be implemented for the FRUCT. Due to its simplicity it proved to be just as fast as predicted. However, since a potentially time-consuming post-processing method is required (see Section 5.1), this track of development was not explored further. Efforts were instead focused on bidirectional motion compensation (BDMC). As with FA, two versions of the BDMC algorithm were implemented: one CPU-based and one OpenCL-based.

The OpenCL-based solution keeps a copy of the previous and next frames in two 2D-image buffers (textures). By using a buffer-swapping procedure similar to the pseudo-code in Figure 8.1, only the newly read frame needs to be sent to the device, as the previous frame was copied in a previous call. The motion vectors for all macroblocks are copied to a buffer on the device, and the weighting factor is set as a kernel argument. The global work-item size is set to the frame width and height, and the local work-group size (see Chapter 6) is set to the specified constant macroblock width and height. In other words, every work item in a work-group belong to the same macroblock, and there is a one-to-one correspondance between pixels and work items.

During kernel execution each work item uses its global position (i.e. pixel position), together with an offset calculated with the weighting factor and block motion vector, for look-ups (sampling) in the image buffers to produce an output value.

The sampling can be done by nearest-neighbour, rounding non-integer sampling locations to the closest integer one (thus giving the same result as the CPU-based solution). Alternatively, sampling can be done by bilinear interpolation where the values of the four closest integer sample points are weighted together (Figure 8.2). The use of bilinear interpolation trades slightly decreased sharpness for increased smoothness in video motion. Since there is special hardware for this kind of interpolation on modern graphics cards there is no major time penalty compared to nearest-neighbour sampling. This would not be the case for a CPU-based implementation.

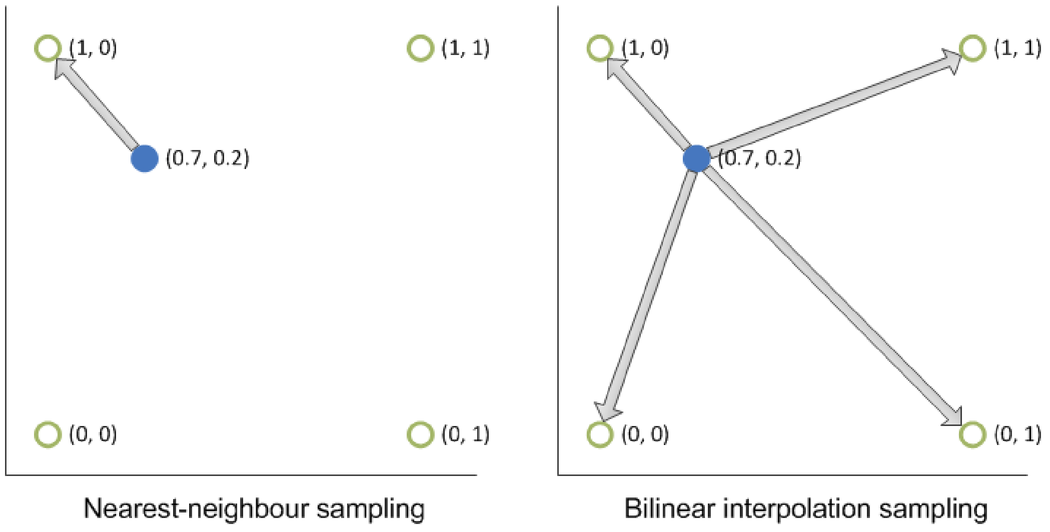


Figure 8.2: The non-integer sampling location (the filled circle) is rounded to the top left sample location. In bilinear interpolation (right side), the non-integer sampling location is interpolated from the four closest integer sample points.

After testing MC (with a separate kernel), averaging and copying of chroma (from previous frame), MC was found to give results which were indistinguishable from chroma averaging in most cases. The cost of increased MC time could not be justified. Copying chroma from previous frame was found to give bad results for videos with moderate motion. For these reasons only averaging of chroma was used, performed on the CPU for flexibility, since the other methods were still available as a configuration option.

9. Results

9.1 Test videos

The videos used for experimental verification were selected from standard test videos[3] used in existing literature. The selection was based on similarity to the actual use case of remote surveillance, meaning that videos with a static camera and low to moderate motion were favourized.

The *Akiyo* test video sequence (Common Intermediate Format (CIF) format) shows a female news anchor presenting the news. It features a static camera and background, with low motion as she moves her head while talking (see Figure 9.1a for a sample image from the video). The most challenging part of the video, from a frame rate up-conversion perspective, is to correctly interpolate movements of lips and eyes, especially blinking.

The *Container* test video sequence (CIF format) also includes a static camera and mostly low motion. It features two slowly moving ships, a flag waving in the wind and some water movement. Also, at the end of the sequence two birds fly by close to the camera at high velocity (Figure 9.1b). This video mainly tests the ability to handle low, predictable motion (the ships), some unpredictable motion (flag and water) and extremely high motion (the birds).

Foreman (CIF format) is one of the most used test sequences for frame rate up-conversion (Figure 9.1c). It features a close-up of a foreman at a construction site, where the camera is static in the first half of the video and quickly panning in the second half. It contains heavy motion, including motion blur, where the motion of the foreman's face is especially hard to estimate and compensate for. It is nowhere near the remote surveillance use scenario, but is included due to its popularity in the field.

[3]Standard videos can be found at http://media.xiph.org/video/derf/

(a) Akiyo (b) Container (c) Foreman

Figure 9.1: CIF test video sequences.

Due to the lack of standard test videos with 1080p resolution fitting the criteria of static camera and low motion we decided to record a few on our own[4]; *Liseberg*, *FerryClose* and *FerryFar*. These videos were recorded with a Nikon D5100 DSLR camera at 30 frames per second (FPS)[5] in H.264-encoded format. They were later converted to raw uncompressed Y'CbCr 4:2:0 format using MEncoder[6].

The *Liseberg* video (Figure 9.2) is shot outside an amusement park. It features fast vertical motion of an attraction, moderate horizontal motion of a tram, and low motion of pedestrians in the background. Also, a fast-moving bird introduces high-motion noise.

FerryClose and *FerryFar*, shown in Figure 9.3 and Figure 9.4, capture an approaching commuter ferry at different distances. Both videos have fairly high motion in the water, with the water covering at least a third or more of the frame area.

[4]The recorded videos are available on http://andreas.isberg.se/edu/masters_thesis/

[5]The videos were recorded in 29.97 FPS, or 30000/1001 to be even more precise. 30 is however the number specified when choosing frame rate and resolution in the camera.

[6]MEncoder can be found on http://www.mplayerhq.hu/

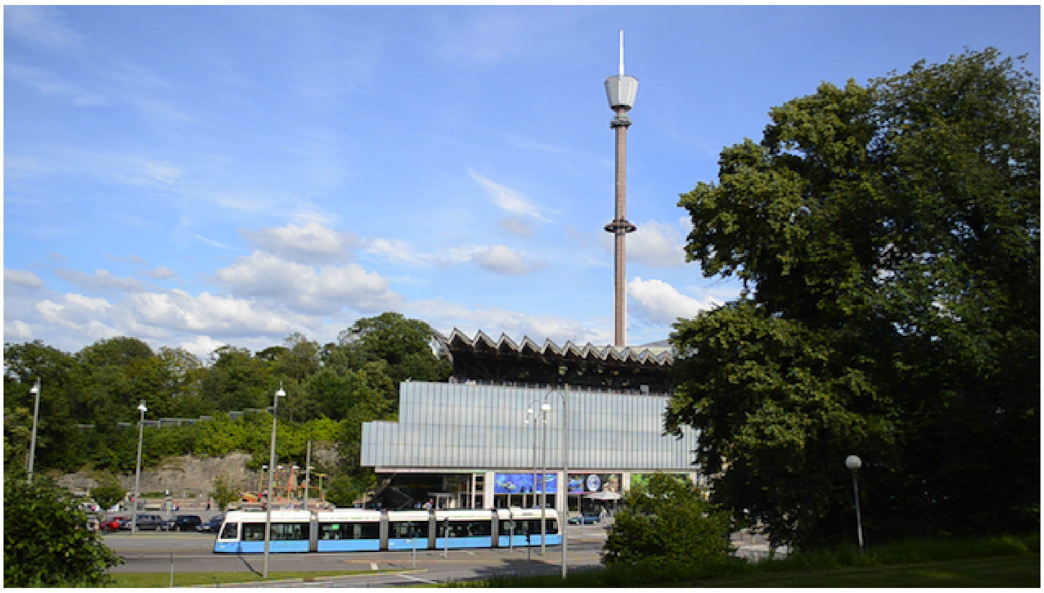


Figure 9.2: *Liseberg* test video (1080p). The tower in the background is an amusement park attraction.

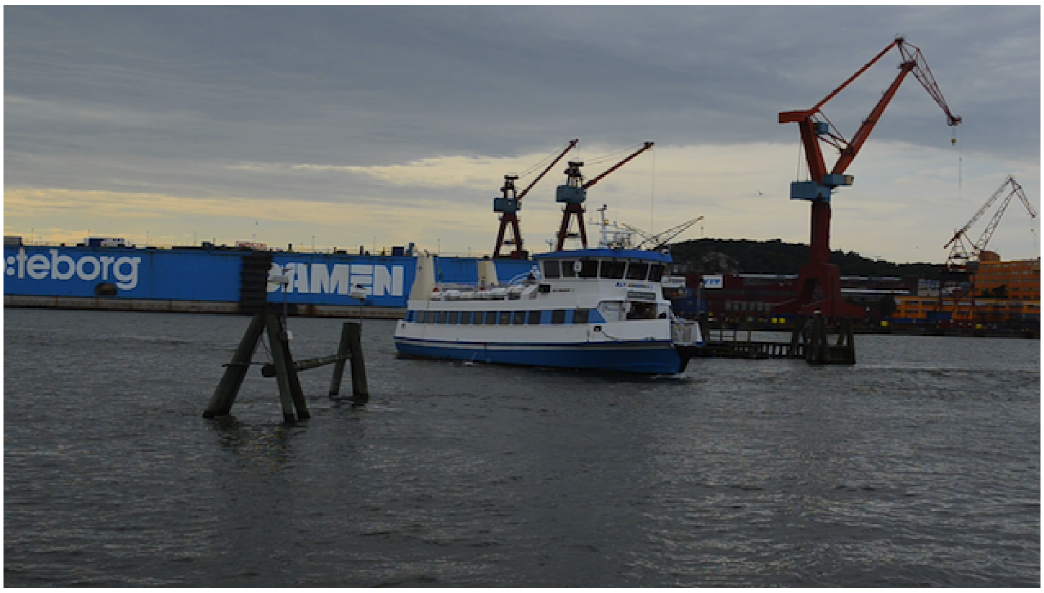


Figure 9.3: *FerryClose* test video (1080p).

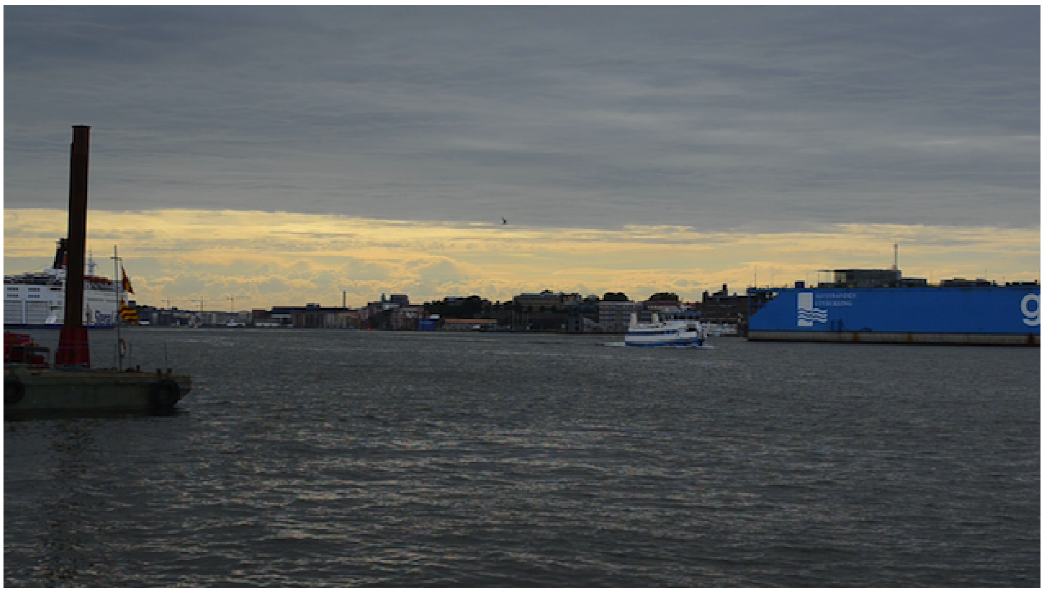


Figure 9.4: *FerryFar* test video (1080p).

9.2 Test system specification

The specifications of systems used for testing can be seen in Table 9.1 below. Two different classes of test systems, LAPTOP and DESKTOP, were used when collecting timing results.

PART	LAPTOP	DESKTOP
Processor	Core i7-2720QM	Core i5-750
Physical cores	4	4
Virtual cores	8	4
RAM amount	4 GB	4 GB
RAM specs	1.33 GHz DDR3	1.6 GHz DDR3
Graphics card	Radeon 6750M	GeForce GTX560 Ti
VRAM	1024 MB	1024 MB
Graphics driver	8.812.0.0	275.33
OS	Win7 Pro	Win7 Pro

Table 9.1: Specifications of test systems.

Both systems used processors from Intel, with different number of physical and logical processor cores. The DESKTOP system had four physical and virtual cores, while the processor in LAPTOP also had four physical cores but a total of eight virtual cores thanks to Hyper-Threading (HT). The number of Open Multi-Processing (OpenMP) threads were based on the number of virtual cores. The graphics card in DESKTOP was a graphics card from Nvidia, which at the time of writing could be placed in the high-end category. The laptop had a mid-range graphics chip from AMD.

Microsoft Visual Studio 2010 was used to compile the code. The compiler was set to full speed optimization, with link-time- and SSE2 code generation turned on, as well as whole program optimization.

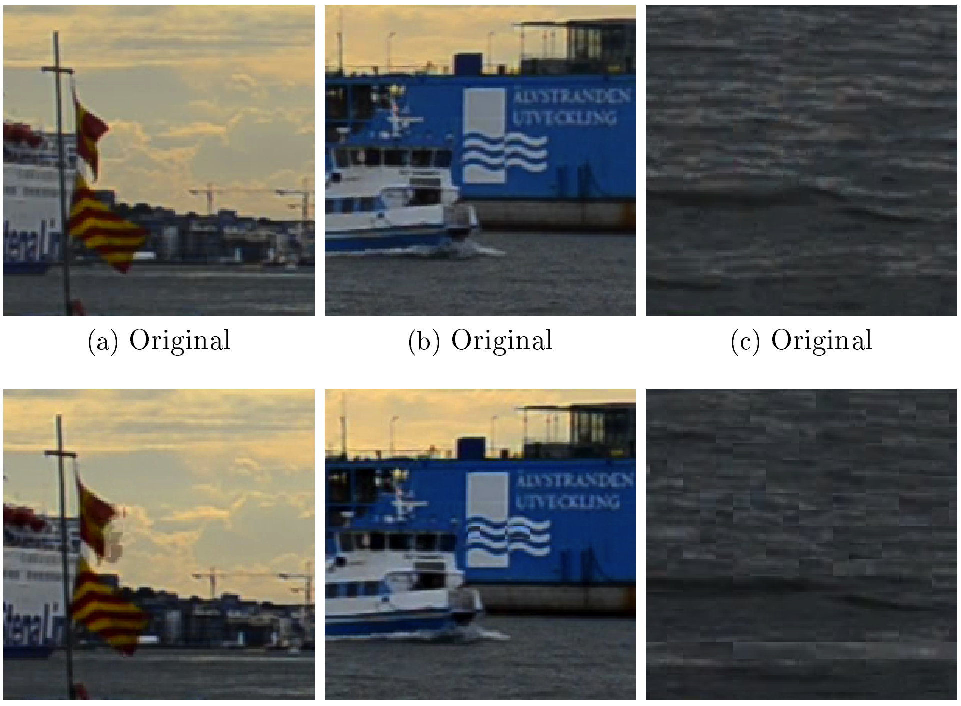
9.3 Subjective quality evaluation

9.3.1 Recorded videos

Looking at a $3x$ up-conversion, from 10 to 30 FPS (again, this is a rounding of 29.97), of the *FerryFar* sequence, the different motion estimation algorithms show some clear differences. The Four Step Search (4SS) algorithm produces the best result for this video. A few artefacts appear by the flag in the lower left corner (Figure 9.5d), and some in the water. The non-uniform motion in the water is well-handled and no artefacts are large or persistent enough to be considered particularly disturbing for the viewer.

Adaptive Rood Pattern Search (ARPS) produces a few artefacts on the ferry and around flag. However, the main difference compared to the other algorithms is the degree of artefacts in the water, which is extremely high for ARPS in this video. This can be explained by the "wandering" behaviour of the algorithm.

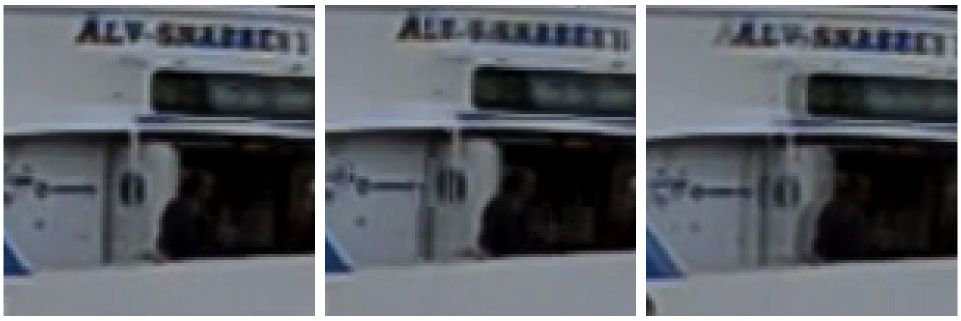
Artefacts in the water produced by the Full Bidirectional Search (FBDS) algorithm are worse than 4SS but significantly better than ARPS. Brief artefacts appear on the ferry, and some on a logo in the background (Figure 9.5e). Full Search (FS) gives similar results to FBDS, with the same kind of artefacts near the logo in the background.


(a) Original (b) Original (c) Original

(d) 4SS (e) FBDS (f) ARPS

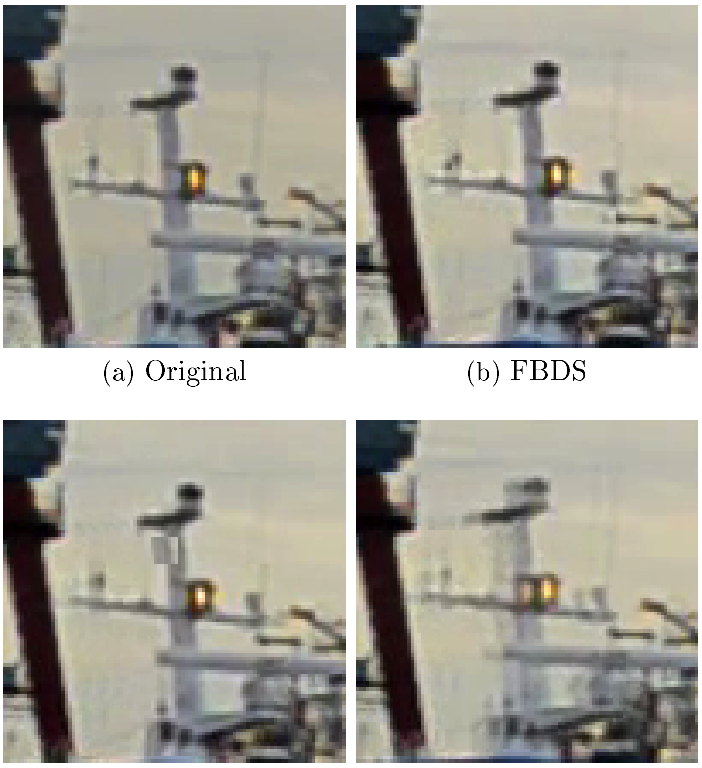
Figure 9.5: Artefacts in $3x$ up-conversion from 10 to 30 FPS of *FerryFar* sequence, 8x8 extended block size. Note the unintended "shadow" of the flag in (d) compared to the original (a), and the misaligned waves in the logo (e) compared to (b). The water in (f) contains several rows of miscoloured blocks.

The water motion in the *FerryClose* sequence, when up-converted from 10 to 30 FPS, is also best handled by 4SS and worst by ARPS. Disturbing artefacts, primarily around the ferry's windows, are produced by 4SS (Figure 9.8f). This is not the case for FBDS, where only a few artefacts can be found on the side and front of the ferry, the thin antenna on top and heads of the people on the upper deck (Figure 9.8d). FBDS also shows problems with a bird flying by fast, which appear to flicker (Figure 9.8e). Despite this, FBDS gives the overall best subjective result for this up-conversion factor, as illustrated in the sharpness comparisons in Figure 9.6. The benefits of up-conversion with motion estimation (ME) and motion compensation (MC), compared to simple frame averaging (FA), is shown in Figure 9.7.



(a) FBDS (b) 4SS (c) FA

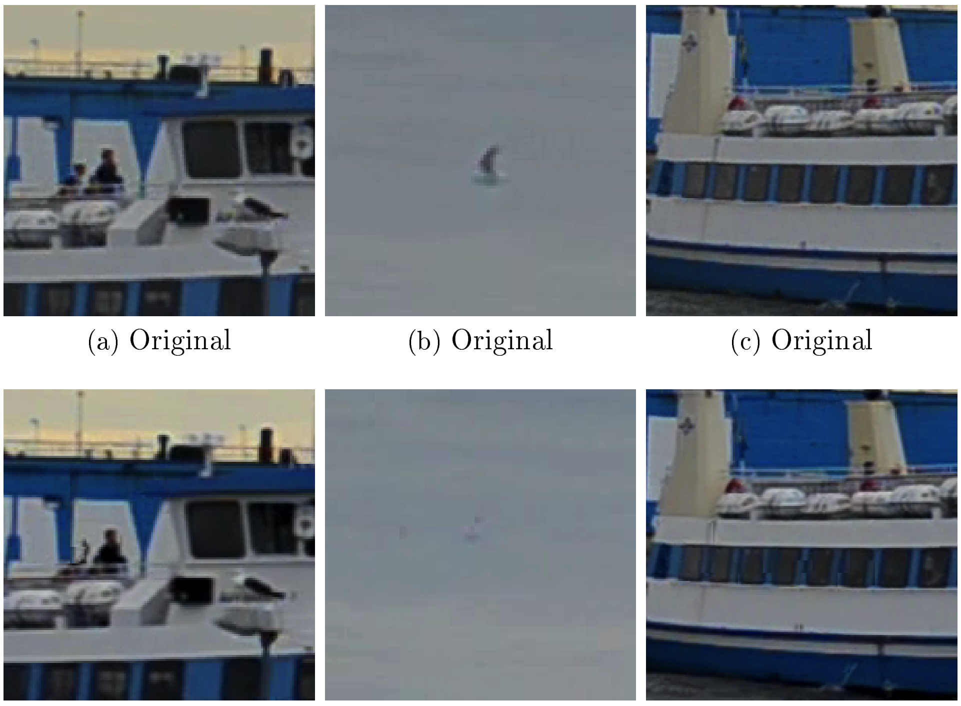
Figure 9.6: Sharpness comparison, $3x$ up-conversion from 10 to 30 FPS of *FerryClose* sequence. With FBDS the text is sharp and readable, while 4SS gives less text shadow but also less readable text compared to FA.



(a) Original (b) FBDS

(c) 4SS (d) FA

Figure 9.7: Sharpness compared to original frame, $3x$ up-conversion from 10 to 30 FPS of *FerryClose*, 8x8 extended block size (see Section 8.2.2). Note the slightly displaced antenna in (b) and (c), and the block artefact in (c). FA is outclassed.


(a) Original (b) Original (c) Original

(d) FBDS (e) FBDS (f) 4SS

Figure 9.8: Artefacts in $3x$ up-conversion from 10 to 30 FPS of *FerryClose* sequence, 8x8 extended block size. The head of the person to the left in (a) is mostly gone in (d), same goes for the bird in (b) when compared to (e). Artefacts produced by 4SS on the side of the ferry can be seen between the windows in (f).

When converting *FerryClose* from 30 to 60 FPS however, doubling the original frame rate, ARPS produce much less artefacts and 4SS few or none. Artefacts produced by FBDS are limited to single blocks on the side and front of the ferry.

For the *Liseberg* sequence an up-conversion from 10 to 30 FPS gives video quality problems for all ME algorithms. Artefacts produced by ARPS around the tram are particularly heavy (Figure 9.9c), but slightly less around the amusement park tower compared to the other algorithms (Figure 9.10c). The limited range of 4SS and FBDS is not enough to find the true motion of the tram (Figures 9.9b and 9.9d). FBDS again can not handle a fast-moving bird, which appear to flicker.

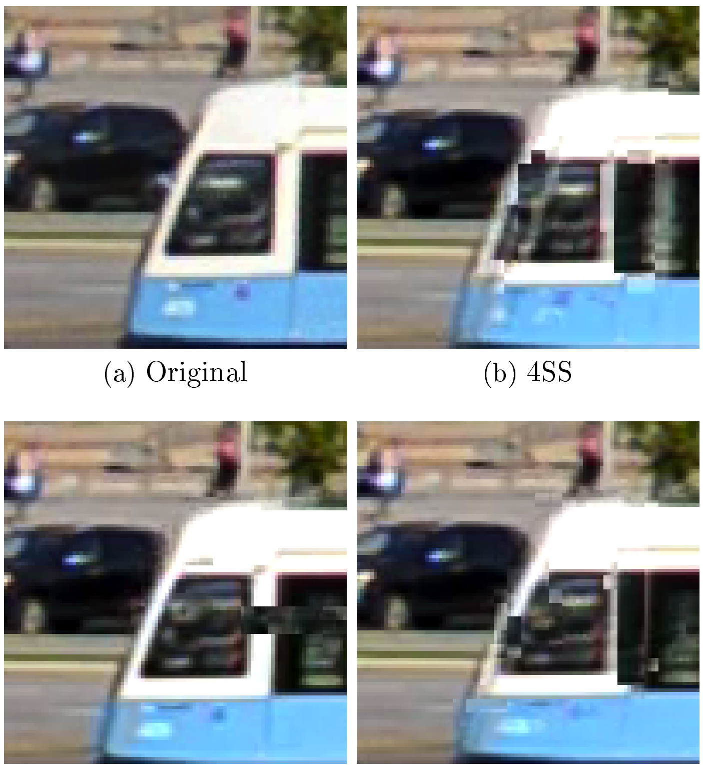

(a) Original (b) 4SS (c) ARPS (d) FBDS

Figure 9.9: Artefacts on the front of the tram in 3x up-conversion from 10 to 30 FPS of *Liseberg* sequence, 4x4 extended block size.

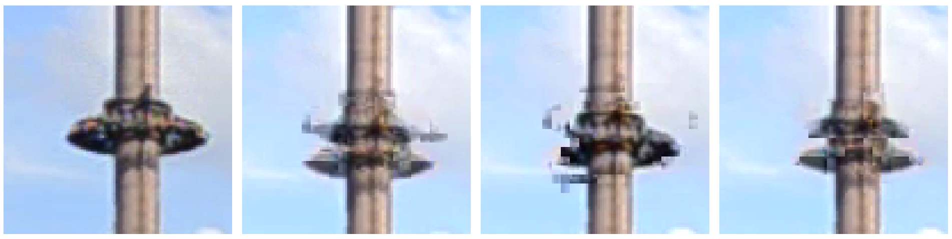
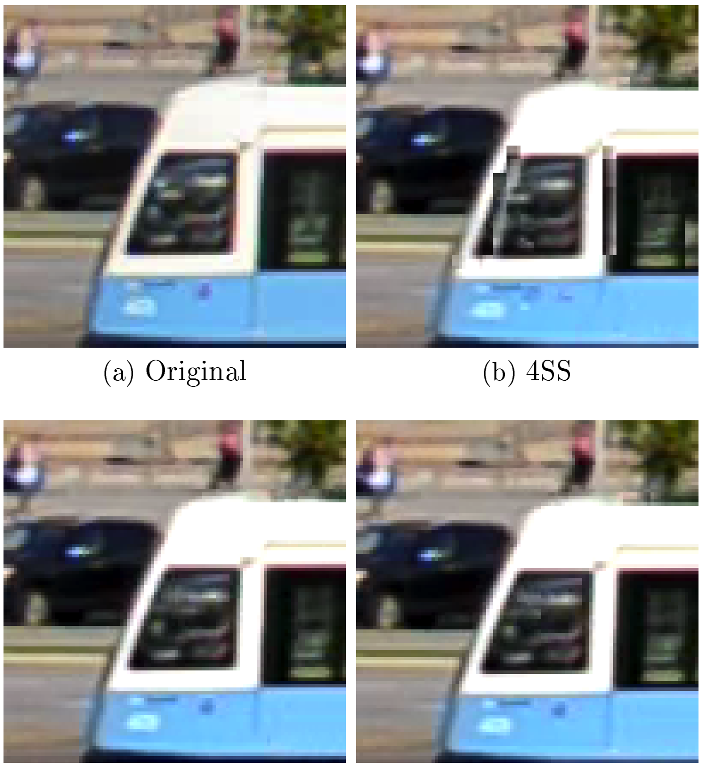

(a) Original (b) 4SS (c) ARPS (d) FBDS

Figure 9.10: Artefacts on a high-motion part (the "ring" is falling fast towards the ground) of an amusement park tower, in 3x up-conversion from 10 to 30 FPS of *Liseberg* sequence, 4x4 extended block size. The limited range of 4SS and FBDS is probably the reason for the severe artefacts in (b) and (d).

When up-converting from 15 to 30 FPS all algorithms give better results on the *Liseberg* sequence (Figure 9.11), though ARPS still produce sporadic artefacts. FBDS gives few to none artefacts on the tram and also rivals ARPS for best results on the vertical motion (Figure 9.12), which at its maximum speed still is too large for the range of FBDS. When going from 30 to 60 FPS, only ARPS has some limited trouble with the tram motion.


(a) Original (b) 4SS

(c) ARPS (d) FBDS

Figure 9.11: Artefact comparison on tram, $2x$ up-conversion from 15 to 30 FPS of *Liseberg* sequence, 4x4 extended block size. In this frame ARPS gave less artefacts than 4SS, but seen to the entire sequence the case is the opposite.

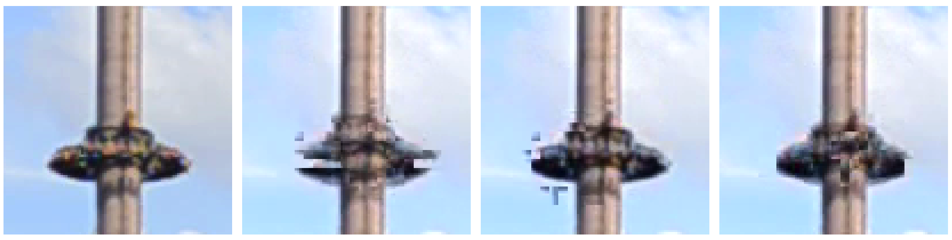

(a) Original (b) 4SS (c) ARPS (d) FBDS

Figure 9.12: Artefacts on tower in *Liseberg* sequence, $2x$ up-conversion from 15 to 30 FPS.

9.3.2 Standard test videos

The motion in the news anchor's face in a $3x$ up-conversion of the *Akiyo* test sequence, going from 10 to 30 FPS, is mostly handled well by 4SS, ARPS and FBDS. Problematic ranges of frames often involve facial regions being covered or uncovered as an effect of quick movements of eyes and lips. Figure 9.13 and Figure 9.15 show such events. As indicated by the semi-visible eyes in Figure 9.13e the news anchor is about to open her eyes. This motion was correctly interpolated by ARPS and FBDS, and with a single block artefact by 4SS. Figure 9.14 is in the middle of vertical face motion. ARPS and FBDS both produce acceptable results, whereas 4SS gives an extra pair of eyebrows. 4SS fares better on frame 137, where FBDS is the worst of the four algorithms compared (Figure 9.15d), making *Akiyo* one of the few test videos where ARPS gives the best result.

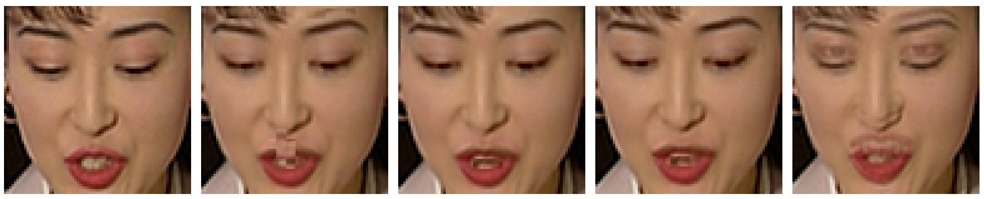

(a) Original (b) 4SS (c) ARPS (d) FBDS (e) FA

Figure 9.13: Excerpt of frame 38, $3x$ up-conversion of *Akiyo* sequence, from 10 to 30 FPS.

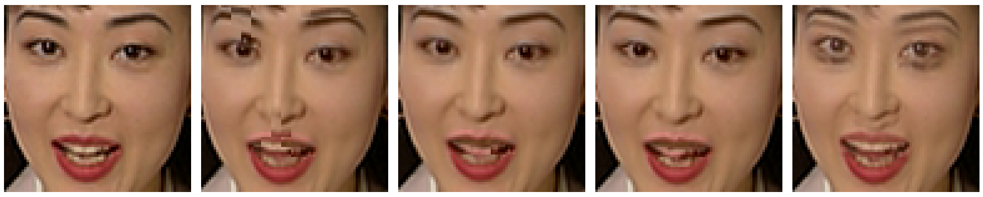

(a) Original (b) 4SS (c) ARPS (d) FBDS (e) FA

Figure 9.14: Excerpt of frame 76, $3x$ up-conversion of *Akiyo* sequence, from 10 to 30 FPS.

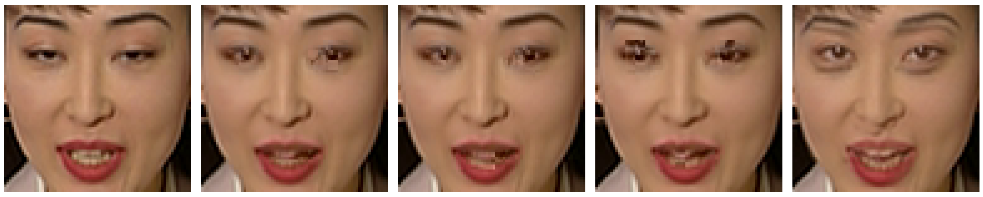

(a) Original (b) 4SS (c) ARPS (d) FBDS (e) FA

Figure 9.15: Excerpt of frame 137, $3x$ up-conversion of *Akiyo* sequence, from 10 to 30 FPS.

The first part of the *Container* video sequence, before two birds enter the scene, contains motion so slow that even a $6x$ up-conversion can give highly acceptable results with 4SS, ARPS, FBDS or even FA (Figure 9.16). Figure 9.17, showing one of the birds, gives an example of how badly such events are handled by the tested algorithms.

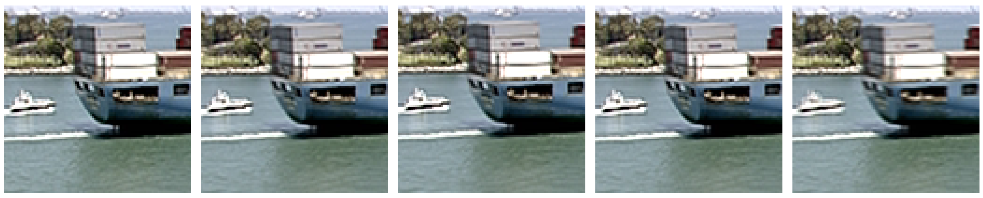

(a) Original (b) 4SS (c) ARPS (d) FBDS (e) FA

Figure 9.16: Excerpt of frame 111, $6x$ up-conversion of *Container* sequence. 4SS, ARPS and FBDS all give a result with acceptable sharp contours of the ships. FA gives a little less sharpness compared to the others.

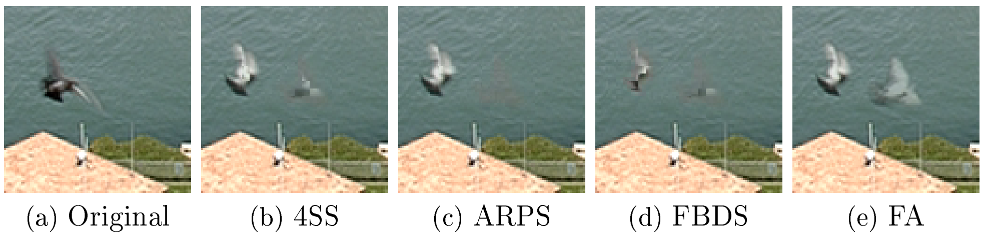

(a) Original (b) 4SS (c) ARPS (d) FBDS (e) FA

Figure 9.17: Excerpt of frame 262, 3x up-conversion of *Container* sequence.

Most parts of the *Foreman* sequence contain motion too large or complicated for the chosen algorithms to tackle, illustrated in Figure 9.18. As stated earlier, good results on this video has not been a focus of our efforts, and the video is included primarily for reference due to its high popularity in this field of research.

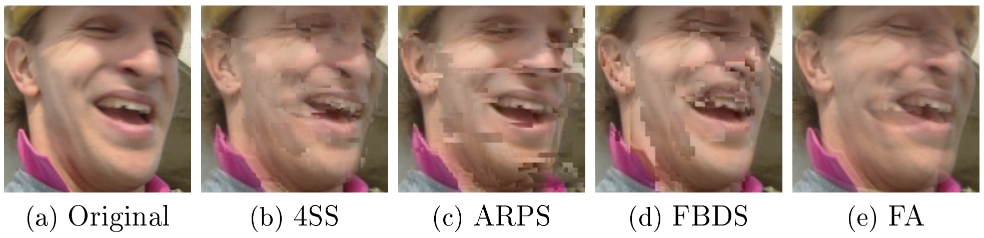

(a) Original (b) 4SS (c) ARPS (d) FBDS (e) FA

Figure 9.18: Excerpt of frame 13, 3x up-conversion of *Foreman* sequence.

With the subjective evaluations of all test videos in mind, the smaller block size of 4x4 produce roughly equal amount of artefacts as 8x8. However, as the artefacts are also smaller, they appear to be less disturbing for the viewer.

9.4 Objective quality evaluation

Most of the peak signal-to-noise ratio (PSNR) results obtained from up-converting the videos indicates that FA performs really well, even though the subjective quality evaluation sometimes suggests otherwise. PSNR uses the pixel differences between two frames, and this difference is small for videos with low motion and low color contrast. This is a major problem with PSNR and the results should only be treated as a rough estimation.

The diagrams below contain average results for each algorithm combined with four different block sizes. They also contain per-frame results for each algorithm using 4x4 extended block size, as this was found to be the best choice in the subjective quality evaluation.

9.4.1 Recorded videos

Structural Similarity (SSIM) was not computed for the 1080p videos because of limitations in the program used for testing.

Figures 9.19 to 9.22 show that 4SS performs best of the motion estimation algorithms on *FerryFar* and *FerryClose*, similar to the results from the subjective quality evaluation. The PSNR is quite constant as the videos contain just small motion.

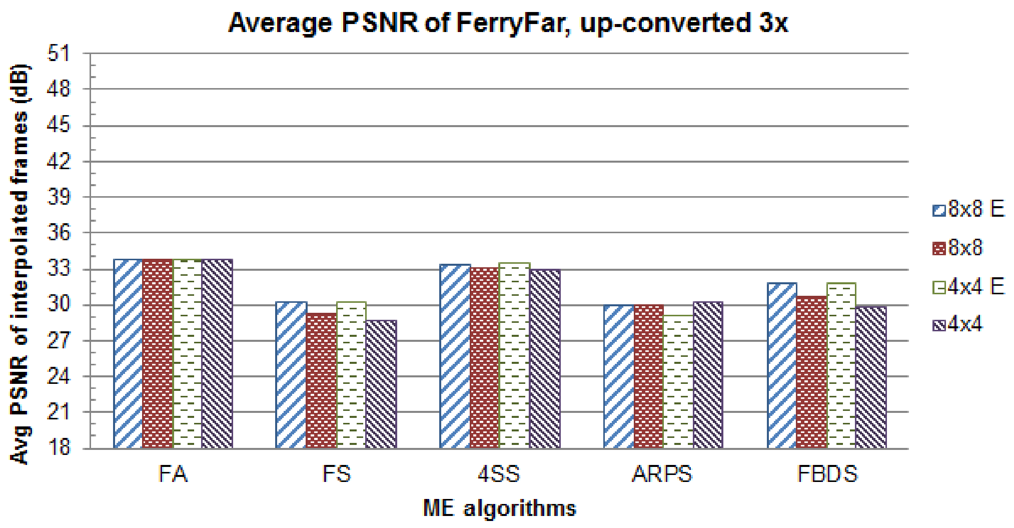


Figure 9.19: Average PSNR of *FerryFar* after up-converting it from 10 to 30 FPS.

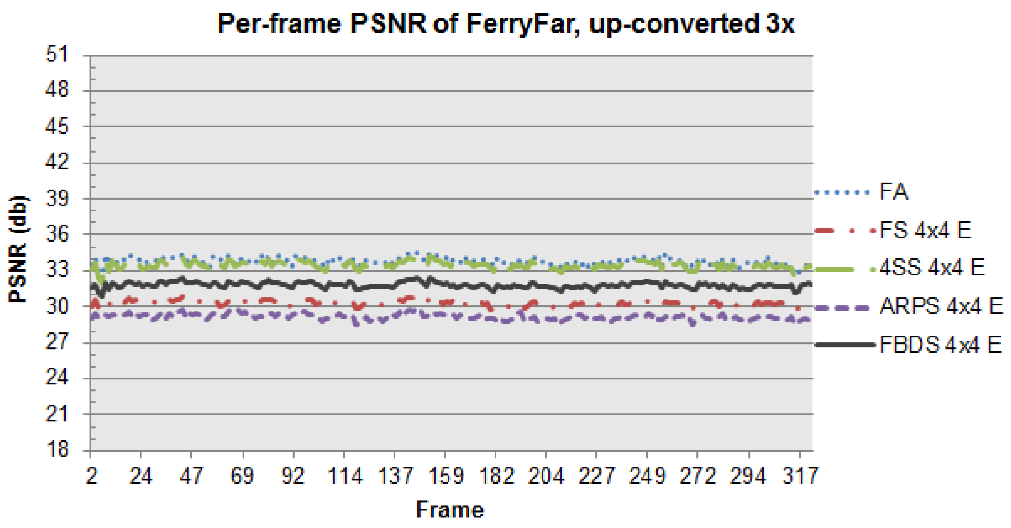


Figure 9.20: Per-frame PSNR of *FerryFar* after up-converting it from 10 to 30 FPS.

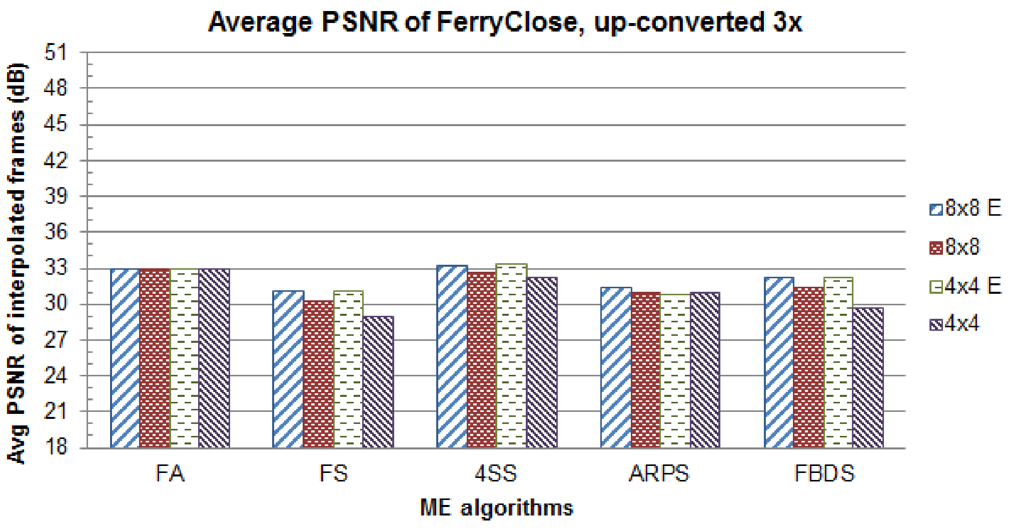


Figure 9.21: Average PSNR of *FerryClose* after up-converting it from 10 to 30 FPS.

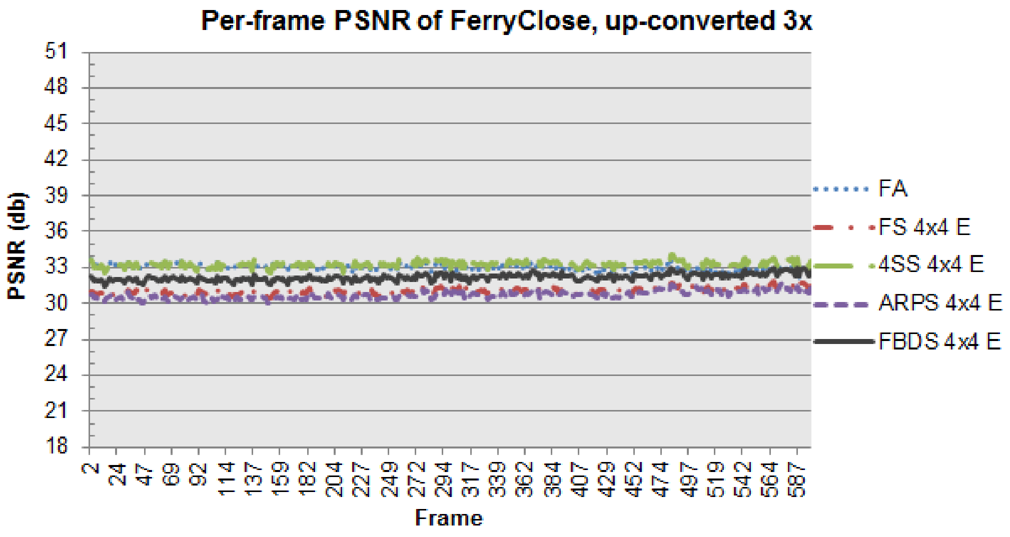


Figure 9.22: Per-frame PSNR of *FerryClose* after up-converting it from 10 to 30 FPS. 4SS performs slightly better than the other algorithms.

Up-converting *Liseberg* from 10 to 30 FPS gives reasonably good objective results for all algorithms, as shown in Figures 9.23 to 9.24. These results are however an effect of the mostly static scene. Comparing these figures to the subjective quality evaluation (Section 9.4.1) confirms that high PSNR values does not necessarily correspond to high quality in the few parts of the video frame where there is motion interesting for a human observer.

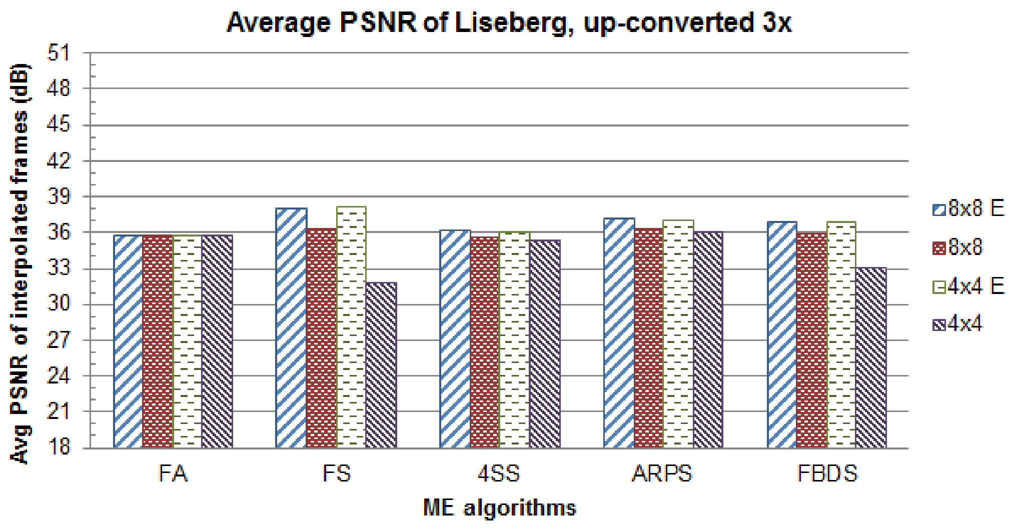


Figure 9.23: Average PSNR of *Liseberg* after up-converting it from 10 to 30 FPS.

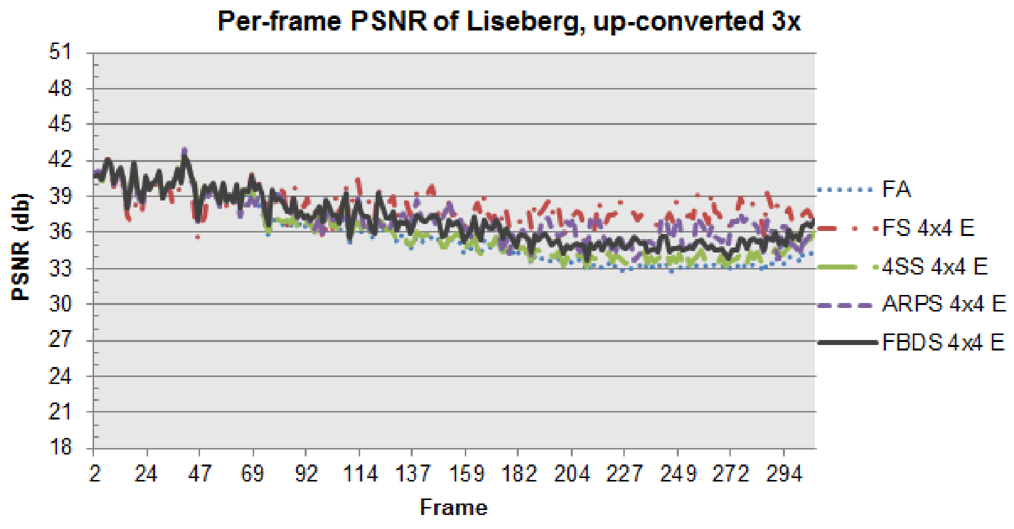


Figure 9.24: Per-frame PSNR of *Liseberg* after up-converting it from 10 to 30 FPS.

All algorithms perform slightly better (by a few decibels) when up-converting *Liseberg* from 15 to 30 FPS. FBDS performs excellent, hitting over 40 decibel on average.

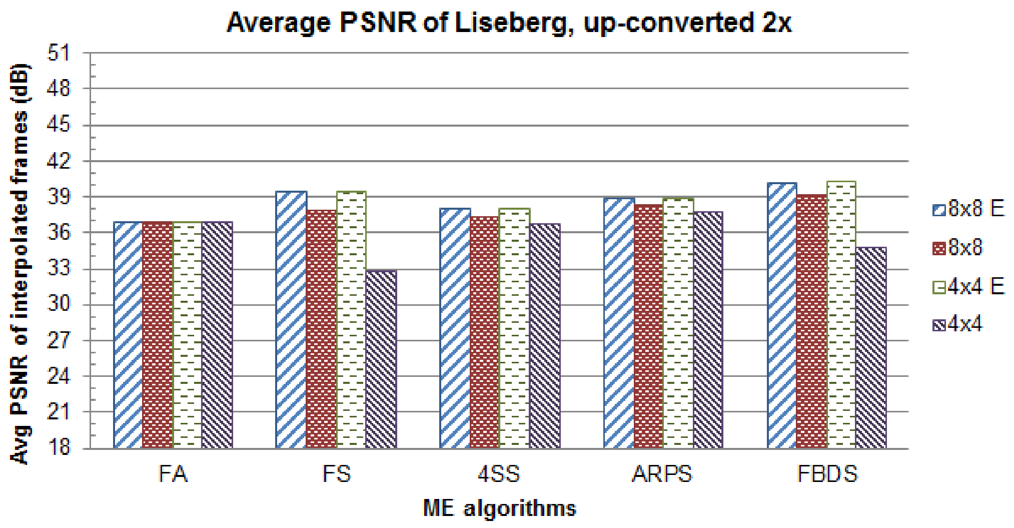


Figure 9.25: Average PSNR of *Liseberg* after up-converting it from 15 to 30 FPS.

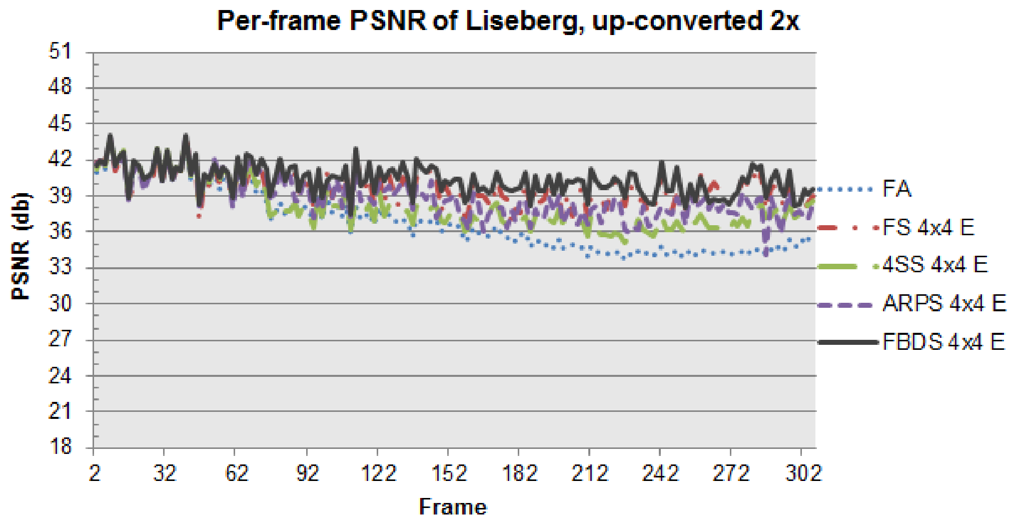


Figure 9.26: Per-frame PSNR of *Liseberg* after up-converting it from 15 to 30 FPS.

9.4.2 Standard videos

All algorithms perform excellent on *Akiyo*, hitting over 40 decibels on average, as shown in Figure 9.27.

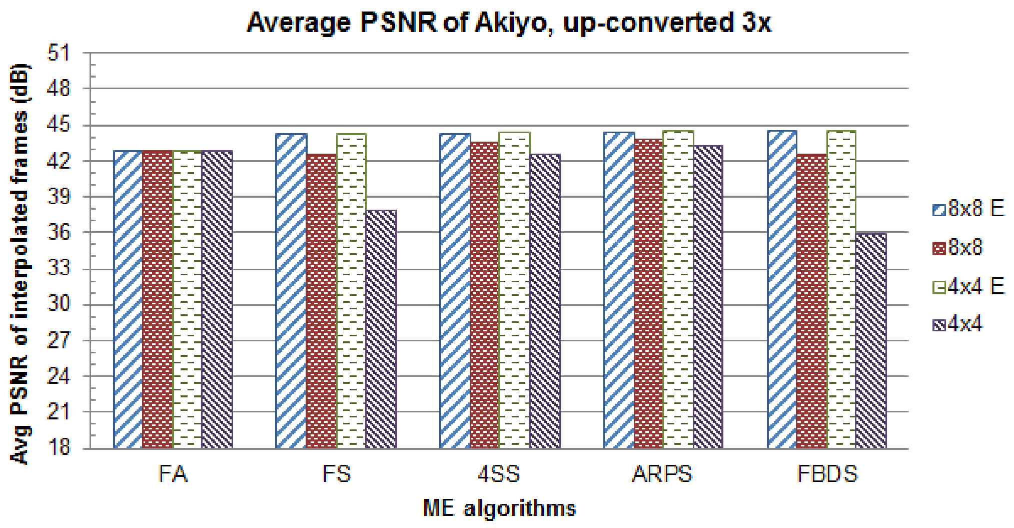


Figure 9.27: Average PSNR of *akiyo* after up-converting it from 10 to 30 FPS.

Figures 9.28 and 9.29 show how the result of FA drops low for many frames. This is likely because of the color contrast between the light skin and the dark areas such as the hair, eyebrows and eyes.

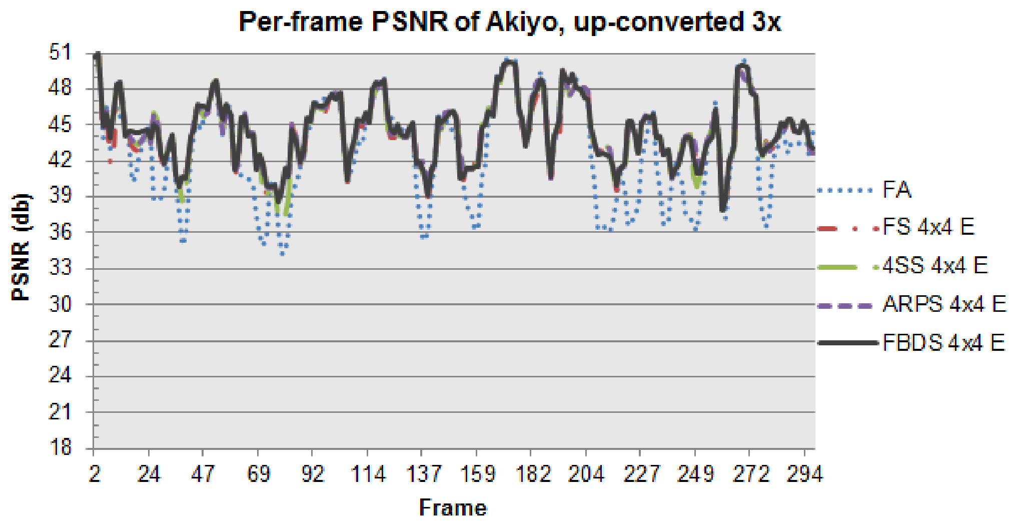


Figure 9.28: Per-frame PSNR of *Akiyo* after up-converting it from 10 to 30 FPS. Notice how FA drops very low for certain frames.

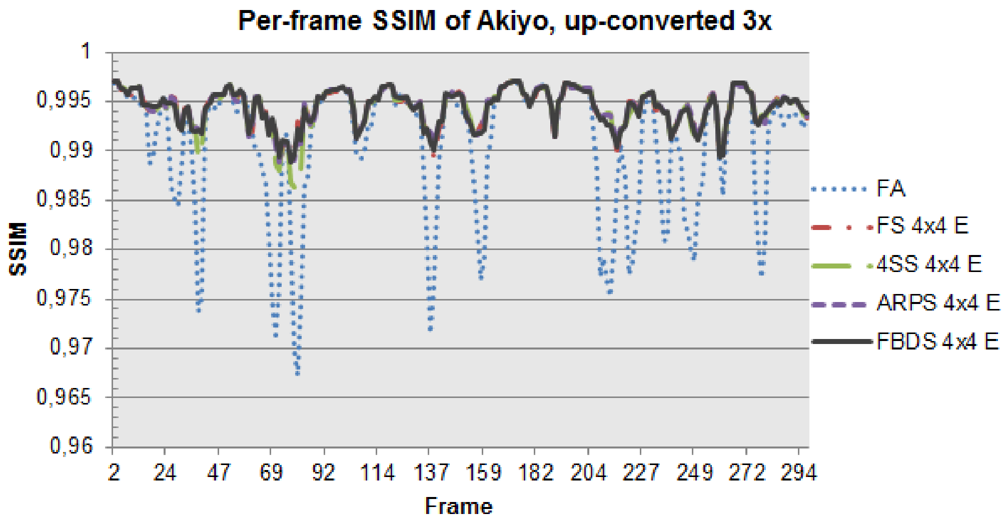


Figure 9.29: Per-frame SSIM of *Akiyo* after up-converting it from 10 to 30 FPS. Notice how FA drops very low for certain frames.

The algorithms also perform excellent on *Container*, as shown in Figure 9.30.

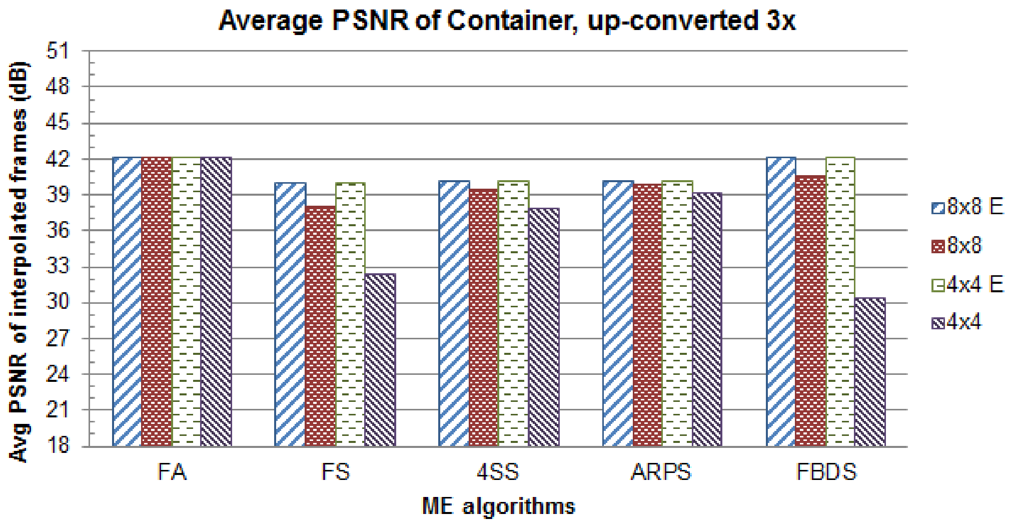


Figure 9.30: Average PSNR of *Container* after up-converting it from 10 to 30 FPS.

Figures 9.31 and 9.32 show how the result of all algorithms drops low between frames 245 and 276. This is due to the fact that none of the algorithms properly can handle the high motion of the two birds.

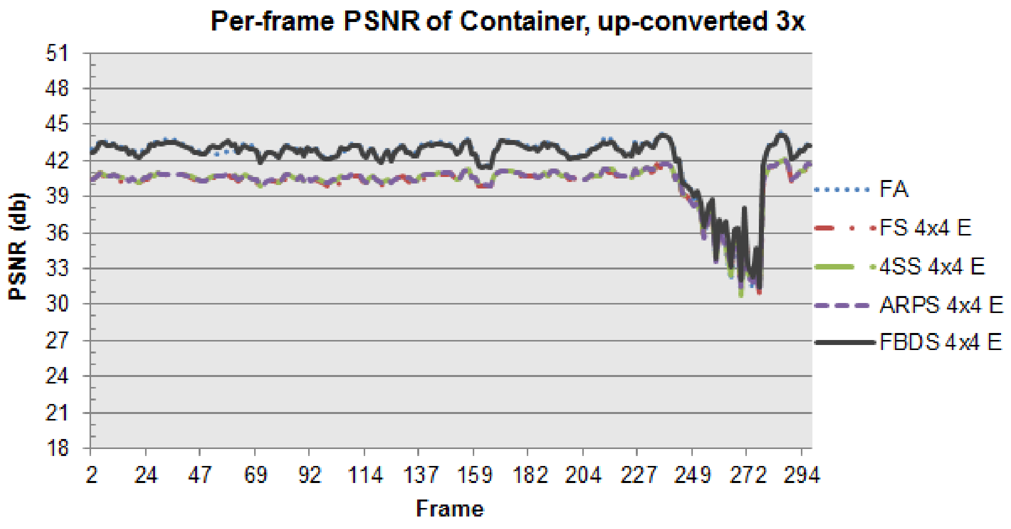


Figure 9.31: Per-frame PSNR of *Container* after up-converting it from 10 to 30 FPS. FBDS performs excellent on most frames, except when the two birds fly by, hitting well over 40 decibels on average.

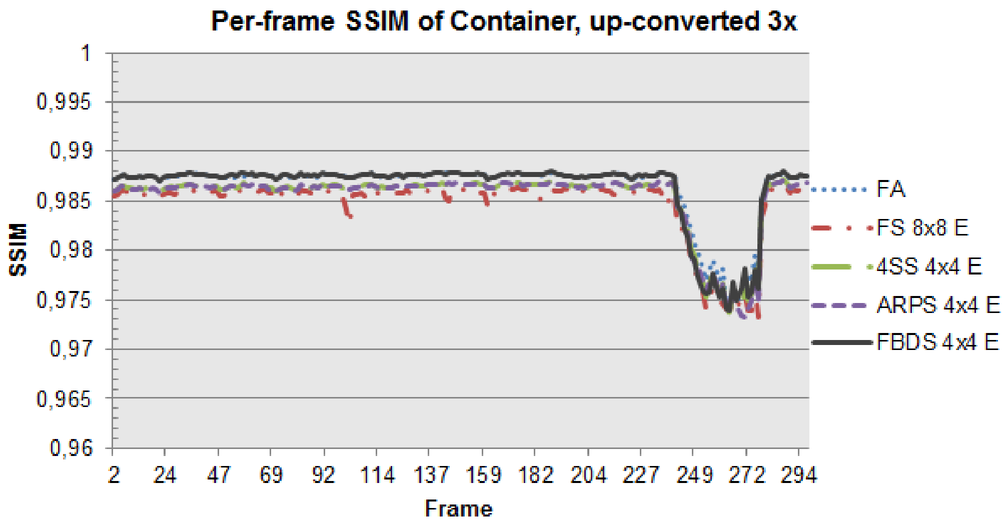


Figure 9.32: Per-frame SSIM of *Container* after up-converting it from 10 to 30 FPS.

As expected, none of the algorithms perform well on *Foreman*, as Figures 9.33

to 9.35 show.

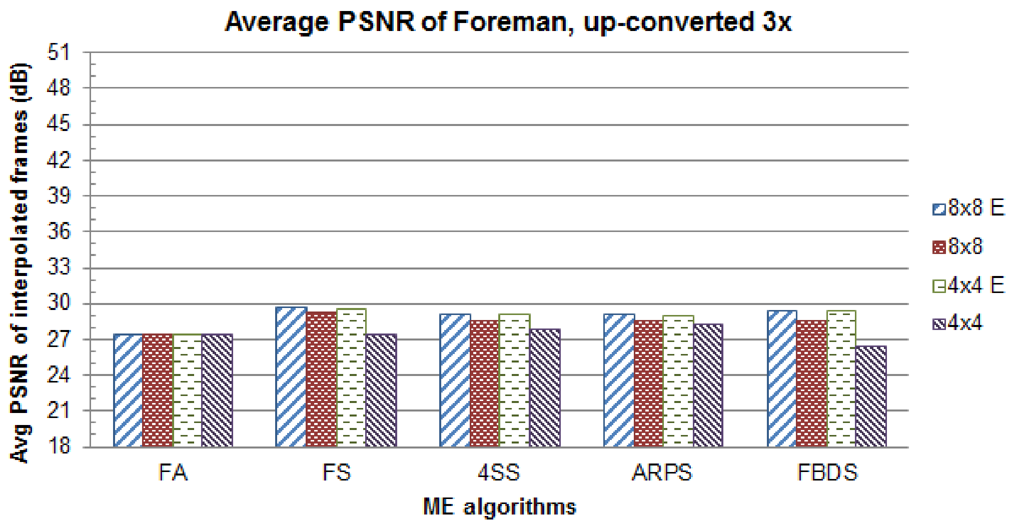


Figure 9.33: Average PSNR of *Foreman* after up-converting it from 10 to 30 FPS.

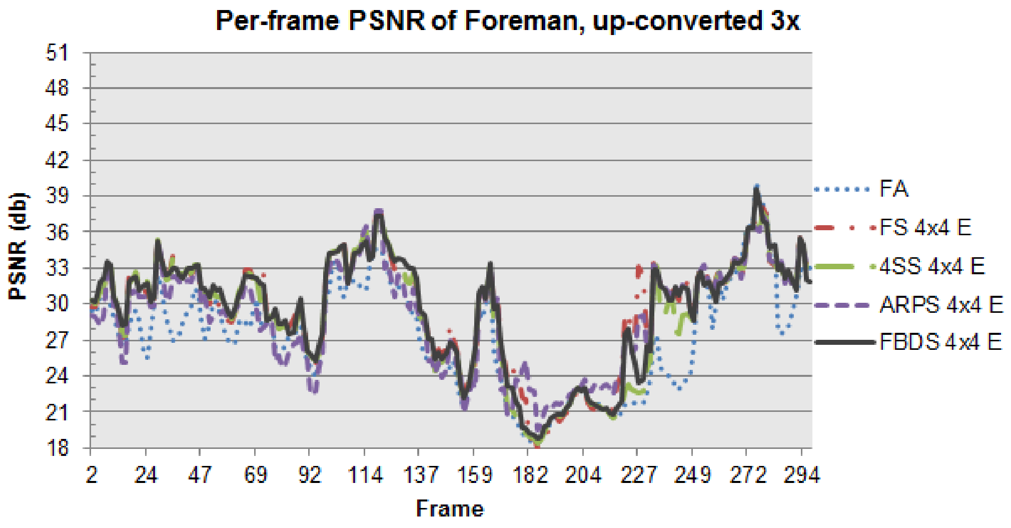


Figure 9.34: Per-frame PSNR of *Foreman* after up-converting it from 10 to 30 FPS.

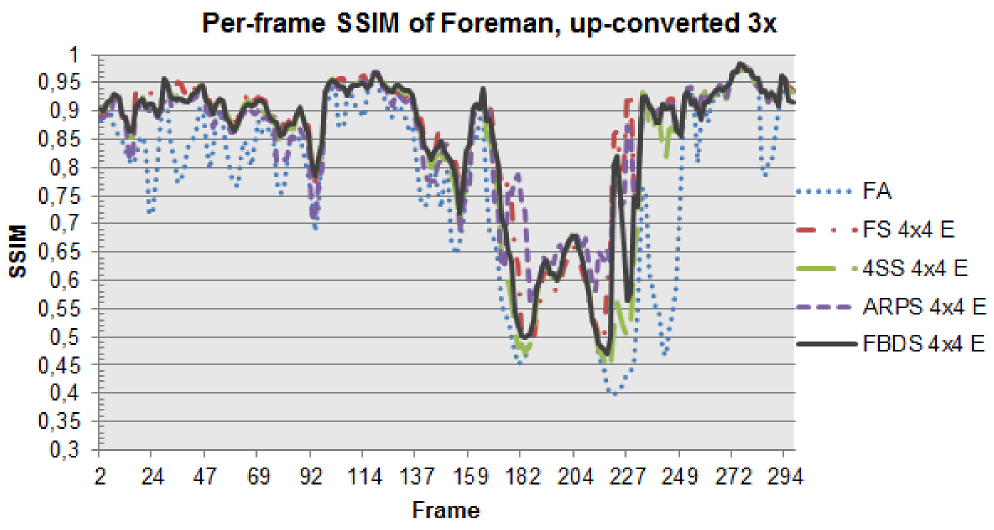


Figure 9.35: Per-frame SSIM of *Foreman* after up-converting it from 10 to 30 FPS. Note that the values of the y-axis ranges from 0.3 to 1.0 instead of 0.960 to 1.0, which was the case for the other standard videos. FA is the worst algorithm for this video.

9.5 Real-time capability

The diagrams below contain per-frame timing results for 4SS, ARPS and FBDS, with 4x4 extended block size, as these were found to give the best quality.

9.5.1 Motion estimation time measurements

Enabling optimizations could in theory give huge improvements. OpenMP could increase the performance by the number of virtual cores in the system, and the Sum of Absolute Differences (SAD) assembly optimization could increase the performance up to 16 times (see Section 8.2.1). Table 9.2 below shows how they actually effect the runtime on ME. The optimizations are not algorithm nor video-type dependent, so the table shows the average per-frame time measurements for 4SS using 4x4 extended block size on the recorded 1080p videos.

Enabling OpenMP on LAPTOP (8 virtual cores) gives a speed increase factor of 3.3, while the same optimization on DESKTOP (4 virtual cores) only increases the speed 2.4 times. Enabling the SAD assembly optimization gives a speed increase factor of 11.5 on LAPTOP, and 10.8 on DESKTOP (Table 9.3).

	LAPTOP	DESKTOP
No optimization	680.6 ms	721.3 ms
OpenMP only	205.6 ms	301.3 ms
SSE only	59.3 ms	66.6 ms
OpenMP and SSE	37.3 ms	43.3 ms

Table 9.2: Average time for different optimizations on ME using 4SS with 4x4 extended block size.

	LAPTOP	DESKTOP
No optimization	1.0x	1.0x
OpenMP only	3.3x	2.4x
SSE only	11.5x	10.8x
OpenMP and SSE	18.2x	16.7x

Table 9.3: Relative time improvements with optimizations, from data in Table 9.2

All algorithms, with 4x4 extended block size and all optimizations enabled, are extremely fast on the standard CIF videos, as Table 9.4 shows.

	LAPTOP	DESKTOP
4SS	1.2 ms	1.2 ms
ARPS	1.1 ms	1.1 ms
FBDS	3 ms	2.7 ms

Table 9.4: Average time for ME for the standard CIF videos using 4x4 extended block size and optimizations.

Figures 9.36 to 9.41 show the performance of the ME algorithms, using 4x4 extended block size and optimizations enabled, on the recorded 1080p videos. Some of the charts show spikes where processing times are increased by up to 100 ms or more for some frames. This is not caused by instantaneous changes in the videos. A likely explanation is the process scheduling of the underlying operating system giving priority to other processes. Setting a higher priority on the main frame rate up-converter thread should remedy this. The issue does however highlight the inherent difficulty of guaranteeing that a maximum processing time limit is not exceeded.

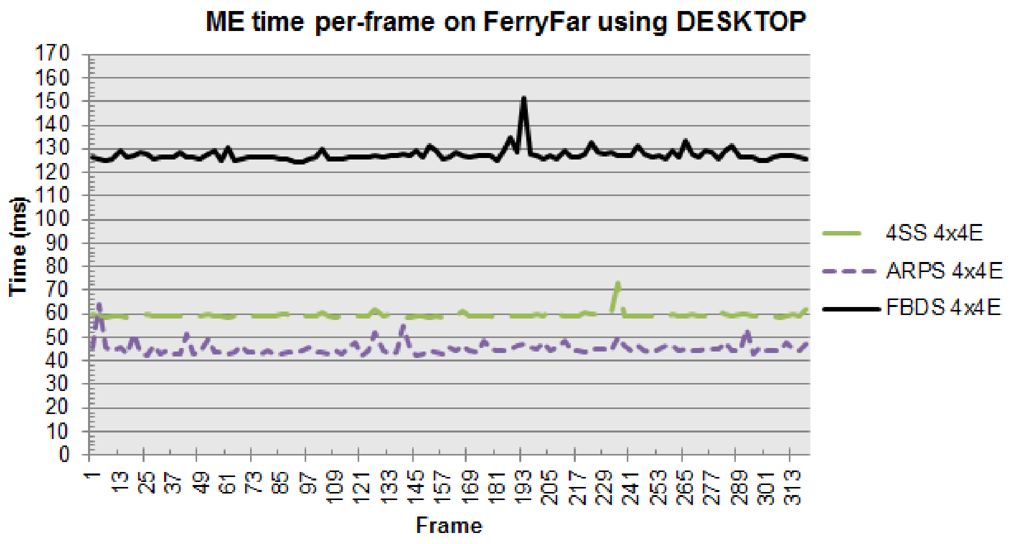


Figure 9.36: ME on *FerryFar* using DESKTOP after up-converting it from 10 to 30 FPS.

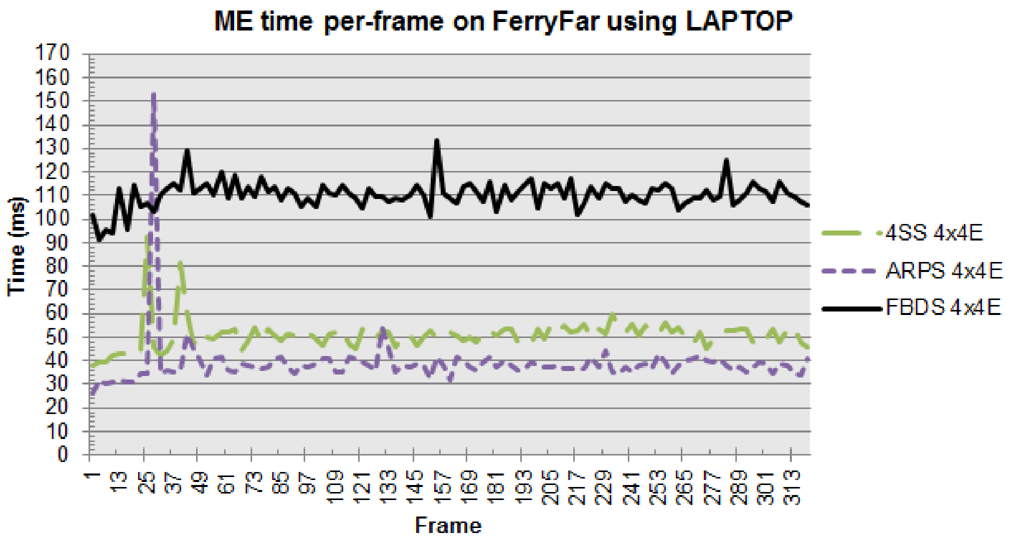


Figure 9.37: ME on *FerryFar* using LAPTOP after up-converting it from 10 to 30 FPS.

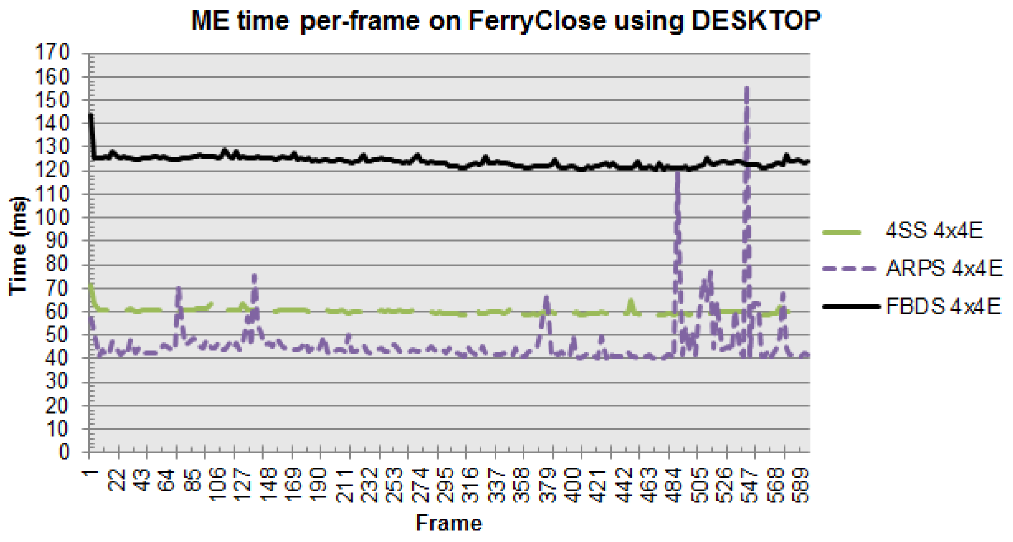


Figure 9.38: ME on *FerryClose* using DESKTOP after up-converting it from 10 to 30 FPS.

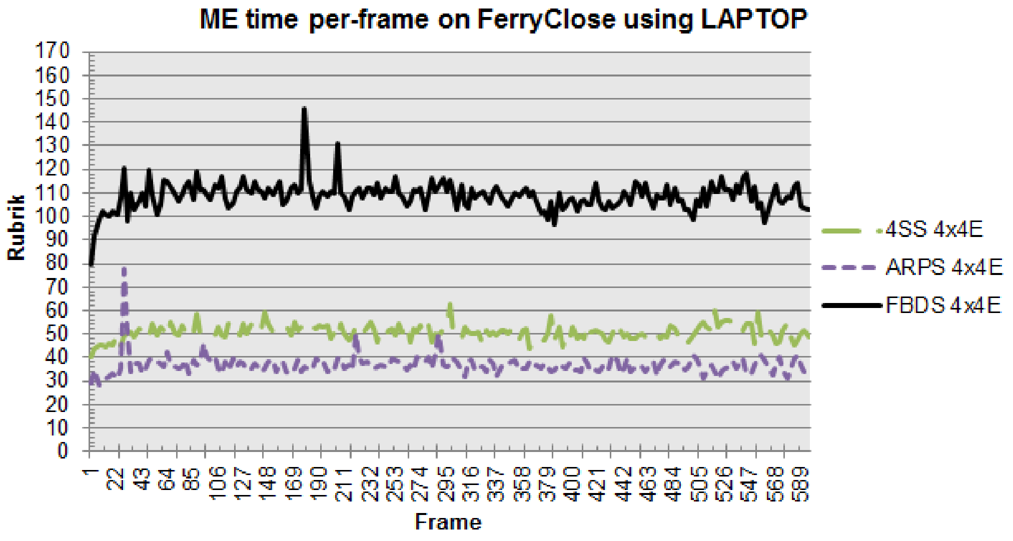


Figure 9.39: ME on *FerryClose* using LAPTOP after up-converting it from 10 to 30 FPS.

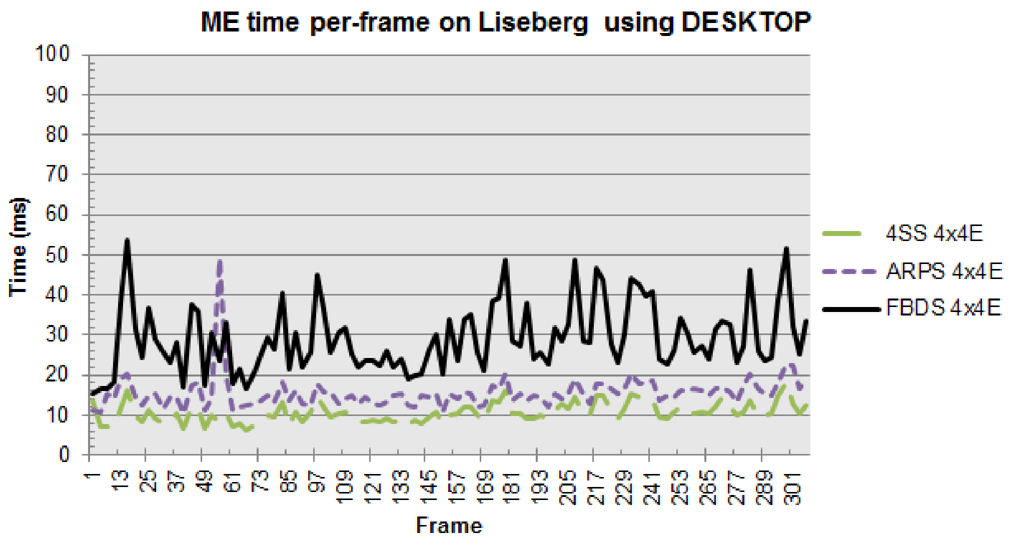


Figure 9.40: ME on *Liseberg* using DESKTOP after up-converting it from 10 to 30 FPS.

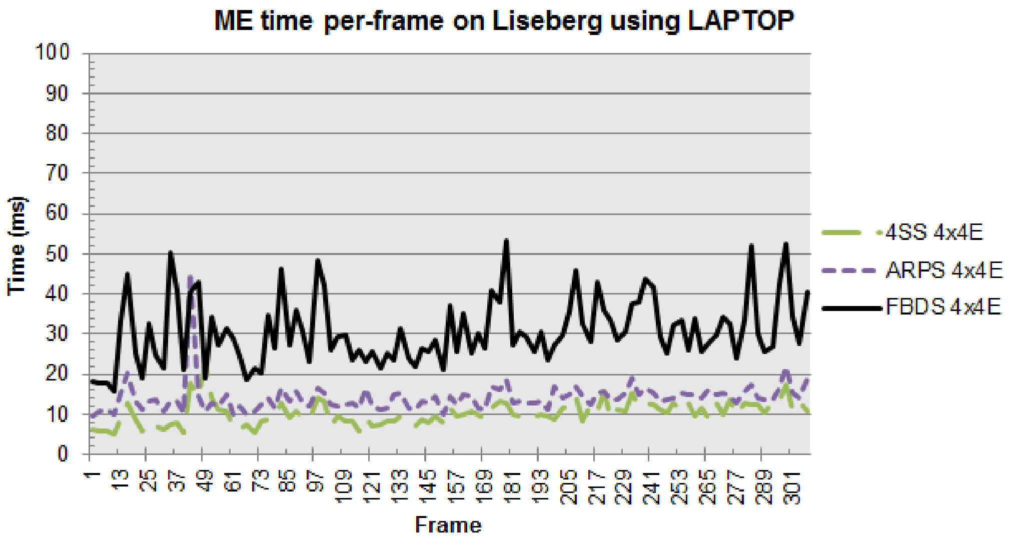


Figure 9.41: ME on *Liseberg* using LAPTOP after up-converting it from 10 to 30 FPS.

9.5.2 Motion compensation time measurements

Table 9.5 below shows the MC processing times for the CPU- and OpenCL-based BDMC implementations. The test was performed on *Liseberg* using 4SS and 4x4 extended block size.

The results show a big performance difference between the CPUs and GPUs in each test system. Looking only at processing times, the CPU-based implementation is a better choice for the LAPTOP test system. However, the GPU-based implementation gives increased quality thanks to the bilinear interpolation. The DESKTOP test system benefits hugely from the high-end graphics card.

	LAPTOP	DESKTOP
CPU	19.7 ms	27.5 ms
OpenCL (GPU)	25.5 ms	14.9 ms

Table 9.5: Comparison of CPU- and OpenCL-based BDMC implementations. The test was performed on *Liseberg* using 4SS and 4x4 extended block size.

Based on the results above, the systems benefit most from the OpenCL-based BDMC implementation. Tables 9.6 and 9.7 below show how OpenMP effects the OpenCL-based BDMC implementation on the two test systems. Differences in motion between videos only give small variations on processing times. The DESKTOP system combined with OpenMP is fast enough for a true real-time implementation (see Section 2.1.2), where each processing step needs to be below 16.7 ms. The difference of using OpenMP in the otherwise GPU-based algorithm comes from the fact that chroma is still processed on the CPU (see Section 8.3).

	LAPTOP	DESKTOP
With OpenMP	25.8 ms	13.1 ms
Without OpenMP	37.6 ms	25.2 ms

Table 9.6: Average time for MC on the recorded 1080p videos.

9.5.3 Total processing times

The total processing time, taking both ME and MC into account, can be used to determine if the real-time goals set in Section 2.1.2 can be accomplished.

	LAPTOP	DESKTOP
With OpenMP	3 ms	0.9 ms
Without OpenMP	6.7 ms	1.7 ms

Table 9.7: Average time for MC on the standard CIF videos.

All videos used in testing was first down-converted by a factor of three and then up-converted by the same factor. This factor means the total processing time must not exceed 50 milliseconds, marked as LIMIT_HIGH in the charts.

The total processing time is low when used on CIF videos. Figure 9.42 shows that all algorithms just take a few milliseconds when used on *Akiyo*. The result is similar for the other CIF videos.

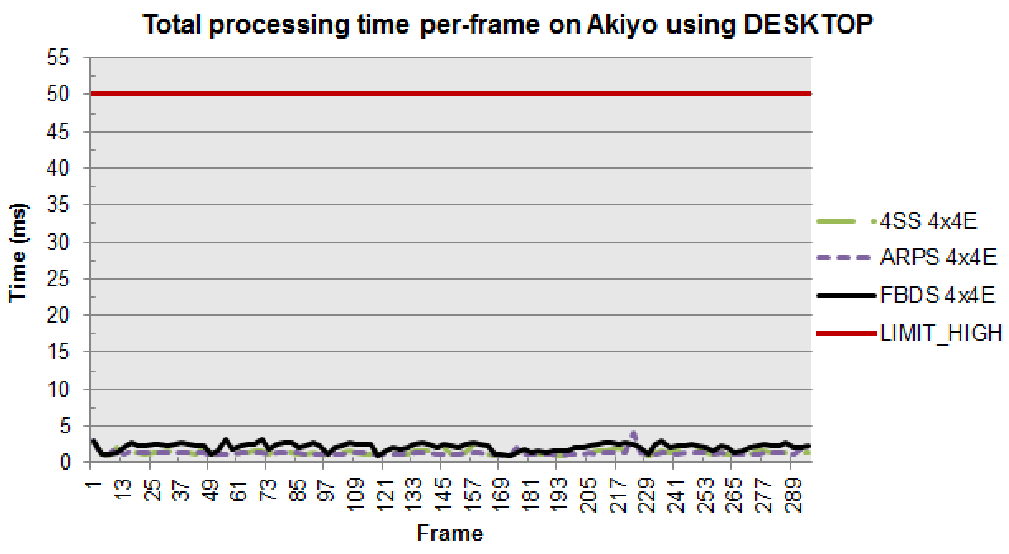


Figure 9.42: Total processing time (ME and MC) on *Akiyo* using DESKTOP after up-converting it from 10 to 30 FPS. The MC method used is BDMC.

On 1080p videos, the total processing time is much higher. Figures 9.43 to 9.46 show that none of the algorithms fall below the mark for *FerryFar* and *FerryClose*.

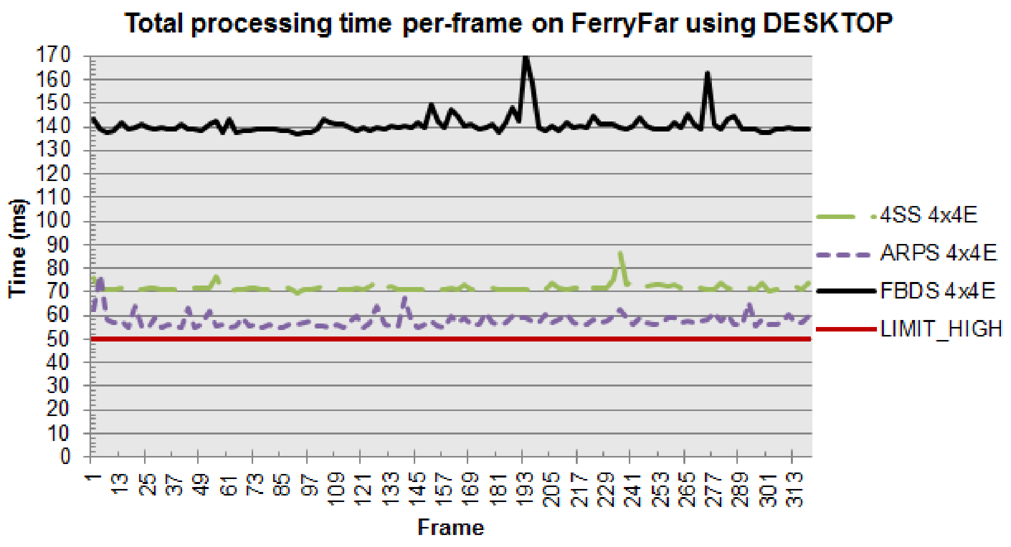


Figure 9.43: Total processing time (ME and MC) on *FerryFar* using DESKTOP after up-converting it from 10 to 30 FPS. The MC method used is BDMC.

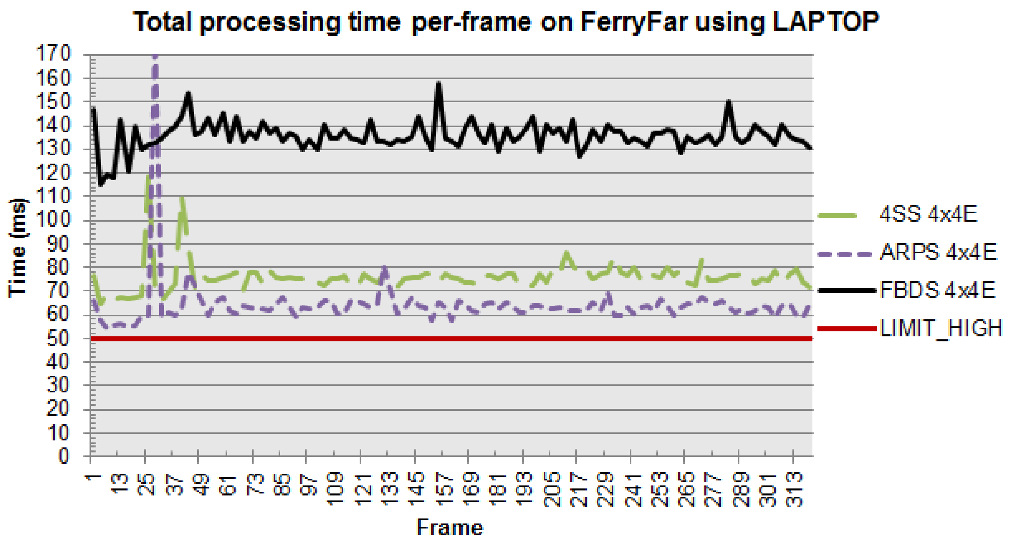


Figure 9.44: Total processing time (ME and MC) on *FerryFar* using LAPTOP after up-converting it from 10 to 30 FPS. The MC method used is BDMC.

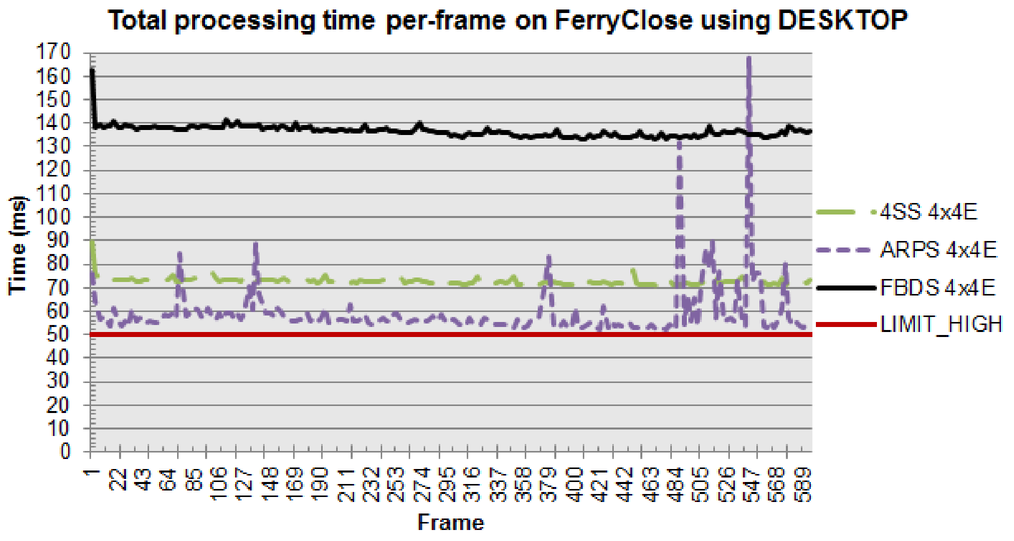


Figure 9.45: Total processing time (ME and MC) on *FerryClose* using DESKTOP after up-converting it from 10 to 30 FPS. The MC method used is BDMC.

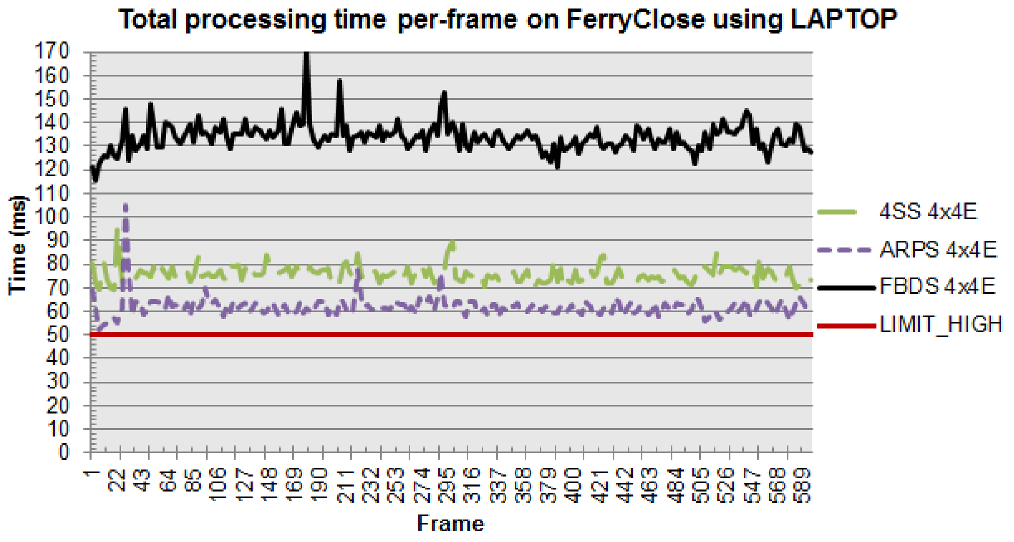


Figure 9.46: Total processing time (ME and MC) on *FerryClose* using LAPTOP after up-converting it from 10 to 30 FPS. The MC method used is BDMC.

Both test systems using 4SS or ARPS in combination with BDMC fall below the mark on *Liseberg*, as Figures 9.47 and 9.48 show.

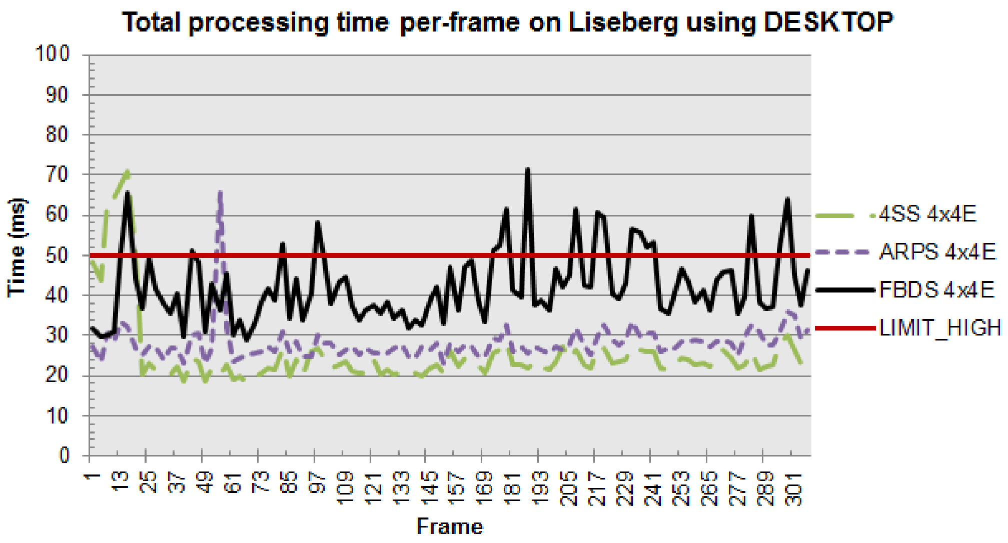


Figure 9.47: Total processing time (ME and MC) on *Liseberg* using DESKTOP after up-converting it from 10 to 30 FPS. The MC method used is BDMC.

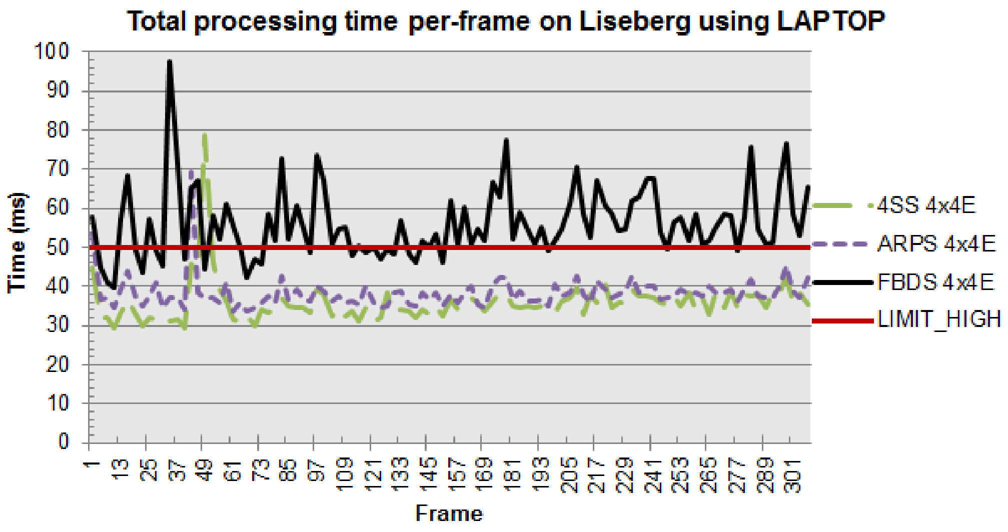


Figure 9.48: Total processing time (ME and MC) on *Liseberg* using LAPTOP after up-converting it from 10 to 30 FPS. The MC method used is BDMC.

FBDS falls below the 50 ms mark, on all recorded 1080p videos, if the block size is changed from 4x4 extended to 8x8 extended using DESKTOP test system. This is shown in Figures 9.49 to 9.51.

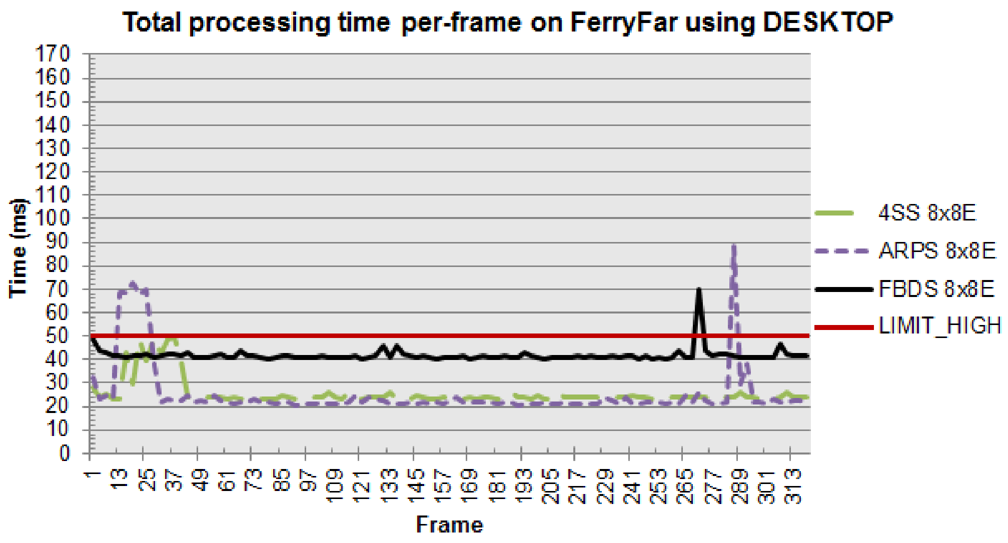


Figure 9.49: Total processing time (ME and MC) on *FerryFar* using DESKTOP after up-converting it from 10 to 30 FPS. The MC method used is BDMC. The blocksize used is 8x8 extended.

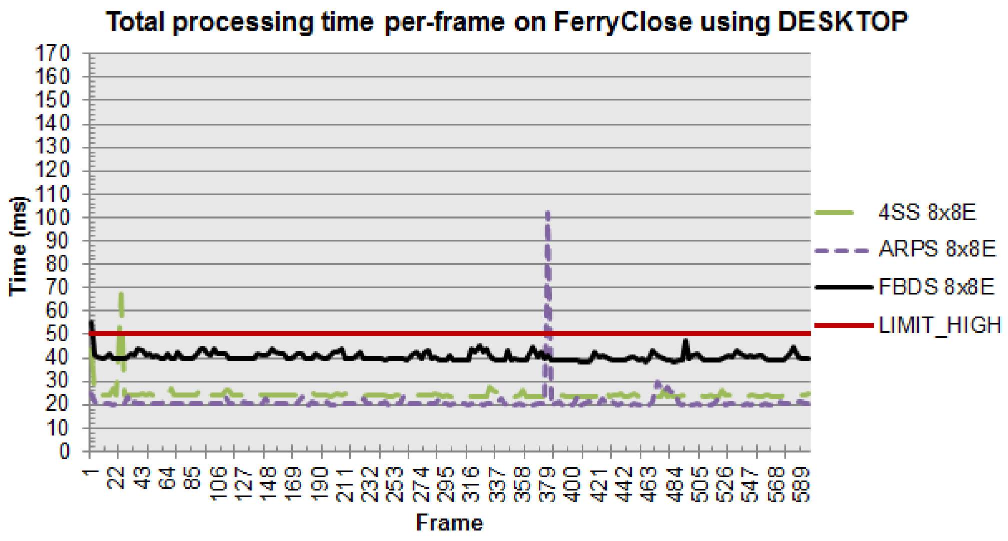


Figure 9.50: Total processing time (ME and MC) on *FerryClose* using DESKTOP after up-converting it from 10 to 30 FPS. The MC method used is BDMC. The blocksize used is 8x8 extended.

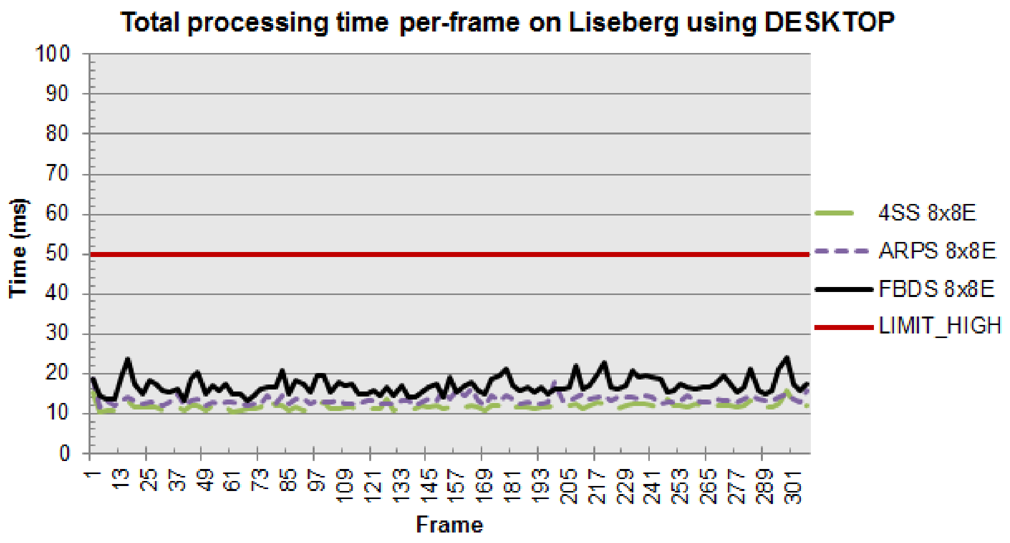


Figure 9.51: Total processing time (ME and MC) on *Liseberg* using DESKTOP after up-converting it from 10 to 30 FPS. The MC method used is BDMC. The blocksize used is 8x8 extended.

Using the LAPTOP test system with 8x8 extended block size, ARPS and 4SS fall below the mark on *FerryFar* and *FerryClose* (Figures 9.52 to 9.53). FBDS breaks the limit on too many frames to be acceptable in the usage scenario. All algorithms fall below the mark on *Liseberg* using the LAPTOP test system with 8x8 extended block size, as shown in Figure 9.54.

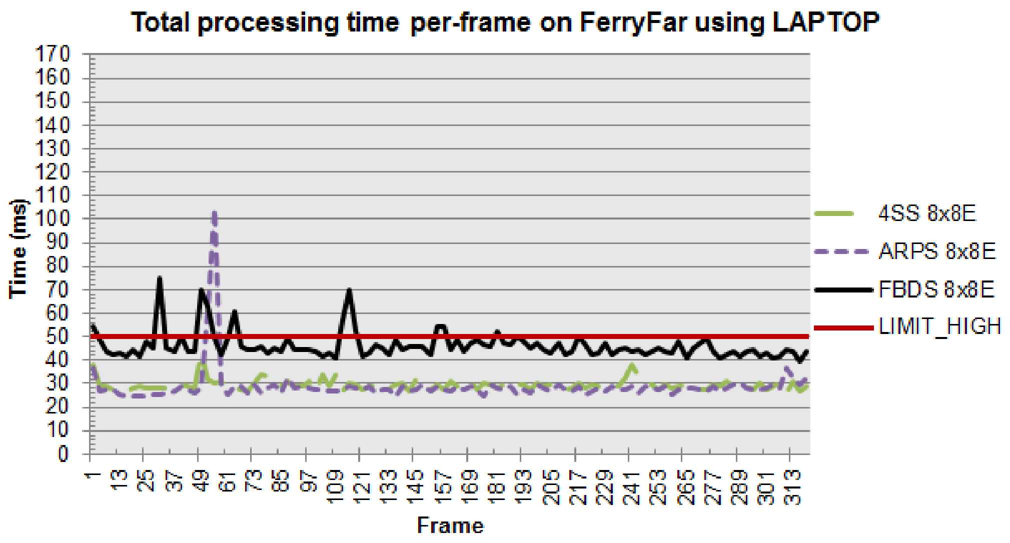


Figure 9.52: Total processing time (ME and MC) on *FerryFar* using LAPTOP after up-converting it from 10 to 30 FPS. The MC method is BDMC. The blocksize is 8x8 extended.

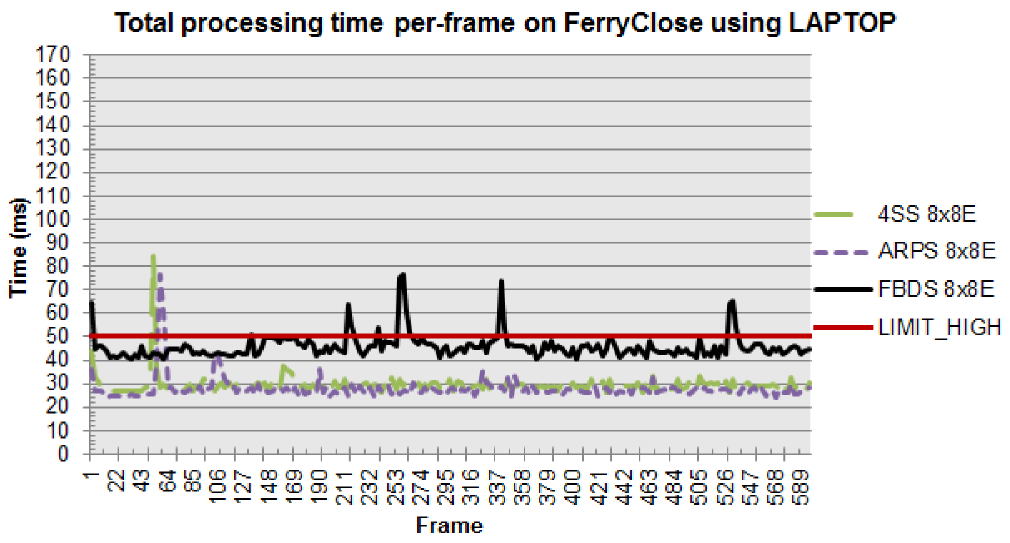


Figure 9.53: Total processing time (ME and MC) on *FerryClose* using LAPTOP after up-converting it from 10 to 30 FPS. The MC method used is BDMC. The blocksize used is 8x8 extended.

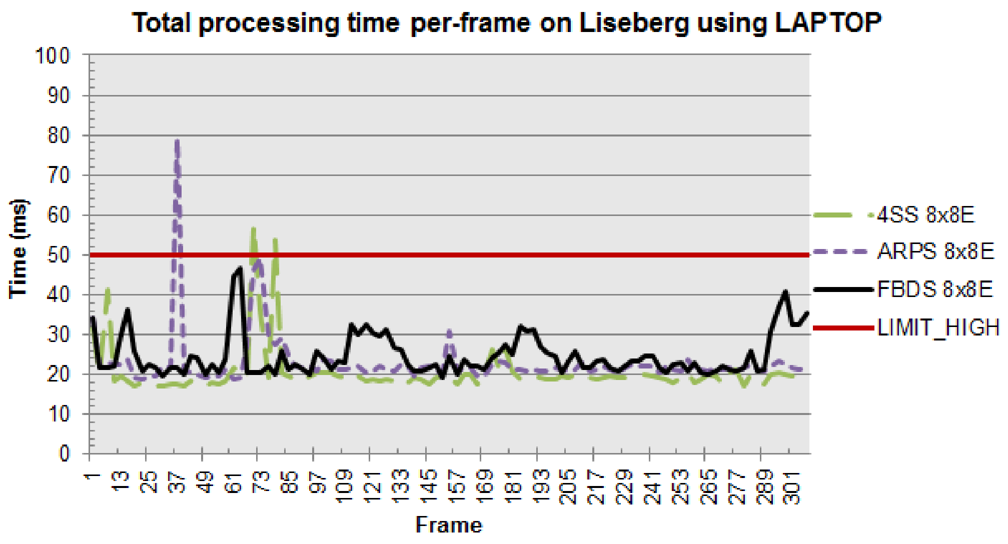


Figure 9.54: Total processing time (ME and MC) on *Liseberg* using LAPTOP after up-converting it from 10 to 30 FPS. The MC method used is BDMC. The blocksize used is 8x8 extended.

10. Conclusions

In this thesis we have successfully implemented a frame rate up-conversion tool (FRUCT). We have compared and evaluated the performance and quality of some block matching algorithms (BMAs) for their use in frame rate up-conversion of remote surveillance video. The unidirectional algorithms Full Search (FS), Four Step Search (4SS) and Adaptive Rood Pattern Search (ARPS) have been tested, as well as the Full Bidirectional Search (FBDS) algorithm. The latter is presented in this thesis as our version of an exhaustive bidirectional search algorithm, taking ideas from ARPS for early termination of search on static areas. We have also explored how Open Computing Language (OpenCL) can be used to improve performance and quality of motion compensation.

Focus has been set on 1080p video filmed with a static camera. Lack of existing test videos fitting this criteria forced us to film three videos of our own: *Liseberg*, *FerryClose* and *FerryFar*. Sample frames from these videos can be seen in Figures 9.2, 9.3 and 9.4. The *Liseberg* video features moderate horizontal and vertical motion with a mainly static background. *FerryClose* feature more motion compared to *FerryFar*, both having a third or more of the frame covered by water waves. All three videos include challenges for frame rate up-conversion.

Testing on the recorded videos was performed by up-converting previously down-converting version of the video by a factor of three. The block sizes used were 8x8 and 4x4 pixels, enlarged to 16x16 during block matching to give better accuracy. motion estimation (ME) was performed on the GPU, using an OpenCL implementation of bidirectional motion compensation (BDMC). A reference implementation of BDMC on the CPU showed to be slower compared to a high-end desktop GPU running the OpenCL version. The GPU implementation also provided higher video quality due to bilinear interpolation.

Results from up-conversion on the recorded videos show FBDS giving the best quality in two of the three videos, even though it has trouble with small or thin objects moving fast (e.g. birds). An up-conversion factor of three on *Liseberg* show disappointing results for all algorithms. However, reducing the factor to two increases quality significantly, with FBDS giving the best result. On *FerryClose* FBDS again gives the best quality, with 4SS at second place. The case is the opposite for *FerryFar*.

Processing time limits the block size to 8x8 on all three videos if FBDS is used. The more powerful GPU found in the desktop test system is then also required to keep times below the 50 ms limit (see Section 2.1.2).

Testing performed on the *Akiyo*, *Container* and *Foreman* test videos in the lower Common Intermediate Format (CIF) resolution shows varying results. 4SS, ARPS and FBDS all perform well on *Akiyo* and *Container*. Processing times are low enough for "true real-time" (16.7 ms) for most combinations of algorithms and block sizes.

Video motion in the intended use scenario does not generally cover as much of the frame as in *FerryClose* and *FerryFar*, where FBDS is close to breaking the limit. The motion in *Liseberg* is a better representation, where FBDS is well below the limit.

Despite the exhaustive search pattern of FBDS it proved to have only roughly twice the processing time compared to 4SS or ARPS. The zero motion pre-judgement turned out to be a great technique for reducing the workload for FBDS. If a sufficiently powerful CPU and GPU is used it may be viable for use in a remote surveillance system.

10.1 Future work

There are many possibilities to extend the work that has been done in this thesis. Wider use of SIMD instructions on the CPU can be used to optimize motion estimation. For example, there are multiple versions of the SSE instruction used in this thesis for SAD. The improvement of using 128-bit instead of 64-bit registers could speed up SAD for a block by a factor of two.

All areas of the image are generally not of interest, or equal interest, for the human observer. This means that computationally heavy parts of the processing could potentially be concentrated on smaller regions of interest. These salient areas may be algorithmically detected (see e.g. Jacobson et al. 2010), or manually defined.

FBDS is a very simple algorithm. A logical step in going forward would be to try out the unidirectional algorithms search patterns in a bidirectional setting. With the real-time goal secured, processor cycles can be spent on further processing stages. For example, no post-processing on the motion vectors is done in the FRUCT. One possible addition would be removal of motion vector outliers, as done in median vector filtering (see e.g. Zhai et al. 2005, Choi et al. 2006). The fully parallelizable block matching algorithms used for motion estimation can make use of the growing number of cores in CPUs (though memory bandwidth may be a limiting factor).

Finally, OpenCL and the steadily-increasing performance of GPUs have great future potential. Since the overhead of transferring frame data to the GPU has already been "paid", the cost of executing additional kernels is less. If the FRUCT is used on a real-time stream to be displayed directly, OpenCL-generated interpolated frames can be moved to the framebuffer internally on the GPU, instead of transferring back to the CPU. Workload can be divided between multiple devices, a possiblity not utilized by us. The kernel used for motion estimation can be improved further, and there is much room for e.g. testing out better work group size settings. OpenCL is not limited to GPUs only. Code may be written once and in the future end up running on a multitude of devices for which there is an OpenCL implementation.

Bibliography

Ashikhmin, M. (2001), Synthesizing natural textures, *in* 'Proceedings of the 2001 symposium on Interactive 3D graphics', I3D '01, ACM, New York, NY, USA, pp. 217–226.

Barjatya, A. (2004), Block matching algorithms for motion estimation, Technical report, ECE department at Utah State University.

Bhat, A., Richardson, I. & Kannangara, S. (2009), A novel perceptual quality metric for video compression, *in* 'Picture Coding Symposium, 2009. PCS 2009', pp. 1–4.

Bovik, A. (2009), *The Essential Guide to Video Processing*, second edn, Elsevier Academic Press, Burlington, Massachusetts, United States of America.

Cetin, M. & Hamzaoglu, I. (2010), An adaptive true motion estimation algorithm for frame rate conversion of high definition video, *in* 'Pattern Recognition (ICPR), 2010 20th International Conference on', pp. 4109–4112.

Chen, Y.-K., Vetro, A., Sun, H. & Kung, S. (1998), Frame-rate up-conversion using transmitted true motion vectors, *in* 'Multimedia Signal Processing, 1998 IEEE Second Workshop on', pp. 622 –627.

Choi, B.-D., Han, J.-W., Kim, C.-S. & Ko, S.-J. (2006), 'Frame rate up-conversion using perspective transform', *Consumer Electronics, IEEE Transactions on* **52**(3), 975–982.

Criminisi, A., Perez, P. & Toyama, K. (2004), 'Region filling and object removal by exemplar-based image inpainting', *Image Processing, IEEE Transactions on* **13**(9), 1200–1212.

Gan, Z., Qi, L. & Zhu, X. (2007), 'Motion compensated frame interpolation based on h.264 decoder', *Electronics Letters* **43**(2), 96–98.

Hong, W. (2009), Low-complexity occlusion handling for motion-compensated frame rate up-conversion, *in* 'Consumer Electronics, 2009.

ICCE '09. Digest of Technical Papers International Conference on', pp. 1 – 2.

Huang, A.-M. & Nguyen, T. (2008), 'Motion vector processing using bidirectional frame difference in motion compensated frame interpolation', *A World of Wireless, Mobile and Multimedia Networks, International Symposium on* **0**, 1–6.

Huynh-Thu, Q., Brotherton, M., Hands, D., Brunnström, K. & Ghanbari, M. (2007), Examination of the SAMVIQ methodology for the subjective assessment of multimedia quality, *in* 'Third International Workshop on Video Processing for Consumer Electronics'.

Intel Corporation (2011), *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z*, Intel Corporation.

ITU Radiocommunication Sector (2002), Recommendation ITU-R BT.500-11 - methodology for the subjective assessment of the quality of television pictures, Technical report.

Jacobson, N., Lee, Y.-L., Mahadevan, V., Vasconcelos, N. & Nguyen, T. (2010), 'A Novel Approach to FRUC Using Discriminant Saliency and Frame Segmentation', *Image Processing, IEEE Transactions on* **19**(11), 2924 –2934.

Kang, S.-J., Yoo, D.-G., Lee, S.-K. & Kim, Y. (2008), 'Multiframe-based bilateral motion estimation with emphasis on stationary caption processing for frame rate up-conversion', *Consumer Electronics, IEEE Transactions on* **54**(4), 1830 – 1838.

Khronos OpenCL Working Group (2008), *The OpenCL Specification, version 1.0.29*.
URL: *http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf*

Koga, T., Iinuma, K., Hirano, A., Iijima, Y. & Ishiguro, T. (1981), Motion compensated interframe coding for video conferencing, *in* 'Proceedings National Telecommunications Conference', pp. G5.3.1 – 5.3.5.

Kozamernik, F., Sunna, P., Wyckens, E. & Steinmann, V. (2005), 'Samviq - a new EBU methodology for video quality evaluations in multimedia', *SMPTE Motion Imaging Journal* pp. 152–160.

Lee, S.-H., Hur, B.-S., Kim, S.-H. & Park, R.-H. (2003), Weighted-adaptive motion-compensated frame rate up-conversion, *in* 'Consumer Electronics, 2003. ICCE. 2003 IEEE International Conference on', pp. 342 – 343.

Lee, Y.-L. & Nguyen, T. (2009), Method and architecture design for motion compensated frame interpolation in high-definition video processing, *in* 'Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on', pp. 1633 – 1636.

Lee, Y.-L. & Nguyen, T. (2010), High frame rate motion compensated frame interpolation in high-definition video processing, *in* 'Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on', pp. 858 –861.

Li, R., Zeng, B. & Liou, M. (1994), 'A new three-step search algorithm for block motion estimation', *Circuits and Systems for Video Technology, IEEE Transactions on* **4**(4), 438 –442.

Luessi, M. & Katsaggelos, A. (2009), Efficient motion compensated frame rate upconversion using multiple interpolations and median filtering, *in* 'Image Processing (ICIP), 2009 16th IEEE International Conference on', pp. 373 –376.

Marzat, J. Dumortier, Y. & Ducrot, A. (2009), Real-time dense and accurate parallel optical flow using cuda, *in* 'Proceedings of The 17th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision', pp. 105–111.

Nie, Y. & Ma, K.-K. (2002), 'Adaptive rood pattern search for fast block-matching motion estimation', *Image Processing, IEEE Transactions on* **11**(12), 1442 – 1449.

Oelbaum, T., Diepold, K. & Zia, W. (2007), A generic method to increase the prediction accuracy of visual quality metrics, *in* 'Picture Coding Symposium (PCS)'.

Po, L.-M. & Ma, W.-C. (1996), 'A novel four-step search algorithm for fast block motion estimation', *Circuits and Systems for Video Technology, IEEE Transactions on* **6**(3), 313 –317.

Richardson, I. E. (2010), *The H.264 Advanced Video Compression Standard*, second edn, John Wiley & Sons, Ltd., Chichester, West Sussex, United Kingdom.

Sasai, H., Kondo, S. & Kadono, S. (2004), Frame-rate up-conversion using reliable analysis of transmitted motion information, *in* 'Acoustics, Speech, and Signal Processing, 2004. Proceedings. (ICASSP '04). IEEE International Conference on', Vol. 5, pp. V – 257–60 vol.5.

Suthaharan, S., Kim, S.-W. & Rao, K. (2005), 'A new quality metric based on just-noticeable difference, perceptual regions, edge extraction and human vision', *Electrical and Computer Engineering, Canadian Journal of* **30**(2), 81 –88.

Tasdizen, O. & Hamzaoglu, I. (2010), Computation reduction techniques for vector median filtering and their hardware implementation, *in* 'Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on', pp. 731 –736.

Thaipanich, T., Wu, P.-H. & Kuo, C.-C. (2009), 'Low complexity algorithm for robust video frame rate up-conversion (fruc) technique', *Consumer Electronics, IEEE Transactions on* **55**(1), 220 –228.

Tominaga, T., Hayashi, T., Okamoto, J. & Takahashi, A. (2010), Performance comparisons of subjective quality assessment methods for mobile video, *in* 'Quality of Multimedia Experience (QoMEX), 2010 Second International Workshop on', pp. 82 –87.

Wang, Z. & Bovik, A. (2002), 'A universal image quality index', *Signal Processing Letters, IEEE* **9**(3), 81–84.

Wang, Z. & Bovik, A. (2009), 'Mean squared error: Love it or leave it? a new look at signal fidelity measures', *Signal Processing Magazine, IEEE* **26**(1), 98–117.

Wang, Z., Bovik, A., Sheikh, H. & Simoncelli, E. (2004), 'Image quality assessment: from error visibility to structural similarity', *Image Processing, IEEE Transactions on* **13**(4), 600 –612.

Wang, Z., Sheikh, H. & Bovik, A. (2002), No-reference perceptual quality assessment of jpeg compressed images, *in* 'Image Processing. 2002. Proceedings. 2002 International Conference on', Vol. 1, pp. I–477 – I–480 vol.1.

Wexler, Y., Shechtman, E. & Irani, M. (2004), Space-time video completion, *in* 'Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on', Vol. 1, pp. I–120 – I–127 Vol.1.

Woodard, J. & Carley-Spencer, M. (2006), 'No-reference image quality metrics for structural mri', *Neuroinformatics* **4**, 243–262. 10.1385/NI:4:3:243.

Zhai, J., Yu, K., Li, J. & Li, S. (2005), A low complexity motion compensated frame interpolation method, *in* 'Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on', pp. 4927 – 4930 Vol. 5.

Zhu, S. & Ma, K.-K. (1997), A new diamond search algorithm for fast block matching motion estimation, *in* ‘Information, Communications and Signal Processing, 1997. ICICS., Proceedings of 1997 International Conference on’, Vol. 1, pp. 292–296.

Glossary

1080p High-definition resolution - 1920 x 1080 pixels. 3, 5–7, 12, 44–46, 56, 65, 66, 70, 71, 74, 79

720p High-definition resolution - 1280 x 720 pixels. 12

bitstream A series of bits. 6, 10, 11, 25, 39

codec Short for encoder and decoder (enCOder-DECoder). 39

decoder Decodes encoded data back to the original format.. 5, 6, 10, 25

down-conversion Decrease the frame rate of a video. 33, 34, 71, 79

encoder Encodes (converts) data from one format to another. E.g. used to compress data.). 5, 6, 10, 25

histogram A graphical representation of data distribution. 10

JPEG image A lossy compression format for images. 33

luma The Y channel of Y'CbCr 4:2:0 containing light information. 34, 35, 39

motion vector Represents the motion between two frames for a block. 6, 10–12, 25–28, 39–41, 81

up-conversion Increase the frame rate of a video. 1, 3, 4, 6–10, 25, 33, 34, 37–39, 43, 47–65, 67–69, 71–79

Y'CbCr 4:2:0 A type of chroma subsampling (less chroma information than luma information). 6, 44

Acronyms

4SS Four Step Search. 13, 16–19, 21, 22, 39, 47–56, 58, 65, 66, 70, 73, 76, 79, 80

ARPS Adaptive Rood Pattern Search. 2, 12, 19–22, 39, 41, 47, 48, 50–55, 65, 73, 76, 79, 80

BDMC Bidirectional Motion Compensation. 21, 27, 28, 41, 71–79

BMA Block Matching Algorithm. 1, 2, 10–13, 21, 39, 40, 79

CIF Common Intermediate Format, 352 x 288 pixel resolution. 12, 43, 66, 71, 80

DMC Direct Motion Compensation. 26, 27, 41

DS Diamond Search. 18, 19, 21

DSCQS Double-Stimulus Continuous Quality-Scale. 36, 37

DSIS Double-Stimulus Impairment Scale. 36, 37

DSP Digital Signal Processing. 29

FA Frame Averaging. 8, 38, 39, 41, 48, 49, 53–55, 61, 62, 65

FBDS Full Bidirectional Search. 2, 22, 23, 39, 47–55, 59, 63, 65, 74, 76, 79, 80

FPGA Field-Programmable Gate Array. 7

FPS Frames Per Second. 3, 4, 7, 33, 44, 47–54, 56–65, 67–69, 71–78

FR Frame Repetition. 8, 9, 38, 39

FRUCT Frame Rate Up-Conversion Tool, see up-conversion. 1, 3–5, 33, 38–41, 79, 81

FS Full Search. 12, 13, 39, 41, 47, 79

GPU Graphics Processing Unit. 30, 39

H.264 H.264/MPEG-4 AVC. 6, 10, 39

HD High-Definition, see 1080p and 720p. 1, 4, 5

HT Hyper-Threading. 47

ITU-R ITU Radiocommunication Sector. 36

JND Just Noticeable Difference. 35

MAD Mean Absolute Difference. 24

MAE Mean Absolute Error. 24

MC Motion Compensation. 9, 12, 26, 33, 38, 39, 41, 42, 48, 70–78

ME Motion Estimation. 5, 9–11, 25, 26, 33, 39–41, 48, 50, 65–79

MOS Mean Opinion Score. 36

MOSp Predicted Mean Opinion Score. 35

MSE Mean Squared Error. 24, 25, 34

MVF Motion Vector Field, see motion vector. 10

NTSS New Three Step Search. 13–16

OBMC Overlapped Block Motion Compensation. 10

OpenCL Open Computing Language. 1, 2, 29–31, 38, 39, 41, 79, 81

OpenMP Open Multi-Processing. 47, 65

PSNR Peak Signal-to-Noise Ratio. 25, 34, 35, 37, 55–64

SAD Sum of Absolute Differences. 23, 24, 40, 65, 66

SAE Sum of Absolute Errors. 24

SAMVIQ Subjective Assessment Method for Video Quality. 37

SSD Sum of Squared Differences. 24, 25

SSE Streaming SIMD Extensions, see Single Instruction, Multiple Data (SIMD). 24, 40

SSIM Structural Similarity Index, see (Wang et al. 2004). 35, 37, 56, 62, 63, 65

TSS Three Step Search. 13–17