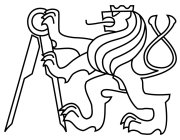
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF MASTER'S THESIS

Title:	Experiments with standard HEVC for video compression
Student:	Bc. David Němec
Supervisor:	Ing. Ivan Šimeček, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Systems and Networks
Department:	Department of Computer Systems
Validity:	Until the end of summer semester 2016/17

Instructions

Describe video compression standard HEVC. Compare different implementations of HEVC with alternative video compression standards, e.g., H264 and VP9. Focus on encoding speed, compression ratio, and quality of exported video based on different encoder settings.
Suggest and implement GPU accelerated quality evaluation algorithms of exported videos using CUDA and OpenCL. Measure efficiency of both technologies on multiple GPU cards. Analyze measured data and discuss final results.

References

Will be provided by the supervisor.

L.S.

prof. Ing. Róbert Lórencz, CSc.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague December 2, 2015

Czech Technical University in Prague

Faculty of Information Technology

Department of Computer Systems and Networks

Master's thesis

Experiments with standard HEVC for video compression

Bc. David Němec

Supervisor: Ing. Ivan Šimeček, Ph.D.

16th February 2017

Acknowledgements

This research was supported by CTU Faculty of Information Technology and NVIDIA, which provided necessary GPU required for testing part of this thesis. And most importantly many thanks to my supervisor Ing. Ivan Šimeček, Ph.D. for guidance.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In V Praze on 16th February 2017 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Němec, David. *Experiments with standard HEVC for video compression.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstrakt

Cílem této diplomové práce je analyzovat nejnovější HEVC video kompresní formát a jeho porovnání a alternativami jako jsou H.264 a VP9. Část práce je zaměřena na programování aplikace vyhodnocující kvalitu videa, která je akcelerována na GPU za použití technologií CUDA a OpenCL. V první části práce jsou diskutovány hlavní vlastnosti HEVC. Tato část slouží ke správnému porozumění slabých a silných stránek HEVC. Následně jsou identifikovány způsoby využití GPU pro akceleraci porovnání videa. Aplikace pro vyhodnocování kvality používá knihovnu FFmpeg pro enkódování/dekódování videa. Poslední část je zaměřena na testování na více GPU, založené na datech z předchozí analýzy. Závěrem, je diskutována efektivita použitých technologií pro akceleraci vyhodnocující kvality videa.

Klíčová slova HEVC, H.265, x265, VP9, H.264, x264, NVENC, CUDA, OpenCL, SSIM, PSNR Kvalita videa

Abstract

The aim of this thesis is to investigate and analyze newest HEVC video compression standard and compare it with alternatives like H.264 and VP9. Major part of thesis is focused on programming video quality evaluation program, which is accelerated on GPU using CUDA and OpenCL. In the first part, we discuss main features of HEVC. This part leads to correct understanding of weaknesses and strengths of HEVC. The thesis then identifies best way of usage GPU in video comparison. Programmed quality evaluation application uses FFMPEG library for video encoding/decoding. Last part of thesis is focused on testing on multiple GPUs based on collected data from previous analysis. In conclusion, the thesis argues about efficiency of used technologies in quality evaluation task.

Keywords HEVC, H.265, x265, VP9, H.264, x264, NVENC, CUDA, OpenCL, SSIM, PSNR, Video Quality

Contents

List of Figures

List of Tables

Introduction

Daily requirements for saving, sharing or streaming video have been rising exponentially. End users are requiring better and better quality from video, which leads to higher and higher storage requirements. Storing video without any compression is a problem. Even today when storages in terabytes are common. Usage of lossless compression algorithms does not provide required compression ratio and even does not make sense since it is possible to achieve similar video quality without user noticing difference.

Currently most common compression standard for video streams is called H.264 and it has been a huge success. It's scaled remarkably well since it was first proposed and is capable of handling 3D, 60fps and even high resolution.

But we are starting to get to tipping point of high display resolutions and higher range of colors. The point at which a series of small changes like high resolution monitors, virtual reality and better screen panel technology becomes significant enough to cause a larger, more important change. All these new technologies are capable to display video stream in higher quality then ever before. That creates enormous requirements for video compression standards. One of the main reasons for creating HEVC standard has been effort to increase compression ratio.

Today it is required to store videos in both higher resolution and higher quality which require more storage. As the cost of processing power have been reduced it was possible to create video codec that can reduce storage requirements for the cost of higher processing power.

First version of HEVC was created in 2013 to utilize more processing power and address this issue. HEVC was developed by JCT-VC consisted of companies ISO/IEC Moving Picture Experts Group (MPEG) and ITU-T Video Coding Experts Group (VCEG).

Sadly, HEVC is loaded with a lot of patents. For that reason, commercial use of HEVC require paying licensing fees to companies such as MPEG LA, HEVC Advance, and Technicolor SA. This is main reason why adoption of HEVC is considered to be very slow and many similar implementations exits. Predecessor H264 also requires licensing but licensing fees are considerably lower.

As a response to this situation companies Amazon, Cisco, Google, Intel Corporation, Microsoft, Mozilla, and Netflix founded non-profit organization Alliance for Open Media (AOMedia). Latter companies AMD, ARM, Adobe, Ateme, Ittiam and Vidyo joint this organization. First project is to develop free alternative to HEVC. First version of this codec will be called AV1 a should be finished by March 2017. It is expected that after release AV1 will have higher market penetration even though HEVC has an advantage because it is part of ISO standard.

Main focus of this thesis is to analyze algorithms for video comparison and possibility of their parallel execution on GPUs. Output of this thesis is also program that implements video comparison algorithms with utilization of GPU when possible.

This thesis is separated into four parts. First part of this thesis contains terminology and methods of video comparison. Second part compares available technologies enabling parallel acceleration on GPUs.

Third part contains description of application architecture and specific methods used to achieve highest possible performance.

Finally, last part is dedicated to set objective method for video comparison and test available codecs.

Chapter 1

Video codec theory

This chapter is dedicated to describe how video streams are stored on disc, which processes needs to be done to use store video files and why it is necessary to have multiple options to store video stream in multiple ways.

Network speeds continue to increase rapidly, high bitrate connection are now very common and storage capacities are getting bigger and bigger. It might not seem obvious why it is needed to compress videos. There are two main reasons. First of all, data requirements necessary to store raw data are still too big and even if would manage to store/transfer all data most of users would not see sufficient increase (or none at all) in video quality.

Video coding format is representation format for transmitting or storing digital video content. Specific video coding format is called video codec. Codec is shortcut for coder and decoder. Video coding is process of compressing and decompressing data from/to encoded stream to digital video signal.

Currently lossless codecs provide only small decrease in video size. With exceptions like cases where most of video scene is static, etc... Lossless codecs achieve data compression by removing redundancy in the data. Most of codecs use lossy compression, which achieves greater compression with price of distortions in encoded video.

Video is representation of visual scene using slideshow of images. Frames in video stream usually change with 1/24 or 1/30 second intervals (24/30 frames per second).

Most common examples of video coding formats are following:

- **MPEG-4 Part 2** was ancestor to H.262/MPEG-2 Part 2 based on discrete cosine transform compression standard with support of interlaced video.
- **H.263** Predecessor of H.264. low bitrate standard designed for video conferencing. Technically uses same core standard MPEG-4 Part 2.
- **H.264** Also called MPEG-4 Part 10 or Advanced Video Coding (MPEG-4 AVC). H.264 is block-oriented motion-compensation-based standard. Currently it is most used codec for video compression. x264 is video encoder application library available under the terms of the open source GNU GPL 2 license.
- **H.265** Also called HEVC High Efficiency Video Codec successor of H264 is considered to be first of next generation codecs. Patented and paid for commercial projects. x265 is video encoder application library available under the terms of the open source GNU GPL 2 license.
- **Google codecs (VPX)** VP8, VP9 are proprietary video compression formats created by On2 Technologies. Google also released libvpx library under BSD license. VP9 is direct concurrence to HEVC. It is mostly used on YouTube. Most of Internet Browsers are capable of playing VP9 encoded video (notable exceptions are Internet Explorer and Safari).
- **AV1** Direct concurrence of HEVC and successor to VP9. It is being developed by Alliance for Open Media (AOMedia). Designed for real-time applications. Current efficiency is similar to HEVC but targeted performance should be 50% above HEVC.

1.1 Picture coding

Digital picture is usually two-dimension image. Dimension sizes are defined by picture resolution. Resolution can be dynamic in case of vector pictures or static for bitmap pictures. Vector graphics use polygons to represent images while bitmaps are represented as grid of pixels (colors). With a few exceptions (like Flash) pictures in videos are coded as bitmaps.

Single picture (also called frame) in video can be represented by multiple formats [10] . Eligible formats are following:

- Luma (Y) only (monochrome).
- Luma and two chroma (YCbCr or YCgCo).
- Green, Blue and Red (GBR, also known as RGB)

Single component of pixel coding will be referred to as channel. Today every single channel is mostly 8bit. Monitors (mostly with IPS matrix) are starting to support 10bit colors, but today only professional graphic cards (Nvidia Tesla) support video output in more than 8bit per color. 10bit bits per channel are supported by Nvidia only in exclusive full screen DirectX mode.

HEVC is first modern format which officially supports more than 8bit per channel by design.

1.1.1 YCbCr

YCbCr is also sometimes referred as YUV, which is analog encoding of color information in television systems, but in computer science it refers to YCbCr [11].

- Luma (Y) is channel that determines brightness of color
- Chrominance (U and V) are channels that determine color itself.

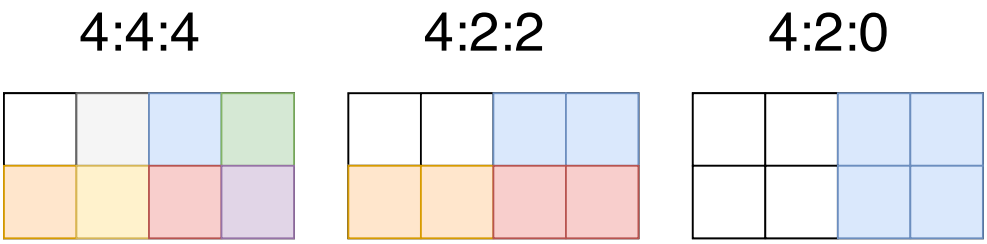


Figure 1.1: Chroma subsampling

Representation

- In monochrome sampling there is only one sample array, which is nominally considered the luma array.
- In 4:2:0 sampling, each of the two chroma arrays have half the height and half the width of the luma array.
- In 4:2:2 sampling, each of the two chroma arrays have the same height and half the width of the luma array.
- In 4:4:4 sampling, each of the two chroma arrays have the same height and same width of the luma array.

Bandwidth required to store one sample (relative to 4:4:4) is sum of all factors divided by twelve. Every representation except 4:4:4 uses chroma subsampling to lower bandwidth.

When chroma subsampling is used it implementation might differ in implementation. Most common is usage of top left value as can be seen in figure 1.1. One of implementation could also be average of values.

Chroma subsampling can create holloing artifact around sharp edges. That can be problem for keying (technique used to remove video background) with blue or green screening. Second problem can be out-of-gamut colors (negative value in channel).

1.1.2 RGB

RGB is additive color model in which red, green, and blue light are added together in various ways to reproduce a broad array of colors [11]. This color model is used by monitors to produce specific colors by light mixing as can be seen in figure 1.2.

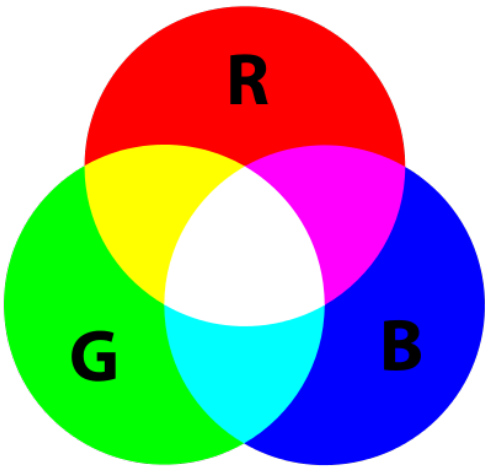


Figure 1.2: RGB Additive Mixing

Representation
RGB is usually represented by single array containing information of all colors in following order RGB—RGB—RGB (figure 1.3). Pixels are ordered from top left pixel to right bottom by rows.

1.1.3 Interlaced frames

Video frames can be encoded in progressive or interlaced modes [1]. In progressive video all frames are have same resolution of video sequence.

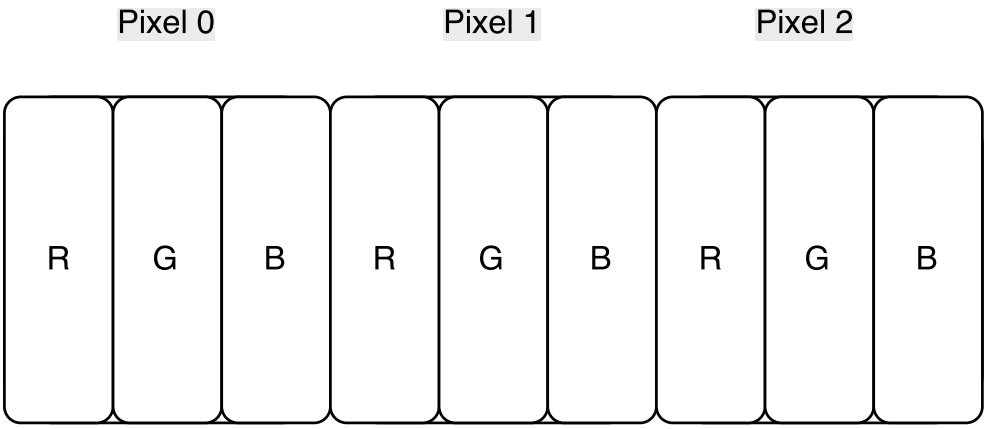


Figure 1.3: RGB representation

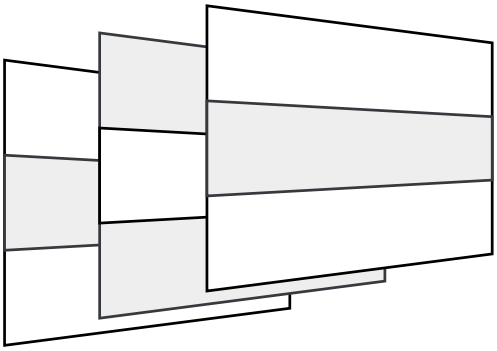


Figure 1.4: Interlaced frame switching

Idea with interlaced frames is to double perceived frame rate by user with similar bandwidth. All frames (called in this case half frames) have half height of original video sequence. These half frames define odd or even rows in frame and are alternately switched (figure 1.4). So every odd frame contains odd row and every even frame contains even row. That means every time user sees two frames at the same time.

Interlaced video requires display capable of showing individual fields in sequential order. Ideal displays are CRT, where afterglow from frame display naturally joints multiple frames into one.

Sadly, with today displays which are mostly based on LCD technology, Interlaced video needs to be Deinterlaced. Otherwise user would notice that frames are being combined (figure 1.5). This effect can be seen with Scene Pan or with fast moving objects.

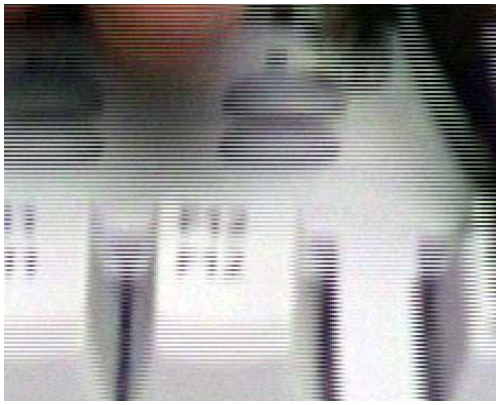


Figure 1.5: Interlaced frame without deinterlacing [1]

Deinterlacing can be done in several ways [1]:

- **Bob** - Doubles framerate, frames are displayed, one after the other
- **Blending** - Blends frames into each other
- **Weaving** - Consecutive fields are added together (basically leaves video as it is)
- **Smart (Adaptive)** - detects rate of movement and combines Bob and Weave for best effect. Also called Progressive scan.

1.1.4 Frame types

In video compression several types of frames are used. Frames are separated into two categories Key frames and Predicted frames. Keyframes are frames which can be decoded independently on other frames. Predicted frames are frames which needs other frames to be decoded.

- **I-Frame** - Independent frame which can be decoded independently on other frames. I-Frame holds information of entire frame.
- **P-Frame** - Predicted frame decoding is dependent on previous frame/s. Picture holds information only for changes relatively to previous frame.
- **B-Frame** - Bi-predicted frame decoding is dependent on previous and future frame/s.

1.1.5 Macroblocks

Macroblock is block of 16x16, 8x8 or 4x4 region which contains residual frame samples after prediction. With chroma subsampling formats macroblock are

scaled accordingly. For example chroma 4:2:0 (16 x 16 luma samples, 8 x 8 Cb and 8 x 8 Cr samples)

1.2 Licenses

Specific codec implementations can use different license, that might limit its usage. Or be used under certain conditions. Common licenses used for codecs does not limit end user in any way, but in many cases limit developer.

GNU General Public License or GPL is the most popular software licence used for many projects including Linux kernel [12]. This licence grants user to use, change and redistribute the software, but changes also must be available under same licence.

The GNU Lesser General Public License is very similar to GNU General Public License [12]. But it is more oriented to software libraries. Main reason is to allow non-GPL application to link and use these libraries. Any changes in libraries still must be available under same license.

The BSD License is more liberal compared to GNU [13]. License was originally used in BSD operating system. BSD license lets user to freely use source code. Source code is provided in 'as-is' state.

1.3 MPEG-4 Part 2

MPEG-4 Part 2 is older standard developed by MPEG . Latest standard to this compression format was released in 2004 defined by ISO/IEC 14496-2:2004. Most known codecs which implemented this standard are DivX and Xvid. Currently these codes are obsolete. [14]

It is block based standard that uses motion compensation followed by DCT (Discrete Cosine Transform is lossy compression for audio and image e.g. JPEG) and Quantization (lossy compression technique used for image processing). Basically same process is in standard of H.263.

1.4 H.264

H.264 also called MPEG-4 Part 10 or Advanced Video Coding (MPEG-4 AVC). It is currently dominating video codec. First version was completed in year 2003. Over following years Video Coding Experts Group released a lot of new versions with new features. Currently latest version is v22 released in February 2014, which enhanced support for 3D video. [15]

Instead of I-Frames, P-Frames and B-Frames H.264 uses I-Slices, P-Slices and B-Slices. Slices are regions of frame that are encoded separately. One frame contains one or more slices. Each slice contains one or more macroblock.

Table 1.1: H.264 codec list

Project	**License**	**Encoder**	**Decoder**	**Language**
x264	GPL 2	✓		C
OpenH264	BSD	✓	✓	C++
FFmpeg	LGPL 2.1 or later		✓	C

H.264 contains many techniques to improve efficiently over older codecs. Some of the most important features are:

- **Inter prediction process** Encoder can use up to 16 frames (even future frames) as reference to encode frame.
- **Motion compensation** - algorithmic technique used to predict a frame in a video. In H.264 block motion compensation (BMC) is used. Frames are partitioned into macroblocks and based on previous and future frames shift and difference is calculated. Shift is represented by motion vector. Motion vector can be represented using non-integer numbers in case shift would not be decimal (sub-pixel precision) and interpolate pixels. One macroblock can be splitted into multiple block called partitions.
- **Chroma subsampling** - Support of monochrome (4:0:0), 4:2:0, 4:2:2, and 4:4:4
- **Independent channels** - Channels can be encoded independently. That leads to easy parallelization of encoding (only supported by 444 profile).
- **Lossless macroblock coding**
- **Higher bit depth** - Support for depth precisions ranging from 8 to 14 bits per sample of channel.

H.264 is accepted in following standards:

- H.264 : Advanced video coding for generic audiovisual services. Currently in version ITU-T H.264 (10/2016) [15]
- ISO/IEC JTC 1/SC 29/WG 11 Motion Picture Experts Group (MPEG) – publishes the H.264 standard as ISO/IEC 14496-10:2012. [16].

1.5 H.265

High Efficiency Video Coding (HEVC), also known as H.265, is a new video compression standard, developed by the Joint Collaborative Team on Video Coding (JCT-VC). [10]

H.265 is evolution of existing video codec H.264 and is being developed in response to the growing need for higher compression of moving pictures for various applications such as Internet streaming, communication, videoconferencing, digital storage media and television broadcasting. It is also designed to enable the use of the coded video representation in a flexible manner for a wide variety of network environments.

HEVC is mostly extension of H.264 as it uses same techniques. Block sizes were increased from maximum of 16x16 to 64x64 with improved segmentation. Motion vector prediction, motion compensation and intra-prediction were improved and new step called sample-adaptive offset filtering was added. Technologies that are obsolete were removed, for example interlacing are no longer supported.

Table 1.2: HEVC codec list [9]

Project	**License**	**Encoder**	**Decoder**	**Language**
cclxv	GPL 2	✓	✓	C++11
f265	BSD	✓		C
FFmpeg	LGPL 2.1 or later		✓	C
HM	BSD	✓	✓	C++
HomerHEVC	LGPL 2.1 or later	✓		C
kvazaar	GPL 2	✓		C
libav	LGPL 2.1 or later		✓	C
libde265	LGPL 3 or later		✓	C++
openHEVC	LGPL 2.1 or later		✓	C
x265	GPL 2 or later	✓		C++

HEVC is accepted in following standards:

- ITU-T Study Group 16 – Video Coding Experts Group (VCEG) – publishes the H.265 standard as ITU-T H.265. Currently in version ITU-T H.265 (V3) (04/2015) [10]
- ISO/IEC JTC 1/SC 29/WG 11 Motion Picture Experts Group (MPEG) – publishes the HEVC standard as ISO/IEC 23008-2. Currently in version ISO/IEC 23008-2:2015. [17].

Sadly, HEVC is loaded with a lot of patents. For that reason, penetration on market is very low and seems to increase with very low speed.

1.6 Video Quality Evaluation

Video quality is a characteristic of a video passed through a video transmission/processing system, a formal or informal measure of perceived video degradation (typically, compared to the original video). Video quality evaluation is performed to describe the quality of a set of video sequences. [18]

Video quality evaluation can be separated into two categories.

- Subjective video quality.
- Objective video quality

Subjective video quality procedures measurements are described in ITU-R Recommendation BT.500 and ITU-T recommendation P.910. Their main idea is that video sequences are shown to a group of viewers and then their opinion is recorded and averaged to evaluate the quality of each video sequence. However, the testing procedure may vary depending on what kind of system is tested.

Objective video quality is measured as ratio between information contained in original video and information in transcoded video. Objective video quality procedures measurements are divided into following categories:

Full Reference Methods (FR): Full Reference Methods compute the quality difference by comparing the original video sequence against the received video sequence. Usually, every pixel from the source is compared against the corresponding pixel at the received video. In case of different resolutions larger frame is downscaled to smaller frame. Full Reference Methods are the most accurate. [19]

- **Picture Quality Methods(PQ):** Picture Quality Methods that are used for video quality rating (such as PSNR or SSIM) are image quality models, whose rating is calculated for every frame of a video sequence. This quality measure of every frame can then be recorded over time to calculate the quality of an entire video sequence. Quality of entire video sequence is usually calculated as average of all evaluated frames. Some of additional calculated values can be minimum and maximum.

Reduced Reference Methods (RR): Reduced Reference Methods provide a solution that lies between FR and NR models. They are used when all the reference video is not available. They are designed to predict the perceptual quality of distorted images with only partial information about the reference images. Calculations are usually done by statistical modeling of the discrete cosine transform (DCT) coefficient distributions. [20]

No-Reference Methods (NR): No-Reference Methods compute the quality of a video sequence without any reference to the original video sequence. Interestingly, human observers can easily recognise quality of distorted image even without previout reference. But for computer algorithms it is very difficult task. [21]

- **Pixel-Based Methods (NR-P):** Pixel-based Methods use a decoded representation of the video sequence and analyze the quality after full image reconstruction. Quality evaluation is mostly possible only when the prior knowledge about the image distortion types is available. For example, capability to effectively predict DCT-based image quality requires knowledge of most significant artifacts generated during the JPEG compression process.
- **Bitstream/Parametric Methods (NR-B):** These Methods use statistics of various coding parameters from the transmission container and video data stream, like packet headers, motion vectors and quantization parameters.
- **Hybrid Methods (Hybrid NR-P-B):** Hybrid Methods use parameters extracted from the data stream and decoded video sequence. They are therefore a mix between NR-P and NR-B models.

1.6.1 PSNR

Peak Signal to Noise Ratio is most commonly used full reference algorithm [22]. PSNR is measured in dB (decibels) on a logarithmic scale. PSNR uses MSE (Mean Square Error) between original and comparing image. MSE is calculated by equation 1.1. Calculation of PSNR (equation 1.2) is quick and easy that's why it is still one of most favorite formulas to calculate image quality [23].

$$MSE = \frac{1}{mn} \sum_{y=0}^{height} \sum_{x=0}^{width} \left[I\left(x,y\right) - J\left(x,y\right) \right]^2 \tag{1.1}$$

Higher value means better quality. Scale ranges between zero and infinity. Infinity (or undefined) value occurs when pictures are identical and formula

contains dividing by zero (or logarithms of zero). Usually it is expected to have PSNR between 30 and 50 dB for static images when calculated only on luma channel.

$$PSNR_{dB} = 10 \cdot log_{10} \left(\frac{MAX^2}{MSE} \right) \tag{1.2}$$

$$PSNR_{dB} = 20 \cdot log_{10} (MAX) - 10 \cdot log_{10} (MSE) \tag{1.3}$$

$I(x, y)$ and $J(x, y)$ are sample on location x, y from original and comparing image. MAX is maximum value of one sample. Images usually use 8-bit colors where MAX value is then 255.

PSNR does necessarily correlate with subjective quality. PSNR that uses contrast perception is called PSNR-HVS-M.

1.6.2 SSIM

Structural Similarity Index (SSIM) as an image quality metric.[22] It calculates the visual impact of three characteristics of an image: luminance 1.4, contrast 1.5 and structure 1.6. The overall index is a multiplicative combination of these three terms 1.7.

$$l(x, y) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}, \tag{1.4}$$

$$c(x, y) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2}, \tag{1.5}$$

$$s(x, y) = \frac{\sigma_{xy} + C_3}{\sigma_x^2 + \sigma_y^2 + C_2}, \tag{1.6}$$

$$SSIM(x, y) = l(x, y)^\alpha \cdot c(x, y)^\beta \cdot s(x, y)^\gamma \cdot \tag{1.7}$$

where μ_x, μ_y, $\sigma x, \sigma y$, and σxy are the local means, standard deviations, and covariance for various windows images x, y. By default, $C1 = 0.01 * L, C2 = 0.03 * L, C3 = C2/2$. SSIM uses these regularization constants to avoid instability for image regions where the local mean or standard deviation is close to zero. L is dynamic range of pixel values, typically $2^{bits\ per\ pixel}$ (255 when calculation only with luma).

Alpha, beta, gamma are coeficients which have defaut value 1.

Usually this formula is applied only on luma (because human eye is most sensitive to luminance), but it can be also applied to luma with chrominances or RGB. Result of SSIM varies between -1 and 1. Where 1 represents identical images. Windows sizes for calculating means, deviations and covariance is usually 9x9 pixels.

SSIM is originally designed only for static images, but it can be also used for video sequences. Usually SSIM for video sequences is calculated as mean of all calculated frames.

Three-component SSIM is a variant of SSIM which suggests weighted average of luminance, contrast and structure. Some variants use weighting of $0.5, 0.25, 0.25$ or $1.0, 0.0, 0.0$ that completely ignores contrast and structure.

Multi-Scale SSIM (MS-SSIM) extends technique by making multiple SSIM evaluations on different image scales. This is performed by repeatedly downscaling images by factor of two. With MS-SSIM the scale of images is becoming less important. This technique can be used to compare images that have been upscaled or downscaled [24].

Arithmetic local mean (also called Average or Expected value) is calculated as sum of all samples divided by their count 1.8. Average is usually marked as $E(X)$ or μ.

$$E(X) = \sum_{x} x \cdot f(x) \tag{1.8}$$

Variance is mean of square deviations 1.9. It represents expected deviantion from local mean. Variance us usually marked as $Var(X)$ or μ^2 (variance is squared but sometimes it is not written).

$$Var(X) = E\left[(X - E[X])^2\right] \tag{1.9}$$

$$Var(X) = E\left[X^2\right] - (E[X])^2 \tag{1.10}$$

Covariance measures correlation between two sets of variables of same size. Covariance is calculated as mean value of multiplied deviantions of X and Y 1.11. For uncorrelated variates is covariance zero. Formula can be simplified to mean value of multiplied X and Y minus multiplied mean values of X and Y [25]. Covariance is marked as $cov(X, Y)$.

$$cov(X, Y) = E[(X - E[X])(Y - E[Y])] \tag{1.11}$$

$$cov\,(X,Y) = E\,[XY] - E\,[X]\,E\,[Y] \tag{1.12}$$

1.6.3 Video Quality Comparation

There are continuous discussions about optimality of video comparison algorithms. Most important is how is video quality perceived by human and that can be very subjective. It might be very hard to replicate any results even though few of techniques have been standardized as ITU-R Recommendation BT.500-13 [26] or ITU-T Recommendation P.910 [27].

Objective video quality are clearly way to go. Algorithms try to predict human judgment of picture quality. Algorithms often use metrics that can be easily evaluated by a program.

Many implementations of these algorithms are executed mostly on Luma part of video sequence as human eye is most sensitive to brightness and color information is not that important.

In case of evaluation of video quality in RGB human eye is most sensitive to blue color, but perception differences are not as high.

Many uses of this quality comparison require evaluation in real time. That in many cases limits algorithm complexity. In this thesis we will compare two well-known objective image quality metrics SSIM and PSNR, that are most common algorithms used for measuring the similarity between two images. During implementation we will focus on parallelization of these algorithms.

Chapter 2

GPU programming

This chapter is dedicated to describe available technologies for GPU programming, explaining difference between CPU/GPU and mention advantages/disadvantages with usage of GPU.

The first GPUs were designed as graphics accelerators, supporting only specific fixed-function pipelines. Starting in the late 1990s, the hardware became increasingly programmable, culminating in NVIDIA's first GPU in 1999. Less than a year after NVIDIA coined the term GPU, artists and game developers weren't the only ones doing ground-breaking work with the technology: Researchers were tapping its excellent floating point performance.

Thanks to the big data parallelism GPU evolved into specialized SIMD processors, but with many extensions that's why they are called SIMT (Single Instructions Multiple Threads). Every thread has own identity, for example own registers.

Because it is challenging to continue to increase frequency of processors, but it is easy to add more cores. Thanks to that we have two developmental lines.

1. multi-core (Several full cores on same chip). These line is currently used by CPUs. Optimized to perform sequential code. Adding of more cores is difficult because of 1D architecture. Memory access latency is being lowered by hierarchy of L1, L2 and L3 caches. Most logic on one chip dedicated to effective execution of instructions and data fetching.
2. many-core (Many simple cores on same chip). These line is currently used by GPUs. Optimized for parallel computing. Adding of more cores is easy thanks to 2D architecture. Memory access latency is being lowered by thread switching and small cache. Compared to CPU signifficantly lower space on chip is dedicated to control logic.

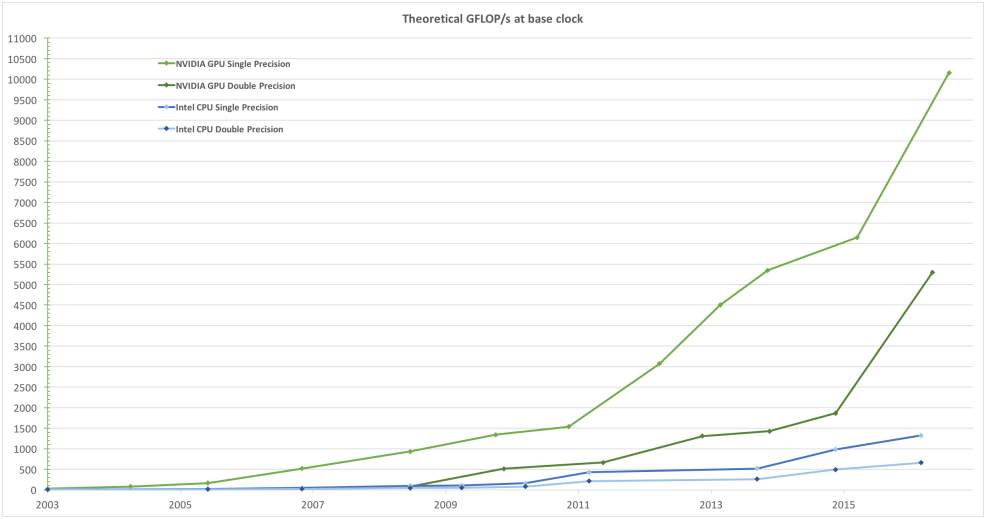


Figure 2.1: Floating-Point Operations per Second for the CPU and GPU [2]

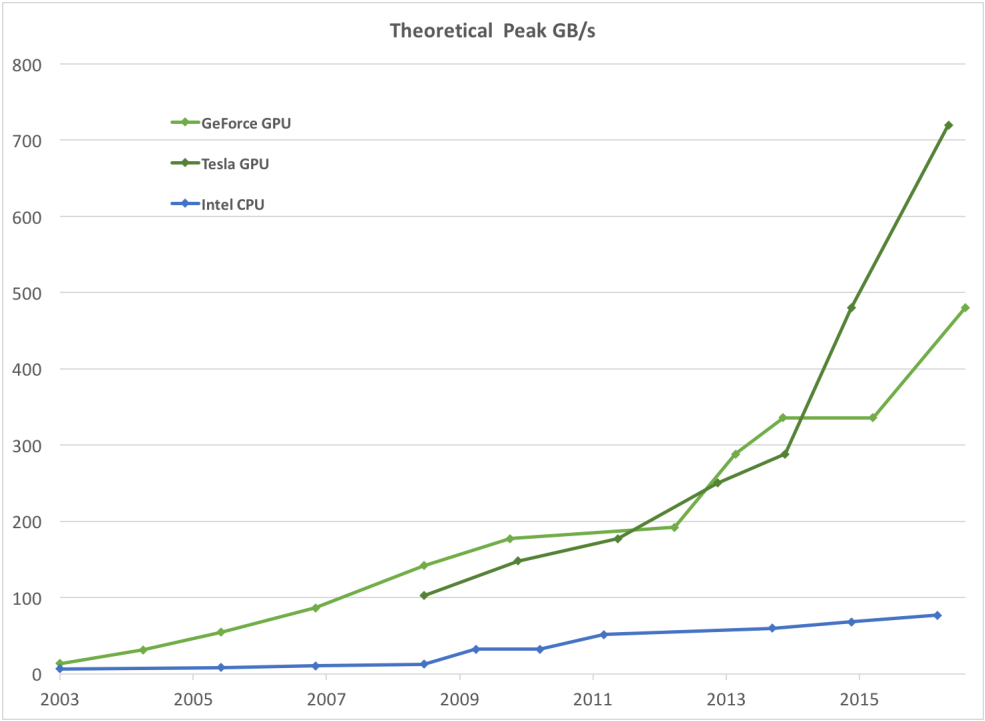


Figure 2.2: Memory Bandwidth for the CPU and GPU [2]

The reason behind the increasing difference in floating-point capability (figure 2.1) between the CPU and the GPU is that the GPU is specialized for compute-intensive, highly parallel computation and therefore can be designed

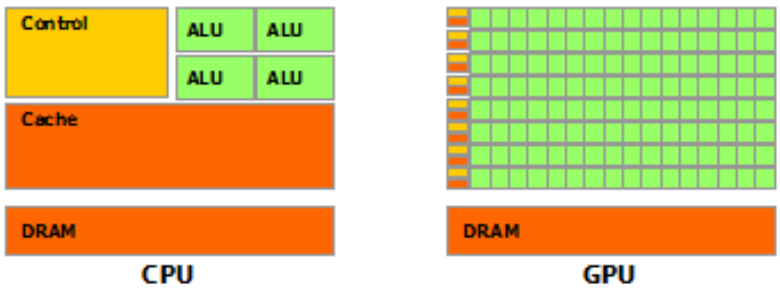


Figure 2.3: The GPU Devotes More Transistors to Data Processing [2]

so more transistors would be devoted to data processing rather than data caching and flow control, as schematically illustrated by figure 2.1

CPU is oriented to high performance for sequential code, that why a lot of space on CPU is occupied by flow control and data caching. Today processors use out-of-order execution. So processor can execute instructions that already have available data to process. Without this function many instruction cycles would be wasted while waiting for data.

GPU executes same program for many data elements in parallel. For this reason, requirements for flow control and data caching are not as complex and more space on chip can be used for data processing. Memory access latency can be easily hidden with calculations.

For effective use of GPU, it is required that code must be easily parallelizable and without often global synchronization. GPU might not be effective when code must often communicate with host or transfers a lot of data between GPU and host.

There are several programming languages used for GPU programming. Most known are CUDA, OpenCL OpenACC and OpenMP (supports GPU computing from version 4).

2.1 CUDA

CUDA is a parallel computing platform and programming model invented by NVIDIA. It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU). List of supported Graphic Cards can be found on this Nvidia webpage [28]

NVIDIA is considered to be leader in Graphic cards. Servers based on GPUs are mostly on hardware delivered by NVIDIA. Server line on GPUs by NVIDIA is called Tesla series and consumer line is called GeForce. Hybrids between these two segments are currently called TitanX.

Figure 2.4: Official Nvidia Logo [3]

NVIDIA knew that blazingly fast hardware had to be coupled with intuitive software and hardware tools, and invited Ian Buck to join the company and start evolving a solution to seamlessly run C on the GPU. Putting the software and hardware together, NVIDIA unveiled CUDA in 2006, the world's first solution for general-computing on GPUs

Today, the CUDA ecosystem is growing rapidly as more and more companies provide world-class tools, services and solutions.

If you want to write your own code, the easiest way to harness the performance of GPUs is with the CUDA Toolkit, which provides a comprehensive development environment for C and C++ developers. CUDA works with programming languages such as C, C++, and Fortran.

The CUDA Toolkit includes a compiler, math libraries and tools for debugging and optimizing the performance of your applications. You'll also find code samples, programming guides, user manuals, API references and other documentation to help you get started.

NVIDIA provides all of this, including NVIDIA Parallel Nsight for Visual Studio, the industry's first development environment for massively parallel applications that use both GPUs and CPUs.

2.1.1 Terminology

- **Host** - GPUs cannot work without CPU. In GPU computing CPU provides data and instructions to GPU, that why CPU is also called host.
- **Kernel** - Instructions designed to run in parallel on GPU.
- **Thread** - Single instance of kernel

- **Device** - Computing device (GPU) that runs threads in parallel
- **Warp** - Group of 32 threads
- **SM** - Streaming multiprocessor consisted of CUDA cores
- **(Thread) Block** - Streaming multiprocessor consisted of CUDA cores
- **Grid** - Set of thread block that can be executed on SM

2.1.2 Architecture

In this chapter we will discuss architectures of GPUs. Terminology and architecture will mostly be from new Nvidia GPUs, since CUDA is officially supported only on GPUs by Nvidia. AMD announced Heterogeneous-compute Interface for Portability. That should provide compatibility layer with CUDA for AMD GPUs. HIP tool can port CUDA runtime API's directly into C++. But from AMD testing this process is not completely automatic. Only 90% of code can be automatically converted into C++.

Cuda in available on architecture Fermi (figure 2.5) or newer. Developer tools are part of CUDA Toolkit. Latest version of CUDA Toolkit 8.0 was released in September 2016. Version 8.0 provides Support for Pascal Architecture, Unified Memory Performance optimization, support for NVIDIA NVLink. and up to two times faster compilation.

NVIDIA NVLink is a high-bandwidth interconnect between CPU and GPU. Data transmitting bandwidth is 5 to 12 times higher than PCIe 3. PCI Express (Peripheral Component Interconnect Express) is today standard bus (almost exclusively) for connecting GPUs. PCIe standard defines speeds and number of lanes which connected device can use. GPUs are connected with maximum of 16 lanes with speed of 985MB/s per lane, that's 15.75GB/s in total (calculated for PCIe 3). PCIe is full duplex so it is possible to achieve full speed for receiving and sending data at the same time. NVLink allows speeds up to 80GB/s between CPU and GPU or between two GPUs.

CUDA maps thread to a hierarchy of processors on the GPU; a GPU executes one or more kernel grids; a streaming multiprocessor (SM) executes one or more thread blocks; and CUDA cores and other execution units in the SM execute threads. The SM executes threads in groups of 32 threads called a warp. While programmers can generally ignore warp execution for functional correctness and think of programming one thread, they can greatly improve performance by having threads in a warp execute the same code path and access memory in nearby addresses.

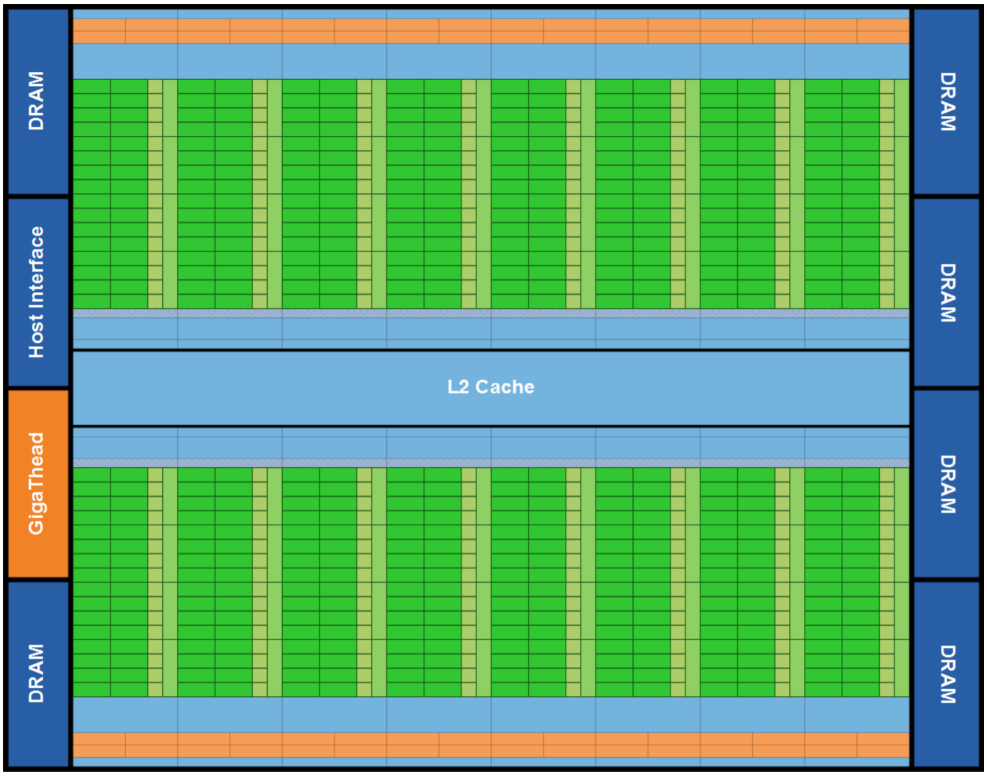


Figure 2.5: Fermi GPU Architecture [4]

Every GPU can differ in following:

- GPU frequency
- Memory size and Speed
- Number of SM
- Number of CUDA cores in one SM(XM)
- Cuda capabilities (table 2.1)

Each stream multiprocessor has 32 CUDA Cores (also called Cuda Processors). Each SM also contains instruction cache, shared memory, register array, warp scheduler and dispatch unit (figure 2.6).

Each CUDA Core has pipelined ALU (arithmetic logic unit) and FPU (Floating Point Unit). Prior Fermi architecture ALU was limited to 24-bit precision for multiply operations, for this reason 32-bit precision needed to be emulated with sequence of multiple instructions.

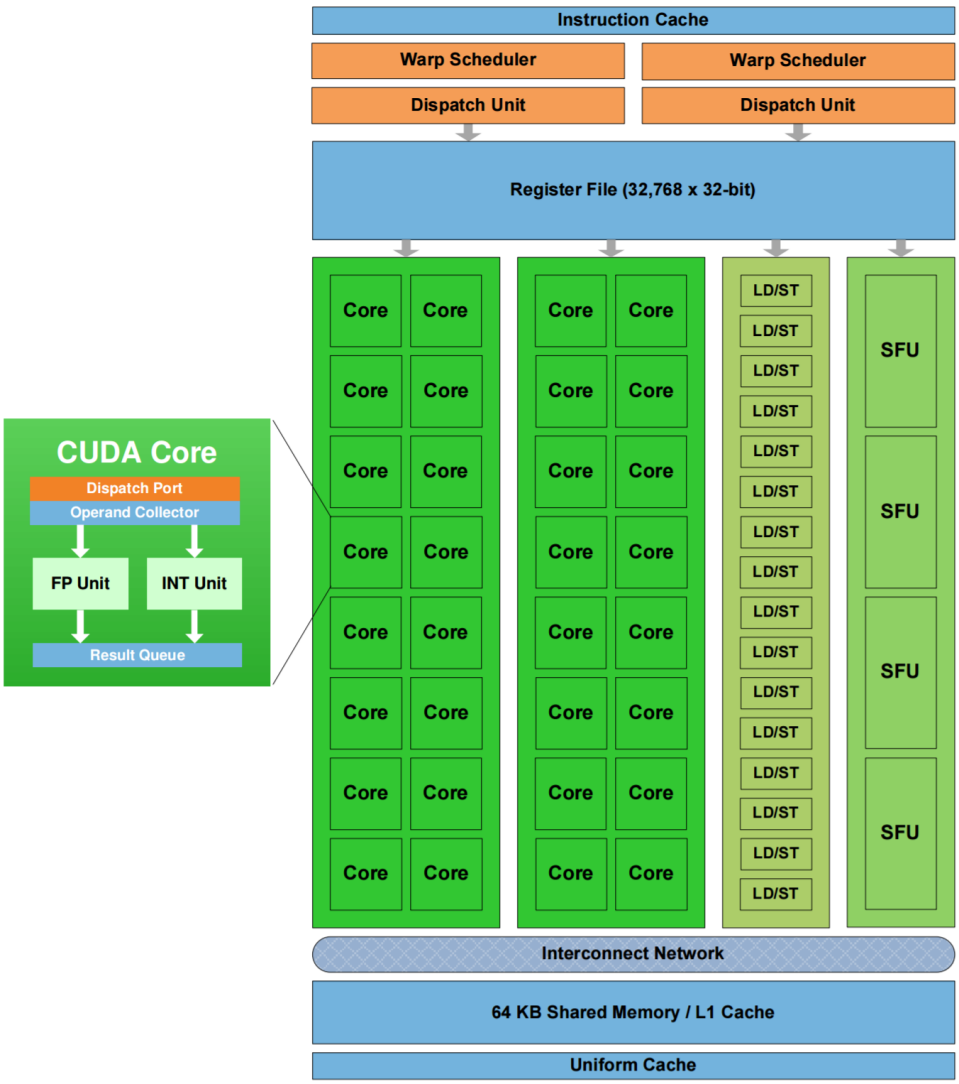


Figure 2.6: Fermi single stream multiprocessor [4]

Each stream multiprocessor has defined number of load/store units (LD/ST units) (defined by compute capability) allowing simulations. These units load and store data to cache or DRAM.

CUDA Cores are very simple and minimalistic. For that reason, more complex instructions are not executed on CUDA cores but on SFUs (Special Function Units). Instructions like sin, cosine, square root must be executed on SFUs. Every SM has limited number of SFUs. For example, Fermi has 4 SFUs per SM. That means in case every CUDA Core in warp wants to execute instruction on SFU execution time would be 8 instructions instead of

one. SFUs work only with single precision.

Stream Multiprocessor also contains number of DP Units designed to execute double precision computations. Double precision performance is usually very low compared to single precision especially on cunsumer GPUs. Double precision performance ranges from 1/2nd to 1/32rd of single precision performance.

Nvidia releases new GPU generation every year and names every architecture generation by famous mathematicians. Stream multiprocessors went through minor changes over past few years. Nvidia changes architecture more or less every two years (similar to Intels tick-tock).

- **SMX (Kepler architecture)** is used in GeForce 600 series GeForce 700 series and server Quadro K series. Each SMX has four warp schedules and eight dispatch units (allowing four warps per SM to be executed at the same time). One SMX contains 192 single precision CUDA cores, 64 double precision units, 32 special function units (SFU), and 32 load/store units (LD/ST) [5]
- **SMM (Maxwell Architecture)** is used in GeForce 800M series GeForce 900 series and server Quadro M series. One SMM contains 128 single precision CUDA cores, 4 double precision units, 32 special function units (SFU), and 32 load/store units. Double precision units are not depicted on diagram. [29]
- **SMM (Pascal Architecture)** is currently newest architecture and is used in GeForce 1000 series. Architecture is similar to Maxwell. [30]

2.1.3 Memory hierarhy

GPU cannot access memory of CPU and also CPU cannot access memory in GPU. To work with same memory block on CPU and GPU it is necessary to explicitly synchronize them.

CUDA defines following memory types:

- **Registers** are located in register array. They are very fast and they are allocated for each thread in SM. Number of registers per thread is specified by kernel. Number of threads on one SM can be limited in case of high register requirement of one each thread.
- **Global memory** is located outside of GPU chip. Most commonly used memory type is GDDR5, but in close future this type of memory should be replaced with HBM (some GPUs already use it). Global memory

Table 2.1: Cuda Compute capability

	3.0	**3.5**	**3.7**	**5.0**	**5.2**	**6.0**	**6.1**	**6.2**
Maximum resident grids	16	32	32	32	32	128	32	16
Maximum x-dimension grid	2^32-1							
Maximum y,z-dimension grid	2^16-1							
Maximum number of threads per block	1024							
Warp size	32							
Maximum number of resident blocks per multiprocessor	16			32				
Maximum number of resident warps per multiprocessor	64							
Maximum number of resident threads per multiprocessor	1536	2048						
Constant memory size	64 KB							
Maximum number of instructions per kernel	512 million							
Number of warp schedulers	4					2	4	
Number of instructions issued at once by scheduler	2							

is accessible from every thread blocks. Since CC 2.0 Global memory is cached. For maximum Bandwidth it is required by thread to use coalesced access. This memory can also be accessed by host.

- **Shared memory** is located inside each SM. Accessible by each thread inside single thread block. Reading from same address is broadcasted. Reading from multiple addresses in share memory leads to serialization of read requests.

- **Local memory** is part of global memory with interleaved addressing. Local memory accesses only occur for some automatic variables. Automatic variables that the compiler is likely to place in local memory are:

  1. Arrays for which it cannot determine that they are indexed with constant

2. Large structures or arrays that would consume too much register space
3. Any variable if the kernel uses more registers than available (this is also known as register spilling)

- **Constant memory** is read-only memory. It is part of global memory, but uses own cache.
- **Texture memory** is read-only memory designed for textures. Texture memory is located inside global memory but with custom 2D cache.
- **Read-only memory** is similar texture memory but can be used for other structures. Available from CC 3.0.

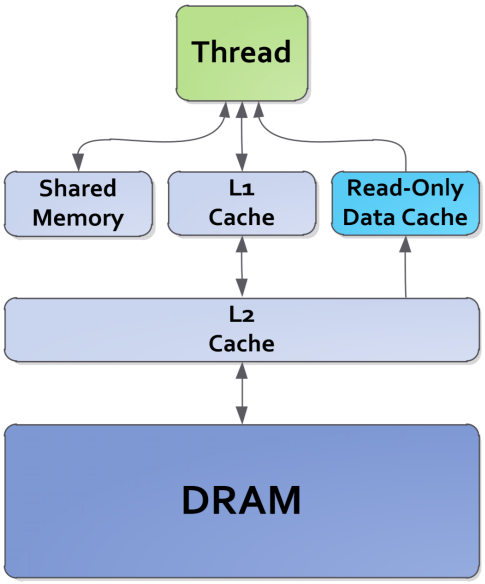


Figure 2.7: Kepler memory architecture [5]

Unified memory is available since CUDA SDK 6.0. It is simplest way how memory synchronization between host and device can be achieved. Programmer can allocate memory block on CPU, that is synchronized with GPU when needed.

2.1.4 Programming Basics

Kernels can be written directly by PTX (CUDA instruction set), but it is more time effective to use one of the supported high level languages such as

C or C++. Full C++ is supported for the host code. However, only a subset of C++ is fully supported for the device code.

Files with kernel code must be compiled by nvcc. Files can contain mix of device and host code. nvcc compiles kernel code into PTX or into CUDA binary (also referred to as cubin). Cubin files are by default embedded into host executable file. PTX code must be at runtime compiled into binary code by device driver. This is called just-in-time compilation. JIT compilation combines the speed of compiled code with the flexibility of interpretation, with the overhead of compiling. Host code is generated to automatically select at runtime the most appropriate code to load.

Kernels designed to run on GPU are functions with custom function qualifiers. There are three main function type qualifiers that can be used to modify function behavior and two minor modifiers:

1. **__device__** qualifier declares a function that is executed on device and can be only called from device.

2. **__global__** qualifier declares a function that is executed on device and can be called from host or device (from CC 3.2). Call of __global__ function is asynchronous. Meaning it returns before function is executed. For this reason, return values must be void.

3. **__host__** qualifier declares a function that is executed on host and can be called only from host (this modifier is default).

4. **__noinline__** can be used to as a hint to compiler to not inline function. (functions marked as inline must be in same file as kernels that call this functions)

5. **__forceinline__** can be used to force function to be inlined by compiler.

Programmers split kernel threads into two dimensions. First dimension is number of blocks executed on GPU and second is number of threads within a block. Each of these dimension can be separated into up to three dimensions (x,y,z). Inside kernel are predefined variables that can be used to identify thread index.

- **threadIdx** is three component vector that uniquely identifies thread inside thread block. Variable can be used to identify index in up to three dimensions (x, y, z).

- **blockDim** is three component vector that provides information about dimensions' sizes in block (x, y, z).

Listing 2.1: GPU Kernel definition

```
1  // definition of kernel
2  __global__ void HelloWorld()
3  {
4          //done on GPU
5          printf("Hello World!\n");
6  }
7
8  int main()
9  {
10         // calling of kernel from main
11         HelloWorld <<<1, 1>>>();
12 }
```

- **blockDim** is three component vector that uniquely identifies thread block. Variable can be used to identify index in up to three dimensions (x, y, z).

Variables are by default stored in registers. But in some cases compiler might chose to place some variables inside local memory. Variable Type Quantifiers can be used to change this behaviour.

1. __**device**__ qualifier declares a variable that is located on device. Memory space is located in global memory space if no other qualifiers is specified. Variables has lifetime of application.

2. __**constant**__ qualifier declares a variable that is located inside constant memory space. Qualifier is used with combination with __device__.

3. __**shared**__ qualifier declares a variable that is located inside shared memory space. Qualifier is used with combination with __device__. Memory is accessible only from same thread block.

4. __**managed**__ qualifier declares a variable that can be referenced from both device and host.

5. __**restrict**__ qualifier declares a restricted variable. By making variables restricts compiler will not that variables are not aliased. This can help the compiler to reduce number of instructions.

CUDA contains vector types derived from integer and float. These structures consist of one to four components. Components are accessible through fields x, y, z and w. Vector types needs to fulfill memory alignment requirements (table 2.2).

Table 2.2: Alignment Requirements in Device Code

Type	**Alignment**
char1, uchar1	1
char2, uchar2	2
char3, uchar3	1
char4, uchar4	4
float1	4
float2	8
float3	4
float4	16
double1	8
double2	16

Operation request on GPU can be enqueued in stream. Stream is sequence of commands that are executed in order. If no stream is specified default stream is used. Following device operations can run in parallel:

- Kernel launches
- Memory copies within a single device's memory;
- Memory copies from host to device of a memory block of 64 KB or less;
- Memory copies performed by functions that are suffixed with async
- Memory set function calls.
- Kernel Executions (CC 2.0 and higher)

2.1.5 Profilation tools

Nvidia provides profiling tools and API that helps with understanding runtime performance and code optimization. Profiler by Nvidia is called NVIDIA Visual Profiler. By this profiler it is possible to analyze activity on CPU and GPU. Profiler can be installed as a part of CUDA Toolkit which is available for all OS. Profiler can be integrated into Visual Studio or Eclipse with Visual Studio Edition for Windows or Nsight Eclipse Edition for Linux and Mac OS.

By default, running application through profiler would result in profiling entire run of application. To limit scope of profiling it is possible to to call $cudaProfilerStart()$ to start profiling and $cudaProfilerStop()$ to stop profiling.

Profiler also by default launches kernels synchronously unless Enable concurrent kernel profiling option is set in profiler setting.

2.2 OpenCL

The Open Computing Language (OpenCL) is open parallel computing API designed to allow application developing on GPUs and other coprocessors. OpenCL was initially developed by Apple Inc., which holds trademark rights. First version of OpenCL was released in 2008 by The Khronos Group.

The Khronos Group is non-profit consortium dedicated to creating of open standards in many areas like parallel computing, graphics and video coding. Company is funded by members. Membership in Khronos costs up to $75,000 for higher privileges like setting Khronos strategy and budgetary priorities and final ratification of the specification. Between main contributors are companies like AMD, Apple, ARM, IBM, Intel, NVIDIA, Qualcomm and Xilinx.

OpenCL was created because prior to OpenCL only proprietary programming languages were available. These languages were vendor locked, limiting possibility to create cross-platform applications. Best example of proprietary implementation is CUDA by Nvidia. Using the OpenCL API, developers can launch compute kernels written using a limited subset of the C programming language on a GPU.

Main benefit of OpenCL is ability to run same code on different computation platforms. OpenCL is compatible with OpenGL, so it is possible to share data structures between them.

As can be seen in figure 2.8 Nvidia supports OpenCL from its release, but integration of new versions of OpenCL is slowing. Support for version 2.0 is already available on GPUs by AMD and Intel but was not even announced on Nvidia GPUs.

2.2.1 Terminology

Terminology that is exactly same as in CUDA is not repeated here.

1. **Compute Unit** - Device consists of one or more compute units. Compute unit consists of one or more processing elements.

2. **Device** is collection of computing units capable of executing commands from OpenCL Command Queue

3. **work-item** is single independent instance of kernel.

4. **work-group** consist of work-items that can be bounded together.

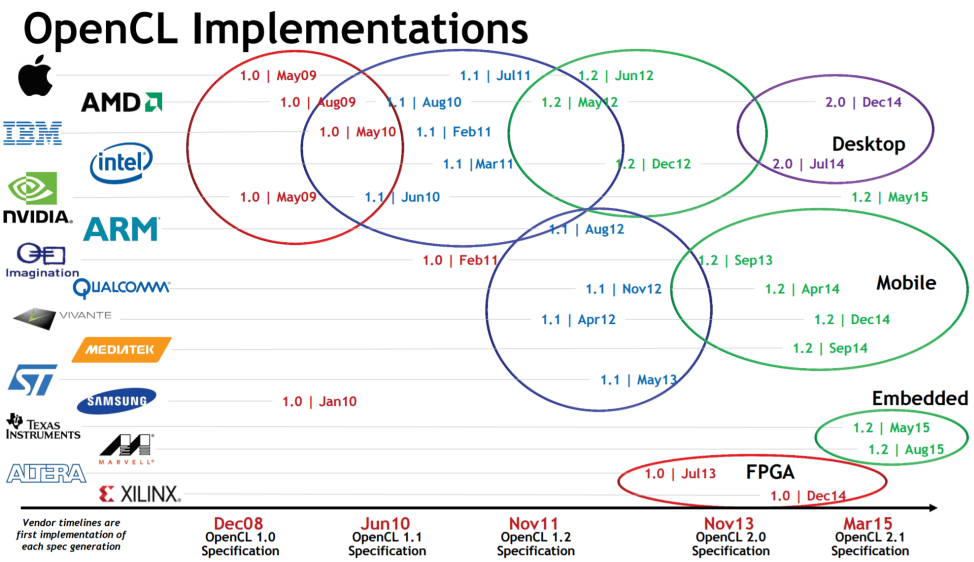


Figure 2.8: OpenCL Implementations [6]

2.2.2 Architecture

OpenCL architecture hierarchy is splitted onto four main models:

1. Platform model
2. Memory model
3. Execution model
4. Programming model

2.2.2.1 Platform model

OpenCL defines host, that can consist of one or more OpenCL Devices. OpenCL devices than consists of one or more compute units which are divided into processing elements. OpenCL Application sends commands (that are specified in kernels) which are executed on processing elements (figure 2.9).

Processing elements in Compute unit execute instructions with Single Instructions Multiple Data model where each step all processing elements must execute same instruction or as Single Program Multiple Data where same kernel is running all processing elements but asynchronized.

To support variety of devices for this reason OpenCL specifies following versions:

1. **Platform version** specifies supported version of OpenCL runtime.

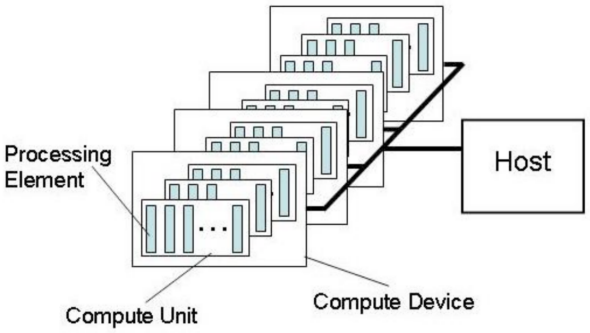


Figure 2.9: OpenCL platform model [7]

2. **Device version** specifies set of supported capabilities by device. Device version cannot be higher than platform version.

3. **Language version** specifies set of supported programming language features for kernels. Language version cannot be higher than Platform version but can be higher than device version.

OpenCL is backwards compatible. For that reason, device is not required to support older version.

2.2.2.2 Memory model

1. **Global Memory** is read/write memory accessible from all work groups. Global memory can be cached is device specifications allow it.

2. **Constant Memory** is read only memory. This area of memory is required to remain constant during kernel execution.

3. **Local Memory** is local read/write memory of work group. All work-items from same work group share this memory area.

4. **Private Memory** is read/write memory of single work-item.

2.2.2.3 Execution model

OpenCL application consists of two parts. Fist part is host program that is being executed on host. Second part are kernels (can be one or more kernels) that are executed on device.

Host program defines context for execution of the kernels. Within context can be create one or more command queues. Host inserts commands into command queue which are then executed on device. Commands in queue are executed in in-order execution model.

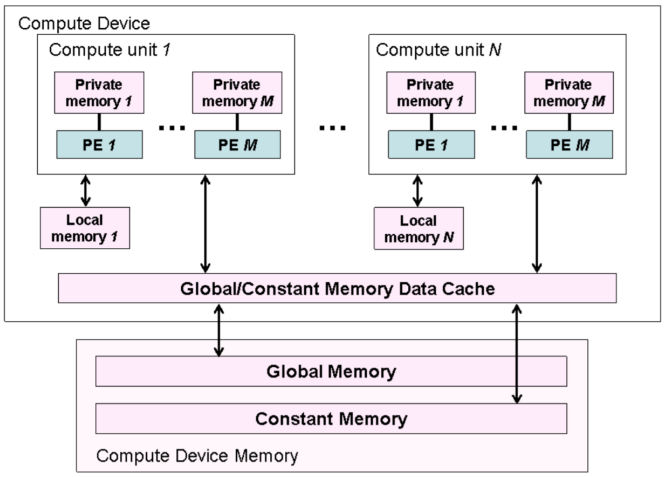


Figure 2.10: OpenCL memory model [7]

Kernel can be inserted into queue for execution. Instance of kernel is called work-item. Every work-item can by identified by N-dimensional index. Each work-item executes same kernel function but on different data. Space dimensions can be 1, 2, and 3.

Work-items are grouped into work-groups (figure 2.11). All work-items from same group must be executed on same device so it would be possible for all work-items within same group to share data in local memory and to allow easy synchronization.

Both work-items and work-groups can be executed in-order or out-of-order. For that reason, it is required that result of work-items cannot depend execution order.

Execution model provides two types of kernels. Kernels that are written in OpenCL C language and compiled by compiler. And Native kernels that are can contain extended instructions for specific device. Native kernels may not work on other devices.

2.2.2.4 Programming model

OpenCL supports both data parallel and task parallel models (even both at same time). Primary focus is on data parallel model.

1. Data parallel model (SIMD) executes same sequence of instructions on multiple data. Every work-item has associated own index and address space.

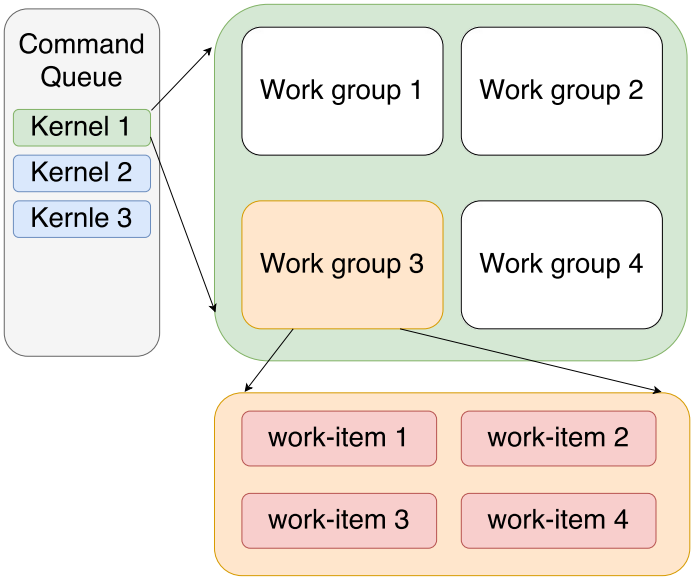


Figure 2.11: OpenCL execution model

2. Task parallel model executes instance of kernel independently of any index.

2.2.3 Programming Basics

OpenCL Development framework consist of three main parts:

1. **Language Specification** describes interfaces and syntax for kernels. Kernels can be precompiled or compiled at runtime. Language is ISO C99 with extensions for parallel computing and some restrictions.

2. **Platform API** lets developer know about existence of OpenCL compatible devices. Developer can then use this layer to detect device and create command queues for that device.

3. **Runtime API** is used for managing object like command queues, memory objects and kernel objects.

Kernel function in application is identified by qualifier __kernel. Kernel needs to be specified in own files. From Kernel files are created program objects. Program objects can be created online or offline. Online creating can be done by calling function clCreateProgramWithSource(). After this step program object is ready for compilation. OpenCL program executable (kernel) can be compiled by function clBuildProgram.

Listing 2.2: OpenCL Kernel definition

```
1 __kernel void HelloWorld(__global char * out) {
2         size_t tid = get_global_id(0);
3         out[tid] = hw[tid];
4 }
```

It is expected that memory area of host and device are independent. For this reason OpenCL API allows developer to allocate area in global memory and add memory commands to command queue. Memory commands can synchronous or asynchronous. Synchronous call return at the moment when command was executed and memory on host is no longer needed (in case of data transfer).

In case memory of host and device are not independent it is possible to map/unmap memory regions.

2.3 OpenMP

OpenMP is API for shared memory parallelism. Programs with OpenMP can be written in programming languages like C, C++ and Fortran.

Programmers can easily create parallel programs using directives, library routes and environment variables defined by OpenMP.

Main difference between OpenMP and CUDA (or OpenCL) is that OpenMP does not explicitly define how should program be executed on device, but rather lets developer to specify code, that can be executed in parallel.

Support for GPUs was added with version 4.0 released in year 2012. However, there is still limited number of production ready tools. This technology currently works best with Xeon Phi accelerator by Intel.

Xeon Phi is massively-parallel multicore processor targeted to super computers. Architecture is based on Intel Xeon but Xeon processors dedicate only 20% of space on chip to actual cores while Xeon Phi dedicates major space on chip to cores 2.12. Cores (figure 2.13) are based on Intel® Pentium cores. In this architecture up to 61 cores are places on single chip and connected by bidirectional ring with fully coherent F2 caches. Each core can run up to four threads to hide memory access latency. Each core has implemented 512-bit

vector instructions, so it is possible to execute sixteen 32-bit vector instructions. Theoretical performance of Intel Xeon Phi processors can be calculated with following formulas:

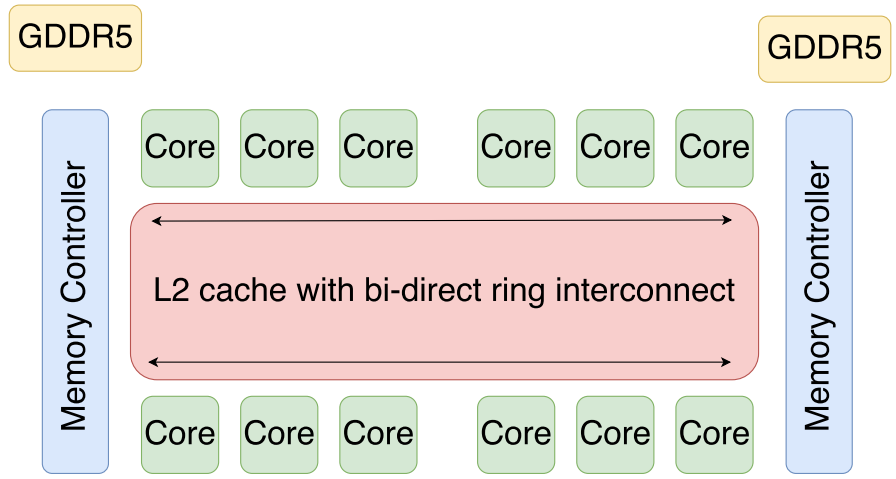


Figure 2.12: Xeon Phi Architecture

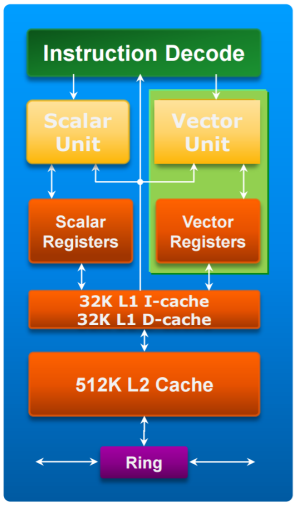


Figure 2.13: Xeon Phi Single Core Architecture [8]

- **Single Precision** = 16 (SP SIMD Lane) * 2 (FMA) * 1.1 (GHZ) * 60 (# cores) = 2112 GFLOP/sec
- **Double Precision** = 8 (DP SIMD Lane) * 2 (FMA) * 1.1 (GHZ) * 60 (# cores) = 2112 GFLOP/sec

Formulas are expecting that one core is dedicated to Operating System.

Table 2.3: Optimization techniques for GPU and Xeon Phi

Method	**GPU**	**Phi**
Libraries	CUDA Libraries + others	Intel MKL + others
Directives	OpenACC	OpenMP + Phi Directives
Programming Models	Cuda or OpenCL	Vector Intrinsics

OpenACC is alternative to OpenMP. While OpenMP is more CPU focused, OpenACC is standard designed to simplify parallel programing on GPUs. OpenACC is designed for portability across operating systems, host CPUs, and a wide range of accelerators, including APUs, GPUs, and many-core coprocessors. Big advantage of OpenACC is implementation of OpenACC 2.0a in GCC 6 release series.

Nvidia recommends using PGI Community Edition compiler by PGI. Compiler includes all OpenMP and OpenACC features. Compiler is available for Linux and macOS while Windows version is available only in paid PGI Professional Edition.

If we compare difference in programming for Xeon Phi and GPUs it required basically same optimization techniques for both platforms.

Chapter 3

Realization

In this chapter I will describe architecture of developer application. Firstly I will describe used frameworks and why I've chosed them.

Program is written in programming language Visual C++ 2015 (also known as Visual C++ 14.0). CUDA is designed to work with programming languages like C, C++ or Fortran. OpenCL is available for C or C++. I prefer object oriented programming, for that reason I've selected C++ as main developing language. Project was developed under Windows. but it should be possible to easily port it to Linux or MacOS.

3.1 Application architecture

Application is programmed with architecture that can be easily extended. That allowed to easily add multiple comparison algorithms programed with multiple technologies. I've implemented video quality evaluation algorithms like PSNR and SSIM described in first chapter. First implementations were CPU only, then I've tried to utilize GPU using CUDA and OpenCL.

Application takes video streams data from Source modules. One source module provides reference data and second compressed data to compare. Evaluation of frames generated by Comparator Modules is done on Comparator modules. Aplication architecture can be seen in figure 3.1.

Source module generates decoded video frame in AVFrame structure. AVFrame structure origins from FFMPEG framework and in described in their documentation. This structure is used even when source of video might not origin from FFMPEG framework. All sources must be inherited from DaxAVSource class (Listing 3.1). That requires implementing getNextFrame method. All

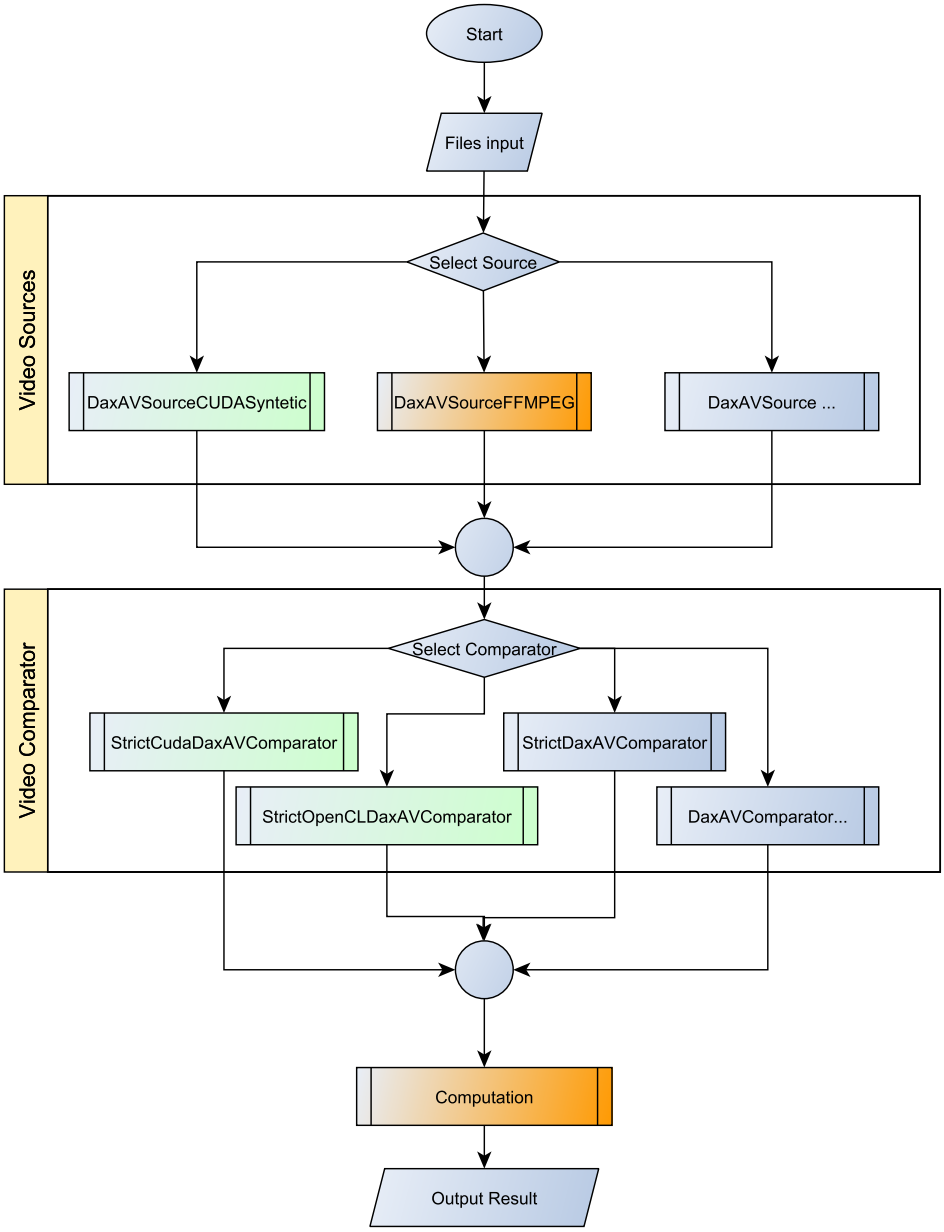


Figure 3.1: Flowchart for video comparition

other methods are optional. Method getNextFrame should return NULL after all frames were generated. Source modules are executed on main thread.

Developed application consists of following source modules:

- DaxAVSourceSyntetic

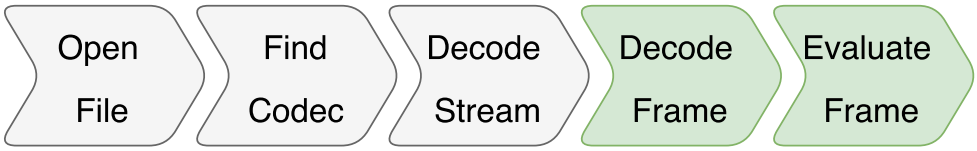


Figure 3.2: Application frame evaluation

Listing 3.1: DaxAVSource

```
1  class DaxAVSource
2  {
3  public:
4          bool reference = false;
5          int frameCount = 5000;
6          int width = 1920;
7          int height = 1080;
8          int currectFrame = 0;
9          char * filename;
10         virtual void getParameters(int argc, char**
               ↪ argv);
11         virtual int init();
12         virtual bool getNextFrame(AVFrame * frame) =
               ↪ 0;
13 };
```

- DaxAVSourceFFMPEG

Comparator modules compute difference between two video streams. All comparators must be inherited from DaxAVComparator class (Listing 3.2) Most important is method compute which takes structure AVImageComparatorData, that has two frames in AVFrame structure and contains additional information about frames.

Basic application consists of following comparator modules:

- SimpleStrictAVComparator
- PSNRSeqDaxAVComparator
- SSIMDaxAVComparator
- CudaSimpleStrictAVComparator
- CudaSSIMDaxAVComparator
- OpenCLStrictDaxAVComparator

Listing 3.2: DaxAVComparator

```
1  class DaxAVComparator
2  {
3  public:
4          double diff = 0;
5          long frameCount = 0;
6          int height = 0;
7          int width = 0;
8          DaxAVComparator();
9          ~DaxAVComparator();
10
11         virtual void getParameters(int argc, char**
               ↪ argv);
12         virtual int init(int height, int width);
13         virtual void registerFrame(AVFrame * frame);
14         virtual void customInit();
15         virtual void evaluateFrame(
               ↪ AVImageComparatorData data);
16         virtual void processed();
17         virtual bool wantsFramesInRGB();
18         virtual bool supportsMultiThreading();
19 };
```

3.2 Multimedia framework

To support as many codecs as possible I've wanted to integrate multimedia framework that would be able to decode all sorts of them. Since I've had very good experience with FFmpeg from my previous projects. Thats why FFmpeg was my first choise as source module.

FFmpeg is the leading multimedia framework, able to decode, encode, transcode, mux, demux, stream, filter and play pretty much anything that humans and machines have created. It supports the most obscure ancient formats up to the cutting edge. No matter if they were designed by some standards committee, the community or a corporation. It is also highly portable: FFmpeg compiles, runs, and passes our testing infrastructure FATE across Linux, Mac OS X, Microsoft Windows, the BSDs, Solaris, iOS, tvOS etc. under a wide variety of build environments, machine architectures, and configurations.

Between most important supported codecs is x.265 mentioned in section 1.5 and x.264 mentioned in section 1.4.

Listing 3.3: FFMPEG build script

```
Download the script
git clone this repo:
$ git clone https://github.com/rdp/ffmpeg-windows-build-helpers.git
$ cd ffmpeg-windows-build-helpers

Or do the following in a bash prompt instead of git clone:
$ mkdir ffmpeg_build
$ cd ffmpeg_build
$ wget https://raw.github.com/rdp/ffmpeg-windows-build-helpers/master/cross_compile_ffmpeg.sh -O cross_compile_ffmpeg.sh
$ chmod u+x cross_compile_ffmpeg.sh

Now run the script:

$ ./cross_compile_ffmpeg.sh --build-ffmpeg-shared=y --gcc-cpu-count=4 --disable-nonfree=n --compiler-flavors=multi
```

FFmpeg is licensed under the GNU Lesser General Public License (LGPL) version 2.1 or later. However, FFmpeg incorporates several optional parts and optimizations that are covered by the GNU General Public License (GPL) version 2 or later. If those parts get used the GPL applies to all of FFmpeg.

I've also looked for alternatives. Second biggest multimedia framework is AVconv. AVconv was created as fork from FFmpeg. This fork was created because group of original developers wanted to focus more on maintaining current state of library, rather than to keep development velocity with more features. While FFMpeg integrates changes from AVconv, but AVconv ignores any progress in FFmpeg. It has been suggested to merge both projects together, but it seems that future of AVconv is uncertain as it lost o lot of support.

FFmpeg used in this project are custom builded using ffmpeg-windows-build-helpers script [31]. Build was done in Bash on Windows 10 anniverssary update with Windows Subsystem for linux. Build parameters can be seen in listing 3.3.

Listing 3.5: Application examples

```
# Evaluate video quality by strict comparator (cpu)
-c Strict -r "F:\\Documents\\Local\\AVTest\\
   ↪ sintel_loseless.mp4" -f "F:\\Documents\\Local\\
   ↪ AVTest\\sintel_265.mp4"

# Evaluate video quality by SSIM comparator (Cuda)
-c SSIMCuda -r "F:\\Documents\\Local\\AVTest\\
   ↪ sintel_loseless.mp4" -f "F:\\Documents\\Local\\
   ↪ AVTest\\sintel_265.mp4"

# Generate frames from video sequence
-g -p -r "F:\\Documents\\Local\\AVTest\\
   ↪ sintel_loseless.mp4"
```

3.3 Usage and examples

Parrameters of developed application can be printed with parammeter $-h$ or $--help$.

Listing 3.4: Comparator parameters

```
-f Input file name
-r Reference input file
-c Comparator name
-s Syntetic reference
-g Generate frames
-p Frames as PNG
-h or --help Prints help

-N Block size for SSIM evaluation
-kernel Kernel used for parrallel reduction
-blocksize Number of threads in block

--verbose Print detail about frame execution
--frameLimit Limit frame count from input file
--forceRGB Force execution in RGB (works only if
   ↪ comparator supports RGB)
--forceYUV Force execution in YUV (works only if
   ↪ comparator supports YUV)
--threadCount Thread count used for Video Quality
   ↪ Comparation
```

3.4 Kahan summation algorithm

During development of SSIM comparators on CPU and GPU I've noticed different slightly results. Sum of differences for example video sequence on CPU was 174028.031 while sum of differences on GPU was 173937.578.

Firstly, I've tried to change computations on GPU to double, because I've checked values on CPU and on GPU (before parallel reduction) and numbers were exactly same, so I've figured that problem is reduction. That did not change result but noticeably slowed execution time. So I've reverted computation to float precision.

After implementing Kahan summation algorithm on CPU code I've got result 173937.594, which is very close to GPU result. And after dividing with 174080 (number of blocks) result was 0.999181 (same on both CPU and GPU).

Kahan summation algorithm (also known as compensated summation) a method of summing series of number represented with limited precision [32]. This is done by remembering errors during calculations as can be seen in listing 3.6. With compensated summation error depends only on floating point precision.

Listing 3.6: Kahan summation algorithm C++

```
1 void reduceCPUKahan(float& sum, float numberToAdd,
    ↪ float& c)
2 {
3         double y = numberToAdd - c;
4         double t = sum + y;
5         c = (t - sum) - y;
6         sum = t;
7 }
```

To minimalize problem with float errors float operations were changed to int operations if possible. Operations that require division were added as a last step in frame execution. Same algorithm was also added to accumulate values from all frames evaluation before calculating mean.

3.5 Cuda vs. OpenCL

I've started Cuda implementation in version 7.5. Project was in later phase ported to Cuda 8, that was released in July as Release Candidate and full version was release in September. Open CL is used in version 1.2 as it is latest version supported on Nvidia GPUs.

OpenCL in generic model for GPU computing, for that reason it requires a lot more steps compared to Cuda. Many steps in OpenCL like creating command queue can be done also in CUDA, but creating of custom command queues is only for optimization.

Cuda execution Phases

1. Selecting of appropriate device (optional)
2. Allocating of memory objects
3. Copying data to device
4. Start kernel execution
5. Copying results to host
6. Deallocating of memory objects

OpenCL execution Phases

1. Selecting of platform
2. Selecting of device
3. Creating of context
4. Creating od command queue
5. Allocating of memory objects
6. Copying data
7. Creating program object
8. Creating executable program
9. Creating kernel object
10. Setting kernel parameters
11. Start kernel execution
12. Deallocating of memory objects

Code written in OpenCL can be executed on all OpenCL capable devices. There is no guarantee code will run on all devices since code might expect some minimal size of global memory, etc ... Without knowledge of architecture it is hard to create code that would be effective on all platforms. Thankfully in case of GPUs differences in architectures are relatively small. AMD GTN (Graphics Core Next), Intel EUA (Execution unit Architecture), and Nvidia Fermi (and newer) are very similar.

Because of generic model, OpenCL does not support platform specific features. For example, OpenCL does not support Shift down instructions used for parallel reduction on Kepler architecture and newer.

Compiling of Cuda code takes long time compared to OpenCL. Even though Cuda 8.0 speeded up compiling up to two times, it is still considerably slower compared to OpenCL, that does not require compilation on kernels. Cuda requires dedicated compiler by Nvidia as Cuda code contains syntax that extends standard C. Compile time of Cuda can be increased by removing support for older architectures.

On other hand OpenCL needs to compile at runtime and thus it takes longer time to execute. This in not factored into measurement because time measurements are done without starting initialization.

IntelliSense in Visual Studio provides features like List Members, Parameter Info, Quick Info, and Complete Word. Many aspects of IntelliSense are language-specific. IntelliSense in Visual Studio does not manage to understand syntax of calling CUDA kernels. In case of compile error IntelliSense assumes no CUDA files can be compiled and that creates imaginary compile errors. It's not exactly problem of CUDA, but problem on integrating CUDA to IDE.

Few times I've managed to freeze entire PC with OpenCL by running kernel compilation during execution of another kernel. This happened because of failing GPU driver, but I've did not found any solution for that. My workaround was to simply precompile all kernels at beginning, so this situation would not happen. Similar thing happened few times during profiling CUDA on GPU, but it was very rare occasions.

3.6 GPU implementation

In this chapter we will discuss which parts of program could be done on Graphic Cards and which actually were programmed.

Since evaluation of single frame took 120ms while frame decoding took only 40ms. It was obvious that moving frame evaluation to GPU would be first step. For start I've started to optimize simplest available comparator and that is SimpleStrictAVComparator. Its CUDA implementation is called CudaSimpleStrictAVComparator and OpenCL implementation is OpenCLStrictDaxAVComparator.

To achieve high occupancy of CUDA cores, it is necessary to select optimal number of blocks and number of threads in block. Cuda offers cudaOccupancyMaxPotentialBlockSize which heuristically calculate a block size that achieves maximum occupancy. This function calculated that for PSNR ideal number of threads on block would be 1024 and minimal recommended number of blocks is 30. For Full HD video sequences is number of blocks 2025 for PSNR kernel. For SSIM number of threads in block were calculated to 640. Nvidia Profiler showed 100% occupancy with usage of recommended settings. Block size can be changed manually by parameter -blocksize.

To perform competition of one frame on GPU it is necessary to move decoded frames to global memory on GPU. That is 1920*1080*3*2 bytes (11.86

MB) in case of two Full High Definition videos. Then calculate difference in preallocated array (5.93 MB) and perform parallel reduction. From parallel reduction we get just one number, which must be moved back to host RAM.

3.6.1 Parrallel reduction

Tree-based approach used within each thread block. GPU needs to be able to use multiple thread blocks to process very large arrays. Each thread block reduces a portion of the array. After that there is problem with synchronization between thread blocks. CUDA doesn't provide any way to synchronization between all thread blocks because it would be too expensive to build it in hardware inside GPU.

As a solution to this problem parallel reduction is decomposed into two kernels. Launch of second kernel serves as synchronization point. Launch of kernel has low software overhead and almost none hardware overhead.

1. Kernel 1 - interleaved addressing with divergent branching
2. Kernel 2 - interleaved addressing with bank conflicts
3. Kernel 3 - sequential addressing
4. Kernel 4 - first add during global load
5. Kernel 5 - unroll last warp
6. Kernel 6 - completely unrolled
7. Kernel 7 multiple elements per thread

Parallel reduction on GPU is done in $Log(N)$ parallel steps, each step S does $N/2^S$ independent operations. Step complexity is $O(logN)$. Implementation is work-efficient. i.e. does not perform more operations than sequential algorithm. Time complexity is $O(N/P + logN)$. Cost is $O(NlogN)$ so it is not cost efficient.

Brent's theorem suggests using $O(N/logN)$ threads. Then each thread does $O(logN)$ of sequential work. Cost is than $O((N/logN) * logN) = O(N)$ and that is cost effective. This can be achieved using algorithm cascading. Parralel reduction combines sequential and parrallel reduction. Each thread loads and sums multiple elements into shared memory. Tree-based reduction in shared memory.

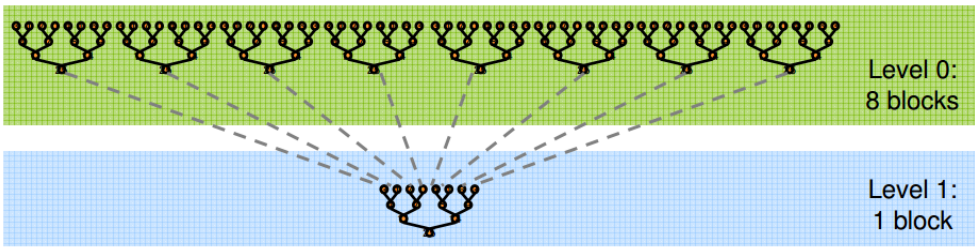


Figure 3.3: Cuda parrallel reduction

Table 3.1: Cuda kernels for parralel reduction

	Time (2^22 Ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1	8.054 ms	2.083 GB/s		
Kernel 2	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6	0.381 ms	43.996 GB/s	1.41x	21.16x
Kernel 7	0.268 ms	62.671 GB/s	1.42x	30.04x

Parallel reduction is a common building block for many parallel algorithms thats why Nvidia provides several prepared kernels under GNU licence to show how parallel reduction can be optimalized. Most recent changes in parrallel reduction optimalization were done in Kepler architecture with Shuffle On Down instructions.

Kepler's shuffle instruction (SHFL) makes posible to read register of other threads within same warp. Earlier it was necessary to exchange data using shared memory. That meant writing data to memory, sychronize threads in same warp and read data back from shared memory.

Fastest known CUDA algorithms are released as CUB library [33]. CUB provides state-of-the-art, reusable software components for every layer of the CUDA programming model. CUB simplifies high performance program and kernel development by taking care to implement the state-of-the-art in parallel algorithms.

As a result, CUB implementations demonstrate much better performance when compared to more common parallel libraries such as Thrust.

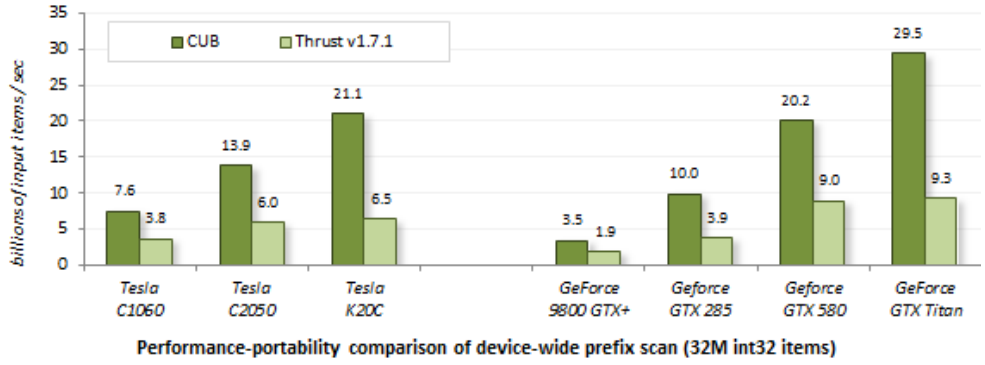


Figure 3.4: CUB performance comparition

Sadly parralel reduction in CUB and all Kernels provided by NVIDIA are not in char but in int. Thats because GPUs are designed to work with ints or floats (single precision). Since most of videos use just 8bits per channel then GPU must work with four times more data during parrallel reduction than necessary. In future it is expected to use wider range of colors and use up to 16bits per channel. Than it would be required to read just two times more data. For this reason GPU must have alloced 23.72 MB per frame for measured differences.

If parameter –kernel is not specified, then a CUB parallel reduction is used. Parallel reduction from CUB is approximately two times faster compared to custom kernels.

3.7 Optimalizations

Execution of frames was moved from main thread to background thread. Thanks to this modified architecture. Source can create decode frame during frame evaluation. If comparator return true from supportsMultiThreading(), then evaluation can run on multiple threads. Thanks to this CPU evaluation scaled almost linearly.

GPU comparators also benefit from multiple CPU threads. Every thread on host uses own GPU command queue for frame evaluation. This behavior was achieved by specifying command queue for all async commands. Easier way to achieve this (from CUDA 7) is specifying –default-stream per-thread parameter to nvcc. This change was expected to bring simultaneous kernel execution and data copy to device. In many cases source (FFMPEG) does not manage to decode frame fast enough to achieve this effect. Difference in usage of multiple streams for execution can be best seen in Nsight Performance

analysis. Performance analysis of single stream execution can be seen at figure 3.5 while multiple stream execution can be seen at figure 3.6. Figure 3.6 also shows that multiple independent data transfers are done without need to wait for evaluation by kernel.

Figure 3.5: Cuda single stream execution

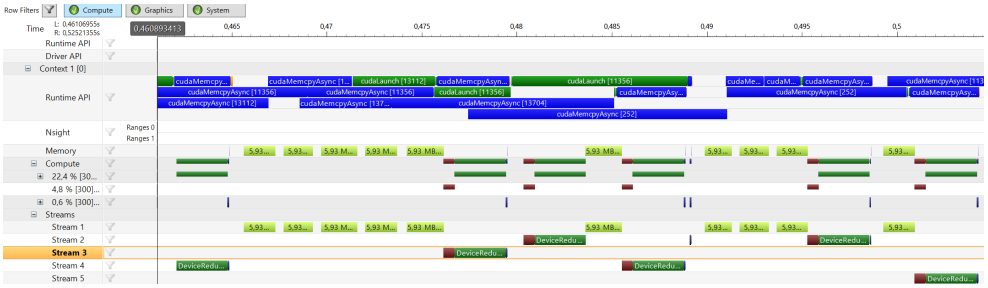


Figure 3.6: Cuda multiple stream execution

Another optimization was preallocating structures and memory both on host and device. Most important were AVFrame structures which contain data objects of frame. These structures are preallocated and stored in class called MultiBuffer. This class contains AVFrame structures inside a stack. During each call of getNextFrame on Source Module it pops one instance of AVFrame from stack. Instances of AVFrame are after evaluation collected and push back to stack. This allows to generate more frames in advance in case of slow quality evaluation.

3.8 Frame decoding optimalizations

Decoded framed are always in representation they were encoded in. FFMPEG provides function sws_scale that converts any frame data into defined representation (for example RGB), but this function was also performance bottleneck. Evaluation of single frame took 40ms while from that period 32ms was on sws_scale function.

In chapter 1.1 we've discussed different color spaces and how they are represented. Test data are mostly represented in YUV and since data evaluation

Listing 3.7: YUV to RGB Forumula

```
1 Y = Y
2 Cb = Cb - 128
3 Cr = Cr - 128
4 R =  Y + Cr + (Cr >> 2) + (Cr >> 3) + (Cr >> 5);
5 G =  Y - ((Cb >> 2) + (Cb >> 4) + (Cb >> 5)) - ((Cr
    ↪ >> 1) + (Cr >> 3) + (Cr >> 4) + (Cr >> 5));
6 B =  Y + Cb + (Cb >> 1) + (Cb >> 2) + (Cb >> 6);
```

was designed to be in RGB sws_scale needed to convert data from YUV to RGB. Easiest way to solve this is converting video sequences into RGB. Then decoded frames are directly in RGB and no converting needs to be done.

To find out why changing representation from YUV to RGB takes so long I've implemented custom function from YUV to RGB. Implementation was sequential a from tests I've find out it was exactly same speed as sws_scale. So FFMPEG does not accelerate representation conversion in any way. Conversion from YUV to RGB in standardized with following formula (8 bits per channel):

I've also created GPU accelerated version. While converting data on GPU was extremely fast this time bottleneck was PCIe limit. In case of executing data on CPU copying data to/from GPU took most of time. But even with need to move data, scaling on GPU took only 2ms (copy times included).

In case of comparing data on GPU there is no need to copy data back to host. That allowed to lower execution time of single frame to only 7ms.

Nvidia GPUs contain separate chip (ASIC) speciffically designed to encode and decode video sequences. Nvidia provides VIDEO CODEC SDK designed to allow developers to use it. Hardware-Accelerated Video Encoding uses NVENC (figure 3.8) and Hardware-Accelerated Video Decoding uses NVDEC (also can be called CUVID) as can be seen in figure 3.7.

FFMPEG uses its own decoding library to decode video. This decoder works with multiple threads. This behavior can be changed to use other decoder such as NVDEC.

Supported coders and decoders change with each GPU generation. For example, VP9 and 10bit support is available only on Pascal generation. List

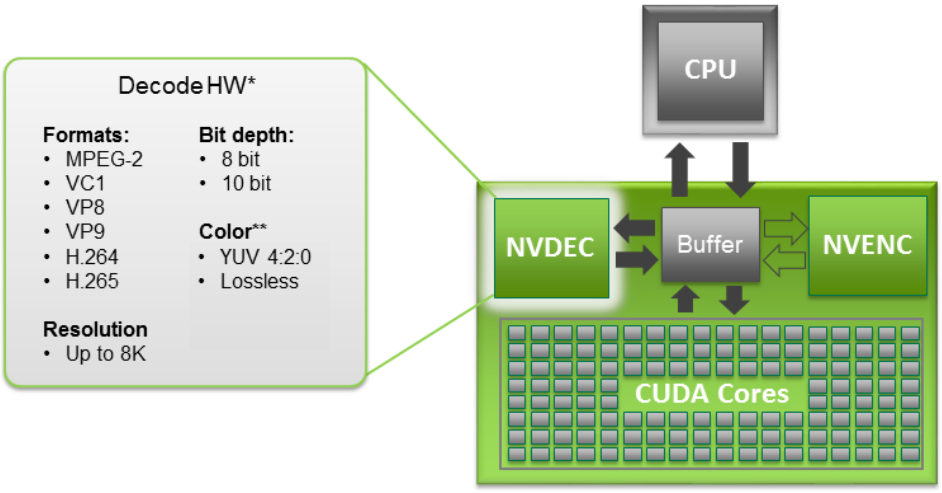


Figure 3.7: NVDEC

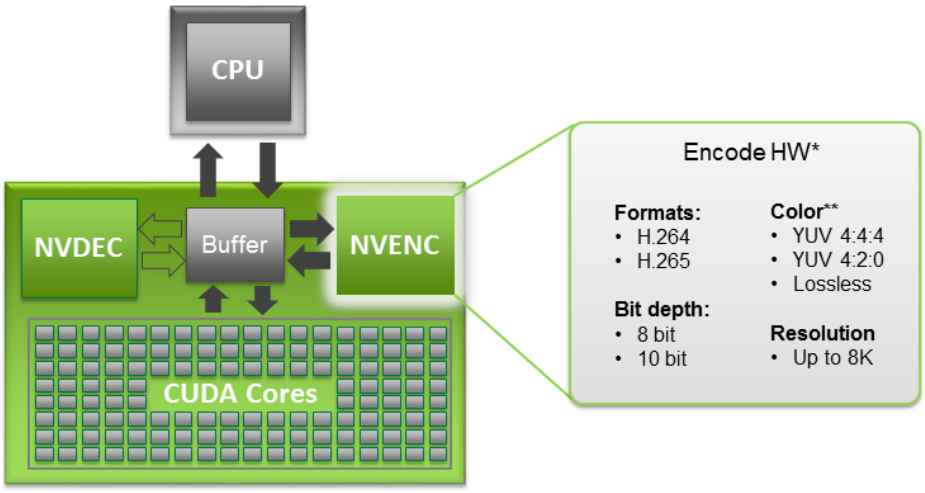


Figure 3.8: NVENC

of codecs supported for each generation can be found on Nvidia website [34]. FFMPEG can use NVENC and NVDEC if following conditions are meet:

- A supported GPU
- Supported drivers
- ffmpeg configured without –disable-nvenc

FFMPEG does provide any way to leave decoded data in buffer and copy them in device memory. That's why data are always copied from buffer to host and then to global memory on device as can be seen in figure 3.9.

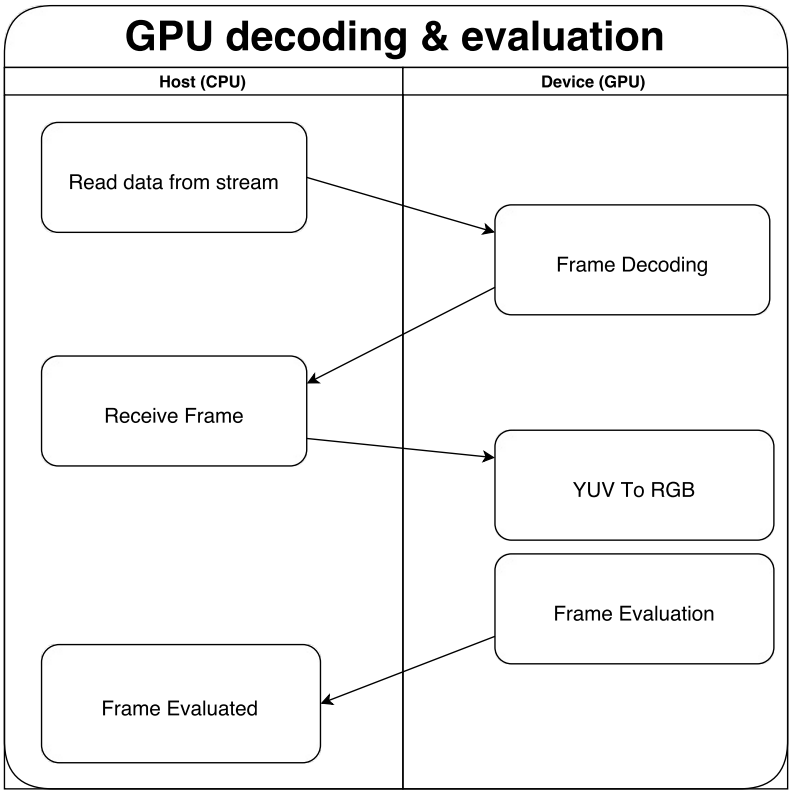


Figure 3.9: NVDEC decoding and evaluation

Chapter 4

Codec Testing

4.1 Testing Data

Most of test videos are not provided on disk as they require to big data space. All video sequences in raw format can be downloaded from internet or generated by developed application.

4.1.1 Syntetic video sequences

I've created simple Source module that generates simple syntetic video sequences. This video sequence is created of squares that move from the left to the right. It has very high compress ratio even for lossless codecs. Resolution of output images is by default Full HD (1920x1080).

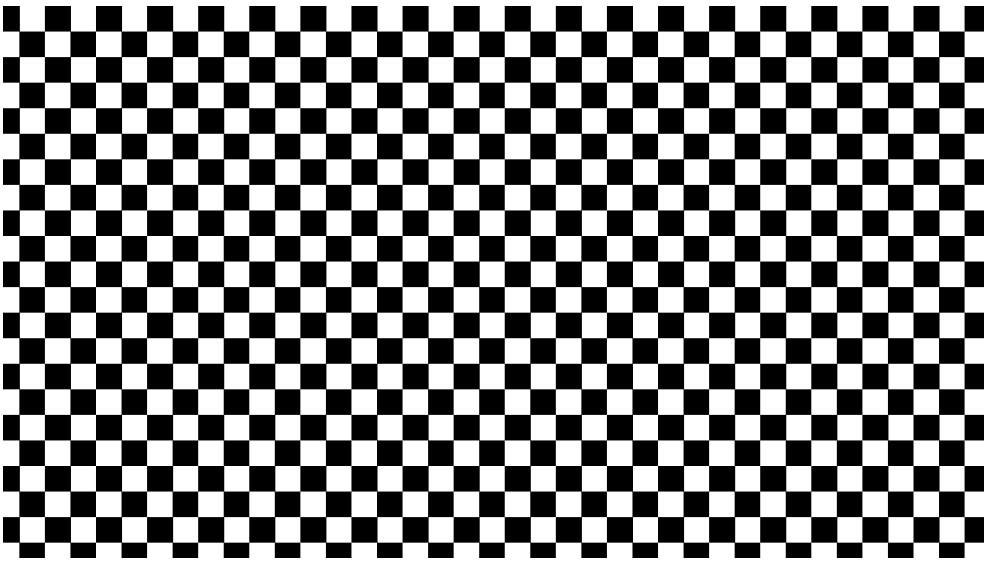


Figure 4.1: Square example image

Frames of video sequence can be generated by following command:

Listing 4.1: Square video sequence

```
1 HEVC_comp -g -p -s DaxAVSourceSyntetic
```

4.1.2 Real video sequences

4.1.2.1 Sintel

Sintel is an independently produced short film, initiated by the Blender Foundation as a means to further improve andvalidate the free/open source 3D creation suite Blender. With initial funding provided by 1000s of donations via the internet community, it has again proven to be a viable development model for both open 3D technology as for independent animation film.

Figure 4.2: Sintel example image

Sintel is licensed as Creative Commons Attribution 3.0. That means it can be shared and showen, for as long you include the credit scroll of the film itself.

Video sequence contains 1000 frame snippet with 24 frames per second (around 40 sec) of this video. This video sequence contains high movement and camera panning. Color format is YUV 444, so there is no chroma subsampling. This video sequence needs to be downloaded from official websites frame by frame. I've created script (listing 4.2) that downloads required frames and creates video sequence.

4.1.2.2 HoneyBee

HoneyBee is video sequence of 600 frames. Video sequence can be downloaded on website of Ultra Video Group [35]. Used versions are 8-bit, YUV, RAW

Listing 4.2: Sintel download and convert script

```
#!/bin/bash

url_template="http://media.xiph.org/sintel/sintel-1080-png/"
# or
# url_template="http://media.xiph.org/sintel/sintel-4k-png/"


for i in {6000..6999}; do
    wget -O $(printf '%08d.png' $i) "$(printf "$url_template" $i)"
done


# Creation of lossless format:
ffmpeg -framerate 24 -i frame%08d.png -c:v libx265 -preset ultrafast -x265-params lossless=1 -r 30 sintel_loseless.mp4

#Convert to h265:
ffmpeg -i sintel_loseless.mp4 -c:v libx264 -b:v 4000k -minrate 4000k -maxrate 4000k -bufsize 1835k sintel_264.mp4

#Convert to h264:
ffmpeg -i sintel_loseless.mp4 -c:v libx265 -b:v 4000k -minrate 4000k -maxrate 4000k -bufsize 1835k sintel_265.mp4
```

sequences in 1080p and 4K resolutions. Chroma subsampling of reference video is YUV 420. Scene is mostly static.

Figure 4.3: HoneyBee

4.1.2.3 ShakeNDry

ShakeNDry is video sequence of 300 frames. Video sequence can be downloaded on website of Ultra Video Group [35]. Used versions is 8-bit, YUV, RAW sequences in 1080p. Chroma subsampling of reference video is YUV 420. Scene contains a lot of independend movement that require high very bandwidth.

4.2 Test method

Decoder in following tests is always native decoder from FFMPEG. GPU decoding acceleration was not used for testing since not all GPUs have same support of codecs. Also currently no tested GPUs support decoding of YUV 444.

4.2.1 Testing data encoding

Reference video sequences are always in lossless formats. Commands for conversion can be found on appendix.

Folowing codecs will be tested:

- x.264 (H.264)

Figure 4.4: ShakeNDry

- nvenc264 (H.264)
- libvpx-vp9 (VP9)
- x.265 (HEVC)
- nvenc265 (HEVC)

4.3 Testing enviroment

Testing is done under operating system is Windows 10 Anniversary edition with Windows Subsystem for Linux. In test does not explicitly specifies GPU, then Nvidia Geforce GTX 1070 is used.

4.3.1 Hardware Specifications

CPU	Core i7 3770k
GPU	variable
RAM	Kingston 16GB (4x4GB) DDR3 1600MHz
PCI-e	3.0

Core i7 3770k Specs:

# of Cores	4
# of Threads	8
Processor Base Frequency	3.50 GHz
Max Turbo Frequency	3.90 GHz

GPU Engine Specs:

GPU specification	TITAN X	MSI GTX 1070	GTX 680
Architecture	Maxwell	Pascal	Kepler
CUDA Cores	3072	1920	1536
Base Clock (MHz)	1000	1607	1006
Boost Clock (MHz)	1075	1797	1058
Memory Speed	7.0 Gbps	8 Gbps	6.0 Gbps
Memory Config	12 GB	8 GB	2048MB
Memory Interface	384-bit GDDR5	256-Bit GDDR5	256-bit GDDR5
Memory Bandwidth	336.5 GB/s	256 GB/s	192.2 GB/s
Bus Support	PCI-e 3.0	PCI-e 3.0	PCI-e 3.0
Graphics Power (W)	250 W	150 W	195 W

4.4 Tests

Measuring of transcoding times was done under powershell using Measure-Command. All data are stored on SSD to minimalize impact of reading and writing data to persistent storage.

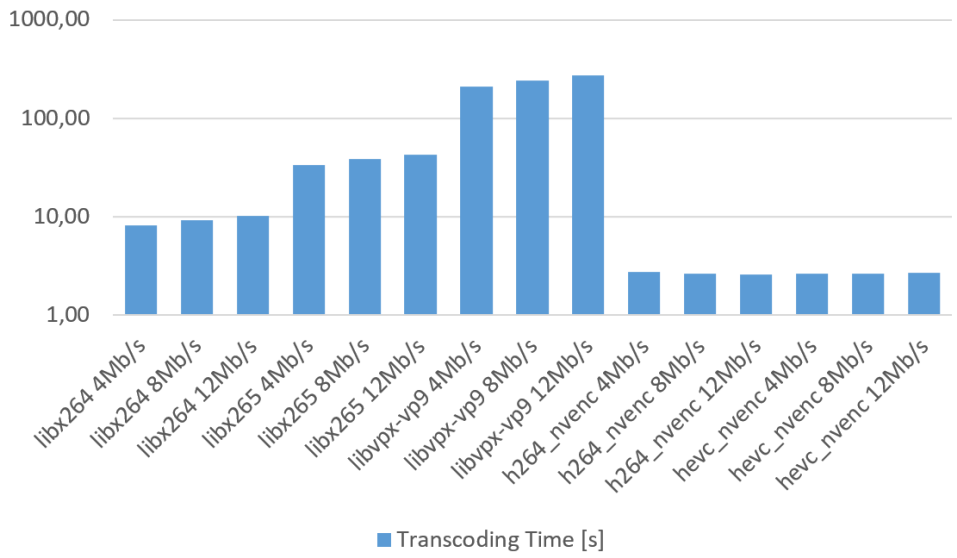
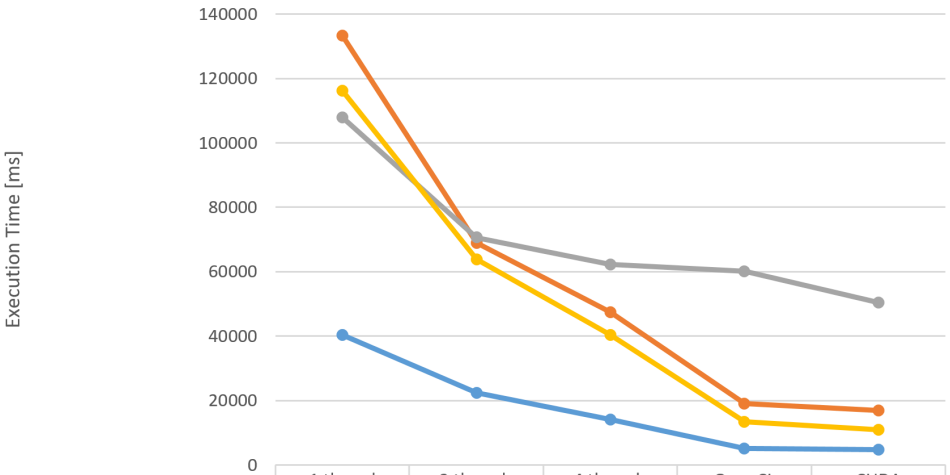


Figure 4.5: Transcoding times of codecs

Transcoding times (figure 4.5) of software HEVC codec are four times higher compare to H.264. On the other hand, encoding times with hardware acceleration of H.264 and HEVC does not change. Encoding of VP9 took six times

more compared to HEVC. Main reason for slow encoding times of VP9 is only one thread implementation of encoding.

Time measuring of video quality evaluation in implemented directly inside developed application. Time measuring is done without initialization and deinitialization. So for example runtime kernel compilations (if done in initialization) does not affect result.



	1 thread	2 threads	4 threads	OpenCL	CUDA
HoneyBee 1920x1080	40358	22425	14161	5184	4726
HoneyBee 3840x2160	133265	69017	47438	19128	16862
sintel yuv 444	107851	70621	62220	60196	50339
square (high compression)	116178	63880	40440	13429	11013

Figure 4.6: Multiple sequences test

Video Quality Evaluation times massively differ based on selected video sequence as can be seen on figure 4.6. Video sequences with 4K resolution require to evaluate four times more data. For that reason, evaluation of 4K video takes almost four times more time. Evaluation of video sequence in color space yuv444 took three times more time compared to yuv420 even though same data bandwidth need to be compared.

Both PSNR and SSIM algorithms scaled almost linearly with number of available cores on CPU as can be seen on figure 4.7. That was expected since CPU evaluations are not parrallized, but multiple frames are executed at the same time on different threads.

As for GPU implementations both CUDA and OpenCL achieve GPU utilization of up to 30%.

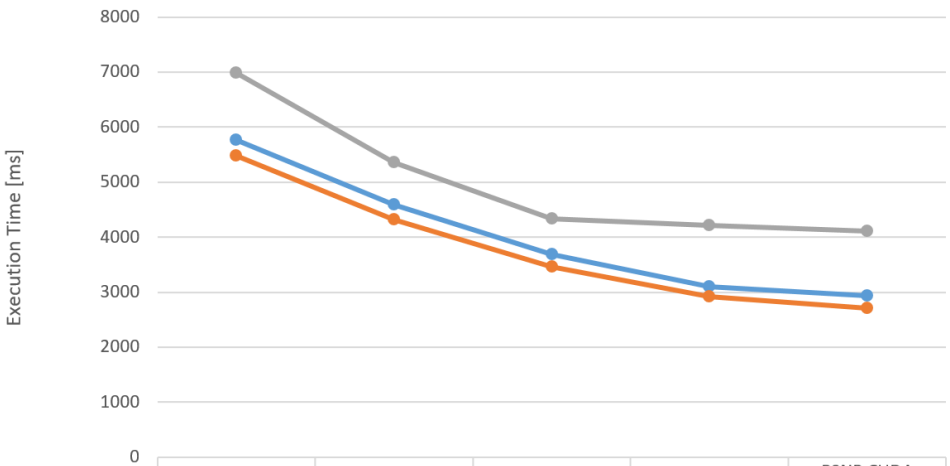
	1 thread	2 threads	4 threads
PSNR	39616	20961	13189
SSIM	183730	90607	51796

Figure 4.7: ShakeNDry Execution time (cpu)

Results in figure 4.8 shows that OpenCL implementations are arround 20% slower compared to CUDA. Asynchronous evaluation in multiple streams helped to decrease even more, since it was possible to interspace data copying and kernel evaluation. Evaluation only on Luma channel significantly decreased size of evaluated data. Last optimalization with registread memoty increased transfer speed between host and device from 3840 MB/s to 11755 MB/s.

All tested graphic cards are connected by PCIe 3. For that reason copy speeds are same. As we can see in figure 4.8 fastest execution was on Titan X. Decoding speeds are also constant since all video sequences are decoded by CPU. But kernel evaluation takes less time on Titan X. That is thanks to higher global memory bandwidth and more CUDA cores of Titan X. Execution performance does not scale linearly with peak performance of GPU since PCIe 3 bandwidth limits performance. Thanks to that mainstream GPUs like GTX 1070 are much more cost effective.

In analysis we've suggested that quality evaluation can be done on entire RGB scale or just Luma channel. Results on both SSIM (figure 4.9 for Luma



	PSNR OpenCL	PSNR CUDA	PSNR CUDA Streams	PSNR CUDA Streams Luma	PSNR CUDA Streams Luma Registered
GTX 1070	5762	4584	3690	3105	2932
GTX Titan X	5485	4321	3457	2917	2701
GTX 680	6984	5359	4339	4217	4103

Figure 4.8: ShakeNDry Execution time (gpu)

and figure 4.10 for RGB) and PSNR (figure 4.11 for Luma and figure 4.12 for RGB) show, that both evaluation methods correlate and thus evaluation on Luma can decrease memory bandwidth to one third compared to RGB.

Results of PSNR with SSIM seems to correlate on most of tested codecs with exception of VP9. This effect can be seen on both RGB and Luma.

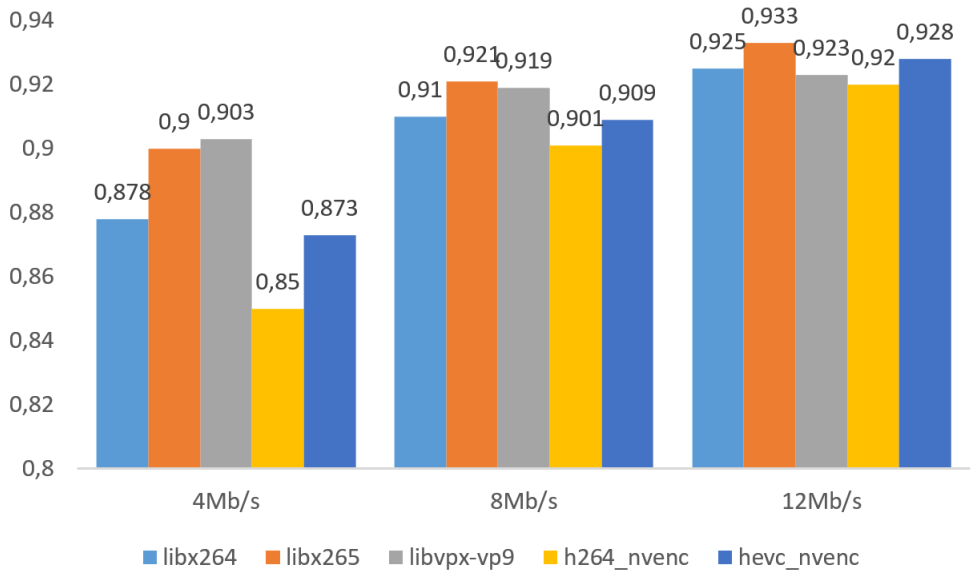


Figure 4.9: ShakeNDry Calculated SSIM on Luma

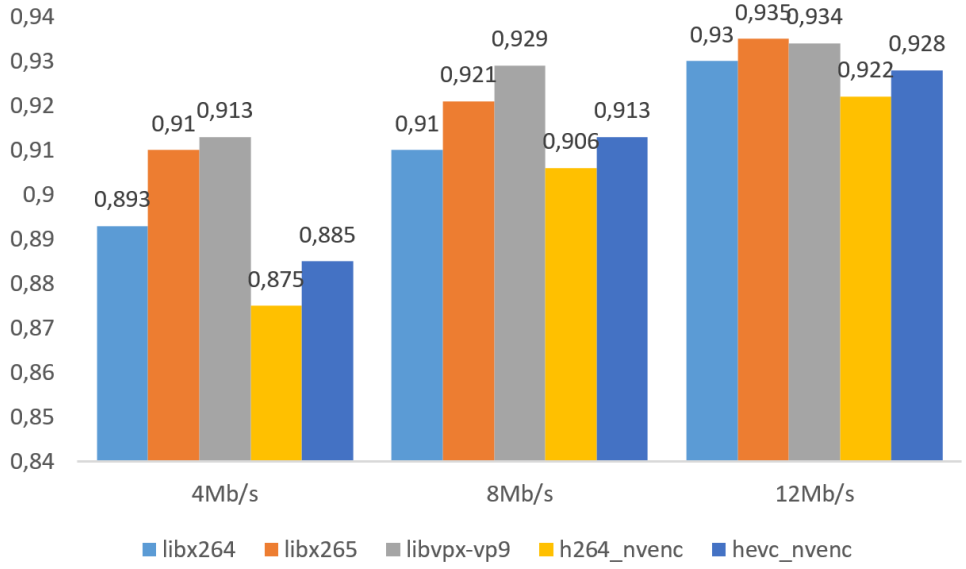


Figure 4.10: ShakeNDry Calculated SSIM on RGB

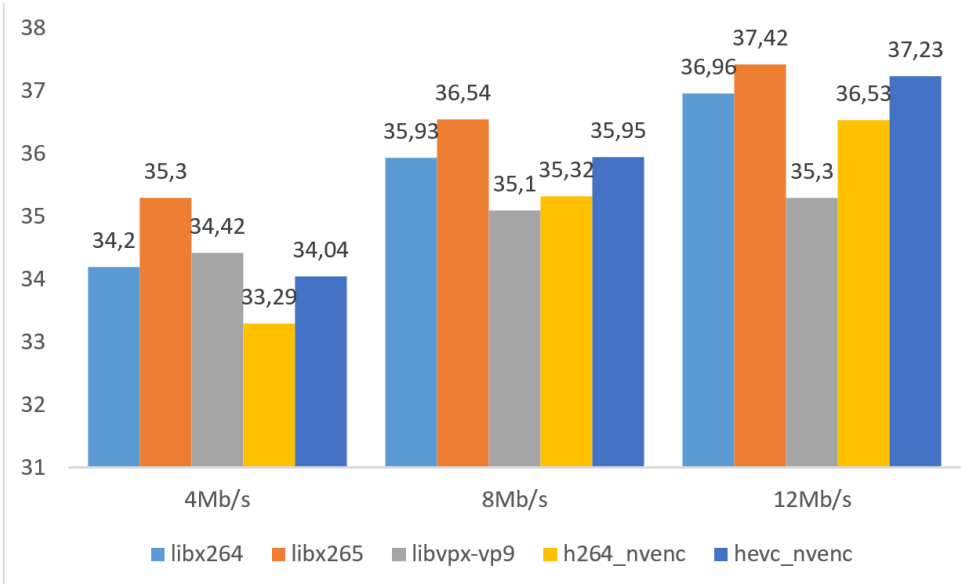


Figure 4.11: ShakeNDry Calculated PSNR on Luma

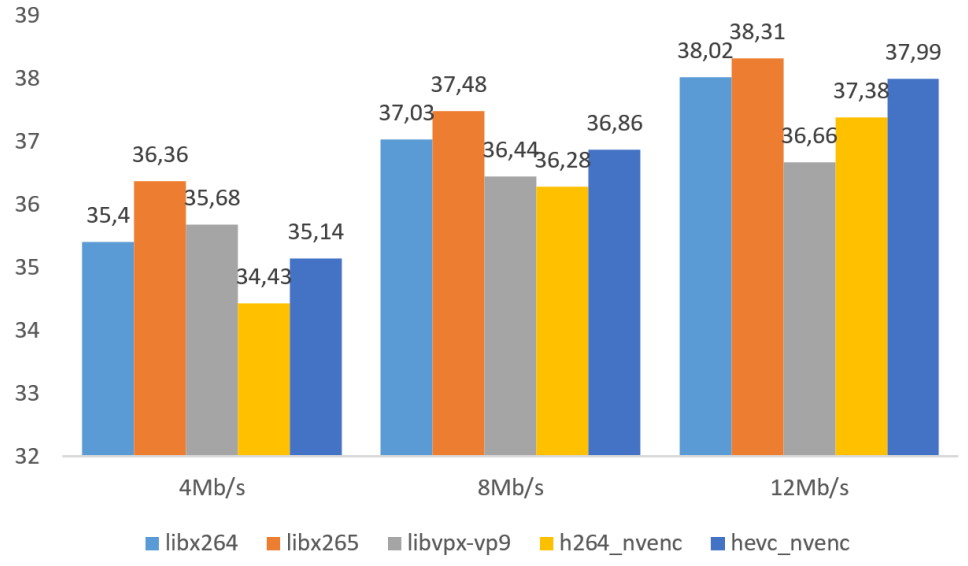


Figure 4.12: ShakeNDry Calculated PSNR on RGB

Conclusion

Main focus of this master thesis was to describe possible ways to evaluate video quality of video streams encoded using cutting edge technologies like HEVC.

In first chapter I've described ways of saving video sequences into data streams. I've identified standards that are most commonly used to compress video and described video quality evaluation algorithms.

In second chapter I've described available technologies enabling computations on graphical processing units. Main focus was on CUDA and OpenCL. I've discussed advantages and disadvantages of both technologies.

In next chapter I've created video quality evaluation application with architecture that can be easily extended. That allowed to easily add multiple comparison algorithms programed with multiple technologies. I've implemented video quality evaluation algorithms like PSNR and SSIM described in first chapter. First implementations were CPU only, then I've tried to utilize GPU using CUDA and OpenCL.

CUDA provided more tools and offered higher performance compared to OpenCL. OpenCL algorithms took longer to program, since OpenCL required a lot more steps to successfully run code on GPU. Situation might change a little in case NVidia would provide support for OpenCL 2.0 which was released in 2013. Even though I've considered OpenCL more difficult to program with I must admit that most of additional steps compared to CUDA are there for a good reason and CUDA does not requires they only because it supports hardware specific devices.

Expected results of speedup on GPU differ dramatically based on tested video sequence. Video sequences, that could not be decoded using hardware,

like HECV with YUV444 coding, showed only 10% speedup compared to multithread execution on CPU since bottleneck was in long decoding times on CPU. On the other hand, video sequences that could be decoded directly on GPU, were evaluated up to four times faster.

Lastly I've used programmed application to test most common video compression standards. In test were codecs like H.264 (x.264/nvenc264), H.265 (x.265/nvenc265) and VP9 (libx-vp9). Result looks promising. Codec H.265 (also called HEVC) seems to be almost 50 % more efficient than H.264. Software implementations (x.264 and x.265) of codecs offers better compressing ratio compared to hardware compression (nvenc264 and nvenc265), but differences seems to be not that big. With much higher speeds of hardware encoders I recommend nvenc265 for encoding end user content. Professionals might still use software implementations, but I would recommend usage of H.264 standard only in case of need of backward compatibility.

Results confirmed that evaluation can be done just on Luma channel of video sequence. That lowers memory bandwidth needed to evaluate video to just one third compared to RGB color space.

Tests also confirmed correlation between results of PSNR and SSIM. Both algorithms almost scaled linearly with number of available cores on CPU. As for GPU implementations both CUDA and OpenCL achieved GPU utilization of up to 30 percent. OpenCL was a little behind of CUDA performance, since it could not use architecture specific instructions.

Developed program will be available under MIT license on GitHub. As for future development I would like to integrate developed comparators into FFMPEG, rather than continue development as standalone application. FFMPEG is the leading multimedia framework for decoding and encoding. I've worked with FFMPEG framework even before I've started writing this thesis and I've had very good experience with it. I've integrated FFMPEG into platforms by Apple like iOS and tvOS and even contributed with bug reports and fixes. Community around FFMPEG is very active and many developers could benefit from GPU accelerated video evaluation or even minor things like GPU accelerated conversion of YUV to RGB. Since in developed application I already use structures defined by FFmpeg integration process should not take very long.

Bibliography

[1] What is deinterlacing? The best method to deinterlace movies. Dostupné z: http://www.100fps.com/

[2] Programming Guide :: CUDA Toolkit Documentation. Dostupné z: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[3] Sales and Marketing Tools. Dostupné z: http://www.nvidia.com/object/partner-sales-marketing-tools.html

[4] NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Dostupné z: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[5] NVIDIA's Next Generation CUDA Compute Architecture: Kepler. Dostupné z: hhttps://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[6] Khronos SIGGRAPH Asia November 2015. Dostupné z: https://www.khronos.org/assets/uploads/developers/library/2015-sigasia/SIGGRAPH-Asia_Nov15.pdf

[7] The OpenCL Specification version 1.2, Document Revision: 19. Dostupné z: https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf

[8] (Intel), m.-r.-r.: Intel® Xeon Phi™ Core Micro-architecture. Mar 2016. Dostupné z: https://software.intel.com/en-us/articles/intel-xeon-phi-core-micro-architecture

[9] Open Source HEVC Codec Projects. Dostupné z: https://www.parabolaresearch.com/blog/2014-09-12-open-source-hevc-codec-projects.html

[10] ITU-T H.265. http://www.itu.int/ITU-T/recommendations/rec.aspx?rec=11885, (Accessed on 03/29/2016).

[11] Branden Lambrecht, C. J. v. d.: *Vision models and applications to image and video processing*. Kluwer Academic, 2001.

[12] gnu.org. Dostupné z: https://www.gnu.org/home.en.html

[13] The 3-Clause BSD License. Dostupné z: https://opensource.org/licenses/BSD-3-Clause

[14] ISO/IEC 14496-2:2004. Dostupné z: http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=39259

[15] Tsbmail: H.264 : Advanced video coding for generic audiovisual services. Dostupné z: https://www.itu.int/rec/T-REC-H.264-201610-I/en

[16] ISO/IEC 14496-10:2012. Dostupné z: http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=61490

[17] ISO/IEC DIS 23008-2 - Information technology – High efficiency coding and media delivery in heterogeneous environments – Part 2: High efficiency video coding. http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=69668, (Accessed on 03/29/2016).

[18] Richardson, I. E. G.: H.264 and MPEG-4 video compression: video coding for next generation multimedia. 2008.

[19] A Statistical Evaluation of Recent Full Reference Image Quality Assessment Algorithms - IEEE Xplore Document. Dostupné z: http://ieeexplore.ieee.org/document/1709988/

[20] Reduced-Reference Image Quality Assessment by Structural Similarity Estimation - IEEE Xplore Document. Dostupné z: http://ieeexplore.ieee.org/document/6193206/

[21] No-reference perceptual quality assessment of JPEG compressed images - IEEE Xplore Document. Dostupné z: http://ieeexplore.ieee.org/abstract/document/1038064/

[22] Image quality assessment: from error visibility to structural similarity - IEEE Xplore Document. Dostupné z: http://ieeexplore.ieee.org/abstract/document/1284395/

[23] Nikolay Ponomarenko homepage - PSNR-HVS-M download page. Dostupné z: http://www.ponomarenko.info/psnrhvsm.htm

[24] White Paper: Advancing To Multi-Scale SSIM. Dostupné z: http://videoclarity.com/videoqualityanalysiscasestudies/wpadvancingtomulti-scalessim/

[25] Covariance and Correlation. Dostupné z: http://www.math.uah.edu/stat/expect/Covariance.html

[26] Brweb: BT.500 : Methodology for the subjective assessment of the quality of television pictures. Dostupné z: http://www.itu.int/rec/R-REC-BT.500-13-201201-I

[27] Tsbmail: P.910 : Subjective video quality assessment methods for multimedia applications. Dostupné z: https://www.itu.int/rec/T-REC-P.910/en

[28] CUDA GPUs. Oct 2016. Dostupné z: https://developer.nvidia.com/cuda-gpus

[29] NVIDIA's Next Generation CUDA Compute Architecture: Maxwell. Dostupné z: http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF

[30] NVIDIA's Next Generation CUDA Compute Architecture: Pascal. Dostupné z: http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf

[31] Rdp: ffmpeg-windows-build-helpers. Feb 2017. Dostupné z: https://github.com/rdp/ffmpeg-windows-build-helpers

[32] Pracniques: further remarks on reducing truncation errors. Dostupné z: http://doi.acm.org/10.1145/363707.363723

[33] (1) What is CUB? Dostupné z: http://nvlabs.github.io/cub/

[34] NVIDIA VIDEO CODEC SDK. Jan 2017. Dostupné z: https://developer.nvidia.com/nvidia-video-codec-sdk

[35] 4K video sequences. Dostupné z: http://ultravideo.cs.tut.fi/

Appendix A

Codec conversion commands

Listing A.1: HoneyBee conversion example

```
1 Measure-Command {./ffmpeg -i ref/HoneyBee_3840x2160.
    ↪ mp4 -c:v libx264 -b:v 4M x264/
    ↪ HoneyBee_3840x2160_4M.mp4}
2 Measure-Command {./ffmpeg -i ref/HoneyBee_3840x2160.
    ↪ mp4 -c:v libx264 -b:v 8M x264/
    ↪ HoneyBee_3840x2160_8M.mp4}
3 Measure-Command {./ffmpeg -i ref/HoneyBee_3840x2160.
    ↪ mp4 -c:v libx264 -b:v 12M x264/
    ↪ HoneyBee_3840x2160_12M.mp4}
4 Measure-Command {./ffmpeg -i ref/HoneyBee_3840x2160.
    ↪ mp4 -c:v libx265 -b:v 4M x265/
    ↪ HoneyBee_3840x2160_4M.mp4}
5 Measure-Command {./ffmpeg -i ref/HoneyBee_3840x2160.
    ↪ mp4 -c:v libx265 -b:v 8M x265/
    ↪ HoneyBee_3840x2160_8M.mp4}
6 Measure-Command {./ffmpeg -i ref/HoneyBee_3840x2160.
    ↪ mp4 -c:v libx265 -b:v 12M x265/
    ↪ HoneyBee_3840x2160_12M.mp4}
7 Measure-Command {./ffmpeg -i ref/HoneyBee_3840x2160.
    ↪ mp4 -c:v libvpx-vp9 -b:v 4M vp9/
    ↪ HoneyBee_3840x2160_4M.mkv}
8 Measure-Command {./ffmpeg -i ref/HoneyBee_3840x2160.
    ↪ mp4 -c:v libvpx-vp9 -b:v 8M vp9/
    ↪ HoneyBee_3840x2160_8M.mkv}
9 Measure-Command {./ffmpeg -i ref/HoneyBee_3840x2160.
    ↪ mp4 -c:v libvpx-vp9 -b:v 12M vp9/
    ↪ HoneyBee_3840x2160_12M.mkv}
```

```
10 Measure-Command {./ffmpeg -i ref/HoneyBee_3840x2160.
   ↪ mp4 -c:v h264_nvenc -b:v 4M nvenc264/
   ↪ HoneyBee_3840x2160_4M.mp4}
11 Measure-Command {./ffmpeg -i ref/HoneyBee_3840x2160.
   ↪ mp4 -c:v h264_nvenc -b:v 8M nvenc264/
   ↪ HoneyBee_3840x2160_8M.mp4}
12 Measure-Command {./ffmpeg -i ref/HoneyBee_3840x2160.
   ↪ mp4 -c:v h264_nvenc -b:v 12M nvenc264/
   ↪ HoneyBee_3840x2160_12M.mp4}
13 Measure-Command {./ffmpeg -i ref/HoneyBee_3840x2160.
   ↪ mp4 -c:v hevc_nvenc -b:v 4M nvenc265/
   ↪ HoneyBee_3840x2160_4M.mp4}
14 Measure-Command {./ffmpeg -i ref/HoneyBee_3840x2160.
   ↪ mp4 -c:v hevc_nvenc -b:v 8M nvenc265/
   ↪ HoneyBee_3840x2160_8M.mp4}
15 Measure-Command {./ffmpeg -i ref/HoneyBee_3840x2160.
   ↪ mp4 -c:v hevc_nvenc -b:v 12M nvenc265/
   ↪ HoneyBee_3840x2160_12M.mp4}
```

Appendix B

List of Abbreviations

ALU Arithmetic Logic Unit

AMD Advanced Micro Devices

API Application Program Interface

APU Accelerated Processing Unit

ARM Advanced RISC Machine

AVC Advanced Video Coding

BMC Block Motion Compensation

BSD Berkeley Software Distribution

CPU Central Processing Unit

CRT Cathode Ray Tube

CTU Czech Technical University

CUDA Compute Unified Device Architecture

DCT Discrete Cosine Transform

DDR Double Data Rate

DRAM Dynamic Random Access memory

FFMPEG Fast Forward MPEG

FIT Faculty of information Technology

FMA Fused Multiply Add

FPU Floating Point Unit

GBR Green, Blue and Red

GCC GNU Compiler Collection

GDDR Graphics Double Data Rate

GFLOP GigaFlop

GHZ Gigahertz

GNU General Public License (GNU GPL or GPL)

GPL General Public License

GPU Graphical Processing Unit

HBM High Bandwidth Memory

HDR High Dynamic Range

HEVC High Efficiency Video Coding

HIP Heterogeneous-compute Interface for Portability

HVS Human Visual System

IBM International Business Machines

IEC International Electrotechnical Commission

IPS In-Plane Switching

ISO Organization for Standardization

ITU International Telecommunication Union

JCT-VC The Joint Collaborative Team on Video Coding

JIT Just-In-Time

JPEG Joint Photographic Experts Group

LCD Liquid Crystal Display

LGPL GNU Lesser General Public License

MKL Math Kernel Library

MPEG Moving Picture Experts Group

MSE Mean Squared Error

OLED Organic Light Emitting Diodes

OPENACC Open Accelerators

OPENCL Open Computing Language

OPENMP Open Multi-Processing

PCI Peripheral Component Interconnect

PDF Portable Document Format

PGI Premiere Global Services

PNG Portable Network Graphics

PSNR Peak Signal-To-Noise ratio

PTX Parallel Thread eXecution

RAM Random Access memory

RGB Read, Green, Blue

SDK Software Development Kit

SFU Special Function Unit

SHFL Shuffle instruction

SIMD Single Instruction Multiple Data

SIMT Single Instruction Multiple Thread

SMM Streaming Multiprocessor Maxwell

SMX Streaming Multiprocessor (Kepler)

SSIM Structural Similarity

VCEG Video Coding Experts Group

YUV Luma (Y) and two chrominance (UV)

Appendix C

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `exe` ..... adresář se spustitelnou formou implementace
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `thesis.ps` ..... text práce ve formátu PS