SVEUČILIŠTE U ZAGREBU
GRAFIČKI FAKULTET

LUKA KEDMENEC

DCT BASED WATERMARKING METHOD FOR SOCIAL MEDIA

DIPLOMSKI RAD

Sveučilište u Zagrebu
Grafički fakultet

DIPLOMSKI RAD

DCT based watermarking method for social media

Mentor: Student:

doc.dr.sc. Ante Poljičak Luka Kedmenec

Zagreb, 2015

I would like to thank my mentors Ante Poljičak, Guillerno Botella Juan and Carlos Garcia Sanchez for the hard work, effort, guidance and support throughout this work. I would also like to thank my family and friends for a constant care and support and undoubtful faith in my work.

Abstract

In this memory a steganographic method for image protection on social network is developed. The method hides additional information such as a logo or serial number. Uploading the images on the social network has induced image degradation on the original images. Based on the analysis of this attack a method is developed based on the discrete cosine transform - DCT where additional information is hidden by modifying the coefficients in the cosine domain. The coefficient is chosen after thorough analysis based on Peak Signal to Noise Ratio (PSNR), Structuram Similarity Index (SSIM) and bit error rate (BER). It is concluded that the coefficients representing middle range of frequencies in the cosine domaine is best suited for modifications. The reason for this is that the modification of these frequencies does not degrade the picture enough for a human eye to see it but are still able to withstand various types of image attacks such as scaling, rotation, cropping and JPEG compression. The method was evaluated and it has shown that bit error rate (BER) for different attacks was below 1%. The method is implemented on 4 different hardware platforms using OpenCL and then tested for performance in speed and complexity of the implementation. Acceleration was based on creating a kernels for most time consuming functions DCT and iDCT.

Key words

Steganography, DCT, OpenCL, Social networks, hardware implementation, hardware acceleration

Sažetak

U ovom je radu razvijena steganografska metoda za zaštitu slika na društvenim mrežama. Metoda ugrađuje informacije poput loga ili serijskog broja. Prijenosom slika na društvenu mrežu dolazi do neželjene degradacije izvorne slike. Na temelju analize napada socijalne mreže razvijena je metoda koja se temelji na diskretnoj kosinusnoj transformaciji – DCT u kojoj se dodatne informacije skrivaju modificiranjem koeficijenata u kosinusnoj domeni. Koeficijent je izabran nakon temeljite analize odnosa šuma i signala (PSNR), indeksa strukturalne sličnosti (SSIM) i učestalost pogrešnih bitova (BER). Koeficijenti srednje frekvencije su se pokazali kao idealni za modifikaciju i ugradnju dodatnih informacija. Modificiranjem tih frekvencija ne degradiramo sliku u toj mjeri da je promjena vidljiva ljudskom oku, a još uvijek je u stanju izdržati različite vrste napada poput skaliranja, šuma, JPEG kompresije, zamućenja i ostalih. Metoda je testirana i rezultati učestalosti pogrešnih bitova (BER) kod ekstrakcije za različite napade su ispod 1%. Ova metoda je implementirana i testirana na 4 različite hardverske platforme s ciljem ubrzavanja izvršavanja koda koristeći OpenCL. Dobivena ubrzanja su postignuta izgradnjom kernela za vremenski najzahtjevnije funkcije DCT i iDCT.

Ključne riječi

Steganografija, DCT, OpenCL, Društvene mreže, hardverska implementacija, hardversko ubrzanje

Contents

1. Introduction

1.1. Steganography, Cryptography and Watermarking

Steganography is the art or practice of concealing a message, image, or file within another message, image or file. The first record of steganography use dates back to 440 BC. It was a message written on the wooden back of a wax tablet to warn of a Greece attacking. These were a primitive methods of steganography that are not used today. There were other methods such as invisible ink or creating a pattern that you would put on top of the text to reveal a hidden message.

Modern steganography started with the greater use of computers in 1985. Since then many methods were developed but most of the methods used in an early steganography times are not used today. Todays methods are ususally the ones based on transformation between spacial and other domains such as cosine, wavelet or Furier where the change is made on certain coefficients to hide the data. The earlier methods such as LSB or least significant bit are used less because of their easy detection and easy extraction of hidden data.

Cryptography is the science of writing in secret code and is an ancient art. The first documented use of cryptography in writing dates back to circa 1900 B.C. when an Egyptian scribe used non-standard hieroglyphs in an inscription. Some experts argue that cryptography appeared spontaneously sometime after writing was invented, with applications ranging from diplomatic missives to war-time battle plans. Cryptography prior to the modern age was effectively synonymous with encryption, the conversion of information from a readable state to apparent nonsense. The originator of an encrypted message shared the decoding technique needed to recover the original information only with intended recipients, thereby precluding unwanted persons from doing the same. New forms of cryptography came soon after the widespread development of computer communications. In data and telecommunications, cryptography is necessary when communicating over any untrusted medium, which includes just about *any* network, particularly the Internet.

Watermarking is a method typicaly used to identify ownership of the copyright and is used on different media such as images, sound or video. There are different types of digital watermarking methods, those visible to a human eye and the ones hidden from a human eye. They have to be robust to the attacks in order for them to preserve data. Like traditional watermarks, some digital watermarks are only perceptible under certain conditions, i.e. after using some algorithm, and imperceptible otherwise. The best example of traditional watermark is the banknote where most of them have a watermark that is exposed only to the backlight[1]. Both steganography and digital watermarking employ steganographic techniques to embed data covertly in noisy signals. But whereas steganography aims for imperceptibility to human senses, digital watermarking tries to control the robustness as top priority.

1.2. High level paradigms for parallel heterogeneus systems

Heterogeneous computing refers to systems that use more than one kind of processor. These are systems that gain performance not just by adding the same type of processors, but by adding dissimilar processors, usually incorporating specialized processing capabilities to handle particular tasks. Heterogeneous computing platforms can be found in every domain of computing, from high-end servers and high-performance computing machines all the way down to low-power embedded devices including mobile phones and tablets. The main problem of these systems is that different processors have different instruction set architectures leading to binary incompatibility and thus inproper work. Heterogeneous computing systems present new challenges not found in typical homogeneous systems. The presence of multiple processing elements raises all of the issues involved with homogeneous parallel processing systems, while the level of heterogeneity in the system can introduce non-uniformity in system development, programming practices, and overall system capability.[2] Areas of heterogeneity can include ISA or instruction set architecture, ABI or application binary interface, API or application programming interface, Low-Level Implementation of Language Features, Memory Interface and Hierarchy, Interconnect and Performance. Heterogeneous System Architecture (HSA) systems utilize multiple processor types (typically CPUs and GPUs), usually on the same silicon die, to give you the best of both worlds: GPU processing, apart from its well-known 3D graphics rendering capabilities, can also perform mathematically intensive computations on very large data sets, while CPUs can run the operating system and perform traditional serial tasks.

There are numeroues heterogeneous computing platforms nowadays, from a smallest and simplest ones in the low-power embedded devices including mobile phones and tablets all the way to the high performance servers and high-end computing machines. Most of todays laptops, desktop computers or tablets use some kind of heterogeneous computing platforms, the most common for laptops and desktop computers are Intels Ivy, Sandy and Haswell CPU's and AMD's APU's and for the tablets and smartphones Nvidia Tegra, Samsung Exynos and Apple A series. Playstation also uses a heterogeneous computing platform like Emotion Engine for Playstation 2 or Spurs Engine, which is a variant of the IBM Cell processor, in Playstation 3 and Playstation 4 that has AMD APU.

In the last 5 years, most of new desktop computers had multicore processors, with dual-core and even quad core processors entering the mainstream of affordable computing. The increased number of cores and multicore processing presented some challenges of its own. The extra cores and cache memory required to fuel their instruction pipelines lead to increased processor size and a greater power consumption.

On the other hand multi-core era also introduced some interesting developments in GPUs, which were growing in sophistication and complexity, spurred on by advances in semiconductor technology. GPUs have vector processing capabilities that enable them to perform parallel operations on very large sets of data and to do it at much lower power consumption compared to the serial processing of similar data sets on CPUs. This is what allows GPUs to drive capabilities such as incredibly realistic, multiple display stereoscopic gaming. Although they were first presented for their ability to improve 3D graphics performance by offloading graphics from the CPU, they became increasingly attractive for more general purposes, such as addressing data parallel programming tasks.

2. Statement of present approach

2.1. Steganography models

With the growth of computer power and storage and advancements in digital communication technology, the difficulties in ensuring individuals' privacy become increasingly challenging. Every person has different privacy levels that they apreciate. Various methods have been investigated and developed to protect personal privacy.The most obvious method is encryption and then comes steganography. Encryption lends itself to noise and is generally observed while steganography is not observable. The term steganography refers to the art of covert communications [1]. Steganography's aim is to make the secret communication undetectable, that is, to hide the presence of the secret message. It modifies the carrier in an imperceptible way only so that it reveals nothing neither the embedding of a message nor the embedded message itself. The recent development of the Internet has brought new attention to steganography. The interest in steganography has been enhanced recently by the emergence of commercial espionage and the growing concerns about homeland security due to terrorism. The purpose of steganography is therefore to hide a secret message in a carrier. With the arrival of the digital era and the generalized usage of the Internet and email for the exchange of files, digital covers such as audio, image and video files have become the most obvious choices. This is partly due to their wide spread use, but also because this type of media usually includes a random noise component in which the secret message may be easily hidden. For decades people strove to develop innovative methods for secret communication.

Digital steganography started with the boost in computer power, the internet and with the development of digital signal processing, information theory and coding theory. In the realm of this digital world steganography has created an atmosphere of corporate vigilance that has spawned various interesting applications, thus its continuing evolution is guaranteed. Steganography is employed in various useful applications, such as advanced data structures, medical imagery, strong watermarks, military agencies, intelligence agencies, document tracking tools, document authentication, general communication, digital elections and electronic money, radar systems and remote sensing. Individuals' details are embedded in their photographs in smart IDs and identity cards.

There are two different types of steganography, spatial or transform domain. In general, steganographic algorithms rely on the replacement of some noise component of a digital object with a pseudorandom secret message [4].

2.1.1. Spatial Methods

There are many versions of spatial steganography and all directly change some bits in the image pixel values in hiding data. Least significant bit (LSB)-based steganography is one of the simplest techniques that hides a secret message in the LSBs of pixel values without introducing many perceptible distortions. Changes in the value of the LSB are imperceptible for human eyes. Spatial domain techniques are broadly classified into:

1. Least significant bit (LSB)

2. Pixel value differencing (PVD)

3. Edges based data embedding method (EBE)

4. Random pixel embedding method (RPE)

5. Mapping pixel to hidden data method

6. Labeling or connectivity method

7. Pixel intensity based method

8. Texture based method

9. Histogram shifting methods [5]

Main advantages of spatial domain LSB technique is that there is less chance for degradation of the original image and more information can be stored in an image but with the big disadvantages of low robustnes and the fact that the hidden data can be lost with image manipulation and easily destroyed by simple attacks.

Because of the simplicity of these methods they are used less than newer and better methods in a transform domain. There are many softwares that can easily retrieve hidden data if the LSB steganography method is used.

2.1.2. Transform based methods

Transform based methods use some kind of transformation to spread the hidden information onto a bigger area which makes data resist to attacks such as compression, cropping or image processing much better than LSB approach. This provides an enhanced security level to steganography method and lead to the development of algorithms. The most comonly used transform based methods are the ones in cosine domain, DCT, wavelet, DWT, and Fourier, DFT or FFT. The main purpose of those kind of steganography methods is to spread the information onto a bigger area.

2.1.2.1. DCT

The Discrete Cosine Transform (DCT) transforms the image from spatial domain to frequency domain. It separates the image into spectral sub-bands with respect to its visual quality, i.e. high, middle and low frequency components.

Most of the steganographic methods change some values in a middle frequencies in the 8x8 blocks because that way they do not degrade the image quality but still have the robust implementation. DCT transforms image from spacial domain into 8x8 blocks of frequency domain. They calculate those values using a 2D DCT-II algorithm. Figure 1 shows the coefficients and their spread in 8x8 block.

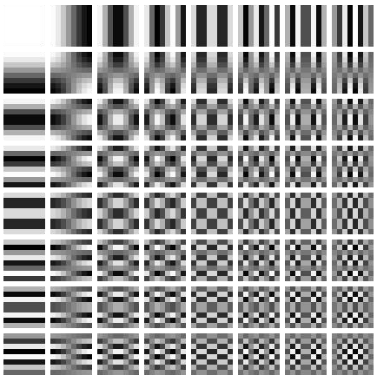


Figure 1. 8x8 block with DCT coefficients

2.1.2.2. DFT

Discrete Fourier transformation is very similar to the DCT and is used in almost the same way as the DCT. After the image is transformed from spatial domain into a frequency domain using discrete Fourier tranformation you get a differet frequency spread. Here the lowest frequencies are in the center of the block and the higher are towards the corners just as depicted in figure 2.

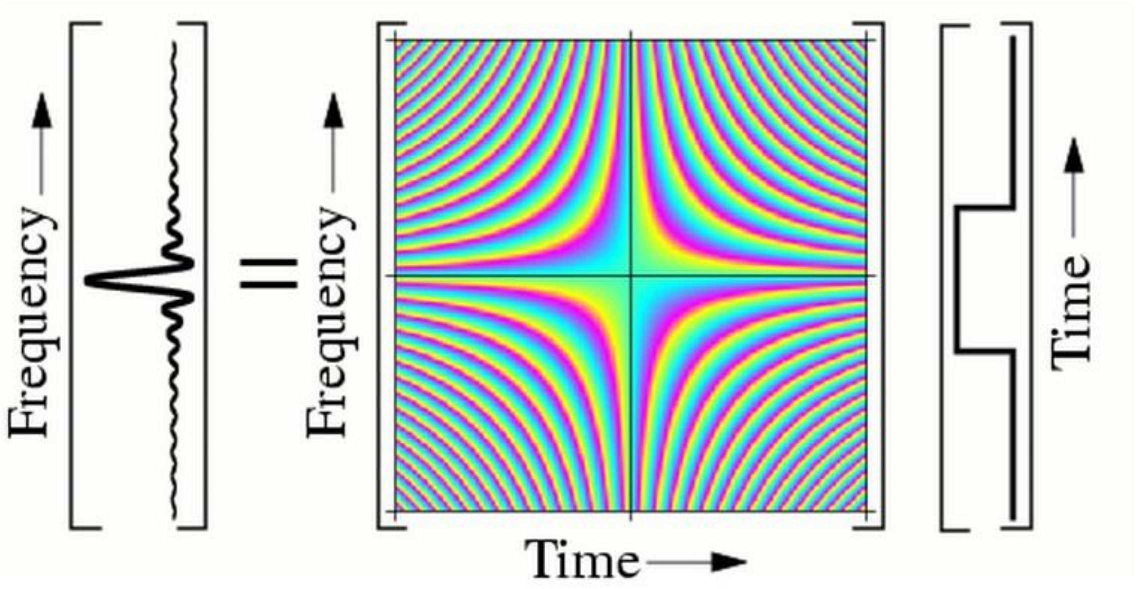


Figure 2. Frequency spread of a Fourier transformation

The same thing applies to the modifiying the values of a coefficients in Fourier tranformation. The lower the frequency the higher the artifacts. It is known that the steganography methods based on DFT are more robust to attacks such as rotation than the methods that use a DCT tranformation [5]. Despite those advantages the biggest problem is the inverse discrete Fourier transformation because of its imaginary and real numbers that can sometimes create problems or inprecision with the tranformation back to the spatial domain.

2.1.2.3. DWT

Discrete wavelet tranformation is the third most comonly used transformation for frequency steganography or transform based steganography methods. It is a common transformation used in JPEG2000 compression. The big advantage over Fourier tranformation is that it captures both frequency and location information. The Figure 3 shows the discrete wavelet transformation in 3 steps. In the first step it is divided into 4 subbands. Subband LL1 represents the horizontal and vertical low frequency components of the image. Subband HH1 represents the horizontal and vertical high frequency components of the image. Subband LH1 represents the horizontal low and vertical high frequency components. Subband HL1 represents the horizontal high and vertical low frequency components. With every further step it separates into more subbands and makes it even more precise. If you want to hide information in the image you have to pick a subband with the right frequency where the change is not visible but the data is preserved[5] .

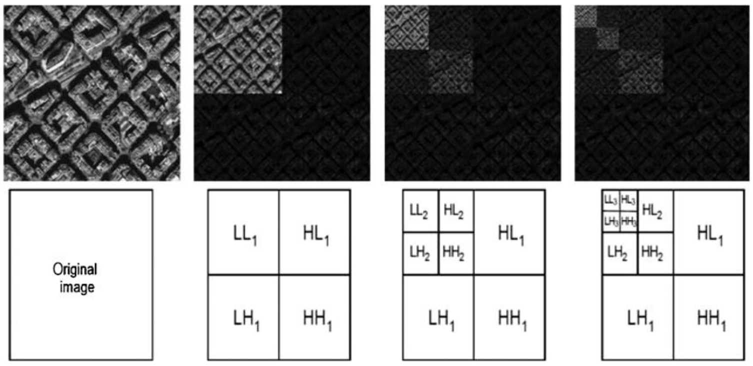


Figure 3. Discrete Wavelet transformation

2.2. Hardware accelerators

The main purpose of hardware acceleration is the use of computer hardware to perform some functions faster than it is possible in software running on a more general-purpose CPU. Examples of hardware acceleration include blitting acceleration functionality in graphics processing units (GPUs) and regular expression hardware acceleration for spam controling the server industry.

Processors are sequential and normaly instructions are executed one by one. There are many ways to improve performance, hardware acceleration is the one that is used in this memory. The main difference between hardware and software is concurrency, allowing hardware to be much faster than software. Computationally intensive software code is made for hardware accelerators. Depending upon granularity, hardware acceleration can vary from a small functional unit to a large functional block (like motion estimation in MPEG-2)[6].

The hardware that performs the acceleration, when in a separate unit from the CPU, is referred to as a hardware accelerator, or often more specifically as a 3D accelerator,cryptographic accelerator, etc. Those terms, however, are older and have been replaced with less descriptive terms like video card or network adapter.

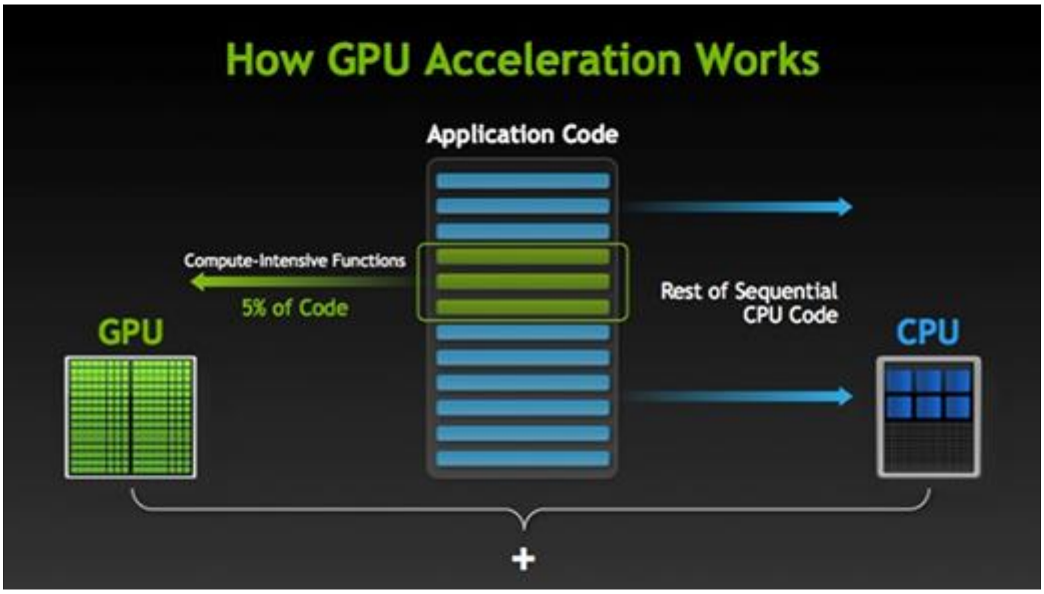
In the hierarchy of general-purpose processors such as CPUs, more specialized processors such as GPUs, fixed-function implemented on FPGAs, and fixed-function implemented on ASICs; there is a tradeoff between flexibility and efficiency, with efficiency increasing by orders of magnitude when any given application is implemented higher up that hierarchy[6].

2.2.1. GPGPUs

GPGPU stands for General-Purpose computation on Graphics Processing Units, also known as GPU Computing. Graphics Processing Units (GPUs) are high-performance many-core processors capable of very high computation and data throughput. At first the GPUs were specially designed for computer graphics and difficult to program, today’s GPUs are general-purpose parallel processors with support for accessible programming interfaces and industry-standard languages such as C. Many developers who port their applications to GPUs achieve acceleration much higher than the optimized CPU implementations. [7]

2.2.1.1. HOW GPUS ACCELERATE APPLICATIONS

GPU-accelerated computing offers unprecedented application performance by offloading compute-intensive portions of the application to the GPU, while the remainder of the code still runs on the CPU as in Figure 4. From a user's perspective, applications simply run significantly faster.



How GPU Acceleration Works

Application Code

Compute-Intensive Functions

5% of Code

Rest of Sequential CPU Code

Figure 4. How GPU acceleration works[8]

2.2.1.2. *CPU VERSUS GPU*

A simple way to understand the difference between a CPU and GPU is to compare how they process tasks. A CPU consists of a few cores optimized for sequential serial processing while a GPU has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously as depicted in Figure 5. GPUs have thousands of cores to process parallel workloads efficiently.

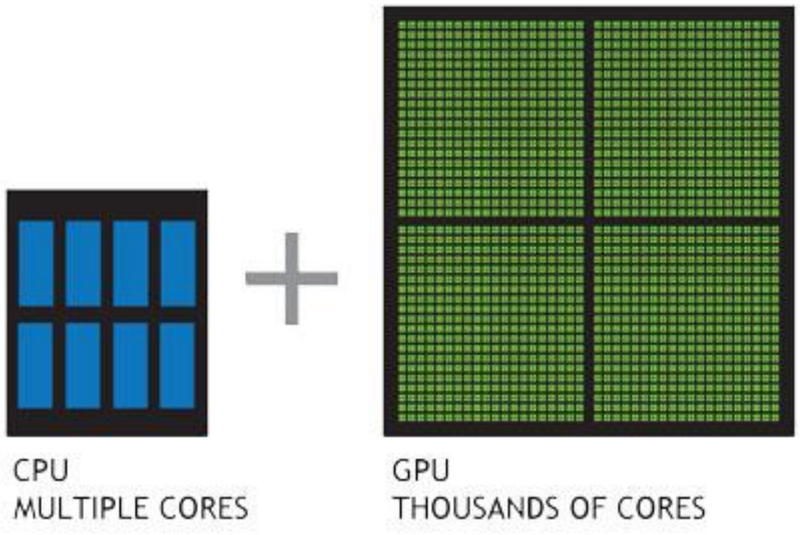


Figure 5. Difference between CPU and GPU

2.2.1.3. Implementation

Implementation of the GPGPUs is increasing over the years and has come to smaller devices such as smartphones and tablets and is not reserved for Desktop computers only.

2.2.1.3.1. Desktop computers

Any language that allows code running on the CPU to poll a GPU shader for return values, can create a GPGPU framework. OpenCL is the currently dominant open general-purpose GPU computing language, and is an open standard defined by the Khronos Group. OpenCL provides a cross-platform GPGPU platform that additionally supports data parallel compute on CPUs. OpenCL is actively supported on Intel, AMD, Nvidia and ARM platforms.

Programming standards for parallel computing include OpenCL (vendor-independent), OpenACC and OpenHMPP. Mark Harris is founder of GPGPU.org, and coined the term "GPGPU".

GPGPU processing is also used to simulate Newtonian physics by Physics engines, and commercial implementations include Havok Physics, FX and PhysX, both of which are typically used for computer and video games.

C++ Accelerated Massive Parallelism is a library that accelerates execution of C++ code by taking advantage of data-parallel hardware on GPUs.

2.2.1.3.2. Mobile computers

With the rise of processing power of mobile GPUs, general-purpose programming became available also on mobile devices running major mobile operating systems. Google Android 4.2 enabled running Renderscript code on the mobile device GPU. Apple introduced a proprietary Metal API for iOS applications, capable of executing arbitrary code through Apple's GPU compute shaders.

2.2.2. Field Programmable Gate Array (FPGA)

FPGAs are programmable semiconductor devices that are based around a matrix of Configurable Logic Blocks (CLBs) connected through programmable interconnects. As opposed to Application Specific Integrated Circuits (ASICs), where the device is custom built for the particular design, FPGAs can be programmed to the desired application or functionality requirements. Although One-Time Programmable (OTP) FPGAs are available, the dominant type are SRAM-based which can be reprogrammed as the design evolves[9].

FPGAs allow designers to change their designs very late in the design cycle– even after the end product has been manufactured and deployed in the field. A FPGA block structure is depicted in Figure 6.

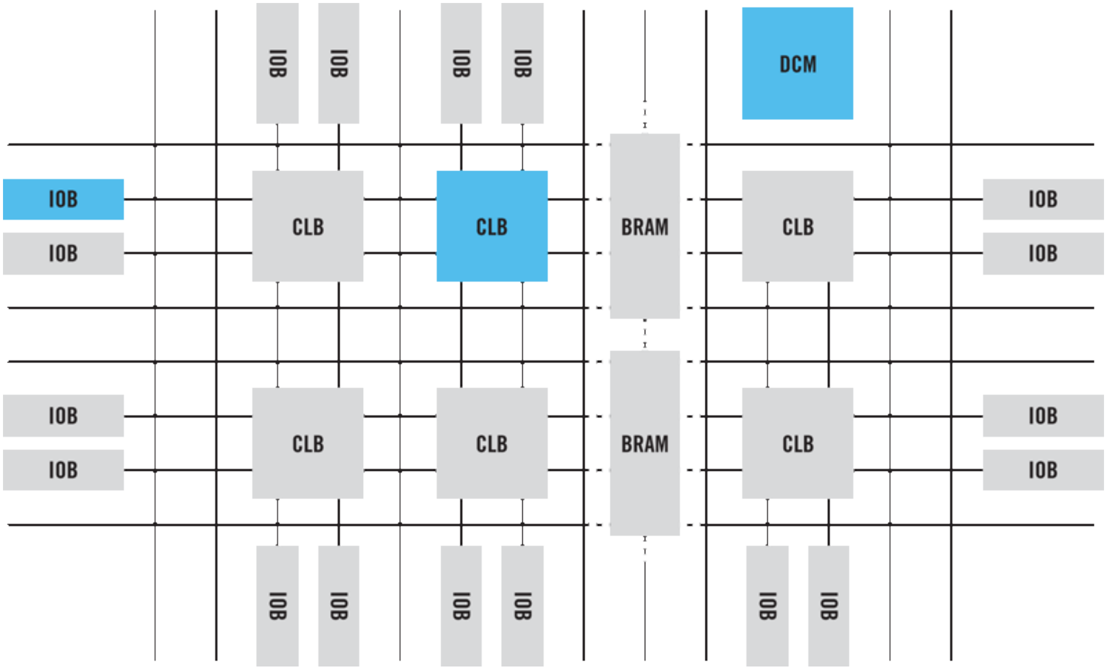


Figure 6. FPGA block structure

2.2.2.1. Common FPGA Features

FPGAs have evolved far beyond the basic capabilities present in their predecessors, and incorporate hard (ASIC type) blocks of commonly used functionality such as RAM, clock management, and DSP. the basic components in an FPGA are Configurable Logic Blocks (CLB), Basic SelectIO(IOB), Interconnect, Memory and Complete Clock Managament (DCM).

2.2.2.1.1. Configurable Logic Blocks

The CLB is the basic logic unit in a FPGA. Exact numbers and features vary from device to device, but every CLB consists of a configurable switch matrix with 4 or 6 inputs, some selection circuitry (MUX, etc), and flip-flops. The switch matrix is highly flexible and can be configured to handle combinatorial logic, shift registers or RAM. Figure 7 depicts the CLB structure.

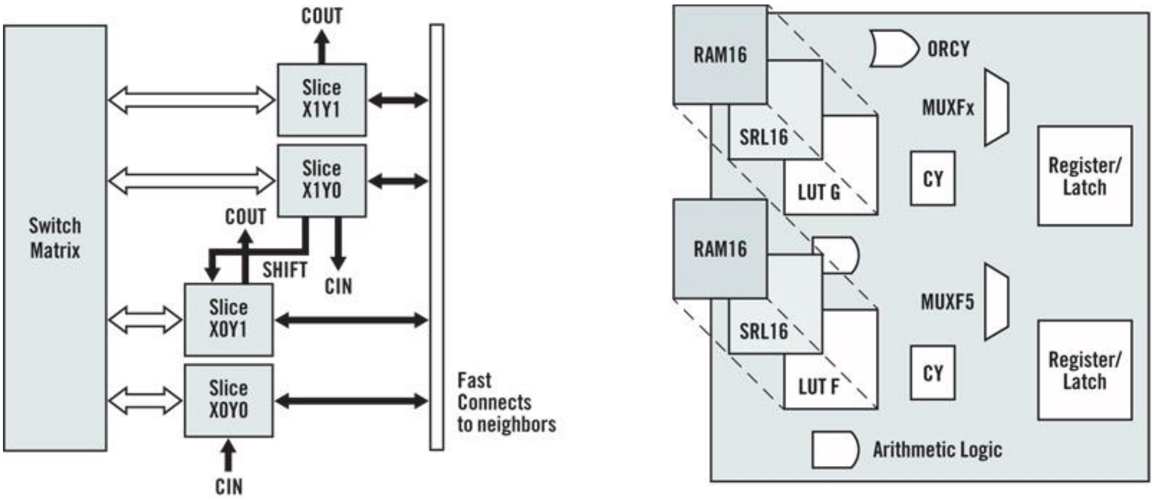


Figure 7. CLB structure

2.2.2.1.2. Interconnect

While the CLB provides the logic capability, flexible interconnect routing routes the signals between CLBs and to and from I/Os. Routing comes in several flavors, from that designed to interconnect between CLBs to fast horizontal and vertical long lines spanning the device to global low-skew routing for Clocking and other global signals. The design software makes the interconnect routing task hidden to the user unless specified otherwise, thus significantly reducing design complexity.

2.2.2.1.3. SelectIO (IOBs)

Today’s FPGAs provide support for dozens of I/O standards thus providing the ideal interface bridge in your system. I/O in FPGAs is grouped in banks with each bank independently able to support different I/O standards. Today’s leading FPGAs provide over a dozen I/O banks, thus allowing flexibility in I/O support. The structure of IOBs is depicted in figure 8.

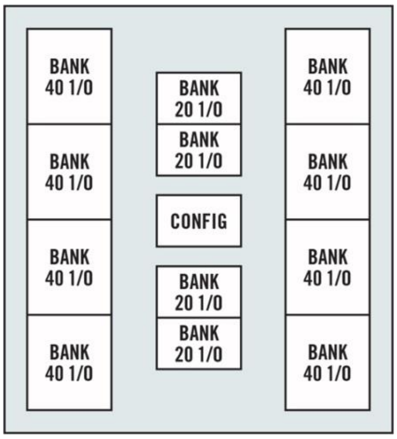


Figure 8. SelectIOb structure

2.2.2.1.4. Memory

Embedded Block RAM memory is available in most FPGAs, which allows for on-chip memory in your design. Somee FPGAs provide up to 10Mbits of on-chip memory in 36kbit blocks that can support true dual-port operation.

2.2.2.1.5. Complete Clock Management

Digital clock management is provided by most FPGAs in the industry. The most advanced FPGAs offer both digital clock management and phase-looped locking that provide precision clock synthesis combined with jitter reduction and filtering[9].

2.2.2.2. Implementation of FPGA

Technically speaking, an FPGA can be used to solve any problem which is computable. This is trivially proven by the fact FPGA can be used to implement a soft microprocessor.Their advantage lies in that they are sometimes significantly faster for some applications due to their parallel nature and optimality in terms of the number of gates used for a certain process. Specific applications of FPGAs include digital signal processing, software-defined radio, ASIC prototyping, medical imaging, computer vision, speech recognition, cryptography, bioinformatics, computer hardware emulation, radio astronomy, metal detection and a growing range of other areas.

FPGAs originally began as competitors to CPLDs and competed in a similar space, that of glue logic for PCBs. As their size, capabilities, and speed increased, they began to take over larger and larger functions to the state where some are now marketed as full systems on chips (SoC). Particularly with the introduction of dedicated multipliers into FPGA architectures in the late 1990s, applications which had traditionally been the sole reserve of DSPs began to incorporate FPGAs instead.

Traditionally, FPGAs have been reserved for specific vertical applications where the volume of production is small. For these low-volume applications, the premium that companies pay in hardware costs per unit for a programmable chip is more affordable than the development resources spent on creating an ASIC for a low-volume application. Today, new cost and performance dynamics have broadened the range of viable applications.

The most common applications of this technology are in Aerospace and Defense, Medical Electronics, High Performance Computing, Industrial, Medical, Scientific Instruments, Security, Video & Image Processing, Wired Communications and Wireless Communications [10].

2.2.3. Multicores and Manycores

Processors were originally developed with only one core, executing one thread at a time. Programmers regularly employ "multithreading" to allow the processor to switch between threads giving the impression that the threads are running concurrently, although the tasks can only be processed one at a time. The only way to improve processing throughput is to run the core faster which requires more energy.

2.2.3.1. Multicore

A multicore processor is typically made up of two, four, six or eight independent processor cores in the same silicon connected through an on-chip bus, a central intersection through which all information must flow between processor cores or between cores and memory and I/O. Multicore processors do execute threads concurrently, typically boost performance in compute intensive processes and will use use less power than coupling multiple single-core processors. However as more cores are added the on-chip bus creates an information traffic jam as all the data must travel through the same path, limiting the benefits of multiple cores.

2.2.3.2. Manycore

A manycore processor is one in which the number of cores is large enough that traditional multi-processor techniques are no longer efficient, such processors will either be more application specific or require a different approach. Some manufacturers address the multi-processor scalability problem with a revolutionary new chip architecture that can harness the processing power of hundreds of cores on a single chip, this provides general purpose manycore processors with 16 to 100 identical processor cores (tiles) interconnected with on-chip network[11].

2.2.4. Embedded systems

Those computer systems that do not look like computer systems to the everyday user are embedded systems. They are the hidden computer systems that form a part of a larger system or product, part of anything from toys to trucks, from mobile phones to medical devices.

There are more microprocessors around the globe used in embedded systems rather than in PCs. With the increasing power and number of devices, our lives become „smarter“ and the number of embedded systems increase rapidly . A consequence of an insatiable drive towards having control over devices and access to data anywhere, anytime. Traditionally the requirements placed upon embedded systems are quite different to those applied for desktop computing. Because embedded systems are in general designed to accomplish a very specific task or group of tasks there is no single characterization that applies to the whole gamut of embedded systems. However some combination of the variables of robustness, small size and weight, real-time requirements, long life cycle and low price could be expected to figure in the design criteria for most embedded systems.

Less tolerance for malfunctions in some cases may be simply a convenience and cost issue, such as the lack of permanent I/O connections which makes debugging more difficult, or it can be far more serious such as the failure a mission critical component which could have extreme consequences.

Real time requirements combine the constraint of time and correctness - not only does the computation need to be correct but it also needs to be at the correct time. This requires an estimate of the worst case performance, which on complicated architectures can be difficult, and leads to overly conservative estimates[12].

Continuing to use the traditional approach to the modern complex designs can become chaotic and ad hoc – and require very experienced personnel. Complexity can increase to the point where it becomes inefficient not to have an operating system to handle various tasks on behalf of the application. This brings in its own issues: Embedded-system developers must select an operating system platform prior to starting to their application development process. The timing of this decision forces developers to choose the embedded operating system for their device based upon current requirements and so locking in their future options to a large extent.

Since the embedded system is dedicated to specific tasks, design engineers can optimize it to reduce the size and cost of the product and increase the reliability and performance. Some embedded systems are mass-produced, benefiting from economies of scale.

Embedded systems range from portable devices such as digital watches and MP3 players, to large stationary installations like traffic lights, factory controllers, and largely complex systems like hybrid vehicles, MRI, and avionics. Complexity varies from low, with a single microcontroller chip, to very high with multiple units, peripherals and networks mounted inside a large chassis or enclosure.

Embedded Systems talk with the outside world via peripherals, such as:

- Serial Communication Interfaces (SCI): RS-232, RS-422, RS-485 etc.
- Synchronous Serial Communication Interface: I2C, SPI, SSC and ESSI (Enhanced Synchronous Serial Interface)
- Universal Serial Bus (USB)

- Multi Media Cards (SD Cards, Compact Flash etc.)
- Networks: Ethernet, LonWorks, etc.
- Fieldbuses: CAN-Bus, LIN-Bus, PROFIBUS, etc.
- Timers: PLL(s), Capture/Compare and Time Processing Units
- Discrete IO: aka General Purpose Input/Output (GPIO)
- Analog to Digital/Digital to Analog (ADC/DAC)
- Debugging: JTAG, ISP, ICSP, BDM Port, BITP, and DP9 ports.

Embedded systems range from no user interface at all — only sending and receiving electric signals — to a full graphical user interface like on a modern computer. Quite often they will have a few buttons and a small display and some LEDs. A more complex system may have a touch screen, allowing the meaning of the buttons to change with each screen as in smart phones[13].

2.3. High level paradigms for parallel hetereogeneus systems.

2.3.1. MPI

The Message Passing Interface Standard (MPI) is a message passing library standard based on the consensus of the MPI Forum, which has over 40 participating organizations, including vendors, researchers, software library developers, and users. The goal of the Message Passing Interface is to establish a portable, efficient, and flexible standard for message passing that will be widely used for writing message passing programs. As such, MPI is the first standardized, vendor independent, message passing library. The advantages of developing message passing software using MPI closely match the design goals of portability, efficiency, and flexibility. MPI is not an IEEE or ISO standard, but has in fact, become the "industry standard" for writing message passing programs on HPC platforms.

In order to program parallel computers MPI is a language-independent communications protocol that is most commonly used. Both point-to-point and collective communication are supported. MPI is a message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation. MPI's goals are high performance, scalability, and portability. MPI remains the dominant model used in high-performance computing today.

MPI is not sanctioned by any major standards body; nevertheless, it has become a de facto standard for communication among processes that model a parallel program running on a distributed memory system. Actual distributed memory supercomputers such as computer clusters often run such programs. The principal MPI-1 model has no shared memory concept, and MPI-2 has only a limited distributed shared memory concept. Nonetheless, MPI programs are regularly run on shared memory computers. Designing programs around the MPI model (contrary to explicit shared memory models) has advantages over NUMA architectures since MPI encourages memory locality.[14]

Although MPI belongs in layers 5 and higher of the OSI Reference Model, implementations may cover most layers, with sockets and Transmission Control Protocol (TCP) used in the transport layer.

Most MPI implementations consist of a specific set of routines (i.e., an API) directly callable from C, C++, Fortran and any language able to interface with such libraries, includingC#, Java or Python. The advantages of MPI over older message passing libraries are portability (because MPI has been implemented for almost every distributed memory architecture) and speed (because each implementation is in principle optimized for the hardware on which it runs).

MPI uses Language Independent Specifications (LIS) for calls and language bindings. The first MPI standard specified ANSI C and Fortran-77 bindings together with the LIS. The draft was presented at Supercomputing 1994 (November 1994) and finalized soon thereafter. About 128 functions constitute the MPI-1.3 standard which was released as the final end of the MPI-1 series in 2008.

At present, the standard has several versions: version 1.3 (commonly abbreviated MPI-1), which emphasizes message passing and has a static runtime environment, MPI-2.2 (MPI-2), which includes new features such as parallel I/O, dynamic process management and remote memory operations, and MPI-3.0 (MPI-3), which includes extensions to the collective operations with non-blocking versions and extensions to the one-sided operations. MPI-2's LIS specifies over 500 functions and provides language bindings forANSI C, ANSI C++, and ANSI Fortran (Fortran90). Object interoperability was also added to allow easier mixed-language message passing programming. A side-effect of standardizing MPI-2, completed in 1996, was clarifying the MPI-1 standard, creating the MPI-1.2.

MPI-2 is mostly a superset of MPI-1, although some functions have been deprecated. MPI-1.3 programs still work under MPI implementations compliant with the MPI-2 standard.

MPI-3 includes new Fortran 2008 bindings, while it removes deprecated C++ bindings as well as many deprecated routines and MPI objects.

MPI is often compared with Parallel Virtual Machine (PVM), which is a popular distributed environment and message passing system developed in 1989, and which was one of the systems that motivated the need for standard parallel message passing. Threaded shared memory programming models (such as Pthreads and OpenMP) and message passing programming (MPI/PVM) can be considered as complementary programming approaches, and can occasionally be seen together in applications, e.g. in servers with multiple large shared-memory nodes.

2.3.2. OpenMP

OpenMP (Open Multi-Processing) is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran, on most processor architectures and operating systems, including Solaris, AIX, HP-UX, Linux, Mac OS X, and Windows platforms. It consists of a set of compiler library routines, directives, and environment variables that influence run-time behavior.

OpenMP Architecture Review Board (or OpenMP ARB) is the nonprofit technology consortium that manages OpenMP, jointly defined by a group of biggest software and hardware companies, including AMD, IBM, Intel, Cray, HP, Fujitsu, Nvidia, NEC, Red Hat, Texas Instruments, Oracle Corporation, and more.

OpenMP uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the standard desktop computer to the supercomputer.

Both OpenMP and Message Passing Interface (MPI) are able to run on a computer in a hybrid model for an application build, or more transparently through the use of OpenMP extensions for non-shared memory systems.

OpenMP has been implemented in many commercial compilers. For instance, Visual C++ 2005, 2008, 2010, 2012 and 2013 support it (OpenMP 2.0, in Professional, Team System, Premium and Ultimate editions), as well as Intel Parallel Studio for various processors. Oracle Solaris Studio compilers and tools support the latest OpenMP specifications with productivity enhancements for Solaris OS (UltraSPARC and x86/x64) and Linux platforms. The Fortran, C and C++ compilers from The Portland Group also support OpenMP 2.5. GCC has also supported OpenMP since version 4.2.[15]

People often make a mistake expecting to get an N times speedup when running a program parallelized using OpenMP on a N processor platform. However, this seldom occurs for these reasons.When a dependency exists, a process must wait until the data it depends on is computed.When multiple processes share a non-parallel proof resource (like a file to write in), their requests are executed sequentially. Therefore, each thread must wait until the other thread releases the resource.Only certain parts of a program can be parallelized by OpenMP, which means that the theoretical upper limit of speedup is limited according to Amdahl's law. N processors in a symmetric multiprocessing (SMP) may have N times the computation power, but the memory bandwidth usually does not scale up N times. Quite often, the original memory path is shared by multiple processors and performance degradation may be observed when they compete for the shared memory bandwidth.Many other common problems affecting the final speedup in parallel computing also apply to OpenMP, like load balancing and synchronization overhead.[15]

2.3.3. OpenACC

The OpenACC Application Program Interface describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator. OpenACC is designed for portability across operating systems, host CPUs, and a wide range of accelerators, including APUs, GPUs, and many-core coprocessors.

The directives and programming model defined in the OpenACC API document allow programmers to create high-level host+accelerator programs without the need to explicitly initialize the accelerator, manage data or program transfers between the host and accelerator, or initiate accelerator startup and shutdown[16].

All of these details are implicit in the programming model and are managed by the OpenACC API-enabled compilers and runtimes. The programming model allows the programmer to augment information available to the compilers, including specification of data local to an accelerator, guidance on mapping of loops onto an accelerator, and similar performance-related details.

OpenACC is a directive-based HPC parallel programming model. OpenACC is designed for performance on many types of platforms (e.g., GPUs, many-core and multi-core). OpenACC is complementary to existing HPC programming models including OpenMP, MPI, CUDA and OpenCL.

Based on directives from the programme, in the form of Fortran comment statements and C/C++ pragmas, that are ignored by other compilers, OpenACC compilers automatically map compute-intensive loops to parallel/vector execution units. OpenACC compilers can manage data movement between CPU host memory and a separate memory on the accelerator. In addition, the OpenACC API provides the programmer with directives to override the compiler's mapping and data movement decisions when necessary. Using OpenACC, programmers can quickly determine if their code will benefit from acceleration. Because it is directive-based, OpenACC requires fewer structural code changes than low-level accelerator programming models like CUDA or OpenCL.

OpenACC is a high-level, performance portable programming model that helps developers to extend the performance of clustered MPI and shared-memory OpenMP-based applications to high-performance, energy-efficient acclerators running thousands or millions of lightweight threads. It defines a comprehensive set of directives for programming massively parallel systems in standard-compliant C, C++ and Fortran[16].

2.3.4. CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing platform and first programming model that enabled high level programming for GPUs (Graphics Processing Units) and thereby made them available for general purpose computing. Previously, if one had to use GPUs for doing general purpose computation then they had to use the Graphics API provided by OpenGL, DirectX and other related graphics APIs and map there computation onto them. All these issues was overcome by CUDA and so now GPUs are also been called GPGPUs that is General Purpose computing on Graphics Processing Units. So writing code in CUDA programmers can speed up their algorithms to a massive extent due to amount of parallelism offered by GPUs.

With millions of CUDA-enabled GPUs sold to date, software developers, scientists and researchers are finding broad-ranging uses for GPU computing with CUDA. There are multiple examples of people from different fields using CUDA to accelerate their programs in their field of action. There are examples in medicine to identify hidden plaque in arteries. Heart attacks are the leading cause of death worldwide. Harvard Engineering, Harvard Medical School and Brigham & Women's Hospital have teamed up to use GPUs to simulate blood flow and identify hidden arterial plaque without invasive imaging techniques or exploratory surgery. Further use is in air traffic flow analyzing where The National Airspace System manages the nationwide coordination of air traffic flow. Computer models help identify new ways to alleviate congestion and keep airplane traffic moving efficiently. Using the computational power of GPUs, a team at NASA obtained a large performance gain, reducing analysis time from ten minutes to three seconds. Other implementation of CUDA is found in visualizing molecules. A molecular simulation called NAMD (nanoscale molecular dynamics) gets a large performance boost with GPUs. The speed-up is a result of the parallel architecture of GPUs, which enables NAMD developers to port compute-intensive portions of the application to the GPU using the CUDA Toolkit[17].

The CUDA platform is accessible to software developers through CUDA-accelerated libraries, compiler directives such as OpenACC, and extensions to industry-standard programming languages including C, C++ and Fortran. C/C++ programmers use 'CUDA C/C++', compiled with "nvcc" – NVIDIA's LLVM-based C/C++ compiler. Fortran programmers can use 'CUDA Fortran', compiled with the PGI CUDA Fortran compiler from The Portland Group.

In addition to libraries, compiler directives, CUDA C/C++ and CUDA Fortran, the CUDA platform supports other computational interfaces, including the Khronos Group's OpenCL, Microsoft's DirectCompute, OpenGL Compute Shaders and C++ AMP. Third party wrappers are also available for Python, Perl, Fortran, Java, Ruby, Lua, Haskell, R, MATLAB, IDL, and native support in Mathematica.

In the computer game industry, GPUs are used not only for graphics rendering but also in game physics calculations (physical effects such as debris, smoke, fire, fluids); examples include PhysX and Bullet. CUDA has also been used to accelerate non-graphical applications in computational biology, cryptography and other fields by an order of magnitude or more.

CUDA provides both a low level API and a higher level API. The initial CUDA SDK was made public on 15 February 2007, for Microsoft Windows and Linux. Mac OS X support was later added in version 2.0, which supersedes the beta released February 14, 2008. CUDA works with all Nvidia GPUs from the G8x series onwards, including GeForce, Quadro and the Tesla line. CUDA is compatible with most standard operating systems. Nvidia states that programs developed for the G8x series will also work without modification on all future Nvidia video cards, due to binary compatibility[18].

2.3.5. OpenCL

In this research OpenCL is used to accelerate the program written in C to get better performance and faster execution. That is why there is a higher focus on OpenCL accelerated systems in this part of the memory. OpenCL is free and public standard that everyone can download, all the development tools that programmer may need. This is language standard where programmers can write applications that will be running on the some sort of hardware, mostly CPU, GPU, DSP etc. Programmers can run the code and don’t have concern which company designed the processor or how many cores it contains. The code will compile and execute on Intel’s Core processors, Nvidia’s Fermi processors, IMB’s Cell Broadband Engine or even AMD’s Fusion processors. All of these processors have same function in personal computer but not everyone is programmed in the same way. Each of these devices has its own instruction set, and before OpenCL, programmer needed to know three or more different languages to be able to program those devices. Therefore Apple Inc. which uses different devices from different vendors in their own products and their programmers need to know multiple languages. In order to program those devices they started to develop new language standard called OpenCL. In June 2008 Apple and other companies formed OpenCL Working Group in Khronos Group, a consortium of companies whose aim is to advance graphic and graphical media, which had task to develop new language standard. First version called OpenCL 1.0 was released in November 2008. [19]

Because of a close resemblance to C/C++, OpenCL was not so new or different, but improvements are defined like a set of data types, data structures and functions. It also has three main advantages comparison with C or C++ languages. Posibly the main advantage of this language is its portability, OpenCL has suitable motto for this, “Write once, run on anything” which means that it will be suitable for every hardware but maybe not optimized. In general this means that programmers can write the code and run it on any compliant device, no matter what device is this, multicore processor or graphic card. This is big advantage over regular high-performance computing, where programmer need to know vendor-specific language to program this specific hardware. This could target multiple devices at once and those devices don’t have to be from same vendor or have to have same architecture, just have to be OpenCL - compliant and everything works perfectly. For example, if user uses multicore processor from AMD and graphics card from Nvidia, normally programmer wouldn’t be able to target both systems at once because each of those requires a separate compiler and linker, but with OpenCL program programmer can develop executable code for both devices. In that case programmer can unify hardware to perform certain task only with single program and if he wants to add more devices, he doesn’t have to rewrite the code, just need to rebuild the program.

Another advantage of OpenCL is standardized vector processing, the term vector in this case is used in one of three different ways. Physical or geometric vector are used in graphics to identify directions. Mathematical vector are specified as two-dimensional collections of elements, called a matrix. Computational vector is data structure that has multiple elements of same data type. Computational vectors are very important to OpenCL because high-performance processors can operate on multiple values at once. Every processor nowadays are able to process vectors, but C/C++ don’t define basic vector data types because vector instructions are usually vendor-specific. NVidia devices use PTX instructions, IBM devices require AltiVec instructions and Intel processors use SSE extensions to process vectors and those instructions don’t have anything in common. OpenCL code can unite vector habits and run them on compliant processor. When applications are complied, Nvidia’s compiler will produce PTX instructions, IMB compiler will produce AltiVec instructions and so on. With this kind of programming in OpenCL, high - performance applications will be available on multiple platforms.

Parallel programming is the third big advantage in OpenCL. This advantage is used when programmers compute tasks to multiple processing elements to be performed at the same time. In OpenCL language those tasks are called kernels. Each kernel is specially coded function that will execute one or more compliant devices. Those kernels are sent to device or devices by host application, which is basic C/C++ application that runs on the user development system. For example, host sends kernels to a single device, like the GPU on the computer’s graphic card, but also the same CPU on which the host application is running can execute them. To manage connected devices, host applications use container called context which is shown in Figure 9., this shows how host interact with kernel and device. In general this works pretty easy, from kernel container called a program, host selects a function to create a kernel, and then he associates the kernel with argument data and sends it to a command queue. This is the mechanism through which the host tells devices what to do. Once the kernel is enqueued, the device can execute this particular function. In this way applications can configure different devices to execute different tasks and every task can operate on different data.

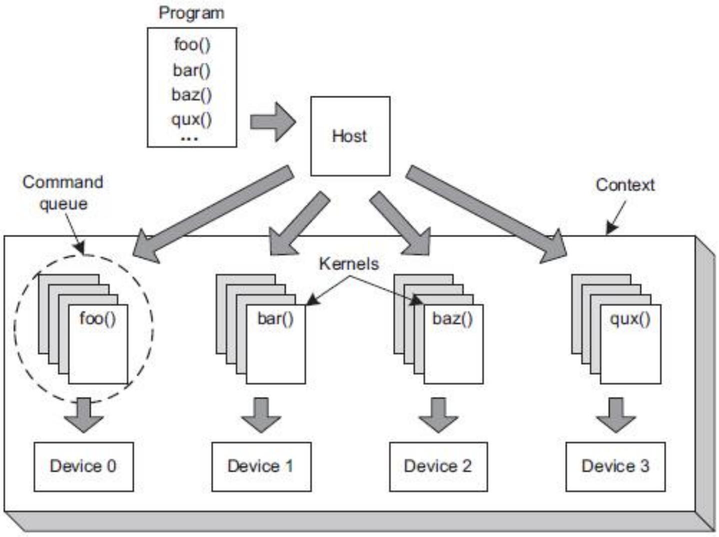


Figure 9. Parallel programming [20]

OpenCL is developed by Apple Inc., which has trademark rights and have initial collaboration with AMD, IBM, Qualcomm, Intel and Nvidia. In June 2008 Apple submitted initial proposal to the Khronos Group when is formed Khronos Compute Working Group with representatives from CPU, GPU, embedded-processor and software companies. The group worked for five months to finish the technical details for OpenCL 1.0. This version of OpenCL is released for public in December 2008. OpenCL 1.0 was released with Mac OS X Snow Leopard, which further extends support for hardware with OpenCL. This software lets any application to use of GPU computing power previously available only to graphics applications. OpenCL is based on the C programming language and it is proposed as an open standard. Later on AMD have decided to support OpenCL instead of developing own language. Nvidia announced full support for the OpenCL 1.0 specification to its GPU Computing Toolkit. In October 2009, IBM released its first OpenCL implementation. [20]

After support of other companies OpenCL 1.1 is developed in June 2010, which contains significant functionality for enhanced parallel programming flexibility, functionality and performance. In this version is included, new data types including three-component vectors and additional image formats, handling commands from multiple host threads and processing buffers across multiple devices, operations on regions of a buffer including, read, write and copy of 1D, 2D or 3D rectangular regions, enhanced use of events to drive and control command execution. Additional OpenCL built-in C functions such as integer clamp, shuffle and asynchronous strided copies also is improved efficient sharing of images and buffers by linking OpenCL and OpenGL events. In November 2011 the Khronos Group has released Open CL 1.2 specification, which added significant functionality over previous versions in way of performance and features for parallel programming. Features that are included are device partitioning, separate compilation and linking of objects, enhanced image support, built-in kernels which means custom devices that contain specific unique functionality are now integrated more closely into the OpenCL framework. Included features are also DirectX functionality, which allows DX9 media surface sharing for efficient sharing between OpenCL and DX9 or DXVA media surfaces. The Khronos Group announced and release OpenCL 2.0 specification in November 2013. OpenCL 2.0 include updates and additions like shared virtual memory, nested parallelism, generic address space, C11 atomics, pipes, Android Installable Client Driver Extensions, industry support etc.

2.3.5.1. OpenCL specifications

OpenCl specifications are defined in four main parts, called models. First model is Platform model, which specifies how many of processors (the host) are capable of executing OpenCL code (the device). This model defines how are OpenCL functions (called kernels) executed on the devices. Second model is Execution model, which defines environment, which is configured on the host and how kernels are executed on the specific device. This model includes settings that OpenCL provides for host-device interaction. Third model called Memory model, defines memory hierarchy that kernels use regardless of the actual memory architecture. Fourth, Programming model defines how the assembled model is mapped to physical hardware.

2.3.5.1.1. Platform model

Platform model uses a single host that coordinates execution on devices. Those platforms are specific for implementations on different vendors of the OpenCL application programming interface (API). Thus, devices are targeted to vendors that can interact with them. When programmers are writing an OpenCL code, in platform model they present an abstract device architecture. The platform model defines a device as an array of compute units where every compute unit is functionally independent from the rest because of scalability of this model. Those compute units are divided into processing element which is shown in Figure 10.

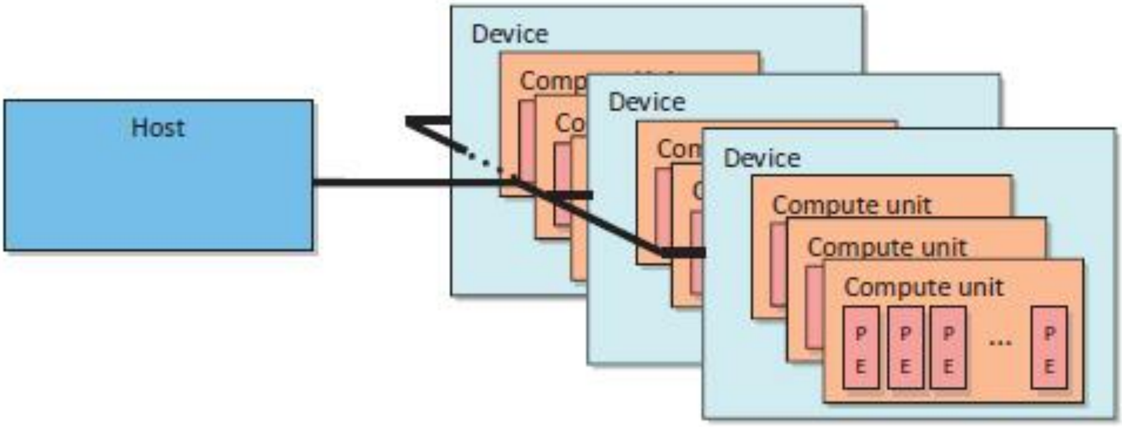


Figure 10. Platform model [19]

2.3.5.1.2. Execution model

This model has environment that configure the context on the host. Command and data has to be configured to the kernel that is going to be executed on a device. In this language, a context is an container that exists on the host, it coordinates how host-device interaction communicate, manage the memory object that is available to the device, also keep track of the program and kernel that is created for device. In the context, has to be used properties argument to restrict the environment of the context. This may provide a specific platform or enable graphics operations or even enable other parameters in the future. If context is limited, then programmer can use context for multiple platforms and use system that contains several vendors. When creating a context OpenCL allows user callbacks that have to be provided so that can be used to report error information. In this language there is a function that creates a context that automatically includes all devices like CPU, GPU, etc., after creating a context there is a function that present number of devices and structure of devices. All those functions and code writing to set context is very tedious, but after programmer write these steps once, he can use it for almost every project.

2.3.5.1.3. Memory model

Generally, memory systems are different between computing platforms. Nowadays all CPUs support automatic caching, but many GPUs do not. To support code portability, OpenCL defines summary memory model so that programmers can target when writing code and vendors can map to memory hardware. Those memory spaces are shown in Figure 11 below.

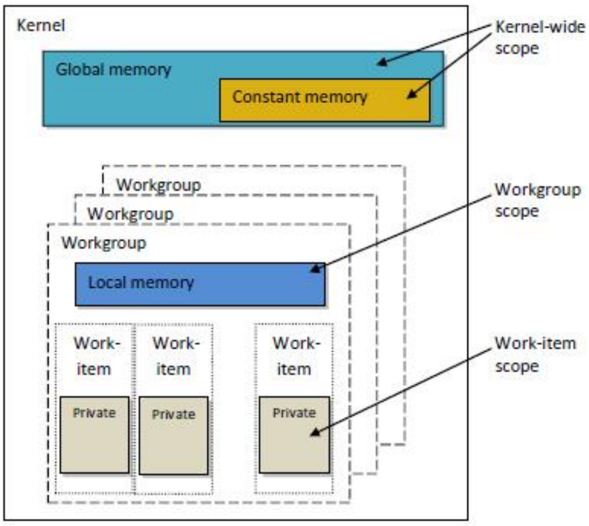


Figure 11. Memory spaces [19]

In OpenCL programs these memory spaces are relevant. Each of those spaces has associated keywords and it is used to specify where a variable should be created or where is the place for certain data. Memory model is divided into four levels of memory. Global memory is like main memory on a CPU-based host, it is visible to all compute units on the device. Every time when data is transferred from host to the device, the data will take place in global memory and conversely. Constant memory is designed for read-only data type but also for data where access to elements is simultaneously by all work-items. If values never change, then those types also get into this category. This model is modelled as part of global memory. When memory object is transferred to global memory can be specified as constant. Local memory is memory which address space is unique for compute device. Local memory is commonly implemented as on-chip memory. It is modelled as being shared by a workgroup. These memories have much shorter latency and much higher bandwidth than global memory. Private memory is private as the name says, it is unique to an individual work-item. Local variables and nonpointer kernel arguments are private by default. These variables are usually mapped to registers whereas private arrays and spilled registers are mapped to an off-chip memory. The memory spaces in this language are similar to the models of GPUs nowadays. Detailed relationship between OpenCL memory spaces and AMD 6970 GPU is shown in figure 12. [19]

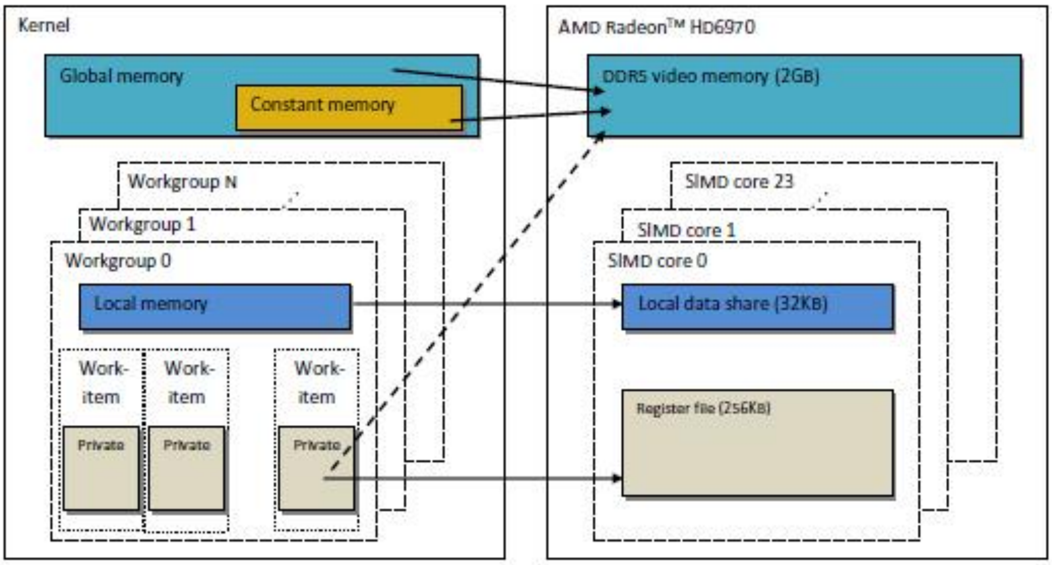


Figure 12. Comparison between OpenCL memory spaces and AMD 6970 GPU [19]

OpenCL is very powerful toolset for writing parallel programs, which can be run on high-performance processors. If programmer uses OpenCL, he doesn’t have to know any other programming language because of OpenCL-compliant hardware. Programmers can write the code once and run it on any compliant hardware. For high-performance computing, portability and parallel programming are big advantages which OpenCL language has. In order to compute in high-performance fashion OpenCL has the advantage because of its portability and parallel programming.

3. Methodology

3.1. Proposed approach

The proposed method is blind, i.e. the original image is not needed to extract the hidden information. The information is embedded into a Y component after the RGB image is transformed into YC_bC_r color space and then transformed with DCT (Fig. 13). The security of the hidden information is ensured by random distribution of bits which is controlled by a key.

3.1.1. Encoder

To embed hidden information four input values are needed: host image that we want to protect, data to be hidden (e.g. another picture, logo or trademark number), factor of implementation and the secret key for pseudorandom generator. Pseudorandom key generator creates array of numbers that represent the position of 8 × 8 blocks in which the portion of information, i.e. 1 bit of a copyright image or a trademark number, is embedded. Factor of implementation α controls the strength of the implementation of hidden information. Higher implementation factor means more robustness; however, it also introduces more degradation to the host image. The factor of implementation α is multiplied by the normalized mean value of all 64 coefficients in the block and with 1 or -1 depending on the bit value of the information to be hidden (1).

$$M'_{j,k} = \frac{\alpha W_k}{64} \sum_{i=1}^{64} M_i \ . \qquad (1)$$

Where $M'_{j,k}$ denotes watermarked coefficient j in block k, M_i original coefficients in block k, W_k denotes k-th bit of the hidden information, and α denotes implementation factor.

The choice of blocks used to hide information is done using secret key k as a seed for a pseudorandom number generator. Therefore, each bit of the logo/trademark number is pseudo randomly distributed over the entire host image. The 8 × 8 block has 64 pixels in which the embedding pixel can be embedded. However, not all positions are equal in terms of robustness or perceptibility. Lower frequency coefficients are perceptually more important and usually more robust to attacks. Therefore, as a trade-off, we implement information in middle frequency coefficient that can sustain SN attack, but does not introduce perceptually significant change to the host image in order not be visible to a human eye. One bit at the time is embedded until the entire information is embedded into the host image we want to protect.

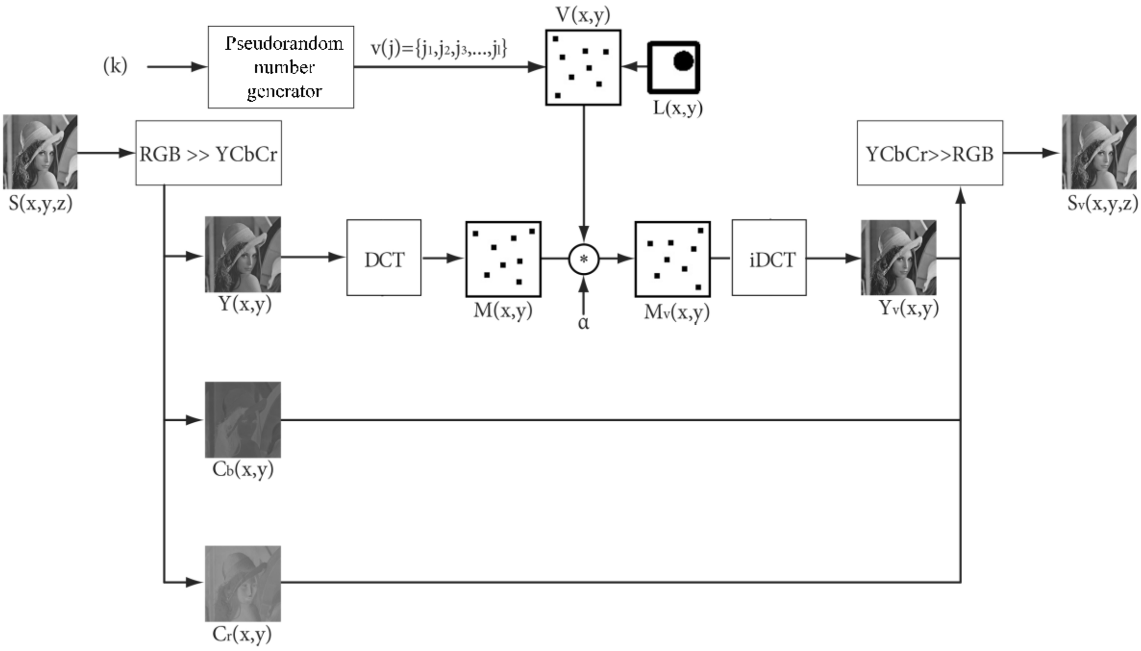


Figure 13. Block diagram of the encoder

3.1.2. Decoder

To extract the information at the decoder (Fig. 14) the only input variable needed, besides the host image, is the key. Without a key it is not possible to know the exact position of the blocks that contain hidden information. In addition, since the hidden information is randomly distributed throughout the host image, the key is used to get the exact order of the hidden bits to correctly retrieve embedded logo/trademark number.

In decoder the key is used as a seed for the pseudorandom number generator. After that decoder extracts hidden information by extracting the values of the coefficients in particular blocks. If the value of the coefficient is positive than the recovered bit has a value of 1, otherwise it has a value 0. Then the vector is transformed into a matrix that represents the bitmap values of a hidden logo. Finally, the matrix is exported as a bitmap document that depicts the logo that was originally hidden in the host image (Fig. 15).

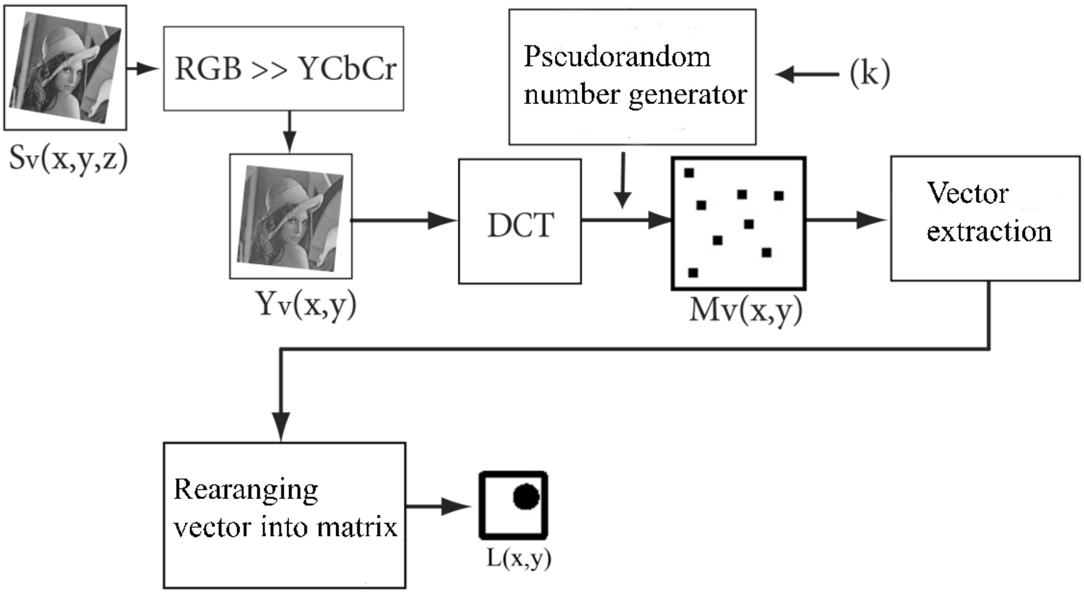


Figure 14. Block diagram of the decoder

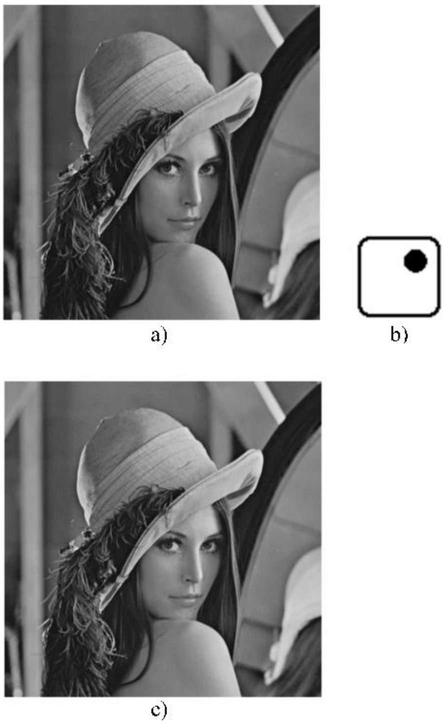


Figure 15. Example of the data hidden in an image: a) Original Lenna, b) Copyright logo, c) Lenna with hidden logo (PSNR = 46.3 dB)

3.1.3. Testing of the Method

To test the method, the image database of 100 images was used. This test database included full color and grayscale images ranging from face to aerial images. The size of the images is 1024 × 768 pixels. The images were previously uncompressed tiff and bmp images in order to avoid any possible error during characterization of the social network (SN) attack.

It is well known that high frequency DCT coefficients carry less perceptually important information while lower frequency DCT coefficients carry more perceptually important information. This means that embedding information in low frequency coefficient will have a greater impact on the quality of the watermarked image than embedding information in high frequency. For an 8 × 8 DCT block high, medium and low frequency coefficients are shown in Fig. 16. On the other hand, robustness to JPEG compression of the high frequency DCT coefficients is smaller than the robustness for low frequency coefficients. For that reason, this method is “fine-tuned” to SN attack in a way that it uses coefficients that are in the medium frequency range providing low perceptibility but still having good resistance to SN attack.

3.1.4. Characterization of the SN attack

At first the SN attack by uploading uncompressed images to the social network was characterized, and then compressed images were downloaded. Impact on the DCT coefficients in an 8 × 8 block is determined by mean difference in coefficient values before and after the attack. From Fig. 16 is clearly visible that the least robust coefficients are those that represent the highest frequency (and vice versa). Therefore the medium frequency zone (Fig. 17) is chosen to embed the hidden information.

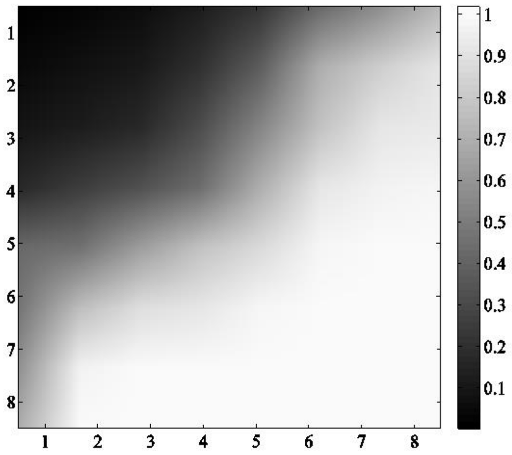


Figure 16. Normalized mean difference of DCT coefficents in a 8 × 8 block when image sustains SN attack

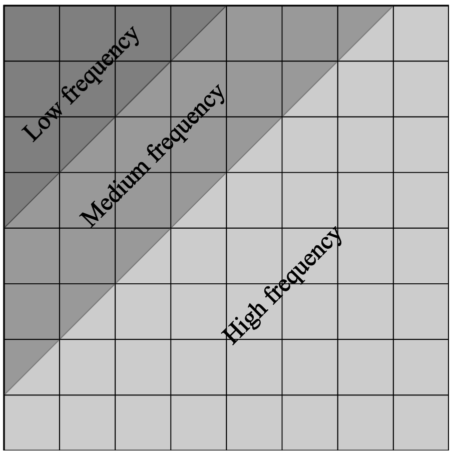


Figure 17. Low, medium and high frequency DCT coefficents in an 8 × 8 block

3.1.5. Optimal implementation factor

Before testing the method against different attacks, the influence of the implementation factor on the optimal implementation factor α has to be determined. For hidden data a bitmap of size 50 × 50 pixels was used (Fig. 21a). To find optimal implementation factor two experiments were conducted to determine the Bit Error Rate (BER) on the images that were not previously attacked, and to determine the degradation of the original image due to information hiding.

The first test was BER test on un-attacked images. It was used to determine how high the factor of implementation must be in order to get hidden information back from the protected image without any loss. The results showed that the factor of implementation must be at least 1.2 to fully recover embedded information (Fig. 18). The test of degradation of the original host image was conducted by calculating the influence of the implementation factor on PSNR value. As depicted in Fig. 19 the mean PSNR value for image test set drops below 40 dB form implementation factor above 2.8. Preliminary analysis showed that the optimal value of the implementation factor is 2. This value gives 0% BER while PSNR value is still above 42 dB which, according to [21], represent low level of degradation.

The last test was SSIM that shows how high is the degradation of image depending on the size of the logo that was embedded into original image. Tests were made on logo size between 10x10 and 50x50 on a 1024x768 image. That correlates to 1% to 20% of the 8x8 blocks used on original image for embedding the logo image. The results are depicted in Fig. 20.

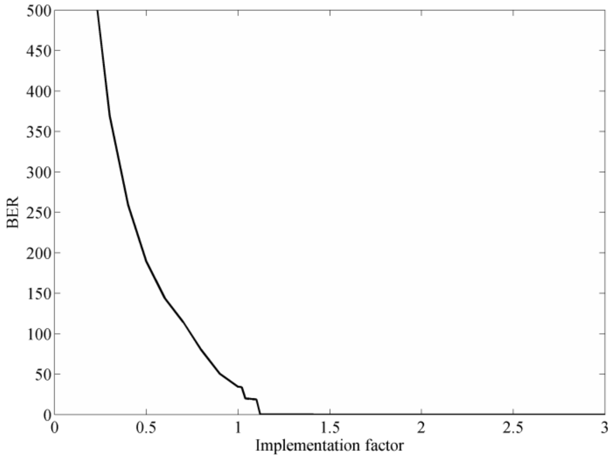


Figure 18. Infulence of implemetation factor on BER

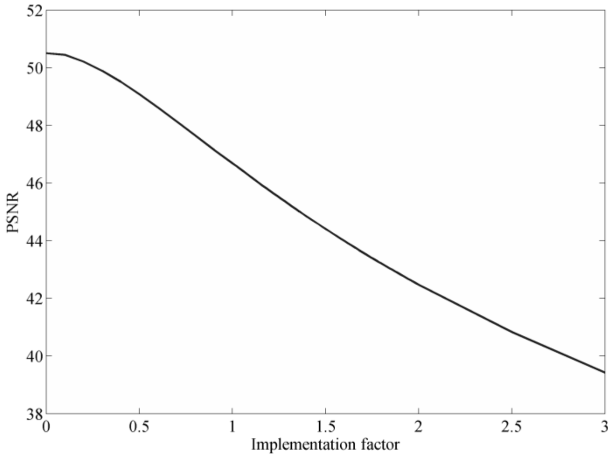


Figure 19. Influence of implementation factor on PSNR

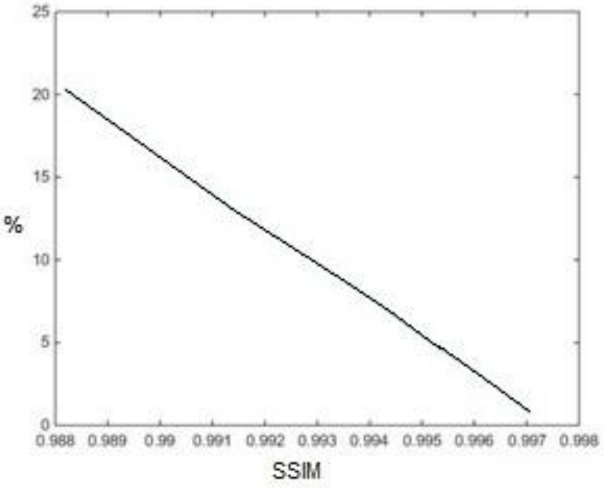


Figure 20. Influence of size of embeding logo on SSIM

3.1.6. Robustness against attacks

After we have empirically determined the strength of the implementation factor we could test the robustness of the proposed method against different attacks. The embedded hidden information was bitmap of size 50 × 50 pixels (2500 bits of information). Due to brevity, here we give results for sharpen, blur (3x3), JPEG (25, 50, 75), scale (0.5, 2), noise (Gaussian noise 1- 0 mean, 0.01 variance, noise 2- 0 mean, 0.001 variance) and SN attack because they are commonly used by average social network user.

After attacking it with the earlier mentioned types of attack, BER, PSNR and SSIM methods were used to see how much information can be retrieved from the attacked watermarked images and PSNR and SSIM to see how different the extracted logos are from the original. Results are shown in Table 1. The proposed method is very robust to blur, sharpen and scale attacks with practically all bits correctly extracted. JPEG compression is more challenging and mean BER, depending on the quality settings was from 6.2 % (JPEG 75) to 16.9% (JPEG 25). For the SN attack, mean BER was 8.87, which is slightly better than for JPEG 50 attack. PSNR and SSIM are as expected lowest for the JPEG 25 and Noise 2 (Gaussian noise, 0 mean, 0.01 variance). The examples of the extracted hidden logo are shown in Fig. 20. [22]

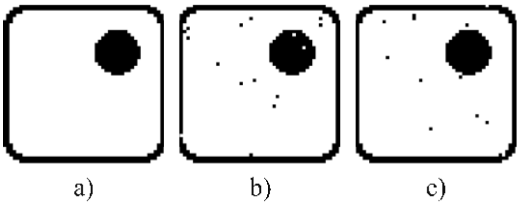


Figure 20. Result of the extraction of the hidden logo; a) original; b) after SN attack (BER = 0.6%) and c) after Blur attack (BER = 0.5%)

Table 1. BER, PSNR and SSIM for different attacks

Attack	***Blur 3×3***	***Unsharp***	***Scale 0.5***	***Scale 2***	***Noise 1***
BER [%]	0.0	0.0	0.1	0.0	1.8
PSNR	∞	∞	***51.6***	∞	***49.3***
SSIM	1	1	0.9986	1	0.9912
Attack	***Jpeg 25***	***Jpeg 50***	***Jpeg 75***	***SN***	***Noise 2***
BER [%]	16.9	9.2	6.2	8.7	12.2
PSNR	32.9	43.3	45.4	43.7	41.3
SSIM	0.9669	0.9887	0.9895	0.9890	0.9853

3.2. Hardware implementation

3.2.1. Profiling

In order to accelerate the code execution profiling of the current code has been done. In software engineering, profiling is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization. Profiling is achieved by instrumenting either the program source code or its binary executable form using a tool called a profiler (or code profiler). Profilers may use a number of different techniques, such as event-based, statistical, instrumented, and simulation methods. Figure 21 show the profiling results of a encoder. Profiling was also performed for decoder and the results are depicted in Figure 22.

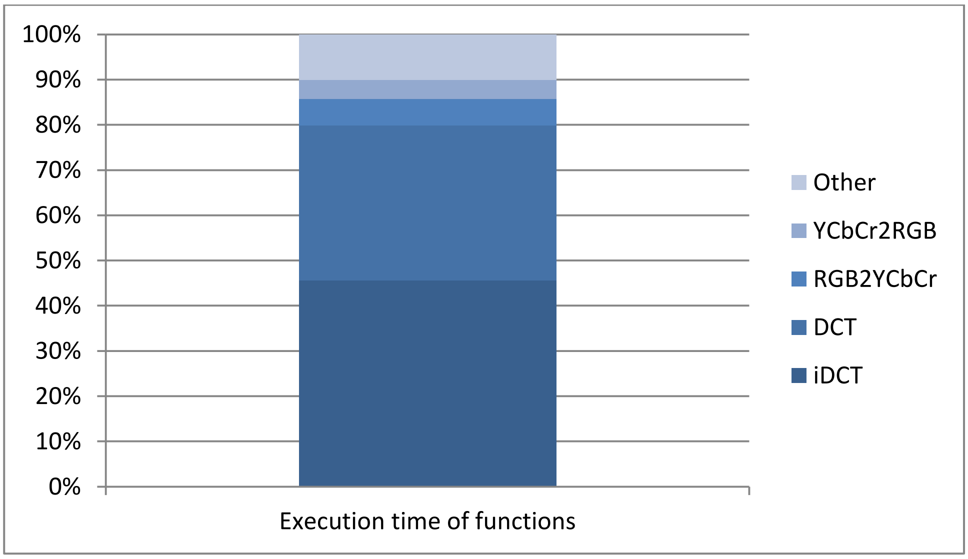


Figure 21. Encoder profiling.

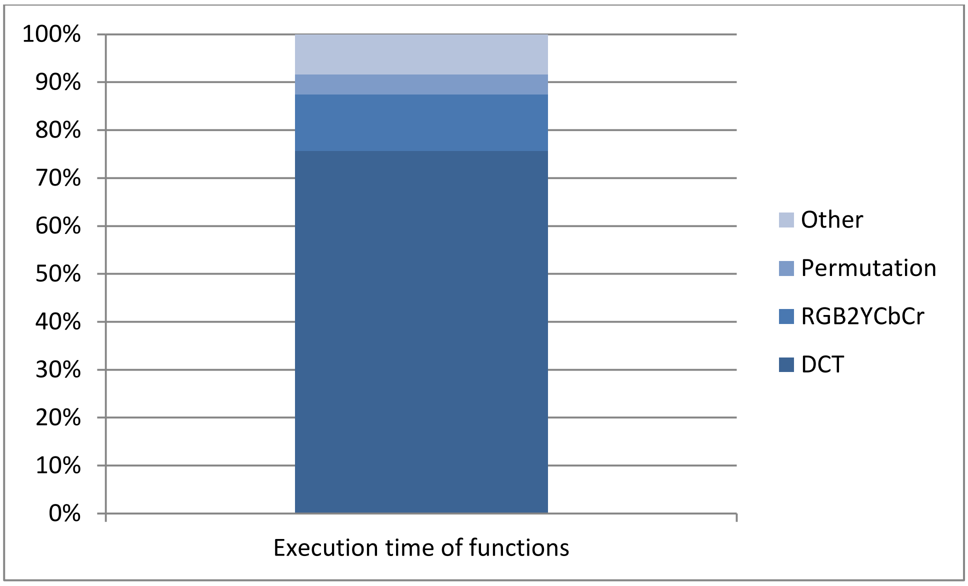


Figure 22. Decoder profiling.

Profiling gave us the expected results. DCT and iDCT were the most time consuming functions and that they are the bottleneck of this algorithm so the decision was made that they are the ones that will be accelerated on hardware.

3.2.2. Devices specifications

The devices used for testing are Intel(R) Xeon(R) CPU E3-1225 v3, Nvidia GeForce GTX980, AMD APU A10-6800K and AMD HD8670D. Hardware specifications of those devices and of a host computer with Intel i3 3110m processor and AMD HD 7670M graphics card are listed in table 2.

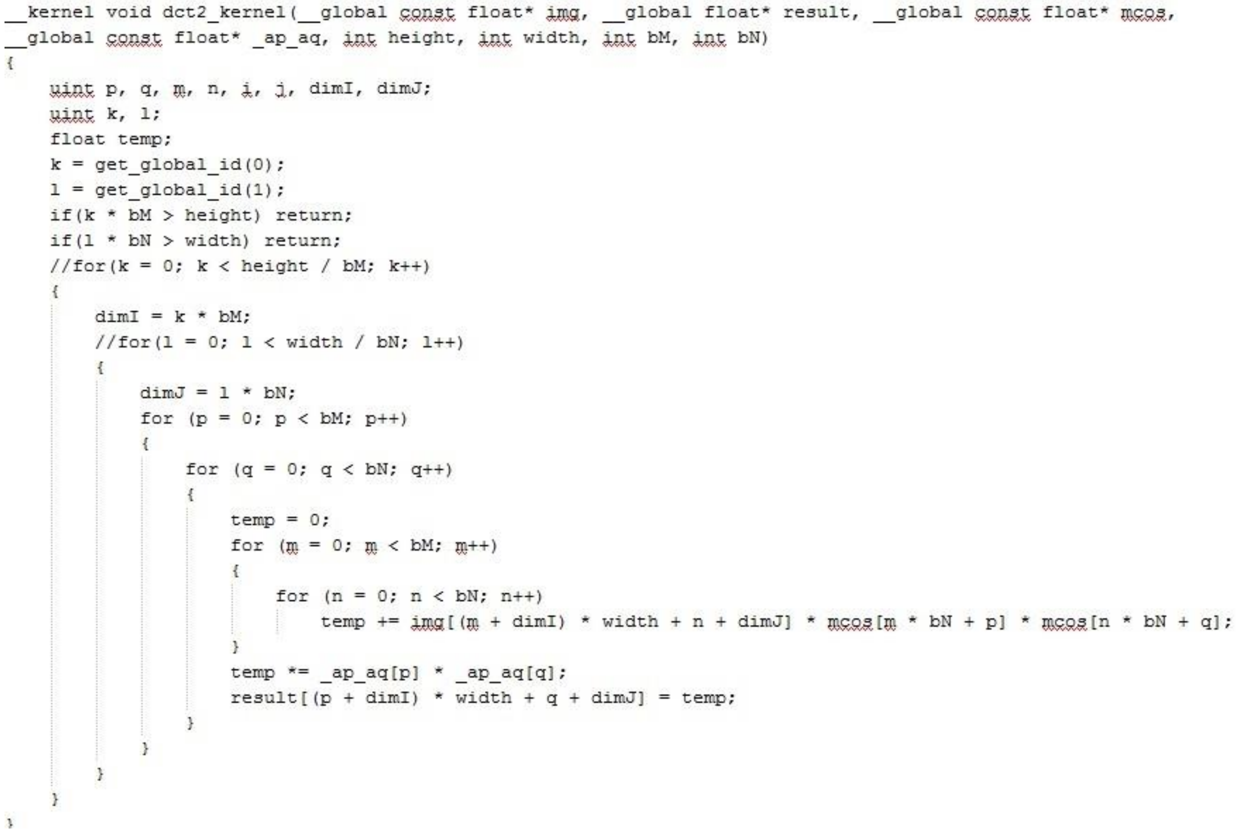
Table 2. Device specifications.

	Intel Xeon CPU E3-1225	NVIDIA GeForce GTX 980	AMD A10-6800K APU	AMD HD8670D	Intel i3 3110M	AMD HD 7670M
OpenCl version	1,2	1,2	1,2	1,2	1,2	1,2
Max. Compute Units	4	16	4	6	4	6
Local Memory Size	32 KB	47 KB	32 KB	32 KB	32 KB	32 KB
Global Memory Size	32101 MB	4095 MB	7197 MB	736 MB	7190 MB	1024 MB
Max Alloc Size	8025 MB	1023 MB	2048 MB	376 MB	2048 MB	376 MB
Max Work-group Size	8192	1024	1024	256	1024	256
Max Work-item Dims	(8192 8192 8192)	(1024 1024 64)	(1024 1024 1024)	(256 256 256)	(1024 1024 1024)	(256 256 256)

Host computer has Intel i3-3110M processor with 4 cores and 2,4Ghz core speed, it consumes 35W [27]. AMD HD7670M is the graphic card with 1Gb of memory and with the core speed of 600 Mhz and memory speed of 900 Mhz (1700 Mhz effective) and a BUS width of 128 bit [28]. Intel Xeon CPU E3-1225 [23] has 4 cores and 4 threads on base frequency of 3,2 Ghz and in boost mode up to 3,6 Ghz on a single core. Max memory size iz 32 Gb and a cache size of 8Mb and consumes 84 W. Nvidia GeForce GTX 980 [24] is a graphic card with 4 Gb of memory with a core speed of 1126 Mhz and memory speed of 1750Mhz (7000Mhz effective). It has a 256-bit BUS width and consumes 165 W. AMD A10-6800K APU[25] has a 4 core processor on 4.1 Ghz with a 4.4 Ghz boost option, it has 4MB of total L2 cache and consumes 100 W. AMD HD6870D[26] is a integrated graphics chip on AMD A10-6800K APU, it has core speed of 844 Mhz and a memory clock at 1067 Mhz, 128-bit BUS width and a power consumption of 100W.

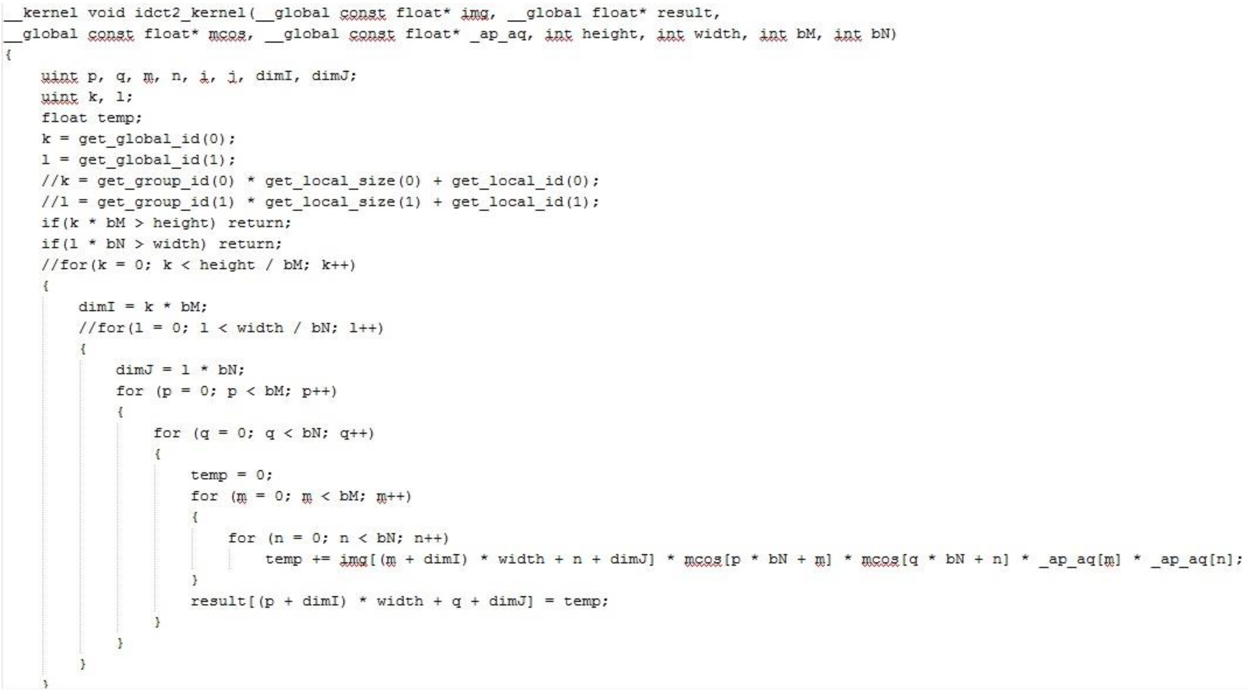
3.2.3. Implementation

OpenCL was used to implement code to hardware. It is a widely used, reliable solution for hardware implementation. Using this aforementioned hardware kernel execution times of DCT and iDCT functions were measured. When creating kernels only a external loops were parallelized. There are more inner loops that could be parallelized but the decision was made on a macroblock level and not on the pixel level. If we want to accelerate DCT and iDCT functions even more than the inner loops of each 8x8 block should be parallelized. Figure 23 show the DCT kernel code and Figure 24 shows iDCT kernel code. Variables k and l represent the block number and are calculated in the way that width and the height of the image were divided with block size.



```
__kernel void dct2_kernel(__global const float* img, __global float* result, __global const float* mcos,
__global const float* _ap_aq, int height, int width, int bM, int bN)
{
    uint p, q, m, n, i, j, dimI, dimJ;
    uint k, l;
    float temp;
    k = get_global_id(0);
    l = get_global_id(1);
    if(k * bM > height) return;
    if(l * bN > width) return;
    //for(k = 0; k < height / bM; k++)
    {
        dimI = k * bM;
        //for(l = 0; l < width / bN; l++)
        {
            dimJ = l * bN;
            for (p = 0; p < bM; p++)
            {
                for (q = 0; q < bN; q++)
                {
                    temp = 0;
                    for (m = 0; m < bM; m++)
                    {
                        for (n = 0; n < bN; n++)
                            temp += img[(m + dimI) * width + n + dimJ] * mcos[m * bN + p] * mcos[n * bN + q];
                    }
                    temp *= _ap_aq[p] * _ap_aq[q];
                    result[(p + dimI) * width + q + dimJ] = temp;
                }
            }
        }
    }
}
```

Figure 23. DCT kernel code.



```
__kernel void idct2_kernel(__global const float* img, __global float* result,
__global const float* mcos, __global const float* _ap_aq, int height, int width, int bM, int bN)
{
    uint p, q, m, n, i, j, dimI, dimJ;
    uint k, l;
    float temp;
    k = get_global_id(0);
    l = get_global_id(1);
    //k = get_group_id(0) * get_local_size(0) + get_local_id(0);
    //l = get_group_id(1) * get_local_size(1) + get_local_id(1);
    if(k * bM > height) return;
    if(l * bN > width) return;
    //for(k = 0; k < height / bM; k++)
    {
        dimI = k * bM;
        //for(l = 0; l < width / bN; l++)
        {
            dimJ = l * bN;
            for (p = 0; p < bM; p++)
            {
                for (q = 0; q < bN; q++)
                {
                    temp = 0;
                    for (m = 0; m < bM; m++)
                    {
                        for (n = 0; n < bN; n++)
                            temp += img[(m + dimI) * width + n + dimJ] * mcos[p * bN + m] * mcos[q * bN + n] * _ap_aq[m] * _ap_aq[n];
                    }
                    result[(p + dimI) * width + q + dimJ] = temp;
                }
            }
        }
    }
```

Figure 24. iDCT kernel code.

3.2.4. Performance results

The tests were made on both encoder and decoder on a 10 different image sizes. Input image sizes were from 9,717 Mpx down to 0,290 Mpx. After the execution time of every function for both encoder and decoder in C was made, when the C code was being profiled, they were compared to the execution times in OpenCL. Since the parallelism is made on most time consuming functions those functions are compared. In the Figure 25 the speedup of an OpenCL set to use CPU parallelism of a DCT and iDCT functions is depicted with both encoder and decoder speedups. Y axis represents speedup and X axis a different image sizes. Since decoder does not have an iDCT function, only DCT speedup was depicted. It is clear that there is a significant speedup, up to 12 times faster than the execution time of a plain C functions. There is a slightly higher speedup for iDCT function in almost every test with 10 different image sizes.

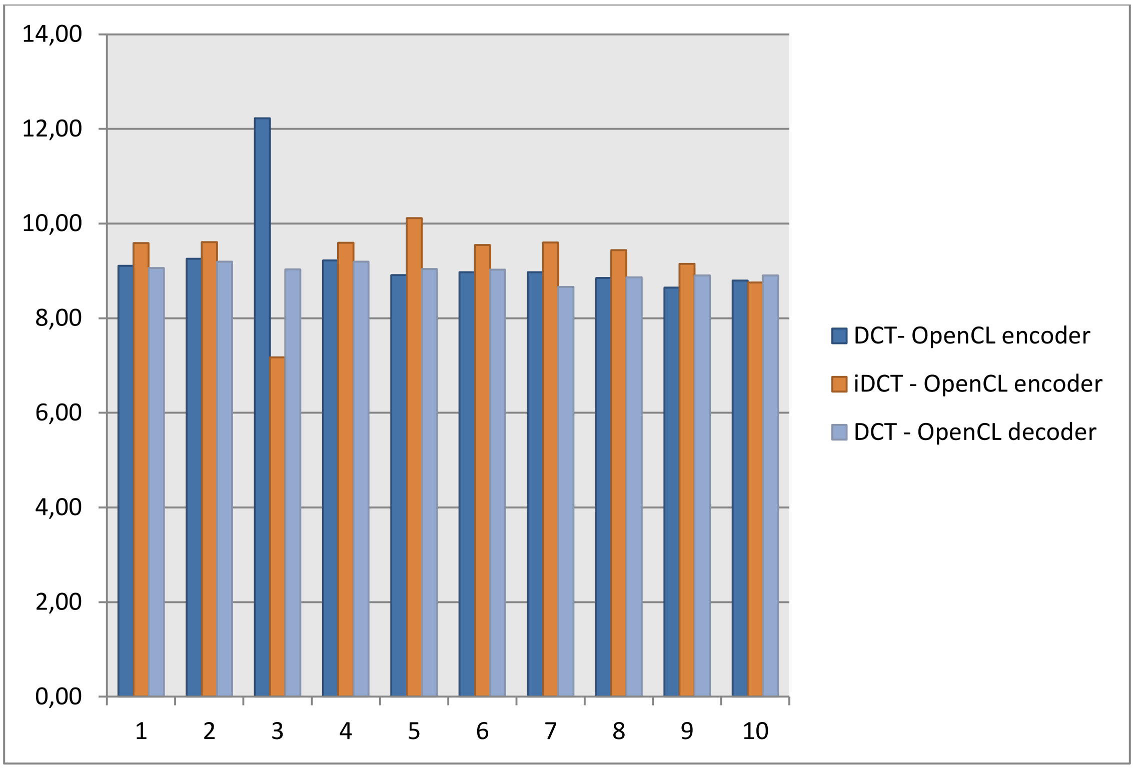


Figure 25. OpenCL CPU speedup.

In the Figure 26 the speedup of an OpenCL code set for GPU execution of DCT and iDCT kernels is depicted. The results for the GPU are even better with the maximum speedup of 15 times the plain C execution time. Again in most cases the highest speedup is on iDCT function.

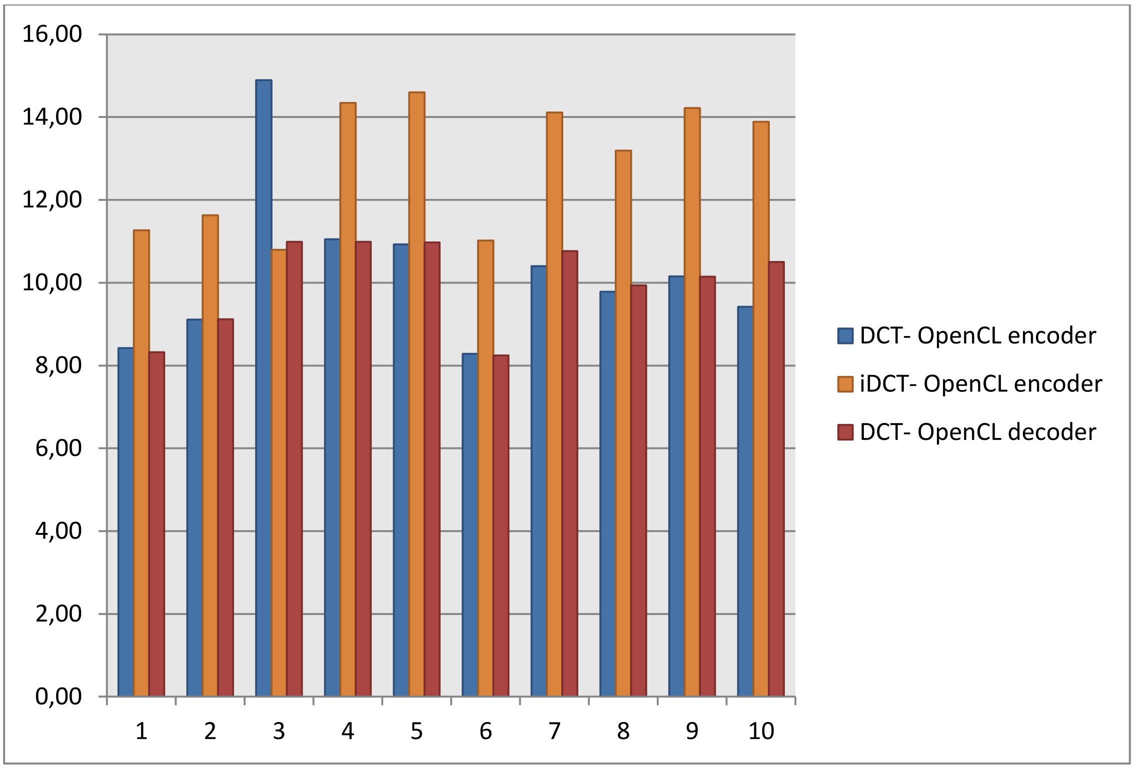


Figure 26. OpenCL GPU speedup.

Now when the kernel execution times are clearly showing speedup the comparison of the whole code was made. Since the rest of the code, besides DCT and iDCT functions, was not parallelized the execution time of those functions remained the same so the whole execution time is actually an execution time of DCT and iDCT functions plus the execution time of the rest of the functions. Figure 27 shows overall execution time of the whole encoder for plain C code and OpenCL CPU and GPU execution times. Y axis represents time in seconds and X axis a different image sizes. Execution time of a plain C code is twice as long compared to OpenCL execution times. GPU shows little advantage over CPU in overall execution time.

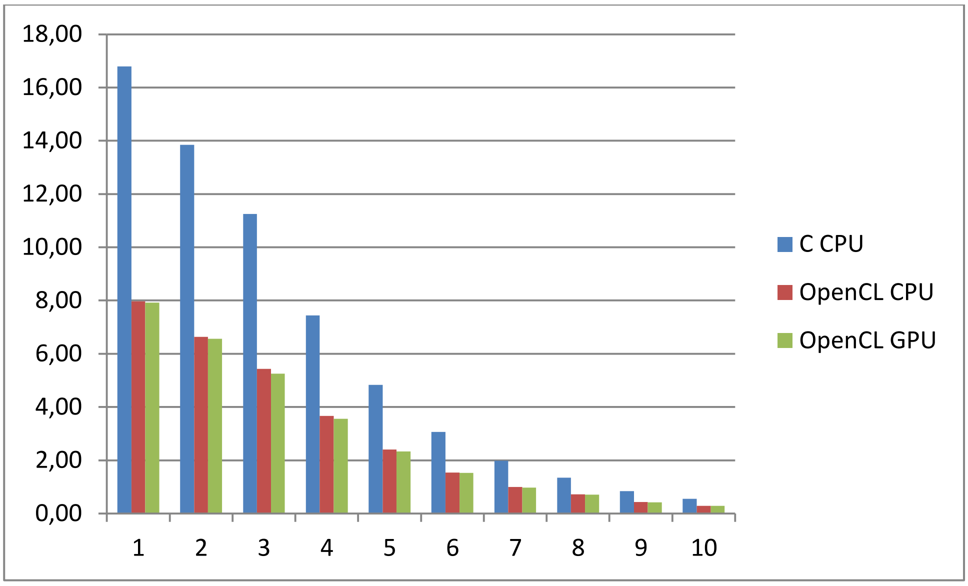


Figure 27. Overall time execution comparison for encoder.

Execution time of decoder is shown in the Figure 28 where the results are similar to the encoder execution time.

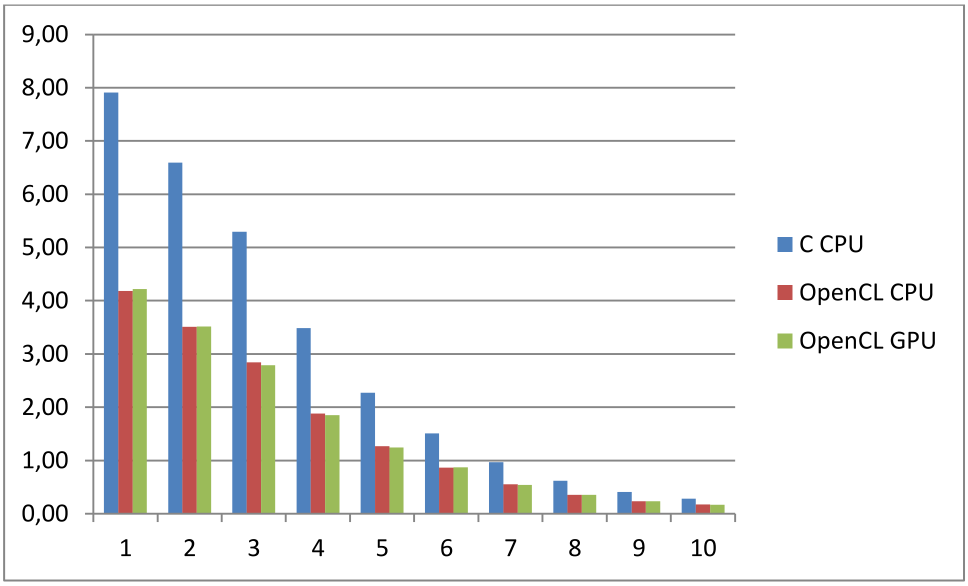


Figure 28. Overall time execution comparison for decoder

In order to accelerate the execution times even more the test were performed on 4 different devices mentioned before. Execution times of DCT kernel for encoder on ten different image sizes were measured and presented in figure 29. Nvidia GTX980 shows the maximum speedup of 95 times the plain C execution time of the same function. Intel Xeon performed well with the average of 26 times speedup, AMD APU 13 times and HD8670D integrated graphics card performed, on average, 5 times faster than the plain C function on a computers CPU.

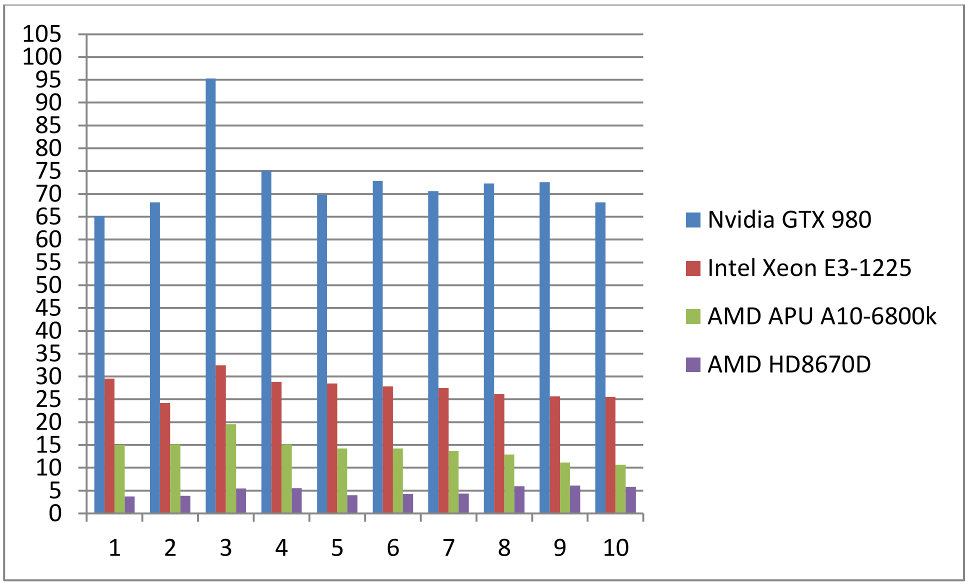


Figure 29. Speedup of a DCT kernel for encoder.

Speedup of a iDCT kernel is shown in the Figure 30. The best result is up to 75 times speedup for a GTX 980. Although DCT and iDCT functions are very similar they both have different execution times and it is clear that the iDCT function is more time consuming.

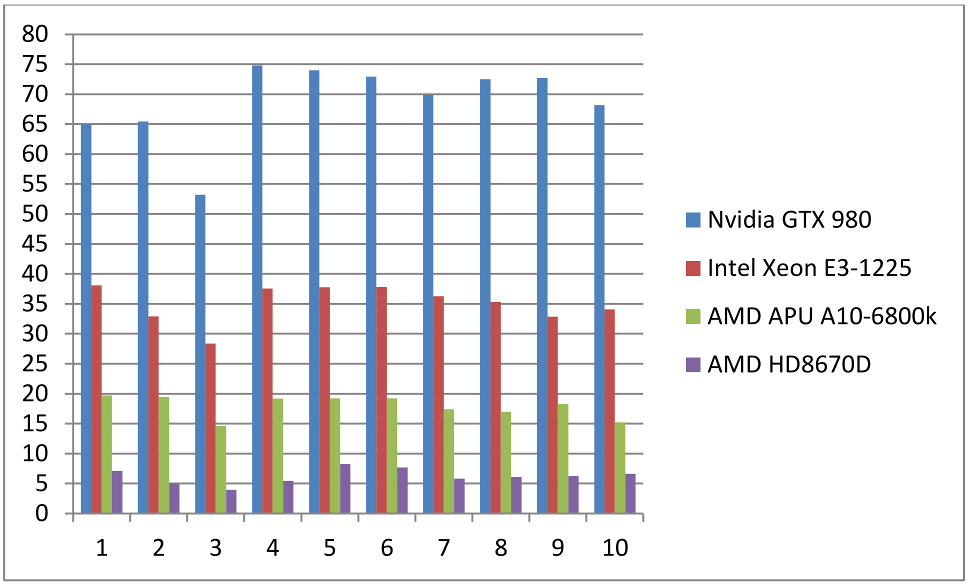


Figure 30. Speedup of an iDCT kernel for encoder.

The last test was made for the DCT execution time for decoder where it showed similar results to the one for the encoder as depicted in figure 31 but at a bit lower rate. Decoders DCT execution time is showing the maximum speedup of up to 70 times faster than the plain C code.

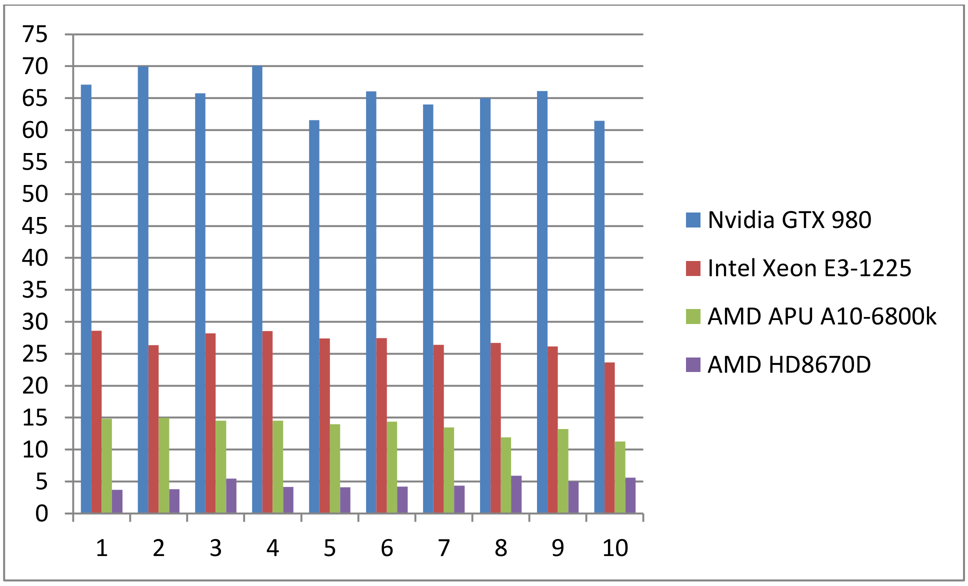


Figure 31. Speedup of a DCT kernel for decoder.

Kernel execution times for all devices are shown in table 3. Execution times are expressed in milliseconds to give the most precise results. It is clear that the most powerful device will give the most speedup. With its maximum computing units of 16 and a high clock levels, the Nvidia GTX 980 showed unparalleled performance outshining the rest of the devices. The rest of devices also showed an improvement in execution time and with the hardware specifications comparison the execution times for each device have been as predicted.

Table 3. Execution times of all kernels for every platform

	Nvidia GTX 980			Intel Xeon E3-1225			AMD APU A10-6800k			AMD HD8670D		
	encoder		decoder	encoder		decoder	encoder		decoder	encoder		decoder
	DCT	iDCT	DCT	DCT	iDCT	DCT	DCT	iDCT	DCT	DCT	iDCT	DCT
input	64,648	87,214	62,463	142,772	148,767	146,582	280,833	287,548	282,664	1131,700	795,982	1129,835
input1	51,061	70,13	49,377	143,744	139,608	131,091	229,736	236,292	230,535	906,512	917,111	907,39
input2	39,034	52,431	41,949	114,691	98,325	97,888	189,913	190,553	189,735	678,223	702,693	505,39
input3	24,033	32,371	25,638	62,615	64,428	62,896	119,382	126,319	123,561	324,213	446,101	429,972
input4	16,167	21,457	18,379	39,623	42,028	41,253	79,401	82,61	80,829	280,148	191,645	275,691
input5	9,991	13,492	10,979	26,167	26,031	26,409	51,256	51,128	50,52	171,519	128,364	173,284
input6	6,630	9,013	7,33	17,011	17,379	17,786	34,218	36,221	34,798	107,567	108,145	107,208
input7	4,122	5,544	4,552	11,408	11,377	11,092	23,098	23,674	24,886	50,281	65,941	49,978
input8	2,673	3,591	2,935	7,557	7,946	7,424	17,440	14,272	14,697	31,898	41,798	38,732
input9	1,864	2,508	2,051	4,981	5,018	5,328	11,919	11,24	11,201	21,931	25,867	22,4

4. Conclusion

In this thesis the main task was to create a steganography method based on discrete cosine tranformation and accelerating it with the hardware implementation. Both tasks were succesfully achieved with good results. Although there are many different types of securing your images many of them do not have a great capacity but are more resistant to image manipulation. In this thesis a method that has a great capacity was developed, is resistant enough and is pleasing to an eye meaning the changes made on images are not visible to a human eye. The implementation factor does not need to be high in order for information to be successfully embedded. The key for the pseudo number generator ensures that the embedded information is not retrievable if the key is not known and that way adds additional safety feature. The embedded information is retrieved in high percentage thus confirming the effectiveness of this method. Hardware implementation was successful and gave a promising numbers regarding acceleration. Although platforms may change the gain in performance is noticable on any platform regarding its specifications. The best results were on Nvidia Geforce GTX980 GPU that showed acceleration up to seventeen times greater than the one on the AMD HD8670D. Other hardware accelerators also performed well and showed a noticable gain in performance.

5. Future work

The next step regarding steganography methods would be to improve the resistance to other types of attacks such as crop or rotate and finetuning the embedding process to gain even higher PSNR and SSIM values without loosing the robustnes. One of the ways would be to separate the image into areas and embedd the copyright logo into each area. That way we would be able to preserve the embedded information even though certain parts of the protected image would be lost but in order to do so some kind of markers would have to be created to tell where the certain area starts.

The future work regarding hardware implementation and acceleration of executing the code would be to improve kernels for DCT and iDCT. Since they are currently parallelized only on macroblock level the next step would be to create kernels for inner loops of a DCT and iDCT function and that way use the hardware resources better and create additional acceleration. Later, to maximize the acceleration, the best thing would be to create kernels for all the functions so that the whole code is executed on hardware. That way it could be implemented into some kind of device that would be used to protect your images without the need of a computer. That device could be programmed once, where user would set the parameters needed for the copyright protection, and use it only to protect images maybe even directly from the photo camera.

6. Bibliography

[1] Cox Ingemar J. *Digital watermarking and steganography*. Morgan Kaufmann, Burlington, MA, USA, 2008

[2] http://www.linuxjournal.com/article/8368 - *Heterogeneous Processing: a Strategy for Augmenting Moore's Law*, 12.6.2015.

[3] Kurak C., McHugh J. *A cautionary note on image downgrading*, Proceedings Eighth Annual Computer Security Application Conference, 1992, 153–159.

[4] Alvarez P., *Using Extended File Information (EXIF) File Headers in Digital Evidence Analysis* International Journal of Digital Evidence. Vol. 2, Vo. 3,(2004), pp. 1–5

[5] Mehdi H., Mureed H (2013), *A Survey of Image Steganography Techniques,* available at: http://www.sersc.org/journals/IJAST/vol54/11.pdf 12.6.2015.

[6] http://www.wellho.net/regex/hardware.html - *Regular Expressions in hardware,* 8.6.2015.

[7] http://gpgpu.org/about - *About GPGPU.org,* 8.6.2015.

[8] http://www.nvidia.com/object/what-is-gpu-computing.html - *WHAT IS GPU ACCELERATED COMPUTING?*, 8.6.2015.

[9] http://www.xilinx.com/fpga/ - *What is FPGA*, 8.6.2015.

[10] http://www.eetimes.com/document.asp?doc_id=1303129 - *Xilinx aims 65-nm FPGAs at DSP applications,* 8.6.2015.

[11] http://www.argondesign.com/news/2012/sep/11/multicore-many-core/ - *Multicore or Manycore, which is apropriate?,* 8.6.2015.

[12] http://www.clarinox.com/docs/whitepapers/Whitepaper_06_CrossPlatformDiscussion.pdf - *Cross Platforms*, 8.6.2015.

[13] http://www.barrgroup.com/Embedded-Systems/glossary - *Embedded system glossary*, 8.6.2015.

[14] https://computing.llnl.gov/tutorials/mpi/#What - *Message passing interface*, 14.6.2015.

[15] http://openmp.org/openmp-faq.html#OMPAPI.General - *OpenMP, FAQ on OpenMP*, 14.6.2015.

[16] http://www.openacc-standard.org/About_OpenACC - *About OpenACC*, 14.6.2015.

[17] http://www.nvidia.com/object/cuda_home_new.html - *What is CUDA*, 13.6.2015.

[18] http://www.techdarting.com/2013/06/cuda.html - *CUDA*, 14.6.2015.

[19] Gaster B. R.., Howes L., Kaeli, D., Mistry P., Schaa D. (2012). *Heterogeneous Computing with OpenCL,* Elsevier Ink, Waltham USA

[20] Scarpino M. (2012). *OpenCL in Action*, Manning Publications Co, New York

[21] Cheddad A., Condell J., Curran K., Mc Kevitt P.,(2010) *Digital image steganography: Survey and analysis of current methods,* Signal Processing, vol. 90, no. 3, (Mar. 2010) pp. 727–752

[22] Kedmenec L., Poljičak A., Mandić L.,(2014) *Copyright protection of images on a social network,* ELMAR (ELMAR), 2014 56th International Symposium, Zadar

[23] http://ark.intel.com/products/75461/Intel-Xeon-Processor-E3-1225-v3-8M-Cache-3_20-GHz - *Specifications*, 15.6.2015.

[24] http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-980 - *Specifications*, 15.6.2015.

[25] http://www.amd.com/en-us/products/processors/desktop/a-series-apu - *AMD A series APU processors*,15.6.2015.

[26] http://www.pc-specs.com/gpu/AMD/APU_Family/Radeon_HD_8670D/1787 - *AMD Radeon HD 8670D*, 15.6.2015.