TALLINN UNIVERSITY OF TECHNOLOGY

Faculty of Information Technology

Department of Computer Engineering

IAY70LT

Karl Janson 131119IASM

IMPLEMENTATION OF HEVC COMPUTATIONAL KERNELS ON DRRA

Master thesis

Supervisor:

Muhammad Adeel Tajammul

M.Sc. (SoC)

Early stage researcher

Tallinn 2015

Author’s Declaration of Originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Karl Janson

15.06.15

Abstract

In the field of high performance computing, hardware accelerators are used to fit computational requirements by exploiting the parallelism that a hardware implementation can offer and taking some load off the general-purpose sequential processor.

This thesis proposes an implementation framework that helps with the task of profiling an application in order to find the computational kernels that can be implemented on a hardware accelerator. Furthermore, a method for extracting these kernels and implementing them on a on the Dynamically Reconfigurable Resource Array (DRRA) based hardware accelerator, is explained.

The usage of the framework is demonstrated by profiling the High Efficiency Video Coding (HEVC) decoder in order to find its computational kernels. The maximum parallelism of the kernels is then investigated and the DRRA-based implementation is described.

This thesis is written in English and is 80 pages long, including 8 chapters, 39 figures and 3 tables.

Annotatsioon

HEVC arvutuslike tuumade implementeerimine DRRA-l

Et täita arvutuslike nõudeid, kasutatakse tihtipeale suurt arvutusjõudlust vajavate ülesannete lahendamiseks riistvaralisi kiirendeid.

Kiirendite kasutamise eeliseks on ühest küljest nende suur paralleelsus, mis võimaldab teha mitut arvutust samaaegselt, samas, kui tavaprotsessorite puhul on paralleelsus võrdlemisi väike (enamasti kaasajal alla kümne tuuma). Teisest küljest aga jätab riistvaraline kiirendi vabaks rohkem ressursse süsteemi põhiprotsessoril, mistõttu jällegi süsteemi üldine jõudlus kasvab.

Selles lõputöös pakutakse välja teostuse raamistik rakenduse profileerimiseks, et tuvastada selle arvutuslikud tuumad. Lisaks sellele käiakse välja meetod nende tuumade teostuseks *Dynamically Reconfigurable Resource Array* (DRRA) peal.

Antud raamistiku kirjeldamiseks näitlikustatakse seda protsessi *High Efficiency Video Coding* (HEVC) videodekoodri peal. Esmalt HEVC dekooder profileeritakse, et leida selle arvutuslikult kõige keerukamad komponendid (arvutuslikud tuumad). Seejärel kirjeldatakse meetodit maksimaalse kasuliku paralleelsuse leidmiseks.

Edasi teostatakse üks leitud tuumadest MATLAB keskkonnas ning analüüsitakse võimalusi selle teostamiseks DRRA peal kasutades VESYLA kompilaatorit.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 80 leheküljel, 8 peatükki, 39 joonist, 3 tabelit.

Table of Abbreviations and Terms

DRRA	Dynamically Reconfigurable Resource Array
HEVC	High Efficiency Video Coding
CPU	Central processing unit
FPGA	Field-programmable gate array
JCT-VC	Joint Collaborative Team on Video Coding
CTU	Coding tree unit
CU	Coding unit
CB	Coding block
PB	Prediction block
TB	Transform block
DCT	Discrete cosine transform
IDCT	Inverse discrete cosine transform
DST	Discrete sine transform
CABAC	Context-adaptive binary arithmetic coding
1-D	One-dimensional
2-D	Two-dimensional
FIR	Finite impulse response
DSP	Digital signal processor
DPU	Data-path unit

FSM	Finite-state machine
BSD	Berkeley Software Distribution
GNU	GNU's Not Unix!
GPL	GNU General Public License
GCC	GNU Compiler Collection
LFSR	Linear feedback shift register
VESYLA	Vectorizing Symbolic Language Assembler

Table of Contents

List of Figures

List of Tables

1. Introduction

In the field of high performance computing (HPC), hardware accelerators are commonly used. Such offloading of the more computationally intensive tasks from the general-purpose CPU to a specialized hardware gives the system a noticeable increase in speed.

For decades such hardware accelerators have been used for various different tasks, starting with arithmetic co-processors in old Intel processor based systems, where the hardware acceleration was used to unload the computationally demanding floating-point calculations to a specialized hardware. Another well-known type of hardware accelerators are the graphics processors, which take the responsibility of doing graphics and other digital signal processing (DSP) applications. Nowadays, physics-related calculations also use such accelerators.

Lately, as the prices for computing equipment are getting cheaper and the needs for performance has increased. Hardware accelerators have found their way into almost every area of modern computing. There exist accelerators for not only graphics and arithmetic, but also for high definition audio, video, gaming, different communication solutions and cryptography, etc. Recently, Intel suggested a reconfigurable (FPGA-based) general-purpose hardware accelerator [1].

The motivation behind use of hardware accelerators is to exploit inherent parallelism within the computationally extensive task to achieve a significant speedup. The general-purpose CPUs, though, execute tasks sequentially and their resources are usually also shared by multiple processes.

By using a hardware accelerator, however, it is possible to offload some parts of the computationally demanding algorithm from the general-purpose CPU into a specialized, highly parallelized, integrated circuit. This solution helps to increase the calculation speed of those algorithms in two ways: first, freeing the general-purpose CPU from the data intensive task and thus giving it more time to work on control intensive tasks, and secondly, speeding up the calculation of these computationally demanding tasks themselves by exploiting their data level parallelism.

According to Amdahl’s law [2], an algorithm cannot be parallelized completely, as the speedup of a program, which is achieved by utilizing parallelism, is limited by the time that is taken by sequential part of the program. Fortunately, Amdahl’s law also offers an estimation of speedup that is achievable for an algorithm, by knowing the relation between the fully sequential and the parallelizable parts. Hence, it is important to know the amount of exploitable parallelism within an application. This thesis looks into the amount of speedup that is exploitable from parallelizable computational kernels within High Efficiency Video Coding (HEVC) [3] .

The High Efficiency Video Coding [3] consists of many potentially computationally intensive components, which may require the assistance of a graphics accelerator in order to speed up the decoding. This is especially true, if real-time HD video streaming is required. In this thesis, the Dynamically Reconfigurable Resource Array [4] is used as the hardware accelerator to host the computational kernels of HEVC.

1.1. Task description

The goal of this thesis is to develop an implementation framework for finding and implementing the computational kernels of the application on the hardware accelerator by:

- Describing a method for profiling and analyzing an application to find its computational kernels
- Describing a method for implementing the computational kernels of an application on the hardware accelerator
- Describing the verification method for the implementation
- Giving an example of using the framework by describing the implementation of the computational kernels of the High Efficiency Video Coding (HEVC) decoder on the Dynamically Reconfigurable Resource Array (DRRA).

1.2. Implementation Framework Overview

In this thesis proposes an implementation framework for profiling an application in order to find its computational kernels for implementation on a hardware accelerator. To illustrate the idea, the thesis describes the process of finding the computational kernels of on HEVC decoder and then implementing on the DRRA as the target hardware.

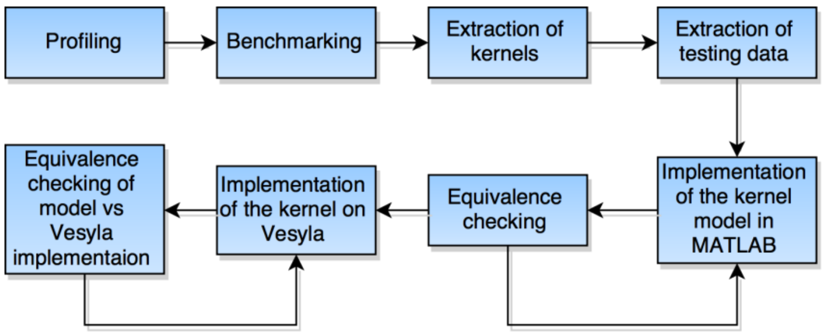

Profiling
Benchmarking
Extraction of kernels
Extraction of testing data
Implementation of the kernel model in MATLAB
Equivalence checking
Implementation of the kernel on Vesyla
Equivalence checking of model vs Vesyla implementaion

Figure 1. Diagram of proposed framework

The proposed framework can be divided six sequential steps (Figure 1):

1. **Profiling of the application:** This step involves the analysis of an open source software implementation of the application of interest. For this step, some profiling tool, which is capable of measuring the instructions that are being used separate functions of the program, is required. In this thesis, the Valgrind [5] software, together with the Callgrind [6] tool, is used.
2. **Benchmarking:** In the benchmarking step, the profiling data is analyzed in order to find the computational kernels that should be implemented in the hardware to achieve speedup of the processing. The kernels should me chosen based on analysis of the parallelizability and computational difficulty. Also, by figuring out the percentage of the parallelizable components in the system, the maximum speedup that can be achieved, can be calculated using the Amdahl's law.
3. **Extraction of the kernels:** The computational kernel that is going to be put on the hardware accelerator should be relevant enough in regards of instruction usage, so that speeding it up would have a measurable effect on the computational process as whole. The kernel should also be parallelizable in order to take advantage of the highly parallel nature of the hardware implementation.

4. **Extraction of testing data:** In case the source code for the application that was profiled before is available, it is also possible to find the implementation of the kernel of interest in the software and modify its code in a way that it would dump all the data passing it into a file. This will generate real-world testing data for testing the implementation of the kernel later in the process.
5. **Implementation of the kernel in MATLAB:** After the computational kernels and the test data are extracted from the software, a MATLAB model of the kernel should be implemented. This is an intermediate step, which is necessary for porting the kernel to DRRA.
6. **Equivalence checking:** After implementing the MATLAB model of the computational kernel, it has to be verified. For this step, the testing data that was extracted from the software, can be used.
7. **Implementation of the kernel in VESYLA-readable format:** Next, the MATLAB implementation of the computational kernel needs to be modified in a manner that can be read by VESYLA, the MATLAB compiler for DRRA.
8. **Equivalence checking between the VESYLA-readable implementation and the MATLAB model:** Finally, the VESYLA-readable implementation should be verified against the MATLAB model. Alternatively, the test cases that were dumped from the software implementation of the algorithm, could be used to verify the equivalence of the VESYLA-readable implementation and the software implementation of the kernel.

1.3. Thesis organization

Following section shows the organization of rest of the thesis:

- In section 2, a short description of previous work related to the topic of this thesis is given.
- In section 3, a general overview of the HEVC video coding standard is given.
- In section 4, discusses identification of computational kernels of HEVC decoder.
- In section 5, discusses short theoretical background of identified computational kernels from section 4.
- In section 6, analyses a sample HEVC decoder’s source code in order to find the computational kernels.

- In section 7, a short overview about DRRA, its compiler VESYLA and the mapping process is given.
- In section 8, Implementation flow consisting of testing data dumped from sample decoder software, MATLAB implementation of HEVS's computational kernel and sample of DRRA implementation of that kernel is presented.
- Finally, some conclusions about the data shown in this thesis is given.

2. Related Work

The High Definition Video Coding standard that is used in this thesis as an example application for describing the proposed implementation framework has existed for a couple of years already, which means that also some other related implementations for it exist. One of the most influential of all the implementations is the semi-official open source software implementation of the entire HEVC video codec [7] that is the basis for this thesis. The sample software is based on the HEVC Text Specification [8] and the overview paper written by the authors of the HEVC standard [3]. In addition, some research exists on profiling the HEVC encoder [9] and on implementing HEVC the decoder on hardware [10].

The hardware platform used in this thesis is the Dynamically Reconfigurable Resource Array (DRRA). Out of the main components of DRRA, interconnection scheme of is described in [11], while the control scheme used in the DRRA is described in [12]. Its data-path units are in [13] and [14] explains address generation scheme used by DRRA. Finally, [4] provides detailed information about all components of DRRA along with information about how it fits in to the world of computational fabrics. The configuration infrastructure of DRRA is in [15] and [16].

In order to simplify mapping of algorithms on DRRA, Vectorizing Symbolic Language Assembler (VESYLA), a high-level pragma-based MATLAB synthesis tool for DRRA, exists. The framework VESYLA is based on is in [17] and [18], while [19] explains VESYLA itself.

3. High Efficiency Video Coding Standard

High Efficiency Video Coding [3], also known as H.265, is the latest version in the series of non-proprietary video coding standards. It was developed by the JCT-VC [3] team as a successor to the H.264/MPEG-4 AVC [20] video coding standard because of the growing demands for higher video compression to serve better the need for ultra-high resolution videos and streaming. Tests [21] show that HEVC can achieve up to 50% better compression rate than MPEG-4 AVC while producing resulting images with similar subjective quality on high-resolution video sequences.

HEVC shares the same overall structure and components with its predecessors, which means that the total increase in performance is achieved by a careful tuning and small updates to the existing components, rather than building a new standard from scratch [3]. In this section, a quick overview of the HEVC video coding standard is given.

3.1. Picture Prediction Methods

To achieve a good compression of a picture, it is necessary to remove as much information as possible without having any noticeable effects on the picture quality. For this reason, HEVC does not encode every picture completely. Because a video is a sequence of multiple pictures with usually only small changes happening between the frames, it makes sense only to store the changes between the frames instead of the entire sequence of the images. During the decoding process, it is possible to produce the original frames by just applying the changes to a previously decoded frame. This type of frame prediction is called inter-picture prediction [3].

However, inter-picture prediction cannot always be used. An example for this case would be the encoding of first frame of a sequence because there are no previous frames that could be used for the inter-picture prediction. Another problem is that as every predicted frame is a modification of the previous one, after some time the small errors accumulate and start producing noticeable artifacts to the image.

To overcome this problem, HEVC includes another way to predict the frames called intra-picture prediction [3]. An intra-picture predicted frame is encoded with only spatial data from the frame itself, so that the decoding process would not depend on any previously decoded frames. The decision about whether a frame should be decoded using intra- or inter-picture prediction is made by the encoder. Usually, in addition to the first frame, an intra-predicted frame is also included in the stream every now and then to mitigate the cumulative errors generated by the inter-picture prediction. This approach also helps to provide random access points to the video.

3.2. Representation of Colors

Although HEVC supports many different color spaces, the most typical color representation used is YCbCr. It consists of the brightness, or luma (Y) component and two color (chroma) components that specify the color's deviation from gray towards blue (Cb) and red (Cr). In HEVC, colors can be represented with either 8- or 10-bit precision. However using 8-bit precision is much more common [3].

The most typical chroma subsampling used in HEVC is 4:2:0 [3], which means that for any rectangular picture with size of W x H, where W is the width and H is the height of the picture in luma samples, W/2 x H/2 of both types of chroma samples are used. This approach helps to lower the amount of information needed to represent the image and therefore reduce size of the encoded picture. This is possible since human eye is more sensitive to brightness than the color.

In this thesis, only the most typical case of YCbCr color space with 4:2:0 sampling and 8-bit precision is viewed.

3.3. Picture Partitioning

To encode a raw video sequence into the HEVC bitstream, each picture of the input video sequence needs first to be partitioned by the encoder into smaller units in order to lower the complexity of calculations and raise the image quality. The rest of the video processing takes place based on these units.

3.3.1. Coding Tree Units

The first level of these units are called coding tree units (CTU). The CTUs are used as a top-level block for processing during the actual encoding or decoding.

CTUs can be viewed as HEVC's version of the macroblocks [20] that were used in earlier video coding standards. Similar to the macroblocks, all CTUs have the same size – either 16x16, 32x32 or 64x64 luma samples, although usually larger size is preferred as it provides a better compression [3]. The size of the CTUs is selected by the encoder at the beginning of the encoding process and it remains constant during the entire video sequence.

3.3.2. Coding Units

The problem with macroblocks used in older video standards is that as they cannot be very large (normally not larger than 16x16 luma samples) [20] or otherwise some of the details in the picture can be lost and the image can get blurry. This, however, sets the limits for the maximum compression that can be achieved.

In order to get a better image compression, the maximum dimensions of the basic coding element, the CTU, have been increased in HEVC to 64x64 luma samples [3], providing much better compression than the macroblock in older standards. This is particularly true for parts of the picture with not much detail , such as a single-colored background.

To overcome the problem of losing details in the picture, HEVC has introduced the concept of further partitioning the CTUs recursively into coding units (CU). A coding unit is the smallest unit that is used for image processing and it can be as large as the CTU (up to 64x64 luma samples) in the areas with little or no detail or as small as 8x8 luma samples for highly detailed areas [3]. This approach enables the HEVC encoded video sequence to have sharp details where they are needed and good compression elsewhere.

Partitioning of the CTUs into CUs is done using the quadtree structure (Figure 2), where a CTU can be recursively divided into four equal-sized CUs. This can be done as long as needed or until the smallest allowed size of 8x8 luma samples of the CU is reached. The decisions about the partitioning are made at the encoding time and the information about the partitioning is stored along with other data in the HEVC bitstream. This means that the decoder can use the information later for correctly reassembling the video sequence.

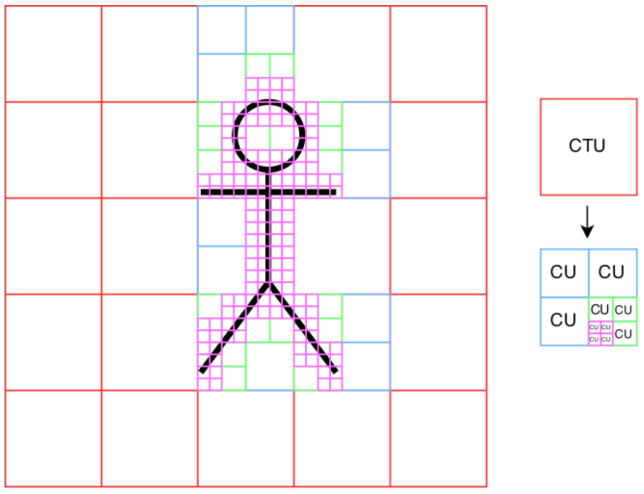


Figure 2. Partitioning of CTUs into CUs

3.3.3. Coding Blocks

Every picture consists of multiple color channels. In case of most typical color representation that is used in HEVC, YCbCr with 4:2:0 chroma subsampling, one luma- and two chroma channels, half the size of the luma channel, are used. For this reason the CU, as a general unit for coding in HEVC, is in turn divided into coding blocks (CB).

Each CB defines the picture data held in the CU for a single channel. The exact size and number on CBs in one CU is defined by the color representation that is used, but in the most typical case, it consists of one luma CB with dimensions equal to the CU's size and two chroma CBs half the size of the CU. Actual image processing is done for every CB separately. In addition to CBs, the CU also contains syntax elements, the data describing the CU such as its size, location in the picture, etc.

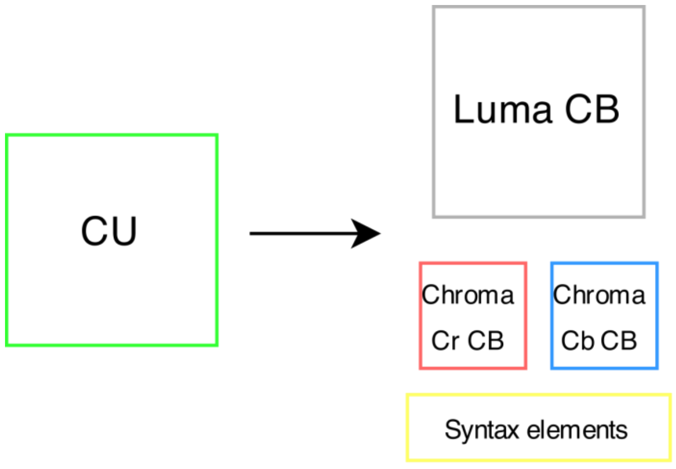


Figure 3. Division of a CU into CBs and syntax elements

In fact, the same logic between CUs and CBs can be also applied to any other partitioning unit in HEVC. The main logic is always the same: while talking about picture partitioning in HEVC, *units* mean the logical coding elements containing all the color channels joined with some additional coding related information. *Blocks*, on the other hand, mean the real frame buffer data for one specific color channel where the actual processing is targeted.

3.3.4. Prediction Block

A CB can be divided further into prediction blocks (PBs), which are used for picture prediction. The way the division is performed depends on whether the current picture uses intra- or inter-picture prediction.

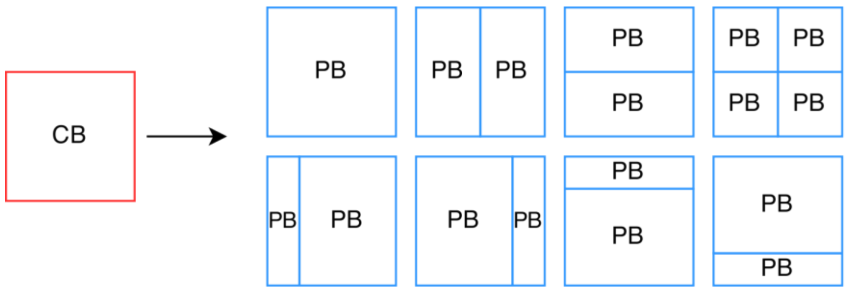


Figure 4. Different ways to divide a CB into PBs

For intra-picture prediction, a PB has usually the same size as the CB, except for the smallest size of CB that is allowed by the HEVC standard, 8x8 luma samples. In this case, the CB can be divided into four PB quadrants, 4x4 luma samples in size, which can have different intra-picture prediction mode each. [3]

In case of inter-picture prediction, a CB can be partitioned into one, two or four PBs, while division into four PBs is done only if the current CB has the minimum size that is allowed by the HEVC standard. In case a CB is divided into four PBs, they will always be equal in size. To divide a CB into two PBs, six different options, including asymmetric division methods, exist [3].

The different ways, how the partitioning of a CB into PBs is done are illustrated in Figure 4. The details about the partitioning, however, are out of the scope of this thesis and in case more information about the topic is desired, the reader is referred to [3].

3.3.5. Transform Blocks

In order to compress the residuals, the output, of the picture prediction process, they need to be transformed into the frequency domain and quantized. To do so, HEVC utilizes two types of integer transforms that are explained in more detail in subsection 5.1.

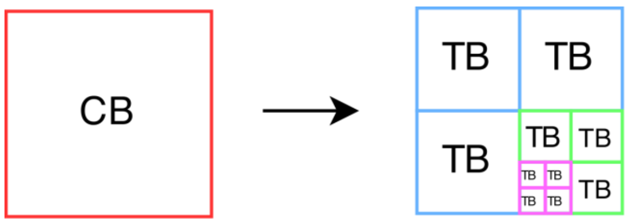


Figure 5. Partitioning of a CB into TBs

The transform is performed on transform block (TB) level that are, like PBs, produced by recursively dividing a CB into four quadrants until needed or until the minimum TB size of 4x4 luma samples is reached as shown in Figure 5. The process itself is very similar to the process of partitioning a CTU into CUs, as described in section 3.3.2 [3].

The size of the chroma TB is always half the size of the luma TB, except in case of 4x4 luma TB, in which the size of the chroma TB is also 4x4 luma samples [3].

3.4. HEVC Encoder

The first step in encoding a picture in a video sequence into HEVC bitstream (Figure 6) is making a decision to about the prediction type that is used (inter- or intra-picture prediction, subsection 3.1). After that, the image is partitioned as explained in subsection 3.3.

Next, the prediction residuals for the picture is generated. This helps to reduce the amount of data that is stored or sent, as only the differences between different blocks are used instead of the entire picture.

The exact method used for generating of the residuals depends on the decision that was made about the type of the prediction in the image partitioning step. In case the intra-picture prediction was chosen, the residuals are generated by only using data from the same picture, so they contain only information about the spatial with neighboring blocks, but not the block in other frames.

In case the inter-picture prediction was chosen, temporal differences from previous, and sometimes even upcoming, frames is used. As a changing of the position of an object in time is actually movement of that object, the residuals that are generated by inter-picture prediction are also called **motion vectors**.

After the generation of the residuals, they are transformed using a 2-dimensional transform into the frequency domain. Then the transform coefficients are quantized and scaled to remove some frequencies that are less important for retaining the picture and not noticeable when removed, thus achieving image compression.

Finally, other information that is related to encoding of the image is also added to the bitstream and the result is compressed using a lossless CABAC-derived entropy encoding algorithm [3].

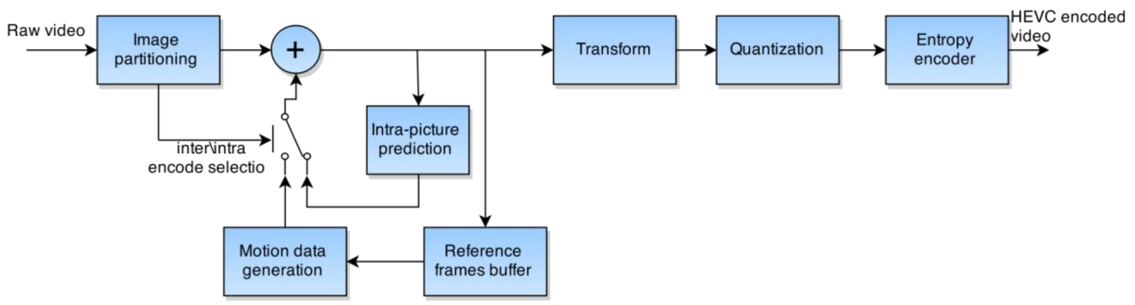


Figure 6. Dataflow graph of the HEVC encoder

3.5. HEVC Decoder

HEVC is designed so that the decoder almost exactly mirrors the inner workings of the encoder as it goes through all the same steps, but in a reverse order and by doing the inverse of all the operations. The dataflow model of the HEVC video stream can be seen in Figure 7.

To decode HEVC video stream, first it is entropy decoded to uncompress the bitstream. The next step is to inverse scale, inverse quantize and then inverse transform it in order to reproduce the prediction residuals that were generated by the encoder.

After the residuals are found, they are used to restore the original picture by adding the residuals to previously decoded blocks.

Finally, the decoded picture is put through two in-loop filters (the deblocking filter and the sample adaptive offset filter) to smooth out some of the artifacts that were generated by the quantization.

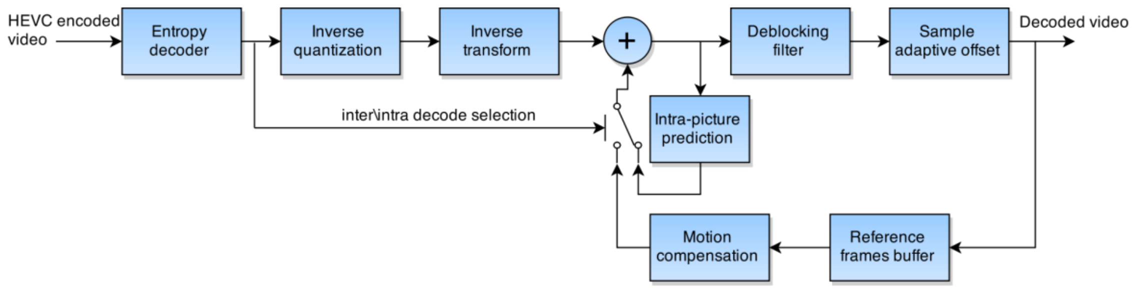


Figure 7. Dataflow graph of the HEVC decoder

4. Profiling the HEVC Decoder

4.1. Sample HEVC Software

In order to find the computational kernels in HEVC, it is necessary to profile it. By profiling, it is possible to find out which component of the codec uses relatively the most instructions, when compared to others, during the processing of a video sequence.

A video codec consists of two parts: a decoder and an encoder. In this thesis, the focus is put on the decoder, as it is the part of the HEVC codec that is also used the most in real life and has, as well, a bit lower complexity. Additionally, there already exists some research about profiling the HEVC encoder [9].

In this thesis, an open source, BSD licensed, HEVC example software [7] was used for experimenting. The software is almost ideal for profiling because it is open source, so it is possible to directly change the source code, if needed. In addition, profiling it will probably also give quite accurate results for a general case of an HEVC decoder as it follows quite directly the HEVC video codec text specification [8], without many optimizations [7].

4.2. General Structure of the Sample Source Code

```
HEVC_sample_sw
├───bin      <– Output folder for compiled binary files
├───build    <– Folder with building information for various platforms
│   ├───linux
│   ├───vc10
│   ├───vc8
│   └───vc9
├───cfg      <– Codec configuration files
├───compat
├───doc      <– Documentation
├───HM.xcodeproj
├───lib
└───source   <– Codec source code
```

Figure 8. Top level folder structure for the example HEVC codec source code

The code is written in C++. Its top-level hierarchy is quite straightforward and common for an open source project. The most important top-level directories are shown in Figure 8. As it can be seen in the figure, it includes building scripts for GNU/Linux (based on makefiles) and for different versions of visual C++. For this thesis, the compilation was done using GNU GCC compiler under the GNU/Linux operating system. None of the codec configuration in the "cfg" folder was changed and the default values were used.

```
HEVC_sample_sw
│
⋮
│
└───source
    ├───App             <- Application folder
    │   ├───TAppDecoder   <- Decoder application code
    │   ├───TAppEncoder   <- Encoder application code
    │   └───utils
    └───Lib             <- Library folder
        ├───libmd5        <- Libraries for calculating MD5 checksums
        ├───TAppCommon    <- Common application related libraries
        ├───TLibCommon    <- Common libraries (used by many different parts)
        ├───TLibDecoder   <- Decoding related libraries
        ├───TLibEncoder   <- Encoding related libraries
        └───TLibVideoIO   <- Video bitstream related libraries
```

Figure 9. HEVC sample codec source code's folder structure

The source code of the sample HEVC codec is sorted into folders mostly by its usage (Figure 9). The same type of organization continues also for the individual files, as most of the file names start with a hint about their usage.

The most important source files are:

- /source/App/TAppDecoder/decmain.cpp: the main file, where the program's execution starts.
- /source/Lib/TLibCommon/typedef.h: this file is included in all the other files in the application, it defines the global variables, types, macros, etc.
- /build/linux/makefile: the main makefile for compilation under the GNU/Linux operating system.

4.3. Profiling of HEVC Sample Decoder

4.3.1. Profiling Process

Profiling of the HEVC decoder was done under GNU/Linux operating system using the open source GPL licensed Valgrind [5] toolkit along with the Callgrind [6] tool. Callgrind is capable of profiling the software by mapping the layout of all the functions in the program together with the information about the number of instructions they use and their relations to other functions. The maps can be later visualized by using another open source tool called KCachegrind [22].

```
decoder_analyzer_debug_gcov:
   $(MAKE) -C lib/TLibVideoIO   debug MM32=$(M32) GCOV_FLAGS=1
   $(MAKE) -C lib/TLibCommon    debug MM32=$(M32) GCOV_FLAGS=1
   $(MAKE) -C lib/TAppCommon    debug MM32=$(M32) GCOV_FLAGS=1
   $(MAKE) -C lib/TLibDecoderAnalyser   debug MM32=$(M32) GCOV_FLAGS=1
   $(MAKE) -C app/TAppDecoderAnalyser   debug MM32=$(M32) GCOV_FLAGS=1
```

Figure 10. Makefile section to add coverage information

In order to profile a program using Valgrind and Callgrind, it is first compiled with the GCOV_FLAGS=1 parameter that instructs the compiler also include coverage related information in the executable. The compiler parameter was added by writing a new section to the main makefile, which compiles the HEVC decoder. The section can be seen in Figure 10.

```
valgrind --tool=callgrind ./TAppDecoderAnalyserStaticd -b input.hevc
-o output.yuv | tee callgrind.txt
```

Figure 11. Command for running Valgrind

The actual process of profiling a program by using Valgrind and Callgrind is quite straightforward. All that is needed is to run Valgrind with the option that instructs it to use the Callgrind tool and add the program to be profiled as parameter, as seen in Figure 11. Valgrind will run the program by acting as an intermediary between the program and the operating system, thus being able catch all the instructions that pass through it. Additionally, as done in Figure 11, the output of the profiling process can be piped into the “tee” command in order to store it in a text file.

4.3.2. Analysis of Profiling Results

After the profiling process is finished, a file with the profiling results, called callgrind.out.xxxx, where xxxx is a four-digit number, is created. This file can be then analyzed in order to determine the computational kernels of the program.

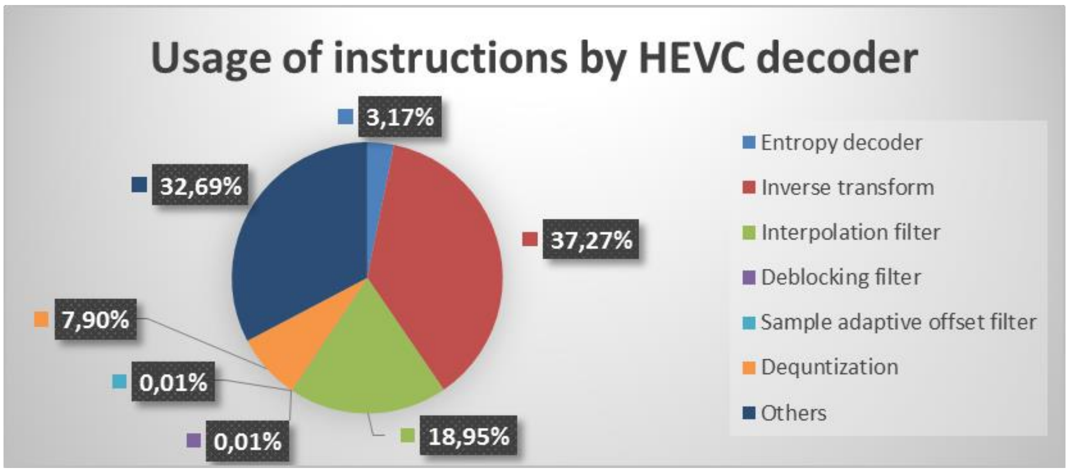


Figure 12. Pie chart describing the usage of instructions per functions in HEVC decoder

Figure 12, which was generated by analyzing the profiling data (appendix 2), shows the relative usage of instructions by different computational kernels of the HEVC decoder. On the figure, only the kernels with larger footprints are shown. Kernels that have a relatively small footprint, together with sequential functionality, are classified in the figure as “others”. As each of the non-sequential functions in this group takes up a relatively small number of instructions, for simplicity, this group can be thought as completely sequential and non-parallelizable, as placing these functions on the accelerator would not make much difference to the overall performance of the HEVC decoder.

According to the chart in Figure 12, in case of the HEVC decoder, the parallelizable part of the application is 100% - 32.68% = 67.32%. However, it can be seen from the figure, that two kernels, the inverse transform, and the interpolation filter, in total, use 56.22% of all the instructions in the HEVC decoder. To be exact, the inverse transform takes up 37.27% and the interpolation filter 18.95% of all the instructions used by the decoder during the profile. It this can also be observed on the HEVC decoder’s map in appendix 9, where it can be seen, that most of the bigger blocks belong to either the inverse transform or the interpolation filter.

As the amount of parallelization that is achievable by only these two functions is very close to the total amount of parallelization achievable in this application, it makes sense to choose the inverse transform and the interpolation filter as the kernels to be implemented on the hardware accelerator.

By knowing the percentage of the parallelizable functionality in the application, it is possible to calculate the theoretical maximum speedup that can be reached by parallelizing this part of the algorithm. To do this, the Amdahl's law [2] can be used (1):

$$S(n) = \frac{1}{(1-P)+\frac{P}{n}} \tag{1}$$

Where:
n ∈ ℕ is the number of thread in execution,
P ∈ ℕ is the percentage of the algorithm that can be parallelized.

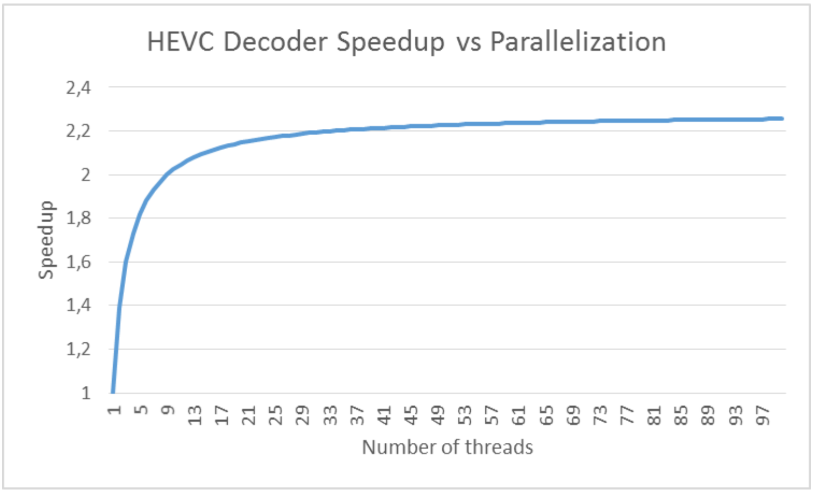


Figure 13. Amount of parallelization vs speedup in the HEVC decoder

As calculated before, by parallelizing the inverse transform and the interpolation filter, it means, that it is possible to parallelize 56.22% of the entire application. By using this number as the reference, it is possible to apply the equation of the Amdahl's law (1) to find out the maximum speedup and the optimal amount of parallelization needed.

As it can be seen from the chart in Figure 13, in the beginning, by increasing the parallelization, the speedup increases drastically, but at around 36 concurrent threads, the increase of the speedup decreases considerably and from that point on, increasing the concurrency does not provide a noticeable increase in speed.

This means, that making more than 36 calculations simultaneously in the hardware accelerator does not give a noticeable increase in speed, so it is possible to save on the chip area and cost of the hardware accelerator by not increasing the concurrency beyond that point.

It can be seen from the chart, that the absolute maximum increase in speed by implementing this part of the application in hardware, remains somewhere around 2.2 times the speed of a sequential calculation of the same kernels. Of course, this number is the theoretical maximum that does not take into account the parallelizability of the concrete functions and latency, so in a real system, the maximum amount of speedup that can be obtained, remains even lower.

5. Description of Relevant HEVC Components

In the previous section, the HEVC decoder was profiled and by analyzing the results, two main kernels were found: the inverse transform and the interpolation filter. In this section, the theoretical background information for those computational kernels in the context of HEVC is provided.

5.1. Inverse Transform

5.1.1. HEVC Transform

HEVC uses four different sizes of transforms: 4x4, 8x8, 16x16 and 32x32 luma samples, depending on the TB in use. For computing the transform, HEVC usually utilizes an integer DCT-based transform, but in the case of 4x4 luma TBs in intra-picture predicted frames, an integer DST-based transform is used.

The reason behind using two different transforms is that DST gives about 1% bit-rate reduction in case of 4x4 intra-picture predicted TBs [3]. For all the other cases, the difference between DST- and DCT based transforms is marginal [3], so DCT is used, mostly because there are more computationally efficient algorithms available.

5.1.2. HEVC 2-D Inverse Transform

If the transform coefficients matrix that is outputted from the quantization and scaling phase is defined as d[x][y], then the calculation of the residuals matrix r[x][y] is performed in the following way [8]:

1. A 1-D inverse transform is performed for each column of the coefficients matrix d[x][y], resulting in an intermediate values matrix e[x][y].
2. To ensure, that all the intermediate values can be stored in a 16-bit variable (in case of 8-bit video decoding), 7-bit right shift is performed on the e[x][y] matrix followed by 16-bit clipping operation as shown below:

$$g[x][y] = clip(-32768, 32767, (e[x][y] + 64) >> 7) \qquad (2)$$

3. Finally, a 2nd 1-D inverse transform is performed on the rows of the g[x][y] matrix that was calculated in the previous point, ending up with the residuals final matrix r[x][y]

5.1.3. 1-D Inverse Transform

The equations for both the IDCT- and IDST-like transforms are pretty much the same, although there are some differences in the usage of the transform matrix as for IDCT only a 32x32 matrix (appendix 1) is defined [3] and for smaller transforms subsampling of that is matrix is used. The IDST-like transform uses its specific matrix (appendix 1). The equations for both the IDCT (4) and IDST (3) can be found below [8].

Variables used by the equations (3) and (4) with their explanations:

- n – size of the transform
- $transMatrix_{i,j}$ – transform matrix (appendix 1)
- x_j – input vector for the 1-D transform
- y_j – output vector for the 1-D transform

$$y[i] = \sum_{j=0}^{n-1} (transMatrix[i][j] * x[j]), \tag{3}$$

$$Where\ i,j = 0 \ldots n$$

$$y[i] = \sum_{j=0}^{n-1} (transMatrix[i][k] * x[j]), \tag{4}$$

$$Where\ i,j = 0 \ldots n, k = (1 << (5 - Log_2(n))) * j$$

Properties and comparison of the equations for IDST-like transform (3) and IDCT-like transform (4):

- The relatively complex formula for calculating variable k in equation (4) is caused by the fact that for the IDCT-like transform only 32x32 transform matrix is defined [3] and subsampling of that matrix is required for smaller transforms.

In case separate matrices are used for different sized transforms, equation (4) becomes identical to equation (3).

- If the result of the inverse transform of a column with index n ∈ ℝ is written to the row of the output matrix with index n ∈ ℝ, the matrix is automatically transposed in the process. This means that the exact same equation (and the exact same software function, hardware etc.) can be used to calculate both of the 1-D inverse transforms without having to do an additional transposing operation on the intermediate matrix between the inverse transforms.
- For implementation of the IDCT, the partial butterfly algorithm [23] could also be used to reduce computational complexity.

5.2. Interpolation Filter

5.2.1. Interpolation

Interpolation is a method for increasing the sample rate of a signal by adding samples in the signal. Interpolation can be characterized by the interpolation factor L, which describes the ratio between the input signal and the interpolated signal. Interpolation consists of two steps:

1. Zero-stuffing: the sample rate of the signal is increased by adding L-1 zero-valued samples between every sample of the original signal.
2. Interpolation: a low-pass filter that removes the very high frequency components in the signal that are introduced by zero-stuffing, thus replacing the added zeroes with appropriate intermediate values. The output signal of the interpolation process looks almost exactly like the input, but with L times higher sample rate. Usually, a FIR filter is used for the implementation of an interpolation filter.

5.2.2. Finite Impulse Response Filter

The FIR filter is a type of filter that has a finite impulse response, which means that given an impulse (or any other signal with a finite length), the filter settles to zero in a finite time.

As can be seen from the FIR filter's equation (5) and from its dataflow diagram (Figure 14), the output value of an Nth order FIR depends on the last N values of its input vector. The input x[n] is shifted through the delay line. The signal after every delay element Z^{-1} (every value in the previous N time steps) is multiplied by a filter coefficient (b_i). Finally, all the multiplication results are added together.

As it can be seen from equation (5) and the dataflow diagram in Figure 14, the behavior of the filter depends on the filter coefficients. By setting the coefficients correctly, it is possible to change the filter's output value based on how the input signals change in time.

An example of the scenario described above would be making the outputs of the different taps of the filter to cancel each other out if the change rate of the signal in time is too high, but otherwise copy the signal to the output. This is essentially a low pass filter, letting through lower frequencies, but cancelling out higher ones.

$$y[n] = \sum_{i=0}^{N} b_i * x[n-i], \tag{5}$$

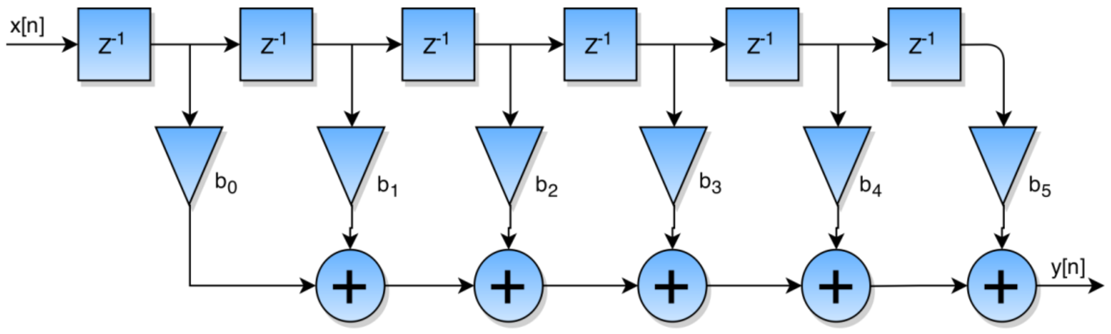


Figure 14. Dataflow diagram of a 6 tap FIR filter

This is also the reason why it is possible to use the FIR filter in the interpolation process to smooth out large changes in the signal that were created by zero-stuffing. Basically, by inserting zeroes into the signal, spectral changes are introduced. The new signal would be largely zeroed out, but would have sudden peaks located on the multiples of the interpolation factor L. This kind of signal can be considered as a sum of a signal identical to the original signal before zero-stuffing, but with L times higher sample rate and another signal with a very high frequency.

If a low-pass filter, that blocks only those very high frequencies is applied to the zero-stuffed signal, the output of the filter is a signal with L times higher sample rate, but otherwize almost identical to the initial signal.

5.2.3. Interpolation in HEVC

In HEVC the main section where interpolation is used is the inter-picture prediction. As mentioned in section 3.1, inter-picture prediction relies on temporal changes in a video stream for predicting frames. For each PB, the encoder searches the reference pictures for another PB, which best matches the current block. A motion vector that describes the relative displacement of the reference block compared to the current one is generated and added to the HEVC bitstream.

The problem with this kind of approach is that any motion in a real world (and thus in a video stream) does not follow the partitioning of a digital video sequence. It means that integer motion vectors (i.e., the object has moved exactly an integer number of blocks) are very rare. In case of a fractional motion vector, the reference block is interpolated to also generate the prediction samples for fractional positions. [3]

For luma channels, HEVC supports motion vectors with the precision of one-quarter luma samples. The accuracy of motion vectors used for chroma samples depends on the chroma subsampling that is used. For the most typical case of 4:2:0 subsampling, the precision of one eighth of the chroma samples is supported. [3]

Table 1. FIR filter coefficients for luma fractional interpolation [3]

Index i	**-3**	**-2**	**-1**	**0**	**1**	**2**	**3**	**4**
hfilter[i]	-1	4	-11	40	40	-11	4	1
qfilter[i]	-1	4	-10	58	17	-5	1	-

The acquisition of fractional luma samples needed for motion prediction is done in HEVC by using an 8-tap FIR filter for the half-sample positions and a 7-tap filter for the quarter-sample positions. The coefficients for the interpolation filter in case of luma samples are shown in Table 1, where hfilter[i] and qfilter[i] specify the coefficients with index i respectively for the half-sample position and the quarter-sample position luma interpolation filters.

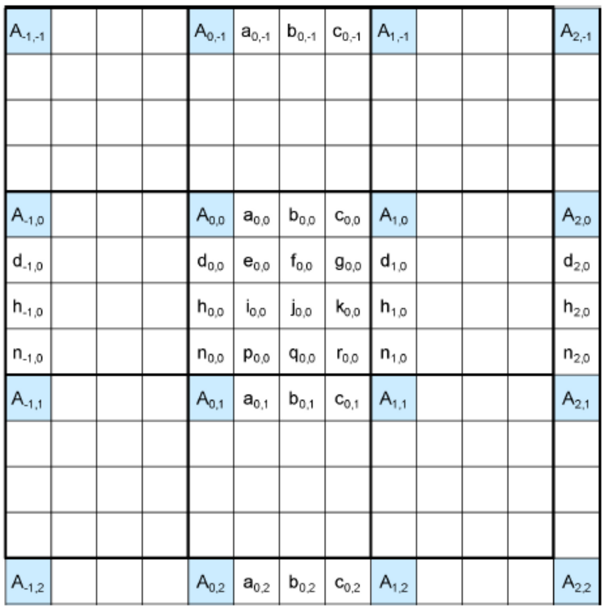


Figure 15. Sample positions in luma interpolation [3]

The samples $a_{0,j}$, $b_{0,j}$, $c_{0,j}$, $d_{0,0}$, $h_{0,0}$, and $n_{0,0}$ in Figure 15 are generated from the integer samples $A_{i,j}$ by applying interpolation using the interpolation filters described above. The rest of the non-integer samples can then be interpolated from the vertically neighboring previously generated samples $a_{0,j}$, $b_{0,j}$ and $c_{0,j}$. The interpolation filtering for luma samples is done by using the following filtering equations [3]:

$$\begin{aligned}
a_{0,j} &= \left(\textstyle\sum_{i=-3..3} A_{i,j}\,\mathrm{qfilter}[i]\right) >> (B-8)\\
b_{0,j} &= \left(\textstyle\sum_{i=-3..4} A_{i,j}\,\mathrm{hfilter}[i]\right) >> (B-8)\\
c_{0,j} &= \left(\textstyle\sum_{i=-2..4} A_{i,j}\,\mathrm{qfilter}[1-i]\right) >> (B-8)\\
d_{0,0} &= \left(\textstyle\sum_{i=-3..3} A_{0,j}\,\mathrm{qfilter}[j]\right) >> (B-8)\\
h_{0,0} &= \left(\textstyle\sum_{i=-3..4} A_{0,j}\,\mathrm{hfilter}[j]\right) >> (B-8)\\
n_{0,0} &= \left(\textstyle\sum_{j=-2..4} A_{0,j}\,\mathrm{qfilter}[1-j]\right) >> (B-8)
\end{aligned}$$

$$\begin{aligned}
e_{0,0} &= \left(\textstyle\sum_{v=-3..3} a_{0,v}\,\mathrm{qfilter}[v]\right) >> 6\\
f_{0,0} &= \left(\textstyle\sum_{v=-3..3} b_{0,v}\,\mathrm{qfilter}[v]\right) >> 6\\
g_{0,0} &= \left(\textstyle\sum_{v=-3..3} c_{0,v}\,\mathrm{qfilter}[v]\right) >> 6\\
i_{0,0} &= \left(\textstyle\sum_{v=-3..4} a_{0,v}\,\mathrm{hfilter}[v]\right) >> 6\\
j_{0,0} &= \left(\textstyle\sum_{v=-3..4} b_{0,v}\,\mathrm{hfilter}[v]\right) >> 6\\
k_{0,0} &= \left(\textstyle\sum_{v=-3..4} c_{0,v}\,\mathrm{hfilter}[v]\right) >> 6\\
p_{0,0} &= \left(\textstyle\sum_{v=-2..4} a_{0,v}\,\mathrm{qfilter}[1-v]\right) >> 6\\
q_{0,0} &= \left(\textstyle\sum_{v=-2..4} b_{0,v}\,\mathrm{qfilter}[1-v]\right) >> 6\\
r_{0,0} &= \left(\textstyle\sum_{v=-2..4} c_{0,v}\,\mathrm{qfilter}[1-v]\right) >> 6.
\end{aligned}$$

Figure 16. Luma interpolation filtering equations [3]. Variable B is the bit depth of the reference samples and ">>" denotes an arithmetical shift right operation.

The reason behind the shifting operation in the equations shown above is to scale the result down so that it would fit into 16-bit variables (or registers, in case of hardware implementation).

In the most usual case, when the bit width of the HEVC video samples is eight (like in this thesis), the filtering operations are separable.

Table 2. Chroma interpolation filter coefficients in case of 4:2:0 chroma subsampling

Index i	**-1**	**0**	**1**	**2**
filter1[i]	-2	58	10	-2
filter2[i]	-4	54	16	-2
filter3[i]	-6	46	28	-4
filter4[i]	-4	36	36	-4

The process of interpolating chroma samples is very similar to the interpolation of luma samples. As mentioned earlier, in the typical case of 4:2:0 chroma subsampling, the motion prediction is done on 1/8th of chroma sample level. In this case, HEVC defines four different four-tapped FIR filters with coefficients shown in Table 2. In the table the filters are shown as filterN[i], where N = 1, …, 4 with i being the index of the coefficient shown in the first row.

The fractional chroma samples of 1/8th, 2/8th, 3/8th and 4/8th are interpolated using filter1[i], filter2[i], filter3[i] and filter4[i], where i = -1, …, 2, respectively. For the remaining fractional chroma samples of 5/8th, 6/8th and 7/8th the mirrored values of the first three filters (filter3[1−i], filter2[1−i], and filter1[1−i], where i = -1, …, 2, respectively), are used [3].

6. Analysis of the HEVC Sample Software

In section 4 the HEVC decoder was profiled and the computational kernels were extracted, while in section 5 a theoretical information about the kernels was given. In this section, the HEVC sample software is analyzed to find the individual C++ functions where these kernels are implemented in order to find the interfaces, where to extract the testing data for the next step. Special attention is given to the inverse transform, as this is the kernel that is implemented during this thesis as a proof of concept.

One of the main tools used in this section is the profiling data generated in section 4. The call graphs produced during the profiling step show exactly which function is calling which function, so they are a map for orienting in the source code to find the relevant functions.

Simultaneously, as the source code is mostly based on the HEVC text specification [8] and is relatively unoptimized, the text specification can help in understanding how the functions themselves work.

6.1. Interpolation filter

To find the implementation of the interpolation filter, first it is necessary to take a look at the call graph that was generated call graph by using the KCachegrind tool. The call graph (Figure 17) shows that there are two different functions for interpolation filter: one for horizontal filter and another for the vertical filter, both part of a class TComInterpolationFilter. In addition, they are both called from the function xPredInterBlk (inter-picture block prediction) that is in a class TComPrediction.

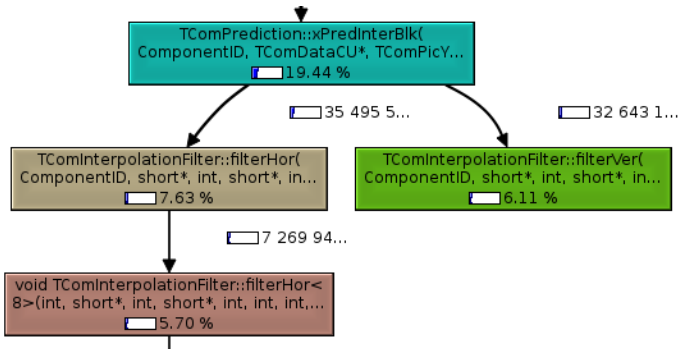

TComPrediction::xPredInterBlk(ComponentID, TComDataCU*, TComPicY...
19.44 %
35 495 5...
32 643 1...
TComInterpolationFilter::filterHor(ComponentID, short*, int, short*, in...
7.63 %
TComInterpolationFilter::filterVer(ComponentID, short*, int, short*, in...
6.11 %
7 269 94...
void TComInterpolationFilter::filterHor< 8>(int, short*, int, short*, int, int, int,...
5.70 %

Figure 17. Call graph for the interpolation filter

```
Void TComInterpolationFilter::filterVer()
{
  if ( fracturalSampleOffset == 0 )
  {
   //Unit FIR filter
    filterCopy()
  }
  else if (isLuma(compID))
  {
    filterVer<NTAPS_LUMA>();
  }
  else
  {
    filterVer<NTAPS_CHROMA>();
  }
}
```

Figure 18. Cleaned up pseudocode of the filterVer() function

In order to find out how the interpolation filter is implemented, analyzing the structure of the functions filterHor() and filterVer() is needed. The structure of both of the functions is almost identical.

A cleaned up pseudocode version of the function filterVer() is shown in Figure 18.As it can be seen in the figure, first it checks, if the fractural sample offset is 0. This means, that no interpolation filtering is necessary because the current motion vector is integer-sized and a unit FIR filter is applied, that just copies the input samples into the output without doing any real filtering.

However, in most cases, interpolation filtering is needed, as integer motion vectors are very rare (subsection 5.2.3). In this case, a vertical filter function template with the correct number of taps (based on whether the current block is a luma or a chroma block), is executed. The function template filterVer<N>(), where N represents the number of taps of the filter, is actually just a wrapper for another function template called TComInterpolationFilter::filter(). The filter() function is in turn a function template of an FIR filter described in subsection 5.2 with an adjustable number of taps.

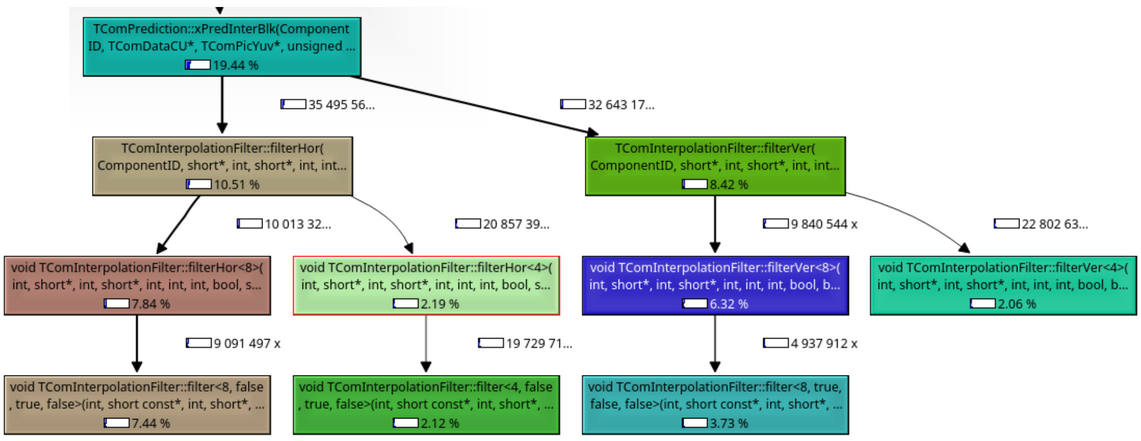

TComPrediction::xPredInterBlk(Component ID, TComDataCU*, TComPicYuv*, unsigned ...
19.44 %
35 495 56...
32 643 17...
TComInterpolationFilter::filterHor(ComponentID, short*, int, short*, int, int...
10.51 %
TComInterpolationFilter::filterVer(ComponentID, short*, int, short*, int, int...
8.42 %
10 013 32...
20 857 39...
9 840 544 x
22 802 63...
void TComInterpolationFilter::filterHor<8>(int, short*, int, short*, int, int, int, bool, s...
7.84 %
void TComInterpolationFilter::filterHor<4>(int, short*, int, short*, int, int, int, bool, s...
2.19 %
void TComInterpolationFilter::filterVer<8>(int, short*, int, short*, int, int, int, bool, b...
6.32 %
void TComInterpolationFilter::filterVer<4>(int, short*, int, short*, int, int, int, bool, b...
2.06 %
9 091 497 x
19 729 71...
4 937 912 x
void TComInterpolationFilter::filter<8, false , true, false>(int, short const*, int, short*, ...
7.44 %
void TComInterpolationFilter::filter<4, false , true, false>(int, short const*, int, short*, ...
2.12 %
void TComInterpolationFilter::filter<8, true, false, false>(int, short const*, int, short*, ...
3.73 %

Figure 19. Relative instruction usage of the interpolation filter

By looking deeper into the call graph by using the KCachegrind tool (Figure 19), it is easy to see that a very big majority of all the instructions used in both the horizontal and the vertical filters are actually consumed by the TComInterpolationFilter::filter() function.

TComInterpolationFilter::filter() is a universal FIR filtering function that is capable of working with any number of taps allowed by the HEVC standard and can thus be used for both vertical- and horizontal filtering. Since this single function takes up a big amount of the instructions used by the entire interpolation filter, moving it to a hardware-based accelerator would increase the decoding speed considerably.

6.2. Inverse transform

As it was also done for the interpolation filter, the analysis of the inverse transform also begins with analyzing the call graph. It can be seen on the inverse transform part of the HEVC decoder’s call graph (Figure 20) that there are four nested functions that are responsible for the inverse transform. The lowest function hints that the inverse transform is implemented using the partial butterfly algorithm [23], which makes sense, as it has lower computational complexity and thus it can perform faster and takes less resources.

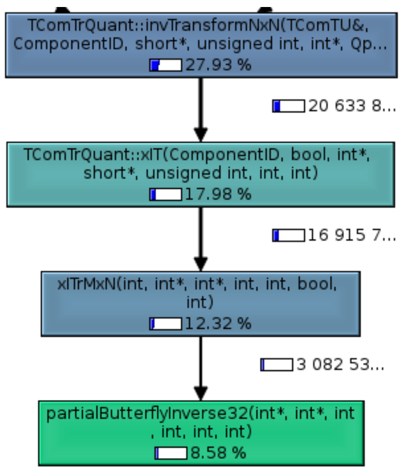

TComTrQuant::invTransformNxN(TComTU&, ComponentID, short*, unsigned int, int*, Qp...
27.93 %
20 633 8...
TComTrQuant::xIT(ComponentID, bool, int*, short*, unsigned int, int, int)
17.98 %
16 915 7...
xITrMxN(int, int*, int*, int, int, bool, int)
12.32 %
3 082 53...
partialButterflyInverse32(int*, int*, int , int, int, int)
8.58 %

Figure 20. A part of the call graph showing the inverse transform

```
int xITrMxN(int coeff[][], int maxTrDynamicRange, int bitDepth){

  int tmp[][], residual[][];

  int shift_1st = 7;
  int shift_2nd = (7 + maxTrDynamicRange - 1) - bitDepth;

  tmp = executeTransform(coeff, shift_1st);
  residual = executeTransform(tmp, shift_2nd);

return residual;
```

Figure 21. Pseudocode for the xITrMxN() function

Figure 21 shows the simplified pseudocode for the xITrMxN() function. This function just gets the coeff matrix as an input and then executes the same transform function on it twice. It also calculates the proper shifting values needed by the HEVC standard (equation 1) and sets them as parameters for the inverse transform function. The clipping part of equation (2) is handled by the called functions themselves.

The function shown in Figure 21 is somewhat simplified and does not show the actual inverse transform execution functionality. The real function in HEVC decoder's sample software also checks for picture prediction type and the current transform's size and executes either an inverse partial butterfly function with a relevant size or, in case of 4x4 intra-picture prediction, the inverse DST function.

The inverse partial butterfly functions that are called by xITrMxN() just implement the regular inverse partial butterfly algorithm [23] with some additions defined by HEVC (equation 1). The inverse DST function simply implements the matrix multiplication (equation 3) of that function as DST lacks the butterfly structure.

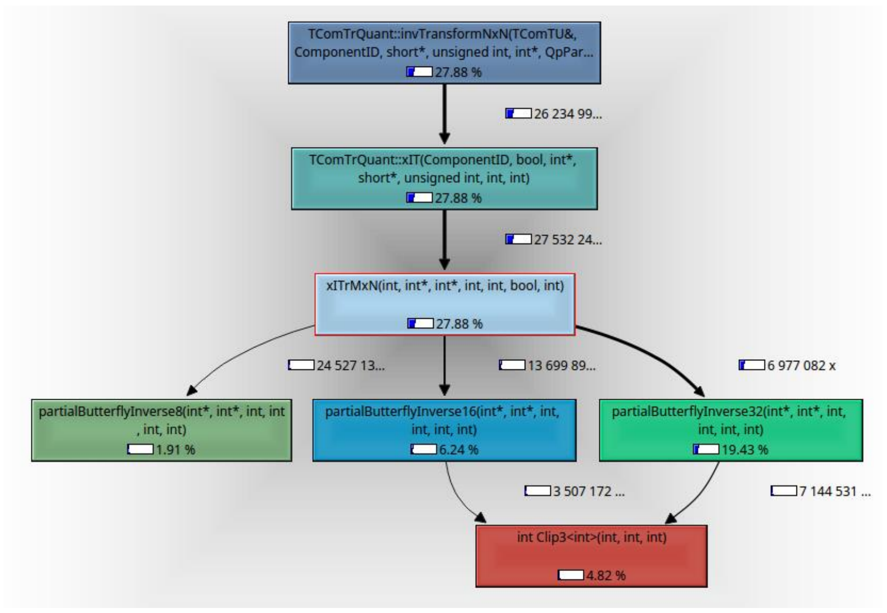

TComTrQuant::invTransformNxN(TComTU&, ComponentID, short*, unsigned int, int*, QpPar...
27.88 %
26 234 99...
TComTrQuant::xIT(ComponentID, bool, int*, short*, unsigned int, int, int)
27.88 %
27 532 24...
xITrMxN(int, int*, int*, int, int, bool, int)
27.88 %
24 527 13...
13 699 89...
6 977 082 x
partialButterflyInverse8(int*, int*, int, int , int, int)
1.91 %
partialButterflyInverse16(int*, int*, int, int, int, int)
6.24 %
partialButterflyInverse32(int*, int*, int, int, int, int)
19.43 %
3 507 172 ...
7 144 531 ...
int Clip3<int>(int, int, int)
4.82 %

Figure 22. Relative amount of instructions used by different parts of the inverse transform.

In Figure 22, it can be seen that a large majority of the instructions used for the inverse transform can be divided between three inverse partial butterfly functions. Meanwhile, both the DCT-like and DST-like 4x4 inverse transforms take up only around 1% of all the instructions that are used in the inverse transform.

In the light of the information provided by Figure 22, it can be mentioned that by moving the implementation of at least three largest DCT-like inverse transform functions on hardware, together with the clipping functionality, a considerable performance increase for the HEVC's inverse transform can be achieved.

7. Dynamically Reconfigurable Resource Array

DRRA or the Dynamically Reconfigurable Resource Array was described by Muhammad Ali Shami in his PhD thesis [4] in 2012 as a reconfigurable DSP platform as a part of a design template to host the signal processing parts of different wireless applications [4].

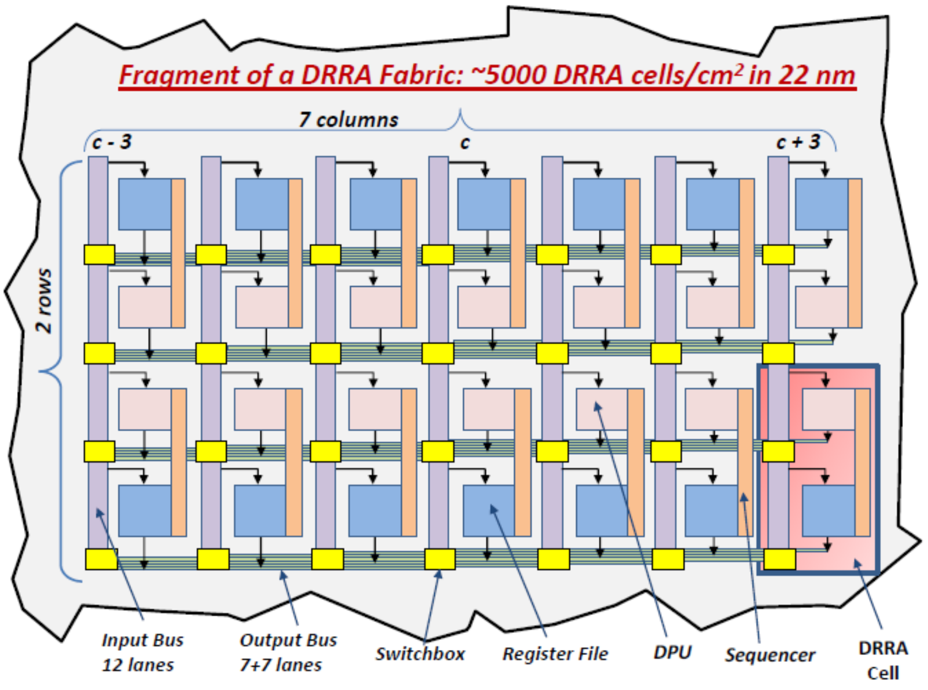


Figure 23. Fragment of the DRRA fabric [4]

7.1. DRRA Fabric

DRRA's fabric is organized into DRRA cells. The current version of DRRA has 10 cells divided between 2 rows and 5 columns [24]. Each cell consists of a DPU, register file and a sequencer. The cells are arranged in a scalable matrix and connected together with an interconnection network consisting of horizontal and vertical busses. The horizontal busses are used as outputs for the DPUs and the register files inside the DRRA cells. The horizontal busses are crossed by vertical busses that act as inputs for the computational elements inside the cells (Figure 23) [4].

On the intersections of the input and output busses, switchboxes, that are used for connecting them, are located (colored yellow in Figure 23). The connections between the busses can be dynamically configured inside the switchboxes. The unconnected lines are isolated from each other using tri-state logic elements. By correct configuration of the switchboxes, it is also possible to utilize multiple DPUs simultaneously in order to create parallelism in the calculations [4].

The configuration data for all the switchboxes in a column is stored in a special configuration memory. The sequencers (colored as orange in Figure 23) of that column can access this memory. Upon initialization, each sequencer configures the local switchboxes to connect the DPU and register file in its cell to correct input- and output busses [4].

Alternatively, during the execution time, a reconfiguration of the switchboxes can also be made in order to, for example, satisfy the need of conditional statements that, based on some condition, may require changing the connections of the inputs and outputs of the computational elements inside the DRRA cell [4].

7.2. DRRA Cell

As described earlier in this thesis, the basic element of a DRRA fabric is the DRRA cell, where all the actual processing takes place. The DRRA cell consists of three main components: (Figure 23) the Data-path unit, which is used for data processing, the register file that is used for storing temporary data and a sequencer for configuring the cell and the switchboxes that are neighboring it.

7.2.1. Data-Path Unit

The DPU is the component in the DRRA cell, where the actual data processing takes place. The DPU used in the DRRA has four inputs and two outputs, the inputs are connected to the closest vertical bus of the DRRA fabric and the outputs to the closest horizontal bus. It also has a two-way connection to the local sequencer, getting the configuration information from the sequencer and sending back different exceptions and other status related data [4].

The inputs and outputs of a DPU can either work separately or act in a group of two in order to represent complex numbers. In the last case, one of the input or output pins of the pair would represent the real part of the complex number and the other one the imaginary part. In case this mode is used, two complex number can be represented on the input of the DPU and one on the output [4].

In the original version [4], the DPU included three partitions:

- *A logic partition*, that included*:*
  - Comparison
  - Logic and arithmetic shifting
  - LSFR
  - Logic AND and OR operations
- *An arithmetic partition*, supporting the combinations of addition, subtraction and multiplication.
- *A post processing partition* that included saturation and truncation of the DPU output.

At the time of writing of this thesis, the DRRA fabric was in the process of being updated and the code for the DPU had been partly rewritten with some of the initial functionality removed. Unfortunately, only very little documentation describing the changes to the DRRA's source code had been made available by its authors.

7.2.2. Register file

The register file is used in the DRRA cell to provide a fast parallel memory access to the DPU. The register file has two ports for reading- and two for writing that are connected to the DRRA's interconnection network, so the contents in the register file are also available for DPUs in other DRRA cells. In the current version of DRRA, each register file can contain 64 words of 32-bit data [4] [24].

7.2.3. Sequencer

In DRRA, besides the DPU and the register file, the DRRA cell also includes a sequencer. The sequencer is being used to configure the DPU, the register file and the fabric's interconnection network in a way, that it would implement the desired algorithm. The sequencer is also responsible for reconfiguring the aforementioned components when needed [4].

The sequencer act as a simple FSM that controls the functioning of all the other components inside the DRRA cell. The main components of the sequencer are the program memory, which holds the instructions for the sequencer, the instruction decoder that decodes the instructions stored in the program memory, the program counter and a simple FSM for maintaining the program counter [4].

Besides the aforementioned functionality, the sequencer also is used for other tasks, that are not relevant to this thesis and the reader is guided to [4] for any additional information on this topic.

7.3. VESYLA

VESYLA is a compiler for DRRA. It is reads specially formatted MATLAB code and maps the algorithm into the DRRA fabric by generating a VHDL test bench, that can be used to configure and simulate the fabric [24].

7.3.1. Pragmas

a) **Register file allocation:**
   Pragma:
   %! RFILE<> [row_index, col_index]
   Example:
   p = [5:15]; %! RFILE<> [0:1]

b) **DPU allocation:**
   Pragma:
   %! DPU [row_index, col_index]
   Example:
   z(5) = y(4j) + z(2); %! DPU [0:0]

Figure 24. Pragmas used by VESYLA

Since VESYLA currently lacks automatic resource allocation mechanism [24], the user allocates resources (register files and the DPUs) manually by adding special pragmas (Figure 24) that specify the type of the allocated resource (register file or DPU) and its location on the DRRA's fabric to the MATLAB code as specially formatted comments.

7.3.2. VESYLA Folder Structure

VESYLA’s directory tree (Figure 25) contains two top-level folders: “VESYLA” and “workspace”. The first one contains the VESYLA compiler. For compilation with VESYLA, a code is in the VESYLA folder’s root, while the compilation results end up in the “code” subfolder. The DRRA fabric and testing related files are located in the “workspace” folder.

```
vesila_toplevel
├───VESYLA              <- VESYLA’s input- and executable's folder
│   ├───code            <- VESYLA output folder
└───workspace           <- DRRA workspace
    ├───fabric          <- DRRA fabric
    │   ├───DiMArch_RTL
    │   ├───mtrf
    │   ├───pce
    ├───testbenches     <- DRRA test bench directory
    │   ├───template    <- Tesbench template files
    │   │   └───result
    │   ├───test1       <- An example test bench
    │   │   └───results
    └───test_util       <- Utilities for profiling
```

Figure 25. VESYLA's directory tree

7.3.3. Known Limitations of VESYLA

At the time of writing this thesis, VESYLA was still in a very early stage of development and had many limitations. In this section, the known limitations are listed, both the ones found in the VESYLA user manual [24] and the ones discovered while working on this thesis. Following limitations are taken into the account while using VESYLA:

- Addition, multiplication, and their combinations are only supported operations.
- Conditional statements are not supported.
- Operations with vectors cannot be done inside loops.
- Functions cannot be used in the MATLAB code.
- Right now, there is no memory support in VESYLA, which also means that it also lacks any support for multidimensional arrays. All memory-related operations have to be done using the register files.

- The register files only can be initialized to growing arrays of numbers.

VESYLA can still be used to map some of the more complex parts of the algorithm to the DRRA fabric. To overcome the problems described above, a more abstract version of the algorithm could be written in MATLAB, then compiled using VESYLA and finalized by making some final modifications directly in the test bench on the assembler level.

7.4. Mapping Process

The process of mapping an algorithm on the DRRA's fabric using VESYLA should be the following:

1. The algorithm should be described in MATLAB
2. A simplified version of the algorithm, described in the previous point of this list, should be made for more complex algorithms, by taking into account the VESYLA's limitations that were explained in subsection 7.3.3.
3. Pragmas should be added to the simplified version of the code (subsection 7.3.1) in order to tell VESYLA how to allocate the resources needed for the mapping process
4. The m-file with the VESYLA-readable algorithm should be put into the VESYLA folder (Figure 25)
5. At this point, the VESYLA executable can be run with the m-file with the algorithm as a parameter. If the compilation is successful and no errors are detected, the test bench that maps the algorithm to the DRRA fabric together with some supporting files are put into the "/VESYLA/code' folder (Figure 25).
6. Next, the generated test bench is simulated. VESYLA already includes the scripts needed for simulation and profiling the result, so the reader is directed to the VESYLA user manual [24] for details.
7. Finally, the simulation and profiling results can be compared with the results from the MATLAB code. For this reason, VESYLA provides the profiler results and an instrumented version of the original MATLAB code. The profiling results of the fabric simulation and the MATLAB code can then be compared to verify their equivalence. Unfortunately, the profiling results generated are in different format and thus not machine-comparable.

8. After the verification of equivalence, the implementation can be modified in the test bench to bypass some of the VESYLA’s limitations, if needed, and then verified using some external method.

8. Implementation Flow

8.1. Test Cases Generation

Test cases for the implementation were generated from the sample software by analyzing the structure of the inverse transform functionality in the code. This information was used to understand how the inverse transform functions communicate with the rest of the code, in order to find interfaces, where it would be possible to separate the two, so that the inverse transform code could be replaced by a hardware accelerator. This interface was then used to dump all the data passing through it into a test bench file for the MATLAB-based implementation of the inverse transform that is described in more detail the next subsection.

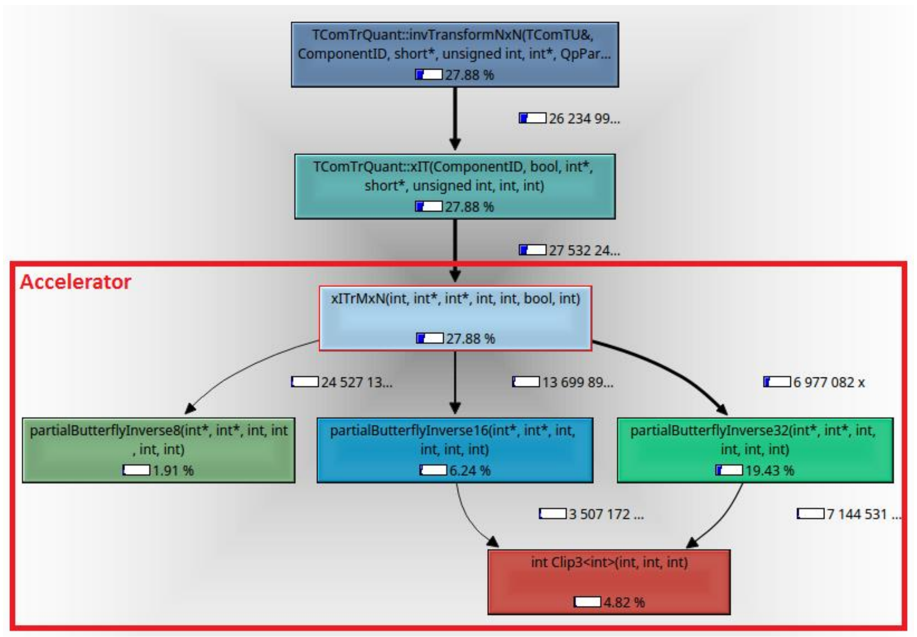

TComTrQuant::invTransformNxN(TComTU&, ComponentID, short*, unsigned int, int*, QpPar...
27.88 %
26 234 99...
TComTrQuant::xIT(ComponentID, bool, int*, short*, unsigned int, int, int)
27.88 %
27 532 24...
Accelerator
xITrMxN(int, int*, int*, int, int, bool, int)
27.88 %
24 527 13...
13 699 89...
6 977 082 x
partialButterflyInverse8(int*, int*, int, int , int, int)
1.91 %
partialButterflyInverse16(int*, int*, int, int, int, int)
6.24 %
partialButterflyInverse32(int*, int*, int, int, int, int)
19.43 %
3 507 172 ...
7 144 531 ...
int Clip3<int>(int, int, int)
4.82 %

Figure 26. Inverse transform functions that should be mapped on the accelerator

The process of finding the interface was done by using results from the analysis of the code that was done in subsection 6.2. The most obvious choice for the interface was the xITrMxN() function (Figure 26), as this is the top-level inverse transform function. By implementing this, and every function it calls, on an accelerator, a complete hardware representation of the entire HEVC’s inverse transform functionality, can be built (see subsection 6.2).

```
/** MxN inverse transform (2D)
*  \param bitDepth           [in]  bit depth
*  \param coeff              [in]  transform coefficients
*  \param block              [out] residual block
*  \param iWidth             [in]  width of transform
*  \param iHeight            [in]  height of transform
*  \param useDST             [in]  Wheather to use DST
*  \param maxTrDynamicRange  [in]
*/
```

Figure 27. cITrMxN() function parameters

The best way to get test data is to dump out every input and output parameter (Figure 27) of the xITrMxN() function. The most suitable place for this is in the higher level function TComTrQuant::xIT(), which is calling the xITrMxN() function (Figure 26). This function was identified before as the function to be implemented on the accelerator.

The data is dumped from the accelerator interface into a text file that uses the MATLAB m-file syntax. This means that the generated file is directly runnable in MATLAB by using the input matrices and parameters defined in the test file to run a testing function, which is described in detail in the next section, and checking its output against the correct output matrix that is also dumped from the sample C++ decoder. This way it can be ensured, that the MATLAB implementation of the inverse transform is functionally identical to the one of the example transform.

An example test case for testing (16x16 inverse transform) can be found in Appendix 3. In reality, each test file can include hundreds or even thousands of test cases for different transform sizes and types.

8.2. MATLAB Implementation

The HEVC’s inverse transform kernel that consists of the functions identified before (Figure 26) were realized in MATLAB to serve as a proof of concept implementation and a launching pad for implementing it on DRRA. The MATLAB implementation was done in a way, that it follows the inverse transform interface of the C++ sample code and accepts the parameters of the xITrMxN() function (Figure 27).

The MATLAB implementation of the HEVC transform consists of four parts: the testbench that is described in more detail in the previous subsection, the verification code, a top-level inverse transform class and a class consisting of 1-D inverse partial butterfly algorithms.

The code for verification of the inverse transform is used to verify the MATLAB implementation of the inverse transform kernel with the testing samples provided by the testbench. It does that by running the inverse transform using the inputs provided by the testbench and then comparing the result with the correct output (code in appendix 6).

The verification code calls the top-level inverse transform class (code in appendix 5), that figures out which type of transform should be used and then runs the individual 1-D transforms. It also takes care of the HEVC specific parts of the transform, like shifting and clipping of the values (subsection 5.1.2). This class also includes a function for the 4x4 inverse DST-like transform. For IDCT-like transform, the inverse partial butterfly functions from a separate class are called. The code for the inverse partial butterflies class can be seen in appendix 4.

8.3. VESYLA-Readable Code

In this subsection, a proof of concept VESYLA-readable version of the 4x4 inverse butterfly for calculating the inverse DCT-like transform is presented. Code for both, the parallel and sequential implementation are shown and their speed is analyzed.

In order to make it VESYLA-readable, user alters the code. First, it is necessary to break the code down into smaller, elementary operations. The 4x4 inverse butterfly can by divided into different calculation blocks as shown in Figure 28.

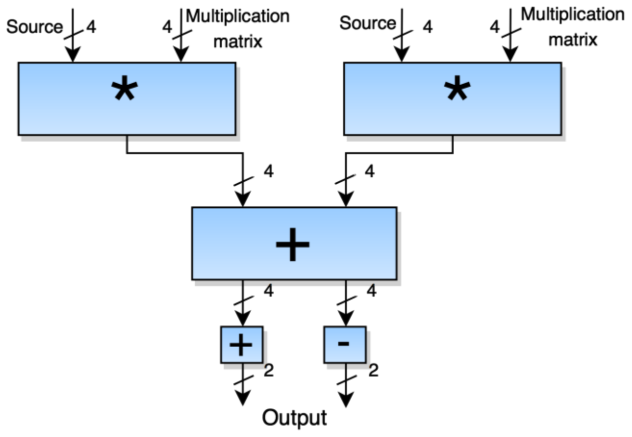


Figure 28. 4x4 inverse butterfly algorithm's flow diagram. It needs to be noted, that in last step, the addition and subtraction use the same inputs, so they cannot be implemented in parallel on DRRA.

One important aspect, when talking about hardware accelerators, is their parallelization capabilities, how much parallelism can the hardware implementation allow, the more parallelization is used, the faster is the implementation is (with some limitation, see section 4.3.2).

As described in subsection 7.2.2, because a register file in DRRA has only two input- and two output ports, only two different writing and two reading operations can be done simultaneously on one register file. This means, that in order to write into the register files, the code has to be partitioned so, that only two writing and two reading operations of a single register are done simultaneously.

Concurrency in a VESYLA-readable MATLAB code is done using for loops. Concurrent statements go inside the loop. If the there are multiple for loops in the MATLAB code, the contents of each loop is calculated concurrently on the DRRA, while the loops themselves are executed sequentially, one after another. The code for the implementation of the algorithm that is accessing a single register file at a time can be found in appendix 7.

This way, however, it is not possible to reach the maximum possible speed by exploiting the parallelism that is found in the algorithm. As it can be seen from Figure 28, there are a lot of possibilities to parallelize the algorithm. This code could be partitioned only into four steps:

1. 8 concurrent multiplications
2. 4 concurrent additions
3. 2 concurrent additions
4. 2 concurrent subtractions (as VESYLA does not support subtraction, it can be modified into multiplication with -1 and addition)

However, by increasing the concurrency, a way to overcome the limit on concurrent register file usage is needed. This can be done by dividing the variables can be divided between multiple register files, so that instead of writing all the eight multiplication results into one register file, four register files could be used, each storing two values. This result of course sacrifices the memory for speed. This effect can, however, somewhat reduced by reusing the register files. A code describing the parallel approach can be found in appendix 8.

Every arithmetic operation on the DRRA consists of four steps:

1. Reading values from the register file (4 clock cycles)
2. Making the calculation (4 clock cycles usually, 3 in case of addition)
3. Soring the result in DPU’s output register (4 clock cycles)
4. Writing the result into a register file (4 clock cycles)

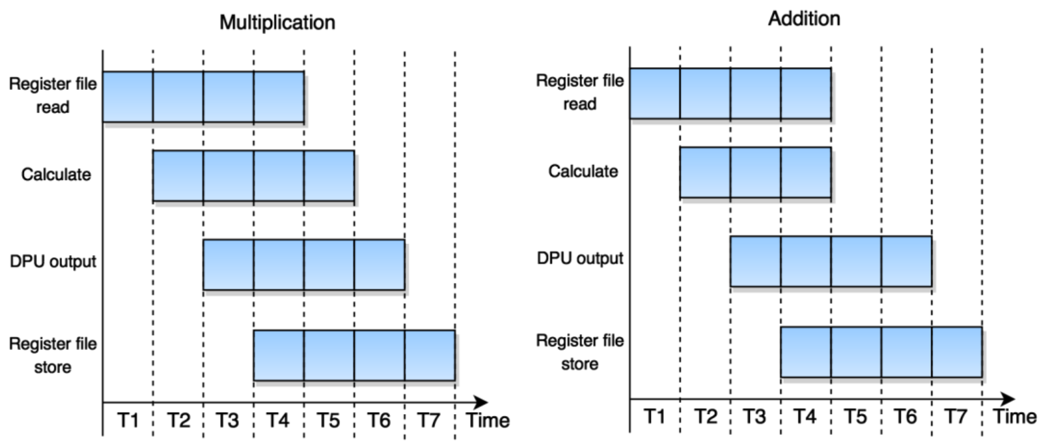


Figure 29. Calculation pipeline in DRRA (implements parallelism of level 4)

As it can be seen in Figure 29, all of the previously mentioned steps are pipelined, so that in any case, the answer in calculated in 7 clock cycles. By knowing that information, it is possible to calculate number of clock samples it the two different implementations on the algorithm. This can be done, by just counting the number of sequential instructions performed and multiplying the number by 7 clock cycles. For this reason, the MATLAB codes in appendixes 7 and 8 can be used. Moreover, based on the number of the clock cycles, the execution time can be measured if the clock speed of the hardware platform is known. Based on the constraints defined in [25], a reasonable estimate for the clock rate would be 450MHz, which would make the length of one cycle 2.222 ns. The information is available in table

Table 3. Comparison of VESYLA-readable implementations of the 4x4 inverse butterfly.

Implementation	**Number of sequential instructions**	**Clock cycles**	**Number of register files used**	**Calculated execution time**
Sequential: Using only two register file operations at once	8	56	5	124.432 ns
Fully Parallel	4	28	15	62.216 ns

As it can be seen in Table 3, by increasing the parallelization the 4x4 inverse butterfly algorithm two-fold decreases the time needed for the execution two times. However, this kind of implementation also increases register file usage three times. The problem with the excessive register file usage for parallel systems could be solved by using memory instead of register files to store data; however, unfortunately this is not an option at the moment, as the current version of VESYLA does not have memory support.

Table 3 also shows the inherent parallelism within the given kernel that the accelerator exploits by pipelining. Parallel implementation also exploits parallelism by allocating parallel resources. Hence, the inherent parallel resources allocated here are limited to 32, while the theoretical limit is 36. However, the maximum limitation of parallelism that can possibly be achieved on a 4x4 inverse DCT transform is 64. This is because in multiplication of two 4x4 matrices there are 16 multiplications in total that can all be made in parallel, made on a pipeline with parallelism of 4. However, for this thesis, the inverse butterfly algorithm was chosen, as it is computationally less complex (requires less elements) and applies parallelism of level 32. This is much closer to the theoretical limit of 36 than 64 that would be the level of parallelization of the matrix multiplication (subsection 4.3.2).

8.4. Verification of VESYLA Implementation

To verify the correctness of the VESYLA implementation, it is tested similarly to the MATLAB version. This is done by copying the code from VESYLA implementation into a function, similar to inverse butterfly functions for MATLAB implementation in appendix 4. Replacement of some of the register file declarations with function parameters is probably also necessary. Then, the similar testing method as the one used for the verification of the MATLAB implementation (section 8.2), can be used.

Conclusions

In this thesis, a detailed framework for the extraction of computational kernels for implementation on a hardware-based accelerator was proposed. To demonstrate the framework, the High Efficiency Video Coding (HEVC) decoder was used as the application. The Dynamically Reconfigurable Resource Array (DRRA) was used as the hardware platform for implementation.

The proposed framework consists of following steps:

1. Profiling of an application
2. Benchmarking of the application
3. Extraction of the kernels
4. Extraction of testing data from the original application
5. Implementation of the kernel in MATLAB
6. Equivalence checking in order to verify the functional equality of the MATLAB implementation of the kernel with the kernel
7. Implementation of the kernel on VESYLA
8. Verifying of the equality between VESYLA implementation with MATLAB model or the original kernel.

In this thesis, first the framework was discussed in detail, followed by a short overview of the HEVC codec that was used as an example for applying the implementation framework.

Then, the profiling and benchmarking of the HEVC decoder was done. The benchmarking results showed that 67.32% of the HEVC video decoder was parallelizable, while 56.22% of all the instructions in the decoder were carried out by just two kernels: the interpolation filter and the inverse transform, which were selected as the kernels for the implementation. A theoretical maximum parallelization and speedup that are achievable if those two kernels would be moved to a hardware accelerator were found. Next, some theoretical background about the kernels of interest was given.

Then the sample software was analyzed in order to find the functions containing the relevant kernels. Once found, the functions which should be replaced with the hardware accelerator to extract the kernels, were identified and were used to dump out real-world testing data into a MATLAB testbench for testing the implementation in the later phases.

Next, the chosen kernel, the inverse transform, was implemented in MATLAB and the generated test cases were used to check the equivalence of the MATLAB implementation with the kernels extracted from the application.

Finally, two implementations of the kernel using VESYLA with different levels of parallelism were given and then compared and the verification process of the VESYLA implementation was described.

Implementation of the inverse butterfly algorithm was chosen over matrix multiplication, because of two reasons. First, it is computationally less complex than the matrix multiplication, so less components are needed for the implementation. Secondly, although matrix multiplication has a theoretical maximum achievable parallelism of level 64, for a 4x4 inverse transform, the inverse butterfly algorithm has a maximum achievable parallelism of level 32. The later one is much closer to the theoretical maximum useful level of parallelism of level 36, which means, that the usage of more elements to achieve better parallelism is not sensible.

As a result of this thesis, now a detailed framework for finding the computational kernels in an application and implementing them on a hardware accelerator exists for DRRA hardware accelerator.

References

[1] "Intel's Hybrid CPU-FPGA," [Online]. Available: http://www.embeddedintel.com/commentary.php?article=2143. [Accessed 14 06 2015].

[2] J. L. Hennessy and D. A. Petterson, Computer Architecture A Quantitive Approach, Fourth Edition, San Francisco: Elsevier, Inc, 2007.

[3] G. J. Sullivan, J.-R. Ohm, T. Wiegand and W.-J. Han, "Overview of the High Efficiency Video Coding (HEVC) Standard," *Circuits and Systems for Video Technology, IEEE Transactions on,* vol. 22, no. 12, pp. 1649-1668, Dec 2012.

[4] M. A. Shami, *Dynamically Reconfigurable Resource Array,* PhD Thesis. KTH Royal Institute of Technology: Sweden, 2012.

[5] "Valgrind's webpage," [Online]. Available: http://valgrind.org/. [Accessed 14 06 2015].

[6] „Callgrind's webpage," [Võrgumaterjal]. Available: http://valgrind.org/docs/manual/cl-manual.html. [Kasutatud 14 06 2015].

[7] "High efficiency video coding (HEVC)," 19 May 2015. [Online]. Available: http://hevc.hhi.fraunhofer.de.

[8] B. Bross, W.-J. Han, J.-R. Ohm, G. J. Sullivan and T. Wiegand, *High Efficiency Video Coding (HEVC) Text Specification Draft 13,* JCTVC-K1003, ITU-T/ISO/IEC Joint Collaborative Team on Video, 2012.

[9] F. Saab, I. H. Elhajj, A. Kayssi and A. Chehab, "Profiling of HEVC encoder," *Electronics Letters,* vol. 50, no. 15, pp. 1061-1063, 2014.

[10] C.-H. Tsai, H.-T. Wang, C.-L. Liu, Y. Li and C.-Y. Lee, "A 446.6K-Gates 0.55-1.2V H.265/HEVC Decoder for Next Generation Video Applications," in *IEEE Asian Solid-State Circuits Conference (A-SSCC),* Singapore, 2013.

[11] M. A. Shami and A. Hemani, "An Improved Self-Reconfigurable Interconnection Scheme for a Coarse Grain Reconfigurable Architecture," in *NORCHIP*, Tampere, 2010.

[12] A. Hemani and M. Shami, "Control Scheme for a CGRA," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2010 22nd International Symposium on*, Petrópolis, 2010.

[13] M. Shami and A. Hemani, "Morphable DPU: Smart and efficient data path for signal processing applications," in *Signal Processing Systems, 2009. SiPS 2009. IEEE Workshop on*, Tampere, 2009.

[14] M. Shami and A. Hemani, "Address generation scheme for a coarse grain reconfigurable architecture," in *Application-Specific Systems, Architectures and Processors (ASAP), 2011 IEEE International Conference on*, Santa Monica, 2011.

[15] M. Tajammul, S. Jafri, A. Hemani, J. Plosila and H. Tenhunen, "Private configuration environments (PCE) for efficient reconfiguration, in CGRAs," in *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*, Ashburn, 2013.

[16] S. Jafri, M. Tajammul, A. Hemani, K. Paul, J. Plosila, P. Ellervee ja H. Tenuhnen, „Polymorphic Configuration Architecture for CGRAs," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on,* kd. PP, nr 99, p. 1, 2015.

[17] O. Malik, A. Hemani and S. M.A., "High Level Synthesis Framework for a Coarse Grain Reconfigurable Architecture," in *NORCHIP*, Tampere, 2010.

[18] O. Malik, A. Hemani and S. M.A., "A Library Development Framework for a Coarse Grain Reconfigurable Architecture," in *VLSI Design (VLSI Design), 2011 24th International Conference on*, Madras, 2011.

[19] O. Malik, A. Hemani and M. Shami, "A pragma based approach for mapping MATLAB applications on a coarse grained reconfigurable architecture," in *Integrated Circuits and Systems Design (SBCCI), 2012 25th Symposium on,* Brasilia, 2012.

[20] ITU-T and ISO/IEC JTC 1, *Advanced Video Coding for Generic Audiovisual Services,* ITU-T Rec. H.264 and ISO/IEC 14496-10 (AVC), 2010.

[21] J.-R. Ohm, G. J. Sullivan, H. Schwarz, T. K. Tan and T. Wiegand, "Comparison of the Coding Efficiency of Video Coding Standards - Including High Efficiency Video Coding (HEVC)," *Circuits and Systems for Video Technology, IEEE Transactions on,* vol. 22, no. 12, pp. 1669-1684, Dec 2012.

[22] "KCacheGrind's web page," [Online]. Available:

http://kcachegrind.sourceforge.net/html/Home.html. [Accessed 14 06 2015].

[23] J.-S. Park, W.-J. Nam, S.-M. Han and S. Lee, "2-D Large Inverse Transform (16x16, 32x32) for HEVC," *Journal of Semiconductor Technology and Science,* vol. 12, no. 2, 2012.

[24] *VESYLA User Manual,* KTH Royal Institute of Technology, 2015.

[25] N. Farahini, S. Li and M. A. Tajammul, "39.9 GOPs/watt multi-mode CGRA accelerator for a multi-standard basestation," in *Circuits and Systems (ISCAS), 2013 IEEE International Symposium on*, Beijing, 2013.

Appendix 1 – Transform matrices

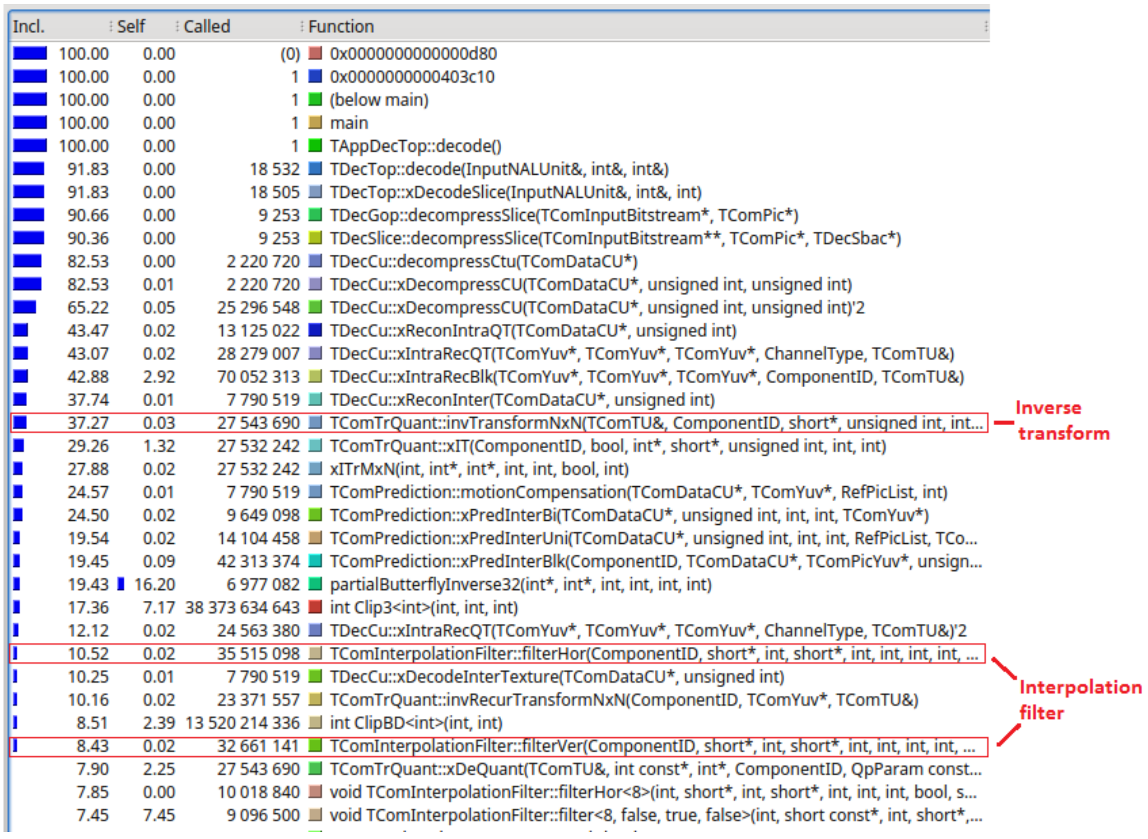
$$H=\begin{vmatrix} 29 & 55 & 74 & 84 \\ 74 & 74 & 0 & -74 \\ 84 & -29 & -74 & 55 \\ 55 & -84 & 74 & -29 \end{vmatrix}$$

Figure 30. Transform matrix used by the DST-like transform

$$H=\left|\begin{array}{rrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr}
64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 \\
90 & 90 & 88 & 85 & 82 & 78 & 73 & 67 & 61 & 54 & 46 & 38 & 31 & 22 & 13 & 4 & -4 & -13 & -22 & -31 & -38 & -46 & -54 & -61 & -67 & -73 & -78 & -82 & -85 & -88 & -90 & -90 \\
90 & 87 & 80 & 70 & 57 & 43 & 25 & 9 & -9 & -25 & -43 & -57 & -70 & -80 & -87 & -90 & -90 & -87 & -80 & -70 & -57 & -43 & -25 & -9 & 9 & 25 & 43 & 57 & 70 & 80 & 87 & 90 \\
90 & 82 & 67 & 46 & 22 & -4 & -31 & -54 & -73 & -85 & -90 & -88 & -78 & -61 & -38 & -13 & 13 & 38 & 61 & 78 & 88 & 90 & 85 & 73 & 54 & 31 & 4 & -22 & -46 & -67 & -82 & -90 \\
89 & 75 & 50 & 18 & -18 & -50 & -75 & -89 & -89 & -75 & -50 & -18 & 18 & 50 & 75 & 89 & 89 & 75 & 50 & 18 & -18 & -50 & -75 & -89 & -89 & -75 & -50 & -18 & 18 & 50 & 75 & 89 \\
88 & 67 & 31 & -13 & -54 & -82 & -90 & -78 & -46 & -4 & 38 & 73 & 90 & 85 & 61 & 22 & -22 & -61 & -85 & -90 & -73 & -38 & 4 & 46 & 78 & 90 & 82 & 54 & 13 & -31 & -67 & -88 \\
87 & 57 & 9 & -43 & -80 & -90 & -70 & -25 & 25 & 70 & 90 & 80 & 43 & -9 & -57 & -87 & -87 & -57 & -9 & 43 & 80 & 90 & 70 & 25 & -25 & -70 & -90 & -80 & -43 & 9 & 57 & 87 \\
85 & 46 & -13 & -67 & -90 & -73 & -22 & 38 & 82 & 88 & 54 & -4 & -61 & -90 & -78 & -31 & 31 & 78 & 90 & 61 & 4 & -54 & -88 & -82 & -38 & 22 & 73 & 90 & 67 & 13 & -46 & -85 \\
83 & 36 & -36 & -83 & -83 & -36 & 36 & 83 & 83 & 36 & -36 & -83 & -83 & -36 & 36 & 83 & 83 & 36 & -36 & -83 & -83 & -36 & 36 & 83 & 83 & 36 & -36 & -83 & -83 & -36 & 36 & 83 \\
82 & 22 & -54 & -90 & -61 & 13 & 78 & 85 & 31 & -46 & -90 & -67 & 4 & 73 & 88 & 38 & -38 & -88 & -73 & -4 & 67 & 90 & 46 & -31 & -85 & -78 & -13 & 61 & 90 & 54 & -22 & -82 \\
80 & 9 & -70 & -87 & -25 & 57 & 90 & 43 & -43 & -90 & -57 & 25 & 87 & 70 & -9 & -80 & -80 & -9 & 70 & 87 & 25 & -57 & -90 & -43 & 43 & 90 & 57 & -25 & -87 & -70 & 9 & 80 \\
78 & -4 & -82 & -73 & 13 & 85 & 67 & -22 & -88 & -61 & 31 & 90 & 54 & -38 & -90 & -46 & 46 & 90 & 38 & -54 & -90 & -31 & 61 & 88 & 22 & -67 & -85 & -13 & 73 & 82 & 4 & -78 \\
75 & -18 & -89 & -50 & 50 & 89 & 18 & -75 & -75 & 18 & 89 & 50 & -50 & -89 & -18 & 75 & 75 & -18 & -89 & -50 & 50 & 89 & 18 & -75 & -75 & 18 & 89 & 50 & -50 & -89 & -18 & 75 \\
73 & -31 & -90 & -22 & 78 & 67 & -38 & -90 & -13 & 82 & 61 & -46 & -88 & -4 & 85 & 54 & -54 & -85 & 4 & 88 & 46 & -61 & -82 & 13 & 90 & 38 & -67 & -78 & 22 & 90 & 31 & -73 \\
70 & -43 & -87 & 9 & 90 & 25 & -80 & -57 & 57 & 80 & -25 & -90 & -9 & 87 & 43 & -70 & -70 & 43 & 87 & -9 & -90 & -25 & 80 & 57 & -57 & -80 & 25 & 90 & 9 & -87 & -43 & 70 \\
67 & -54 & -78 & 38 & 85 & -22 & -90 & 4 & 90 & 13 & -88 & -31 & 82 & 46 & -73 & -61 & 61 & 73 & -46 & -82 & 31 & 88 & -13 & -90 & -4 & 90 & 22 & -85 & -38 & 78 & 54 & -67 \\
64 & -64 & -64 & 64 & 64 & -64 & -64 & 64 & 64 & -64 & -64 & 64 & 64 & -64 & -64 & 64 & 64 & -64 & -64 & 64 & 64 & -64 & -64 & 64 & 64 & -64 & -64 & 64 & 64 & -64 & -64 & 64 \\
61 & -73 & -46 & 82 & 31 & -88 & -13 & 90 & -4 & -90 & 22 & 85 & -38 & -78 & 54 & 67 & -67 & -54 & 78 & 38 & -85 & -22 & 90 & 4 & -90 & 13 & 88 & -31 & -82 & 46 & 73 & -61 \\
57 & -80 & -25 & 90 & -9 & -87 & 43 & 70 & -70 & -43 & 87 & 9 & -90 & 25 & 80 & -57 & -57 & 80 & 25 & -90 & 9 & 87 & -43 & -70 & 70 & 43 & -87 & -9 & 90 & -25 & -80 & 57 \\
54 & -85 & -4 & 88 & -46 & -61 & 82 & 13 & -90 & 38 & 67 & -78 & -22 & 90 & -31 & -73 & 73 & 31 & -90 & 22 & 78 & -67 & -38 & 90 & -13 & -82 & 61 & 46 & -88 & 4 & 85 & -54 \\
50 & -89 & 18 & 75 & -75 & -18 & 89 & -50 & -50 & 89 & -18 & -75 & 75 & 18 & -89 & 50 & 50 & -89 & 18 & 75 & -75 & -18 & 89 & -50 & -50 & 89 & -18 & -75 & 75 & 18 & -89 & 50 \\
46 & -90 & 38 & 54 & -90 & 31 & 61 & -88 & 22 & 67 & -85 & 13 & 73 & -82 & 4 & 78 & -78 & -4 & 82 & -73 & -13 & 85 & -67 & -22 & 88 & -61 & -31 & 90 & -54 & -38 & 90 & -46 \\
43 & -90 & 57 & 25 & -87 & 70 & 9 & -80 & 80 & -9 & -70 & 87 & -25 & -57 & 90 & -43 & -43 & 90 & -57 & -25 & 87 & -70 & -9 & 80 & -80 & 9 & 70 & -87 & 25 & 57 & -90 & 43 \\
38 & -88 & 73 & -4 & -67 & 90 & -46 & -31 & 85 & -78 & 13 & 61 & -90 & 54 & 22 & -82 & 82 & -22 & -54 & 90 & -61 & -13 & 78 & -85 & 31 & 46 & -90 & 67 & 4 & -73 & 88 & -38 \\
36 & -83 & 83 & -36 & -36 & 83 & -83 & 36 & 36 & -83 & 83 & -36 & -36 & 83 & -83 & 36 & 36 & -83 & 83 & -36 & -36 & 83 & -83 & 36 & 36 & -83 & 83 & -36 & -36 & 83 & -83 & 36 \\
31 & -78 & 90 & -61 & 4 & 54 & -88 & 82 & -38 & -22 & 73 & -90 & 67 & -13 & -46 & 85 & -85 & 46 & 13 & -67 & 90 & -73 & 22 & 38 & -82 & 88 & -54 & -4 & 61 & -90 & 78 & -31 \\
25 & -70 & 90 & -80 & 43 & 9 & -57 & 87 & -87 & 57 & -9 & -43 & 80 & -90 & 70 & -25 & -25 & 70 & -90 & 80 & -43 & -9 & 57 & -87 & 87 & -57 & 9 & 43 & -80 & 90 & -70 & 25 \\
22 & -61 & 85 & -90 & 73 & -38 & -4 & 46 & -78 & 90 & -82 & 54 & -13 & -31 & 67 & -88 & 88 & -67 & 31 & 13 & -54 & 82 & -90 & 78 & -46 & 4 & 38 & -73 & 90 & -85 & 61 & -22 \\
18 & -50 & 75 & -89 & 89 & -75 & 50 & -18 & -18 & 50 & -75 & 89 & -89 & 75 & -50 & 18 & 18 & -50 & 75 & -89 & 89 & -75 & 50 & -18 & -18 & 50 & -75 & 89 & -89 & 75 & -50 & 18 \\
13 & -38 & 61 & -78 & 88 & -90 & 85 & -73 & 54 & -31 & 4 & 22 & -46 & 67 & -82 & 90 & -90 & 82 & -67 & 46 & -22 & -4 & 31 & -54 & 73 & -85 & 90 & -88 & 78 & -61 & 38 & -13 \\
9 & -25 & 43 & -57 & 70 & -80 & 87 & -90 & 90 & -87 & 80 & -70 & 57 & -43 & 25 & -9 & -9 & 25 & -43 & 57 & -70 & 80 & -87 & 90 & -90 & 87 & -80 & 70 & -57 & 43 & -25 & 9 \\
4 & -13 & 22 & -31 & 38 & -46 & 54 & -61 & 67 & -73 & 78 & -82 & 85 & -88 & 90 & -90 & 90 & -90 & 88 & -85 & 82 & -78 & 73 & -67 & 61 & -54 & 46 & -38 & 31 & -22 & 13 & -4
\end{array}\right|$$

Figure 31. Transform matrix used by the DCT-like transform

Appendix 2 – Profiling of the HEVC Decoder



Incl.	Self	Called	Function
100.00	0.00	(0)	0x0000000000000d80
100.00	0.00	1	0x0000000000403c10
100.00	0.00	1	(below main)
100.00	0.00	1	main
100.00	0.00	1	TAppDecTop::decode()
91.83	0.00	18 532	TDecTop::decode(InputNALUnit&, int&, int&)
91.83	0.00	18 505	TDecTop::xDecodeSlice(InputNALUnit&, int&, int)
90.66	0.00	9 253	TDecGop::decompressSlice(TComInputBitstream*, TComPic*)
90.36	0.00	9 253	TDecSlice::decompressSlice(TComInputBitstream**, TComPic*, TDecSbac*)
82.53	0.00	2 220 720	TDecCu::decompressCtu(TComDataCU*)
82.53	0.01	2 220 720	TDecCu::xDecompressCU(TComDataCU*, unsigned int, unsigned int)
65.22	0.05	25 296 548	TDecCu::xDecompressCU(TComDataCU*, unsigned int, unsigned int)'2
43.47	0.02	13 125 022	TDecCu::xReconIntraQT(TComDataCU*, unsigned int)
43.07	0.02	28 279 007	TDecCu::xIntraRecQT(TComYuv*, TComYuv*, TComYuv*, ChannelType, TComTU&)
42.88	2.92	70 052 313	TDecCu::xIntraRecBlk(TComYuv*, TComYuv*, TComYuv*, ComponentID, TComTU&)
37.74	0.01	7 790 519	TDecCu::xReconInter(TComDataCU*, unsigned int)
37.27	0.03	27 543 690	TComTrQuant::invTransformNxN(TComTU&, ComponentID, short*, unsigned int, int...
29.26	1.32	27 532 242	TComTrQuant::xIT(ComponentID, bool, int*, short*, unsigned int, int, int)
27.88	0.02	27 532 242	xITrMxN(int, int*, int*, int, int, bool, int)
24.57	0.01	7 790 519	TComPrediction::motionCompensation(TComDataCU*, TComYuv*, RefPicList, int)
24.50	0.02	9 649 098	TComPrediction::xPredInterBi(TComDataCU*, unsigned int, int, int, TComYuv*)
19.54	0.02	14 104 458	TComPrediction::xPredInterUni(TComDataCU*, unsigned int, int, int, RefPicList, TCo...
19.45	0.09	42 313 374	TComPrediction::xPredInterBlk(ComponentID, TComDataCU*, TComPicYuv*, unsign...
19.43	16.20	6 977 082	partialButterflyInverse32(int*, int*, int, int, int, int)
17.36	7.17	38 373 634 643	int Clip3<int>(int, int, int)
12.12	0.02	24 563 380	TDecCu::xIntraRecQT(TComYuv*, TComYuv*, TComYuv*, ChannelType, TComTU&)'2
10.52	0.02	35 515 098	TComInterpolationFilter::filterHor(ComponentID, short*, int, short*, int, int, int, int, ...
10.25	0.01	7 790 519	TDecCu::xDecodeInterTexture(TComDataCU*, unsigned int)
10.16	0.02	23 371 557	TComTrQuant::invRecurTransformNxN(ComponentID, TComYuv*, TComTU&)
8.51	2.39	13 520 214 336	int ClipBD<int>(int, int)
8.43	0.02	32 661 141	TComInterpolationFilter::filterVer(ComponentID, short*, int, short*, int, int, int, int, ...
7.90	2.25	27 543 690	TComTrQuant::xDeQuant(TComTU&, int const*, int*, ComponentID, QpParam const...
7.85	0.00	10 018 840	void TComInterpolationFilter::filterHor<8>(int, short*, int, short*, int, int, int, bool, s...
7.45	7.45	9 096 500	void TComInterpolationFilter::filter<8, false, true, false>(int, short const*, int, short*,...

Figure 32. Profiling information of the HEVC decoder. The chosen kernels are marked with red. All the functions above the inverse transform are top-level functions.

Appendix 3 – Example Inverse Transform Test Case

```
bitdepth = 8;
width = 16;
height = 16;
useDST = 1;
maxTrDynamicRange = 15;

% Parameter interpretation:
% -------------------------
% Bit depth is 8
% Transform size is 16x16
% UseDST: It is an intra block. If we have 4x4 matrix, we should use DST% It
%is not 4x4 matrix, so we are using DCT
% Maximum transform dynamic range (not completely sure, what it means) is 15

% Transform coefficients matrix (output of de-quantization), parameter: coeff
inputMatrix = [...
-114 -114    0    0    0    0    0    0    0    0    0    0    0    0    0    0;
   0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0;
   0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0;
-114    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0;
 228    0 -114    0    0    0    0    0    0    0    0    0    0    0    0    0;
   0  114    0    0    0    0    0    0    0    0    0    0    0    0    0    0;
-114    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0;
 114    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0;
-114    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0;
0       0    0    0    0    0    0    0    0    0    0    0    0    0    0    0;
0       0    0    0    0    0    0    0    0    0    0    0    0    0    0    0;
0       0    0    0    0    0    0    0    0    0    0    0    0    0    0    0;
0       0    0    0    0    0    0    0    0    0    0    0    0    0    0    0;
0       0    0    0    0    0    0    0    0    0    0    0    0    0    0    0;
0       0    0    0    0    0    0    0    0    0    0    0    0    0    0    0;
0       0    0    0    0    0    0    0    0    0    0    0    0    0    0    0];

% Using 16x16 inverse transform, DCT coefficients used are:
coeffMatrix = [...
64  64  64  64  64  64  64  64  64  64  64  64  64  64  64  64;
90  87  80  70  57  43  25   9  -9 -25 -43 -57 -70 -80 -87 -90;
89  75  50  18 -18 -50 -75 -89 -89 -75 -50 -18  18  50  75  89;
87  57   9 -43 -80 -90 -70 -25  25  70  90  80  43  -9 -57 -87;
83  36 -36 -83 -83 -36  36  83  83  36 -36 -83 -83 -36  36  83;
80   9 -70 -87 -25  57  90  43 -43 -90 -57  25  87  70  -9 -80;
75 -18 -89 -50  50  89  18 -75 -75  18  89  50 -50 -89 -18  75;
70 -43 -87   9  90  25 -80 -57  57  80 -25 -90  -9  87  43 -70;
64 -64 -64  64  64 -64 -64  64  64 -64 -64  64  64 -64 -64  64;
57 -80 -25  90  -9 -87  43  70 -70 -43  87   9 -90  25  80 -57;
50 -89  18  75 -75 -18  89 -50 -50  89 -18 -75  75  18 -89  50;
43 -90  57  25 -87  70   9 -80  80  -9 -70  87 -25 -57  90 -43;
36 -83  83 -36 -36  83 -83  36  36 -83  83 -36 -36  83 -83  36;
25 -70  90 -80  43   9 -57  87 -87  57  -9 -43  80 -90  70 -25;
18 -50  75 -89  89 -75  50 -18 -18  50 -75  89 -89  75 -50  18;
9 -25  43 -57  70 -80  87 -90  90 -87  80 -70  57 -43  25  -9];

% Transform result (outputted residual block), parameter: block
```

```
outputMatrix = [ -2 -2 -1 -1  0  0  1  1  1  1  0 -1 -1 -2 -2 -3;
                 -2 -2 -1 -1 -1  0  0  0  1  1  1  1  1  0  0  0;
                 -3 -3 -3 -3 -3 -3 -2 -2 -2 -1  0  0  1  2  2  2;
                 -4 -4 -4 -5 -5 -5 -5 -5 -4 -3 -2 -1  0  1  2  2;
                 -3 -3 -3 -3 -4 -4 -4 -4 -4 -3 -2 -2 -1  0  1  1;
                  0  0  0 -1 -1 -1 -1 -1 -1 -1 -1 -1  0  0  0  0;
                  0  1  1  1  1  1  1  1  1  1  1  0  0  0  0 -1;
                 -1 -1  0  0  1  2  2  3  3  3  2  2  1  1  0  0;
                 -2 -1 -1  0  1  2  3  3  4  4  4  4  3  3  3  2;
                 -3 -3 -2 -2 -1  0  1  1  2  2  3  3  3  3  3  3;
                 -6 -6 -6 -6 -5 -5 -5 -5 -4 -4 -3 -2 -2 -1 -1 -1;
                 -6 -7 -7 -7 -8 -8 -9 -9 -9 -8 -8 -7 -6 -6 -5 -5;
                 -2 -2 -3 -3 -4 -5 -5 -6 -6 -6 -5 -5 -4 -4 -3 -3;
                  2  2  2  2  2  1  1  1  1  1  1  1  2  2  2  2;
                  1  1  1  1  2  2  3  3  3  4  4  4  4  4  3  3;
                 -5 -4 -4 -3 -2 -1  0  1  2  2  2  2  2  1  1  1];

% Run the MATLAB code with the parameters and compare outputs
success = transformVerify.verify(bitdepth, width, height, useDST,
maxTrDynamicRange, inputMatrix, outputMatrix, coeffMatrix);

% Check if the test passed and if not, increase the error counter.
if (success == 0)
    errors = errors + 1
end
% ==================END OF DCT==================
```

Figure 33. Example of a test case for the MATLAB inverse transorm implementation

Appendix 4 – MATLAB code for inverse partial butterflies

```
classdef Butterflies

  properties
  end

  methods (Static)
    % ============= 4x4 partial inverse butterfly =============
    function dst = partialButterflyInverse4(src, shifting, add, size)


      % 4x4 transform matrix
      DCT4x4_MATRIX = [64  64  64  64;
                       83  36 -36 -83;
                       64 -64 -64  64;
                       36 -83  83 -36];

      %One column at a time
      for i = 1:size
        src_col = src(1:4,i);

        %Implementation of the butterfly structure
        O1 = DCT4x4_MATRIX(2,1)*src_col(2) + DCT4x4_MATRIX(4,1)*src_col(4);
        O2 = DCT4x4_MATRIX(2,2)*src_col(2) + DCT4x4_MATRIX(4,2)*src_col(4);
        E1 = DCT4x4_MATRIX(1,1)*src_col(1) + DCT4x4_MATRIX(3,1)*src_col(3);
        E2 = DCT4x4_MATRIX(1,2)*src_col(1) + DCT4x4_MATRIX(3,2)*src_col(3);

        % Calculating final anwer and making the
        % required connections
        dst(i,1) = bitshift(E1 + O1 + add,shifting*-1, 'int32');
        dst(i,2) = bitshift(E2 + O2 + add,shifting*-1, 'int32');
        dst(i,3) = bitshift(E2 - O2 + add,shifting*-1, 'int32');
        dst(i,4) = bitshift(E1 - O1 + add,shifting*-1, 'int32');
      end
    end



    % ============= 8x8 partial inverse butterfly =============
    function dst = partialButterflyInverse8(src, shifting, add)

      % 8x8 transform matrix
      DCT8x8_MATRIX = [ 64  64  64  64  64  64  64  64;
                        89  75  50  18 -18 -50 -75 -89;
                        83  36 -36 -83 -83 -36  36  83;
                        75 -18 -89 -50  50  89  18 -75;
                        64 -64 -64  64  64 -64 -64  64;
                        50 -89  18  75 -75 -18  89 -50;
                        36 -83  83 -36 -36  83 -83  36;
                        18 -50  75 -89  89 -75  50 -18];
      %One column at a time
      for i = 1:8
        src_col = src(1:8,i);

        %Implementation of the butterfly structure
        for k = 1:4
          O(k) = DCT8x8_MATRIX(2,k)*src_col(2) + DCT8x8_MATRIX(4,k)*src_col(4) +...
               DCT8x8_MATRIX(6,k)*src_col(6) + DCT8x8_MATRIX(8,k)*src_col(8);
        end


        EO0 =  DCT8x8_MATRIX(3,1)*src_col(3) + DCT8x8_MATRIX(7,1)*src_col(7);
        EO1 =  DCT8x8_MATRIX(3,2)*src_col(3) + DCT8x8_MATRIX(7,2)*src_col(7);
        EE0 =  DCT8x8_MATRIX(1,1)*src_col(1) + DCT8x8_MATRIX(5,1)*src_col(5);
        EE1 =  DCT8x8_MATRIX(1,2)*src_col(1) + DCT8x8_MATRIX(5,2)*src_col(5);

        % Combining even and odd terms at each hierarchy levels to
        % calculate the final spatial domain vector
        E(1) = EE0 + EO0
        E(4) = EE0 - EO0
        E(2) = EE1 + EO1
        E(3) = EE1 - EO1

        % Calculating final anwer and making the
        % required connections
        for k = 1:4
          dst(i, k) = bitshift(E(k) + O(k) + add, shifting*-1, 'int32');
          dst(i, (k+4)) = bitshift(E(5-k) - O(5-k) + add, shifting*-1, 'int32');
        end
      end
    end




    % ============= 16x16 partial inverse butterfly =============
    function dst = partialButterflyInverse16(src, shifting, add)

      % 16x16 transform matrix
      DCT16x16_MATRIX = [...
                64  64  64  64  64  64  64  64  64  64  64  64  64  64  64  64;
                90  87  80  70  57  43  25   9  -9 -25 -43 -57 -70 -80 -87 -90;
                89  75  50  18 -18 -50 -75 -89 -89 -75 -50 -18  18  50  75  89;
                87  57   9 -43 -80 -90 -70 -25  25  70  90  80  43  -9 -57 -87;
                83  36 -36 -83 -83 -36  36  83  83  36 -36 -83 -83 -36  36  83;
                80   9 -70 -87 -25  57  90  43 -43 -90 -57  25  87  70  -9 -80;
                75 -18 -89 -50  50  89  18 -75 -75  18  89  50 -50 -89 -18  75;
                70 -43 -87   9  90  25 -80 -57  57  80 -25 -90  -9  87  43 -70;
                64 -64 -64  64  64 -64 -64  64  64 -64 -64  64  64 -64 -64  64;
                57 -80 -25  90  -9 -87  43  70 -70 -43  87   9 -90  25  80 -57;
                50 -89  18  75 -75 -18  89 -50 -50  89 -18 -75  75  18 -89  50;
                43 -90  57  25 -87  70   9 -80  80  -9 -70  87 -25 -57  90 -43;
                36 -83  83 -36 -36  83 -83  36  36 -83  83 -36 -36  83 -83  36;
                25 -70  90 -80  43   9 -57  87 -87  57  -9 -43  80 -90  70 -25;
                18 -50  75 -89  89 -75  50 -18 -18  50 -75  89 -89  75 -50  18;
                 9 -25  43 -57  70 -80  87 -90  90 -87  80 -70  57 -43  25  -9];
      %One column at a time
      for i = 1:16
        src_col = src(1:16,i);

        %Implementation of the butterfly structure
        for k = 1:8
           O(k) = DCT16x16_MATRIX( 2,k)*src_col( 2) + ...
                  DCT16x16_MATRIX( 4,k)*src_col( 4) + ...
                  DCT16x16_MATRIX( 6,k)*src_col( 6) + ...
                  DCT16x16_MATRIX( 8,k)*src_col( 8) + ...
                  DCT16x16_MATRIX(10,k)*src_col(10) + ...
                  DCT16x16_MATRIX(12,k)*src_col(12) + ...
                  DCT16x16_MATRIX(14,k)*src_col(14) + ...
                  DCT16x16_MATRIX(16,k)*src_col(16);
        end

        for k = 1:4
          EO(k) = DCT16x16_MATRIX( 3,k)*src_col( 3) + ...
                  DCT16x16_MATRIX( 7,k)*src_col( 7) + ...
                  DCT16x16_MATRIX(11,k)*src_col(11) + ...
                  DCT16x16_MATRIX(15,k)*src_col(15);
        end

        EEO(1) = DCT16x16_MATRIX(5,1)*src_col( 5) + ...
                 DCT16x16_MATRIX(13,1)*src_col(13);
        EEE(1) = DCT16x16_MATRIX(1,1)*src_col( 1) + ...
                 DCT16x16_MATRIX( 9,1)*src_col( 9);
        EEO(2) = DCT16x16_MATRIX(5,2)*src_col( 5) + ...
                 DCT16x16_MATRIX(13,2)*src_col(13);
        EEE(2) = DCT16x16_MATRIX(1,2)*src_col( 1) + ...
                 DCT16x16_MATRIX( 9,2)*src_col( 9);

        %Combining even and odd terms at each hierarchy levels to
        % calculate the final spatial domain vector
        for k = 1:2
          EE(k) = EEE(k) + EEO(k);
          EE(k+2) = EEE(3-k) - EEO(3-k);
        end

        for k = 1:4
          E(k) = EE(k) + EO(k);
          E(k+4) = EE(5-k) - EO(5-k);
        end

        % Calculating final anwer and making the
        % required connections
        for k = 1:8
          dst(i, k) = bitshift(E(k) + O(k) + add, shifting*-1, 'int32');
          dst(i, (k+8)) = bitshift(E(9-k) - O(9-k) + add, shifting*-1, 'int32');
        end
      end
    end
```

```
% ============= 32x32 partial inverse butterfly =============
function dst = partialButterflyInverse32(src, shifting, add)

  % 32x32 transform matrix
  DCT32x32_MATRIX = []; %Matrix removed to save space in the thesis.
                        %In case of interest please refer to appendix 1.

  %Implementation of the butterfly structure
  for i = 1:32
    src_col = src(1:32,i);

    % Combining even and odd terms at each hierarchy levels to
    % calculate the final spatial domain vector
    for k = 1:16
      O(k) = DCT32x32_MATRIX( 2,k)*src_col( 2) + ...
             DCT32x32_MATRIX( 4,k)*src_col( 4) + ...
             DCT32x32_MATRIX( 6,k)*src_col( 6) + ...
             DCT32x32_MATRIX( 8,k)*src_col( 8) + ...
             DCT32x32_MATRIX(10,k)*src_col(10) + ...
             DCT32x32_MATRIX(12,k)*src_col(12) + ...
             DCT32x32_MATRIX(14,k)*src_col(14) + ...
             DCT32x32_MATRIX(16,k)*src_col(16) + ...
             DCT32x32_MATRIX(18,k)*src_col(18) + ...
             DCT32x32_MATRIX(20,k)*src_col(20) + ...
             DCT32x32_MATRIX(22,k)*src_col(22) + ...
             DCT32x32_MATRIX(24,k)*src_col(24) + ...
             DCT32x32_MATRIX(26,k)*src_col(26) + ...
             DCT32x32_MATRIX(28,k)*src_col(28) + ...
             DCT32x32_MATRIX(30,k)*src_col(30) + ...
             DCT32x32_MATRIX(32,k)*src_col(32);
    end

    for k = 1:8
      EO(k) = DCT32x32_MATRIX( 3,k)*src_col( 3) + ...
              DCT32x32_MATRIX( 7,k)*src_col( 7) + ...
              DCT32x32_MATRIX(11,k)*src_col(11) + ...
              DCT32x32_MATRIX(15,k)*src_col(15) + ...
              DCT32x32_MATRIX(19,k)*src_col(19) + ...
              DCT32x32_MATRIX(23,k)*src_col(23) + ...
              DCT32x32_MATRIX(27,k)*src_col(27) + ...
              DCT32x32_MATRIX(31,k)*src_col(31);
    end

    for k = 1:4
      EEO(k) = DCT32x32_MATRIX( 5,k)*src_col( 5) + ...
               DCT32x32_MATRIX(13,k)*src_col(13) + ...
               DCT32x32_MATRIX(21,k)*src_col(21) + ...
               DCT32x32_MATRIX(29,k)*src_col(29);
    end

    EEEO(1) = DCT32x32_MATRIX(9,1)*src_col(9) + ...
              DCT32x32_MATRIX(25,1)*src_col(25);
    EEEO(2) = DCT32x32_MATRIX(9,2)*src_col(9) + ...
              DCT32x32_MATRIX(25,2)*src_col(25);
    EEEE(1) = DCT32x32_MATRIX(1,1)*src_col(1) + ...
              DCT32x32_MATRIX(17,1)*src_col(17);
    EEEE(2) = DCT32x32_MATRIX(1,2)*src_col(1) + ...
              DCT32x32_MATRIX(17,2)*src_col(17);
```

```
        % Combining even and odd terms at each hierarchy levels to
        % calculate the final spatial domain vector
        EEE(1) = EEEE(1) + EEEO(1);
        EEE(4) = EEEE(1)  - EEEO(1);
        EEE(2) = EEEE(2) + EEEO(2);
        EEE(3) = EEEE(2)  - EEEO(2);

        for k = 1:4
          EE(k) = EEE(k) + EEO(k);
          EE(k+4) = EEE(5-k) - EEO(5-k);
        end

        for k = 1:8
          E(k) = EE(k) + EO(k);
          E(k+8) = EE(9-k) - EO(9-k);
        end

        % Calculating final anwer and making the
        % required connections
        for k = 1:16
          dst(i, k) = bitshift(E(k) + O(k) + add, shifting*-1, 'int32');
          dst(i, (k+16)) = bitshift(E(17-k) - O(17-k) + add, shifting*-1, 'int32');
        end
      end
    end
  end
end
```

Figure 34. MATLAB code for inverse partial butterflies

Appendix 5 - Top-level MATLAB inverse transform code

```
classdef IntDct2

properties
end

methods (Static)

% ============= 4x4 partial inverse butterfly =============
function dst = partialButterflyInverse4(src, shifting, clipMinMax)
  if (shifting > 0)
    add = bitshift(1, shifting-1, 'int32')
  else
    add = 0;
  end
        %Calculate IDCT using the butterfly code
  dct = Butterflies.partialButterflyInverse4(src, shifting, add, 4);
        %Apply clipping to the calculated IDCT
  dst = clip(dct, clipMinMax);
end

 % ============= 8x8 partial inverse butterfly =============
function dst = partialButterflyInverse8(src, shifting, clipMinMax)

  if (shifting > 0)
    add = bitshift(1, shifting-1, 'int32');
  else
    add = 0;
  end
      %Calculate IDCT using the butterfly code
  dct = Butterflies.partialButterflyInverse8(src, shifting, add);
   %Apply clipping to the calculated IDCT
  dst = clip(dct, clipMinMax);
end

% ============= 16x16 partial inverse butterfly =============
function dst = partialButterflyInverse16(src, shifting, clipMinMax)
  if (shifting > 0)
    add = bitshift(1, shifting-1, 'int32');
  else
    add = 0;
  end
  %Calculate IDCT using the butterfly code
  dct = Butterflies.partialButterflyInverse16(src, shifting, add);
  %Apply clipping to the calculated IDCT
  dst = clip(dct, clipMinMax);
end

 % ============= 32x32 partial inverse butterfly =============
function dst = partialButterflyInverse32(src, shifting, clipMinMax)
  if (shifting > 0)
    add = bitshift(1, shifting-1, 'int32');
  else
    add = 0;
  end
        %Calculate IDCT using the butterfly code
  dct = Butterflies.partialButterflyInverse32(src, shifting, add);
  %Apply clipping to the calculated IDCT
  dst = clip(dct, clipMinMax);
end
```

```
%4x4 fast inverse DST matrix calculation
function dst = fastInverseDst(src, shifting, clipMinMax)
  if (shifting > 0)
    rnd_factor = bitshift(1, shifting-1, 'int32');
  else
    rnd_factor = 0;
  end

   % 4x4 transform coefficients matrix
  DST4x4_MATRIX = [29  55  74  84;
           74  74   0 -74;
           84 -29 -74  55;
           55 -84  74 -29];

        % One column at a time
  for i = 1:4
    src_col = src(1:4,i);

    for column = 1:4
      intermediateResult = 0;
      for row = 1:4
        intermediateResult = intermediateResult + ...
                             src_col(row) * DST4x4_MATRIX(row,column);
      end

      if (shifting >= 0)
        intermediateResult = bitshift((intermediateResult + ...
                             rnd_factor), shifting*-1, 'int32');
      else
        intermediateResult = bitshift((intermediateResult + ...
                             rnd_factor) , shifting, 'int32');
      end
      dst(i, column) = clip(intermediateResult, clipMinMax);
    end

  end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%TOPLEVEL INVERSE TRANSFORM FUNCTION%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function block = xITrMxN(bitDepth, coeff, iWidth, iHeight, useDST, maxTrDynamicRange)
  %calculate transform parameters
        %Shifting for transform matrix. In case of 8bit video always constant.
  transformMatrixShift = 6;
        %shifting value for first 1D transform.
  shift_1st = transformMatrixShift + 1
        %shifting value for second 2D transform
  shift_2nd = (transformMatrixShift + maxTrDynamicRange -1) - bitDepth;

        %minimum clipping value
  clipMinimum =  -1*(bitshift(1, maxTrDynamicRange, 'int32'));
        %maximum clipping value
  clipMaximum = bitshift(1, maxTrDynamicRange, 'int32') - 1;
        %clipping parameters for the first 1D transform
  clipMinMax_1st = [clipMinimum clipMaximum];

        %Maximum size for the data object
  shrt_max = 32767;
        %clipping values for the second 1D transform
  clipMinMax_2nd = [(-1*shrt_max)-1 shrt_max];
  %Transforms:
  %HEVC transform vs "normal" DCT transform: in HEVC shifting and clipping
  %is done after every 1D transform. In case of 4x4 intra frame DST is used.
  %In this implementation both 1D transforms are identical and automatically
  %transpose theoutput function for the next step.
  %The only difference between 1st and 2nd 1D transform are the shifting
  %and clipping parameters.

  %1st 1D transform:
  switch iHeight
    case 4
      if ((iWidth == 4) && (useDST == 1))
        tmp = IntDct2.fastInverseDst(coeff, shift_1st, clipMinMax_1st);
      else
        tmp = IntDct2.partialButterflyInverse4 ( coeff, shift_1st, clipMinMax_1st)
      end
    case 8
      tmp = IntDct2.partialButterflyInverse8 (coeff, shift_1st, clipMinMax_1st);
    case 16
      tmp = IntDct2.partialButterflyInverse16 (coeff, shift_1st, clipMinMax_1st);
    case 32
      tmp = IntDct2.partialButterflyInverse32 (coeff, shift_1st, clipMinMax_1st);
    otherwise
      disp('Matrix with and illegal size!')
  end

  %2nd 1D transform
  switch iWidth
    case 4
      if ((iHeight == 4) && (useDST == 1))
        block = IntDct2.fastInverseDst(tmp, shift_2nd, clipMinMax_2nd);
      else
        block = IntDct2.partialButterflyInverse4 ( tmp, shift_2nd, clipMinMax_2nd);
      end
    case 8
      block = IntDct2.partialButterflyInverse8 (tmp, shift_2nd, clipMinMax_2nd);
    case 16
      block = IntDct2.partialButterflyInverse16 (tmp, shift_2nd, clipMinMax_2nd);
    case 32
      block = IntDct2.partialButterflyInverse32 (tmp, shift_2nd, clipMinMax_2nd);
    otherwise
      disp('Matrix with and illegal size!')
  end

end
end

end

% Clipping function
function out = clip(num, minmax)
  if (num < minmax(1))
    out = minamax(1);
  elseif (num > minmax(2))
    out = minmax(2);
  else
    out = num;
  end
end
```

Figure 35. Top-level MATLAB inverse transform code

Appendix 6 – MATLAB Inverse Transform Verification Code

```
classdef transformVerify
  properties
  end
  methods (Static)
    %Inverse transform testing and verifying script
    function success = verify(bitDepth, width, height, useDST,
             maxTrDynamicRange, inputMatrix, outputMatrix, coeffMatrix)

      %Run an the transform
      output = IntDct2.xITrMxN(bitDepth, inputMatrix, width,
             height, useDST, maxTrDynamicRange)

      %Compare the output with
      success = isequal(outputMatrix, output);
      %If an mismatch is detected, print out the data for debugging
      if (success == 0)
        disp('Error detected, parameters are the following:')
        bitDepth
        width
        height
        useDST
        maxTrDynamicRange
        inputMatrix
        outputMatrix
        coeffMatrix
        output
        disp('====End of parameter dump====')
      end
    end
  end
end
```

Figure 36. Code for verifying the MATLAB implementation of the inverse transform

Appendix 7 – VESYLA-Readable 4x4 Inverse DCT Transform Using Single Register File at a time

```
src = [1:16];
dst = [1:16];

mulreg = [1:8];
addreg = [1:4];
% Real 4x4 transform coefficients matrix, for testing
%DCT4x4_MATRIX = ...
%[64  64  64  64 83  36 -36 -83 64 -64 -64  64 36 -83  83 -36];

% Placeholder for 4x4 transform coefficients matrix,
% for compiling with VESYLA
DCT4x4_MATRIX = [1:16];

for i = 1:4
    mulreg(i) = DCT4x4_MATRIX(5)*src(4+i);
    mulreg(i+1) = DCT4x4_MATRIX(6)*src(4+i);
end
for i = 1:4
    mulreg(i+2) = DCT4x4_MATRIX(1)*src(i);
    mulreg(i+3) = DCT4x4_MATRIX(2)*src(i);
end
for i = 1:4
    mulreg(i+4) = DCT4x4_MATRIX(13)*src(12+i);
    mulreg(i+5) = DCT4x4_MATRIX(14)*src(12+i);
end
for i = 1:4
    mulreg(i+6) = DCT4x4_MATRIX(9)*src(8+i);
    mulreg(i+7) = DCT4x4_MATRIX(10)*src(8+i);
end

for i = 1:4
    addreg(i) = mulreg(i) + mulreg(i+4);
    addreg(i+1) = mulreg(i+1) + mulreg(i+5);
end
for i = 1:4
    addreg(i+2) = mulreg(i+2) + mulreg(i+6);
    addreg(i+3) = mulreg(i+3) + mulreg(i+7);
end
    %Combining even and odd terms at each hierarchy levels to
    % calculate the final spatial domain vector\
for i = 1:4
    dst(((i-1)*3)+i) = addreg(i+2) + addreg(i);
    dst(((i-1)*3)+i+1) = addreg(i+3) + addreg(i+1);
end
for i = 1:4
    dst(((i-1)*3)+i+2) = addreg(i+3) + addreg(i+1) * (-1);
    dst(((i-1)*3)+i+3) = addreg(i+2) + addreg(i) * (-1);
end
```

Figure 37. VESYLA-Readable 4x4 Inverse DCT Transform Using Single Register File at a time

Appendix 8 – Parallelized VESYLA-Readable 4x4 Inverse DCT Transform

```
%Register files for source data
src1 = [1:16]; %! RFILE<>[0:0]
src2 = [1:16]; %! RFILE<>[0:1]
src3 = [1:16]; %! RFILE<>[1:0]
src4 = [1:16]; %! RFILE<>[1:1]

dst = [1:16]; %! RFILE<>[0:2]

%Register files for storing multiplication results
mulreg1 = [1:16]; %! RFILE<>[1:2]
mulreg2 = [1:16]; %! RFILE<>[0:3]
mulreg3 = [1:16]; %! RFILE<>[1:3]
mulreg4 = [1:16]; %! RFILE<>[0:4]

addreg1 = [1:16]; %! RFILE<>[1:4]
addreg2 = [1:16]; %! RFILE<>[0:5]


% Real 4x4 transform coefficients matrix, for testing
% DCT4x4_MATRIX1 = ...
%    [64  64  64  64 83  36 -36 -83 64 -64 -64  64 36 -83  83 -36];
% DCT4x4_MATRIX2 = ...
%    [64  64  64  64 83  36 -36 -83 64 -64 -64  64 36 -83  83 -36];
% DCT4x4_MATRIX3 = ...
%    [64  64  64  64 83  36 -36 -83 64 -64 -64  64 36 -83  83 -36];
% DCT4x4_MATRIX4 = ...
%    [64  64  64  64 83  36 -36 -83 64 -64 -64  64 36 -83  83 -36];

% Placeholder for 4x4 transform coefficients matrix,
% for compiling with VESYLA
DCT4x4_MATRIX1 = [1:16]; %! RFILE<>[1:5]
DCT4x4_MATRIX2 = [1:16]; %! RFILE<>[0:6]
DCT4x4_MATRIX3 = [1:16]; %! RFILE<>[1:6]
DCT4x4_MATRIX4 = [1:16]; %! RFILE<>[0:7]


for i = 1:2:4
    mulreg1(i)   = DCT4x4_MATRIX1(5)*src1(4+i); %! DPU [0:0]
    mulreg1(i+1) = DCT4x4_MATRIX1(6)*src1(4+i); %! DPU [0:1]
    mulreg2(i)   = DCT4x4_MATRIX2(1)*src2(i);   %! DPU [1:0]
    mulreg2(i+1) = DCT4x4_MATRIX3(2)*src2(i);   %! DPU [1:1]

    mulreg3(i)   = DCT4x4_MATRIX3(13)*src3(12+i); %! DPU [0:2]
    mulreg3(i+1) = DCT4x4_MATRIX3(14)*src3(12+i); %! DPU [2:0]
    mulreg4(i)   = DCT4x4_MATRIX4(9)*src4(8+i);   %! DPU [2:2]
    mulreg4(i+1) = DCT4x4_MATRIX4(10)*src4(8+i);  %! DPU [2:3]
end

for i = 1:2:4
    addreg1(i)   = mulreg1(i) + mulreg3(i);     %! DPU [0:0]
    addreg1(i+1) = mulreg1(i+1) + mulreg3(i+1); %! DPU [1:0]
    addreg2(i)   = mulreg2(i) + mulreg4(i);     %! DPU [0:1]
    addreg2(i+1) = mulreg2(i+1) + mulreg4(i+1); %! DPU [1:1]
end

j=1; %We are using two different index variables for the for cycle

for i = 1:4
    %Combining even and odd terms at each hierarchy levels to
    % calculate the final spatial domain vector
    dst(((j-1)*3)+j)   = addreg2(i) + addreg1(i);     %! DPU [0:0]
    dst(((j-1)*3)+j+1) = addreg2(i+1) + addreg1(i+1); %! DPU [0:1]
end
for i = 1:4
    dst(((j-1)*3)+j+2) = addreg2(i+1) - addreg1(i+1); %! DPU [0:0]
    dst(((j-1)*3)+j+3) = addreg2(i) - addreg1(i);     %! DPU [0:1]
    j=j+1;
end
```

Figure 38. VESYLA-readable 4x4 Inverse DCT transform code

Appendix 9 - HEVC video decoder's layout

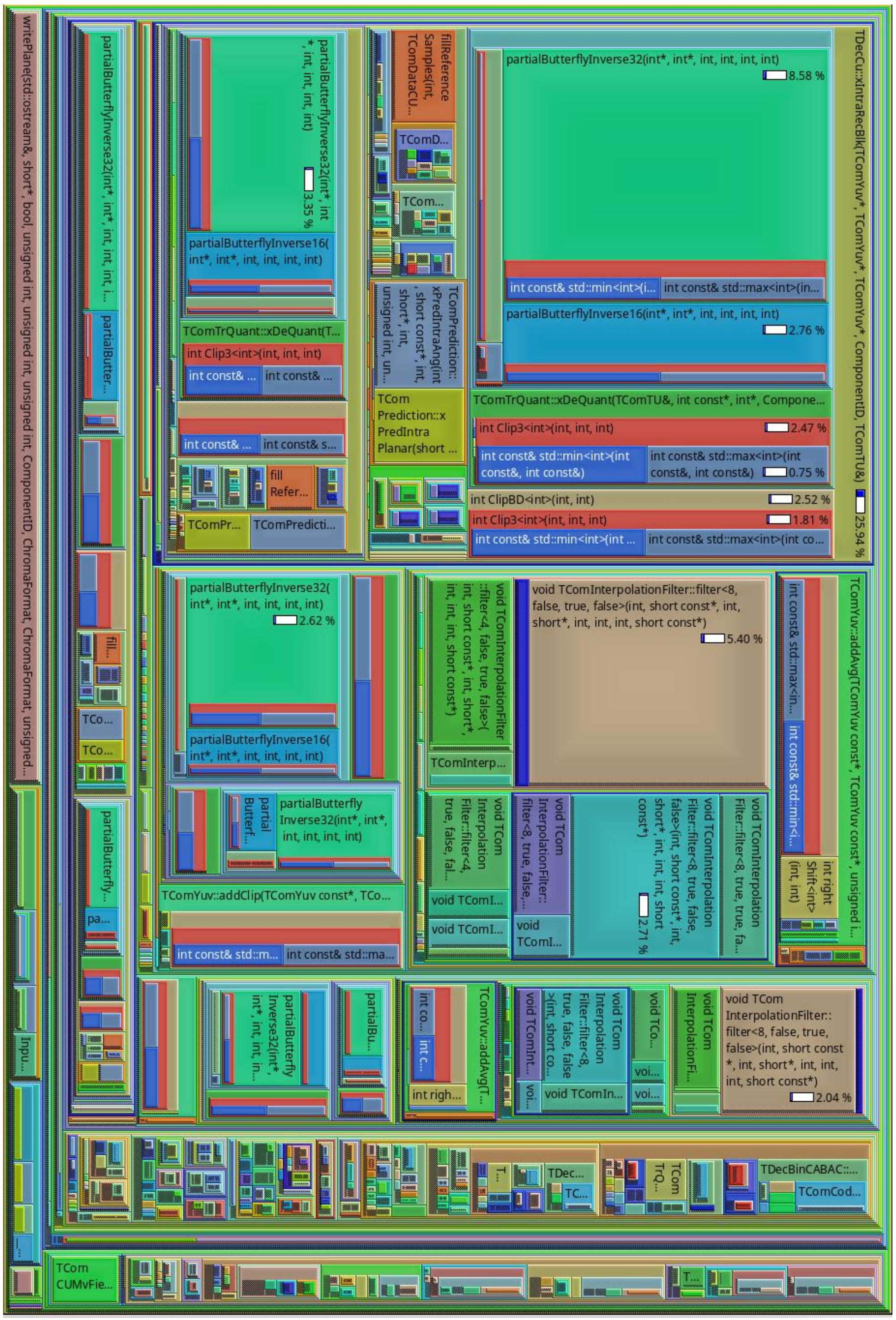


Figure 39. Layout of the HEVC video decoder. Functions with name partialButterflyInverse are part of the inverse transform. Functions belonging to class TComInterpolationFilter, are part of the interpolation filter