Optimizing an H.264 video encoder for real-time HD-video encoding

PER HERMANSSON

KTH Information and Communication Technology

Master of Science Thesis
Stockholm, Sweden 2011

TRITA-ICT-EX-2011:74

ROYAL INSTITUTE
OF TECHNOLOGY

Optimizing an H.264 video encoder for real-time HD-video encoding

PER HERMANSSON

Master's Thesis at Ericsson Research, Sweden
Supervisor at Ericsson Research was Jonatan Samuelsson
Supervisor at KTH was Mats Brorsson
Examiner was Mats Brorsson

Abstract

With the increased demands for higher resolution and higher quality video the requirements for larger storage medium and higher bandwidth has increased as well. One method to cope with these new demands is by introducing new ways to efficiently compress video. One problem with this approach is that better compression means higher computational complexity.

This Master's thesis presents three methods that are used to independently optimize an existing video encoder (using the H.264 codec). Where previous research has mostly focused on standard and lower resolution video, this thesis focuses on encoding HD-video (High-Definition). The implemented optimizations work differently, as an example some approaches makes better use of the computer hardware.

The result of this work is that real-time encoding of HD-video, on a workstation computer, at minimum of 50 frames per second is achieved when combining instruction- and thread-level parallelism. Since the minimum requirement of real-time encoding is 25-frames it was also investigated how the extra complexity can be used to get better compression results.

Referat

Optimering av en H.264 videokodare för realtidskonvertering av HD-video

Dagens ökande efterfrågan på video som både är mer högupplöst men även har bättre visuell kvalitet, ställer nya krav på större lagringsmöjligheter och snabbare överföringskapacitet. Ett sätt att hantera detta krav är att förbättra videokomprimeringen. Problemet med videokomprimering är dock att den är tidskrävande och för att öka komprimeringsgraden krävs högre beräkningskomplexitet.

Detta examensarbete visar på tre olika metoder, som kan användas oberoende av varandra, för att optimera en befintlig videokomprimerare (som använder sig av H.264 standarden). Då tidigare forskning till mesta del arbetat med lågupplöst video ligger vikten i denna rapport på högupplöst video (så kallad HD-video). De implementerade optimeringarna fokuserar på olika aspekter, vissa använder hårdvaran effektivare, medans andra fokuserar på balansen mellan komprimeringsgrad och komplexitet.

Resultatet av arbetet är att målet med realtidskonverting uppnåddes, på en vanlig arbetsdator, vid 50 bilder per sekund när lågnivåoptimering kombinerades med trådparallellisering. Då resultatet ligger över den nedre gränsen för realtidskonvertering på 25 bilder per sekund, presenteras också hur extra komplexitet bäst kan användas för att förbättra komprimeringsgraden.

Contents

Chapter 1

Introduction

1.1 Scenario

Digital image and video compression is about taking raw uncompressed video (as captured by the camera's digital sensor) and take advantage of the redundant information both within a single still image (called a frame) and between previous and following frames. Unlike other types of compression methods, (lossless) video compression algorithms do not result in the same compression ratio as say compressing a text document.

As a consequence of this, digital video is often compressed using lossy compression algorithms. In lossy video compression the decompressed (or restored) video will most often not be identical to the original (raw) footage and will most often have a slight drop in perceived quality. Because of the fact that (lossy) compressed video takes so much less storage than uncompressed video and with only a slight reduction in visual quality, almost all of today's digital video is transmitted in compressed form.

In some situations it's a requirement that the video compression works as fast as the video is created, examples where this occurs exists in live-video broadcasting, video telephony and conferencing. In these situations it's important for the sender to be able to quickly compress and transmit the footage while it's being recorded. Otherwise large memory or temporary storage is needed to temporary hold the raw video. In the case of video telephony and conferencing an additional requirement is to have low latency encoding. Here latency is defined as the time from when the video is first captured until the compressed version reaches the receiver and the decompressed footage is displayed. Having a large latency in video telephony makes two-way communication both unnatural and cumbersome.

1.2 Objective

The goal of this thesis is to attempt to optimize an existing H.264/AVC video encoder in order to achieve real-time compression of video footage. Since compression times are highly dependent on video resolution only high-resolution (HD) video sequences are considered in this study.

The actual definition of real-time encoding is not exactly defined and varies depending on application uses and transport medium. Traditional standard definition content like DVD discs plays between 25 and 30 frames per second. When it comes to 720p HD-video which is the type of video used in this study, the playback frame-rate varies. One example of HD-video usage on the Internet is Youtube which has a frame rate limit of 30 when uploading HD video. A similar scenario is broadcasting of live TV (HDTV) where 720p resolution is normally played back at 50 frames per second.

This thesis investigates how the encoding complexity (time to encode a video), compression efficiency (difference in size before and after compression) and quality are affected when optimizing the video encoder. Even though no specific scenario is targeted in this study, encoding latency is also considered an important factor when comparing different implementations.

1.3 Approach and restrictions

In order to reach the real-time encoding goal a couple of different implementations strategies are evaluated. The different types target different aspects of encoding and can be used independently or together in order to combine the individual speed-ups of each method. The approaches investigated are optimizing cache and memory utilization, using low-level vector instructions, algorithm changes and parallelization of the encoder. In order to limit the scope of the thesis, the following list of aspects are not considered in this report.

Constrained Baseline only The H.264 specification has defined several profiles (baseline, main and high) that allows different level of compression at the cost of greater complexity for both the encoder and decoder. For this thesis work is done on an encoder supporting the baseline profile only. By using the other profile levels more complexity is required during encoding which further increasing the real-time encoding challenge.

Hard deadline encoding Certain types of applications requires encoding to be done at a fixed rate. For these types of applications frames have a fixed number of milliseconds to be encoded at. These types of encodings are more suitable for embedded systems which allows for a more stricter control on its environment, such as operating system scheduling.

Extreme parallelism scaling Since this thesis focuses on real-time encoding for ”common” workstation hardware no work is done on scaling the encoder to large amounts of CPU cores. In the report the effects of parallelism is investigated on a computer with 8 (logical) CPU cores.

Full high-definition video sequences Full-HD video (1080p), which has more than twice as many pixels as 720p, also increases the encoding complexity further. Based on targeted hardware and current encoder performance, it was considered less feasible that this resolution could be encoded in real-time and it therefore not investigated in this work.

Hardware or graphics chipsets With the goal of performing real-time encoding on workstation computers exotic hardware chipsets are not considered in this thesis. Even though general-purpose GPUs are becoming more common in todays computers, it was considered a thesis on its own to adapt the encoder for this type of hardware.

No high latency (a.k.a pipelining) approaches One scenario where real-time encoding is used is in video telephony and conferencing. A common aspect of these types of use-cases is that they also puts constraints on encoding latency. In order to not limit the types of applications that can benefit from this work, methods that result in higher encoding latency are not investigated.

Only SSE2 vectorizations Again due to the focus on supporting workstation hardware and in order to not have to develop multiple versions, only the SSE2 instruction set is used for the vector optimizations. One benefit of using SSE2 is that it’s guaranteed to be supported on all x86-64 CPUs.

1.4 Organization of the report

In Chapter 2 a general background to video compression is given. This is followed by an explanation of the H.264/AVC codec that is video encoding standard used in this thesis. Finally a brief study of previous research that has been done on making video encoding faster is presented in section 2.4.

Chapter 3 starts with a presentation of the encoder targeted in this thesis. Its behavior is analyzed and compared with two other H.264 encoders. This is followed by a technical description of the proposed code and algorithm changes that has been implemented.

The results of the optimizations are presented in Chapter 4. This chapter analyzes the benefits and drawbacks of the various optimizations and compares them with each other and to a different encoder implementation.

Finally conclusions and list of accomplishments are shown in Chapter 4. The chapter is then ended with some topics that could be investigated in order to increase performance even more.

Chapter 2

Background

This chapter attempts to explain the underlying motivation for video-compression and how it can be implemented. The list of video encoding techniques does not explain the whole inner workings of how an encoder operates. Instead it tries to explain the different areas that were targeted as part of this thesis.

2.1 General video compression

Image and video compression is an area with much ongoing research. New demands for higher quality and higher resolution video has increased the needs for better compression. One reason is that bandwidth capacity has not scaled with the new demands for HD-video.

In order to better understand how the data rate increases with different resolution compare SD-resolution video (720 x 486@24p, which is played on a traditional DVD disc) that requires 210 megabits per second (Mbps) to represent in uncompressed form. Compare this number with HD-video (1280 x 720@24p) that requires 332 Mbps. When resolution increases to Full HD (1920 x 1080/60i) uncompressed video requires 932 Mbps which is equivalent to 410 GB per hour of video[33]. At least today the average user has nowhere near this kind of storage space for watching a full length movie nor enough bandwidth to stream Full-HD video across Internet in uncompressed form. A conclusion is therefore that, with the continuing trend towards more higher resolution and higher quality video, compression is still needed, perhaps more than ever.

The act of compressing video is called encoding and the inverse action of uncompressing is called decoding. As computers have become faster over the years so has video encoding methods become more complex in order to get even better compression results. H.264/AVC is today (2010) one of the latest methods in which very good quality per bit-rate is achieved by requiring high computational complexity in the encoding process. In many scenarios this trade-off is often acceptable since encoding is only done once whereas decoding (i.e. playback) can be performed numerous times.

2.1.1 What is video

As input data the encoder takes a video that consists of a series of digital pictures (known as frames). When displayed, each frame consists of pixels containing the color information (RGB color space). What many video standards do is to use the *YCrCb* (luminance, red chrominance, blue chrominance) color space instead. This decision has to do with how the human visual system is less sensitive to color than to luminance (brightness) changes [23]. With the *YCrCb* color space different so called samplings are used to control how much information is stored in each frame. A common format (or pattern) called 4:2:0 states that the red and blue color information is sampled (recorded) only at each forth (i.e. one every 2x2 block) pixel. This severally reduces the amount of information that needs to be analyzed and encoded. Since the luminance data contains twice as much information compared to the color data, most video coding standards use this information for analyzing how to efficiently compress the frame.

Apart from dividing a video sequence into frames, a frame can further be divided into smaller parts. One of those are called *slices* which can contain arbitrary regions of the frame. A feature of using slices is that they are self containing and independent on other slices in the frame. This property makes slices useful for dealing with errors (e.g. due to problems with the video transmission) since an error in one slice is not propagated to the others. A downside of using slices is that since they are independent, the compression efficiency is slightly degraded. This is because of two reasons; first redundant information across slice borders cannot be exploited and secondly because of the extra information required with coding the slice header.

Furthermore a slice is divided into *macroblocks*, which are the basic units that the encoder works with. Dividing the frame into macroblocks makes tasks such as motion estimation and block transformations easier compared to doing these for each individual pixel. The size of a macroblock varies depending on which video codec is used. This thesis focuses solely on the H.264 codec, which has defined the macroblock size to be 16x16 pixels large. With the 4:2:0 color space this means that a macroblock contains 256 luminance, 64 red chrominance and 64 blue chrominance pixels.

2.1.2 Video quality measurement

The quality of the encoded output (i.e. the compressed video) can be measured both subjective and objectively. Subjective quality is about how the Human Visual System (HSV) perceive the decoded output. The objective quality contains all the measurements which can be calculated automatically using an algorithm. Subjective quality is affected by many different factors, e.g. different people have different opinions on quality[23], making a subjective comparison both time-consuming and difficult to evaluate. In this work objective quality algorithms are primarily used to compare different implementations. Using only subjective measurements to validate the results. One of the more popular objective quality algorithms are called the Peak Signal-to-Noise Ratio (PSNR) metric [23]. This is a logarithmic scale that is calculated from the *mean square error* between the original and the decoded image, where a higher PSNR value means better quality.

There are also other types of objective quality metrics. According to Richardson[23] some recent proposals are Structural SIMilarity index (SSIM) and Just Noticeable Difference (JND). When comparing these metrics with subjective test scores there are reported correlations of between 70% to 90% between the objective metric and subjective quality score. The reason why PSNR is used in this work is because of its wide usage and to also be able to compare the results with other research.

Even though PSNR is widely used it has some drawbacks. One example is that it does not relate well to subjective quality, for example a blurred images will get a higher PSNR value even though it's commonly perceived as of lesser quality. Another issue is that since video quality is highly dependent on the bit-rate, two bit-streams can only be compared if they have the same bit-rate. This makes it difficult to compare changes that affect the compression efficiency.

This last problem do have an acceptable solution. To compensate for varying bit-rate both algorithms (or the methods to compare) are executed four times on the same video sequence but with different quality settings. The four PSNR values are then plotted on a graph together with their resulting bit-rate. The output are two values called BDPSNR (Bjøntegaard Delta PSNR) and BDBR (Bjøntegaard Delta Bit Rate) [7] that reflects the overall quality and bit-rate changes between the two algorithms.

2.2 H.264/AVC codec

As stated in the title for this thesis, the video encoding standard used is called H.264/AVC (Advanced Video Coding)[17]. This standard was first approved in 2003 and has been enhanced over the years to include new features. Compared to previous standards H.264 offers very good quality per bit-rate at the cost of requiring large computational complexity in the encoding process.

The standard is probably most known from being used in Blu-ray disc (successor to the DVD disc) and for streaming HD-video over Internet.

Like many other video coding standards the H.264 specification does not define how an encoder should operate in order to be compliant. Instead the specification only states how the syntax of the output (the compressed bit-stream) should look like and how a decoder should interpret it. Even though the encoding process can be designed relative freely, it often becomes complex due to the many different steps that has to be considered in order to achieve good compression efficiency.

As a reference the following sections are provided in order to understand what some of the different tasks, performed by the encoder, are responsible for.

2.2.1 Frame types

A well proven method to achieve good video compression is to only include the difference between the current frame and the previous ones. This works by having frames depend on other frames that have already been compressed (these frames can be either chronologically older or newer). The frame dependencies form a directed graph that does not contain any cycles.

For this to work it is required that some frames (at least the initial one) can be compressed without referencing any other frames. These type of frames, called *Intra-frames* (I-frames), uses the redundant information within the frame in order to compress it. Most often I-frames does not achieve the same compression efficiency as frames coded by referencing other frames, called *Inter-frames* (P-frames), but this dependents on how well the frame's content matches the predefined Intra compression modes. As a result *Intra-frames* do generally require a larger amount of bits to compress than *Inter-frames* are therefore not used as often as *Inter-frames* [23]. Figure 2.1 shows the relationship between Intra- and Inter-frames.

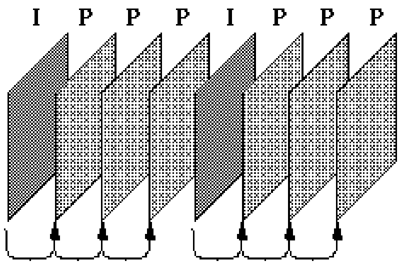


Figure 2.1. Inter- and Intra-frame relationship

A restriction of using Intra frames is that each macroblock can only be coded by not referencing other frames. The opposite does *not* hold meaning that macroblocks in P-frames can be coded with either inter- or intra macroblock. This decision is determined by which mode gives the best compression efficiency (i.e. best quality per bits required) for that specific macroblock.

One reasons why Intra-frames are inserted in the compressed bit-stream is to handle errors during decoding (e.g. from the transport medium) which would otherwise been continuously propagated to the next frame when P-frames are used. Another reason is to support seeking the bit-stream during playback. Without regular I-frames the decoder would have to decode all previous frames (because of inter frame dependencies) before the requested frame could be decoded. When using *I-frames* only the *P-frames* between the requested frame and the nearest *I-frame* needs to be decoded.

As mentioned earlier video compression is lossy (i.e. not completely reversible). One problem that this can result in is that the encoder's and decoder's views of how the frame looks like can start to drift. This can happened because the decoded frame is not exactly equal to what the encoder used when compressing the video. The solution is to have the encoder also immediately decode the image after having compressed it, this so called reconstructed image is then used when calculating the difference for any following frames. This step adds extra complexity to the encoder but ensures that both the encoder and decoder uses the same frames as reference.

2.2.2 Macroblock coding

The two types of methods to code a frame with (inter and intra) affects how the macroblocks within the frame are compressed. Choosing between intra and inter is only one step in order to compress a macroblock. Briefly the actual procedure to compress a specific macroblock is as follows:

1. First the macroblock is predicted to look similar to the original macroblock. This is done using either inter or intra prediction methods.

2. For each pixel in the $16x16$ macroblock the difference between the predicted and the original block is computed. These 256 differences are then what is transmitted in the compressed bit-stream.

3. Before the 256 difference values are added to the bit-stream they are compressed using a transform and quantification step. This compression works by using knowledge that the difference values are most often near zero.

4. The decoder then first predicts the macroblock using same prediction method as the encoder, then uncompresses the 256 difference values and applies them to the predicted macroblock in order to get the final image.

Predicting a macroblock using intra mode uses the neighboring pixels of already coded macroblocks. In H.264 frames are coded using the upper-left macroblock first and then continued row-wise with the macroblock to the right. This means that each macroblock depends on the neighboring pixels to the left, upper-left, upper and upper-right. The H.264 [17] specification have stated four modes that allows prediction of 16x16 blocks and nine modes to predicts different 4x4 blocks. Usually predicting a macroblock using 16 4x4 blocks yields a better prediction than one of the four 16x16 modes but using the 4x4 modes requires higher complexity[23].

On the other hand inter prediction involves dividing the $16x16$ macroblock into sub-blocks (from $16x16$ down to $4x4$) and searching previously coded frames for similar matches. Since each sub-block can match different parts of the reference frame each sub-block is therefore represented by a unique motion-vector. Finding the motion-vector that gives the best prediction is time consuming, especially when small block sizes are used. There are many different search algorithms to use for trading search complexity for slightly less optimal matches.

2.2.3 Transformation and quantization

The goal of the transform step is to take the delta (residual) macroblock and convert it to a format that is easier to compress. Since a macroblock is $16x16$ pixels large and the transform works on $4x4$ blocks it is applied 16 times for each macroblock.

The H.264 specification has defined the transform based on the famous Discrete Cosine Transform (DCT), but with the main difference that it is simplified to be computationally faster. A few properties of the transform is that it only works with real numbers and that it is completely reversible. This is important since the inverse transform is performed by both the encoder and decoder.

Quantization on the other hand is the main contributor for loss of quality in video compression. The output of the transform is a sparse $4x4$ matrix containing mostly zeros but also a few numbers called coefficients. These coefficients are then subject to quantization where they are scaled by a quantization parameter (QP), which is determined by the encoder. As the QP value increases so will the coefficients become smaller and the number of zeros increase. Having many zero coefficients leads to more efficient compression and lower bit-rate for the compressed video. Due of the fact that the quantified coefficients cannot be completely restored to their original value a higher QP also means that quality is degraded.

2.2.4 Rate-distortion optimization

When determining the best mode and sub-block size to code a macroblock with, the encoder can use a rate-distortion (RD) metric in order to trade between how good the prediction is and the number of bits required to use it. If used, this RD metric is then often calculated with the Lagrange cost function, expressed in (2.1).

$$J_{RDO} = D_{SSD} + \lambda R_{Act} \tag{2.1}$$

Here λ is a function based on the QP value used to quantize the coefficients for each macroblock. The function is expressed as

$$\lambda = 0.85x2^{(QP-12)/3}. \tag{2.2}$$

D_{SSD} is the distortion metric and is calculated using the *sum of squared difference* (SSD) functions. The SSD function is used to compare the difference between the original and the *reconstructed* macroblock. R_{Act} represents the cost in number of bits required to encode the macroblock using a given QP and a specific inter or intra mode.

Calculating SSD on the *reconstructed* macroblock has a relatively high computational cost because this involves performing the whole transform and inverse transform process. Since fast encoding requires low complexity in the encoder, a simplified cost function can be used for calculating the rate-distortion metric. The H.264 reference encoder has implemented a low complexity rate-distortion algorithm shown in (2.3).

$$J_{SA(T)D} = D'_{SA(T)D} + \lambda' \mathrm{MV}_{cost} \tag{2.3}$$

In this equation λ' is equal to the square root of the normal (QP-dependent) λ value. $D'_{SA(T)D}$ is the SA(T)D (sum of absolute (transformed) differences) value between the original and the *predicted* macroblock. The predicted block is calculated differently depending on if intra or inter prediction is used. At least in the case of inter prediction the computational cost of computing the predicted block is insignificant compared to calculating the reconstructed block. MV_{cost} is the number of bits required to code all inter motion-vectors in the macroblock.

2.2.5 Deblocking filter

The deblocking filter is performed both by the encoder and decoder after macroblocks have been decoded (or reconstructed), its goal is to reduce blocking artifacts that occurs as a result of the compression. The reason for using the filter on the encoder side is that a filtered image gives better motion compression for future frames. Richardson[23] explains this fact as "the filtered image is often a more faithful reproduction of the original frame than a blocky, unfiltered image".

Running the deblocking filter is only a matter of iterating all the edges between blocks (e.g. 4x4, 8x8 or 16x16 for luma pixels). This is performed both vertically and horizontally with a "boundary strength" parameter that is calculated depending on what type of edge is currently being processed. One reason for having a dynamic strength parameter is to try to only remove artificial edges while preserving edges that existed naturally in the original image.

2.3 Performance optimizations

According Paul Del Vecchio at Intel[32], the first step in optimizing any application for better performance is determining how to measure it. The type of metric however, depends on which application is being targeted. As an example a web application might measure number of requested pages per second whereas a database can count the transaction rate. In this thesis the metric used is number of coded frames per second.

After a performance metric has been determined the second step is finding a good configuring and workload to use for stress-testing the application. An important property with the workload is that it needs to be reproducible, this means that running the program with the same workload should give the same or almost the same performance measurement. An other important property is that the workload should be representative of the normal operating conditions. For this work a set of video sequences where selected based on the HD-VideoBench[3] benchmark for video encoders.

The third step is measuring the program before optimization in order to get a good baseline metric to compare all other measurements with. In the case of video encoding this represents the encoding throughput taken with the original video encoder.

Finally optimizations are evaluated iteratively by making small changes to the program, measuring the effect, compare it with the baseline measurement. Depending on the outcome of the comparison either an alternative version is tried or enhancements are made to the existing version and the iteration restarts.

In this thesis performance optimizations were implemented using two approaches. The first one uses low-level vector (also called SIMD) instructions and the other one parallelizes the encoder. Both approaches works by better utilizing all the capabilities offered by the existing computer hardware. Since both SIMD (single instruction multiple data) and parallelization uses different aspects of the hardware there are no restrictions on making a program use both types of optimizations simultaneously.

2.3.1 SIMD extensions

Vectorization, *instruction level parallelism* (ILP) and *single-instruction-multiple-data* (SIMD) are different names for hardware instructions in the processor that allows working with multiple data elements in parallel. Normally a processor core works by first loading a unit of memory into an internal register, then performing some work on that memory (e.g. addition or multiplication) and finally saving the result back into memory. With SIMD instructions the processor can load a whole segment of memory into a special register and apply the same operation to all elements in the registry at once.

One type of SIMD instructions is the *streaming SIMD extensions* (SSE) which allows up to 16 arithmetic operations to be executed simultaneously. Compared to performing the instructions sequentially this can result in significant speed improvement. In order to add SSE support to a program there are three different methods that can be used:

1. By using a vectorizing compiler.
2. By writing SIMD instructions as assembly code.
3. By using intrinsic SIMD functions which are written like normal C code.

Writing SIMD assembly instructions gives total freedom to the developer but leaves little room for the compiler to optimize the code. On the other hand when using a vectorizing compiler the responsibility is completely moved to the compiler alone, giving the developer no options to affect the output. Using intrinsic functions is a compromise where assembly instructions are slightly abstracted, offering the developer freedom to choose which instructions to use. The real benefit is that since the compiler is responsible for producing the object code it can perform normal code optimizations e.g. reordering and optimal register allocations.

2.3.2 Multi-core parallelism

Compared to SIMD instructions, thread-level parallelism (TLP) works at a much higher abstraction level than instruction-level parallelism. Instead of executing individual instructions in parallel, the same (or different) part(s) of the program can run concurrently using its own processing core inside the CPU.

Traditional software is most often written sequentially meaning that they can only take advantage of a single core on the computer. For many years it was most often been the case that the computers only had a single core so a sequential program did actually take advantage of the whole

CPU. Today almost all general purpose computers are multi-core meaning that they have separate execution and arithmetic units that can operate independently of each other. When a sequential program is run on a multi-core computer only a small fraction of the CPU gets utilized (1/4th on a four core computer). The current trend in CPU development is towards adding more cores instead of increasing the frequency (making each core run faster), which makes it even more important to consider multi-core programming. Without getting into details the major reason for this shift in architecture is that CPU manufacturers are approaching a limit of how small transistors can be used. The cause of this limitation is the so called *power wall* which prevents a single processor from running faster due to power constrains [22].

Threads are a concept offered by the operating system to take advantage of multi-core hardware. When writing a program, threads offers a way for the developer to perform computations in parallel. The actual thread execution and assigning of which CPU core to execute each thread on, is managed by the operating system and transparent to the developer.

2.4 Previous work

Optimizing H.264/AVC encoders and decoders have been a popular research topic for a long time. One of the most popular optimization techniques is by reducing the computationally complexity of encoding. One way to do this is by making the encoder take shortcuts by not evaluating all possible execution paths. Another common topic is by designing new hardware that perform some or all of the tasks in the encoder more efficiently. By using specific encoder chipsets better memory, pipelining and power efficiency can be achieved than by using regular hardware[10]. Another common focus is to target different low- or high-end architectures. Some examples are mobile devices and multi-core architectures[22], where existing algorithms are adopted to better utilize hardware traits like power efficiency and parallel execution.

2.4.1 Instruction-level parallelism (ILP)

Different ways of performing general vectorizing for either the encoder and decoder have been performed by many groups[6, 20, 11, 38, 26]. By performing low level optimizations on intensive parts the proposed implementations often gain good speed-up, but often never near the theoretical limit due to complexity of the algorithms defined in the standards [38]. The proposed implementations often comes without any quality or bit-rate loss, which makes them attractive especially since they do not require any special hardware [11].

Four different functions performed by the encoder have received special interest because of their relative high impact on encoding performance. As mentioned above the speedup is not always great due to the inherent dependencies in the calculation.

Zhou et al. [38] was one of the first to propose vectorizations for the decoder, which performs tasks in common with the encoder. Modules in common with the encoder that received better speed-up are the *SAD-calculation*, *Hadamard transform*, *Sub-pixel search*, *Integer transform and quantization* and *quarter-pixel interpolation.* Their work was continued by Chen et al. [11] which additionally targeted an H.264 encoder and received speed-ups between x1.3 up to x3.6 times faster for various modules. Lai et al. [20] combined SIMD vectorizations with fast inter mode selection and were able to speed-up the encoder by a factor of 18 times for low resolution video sequences with only negligible quality loss. Shengfa et al. [26] targeted the same set of modules as before and were able to speed-up the encoder to make it twice as fast, making it possible to achieve real-time encoding for 4CIF (704x576 pixels) video sequences. Azevedo et al[6] and Sihvo[28] have both attempted vectorization of the deblocking filter which is considered to be one of the most difficult modules to vectorize, because different execution paths can be taken for each pixel. The two groups proposes different implementation strategies but neither of them gives any results for general purpose x86 hardware.

2.4.2 Thread-level parallelism (TLP)

Parallelization, which is closely related to vectorization, is a topic that can be approached from many different angles. The H.264/AVC specification puts some limitations on the order which the different steps have to be computed, meaning that the problem domain is not ”embarrassingly parallel”. As a result of this different parallelization strategies have different costs in terms of overhead and scaling depending on architecture and input data. Despite these dependencies, parallelism can still be achieved to some degree.

The *2D-wave* is a common method where the work is divided per macroblock row of the frame. After a thread has started executing the first row a second thread starts after a slight delay. This solves the problem where macroblock depends on the block above and to the left to have been finished. Figure 2.2 illustrates the dependencies that each macroblock has on its neighbors. It was concluded by Amit and Pinhas [4] that the 2D-wave has some problems which prevents it from reaching perfect scalability. The top three reasons were cache misses, synchronization overhead of the wavefront algorithm and serial code.

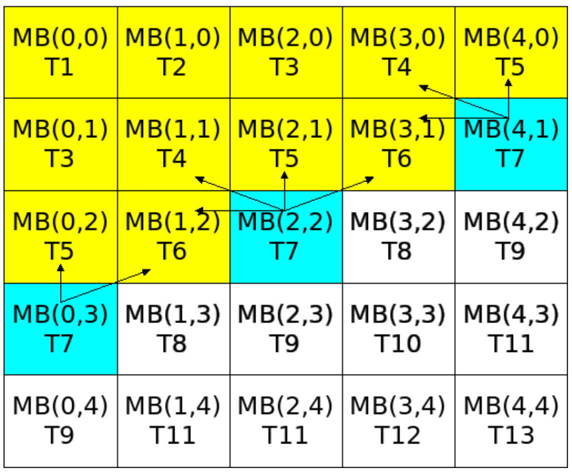


Figure 2.2. Macroblock neighbor dependencies

Assigning macroblocks to threads can be done dynamically using queues and calculating which blocks have all its dependencies calculated. Another way is by statically assigning blocks, in raster order, to threads as they become available. The latter strategy has the problem that since blocks take different time, threads will have to spend lots of time synchronizing for blocks to become ready. For the decoder it was concluded by Alvarez et al. [2] that static scheduling reaches a maximum speedup of 2.51 when using 8 processors. The authors concluded that although dynamic scheduling can result in higher overhead it is the preferred method in order to scale to larger amounts of processors.

The only solution to support more than one hundred CPUs is the *3D-wave* proposed by Azevedo et al. [5]. According to the authors this method achieves the best scaling at the cost of increased latency as frames are pipelined. Because of the real-time requirements where low latency are required this method was not evaluated.

The third method is called *slice-level parallelism* and works by dividing the frame into independent units called slices. A downside of using slices is that the compression efficiency is lower because redundant information between slices cannot be exploited. Rodrigues et al. [25] found this method to be more efficient than the 2D-wave for up to 32 cores, but notes that good scaling was also achieved by combining the two methods.

2.4.3 Motion estimation

Many studies have found inter motion estimation to be one of the most time consuming parts of encoding. Over the years several different algorithms have been proposed, each with different trade-offs between complexity and how close the result is to exhaustive search algorithm[9]. Diamond search (DS) by Zhu et al. [39] and the *hybrid unsymmetrical-cross multi-hexagon-grid search* (UMHexagonS) algorithm proposed by Chen et al. [12] are often used as reference when comparing new search algorithms. Diamond search offers a very low complexity search method which offers relative good compression but has the risk of being trapped in a local search minimum. By stopping the search in a local search minimum you get a prediction that contains more differences than the prediction at the global search minimum. UMHexagonS tries to solve this problem by combining different search patterns. This has the effect of better compression efficiency at a higher cost in complexity. UMHexagonS has also been adopted into the H.264 reference encoder.

2.4.4 Mode selection algorithm

. As mentioned above, the H.264/AVC specification defines many prediction modes. An exhaustive encoder evaluates all of them in order to find the mode with the best rate-distortion value. Fast mode selection is a popular group of algorithms where only a subset of inter and intra prediction modes and sub-block sizes are evaluated in order to find an optimal mode. Although being faster since not all paths are evaluated there is always a trade-off between complexity reduction and lower compression efficiency[16].

2.4.5 Real-time encoding

The topic of real-time encoding has received only limited focus in previous research. None of the different algorithms mentioned above solves the problem of achieving constant encoding rate on its own. Some authors like Bleakley et al. [18] tries to solve this problem by introducing both a fast mode selection algorithm which they combine with an algorithm for dynamically controlling the complexity (i.e. which modes and sub-block sizes are evaluated). The complexity is controlled both by categorizing each macroblock but also using a time scheduling method. The authors were able to achieve real-time encoding (at 20 frames per second for QCIF video) with similar compression efficiency as with the H.264 reference encoder (at 9 fps). The biggest gain was that encoding throughput were stable even for sequences with varying motion and complexity.

Chapter 3

Method

This chapter presents the various optimization techniques used to improve the video encoder. In the order presented the three evaluated optimization approaches are: SIMD optimizations, thread-level parallelism and encoding algorithm improvements. Before they are presented the behavior of the targeted encoder is analyzed and compared to with some other video encoders.

3.1 Benchmark input sequences

In order to benchmark the encoder a varied set of four raw video sequences were selected. Three of the sequences were selected from the HD-VideoBench test suite [3]. This suite is a benchmark for encoding and decoding of HD-video that, amongst others, defines a set of four video sequences which are representative for the ”HD-video domain”. During initial experiments with this benchmark it was discovered that two of the sequences, *pedestrian* and *rush_hour* were very similar in terms of complexity and degree of compression. In order to limit the workload to only four sequences and also add a sequence typical for video conferencing the *rush_hour* sequence were replaced with a recorded videoconferencing called *vidyo1*.

Figure 3.1 shows the selected sequences, which contains different types of motion and varying levels of detail. This is important to consider since this greatly affects the complexity of the encoding. More specifically *vidyo1* represents a video conference scenario with fairly little amount of motion, this sequence is by far the easiest one to compress. *Blue_sky* contains a static scene with the camera panning over the tree tops. *Pedestrian* contains people moving by in front of a static camera. *Riverbed* shows pebbles and moving water in front of a static camera, this last sequence is also the most time-consuming to encode because of few similarities between nearby frames.

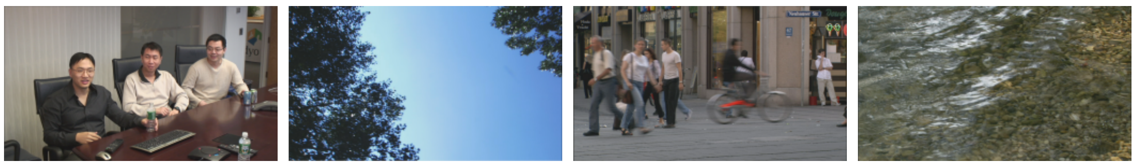

(a) vidyo1 (b) blue_sky (c) pedestrian (d) riverbed

Figure 3.1. Benchmark input sequences

3.2 Analyzing the existing encoder

The main goal of this project was to see if real-time encoding could be achieved on a typical workstation computer. In order to verify this goal simulations were performed on an Intel Core 2 Duo machine. Performance were both evaluated using built-in hardware monitoring counters in the processor and by measuring execution time with varying input video sequences and codec configuration parameters.

3.2.1 Targeted H.264 encoder and environment

The H.264 encoder studied for this work is a pure C program that is developed internally by Ericsson Research[1]. As of this date it fully conforms to the baseline profile of the H.264 specification. During this work only x86 versions of the encoder, that have been compiled using both Microsoft's Visual C++ compiler and Intel's® C++ compiler, were used.

All development was done in Microsoft's Visual Studio development environment. Both development and simulations were run on 32-bits versions of Microsoft's Windows Vista operating system. The encoder itself requires no third party libraries but for some of the experiments the following two external libraries were used:

- Boost C++ libraries[24] used for threading support.
- Intel® Cilk™ Plus[13], which is a framework for easily supporting multi-core computers.

During profiling a couple of external tools were used to analyze the encoder's performance:

- Intel® VTune™ Performance Analyzer, a profiler for measuring program performance.
- Acumem SlowSpotter™[1], which is a memory and cache profiler.

Additionally, for evaluating bit-rate and quality changes, a couple of Python scripts developed internally by Ericsson Research were used to automate BDPSNR and BDBR calculations.

3.2.2 Hotspot profiling

Intel's® VTune™ Performance Analyzer was used to analyze the existing encoder implementation. VTune supports measuring a large amount of events issued by the processor during execution of the program. After the program has finished, VTune provides different performance ratios derived from the events. These metrics can be analyzed at both executable, thread and function level, which makes it very convenient to discover bottlenecks and their impact on overall execution but also potential hotspots for later optimization. The following list contains the main ratios investigated and a short description of what they measure:

- **Cycles per Retired Instruction (CPI)** - A high CPI ratio might mean that instructions require more processor cycles than they should.
- **Cache Miss Impact** - A high value can indicate that cache lines are not used effectively resulting in more time spent with memory access.
- **Branch Misprediction Ratio** - This value indicates how good the processor is at predicting branches that are going to be executed.
- **Bus Utilization Ratio** - Indicates the level of activity between the processor and main memory. Optimal value depends on type of application, more on this later.
- **Translation lookaside buffer (TLB) misses** - A high value means that the program is accessing memory at different locations, causing the virtual memory cache (i.e. the TLB) to miss.

[1] `http://www.ericsson.com`

Analyzing the original encoder implementation was a good opportunity to get familiar with its architecture. After profiling the application, by using the various test sequences, a few encoder functions differentiated themselves from the rest of the code. An example of two functions that spent a large amount of their execution time doing memory read operations is half-pixel interpolation (19.0%) and SATD (18.9%). This was expected as SATD (and SAD) requires reading of large blocks (at most 16x16) that often crosses cache-line boundaries. Half-pixel interpolation is also known for being memory intensive, especially vertical filtering that interpolates based on pixels above and below the current row. Both of these functions were improved using instruction-level vectorization.

When evaluating the overall encoder performance, VTune indicated good performance. Level 1 and 2 data cache misses were very low (almost all functions had L1 data cache misses lower than 1 % of all memory transactions and Level 2 data cache misses were almost non-existent). Branch prediction misses varied slightly between functions, but were on average also low. Furthermore, branch miss-predictions were in the range of 0-5% for all branch instructions executed. The data bus utilization were around 5% and 10% indicating that memory transfer is not a major bottleneck in the program.

These results are also in line with Slingerland and Smith[29] and Xu et al. [35] which concludes that "multimedia applications generate similar or fewer number of data memory references per instruction" [35] when compared with other types of applications (e.g. technical, financial and text based). Surprisingly multimedia applications have fewer number of cache miss rate than other applications and more importantly "larger input data size does not necessarily result in a higher cache miss rate". The authors explains these finding as to be the blocking algorithms used in multimedia applications. When comparing the TLB (memory access) behavior their conclusion was that multimedia applications performs equal or better (when comparing with floating point performance heavy applications).

3.2.3 Comparison to other H.264 encoders

According to Alvarez et al. [3] the two most well-known H.264 encoders are the "JM Reference Codec"[27] and the x264[31] open-source encoder. The JM encoder is the reference H.264 implementation designed by the H.264 standardization bodies for verifying and improving the codec. It's useful for experimenting with the new features but exhibits very low performance (in execution time), since fast execution is not its biggest design goal. x264 on the other hand is an open source project that has written an H.264 encoder from scratch. It has been optimized in probably all ways that are possible and constantly continues to improve its performance. According to the authors some of the optimizations are on motion estimation, SIMD optimizations, and parallel encoding at slice and frame levels.

x264 uses a smart preset system in which the encoding complexity can be lowered for slightly reduced compression and quality. Some throughput measurements were performed on a typical workstation machine (Core 2 Duo at 2.4 GHz) using a recent version (built on the 28 sep. 2010) of x264 with both assembler optimizations and slice-level parallelism. A HD-video sequence with low motion can be coded with frame rates between 5 to 80 frames per second and a video sequence with very high motion can be coded between 1 and 40 frames per second, depending on which preset is used. This means that x264 is very capable of coding HD-video at real-time.

Except for the varying algorithm complexity real-time encoding is also achieved thanks to SIMD assembly optimizations and by parallelizing the encoder. Table 3.1 and 3.2 shows the speed-up of the various optimizations when run on a Core i7 processor (having 4 cores plus hyper-threading and SSE 4.2 support). These measurements were taken using some the following x264 specific settings: baseline profile, medium preset, no rate-control, hex motion search, no scene-cut, no trellis and tuning for PSNR.

From both tables it can be seen that the SIMD speed-up is more than four times faster when run using a single thread. Like-wise the slice parallelism speed-up is almost as good but slightly lower; especially for low motion sequences which were very fast to encode. Although not shown here, it is expected that the gain of parallelizing the encoder will outcome the gain of assembly

Video sequence	SIMD	Slice TLP	Frame TLP	Slice + SIMD	Frame + SIMD
vidyo1	x4.8	x1.63	x1.64	x8.2	x7.2
blue_sky	x4.6	x1.71	x1.94	x7.9	x8.7
pedestrian	x5.0	x1.92	x1.95	x9.1	x9.5
riverbed	x4.5	x1.89	x1.95	x8.2	x8.8

Table 3.1. x264 encoder SIMD and Thread-Level Parallelism speed-up (Core 2 Duo)

Video sequence	SIMD	Slice TLP	Frame TLP	Slice + SIMD	Frame + SIMD
vidyo1	x6.0	x3.1	x4.3	x17.1	x20.5
blue_sky	x5.4	x3.2	x4.4	x16.4	x23.5
pedestrian	x5.6	x3.2	x4.3	x17.7	x24.6
riverbed	x4.9	x4.0	x4.3	x19.8	x22.3

Table 3.2. x264 encoder SIMD and Thread-Level Parallelism speed-up (Core i7)

	x264			targeted encoder		
Video sequence	Bit-rate	PSNR	Frame-rate	Bit-rate	PSNR	Frame-rate
vidyo1	961 kb/s	41.94 dB	4.6 fps	886 kb/s	41.38 dB	8.1 fps
blue_sky	3534 kb/s	41.54 dB	3.0 fps	3591 kb/s	41.09 dB	5.8 fps
pedestrian	3525 kb/s	41.60 dB	2.7 fps	4056 kb/s	40.80 dB	5.3 fps
riverbed	20229 kb/s	39.89 dB	1.6 fps	19911 kb/s	38.82 dB	1.9 fps

Table 3.3. x264 bit-rate and throughput comparison (no optimizations)

optimizations, as the number of cores increases.

Finally for reference, table 3.3 shows a comparison of how both encoders perform using similar settings (where possible). Although this comparison is difficult to make fairly due to both encoders having different algorithms it shows some characteristics of the targeted encoder. As an example we can see that it is about twice as fast in terms of number of coded frames per second (fps) at the cost of lower quality on all sequences. When comparing the compression efficiency it can be seen that the *vidyo1* and *riverbed* resulted in lower bit-rate whereas the other two have slightly higher bit-rate.

3.2.4 Execution breakdown

Iain Richardson's [23] illustration of a general H.264 encoder is shown in Figure 3.2. The encoder can be broken down into two parts, those which can be modified and those which are mandatory. From looking at the picture parts that can be changed are the ME (motion estimation), MC (motion compensation), Choose intra prediction and Intra prediction boxes. Parts that cannot be changed are the remaining boxes: T (residual transformation), Q (quantification), Reorder, Entropy encode, T^{-1} (inverse transformation), Q^{-1} (inverse quantification) and Filter.

Zhou et al. [38] have illustrated in Figure 3.3 (a) how the execution time is spent in an old version of the JM reference encoder. Lai et al. [20] have done a similar execution breakdown using a slightly more recent version of the JM encoder as seen in Figure 3.3 (b). Finally Figure 3.4 shows the execution breakdown of running the target encoder with high complexity rate-distortion settings, Hadamard transform on and CAVLC entropy encoding.

When comparing the three figures it becomes clear that motion estimation is the most time consuming part in video encoding. This is followed by the deblocking and half-pixel interpolation filter which takes about 17% of the total execution time. Other similar entries are the SATD calculation, which takes between 10-12%, Intra prediction that takes around 1% and the Integer transform that takes around 5-6%.

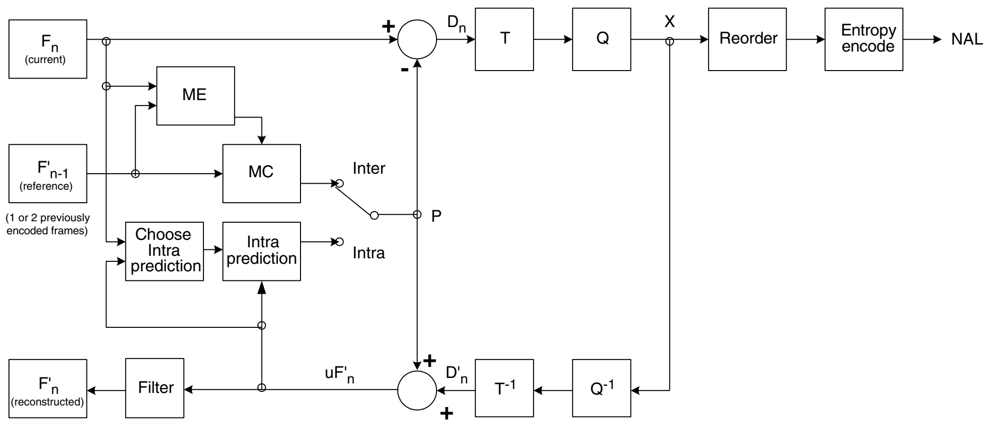
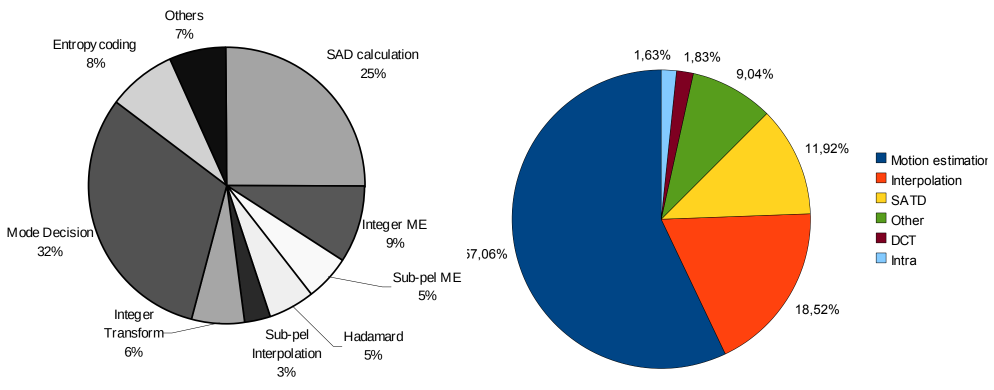


Figure 3.2. H.264 encoder functional overview



(a) Zhou et al. [38] (b) Lai et al. [20]

Figure 3.3. JM reference encoder breakdown

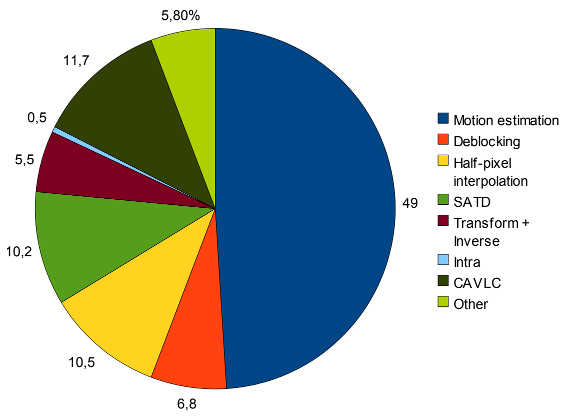


Figure 3.4. Execution breakdown of target H.264 encoder (vidyo1)

3.3 Compiler optimizations

Since optimizing the encoder using code rewrites can be both time consuming and error prone, I have evaluated the benefits of making the compiler perform all optimizations. For this experiment I used the Intel®C++ compiler and compared the result with the Microsoft's Visual C++ compiler with maximum code optimization settings on both compilers. Except for applying only code optimizations, the compiler also supports auto-vectorization and auto-parallelization. The goal of auto-vectorization is for the compiler to execute normal for-loops using SIMD instructions. The set of instructions used (e.g. MMX, SSE, SSE2 etc.) can be customized with compiler flags. The auto-parallelization optimization also targets for-loops but with the goal of running them in parallel. All burdens of managing the threads required for parallelism is completely managed by the compiler and fully transparent to the developer.

3.4 Instruction-level parallelism

The goal of using SIMD instructions is to get extra performance that is most often free in terms of both hardware resources and amount of side-effects of the vectorized function. It should be noted that not all functions can be changed to use this type of programming. Certain program characteristics will result in less or no speedup if vectorized, one example is that loading and saving 128-bit registries to memory requires that the memory is aligned in order to get the best performance. Without this requirement it could happened that a load is split over two cache lines requiring two loads to be issued and waited for before finished.

Since not every piece of code can be changed to exploit this type instruction-level parallelism, only a small (code wise) part of the program is targeted. Vectorization targets were selected both based on what previous work had been done in the literature and after profiling the application with different video sequences and varying motion estimation strategies. By running the encoder with minimal inter and intra motion estimation it was discovered that the biggest bottlenecks where de-blocking, half-pixel interpolation, residual transformation and block reconstruction (inverse transform) shown in Figure 3.4.

Half- and quarter-pixel interpolation Compared to previous video coding standards the H.264 specification allows motion estimation at quarter pixel level, allowing even better coding efficiency than previously. This is as a two step process that starts by pre-processing the whole reference frame with half-pixel interpolation and saving the result. Then during coding a bilinear interpolation filter is performed on-demand when quarter-pixel accuracy is requested. Pre-computing the half-pixel interpolation saves processing time since it is much slower than bilinear interpolation. One implementation by Sohn and Cho [30] strives to minimize the number of memory access by utilizing the symmetry of the half-pixel interpolation coefficients.

Half-pixel interpolation consists of three steps: calculating horizontal-, vertical- and "center" pixels, which is also performed in that order. Figure 3.5 shows the original full pixels (A,B,C,D,...) together with the horizontal half-pixels (b and s), vertical half-pixels (h and m) and the "center" pixel (j), the rests are quarter pixels.

Calculating the horizontal half-pixels were attempted by Warrington et al. [34] and is implemented by rearranging the interpolation equation. The normal 6-tap interpolation formula

$$((E - 5 * F + 20 * G + 20 * H - 5 * I + J) + 16)/32 \tag{3.1}$$

is rewritten to be evaluated from left to right as:

$$(((G + H) * 4 - F - I) * 5 + E + J + 16)/32 \tag{3.2}$$

When evaluating the formula each variable contains 8 luminance samples next to each other (2 bytes per sample is used to avoid loss of information). For horizontal interpolation, samples are read from left to right until the end of the row. When calculating the vertical pixels eight

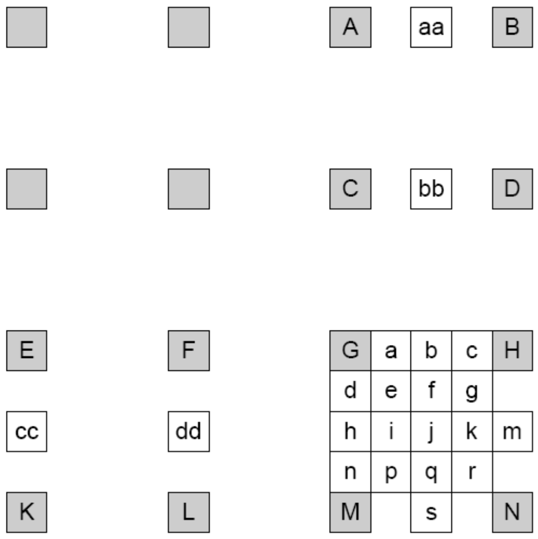


Figure 3.5. Quarter and half-pixels shown together with the original full/integer (gray) pixels

luminance pixels on each row are read starting from the first row and going down. The j pixel is calculated using a little different formula, which requires 4 bytes per sample to avoid overflow making it only possible to calculate 4 pixels in each iteration.

Quarter pixel interpolation is then relatively easy to calculate from the half and full pixel data.

Deblocking filter Deblocking is performed both by the encoder and decoder after a frame is decompressed, in order to remove errors introduced by the compression. Vectorizing this code has proved to be challenging since different execution paths can be taken for each 4x4 sub-macroblock. Deblocking consists of two parts; strength calculation and filtering. Most focus have been made on the filtering step since this is the most time consuming. Proposed implementations have been made for the Cell-processor[22] and also for general PCs [11, 28, 34]. The common theme in all implementations is that the filtering is performed without branches such that all paths are executed, masks are then used to control which of the results are written to memory.

The only difference between horizontal and vertical filtering is how the edge pixels should be loaded. Due to the way SSE arithmetic works by requiring two registers for each operand only horizontal filtering is applicable for vectorization. Vertical filtering is instead implemented by first loading an 8x8 block into eight registers then transposing the matrix and performing horizontal filtering on the transposed block. After filtering is done the block is transposed again and written back to memory.

On problem with vectorizing the filter is that either 6 or 8 different rows of pixels needs to be loaded for each edge (depending on which strength value is used). For strength 1, 2 and 3 three rows of pixels needs to be loaded above the edge and three rows of pixels below the edge. For strength 4 this is increased to include four rows above and four rows below the edge. When using SSE instructions this is problematic since you only have 8 (on x86 architectures) registers available, leaving no free registers to hold intermediate calculations. The method I finally settled on performs filtering on eight pixels at once using 1-byte arithmetic, which in theory leaves half of the register space available for holding temporary values.

Sihvo [28] has listed the different execution branches that should be evaluated based on the strength parameter and the pixel-level condition flag. Processing eight pixels at once means that two 4x4 blocks are involved in each iteration. I have found that the best performance was gained by first determining which branch the two blocks were applicable for and then running those branches only. For each of the eight pixels so called filtering masks (called $cond_i$, $pcond_i$ and $qcond_i$ by

Sihvo) were created and used to merge filtered pixels with pixels that shouldn't be filtered. Also an early termination strategy was used to stop processing if all 8 condition flags were zero meaning that no filtering should be applied to the whole edge. The benefit of early termination is even greater for vertical filtering where the costly transpose and extra memory stores can potentially be completely avoided.

Transform and quantification In the reference (JM) encoder the sequential transformation code, which consists of matrix multiplication, has been replaced with a more efficient multiplication free algorithm (called the "butterfly expression method"). Since the SSE instructions provides support for matrix multiplication using the combined multiply and addition instruction (PMADDWD), Zhou et al. [38] proposed a method to vectorize the transform using the original matrix multiplication method. This approach was then deemed less efficient by Yu et al. [36] which found that vectorizing the butterfly method achieves better speedup than the multiplication method.

The method I implemented performs transform and quantification on two 4x4 blocks using 2-byte precision. It was also combined with other tasks also required by the encoder such as zigzag ordering of the quantified coefficients and calculating the reconstructed blocks by performing de-quantification and inverse transform. The core transform which consists of both a horizontal and a vertical step was implemented by doing the vertical transform, transposing the two 4x4 matrices and doing the vertical transform again.

Luminance and chrominance coding Before the transformation step can be performed the residual data needs to be calculated. This is done by first predicting the macroblock using the motion vectors (for inter prediction). With the predicted data the residual block can be calculated as the difference between the predicted data and the original macroblock. The residual data is then transformed and inverse transformed and the reconstructed block is calculated by applying the inverse transformed residual data on the original block. The last step ensures that the encoder and decoder uses the same data as reference.

All these steps have to be performed for both the luma and chroma components. Expect for the transformation described above, these steps are fairly trivial to vectorize and can be done with either 8 or 16 elements in parallel. The prediction step is limited by how the macroblock should be coded, for example in 4x4 inter mode each 4x4 block has its own motion vector so the gain of vectorizing this is limited since only 4 pixels can be loaded for each row.

SAD, SSD, SATD The abbreviations stands for Sum of Absolute Differences, Sum of Squared Differences and Sum of Absolute Transformed Differences. They are commonly used in the motion estimation phase as a measurement to the difference between the original and predicted block. SAD is (as the name implies) a measurement of the (absolute value) differences between the pixels in the original and reference frames. Using squared difference instead of absolute gives a better indication of the prediction error but has the cost of higher computational complexity. The third method of using SATD has an even higher calculation cost. To calculate this metric the differences are first transformed using the Hadamard core transform before the sum of the transformed values are calculated.

Vectorizing all three methods are fairly trivial. The most complex part is the Hadamard transform, which was mentioned in the transform step above and can be reused here. Also when calculating the squared differences, care must be taken to use enough byte precision to avoid overflow.

3.5 Thread-level parallelism

The approach evaluates how the targeted encoder would benefit from parallelization. As already mentioned in the background, this area has been researched at many different levels. For this thesis two types of parallelization methods are compared; The first one is the popular 2D-wave method and the second one is slice-level parallelism.

3.5.1 Macroblock-level parallelism

There are two types of dependencies within a single frame that limits the effectiveness of parallelism. These two types are:

1. Coded macroblock neighbor dependencies.
2. Macroblock row dependencies.

The first dependency means that each macroblock requires macroblocks to the left, upper-left, upper-right and directly above to have already been finished coded. This dependency is introduced by the following functions in H.264: Inter motion vector prediction, intra prediction and mode decision, half and quarter-pixel interpolation, deblocking filtering, CAVLC entropy coding.

The second type of dependency differs from the previous one in that it can potentially span multiple rows. This effectively minimizes the possibility of encoding multiple macroblocks in parallel. It's introduced by both a skip run counter (mb_skip_run) and the delta quant parameter ($\Delta Quant$). The skip run counter signals how many macroblocks have previously been coded with SKIP. The delta quantification parameter tells how much the QP have changed since the previous macroblock. The solution I used to solve this dependency was by forcing macroblocks in the last column to be coded using a mode that is not SKIP, making mb_skip_run always be zero for the first column in each row. Also, since all simulations are done with fixed QP, the $\Delta Quant$ variable is always zero.

Because of the dependencies on left and upper macroblock neighbors the maximal number of processors that can be used is described by $round((mb\_width + 1)/2)$ [2]. For 720p video this means that the theoretical limit is *40* cores or threads. Due to limitations at the beginning and end of the frame where only a few macroblocks can run in parallel the theoretical maximal speedup for 720p is 21.43. Having more than 40 cores for a normal workstation computer is considered rare for todays computers (2010) but will definitely be a bottleneck as the number of CPUs quickly increase. As a comparison the maximum number of threads that can be used for Full HD-video (1080p) is 60, which is only a slight increase.

There are a couple of drawbacks that impacts the usefulness of macroblock-level parallelism. The first one is that creating "dynamic slices" (i.e. slices having a fixed amount of bytes) are now much more difficult. This means that all slice boundaries needs to be determined before the start of each frame. The impact of this limitation depends on the application requesting doing the encoding. An example is video streaming where it may be desirable to have slices of fixed number of bytes to better utilize network packet sizes. Another limitation is that traditional sequential rate-control algorithms, where the QP value is changed during the encoding, would not work because the current number of coded bytes are not known until the whole frame is encoded.

The biggest problem with macroblock-level parallelism is how to assign macroblocks to threads. This scheduling can be done either statically or dynamically. The biggest difference is whether it is known before execution which thread a specific macroblock is coded by. Since the previous research was not clear on which approach suites this problem best I have evaluated both approaches for this thesis. The two implementation methods below are implemented using static- (Method I) and dynamic-scheduling (Method II).

Method I: MB-parallelism with threads

In this implementation the Boost library[24] is used for parallelizing the encoder with threads. Macroblocks are assigned to threads statically meaning that it is known beforehand which thread a specific macroblock will be coded with. More specifically whole rows are assigned to threads in round-robin fashion. One idea behind this approach is that having a thread be responsible for a whole row should lead to better cache utilization and avoid having threads share the same cache line.

The problem of macroblock dependencies are solved by having a counter for each row that is updated atomically as the thread progresses. A thread then only needs to check the progress of the thread above and compare it with how far its own progress is. Locks and condition variables are then used to temporary block threads that have to wait for the thread above to finish coding its macroblock dependencies. Basically before coding each macroblock each threads checks if $p_{completed}[i-1] < p_{completed}[i]$ holds in which case the thread blocks until the above thread signals its completion. Since each thread only needs to communicate with one other thread the most efficient method was to have a unique lock and condition object per thread instead of one global lock shared by all threads.

Pre- and post-process tasks are also performed in parallel. Deblocking of each coded macroblock is performed after each macroblock have been coded, whereas half-pixel interpolation is performed after the whole row has been completed.

Method II: MB-parallelism with tasks

Task parallelism works by dividing the problem into work elements that can be processed independently. In this situation of coding a frame in parallel each task represent a single macroblock that should be encoded. The problem is then reduced to when to create new tasks that can be executed. Because of the dependencies to macroblock neighbors, as already explained above, a mechanism is needed to keep track of which macroblock can be processed after the current task is finished.

The implementation I settled on uses a small table with an entry for each macroblock in the frame. Each element contains the number of macroblocks that remains to be coded before the macroblock can be started. After a task has been finished it atomically decrements the value of the elements to the right in table and the element one row to the left below. If a table entry is zero after being decremented a new task is spawned for that macroblock. Initially the table is filled with the value 2 except for the first row and first column in which each macroblock only has 1 dependency, more precisely the neighbor to the left and above respectively.

For implementing the task mechanism I used the Intel® Cilk™ Plus framework. The Cilk Plus framework is an extension to C and works by creating tasks out of function calls. Each task or function to be executed in parallel is simply prefixed with the keyword *cilk_spawn*. In order to wait for the task to finish the keyword *cilk_sync* is used as a normal function call. All thread and task execution management is handled by the framework and transparent to the developer. The only changes needed are prefixing the function calls with the *cilk_spawn* and implementing the dependency table mechanism. Algorithm 1 shows a pseudo-code version of the algorithm.

Algorithm 1 Macroblock-level parallelization with dynamic scheduling (using tasks)

```
$Dep[1..Rows][1..Cols] \leftarrow 2$
$Dep[1..Rows][1] \leftarrow 1$
$Dep[1][1..Cols] \leftarrow 1$
cilk_spawn MacroblockTask(1, 1)
cilk_sync

procedure MacroblockTask(Row, Col):
call EncodeMacroblock(Row, Col)
decrement $Dep[Row][Col+1]$
decrement $Dep[Row+1][Col-1]$

if $Dep[Row][Col+1] = 0$ then
    cilk_spawn MacroblockTask(Row, Col+1)
end if
if $Dep[Row+1][Col-1] = 0$ then
    cilk_spawn MacroblockTask(Row+1, Col-1)
end if
end procedure
```

3.5.2 Slice-level parallelism

Slice based parallelism has also been evaluated before and is considered to scale better than macroblock parallelism. The main reason for this is that slices are encoded completely independent of each other, without any communication between threads. If deblocking filter is configured to not run across slice boundaries, then coding and deblocking can be performed completely in parallel. The half-pixel interpolation post-process step can also (mostly) be parallelized. This algorithm uses pixels from up to three rows above and below the current row; therefore six rows at each slice border are processed sequentially, after each frame has been coded in parallel. Although the sequential part is relative small compared to the parallel part it grows as the number of threads/slices increase. It should be noted that this extra filtering can also be parallelized, but requires an extra synchronization step.

The problem mentioned earlier, with limiting slices based on bit-size, is not an issue here since each thread can divide its chunk of rows into an arbitrary amount of slices. The problem is instead slightly different since with increased number of threads the slices becomes smaller. Having more slices than otherwise needed leads to less compression efficiency and higher overhead. Traditional rate-control is still somewhat possible. This can for example be implemented by dividing the frame's encoding quota by the number of threads having each thread strive to reach its assigned byte limit. The drawback is that different parts of the image often require different amount of bytes to code.

Method III: Slice-level parallelism with threads

I've implemented slice-level parallelism using the Boost threading library. With the overall design as follows:

1. Initially all rows in the frame are partitioned into sequential chunks (as equally as possible).
2. In each frame the total number of rows are partitioned into equal sizes based on the number of threads to use.
3. All threads start encoding their assigned rows.
4. After all rows are encoded threads start deblock and half-pixel interpolate their coded macroblocks.
5. Threads wait for all other threads to finish coding and filtering their slices.
6. Remaining half-pixel rows (at slice boundaries) are filtered sequentially (or in parallel).
7. Repeat for next frame.

Initially a configurable number of threads are spawned during encoder startup. At the start of each frame threads are initially waiting but then signaled to start encoding their pre-assigned chunk of rows. During encoding of the frame no synchronization is required between the threads. When all threads have finished encoding their part of the frame the small post-processing step is performed followed by the encoding of the next frame.

3.6 Algorithm changes

Two types of algorithms play an important role for achieving good compression efficiency in H.264. The first one is related to motion estimation and the other one is about macroblock mode selection. Both types have been extensively studied in the past and several implementations have been proposed. When implementing both types of algorithms there is always a trade-off between complexity and the level of compression. As an example a quick motion estimation search will find a target block that is quite similar to the original block but will require more bits to code the difference between the original and found macroblock. This is compared with an exhaustive search that will compare a larger amount of blocks in order to find one that matches equally good or better and thus requires less bits to compensate for the difference.

The difference between a fast mode selection algorithm and full rate-distortion evaluation is that some modes are not investigated when deciding how to code each macroblock. The full rate-distortion algorithm will calculate a rate-distortion metric for all modes and block sizes available in the H.264 specification (i.e. 4 16x16 intra modes, 9 4x4 intra modes and 7 inter block sizes). By trying more intra modes the probability of finding one that matches the original block increases. Likewise by partitioning the macroblock into smaller sub-blocks the distortion is lower since each sub-block can be searched independently. The problem of selecting which mode to evaluate is dependent on the result of the motion estimation algorithm but not vice versa.

What this means is that a fast motion estimation algorithm could also affect the behavior of the mode selection algorithm (but not the other way around). This is especially true for fast motion estimation algorithms which have the risk of being trapped in a local minimum. When the search stops at a local minimum the prediction becomes worse and as a result more bits are required to compensate for the error. When the inter prediction gets worse the effect is that smaller block sizes or intra prediction gets used more than would otherwise be needed.

3.6.1 Inter motion estimation

Motion estimation consists of both a full-pixel search which finds the best match among the pixels in the selected reference frames. Then a quarter-pixel search is done to find the best match among the (interpolated) sub-pixels that are located between the best full-pixel match and its four full-pixel neighbors.

The existing full-pixel search algorithm uses a gradient search method that iteratively compares surrounding pixels and increases the search area each time. This implementation was compared with the following algorithms:

1. Diamond search (DS) is a classic search algorithm comparing neighbor pixels.
2. Improved Diamond search (IDS) as proposed by Hu et al. [15].
3. Efficient Three-Step Search (E3SS) proposed by Jing and Chau [19].
4. Full-search (FS) also known as exhaustive search.

Diamond Search This algorithm works very simply by first calculating the SAD value for the initial start search location (either from origin or from candidate motion vectors). It then calculates the SAD value for its four neighbor pixels below, above, to the left and right. Finally the position that had the lowest SAD value amongst the five calculated pixels are selected as the new search center. This procedure is then repeated until the search center is selected as the best match meaning that all neighbors resulted in higher SAD values.

Improved Diamond Search This is a variation of diamond search where the best surrounding pixels two positions away are searched (instead of the direct neighbors as in DS). When the search loop is finished eight additional SAD calculations are performed on all neighbors surrounding the best match. Again the lowest SAD is selected as the best match and the search finishes. The IDS algorithm has the effect that the lowest SAD is found faster for high motion sequences. The downside is that at minimum 4+8 searches needs to be performed each time making it much slower than DS for low motion sequences.

Because of this fact I've made two modifications to the original algorithm. First when the search loop finishes only four neighbor pixels are searched (as in DS) and secondly early termination is introduced by not performing the last neighbor pixel search when the result of the first search loop is equal to the predicted motion vector.

E3SS One problem with the two algorithms above is that they have a risk of falling into a local minimum trap since they only compare nearby pixels. E3SS is a modification of the popular three-step search algorithm and tries to solve this problem by initially comparing a large square search pattern with a small diamond search pattern. If the best match is found in the diamond search pattern, a normal diamond search is performed until the best match is found. Otherwise traditional three-step-search is performed based on the best match in the large square search pattern.

Full-Search By definition FS always finds the global minimum in the search window since it compares the SAD value of all pixels in the search range. Because of the large number of SAD calculations needed, the complexity for this method is very high. As a result this algorithm is only used for reference when comparing the algorithms listed above.

Quarter-pixel search Together with the full-pixel algorithms above a fast sub-pixel motion estimation algorithm proposed by Lin et al. [21] was also compared with the existing implementation. The algorithm proposed by Lin et al. uses two important concepts in order to find a good match. First it reuses the SAD information from neighbor full-pixels which were calculated in the previous full-pixel search. Secondly it uses an elaborate surface classification scheme with early-termination in order to determine which partition to search in.

3.6.2 Macroblock mode and partition size selection

The other category of algorithms that plays an important role is the mode selection algorithms. Being able to exclude some of the modes and sub-block sizes can provide a huge reduction in complexity. The downside is that lower compression efficiency is often gained when macroblocks are not coded with their most optimal coding mode or split into smaller sub-blocks.

In this thesis three algorithms that have been proposed in previous research are examined. All algorithms claim to be fast and have low complexity with acceptable compression efficiency trade-offs. None of the algorithms have been designed solely for HD-video and for the requirement of very low complexity required to encode HD-video in real-time. Therefore the following modifications are applied to all three algorithms in order to further reduce their complexity:

Low complexity RDO As mentioned in the background (section 2.2.4 on page 13) there are at least two common methods that can be used to evaluate which mode and block size to code a macroblock with. It was estimated that adding additional modes to evaluate *and* using high complexity rate-distortion evaluation would make real-time encoding infeasible with current optimizations and hardware. As a result only low-complexity RD method (as mentioned in equation 2.3 on page 13) is used when testing the three algorithms.

SKIP-prediction A downside of using the low complexity rate-distortion evaluation is that SKIP prediction becomes problematic. When the SAD of the predicted SKIP block is compared with the SAD of the best predicted inter block the bit-cost of the motion-vectors makes SKIP prediction slightly biased. As a consequence of this I concluded that simply comparing the SKIP SAD is not enough for determining if a block should be coded with the SKIP mode. Instead the SKIP mode is only evaluated if $INTER16x16$ has been determined to be the best mode for that macroblock. Furthermore SKIP is also only used if coding the macroblock with $INTER16x16$ would result in no coefficients being sent in the compressed bit-stream.

Small inter block-sizes (4x4) An other significant modification is applied regarding the evaluation of the $INTER8x4$, $4x8$ and $4x4$ modes. Since this paper focuses on HD-video inter prediction modes smaller than $8x8$ are not evaluated. In order to verify this assumption a small experiment was conducted using the x264 encoder with full rate-distortion evaluation turned on. In this experiment it was shown that none of the macroblocks in the *vidyo1* sequence were coded using any of the $INTER8x4$, $4x8$ and $4x4$ modes. The result of encoding the other sequences was that between 0.2% to 0.5% of the macroblocks were encoded with any of these modes. Since the computational cost of the $INTER8x4$, $4x8$ and $4x4$ modes are much higher they are considered less computationally effective and thus not used in the algorithms below.

Algorithm I

The first algorithm is modeled after a paper called ”Fast H.264 Mode Decision Using Previously Coded Information” by Haung et al. [14]. It uses two concepts to ensure that only a few modes are evaluated in low complex video sequences. The first one is early skip detection that is based on an adaptive threshold from previously skipped macroblocks. The other one is that each macroblock is categorized into one of two ”inter mode groups” based on what type neighboring macroblocks have been coded with. As an example if all neighbors to the left and above have been coded as either SKIP or $INTER16x16$ then only the following block sizes are evaluated $16x16, 16x8, 8x16, 8x8$. The proposed algorithm allows even $8x8$ to be skipped if the result of the first three searches resulted in $16x16$ being the best match.

Likewise the decision of checking either $INTRA4x4$ or $INTRA16x16$ is determined by checking if any of the neighbors have being coded with $INTRA$ or $INTER8x8$. In that case $INTRA4x4$ is evaluated; otherwise $INTRA16x16$.

Algorithm II

The second algorithm is based on work described in ”Probability-based coding mode prediction for H.264/AVC” by Zhao et al. [37]. This algorithm is similar to the previous one in that it determines which modes to check based on how neighboring modes are coded. The classification works by ordering all possible INTER modes based on how often they occurs in the neighboring blocks. For example if all upper macroblocks have been coded using $INTER16x16$ and the previous macroblock to the left has been coded using $INTER16x8$; then evaluation order becomes $INTER16x16$, $INTER16x8$, $SKIP$, $INTER8x16$ and finally $INTER8x8$.

According to the authors ”because the list has a descending order of probability to be the best, mode decision could be early terminated when a larger RD cost is obtained”. This means that if the current mode results in a higher RD cost than the previously calculated mode, the iteration stops and the previous one is use to code the macroblock.

Algorithm III

The third implemented algorithm is based after the mode selection algorithm proposed in ”Real-Time H.264 Video Encoding in Software with Fast Mode Decision and Dynamic Complexity Control” by Ivanov and Bleakley[18]. This algorithm works by first classifying each macroblock into one of five classes based on three metrics (SAD, previous RD cost and a ”frame difference metric”). Different modes are then evaluated based on the class that were selected. As an example in the lowest class only the SKIP RD-cost is determined; similarly, in the second lowest class INTRA evaluations are also included. Likewise the complexity increases gradually to, for the highest class, include all inter block sizes and all intra modes.

Chapter 4

Results and analysis

This chapter starts with explaining how the various optimization approaches were measured. This is then followed by a review of gains and drawbacks of using the individual optimization techniques for each of the three optimizations methods. At the end of the chapter the three methods are compared with each other and their results are combined, in order to reach the goal set at the start of this project.

4.1 Benchmark coding options

The four video sequences used for measuring bit-rate, quality and throughput were previously explained in Chapter 3. As mentioned then, all sequences are of 720p resolution and sampled at 25 frames per second. When running the performance analysis 300 frames were used and sequences shorter than this were repeated in order to be 300 frames long.

During encoding the selected coding scheme was I-P-P-P, meaning that the only intra frame is the first one. All sequences were coded at constant quantization parameter (QP), since the goal is to measure the encoder and not the rate-control algorithm. Performance measurements were collected on an Intel Core 2 Duo processor running at 2.40 GHz. In order to measure parallelization scale-up, measurements were also collected on a Core i7 processor having 4 physical cores with hyper-threading (in total 8 logical cores). All simulations were run under the Windows Vista operating system on a 32-bit version of the encoder.

4.2 Instruction-level parallelism

Instruction-level parallelism (SIMD programming using the SSE2 instruction set) were evaluated using two different methods, manual and automatic vectorization of the encoder. Both results are compared with the original version of the encoder. The manual vectorization was implemented mostly using higher level intrinsic functions and some low-level assembly instructions.

4.2.1 Automatic vectorization

Intel's automatic vectorization feature works by targeting loops inside the program. More specifically only the inner-most loop (if nested) are considered a vectorization candidate. The result of the auto vectorization was that out of 550 potential loops only 3.6% was successfully vectorized by the compiler. Table 4.1 illustrates the most common reasons why a specific loop was not vectorized.

30.2%	not inner loop
28.7%	existence of vector dependence
17.0%	low trip count
10.0%	unsupported loop structure
6.6%	loop is not a vectorization candidate
3.6%	unsupported data type
1.5%	vectorization possible but seems inefficient

Table 4.1. Automatic vectorization fail reasons

A compiler settings controlling the "performance gain threshold" can be changed to control which loops should be vectorized. At 100% (default value) only vectorizations that are guaranteed to increase performance are made. When the threshold is decreased more vectorizations are performed at the risk of lowering the performance. By modifying the threshold parameter it became clear that only loops of the third type in the table above (*low trip count*) were vectorized when the threshold is reduced. At 100% only loops with more than 16 iterations (which is a common loop count in the encoder) were vectorized (e.g. the following code: *for(i=0; i<256; i++)*). When lowering the threshold below 100%, loops with an iteration count not known at compile time were targeted (e.g. *for(y=0 ; y<Height ; y++)*). When looking at the generated assembly code it can be seen that these loops starts with an conditional check so that the vectorized version is only executed when the loop count is large. Lowering the threshold further resulted in the loops with 16 iterations to become vectorized. When looking at the generated code it become clear that this code also performs a check to see if the vectorized or the normal version should be used. A guess from looking at the generated code is that, since the compiler knows the loop count for these loops, the check is to see if the memory accessed in the loop is aligned or not.

As seen from the Table 4.2 applying automatic vectorization achieved some additional speed-up compared to using the Intel compiler without any automatic vectorization. By decreasing the threshold the number of automatic vectorizations increased from 0.4% to at most 1.2% of all loops in the program. Despite the small increase in number of vectorizations there were no increase in execution speed-up. For the *vidyo1* sequence the speedup was slightly reduced whereas the other sequences did experience a slight increase in performance. Although some sequences did run faster the gains were too small to be noticeable. The major reasons for such a small speed-up is the small amount of loops that were vectorized. When lowering the vectorization threshold lower performance was achieved; my guess is because no changes were made for ensuring aligned memory allocations, which is important in order to get good vectorization performance.

4.2.2 Manual vectorization

Compared to using automatic vectorization, manual vectorization proved to be a great way to receive some actual speed-up results. A benefit from using vectorization is that the generated bit-

Sequence	Speed-up with default settings%
vidyo1	98.0%
blue_sky	101.8%
pedestrian	103.1%
riverbed	102.4%

Table 4.2. Automatic vectorization speed-up

Function	**Actual speed-up**	**Speed-up from previous research**
Quarter-pixel interpolation	x1.9	x1.3[26], x2.0[11]
Half-pixel interpolation	x3.0	
Deblocking (horizontal filtering)	x3.3	x1.1[38], x5.62[34]
Deblocking (vertical filtering)	x3,0	
Integer Transform + Inverse	x2.5	x1.3[11], x4.3[38], x4.3[36]
Motion estimation (SAD)	x3.4	x2.9[38], x3.5[11], x3.8[26]
SATD	x3.3	x2.6[36], x3.7[26]

Table 4.3. Individual component speed-ups results for manual vectorizations

Video sequence	Speed-up	Speed-up (using the Intel compiler)
vidyo1	216%	266%
blue_sky	198%	230%
pedestrian	184%	222%
riverbed	136%	135%

Table 4.4. Manual vectorization speed-up result (Core 2 Duo)

stream is equal to what was generated using the non-vectorized encoder. This means that neither compression efficiency nor quality were affected by introducing vectorization to the program. The individual speedup of each vectorized function is shown together with speedups that has been stated in previous research in Table 4.3.

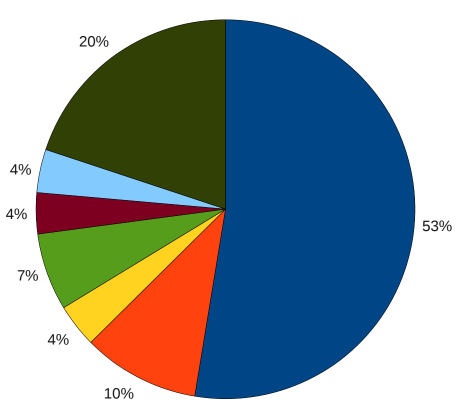
Table 4.4 shows the relative speedup of using the four video sequences. Since the encoding complexity varies depending on which video sequence is used (and especially the degree of motion) the vectorization speed-up also varies as a result of this. The best speedup, which was achieved in a few of the encoding runs, is around twice (200%) as fast when comparing the execution time with a non-vectorized encoder.

Additionally the speed-up of using the Intel compiler before and after the SIMD optimizations shown in the third column. All sequences resulted in better speed-up by using the Intel compiler than by using the Microsoft's Visual C++ compiler. One likely cause is that, since most of the vectorizations were written using intrinsics, the Intel compiler does a better job of optimizing this type of code.

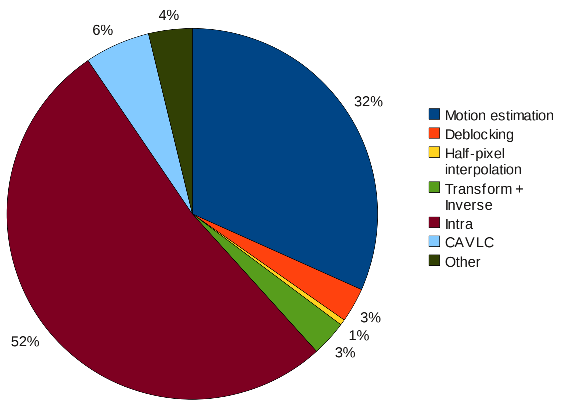
From the chart it is seen that low motion footages receives better speedup than more complex sequences. The reason for this is that the non-vectorized code starts to take a noticeable amount of execution time. For example generating the compressed bit-stream using CAVALC entropy coding is difficult to vectorize.

Figure 4.1 shows how the execution time is spent during encoding of the low motion sequence *vidyo1* and the high motion sequence *riverbed.* Both sequences are considered extreme cases of a fast and slow sequence to encode. In the first figure most of the execution time is spent doing inter prediction (i.e. motion estimation). This section involves both full- and quarter pixel search but also high complexity rate distortion evaluation with SATD. The second largest entry is the deblocking filter which were also successfully vectorized.

In the second figure it is clear that the most time consuming part is intra prediction and not motion estimation. Intra prediction was not directly vectorized as part of this work; mostly because each intra prediction mode requires its own SIMD routine, making the implementation



(a) vidyo1



(b) riverbed

Figure 4.1. Execution breakdown after applying SSE2 optimizations

time consuming. Another reason is that motion estimation is often the most time consuming part of the encoder (see the encoder comparison on page 21).

By comparing with the execution profiling done before SIMD optimizations (see Figure 3.4 on page 23), it is clear that both pictures looks mostly the same. Since the whole encoder was targeted for optimizations and most of the vectorizations received the same speed-up its a good indication that both figures looks the same. This would otherwise have indicated a good target for further vectorization.

In more detail, before SIMD optimizations were implemented motion estimation took about 59% (SATD included) of the total execution time, with SIMD optimizations the same part now takes 53%. The biggest difference is however in the half-pixel interpolation which decreased from 10.5% to 4% of the total execution time. The non-classified parts of the encoder (called ”Other” in the charts) increased, as a result of making the encoder faster, from 5.8% to 20%. This section includes parts of the encoder which are difficult to vectorize and will become an issue as more vectorizations are performed or enhanced.

4.3 Thread-level parallelism

Like the SIMD optimizations, parallelizing the encoder was also evaluated by making the compiler perform the parallelism automatically and by doing all parallelization work manually.

4.3.1 Automatic parallelization

Automatic parallelization was performed by using the Intel C++ compiler and modifying the compiler arguments to state that automatic parallelization should be attempted. The compiler then looks at all the loops in the program and determines if they are suitable for parallel execution.

For the parallelization to be safe the compiler must be sure that the loop has no dependencies between iterations, this however is not always easy to determine. Since the compiler must be able to preserve the sequential semantics, it becomes restrictive about parallelizing loops when dependences are found. As an example when the original encoder implementation was compiled no auto-parallelization took place (out of 476 potential loops). During compilation the compiler states if each loop was parallelized or otherwise a short motivation why it was not. Table 4.5 shows the result of performing automatic-parallelization with default settings.

0%	loop was parallelized
62%	loop has existence of parallel dependence
38%	loop has insufficient computational work
$< 1\%$	loop is not a parallelization candidate

Table 4.5. Result from automatic-parallelization with default threshold

From this table it can be seen that most of the loops (62%) are not parallelized because of dependencies either across loop iterations or by accessing the shared memory. This was in some way expected since the original code was not written for parallelization in mind. Code optimizations like loop-unrolling and incrementing array pointers instead of using the loop indexes could not, as of this date, be optimized by the compiler.

Also from Table 4.5 the third entry (38%) is interesting because this outcome can be tweaked by compiler settings. The compiler uses an internal cost metric to evaluate the complexity of all loops the program. If the estimated cost is higher than the cost of parallelizing the loop, it will be ignored. The compiler offers a method to modify the complexity decision by modifying a threshold value. By setting a lower threshold value, which spans from 100% to 0%, the compiler tries to perform parallelization event though performance is not guaranteed to increase. The default threshold value is 100% which basically means that parallelization is only performed if the compiler is 100% sure of a performance improvement.

When lowering the compiler threshold from the default value of 100% to 90% the compiler was able to actually parallelize some of the loops (1.5%). The effect of this change was however too small to be noticeable (or loops were not used due to settings specified at run-time). With the extreme settings of lowering the threshold down to 0% as much as 30% of the loops were parallelized. As can be seen in Table 4.5 the maximum number of loops that could be parallelized was 38%, but it seems that this number was not reached.

The speed-up from the automatic-parallelization with varying compiler threshold settings were however all negative. The worst case of having a zero threshold value resulted in an encoder that took five times longer to encode the same sequence. As the level of parallelism decreased so did the slowdown (e.g. the encoder was only 3% slower at a threshold of 75%).

OpenMP After this experiment some additional tests were made to evaluate the usefulness of automatic parallelization with OpenMP[8]. In its most basic form OpenMP is a set of compiler directives that can be annotated onto the existing code. The directives (or pragmas) are hints to the compiler how it can run that part of the code in parallel. The compiler is then responsible for managing the actual parallelism implementation, e.g. life-cycle of threads.

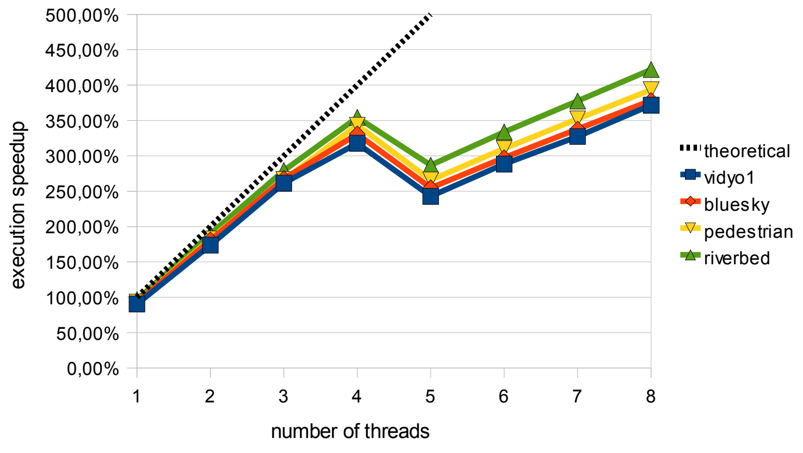


Figure 4.2. Relative speedup for macroblock parallelism with threads

Because of the many data dependencies related to video encoding, OpenMP could not be applied directly without making large architectural changes to the implementation. Instead a small experiment was made by moving the inter prediction logic out of the main coding loop and into a pre-processor stage. This loop was then annotated with the *omp parallel for* directive using additional attributes to control data sharing between threads. It was determined in the profiling of the encoder that inter prediction can take as much as 50% of the total encoding time so this should in theory give some good results.

Benchmarking the application on a computer with two cores showed that all video sequences were encoded in only slightly less time. The best speed-up of 10% was received for the *riverbed* sequence, which also contains the highest degree of motion. The worst speed-up were reported by the *blue_sky* sequence, which were only about 4% faster. The theoretical speed-up of using two cores with 50% sequential code is 33%, but this was however far from reached. There are several reasons why the usefulness of this method is limited. First since only a small part of the encoder were parallelized, the gain of adding more cores is less effective. Secondly because the encoder was not written with parallelization in mind, there is additional overhead required to ensure safe access of data-structures. In this experiment the *firstprivate* OpenMP attribute was used to make each thread have its own copy of the encoder data structure. Although this prevents race-conditions it comes at the cost of copying additional memory at the start of each frame.

For at least this type of application, the conclusion of both experiments is that automatic-parallelization is not beneficial to use and manual parallelization is the way forward.

4.3.2 Manual parallelization

Manual parallelization of the encoder was implemented using two different approaches: macroblock level parallelism and slice level parallelism. Furthermore I implemented macroblock parallelism using both low-level operating system threads and using a task abstraction library.

Method I: MB-parallelism with threads

Figure 4.2 shows the speedup of macroblock-level parallelism (implemented using raw threads) as the number of threads increase. As can be seen from the figure all video sequences scales roughly equally, but as the number of threads increase so does the variance between the four sequences as well. The best scaling at eight threads is received by the *riverbed* sequence. The reason behind this is that *riverbed* is also the most time-consuming sequence making the overhead of the thread communication and the sequential code relatively smaller compared to the work done by all threads.

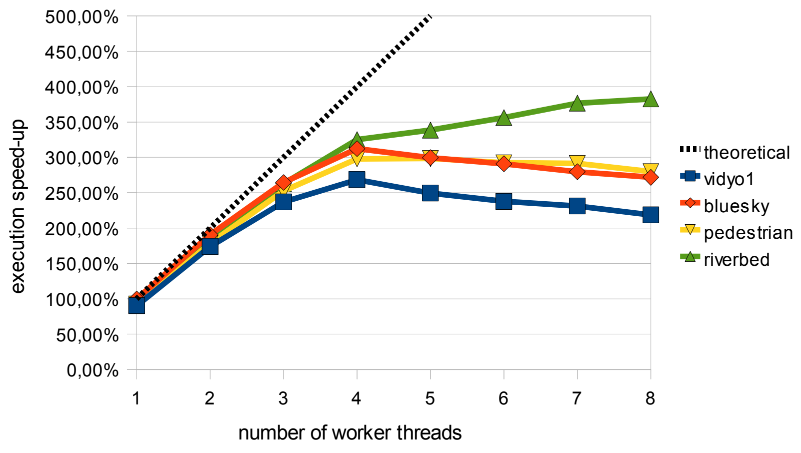


Figure 4.3. Relative speedup for macroblock parallelism with tasks (2 MB/task)

At four threads the best speed-up is 354% (also for *riverbed*) which should be compared to the maximum theoretical limit of 400%. When increased to five threads there is a severe overhead making the whole encoding run slower then with four threads. This is due the way that the operating system schedules threads. Initially the execution is similar to when run using four threads, which starts coding four rows in parallel. When the first thread continues on with the sixth row it has to block because the fifth row has not finished coding its row. This causes overhead both in terms of extra blocking but also because of extra context switching.

When increasing the number of threads up to eight the maximum speed-up increases to 430% *for riverbed* and from 317% (four threads) to 379% for the *vidyo1* sequence. The reason why the figures are not anywhere near the theoretical speed-up of 800% using eight threads is because the CPU only has four physical cores and simulates eight cores using hyper-threading. From these figures its clear that hyper-threading does give a slight increase in performance but those not result in perfect scaling.

Method II: MB-parallelism with tasks

Macroblock-level parallelism was additionally implemented using the Cilk Plus task library. Figure 4.3 shows the speedup as the number of worker threads assigned to the library increases. When comparing the speed-up at four and eight threads to the parallelism implementation using raw threads it can be seen that the threaded version do receive better speed-up for all video sequences and with all numbers of threads. This was to be expected as the library introduces overhead in terms of managing task scheduling but also because nearby macroblocks are not guaranteed to be coded by the same thread which hinders good cache locality.

In order to measure the overhead of the Cilk Plus method, the execution time spent in the encoder, the Cilk Plus library and in operating system calls were measured using 4 and 8 threads. The result from this experiment was that both parallelization implementations spends about the same amount of time in the encoder. The biggest difference was in the calls to the Cilk Plus library which took between 5% (4 threads) and 10% (8 threads) of the total execution time. Since these calls are not made in the version using raw threads, this overhead should be categorized as relative high.

An other difference when comparing the two figures is the scaling behavior of the four video sequences. Whereas the *riverbed* sequence continues to scale with increased number of threads all the other three sequences becomes slower as part of setting a larger worker thread pool.

One major reason why there were negative scaling for the fastest sequences is because of the configurable setting that controls how many macroblocks are encoded per task. This setting, which has no optimal value, depends on how time-consuming the sequence is to encode but also on how

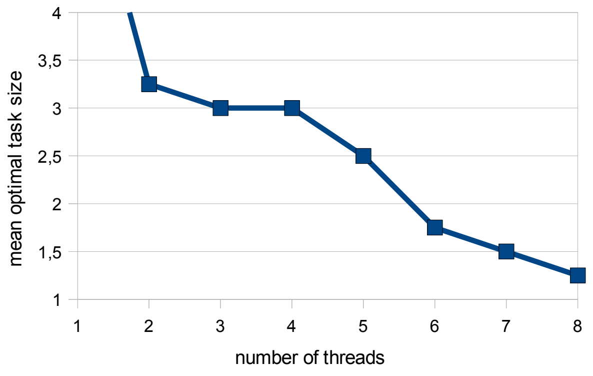


Figure 4.4. Cilk Plus optimal task size for each number of threads used

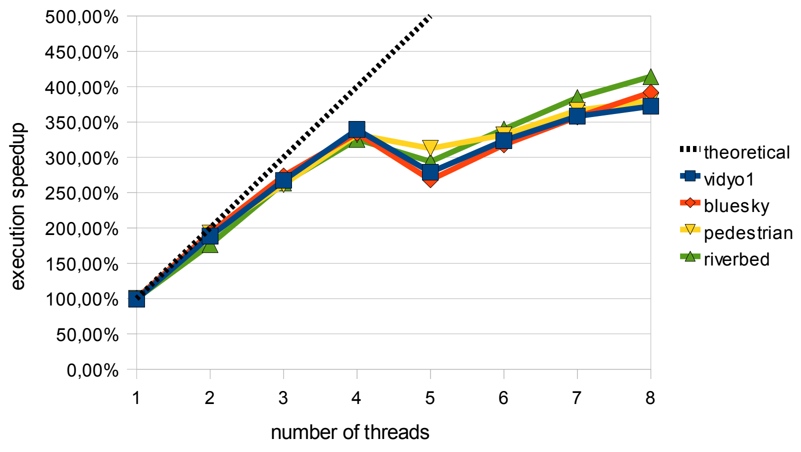


Figure 4.5. Relative speedup for slice parallelism (with threads)

many threads are used. By making each task only responsible to encode a single macroblock, the overhead of task management becomes noticeable for fast sequences (as can be seen in the figure). On the other hand by making each task responsible for encoding multiple macroblocks, the parallelization is hindered as it takes longer time before there are enough tasks to keep all threads occupied.

Figure 4.4 shows how the optimal task-size varies with the number of threads. With four worker threads the best speed-up was received with using 3 macroblocks in each task. When the number of threads increased to eight the best speed-up was measured when using a task size of 1 macroblock per task, except *riverbed* which was more optimal to run with 2 macroblocks per task.

Method III: Slice-parallelism with threads

The third parallelism method was using raw threads to implemented slice-level parallelism. Figure 4.5 shows the speedup of this method as the number of threads increase. As can be seen by looking at the figure, the scaling of the four video sequences are almost identical. The reason for this is that since there is no communication between threads, it matters less if the work done by each thread is either short or long; yet the *riverbed* sequence has slightly better scaling at 8 threads. This is most likely explained by the overhead of synchronizing all threads at the end of each frame, which becomes relatively smaller with more time-consuming sequences.

At four threads the speed-up of the *riverbed* sequence is 325% and at 8 threads it has increased to 414%. Even with slice-level parallelism, which does not have any locking, there is a gap to the theoretical speed-up at four threads of 400%. There are a couple of reasons for this. First there is still a small amount of sequential code which cannot be parallelized. This code is about preparing the internal data-structures at the beginning of each frame. In the case of slice-level parallelism there is also an extra sequential step related to the half-pixel interpolation of the slice borders. Since this can only be performed when all slices have been coded, it's performed as a sequential post-process step when all threads have completed their work. As the number of threads/slices increases so does the amount of sequential code as well.

Since having the sequential code grow as more threads are used is really bad for scaling, I also investigated if this sequential post-processing would benefit from also being parallelized. The results were that all sequences took longer time to encode using four threads or less. Even when using 8 threads only two of the sequences become noticeable faster by parallelizing this section. The reason why parallelizing this code is not immediately useful is that its very short (only 6 pixel-rows per slice border). Also because there is a synchronization penalty in having threads wait for all other to finish. It is however expected that performing this post-process step in parallel would be beneficial when more than 8 threads are used.

4.3.3 Parallelization overhead

Both types of manual parallelization strategies (slice- or macroblock-level) incurs different types of overhead, which affects the compression efficiency.

As previously explained, macroblock-level parallelism has a fixed overhead that is required to solve the skipped macroblock counter dependency. This dependency was circumvented by coding the last macroblock using the normal rate-distortion algorithm expect that skip-prediction is not evaluated (and therefore never taken). The result of this is a slightly higher bit-rate since a macroblock coded with SKIP requires (almost) no bytes to code. In CAVLC entropy coding each non SKIP macroblock signals how many macroblocks were previously coded using SKIP, meaning that SKIP blocks do incur a small bit-rate cost. Table 4.6 shows the overhead of macroblock-level parallelism.

sequence	BD-Rate	BD-PSNR
vidyo1	3.13%	-0.11
blue_sky	0.52%	-0.02
pedestrian	0.07%	0.0
riverbed	0.02%	0.0

Table 4.6. Macroblock-level parallelism overhead

Compared to macroblock level parallelism the overhead for slice-level parallelism increases with the number of threads. As mentioned earlier, the main reason is that more bits are required to compensate for the error introduced by worse prediction. Table 4.7 shows the affect on PSNR and bit-rate when varying the amount of threads (i.e. number of slices). From this table it can be seen that the overhead at four threads is mostly higher than compared to macroblock-level parallelism. At eight threads the bit-rate overhead is more than twice as high for the *vidyo1* sequence and more than 70 times larger for *pedestrian*.

	4 threads		8 threads	
sequence	BD-Rate	BD-PSNR	BD-Rate	BD-PSNR
vidyo1	3.0%	-0.10	7.0	-0.23
blue_sky	1.8%	-0.08	4.0	-0.18
pedestrian	2.8%	-0.14	5.0	-0.24
riverbed	0.7%	-0.04	1.4	-0.08

Table 4.7. Slice-level parallelism overhead

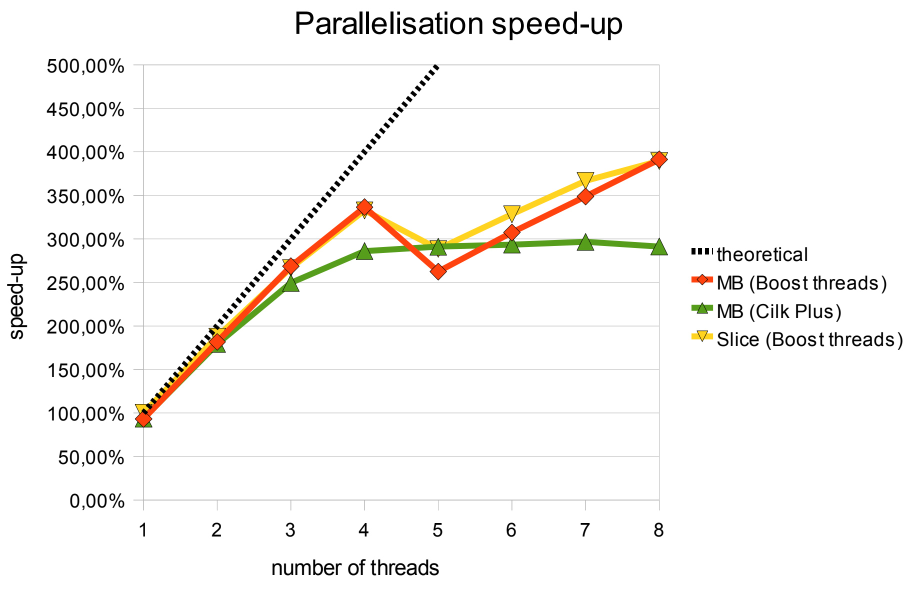


Figure 4.6. Parallelization speed-up comparison (*riverbed*)

4.3.4 Summary of the parallelization optimizations

Figure 4.6 shows the three parallelization methods side-by-side for each number of threads encoding the *riverbed* sequence. The slice-level parallelism curve is based on the best results from running with parallel and with sequential post-processing of the slice boundaries. Likewise the Cilk Plus curve is based on the best speed-ups of running with 1,2,3 and 4 macroblocks per task.

From this graph it is clear that macroblock-level parallelization using raw threads receives the best scaling both at four and eight threads. At four threads there is little difference between the other methods in terms of speed-up. At eight threads both macroblock-level parallelism using threads and slice-level parallelism scales much better than the Cilk Plus method.

In the figure the scaling of x264, using slice based thread-level parallelism (TLP), has also been added as a reference. At four threads the scaling seems to be equal to the implemented slice-level parallelism. When the number of threads increases up to eight the difference starts to increase. From the figure it is clear that x264 scales slightly worse than the implemented method.

An interesting side-note is that Cilk Plus method scales much better then the other methods using five threads. This is probably due to the more efficient assigning of tasks than the two threads methods. Likewise the slice-based parallelism method scales slightly better than macroblock-parallelism (with threads) at five threads. The most likely reason for this is more favorable data-partitioning with five slices than with four (45 macroblock rows is equally divided by 5 but not by 4).

4.4 Encoding algorithm improvements

Two different types of encoding algorithms were evaluated as part of this thesis. The first one is about how to perform fast motion estimation, which is used during inter prediction. The other one is about doing fast prediction mode and block size selection in order to quickly determining how to encode each macroblock.

4.4.1 Inter motion estimation

All motion estimation algorithms were evaluated using low complexity rate-distortion optimization (RDO). Additionally the number of evaluated modes and sizes were reduced to only compare INTER 16x16, INTRA 16x16 and SKIP. Additionally for this comparison only one reference frame was used, Hadamard transform (for RDO-calculation) was turned off, CAVLC entropy coding was used and assembly optimizations turned off.

Table 4.8 shows the result of comparing the compression and quality of all full-pixel motion estimation algorithms with exhaustive search (using a search window of 32 pixels). Full Search compares by definition the whole search window meaning that 1024 search points are compared. Unless otherwise specified the search window for the implemented algorithms are either unbounded or not applicable. Bit-rate and quality changes are evaluated using the BDPSNR (Bjøntegaard Delta PSNR) and BDBR (Bjøntegaard Delta Bit Rate) metrics, as explained in section 2.1.2. Since Full Search is between 80-90% slower than all the other search algorithms, all execution time results are relative to the Gradient search algorithm making it easier to compare the complexity cost.

As expected when looking at Table 4.8 the difference is more noticeable with video-sequences containing a higher degree of motion. The E3SS algorithm is the one closest to Full-search in terms of compression efficiency and quality, it's also the one with the highest complexity. In the other end of the spectrum we have the Diamond search algorithm which has the lowest complexity and also gets the worst result, especially for high motion sequences (i.e. *riverbed*). The most surprising result is on the *pedestrian* sequence where the Improved Diamond method resulted in faster execution than Gradient search even though it on average searched more points. One explanation is of side effects such as more coding of skip blocks and coding fewer residual coefficients as a result of a better search.

Additionally a sub-pixel motion estimation algorithm, which assumes that a full-pixel search has been performed (i.e. the SAD value is remembered) for the neighboring full-pixels, is also evaluated. This algorithm is compared with a sub-pixel search method (called *Pointwise Qpel*) that compares all half- and quarter-pixels surrounding the best full-pixel match. This algorithm can terminate early depending on if the SAD value found is higher or lower than the best match, so its not a true exhaustive quarter-pixel search.

Table 4.9 compares the original pointwise quarter pixel search algorithm with the search algo-

Method	BDRATE (%)	BDPSNR (dB)	# Search Points	ΔTime (%)
		vidyo1		
Gradient search	5.39	-0.18	5.8	0%
Diamond search	5.17	-0.17	5.7	-0.5%
Improved Diamond	4.30	-0.14	9.8	0.7%
E3SS	2.43	-0.08	14.1	1.5%
		pedestrian		
Gradient search	9.59	-0.44	9.7	0%
Diamond search	8.62	-0.40	6.1	-3.5%
Improved Diamond	5.17	-0.24	13.4	-1.5%
E3SS	2.96	-0.14	19.7	1.3%
		riverbed		
Gradient search	5.12	-0.26	16.6	0%
Diamond search	5.33	-0.27	10.2	-2.0%
Improved Diamond	3.55	-0.18	19.0	1.9%
E3SS	2.04	-0.10	29.6	6.0%

Table 4.8. Full-pixel motion estimation algorithms

Method	BD-RATE (%)	BD-PSNR (dB)	# Search Points
		vidyo1	
Gradient + Pointwise Qpel	0.00	0.00	9.46
Dia + Pointwise QPel	-0.21	0.01	8.76
Imp. Dia + Pointwise QPel	-1.04	0.04	9.02
Dia + [21]	1.74	-0.06	9.25
Imp. Dia + [21]	-0.28	0.01	8.9
		pedestrian	
Gradient + Pointwise Qpel	0.00	0.00	15.07
Dia + Pointwise QPel	-0.87	0.04	13.14
Imp. Dia + Pointwise QPel	-4.08	0.2	14.01
Dia + [21]	1.05	-0.05	12.33
Imp. Dia + [21]	-2.57	0.13	13.03
		riverbed	
Gradient + Pointwise Qpel	0.00	0.00	23.72
Dia + Pointwise QPel	0.20	-0.01	21.16
Imp. Dia + Pointwise QPel	-1.50	0.08	21.36
Dia + [21]	3.07	-0.16	16.69
Imp. Dia + [21]	1.05	-0.06	17.84

Table 4.9. Sub-pixel motion estimation algorithms

rithm proposed by Lin et al[21]. Both algorithms are evaluated using the two diamond full-pixel search algorithms presented above. All four combinations are then relative to the original search algorithm which consists of the gradient full pixel search and the pointwise QPel search.

From the table it is clear that the algorithm proposed by Lin et al. do offers an implementation with lower computational cost. This reduction comes however with the drawback of slightly worse compression efficiency (on average 2 percentage points lower) when comparing Lin et al's search algorithm with pointwise search algorithm. The biggest difference lies in the complexity where the algorithm by Lin et al. requires on average 10% less search points than the point-wise search.

Although some good results were seen with the algorithm proposed by Lin et al., especially in combination with the improved diamond search method, this sub-pixel algorithm were not used in any further evaluations of the encoder.

4.4.2 Mode selection algorithms

The goal of evaluating the mode selection algorithm was to see if evaluating more modes would be worth the extra complexity, but also if any of the algorithms proved more effective on HD-video. The three algorithms (described on page 31) are all examples of fast mode selection algorithms that uses different techniques to select prediction modes to evaluate. All algorithms are implemented using the low complexity rate-distortion method mentioned earlier (see page 13). The reason for this choice was that the additional complexity of evaluating more modes and also performing full rate-distortion calculation would make real-time encoding with the current optimizations difficult to achieve.

All algorithms were evaluated considering compression, quality, and complexity differences. Instead of comparing the result with an algorithm that evaluates all methods using full rate-distortion, the algorithms are compared to a more simplistic reference algorithm implemented as follows: The algorithm (called *Ref 1*) compares the rate-distortion value of $INTER16x16$ (and SKIP using the method mentioned earlier) with $INTRA16x16$ prediction (only the DC prediction mode) using the low complexity rate distortion method (equation 2.3).

Furthermore all simulations were performed using one reference frame, CAVLC entropy coding and assembly optimizations turned off. To (somehow) compensate for using the low-complexity rate-distortion method all three methods were run with Hadamard transform (SATD) turned on (but not during full-pixel search).

The result of the comparison is shown in table 4.10. For most of the sequences the evaluated algorithms resulted in better compression efficiency compared to the reference algorithm. One exception is the *blue_sky* sequence where all algorithms resulted in higher bit-rate. This was

Method	BDBR (%)	BDPSNR (dB)	# SADs/Modes	ΔTime (%)
			vidyo1	
Ref 1	0	0	2.1	0
Algoritm 1	-0.09	0.02	5.1	+73%
Algoritm 2	0.34	0.00	5.1	+76%
Algoritm 3	-1.15	0.05	2.9	+24%
			blue_sky	
Ref 1	0	0	2.3	0
Algoritm 1	2.99	-0.11	5.2	+83%
Algoritm 2	2.58	-0.10	5.0	+79%
Algoritm 3	15.19	-0.67	3.1	+46%
			pedestrian	
Ref 1	0	0	2.3	0
Algoritm 1	-8.05	0.44	5.0	+90%
Algoritm 2	-11.13	0.59	5.1	+82%
Algoritm 3	-4.28	0.22	3.1	+51%
			riverbed	
Ref 1	0	0	2.0	0
Algoritm 1	-12.41	0.71	5.9	+117%
Algoritm 2	-13.55	0.78	5.2	+81%
Algoritm 3	-8.15	0.45	3.4	+66%

Table 4.10. Fast mode selection algorithms with low complexity RDO

especially true for Algorithm 3 which performs especially poor. The main reason for this seems to be the macroblock classification scheme that uses static constants which, according to Ivanov and Bleakley[18], were not modeled after high resolution video. This unexpected increase in bit-rate is a major disadvantage of using these algorithms under the mentioned setup.

In the *vidyo1* sequence there are a little or no differences in the produced output compared to the reference algorithm. This is to be expected as compression efficiency is mainly dominated by the number of SKIP blocks used and because all algorithms uses the same skip method (as explained in section 3.6.2).

Overall the results of this evaluation are mixed, for higher motion sequences (*pedestrian* and *riverbed*) better compression was achieved. For all sequences there is severe increase in complexity compared to the reference algorithm, because some sequences results in questionable results makes this approach less valuable.

When comparing the algorithms with each other the one with the best compression efficiency seems to be *Algorithm 2*; although this algorithm do not have the lowest complexity cost it seems to behave between Algorithm 1 and Algorithm 3 when measuring the execution time.

As the result of evaluating more modes using low complexity rate-distortion gave both good and bad results, a second experiment using high-complexity rate-distortion was performed. In this experiment the reference algorithm was updated to use the high complexity RD method as mentioned in equation 2.1. Additionally the gains of enabling Hadamard transform in the low complexity RD evaluation (using SATD) case was also included in the comparison.

Table 4.11 shows the result of this evaluation. Some interesting points are that only using SATD or High RDO alone gives mixed results. Again the *blue_sky* sequence seems to be difficult to encode efficiently. As seen in the previous table the bit-rate increases instead of decreases even though the complexity is higher. Furthermore using SATD instead of SAD gives a slight improvement in both compression and quality, especially for higher motion sequences (*pedestrian* and *riverbed*). The best results are however achieved by combining SATD with high complexity rate-distortion. When using this method the problems with the *blue_sky* sequence are gone and good compression and quality is achieved for all sequences. The biggest downside is the severely increased complexity in the encoding process.

Finally when comparing the two tables there are some interesting differences. By using the combined SATD and High RDO method on the reference algorithm results in better compression efficiency and quality then the mode selection algorithms for all sequences. This is surprising since the Ref 1 algorithm evaluates fewer modes than the implemented mode selection algorithms.

Method	BDBR (%)	BDPSNR (dB)	# SADs/Modes	ΔTime (%)
			vidyo1	
Ref 1	0	0	2.1	0
Ref 1 + SATD	0.7	-0.02	2.2	+25%
Ref 1 + High RDO	-3.2	0.11	4.0	+40%
Ref 1 (Combined)	-1.7	0.06	4.0	+72%
			blue_sky	
Ref 1	0	0	2.3	0
Ref 1 + SATD	0.7	-0.03	2.3	+26%
Ref 1 + High RDO	12.5	-0.61	4.0	+40%
Ref 1 (Combined)	-0.3	0.02	4.0	+70%
			pedestrian	
Ref 1	0	0	2.3	0
Ref 1 + SATD	-2.5	0.13	2.3	+26%
Ref 1 + High RDO	-11.6	0.58	3.9	+37%
Ref 1 (Combined)	-13.6	0.7	3.9	+71%
			riverbed	
Ref 1	0	0	2.0	0
Ref 1 + SATD	-7.2	0.39	2.0	+26%
Ref 1 + High RDO	-19.1	1.18	3.4	+47%
Ref 1 (Combined)	-19.6	1.25	3.4	+79%

Table 4.11. Reference mode selection with higher complexity

Even though the complexity has increased the execution time is lower than Algorithm 2 which were considered the best one.

The conclusion is therefore that in order to get additional compression and quality, high complexity rate-distortion should be considered before evaluating additional modes.

4.5 Method comparison

Table 4.12 and Table 4.13 shows the individual speed-up of the instruction-level parallelism (SIMD) and the two thread-level parallelization methods. As can be seen from the parallelization summary (on page 42), macroblock parallelism and slice parallelism scales almost equally well at eight core on the *riverbed* sequence. When comparing with the rest of the sequences it becomes clear that slice-level parallelism scales slightly better for all other sequences. The reason for this is that slice-parallelism is implemented using a lock free algorithm whereas macroblock-parallelism do experience lock contention for low motion sequences. This means that as the encoder becomes faster (using either SIMD or algorithm optimizations) so does the cost of using locks starts increase.

Furthermore the speed-up of combining SIMD optimizations with parallelism are also shown in the two tables. The theoretical speed-ups of combining the two methods are not shown, but can be calculated by multiplying the SIMD speed-up with the parallelism value. By comparing the (measured) combined speed-ups with the theoretical it becomes clear that for the most time-consuming sequences (*riverbed* and *pedestrian*), the theoretical and measured speed-ups are almost identical. For the *vidyo1* sequences both combined speed-ups are about 30% lower than the theoretical values (which are x4.39 and x4.22 on the Core 2 Duo computer).

When comparing the speed-up with the result for the open source encoder x264 (see Table 3.1 on page 22) it becomes clear that x264 gets most of its speed-up from its assembly optimizations. This should be compared with the targeted encoder that benefits more from the thread parallelization (on the Core i7 computer). There are two major reasons for this; firstly only the SSE2 instructions set is used in this work, whereas x264 have implemented support for up to SSE 4.2. The other reason is that only a small part of the encoder were vectorized as part of this thesis whereas x264 have had extensive tuning for years. Interestingly the target encoder received greater parallelism scaling than the x264 encoder when run without any assembler optimization. More studies using a higher number of cores needs to take place in order to determine if this difference changes as the number of cores increases.

Video sequence	SIMD	Slice TLP	MB TLP	Slice + SIMD	MB + SIMD
vidyo1	x2.44	x1.80	x1.73	x4.1	x3.9
blue_sky	x2.12	x1.86	x1.67	x3.7	x3.4
pedestrian	x1.99	x1.79	x1.62	x3.5	x3.3
riverbed	x1.39	x1.80	x1.79	x2.4	x2.5

Table 4.12. Target encoder speedup with SIMD and Thread-Level Parallelism (Core 2 Duo)

Video sequence	SIMD	Slice TLP	MB TLP	Slice + SIMD	MB + SIMD
vidyo1	x3.19	x3.76	x3.72	x12.4	x11.3
blue_sky	x2.78	x3.92	x3.78	x11.1	x10.3
pedestrian	x2.40	x3.81	x3.93	x9.4	x9.2
riverbed	x1.53	x4.14	x4.21	x6.5	x6.7

Table 4.13. Target encoder speedup with SIMD and Thread-Level Parallelism (Core i7)

Table 4.14 shows the encoding throughput both before and after applying both the SIMD optimizations with slice-level parallelism optimizations. This table contains a rough number on the encoding throughput in terms of number of coded frames per second running on the Core 2 Duo workstation machine. This value was derived as the mean of running the encoder (300 frames) multiple times and varying the quantization parameter (QP) and using either 1 or 2 reference frames. More specifically the reference algorithm was used with Hadamard transform turned off and the low complexity rate-distortion method (equation 2.3).

As stated in the beginning of this thesis, the goal of real-time encoding varies depending on application requirements and uses. By using the lower requirement of 25 frames per second we can see that all sequences are possible to code in real-time. Additionally the first three sequences

	Original encoder		Optimized encoder	
Video sequence	Mean throughput	Std deviation	Mean throughput	Std deviation
vidyo1	15.3 fps	±0.3 fps	67.8 fps	±3.3 fps
blue_sky	13.2 fps	±0.6 fps	52.0 fps	±6.0 fps
pedestrian	13.0 fps	±0.5 fps	50.4 fps	±3.7 fps
riverbed	9.7 fps	±0.9 fps	31.0 fps	±4.1 fps

Table 4.14. Encoder throughput after applying all optimizations (Core 2 Duo)

can be coded at 50 frames per second, which a common frame rate used in HD-TV. Because of slight variations in the throughput only the *vidyo1* sequence is guaranteed to not drop below this limit.

This paper has also investigated the best way to add extra complexity in order to get better compression efficiency. Based on the throughput numbers above the extra complexity could be added to stay above the 25 frames per second limit. From the encoding algorithm evaluation section, it was concluded that modifying the rate-distortion method was the most efficient way to drastically increase the video compression and quality. This method to give the best results in terms of amount of increased complexity. Even though more complex motion estimation algorithms and evaluating more coding modes gave better result, those should only be considered after switching to high complexity rate distortion.

The throughput after applying this change is seen in Table 4.15. It can be observed that the goal of 25 frames per second is achieved for the first three sequences (again *riverbed* is very difficult to encode fast due to high motion). The affects on compression and quality seen in the last two columns is copied from the results in Table 4.11 on page 46.

Video sequence	Mean throughput	Std deviation	BD-Rate [%]	BD-PSNR [dB]
vidyo1	34.4 fps	±2.3	-1.7	0.06
blue_sky	28.5 fps	±3.2	-0.3	0.02
pedestrian	27.2 fps	±2.4	-13.6	0.7
riverbed	16.1 fps	±2.5	-19.6	1.25

Table 4.15. Optimized encoder throughput with high complexity rate-distortion (Core 2 Duo)

Chapter 5

Conclusions

In this Master's thesis two different strategies for optimizing an H.264 video encoder were addressed. It was shown that both instruction-level vectorizations and thread-level parallelism were good methods to increase performance of the targeted encoder. Furthermore it was shown that by combining these two methods the combined speed-up were very close to the theoretical (multiplied) value. One a Core 2 Duo workstation computer an execution speed improvement of between 300% to 400% were achieved for most of the tested sequences.

With the implemented optimizations the initial goal of real-time video encoding were achieved using low complexity encoder settings for all tested video sequences. Furthermore it was shown how extra complexity could be best added, in order to increase the encoder's compression efficiency. The proposed changes offered up to 18% improvement on compression, while still maintaining the soft real-time encoding goal on most of the evaluated sequences.

5.1 Summary of accomplishments

- Optimized the encoder using low-level SIMD (SSE2) instructions; increasing performance to be about twice as fast.
- Compared four fast motion estimation algorithms and listed their different gains and drawbacks. Introduced an earlier-termination condition to one of the algorithm, which severally reduced complexity while still maintaining good search results.
- Concluded that using *low* complexity rate-distortion can only increase compression efficiency and quality to a certain degree. Even when performing extensive evaluations of modes and block sizes. Showed that complexity is better spent by performing high complexity rate-distortion.
- Parallelized the encoder using three different methods. Concluded that best scaling were achieved for slice-level parallelism, but because of negative effects on compression showed that macroblock-parallelism is a good replacement.

5.2 Possible directions for future work

In this report I've presented a few different methods that can be used to make video encoding run faster. Due to limited scope of this thesis, a number of topics still remains to be investigated how they affect the encoding performance. The following list contains some areas which would likely increase the performance even more.

Using a richer SSE instruction set A large part of the encoder was rewritten using vectorization instructions. For this thesis only the SSE2 instruction set was used but later editions (e.g. SSE3 and SSE4) contains even more instructions. It is likely that these new instructions could replace a substantial part of the SSE2 code and thus be written using less instructions leading to increased performance.

Multi-core scaling The goal of this thesis was to target normal workstation hardware meaning that extreme parallelism scaling was not investigated. In order to evaluate the thread communication cost of macroblock parallelization more studies using a larger amount of cores are needed. As stated in previous research parallelism methods that target a single frame will experience lower speed-up as the number of cores increases. It's expected that in order for the encoder to receive good scaling in the future, a multiple frame parallelism approach is most likely needed.

Strict deadline encoding In this thesis no focus was made on encoding with strict real-time constrains, meaning that a fixed frame rate must be ensured. For real-time applications like e.g. live-video broadcasting, delivering video at a constant frame-rate is a requirement. Future work in this area would involve analyzing how an encoder could be updated to spend a fixed amount of time encoding a frame in order to reach a strict deadline.

Dynamic complexity control By better understanding the compression and complexity trade-of, the encoder can modify its settings at runtime to meet real-time encoding requirements. This requires an analysis of the effects (i.e. compression, quality and complexity changes) of specific functions in the encoder. Two example of such a functions are how to perform the rate-distortion evaluations and type of distortion metric (SAD, SSD and SATD). By turning these features on or off at run-time an encoder would be better able to compensate for varying amount of complexity in the video, resulting in a more stable frame-rate.

Parallelization friendly rate-control Macroblock-level parallelism and (to some extent) slice level parallelism makes traditional rate-control algorithms difficult to implement. One reason lies in the difficulty to estimate the bit-rate during encoding of the frame. This is because threads do not know the cost of previously coded macroblocks, which may or may not have been coded already. New types of rate-control algorithms, that take parallelism into account, needs to be investigated.

Bibliography

[1] Acumem AB. Acumem slowspotter™. `http://www.acumem.com/`, September 2010.

[2] M. Alvarez, A. Ramirez, M. Valero, A. Azevedo, C.H. Meenderinck, and B.H.H. Juurlink. Performance evaluation of macroblock-level parallelization of H.264 decoding on a cc-numa multiprocessor architecture. In *Avances en Sistemas e Informática*, volume 6, June 2009.

[3] M. Alvarez, E. Salami, A. Ramirez, and M. Valero. Hd-videobench. a benchmark for evaluating high definition digital video applications. In *IEEE 10th International Symposium on Workload Characterization*, pages 120 –125, sept. 2007.

[4] G. Amit and A. Pinhas. Real-time H.264 encoding by thread-level parallelism: Gains and pitfalls. In *IASTED PDCS'05*, pages 254–259, 2005.

[5] A. Azevedo, B. Juurlink, C. Meenderinck, A. Terechko, J. Hoogerbrugge, M. Alvarez, A. Ramirez, and M. Valero. A highly scalable parallel implementation of H.264. *Trans. on High-Performance Embedded Architectures and Compilers (HiPEAC)*, Sep 2009.

[6] Arnaldo Azevedo, Cor Meenderinck, Ben Juurlink, Mauricio Alvarez, and Alex Ramirez. Analysis of video filtering on the cell processor. Technical report, Delft University of Technology, 2008.

[7] G. Bjøntegaard. Calculation of average PSNR differences between RD-curves. In *ITU-T Q6/SG16, VECG-M33*, 2001.

[8] OpenMP Architecture Review Board. Openmp. `http://openmp.org/wp/`, September 2010.

[9] Canhui Cai, Huanqiang Zeng, and Sanjit K. Mitra. Fast motion estimation for H.264. *Image Communication*, 24(8):630–636, 2009.

[10] Tung-Chien Chen, Shao-Yi Chien, Yu-Wen Huang, Chen-Han Tsai, Ching-Yeh Chen, To-Wei Chen, and Liang-Gee Chen. Analysis and architecture design of an HDTV720p 30 frames/s H.264/AVC encoder. *IEEE Transactions on Circuits and Systems for Video Technology*, 16(6):673 – 688, jun. 2006.

[11] Yen-Kuang Chen, Eric Q. Li, Xiaosong Zhou, and Steven Ge. Implementation of H.264 encoder and decoder on personal computers. *Journal of Visual Communication and Image Representation*, 17(2):509 – 532, 2006. Introduction: Special Issue on emerging H.264/AVC video coding standard.

[12] Zhibo Chen, Jianfeng Xu, Yun He, and Junli Zheng. Fast integer-pel and fractional-pel motion estimation for H.264/AVC. *Journal of Visual Communication and Image Representation*, 17(2):264 – 290, 2006.

[13] Intel Corporation. Intel® cilk™ plus. `http://software.intel.com/en-us/articles/intel-cilk-plus/`, December 2010.

[14] Yi-Hsin Haung, Che-Yu Chang, and H.H. Chen. Fast H.264 mode decision using previously coded information. In *Tenth IEEE International Symposium on Multimedia*, pages 82 –88, dec. 2008.

[15] Feng Hu, Jia Liu, and Shuzhen Chen. Improved diamond search algorithm for H.264/AVC video coding standard. *Wuhan University Journal of Natural Sciences*, 13:62–66, 2008. 10.1007/s11859-008-0112-6.

[16] Yi-Hsin Huang, Tao-Sheng Ou, and H.H. Chen. Fast decision of block size, prediction mode, and intra block for H.264 intra prediction. *IEEE Transactions on Circuits and Systems for Video Technology*, 20(8):1122 –1132, aug. 2010.

[17] *Information technology – Coding of audio-visual objects – Part 10: Advanced video coding*, ISO/IEC 14496-10:2004 edition, 2004.

[18] Yuri V. Ivanov and C. J. Bleakley. Real-time H.264 video encoding in software with fast mode decision and dynamic complexity control. *ACM Trans. Multimedia Comput. Commun. Appl.*, 6(1):1–21, 2010.

[19] Xuan Jing and Lap-Pui Chau. An efficient three-step search algorithm for block motion estimation. *IEEE Transactions on Multimedia*, 6(3):435 – 438, june 2004.

[20] Yu-Lun Lai, Yu-Yuan Tseng, Chia-Wen Lin, Zhi Zhou, and Ming-Ting Sun. H.264 encoder speed-up via joint algorithm/code-level optimization. In Shipeng Li, Fernando Pereira, Heung-Yeung Shum, and Andrew G. Tescher, editors, *Visual Communications and Image Processing*, volume 5960, page 596038. SPIE, 2005.

[21] Weiyao Lin, D. Baylon, K. Panusopone, and Ming-Ting Sun. Fast sub-pixel motion estimation and mode decision for H.264. In *IEEE International Symposium on Circuits and Systems*, pages 3482 –3485, May 2008.

[22] C. H. Meenderinck. *Improving the Scalability of Multicore Systems, with a Focus on H.264 Video Decoding.* PhD thesis, Delft University of Technology, 2010.

[23] Iain E. Richardson. *The H.264 advanced video compression standard.* Wiley, second edition, 2010.

[24] Rene Rivera. Boost c++ libraries. `http://www.boost.org`, August 2010.

[25] António Rodrigues, Nuno Roma, and Leonel Sousa. p264: open platform for designing parallel H.264/AVC video encoders on multi-core systems. In *Proceedings of the 20th international workshop on Network and operating systems support for digital audio and video (NOSSDAV)*, pages 81–86, New York, NY, USA, 2010. ACM.

[26] Yu Shengfa, Chen Zhenping, and Zhuang Zhaowen. Instruction-level optimization of H.264 encoder using SIMD instructions. In *International Conference on Communications Circuits and Systems Proceedings*, volume 1, pages 126 –129, jun. 2006.

[27] Karsten Sühring. H.264/AVC software coordination. `http://iphome.hhi.de/suehring/tml/`, September 2010.

[28] Tero Sihvo. Subword parallel conditional execution in H.264/AVC deblocking filter implementation. In *International Conference On Signals And Electronic Systems*, pages 411–414, sep. 2008.

[29] Nathan T. Slingerland and Alan Jay Smith. Cache performance for multimedia applications. In *ICS '01: Proceedings of the 15th international conference on Supercomputing*, pages 204–217, New York, NY, USA, 2001. ACM.

[30] Chae-Bong Sohn and Hye-Jeong Cho. An efficient SIMD-based quarter-pixel interpolation method for H.264/AVC. In *International Journal of Computer Science and Network Security*, volume 6, pages 85–89, November 2006.

[31] The VideoLAN team. X264 - a free H.264/AVC encoder. `http://www.videolan.org/developers/x264.html`, September 2010.

[32] Paul Del Vecchio. De-mystifying software performance optimization. `http://software.intel.com/en-us/articles/de-mystifying-software-performance-optimization`, December 2008.

[33] Ben Waggoner. Understanding HD Formats. `www.microsoft.com/windows/windowsmedia/howto/articles/understandinghdformats.aspx`, January 2004.

[34] S. Warrington, H. Shojania, S. Sudharsanan, and Wai-Yip Chan. Performance improvement of the H.264/AVC deblocking filter using SIMD instructions. In *IEEE International Symposium on Circuits and Systems*, pages 4 pp. –2700, 2006.

[35] Z. Xu, S. Sohoni, R. Min, and Y. Hu. An analysis of cache performance of multimedia applications. *IEEE Transactions on Computers*, 53(1):20 – 38, jan. 2004.

[36] Sang-Jun Yu, Chae-Bong Sohn, Seoung-Jun Oh, and Chang-Beom Ahn. Multimedia: An SIMD - based efficient 4x4 2d transform method. In Osvaldo Gervasi, Marina Gavrilova, Vipin Kumar, Antonio Laganà, Heow Lee, Youngsong Mun, David Taniar, and Chih Tan, editors, *Computational Science and Its Applications – ICCSA 2005*, volume 3480 of *Lecture Notes in Computer Science*, pages 166–175. Springer Berlin / Heidelberg, 2005.

[37] Tiesong Zhao, Hanli Wang, Sam Kwong, and Sudeng Hu. Probability-based coding mode prediction for H.264/AVC. In *Proceedings of 2010 IEEE 17th International Conference on Image Processing*, Sep 2010.

[38] Xiaosong Zhou, Eric Q. Li, and Yen-Kuang Chen. Implementation of H.264 decoder on general-purpose processors with media instructions. In Bhaskaran Vasudev, T. Russell Hsing, Andrew G. Tescher, and Touradj Ebrahimi, editors, *Image and Video Communications and Processing*, volume 5022, pages 224–235. SPIE, 2003.

[39] Shan Zhu and Kai-Kuang Ma. A new diamond search algorithm for fast block-matching motion estimation. *IEEE Transactions on Image Processing*, 9(2):287 –290, feb. 2000.

TRITA-ICT-EX-2011:74

www.kth.se