UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

GABRIEL DIEGO TEIXEIRA

Desenvolvimento de uma arquitetura de hardware de um estimador de vetores de movimento de precisão sub-pixel seguindo o padrão HEVC

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Sérgio Bampi

Porto Alegre
2014

RESUMO

Este artigo descreve o Trabalho de Graduação 2 de Gabriel Diego Teixeira o qual propõe uma arquitetura de hardware de um estimador de vetores de movimento de precisão sub-pixel seguindo o padrão HEVC. Este trabalho tem como motivo a necessidade de se melhorar o desempenho dos codificadores de vídeo, em especial do novo padrão HEVC, que requerem uma grande capacidade computacional para realizar a compressão de uma seqüência de vídeo em alta definição com uma boa qualidade visual e alta taxa de compressão. O padrão HEVC possui diversos módulos e o módulo de estimativa de movimento é o consome mais recursos durante a codificação, embora também seja responsável pela maior parte dos ganhos na compressão de vídeo. Neste trabalho é apresentado uma proposta de arquitetura de hardware do módulo de estimativa de movimento fracionário, que faz parte do módulo de estimativa de movimento, que tem como objetivo reduzir a intensidade computacional do codificador.

Palavras-chave: Video-digital, HEVC, Estimativa de movimento, Estimativa de movimento fracionário

Development of a sub-pixel hardware motion vector estimation architecture according to the HEVC standard

ABSTRACT

This article describes the final project of Gabriel Diego Teixeira which proposes an hardware architecture of a sub-pixel motion vector estimator according to the HEVC standard. This project has as motivation the need to improve the video encoders, specially the new HEVC standard, which require a high computing capacity to perform the compression of a high definition video sequence with a good visual quality and compression rate. The HEVC standard has many modules and the motion estimation module is the most computing intensive task during the encoding, although it also allows most of the gains in compression during the video encoding. In this article is presented a proposition of hardware architecture of the fractional motion estimation module, which is part of the motion estimation module, which has as objective the reduction of the computing intensity of the encoder.

Keywords: Digital video, HEVC, Motion estimation, Fractional motion estimation.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ASIC	Application Specific Integrated Circuit
CODEC	Encoder-Decoder
CTB	Coding Tree Block
CTU	Coding Tree Unit
CB	Coding Block
CU	Coding Unit
DCT	Discrete Cosine Transform
DST	Discrete Sine Transform
GPU	Graphics Processing Unit
HEVC	High Efficiency Video Coder
MPEG	Motion Picture Experts Group
FPGA	Field Programmable Gate Array
PSNR	Peak-Signal Noise Ratio
PB	Prediction Block
PU	Prediction Unit
RGB	Red Green Blue
SVN	Subversion
TB	Transform Block
TU	Transform Unit
VHS	Video Home System

SUMÁRIO

1 INTRODUÇÃO

Existe uma demanda crescente por soluções audiovisuais que exigem uma qualidade cada vez maior da imagem. Os sites de compartilhamento de vídeo, os serviços de distribuição de mídia online e as ferramentas de vídeoconferência são uma amostra deste mercado. Em outra direção a baixa capacidade de armazenamento e a congestão das redes impõem uma demanda cada vez maior sobre a compressão deste material. E ainda existe a demanda por uma menor consumo de energia dos dispositivos. Mesmo com o avanço da tecnologia a preferência das pessoas mudam para uma outra tecnologia ainda mais restrita. A migração da plataforma PC para a plataforma móvel é uma amostra clara desta tendência, pois a última é mais limitada em armazenamento, rede e capacidade computacional pois esta plataforma mais recente usa a tecnologia Flash no lugar de disco rígido para o armazenamento, se conecta por 3G no lugar de rede cabeada e usa processadores de baixo consumo para dissipar pouca energia da bateria.

Como forma de atender a estas demandas foi lançado o padrão HEVC (SULLIVAN 2012). Com o HEVC é possível comprimir um vídeo com metade do número de bits que o H.264/AVC, popular padrão de codificação que antecedeu o HEVC, mantendo a mesma qualidade visual. Por outro lado este codec exige uma capacidade computacional muito superior ao H.264/AVC (BOSSEN 2012), o que demanda o desenvolvimento de tecnologias que permitam a compressão de vídeo em tempo real ou dentro de um tempo aceitável.

Para obter uma alta taxa de compressão o HEVC emprega várias técnicas novas ou que já existiam no H.264/AVC e que foram aperfeiçoadas. Uma destas técnicas é a estimativa de movimento. A estimativa de movimento melhora a compressão do vídeo removendo a redundância temporal entre os quadros de um vídeo (JACK 2007). Esta técnica se baseia no fato de que dois quadros próximos são muito similares exceto pelo deslocamento de alguns objetos. Assim para decodificar um bloco é necessário somente buscar um bloco no quadro anterior na mesma posição deslocado por uma distância indicada por um vetor de movimento. Os objetos na cena não necessariamente se deslocam por um número inteiro de pixels. Assim

é necessário buscar não um bloco em uma posição exata no quadro anterior mas sim um bloco virtual cujos pixels se encontram em uma posição entre os pixels reais. Esta técnica é chamada de estimativa de movimento fracionário. A estimativa de movimento envolve um grande volume computacional, em especial se for adicionada a estimativa de movimento fracionário nesta etapa, o que motiva o desenvolvimento de técnicas de cálculo de vetores de movimento eficientes.

1.1 Motivação

Diante da grande complexidade computacional do HEVC, é importante o desenvolvimento de arquiteturas que permitam uma codificação eficaz. O software de referência HM na versão 8.0 codifica vídeo a um ritmo cerca de 1000 vezes mais lento do que uma câmera de vídeo seria capaz de capturar (BOSSEN 2013) o que torna ele pouco prático para o uso comercial. Mesmo que fosse possível gravar o vídeo usando um codec em tempo real e depois transcodificar o vídeo em HEVC, a codificação de um vídeo de 2 horas de duração levaria mais de 83 dias. Naturalmente o software de referência HM não é recomendado para esse tipo de uso pois ele é feito apenas para validar uma implementação e não é otimizado para um processamento eficiente.

Para se alcançar um desempenho razoável para permitir a codificação em tempo real é necessário fazer implementações onde o tempo de execução do codificador seja rápido o bastante para a aplicação, sem que as outras qualidades sejam sacrificadas. Isto é possível modificando vários aspectos como a implementação (em software usando aceleração em SIMD/GPU ou em hardware usando ASIC ou FPGA), ou realizando refinamentos no algorítmo.

Como um dos módulos que consomem mais recursos computacionais no HEVC é o módulo de busca de vetores de movimento, em especial quando for feita a busca de vetores de movimento com precisão sub-pixel, é importante trabalhar sobre este módulo para que sejam obtidos os maiores ganhos globais de desempenho. Assim, vários trabalhos tentam refinar o algoritmo deste módulo para que ele execute mais rapidamente, com uma baixa perda para a qualidade do vídeo resultante (KAO 2006) (SUH 2004) (HE 2013) (DAI 2012). Estas técnicas existem desde a introdução do H.264/AVC o que explica a existência de trabalhos sobre o assunto antes da existência do HEVC.

1.2 Proposta do trabalho

Como forma de melhorar a eficiência dos atuais codificadores HEVC, este trabalho tem como objetivo implementar um bloco de estimativa de movimento fracionário, conhecido na literatura como Fractional Motion Estimation ou FME em hardware usando uma arquitetura eficiente. O estudo do funcionamento básico do HEVC e mais detalhadamente o funcionamento do bloco de estimativa de movimento e do bloco de estimativa de movimento fracionário será fundamental para a compreensão do mesmo como um todo. O estudo de arquiteturas de estimativa de movimento fracionário já implementados para H.264/AVC e HEVC possibilitará o conhecimento do estado da arte. Posteriormente será analisado o software de referência para uma compreensão técnica detalhada do algoritmo usado em HEVC. Finalmente será implementado um bloco de hardware que faz a busca de vetores de movimento, com testes de desempenho para comparar a performance em relação a outras arquiteturas existentes.

1.3 Estrutura do texto

Este trabalho está dividido em seis capítulos. Os capítulos iniciais serão focados na apresentação de conceitos fundamentais e os finais na apresentação da implementação e resultados.

No segundo capítulo serão apresentados os conceitos fundamentais de codificação de vídeo como a compressão de dados sem perdas e com perdas, os formatos de pixel, transformada, quantização e a estimativa de movimento, a última foco deste trabalho.

O terceiro capítulo será dedicado a uma apresentação sobre o funcionamento de codecs de vídeo, com foco no HEVC. Neste capítulo são apresentados também quais são as qualidades esperadas de um codec para que seja feita uma implementação eficiente, alguns codecs que precederam o HEVC e formas modernas de se implementar um codec. Neste capítulo será feita uma apresentação técnica do HEVC, voltado a apresentação do modelo de estimativa de movimento usado no HEVC.

O quarto capítulo será dedicado a apresentação de como é implementado um codec e como foi realizado o trabalho.

No quinto capítulo será feita uma descrição dos resultados obtidos mostrando as medidas usadas, a validação e uma comparação com outros trabalhos e no sexto será feita uma conclusão dos resultados.

2 FUNDAMENTOS TEÓRICOS

Este capítulo é dedicado a apresentação de conceitos básicos de compressão de vídeo. Os conceitos aqui apresentados não são estritamente os que são aplicados no padrão HEVC, mas sim são uma visão básica de como ele funciona pois estes conceitos não são o foco do trabalho. Mais informações sobre cada conceito devem ser encontradas nas referências apresentadas no texto ou em outras publicações.

2.1 Fundamentos sobre vídeo digital

Um vídeo digital é uma seqüência de quadros, que são imagens digitais. Para dar uma impressão de movimento a quem estiver assistindo a um vídeo, os quadros devem ser apresentados sucessivamente. A taxa em que os quadros são apresentados é dado o nome de quadros por segundo ou fps, do ingês frames per second. Esta taxa varia dependendo da aplicação. Vídeos para de monitoramento pode ter uma taxa de apenas 1 fps para maximizar a compressão enquanto vídeos 3D tem taxas de até 120 fps sendo que a taxa mais comum usada para vídeo digitais é de 30 fps.

Cada quadro é uma matriz de duas dimensões de unidades chamadas de pixels (de picture elements). Cada pixel representa uma tonalidade de cor e pode ter vários canais, geralmente 3 para vídeos coloridos podendo ser 1 para vídeos mono-cromáticos (preto e branco), 4 caso inclua a transparência, também conhecido como canal alfa, ou diversos para imagens de satélite.

Cada quadro possui um número fixo de pixels na direção horizontal e vertical. O número de pixels em cada direção é chamado de resolução. A resolução é um valor bastante variado também e algumas resoluções possuem nome como QCIF (176x144), VGA (640x480) ou Full HD 1080p (1920x1080).

2.2 Fundamentos sobre compressão de dados

A informação é transmitida através de símbolos de um alfabeto. No caso da escrita o alfabeto pode ser o alfabeto latino e os símbolos são as letras. No caso de uma imagem os pixels são os símbolos e o alfabeto são os valores que um pixel pode assumir (um caso comum é entre 0 e 255). Se justapomos vários símbolos temos uma mensagem. Um exemplo: temos um alfabeto composto pelo símbolos A, B, C e D. Um mensagem possível é AAAAAAAABBBBCCDD. Se quisermos transmitir esta mensagem em meio digital é necessário codificá-la em bits. Uma codificação possível seria A ⇒ 00, B ⇒ 01, C ⇒ 10 e D ⇒ 11. Isso resultaria na mensagem 00.00.00.00.00.00.00.00.01.01.01.01.10.10.11.11.

O princípio fundamental para se comprimir uma mensagem é o de reduzir o número de bits atribuídos aos símbolos cuja probabilidade de ocorrer na mensagem é mais alto. Para a mensagem do exemplo acima uma codificação mais eficaz seria A ⇒ 0, B ⇒ 10, C ⇒ 110 e D ⇒ 111. Note que o símbolo A é o que possue a maior probabilidade logo pode ser codificado usando menos bits. Os símbolos C e D possuem baixa probabilidade logo podem ser codificados com mais bits. A mensagem resultantes para o caso acima seria 0.0.0.0.0.0.0.0.10.10.10.10.110.110.111.111. A mensagem que precisava de 32 bits para ser codificada agora precisa de somente 28 bits.

Um maneira de reduzir ainda mais o número de bits é o de mapear a mensagem original para uma outra codificação. No exemplo usado é possível observar que os símbolos ocorrem muito repetidamente logo podemos representar cada sequência de símbolos ao invés de representar cada símbolo um a um, o que elimina ainda a redundância na mensagem original. A mensagem neste caso pode ser representada de maneira equivalente usando a seqüência A[8]B[4]C[2]D[2] onde o número entre colchetes representa o número de vezes em que o símbolo anterior é repetido. Assim a codificação uma binária possível seria A ⇒ 00, B ⇒ 01, C ⇒ 10 e D ⇒ 11 e o número entre colchetes seria representado por [1] ⇒ 000 [2] ⇒ 001 até [8] ⇒ 111 ([0] não precisa ser codificado). A mensagem resultante seria 00.111.01.011.10.001.11.001. Assim a mensagem pode ser comprimida ainda mais para 20 bits.

O limite teórico do número de bits mínimo para representar uma mensagem é dada pela entropia de cada símbolo. A entropia de um símbolo é dada pela equação:

$$\text{Entropia(simbolo)} = -\log_2(\text{probabilidade(simbolo)})$$

No caso em que a probabilidade do símbolo for zero, a entropia é infinita. Neste caso particular, não necessidade de codificar o símbolo pois ele não deve ocorrer na mensagem.

Note que isso abre também a possibilidade de símbolos serem codificados com um número fracionário de bits. Neste caso existem técnicas diferentes que codificam os símbolos em grupo como a codificação aritmética (SAYOOD 2000), que é a técnica efetivamente usada em HEVC.

2.3 Formatos de pixel

Um vídeo é composto por uma seqüência de imagens e uma imagem é uma matriz bidimensional de pixels. Cada pixel é pequeno quadrado ou retângulo que possue uma cor sólida. Essa cor pode ser representado de diversas maneiras. Uma cor é a combinação linear das cores básicas que o olho humano pode enxergar: vermelho, verde e azul. Uma maneira clássica para codificar as cores de um pixel é o de associar 8 bit para cada cor, o que resulta em 24 bits por pixel. Esta codificação é chamada de RGB24. Foi demonstrado que o olho humano é capaz de perceber melhor a luminância de uma imagem do que as cores da mesma. Assim foi criado uma codificação equivalente ao RGB, mas que separa a informação de luminância da crominância. Uma transformação possível para esse tipo de formato de pixel é a equação seguinte:

```
Y = (R + G + B)/3
U = Y – R
V = Y - B
```

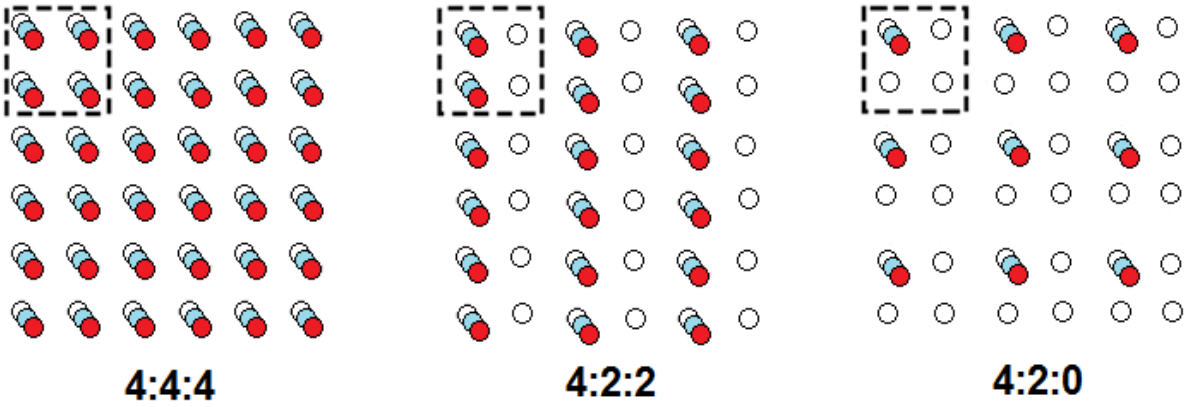
Onde o canal Y corresponde a imagem em tons de cinza (também chamado de luma) e U e V correspondem ao desvio ao vermelho e azul da imagem (chamados de croma). A este tipo de formato de pixel se dá o nome de YUV. Naturalmente esta equação simples precisa de alguns ajustes para que seja representada corretamente em 8 bits sem sinal além de levar em consideração que olho humano também tem melhor sensibilidade no canal verde de cores do que os outros. Uma equação comumente usada é (CONVERTING 2011):

```
Y = (( 66 * R + 129 * G +  25 * B + 128) / 256) +  16
U = ((-38 * R -  74 * G + 112 * B + 128) / 256) + 128
V = ((112 * R -  94 * G -  18 * B + 128) / 256) + 128
```

Assim uma imagem pode ser vista como a sobreposição de três imagens: uma preto e branco e duas com as cores. Conforme já demonstrado o olho humano percebe melhor a luminância do que a crominância, logo pode ser descartado uma parte das informações de cores com uma perda mínima de qualidade para o observador. Assim, é possível guardar somente um em cada dois pixels de crominância na direção vertical e um em cada dois na

direção horizontal. A estes tipo de codificação é dado o nome de YUV 4:2:2 ou YUV 4:2:0. No caso do YUV 4:2:2 metade dos pixels de crominância na direção horizontal são omitidos e no YUV 4:2:0 os pixels na direção vertical também são omitidos. O formato onde todos os pixels são representado se chama de YUV 4:4:4. A figura 2.1 ilustra como funciona esta técnica:

Figura 2.1 - Ilustração dos tipos de sub-amostragem no formato YUV



Fonte: (AFONSO 2012)

Estas técnicas permitem economizar 33% no número de bits no caso do YUV 4:2:2 e 50% no caso do YUV 4:2:0. Não confundir com a compressão de dados pois neste caso estamos perdendo um pouco de informação da imagem original, embora tal perda seja imperceptível a olho nu. Quando a compressão ocorre devido a uma modificação na imagem original é dado o nome de compressão com perdas.

2.4 Transformada e quantização

Uma maneira de reduzir ainda mais o número de bits que uma imagem ocupa é dividindo a imagem em blocos e aplicar uma função sobre todos os pixels do bloco que reduza ainda mais a redundância entre os símbolos. A divisão em blocos, dependendo do codec, é feita usando blocos de tamanho 4x4 pixels até 32x32, sendo sempre um quadrado com o lado medindo uma potência de 2. Essa divisão facilita o cálculo da função de transformada posteriormente.

Sobre cada um desse blocos separadamente é aplicada uma função chamada transformada. Existem várias funções de transformada que são usadas para a compressão de imagens. Uma transformada comum que pode ser usada é a transformada de cossenos discreta, chamada também de DCT. Esta transformada é baseada na transformada de Fourier e

é muito usada na sua forma original ou em variações baseadas que levam em conta a redução da complexidade da implementação e uma melhor compressão.

Tomamos como exemplo o bloco 8x8 abaixo (somente os pixels do canal Y):

```
 83  98 107  95  90 100  98 100
 94 108 103  91  87  91  88 105
 99  99  89  93  93  90  85  93
 96 102  95  98  94  90  92  84
 96  97  91  94  91  88  90  83
 89  86  81  92 100 100  90  86
 96  99  90  93 100  96  77  87
 91  90  87  93  84  82  87  96
```

A DCT aplicada diretamente resulta no bloco:

```
741.50  13.25  -1.23  -3.04   0.00 -9.82 -1.66 -7.58
 17.23  -1.08   5.64 -13.96 -12.84 -1.82  3.12  1.12
  2.71  -9.52   5.43 -14.38  -2.93 -4.04  8.08 -1.45
  2.96 -10.07 -10.87   6.27  -7.23  0.19  3.21  1.68
 -1.00  -0.95  -0.14  -8.52  -6.50 11.03 -4.27  2.45
  4.64   2.33 -10.55   6.60  -1.09 -2.54  0.92 -0.02
 -6.53 -12.40  -2.42   5.86   3.38  5.98 -0.93  0.27
 -1.89   6.02   0.97   1.27   2.43 -1.46  1.22 -0.14
```

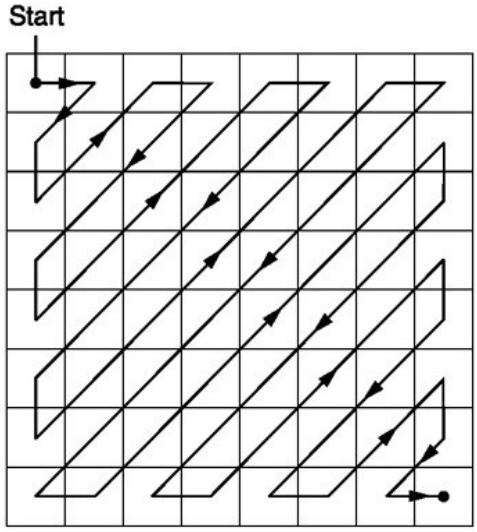
É possível observar que os coeficientes mais altos e a esquerda na matriz da transformada possuem valores maiores do que o pixels mais baixos e a direita. É possível observar também que a transformada resulta em valores em ponto flutuante, que precisam de mais bits para ser codificados do que o bloco original. Um passo importante para aumentar a compressão do bloco é a quantização. É a partir dela é que temos uma redução significativa do número de bits codificados. Por outro lado quantização causa uma perda na qualidade visual que pode ser muito significativa. Usando o mesmo exemplo, a quantização do bloco é efetuada dividindo todos os coeficientes por 8, arredondando os resultados para o inteiro mais próximo:

93	2	0	0	0	0	0	0
2	0	1	-1	-1	0	0	0
0	0	1	-1	0	0	1	0
0	0	0	1	0	0	0	0
0	0	0	0	0	1	0	0
1	0	0	1	0	0	0	0
0	-1	0	1	0	1	0	0
0	1	0	0	0	0	0	0

O arredondamento tem de ser feito para o inteiro mais próximo. Um erro comum é fazer o arredondamento para o zero ou para o menos infinito. Isso pode causar um ruído que dá a aparência que o pixel mais alto a esquerda do bloco é mais claro ou escuro do que o restante. É possível observar que o bloco no exemplo possui uma grande quantidade de coeficientes de valores baixos, ou mesmo zero, espalhados em uma grande área.

Com o intuito de maximizar o número de zeros em seqüência e concentrar os coeficientes de maior valor no início da seqüência, os coeficientes do bloco são percorridos em zigzag. A figura 2.2 mostra um exemplo do padrão zigzag em um bloco 8x8:

Figura 2.2 - Ilustração do padrão zigzag



Fonte: (LIU 2014)

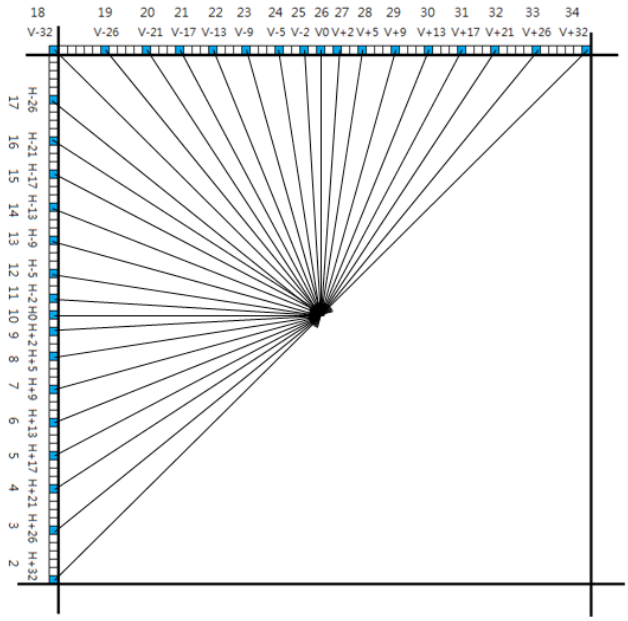
Observe que se os coeficientes quantizados do exemplo anterior forem percorridos em zig-zag é possível obter longas seqüência de zeros que podem ser comprimidos mais eficientemente.

Os codec mais modernos, como o HEVC, usam variações da transformada que têm como objetivo diminuir a complexidade do cálculo assim como obter melhor compressão.

2.5 Predição Intra

A predição intra é uma técnica que permite prever um bloco a partir de um bloco adjacente. Nesta técnica os pixels adjacentes ao bloco que está sendo decodificado são usados como referência para prever os pixels deste bloco. Isto permite que imagens onde o conteúdo é relativamente contínuo e repetitivo possuam taxas de compressão ainda maiores. A figura 2.3 ilustra como esta técnica é usada no HEVC:

Figura 2.3 - Modos de predição intra como no HEVC



Note que aplicando esta técnica é possível se valer da correlação entre os pixels em um bloco com os seus vizinhos, em especial quando o bloco for parte de uma superfície com uma textura repetitiva ou no limite de um objeto com um lado reto.

2.6 Estimativa de movimento

Até agora foram discutidas técnicas de compressão que se aplicam a imagens estáticas. Um vídeo por ser visto como uma seqüência de imagens estáticas, também chamados de quadros. Observando os quadros de um vídeo em sequência é possível ver que existe uma correlação entre eles, exceto em cortes de cena. A figura 2.4 mostra dois quadros consecutivos de um vídeo e ilustra como eles são visualmente semelhantes:

Figura 2.4 - Dois quadros consecutivos em uma seqüência

Fonte: Rede Globo

Nesta figura é possível notar que o plano de fundo dos quadros são relativamente semelhantes embora o rosto da apresentadora possua algumas diferenças como a boca, pois ela está falando e a posição do corpo pois ela está se mexendo levemente a medida que fala. Assim é possível codificar somente a diferença entre estes quadros, isto é, a diferença do valor entre cada pixel na mesma posição em cada quadro.

Simplesmente codificar a diferença entre os quadros já contribui com ganhos expressivos, mas ainda não é a solução ótima pois o objetos se movem na cena. Quando isso ocorre, essa diferença acaba sendo muito expressiva gerando um ruído indesejável na imagem que resulta em uma compressão abaixo da desejada.

Para diminuir os efeitos de objetos que se movem na cena existe a chamada predição de movimento. De maneira similar a transformada, a imagem é dividida em blocos. Estes blocos podem ser quadrados (como na transformada) ou retangulares. O codificador busca no quadro imediatamente anterior ao atual um bloco do mesmo tamanho que reduza o máximo possível o custo em bits para codificar a diferença entre os blocos. O cálculo do deste custo, chamado custo de movimento, normalmente é feito calculando a soma do valor absoluto das diferenças entre cada pixel dos blocos, as vezes também calculando o número de bits necessário para codificar o vetor de movimento. O codec codifica então a diferença entre estes dois blocos (chamado de resíduo) usando a transformada mais o vetor de movimento, que servirá para que o decodificador encontre o bloco anterior novamente para efetuar a reconstrução do bloco atual (AFONSO 2012).

Os objetos não necessariamente se movem na cena por um número fixo de pixels. Assim o bloco no quadro anterior que corresponde ao de menor diferença em relação ao atual

pode estar não em um bloco exatamente igual a um no quadro anterior, mas sim em um bloco cujos pixels estão entre os pixels do bloco anterior. Estes pixels no bloco anterior devem ser calculados usando um filtro de interpolação e o bloco resultante dessa predição é usado para o cálculo do bloco atual. A etapa de estimativa de movimento corresponde a uma grande parte da complexidade computacional de um codec (BOSSEN 2012). Em alguns casos, somente esta etapa corresponde a mais de 50% do tempo de processamento do codificador. É principalmente por causa disso que um codificador de vídeo é muito mais complexo e demanda muito mais poder computacional do que um decodificador.

3 CODIFICAÇÃO DE VÍDEO

Existe um grande número de aplicações nos dias de hoje que são baseados na transmissão ou armazenamento de video digital como a TV aberta e paga, video-chat, tele-medicina, video-monitoramento, cinema, discos Blu-ray entre vários outros. Porém, video digital exige uma grande capacidade de banda passante para ser transmitida ou de espaço de armazenamento caso for necessário armazenar um video com alta definição (o que nos padrões atuais isso é a resolução de 1920x1080 ou superior como 3840x2160 ou até mesmo 7680x4320), para maximizar a experiência de quem estiver assistindo a seqüência de imagens gravadas ou transmitidas (SULLIVAN 2012).

Armazenar ou transmitir vídeo descomprimido é muito caro para os padrões atuais. Tomando como exemplo uma câmera capturando uma sequência de vídeo de resolução de 1920 pixels de largura e 1080 pixels de altura, 24 bits para cada pixel e com 30 quadros por segundo - o que é um padrão médio para os dias atuais - nós temos aí cerca de 178 Megabytes sendo gerados a cada segundo. Se consideramos que esta seqüência seja gravada ao longo de 2 horas, temos então 1,22 Terabyte para gravar toda esta seqüência. Para reduzir o volume de dados que um vídeo digital precisa para ser guardado ou transmitido é necessário empregar tecnologias de compressão de vídeo conforme apresentado no capítulo 2.

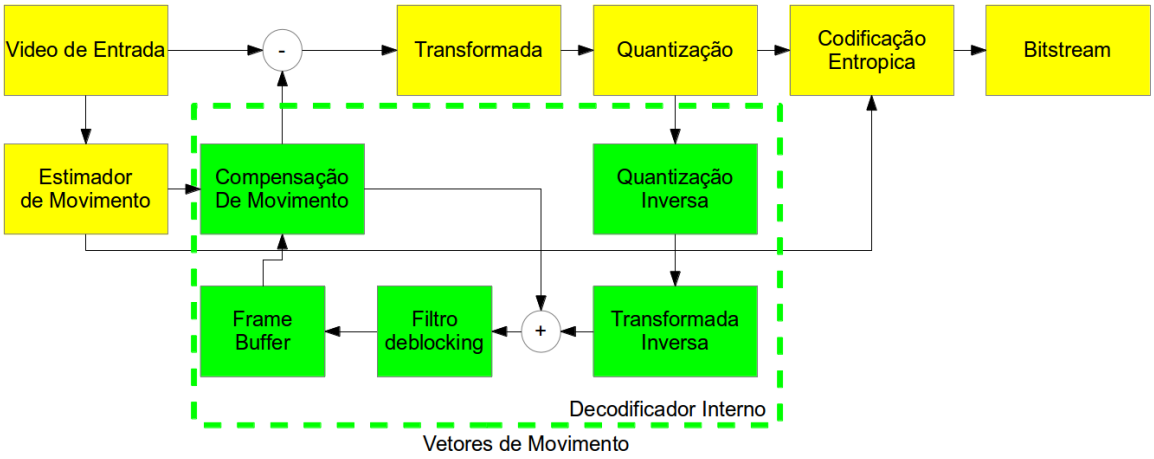
Assim normalmente é usado um codec que execute algorítimos de compressão e descompressão. O algoritmo que realiza a descompressão do vídeo comprimido deve reverter as operações que foram executadas pelo algoritmo que realizou a compressão do vídeo para obter a seqüência original. Como já citado antes, nem sempre é possível obter exatamente a mesma seqüência original, mas sim uma seqüência visualmente semelhante com um pouco de ruído para melhorar a compressão. Como nem sempre o vídeo é comprimido pelo mesmo dispositivo que descomprime ele, como no caso em que um vídeo é gravado por uma câmera portátil e exibido em um laptop, por exemplo, é necessário que todos os dispositivos usem um método padronizado para comprimir e descomprimir os vídeos.

O algoritmo de descompressão normalmente é padronizado por uma especificação ou por um software de referência, o que garante que um vídeo produzido por uma implementação possa ser visualizado em qualquer outra implementação do mesmo codec. A um padrão de codificação/decodificação de vídeo é dado o nome de formato. Normalmente só o decodificador é padronizado, cabendo ao desenvolvedor do codificador a tarefa de encontrar um meio de codificar um vídeo de modo que a qualidade visual seja aceitável quando ele é decodificado por um decodificador conforme. Esta padronização permite que um video

gravado por um dispositivo que implemente um codificador seguindo um padrão possa ser decodificado por um outro dispositivo que implemente o decodificador do mesmo formato.

Existem diversos codecs de vídeo. Em geral os codecs da família MPEG, o qual o HEVC faz parte, e também vários outros, seguem uma estrutura básica como a mostrada na figura 3.1.

Figura 3.1 - Diagrama de blocos de um codificador de vídeo



Fonte: Desenho próprio

Conforme já explicado no capítulo anterior, o bloco de transformada aplica uma função que transforma os pixels para o domínio da freqüência, que serve para facilitar a compressão da imagem. O bloco de quantização diminui a quantidade de bits necessários para codificar cada coeficiente da transformada, o que causa como conseqüência uma perda na qualidade da imagem. O bloco de codificação entrópica codifica cada um dos coeficientes quantizados, vetores de movimento e outros símbolos usando o menor número de bits possível, baseado nas estatísticas da imagem. Internamente um codificador também decodifica as imagens após o processo de quantização para que as imagens resultantes sirvam de referência para o bloco de estimativa de movimento. O bloco de estimativa de movimento busca qual é o bloco do quadro anterior (se houver) que mais se assemelha ao bloco do quadro atual para que seja codificado somente a diferença entre estes blocos, o que resulta em uma imagem que precisa de muito menos bits para codificar.

3.1 Qualidades de um codec

Um codec de vídeo pode ser avaliado como sendo de qualidade em diversos aspectos entre as quais podemos citar:

1. Volume de dados resultante

2. Qualidade visual do vídeo resultante
3. Atraso entre a entrada do vídeo para ser comprimido e a saída do vídeo descomprimido
4. Resistência a erros na transmissão ou armazenamentos
5. Desempenho do codificador e do decodificador que tem os seguintes sub-parâmetros:
    a. Tempo de execução
    b. Memória usada
    c. Área de circuito necessária (no caso de uma implementação em hardware)
    d. Consumo de energia elétrica

Tais qualidades podem ser dependentes tanto do formato usado quanto da implementação do mesmo. Além disso alguns destes parâmetros podem ser divergentes com outros, como o volume de dados e a qualidade visual. Neste caso é necessário encontrar o melhor compromisso entre todos os parâmetros para se ter um ótimo equilíbrio. É importante salientar que a melhora na qualidade no áudio induz a pessoa que está assistindo à seqüência a pensar que o vídeo possui uma qualidade superior, logo a qualidade e o volume de bits do áudio que acompanhar o vídeo, caso aplicável, deve também ser levada em conta (BELMUDEZ 2009).

3.2. Formatos anteriores ao HEVC

Existe uma grande variedade de formatos de codificação de vídeo que foram introduzidos antes do HEVC. Estes formatos respondem por diversos nichos de mercado. Dentre eles podemos citar o MPEG2, o Theora, o VP8 e H.264/AVC.

O MPEG2 é um antigo formato desenvolvido pelo consórcio MPEG e que foi usado nos primeiros sistemas de TV digital, aparelhos de DVD e foi um dos primeiros codecs populares na internet. O sucesso do MPEG2 resultou no subseqüente desenvolvimento de novos padrões com qualidade de imagem e compressão superiores pelo consórcio MPEG como o MPEG4, o H.264/AVC e o HEVC.

Os formatos padronizados pelo consórcio MPEG atraíram fortes críticas por terem o uso restrito por patentes que demandam o pagamento de royalties, o que dificulta a implementação de codecs por parte de projetos de software livre, em especial navegadores de internet (como o Mozilla Firefox e o Google Chrome).

O Theora e o VP8 são codecs lançados ao público livre de royalties. Estes codecs tem como objetivo dar a comunidade de software livre uma alternativa aos codecs pagos.

O H.264/AVC é provavelmente o codec mais popular em uso atualmente com várias implementações em hardware e software e uma penetração de mercado muito próxima de 100%. O H.264/AVC também é um dos melhores formatos no mercado em termos de qualidade visual e volume de dados resultante, embora o tempo de processamento e memória usada seja bastante alto, compensado pelo fato de que existem várias implementações em hardware e a existência de perfis de baixo nível que sacrificam a qualidade visual e de compressão por um processamento mais eficiente.

3.3. HEVC

O H.264/AVC sendo extremamente popular passou a mostrar limitações para a demandas mais atuais como o vídeo em resolução 4k e a crescente popularidade por serviços de streamming de vídeo e vídeo sob demanda os quais são responsáveis por uma grande parte do tráfego nas redes atualmente (SULLIVAN 2012). Estes novas aplicações criaram uma forte demanda por um codec com uma taxa de compressão ainda maior com uma maior qualidade visual. Motivado por esta demanda foi desenvolvido o padrão HEVC.

O HEVC é um formato de compressão de vídeo lançado em 2013. O HEVC é um padrão que apresenta uma economia média de 40% o número de bits comparado a um vídeo de qualidade equivalente codificado usando o padrão H.264/AVC (OHM 2013).

3.3.1 Estrutura de particionamento da imagem

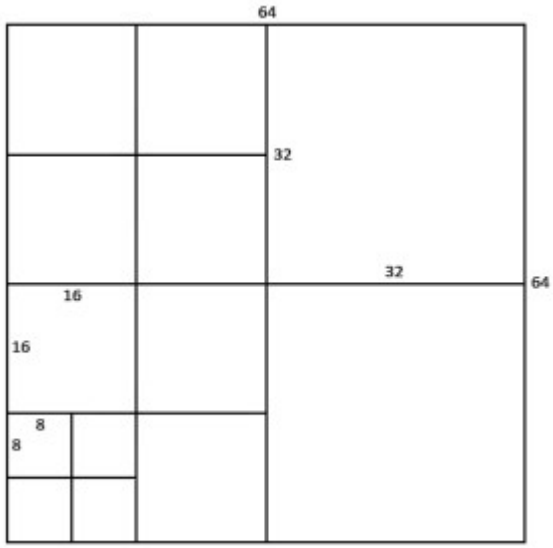
O HEVC possui quatro conceitos distintos de unidades: Unidade de árvore de codificação (CTU), Unidade de codificação (CU), Unidade de predição (PU) e Unidade de transformada (TU). A estes conceitos estão associados respectivamente o Bloco de árvore de codificação (CTB), Bloco de codificação (CB), Bloco de predição (PB) e Bloco de transformada (TB). Cada bloco corresponde a uma matriz 2-D de pixels de um canal de cores da sua respectiva unidade. Isto é, uma unidade de transformada, por exemplo, contém três blocos de transformada, um luma e dois croma, mais a sinalização associada (KIM 2012).

Um quadro é composto por slices, que são fatias do quadro. Estas fatias são independentes entre si logo se uma for corrompida isso não afeta as outras. Isso permite também que eles sejam codificados em paralelo. Por outro lado dividir o quadro em slices aumente o número de bits necessário para codificá-lo. Quando o codificador não precisa agregar estas qualidades ao vídeo, ele pode optar por usar só um slice para o quadro inteiro.

Um slice possue vários CTUs em ordem raster, começando do alto da imagem para baixo, da esquerda para direita. Estas CTUs podem ter o tamanho de 8x8, 16x16, 32x32 ou 64x64, conforme especificado na sinalização no início do quadro. Uma CTU é análoga ao conceito de macro-bloco que existia no H.264/AVC, embora no H.264/AVC eles fossem restritos ao tamanho 16x16.

Cada CTU possui vários CUs. O particionamento de uma CTU em CUs é feita através de uma árvore quadrática. Uma CTU corresponde a uma ou quatro CUs e cada CU pode ser subdividida em 4 CUs até o limite mínimo de 8x8 para cada CU, conforme ilustrado pela figura 3.2.

Figura 3.2: Ilustração da divisão de uma CTU em CUs



Fonte: (KIM 2012)

Dentro de cada CU existe separadamente as PUs e as TU. Uma PU é uma unidade que contém as informações básicas de predição. Uma PU pode ser subdividida também, em várias formas quadradas e retangulares mas só uma vez. Uma TU é uma unidade que contém o resíduo da predição que recebe a transformada ou a transformada do bloco inteiro caso for um bloco do tipo intra. Uma TU é uma subdivisão de uma CU também e pode ser subdividida recursivamente somente em subdivisões quadradas. Por isso, as margens de uma TU podem ficar sobrepostas as margens de uma PU (KIM 2012).

3.3.2 Transformada e quantização

Os TBs podem ter o tamanho de 4x4, 8x8, 16x16 ou 32x32 e são codificados usando uma transformada baseada na DCT. No caso especifico de um TB 4x4 luma também existe

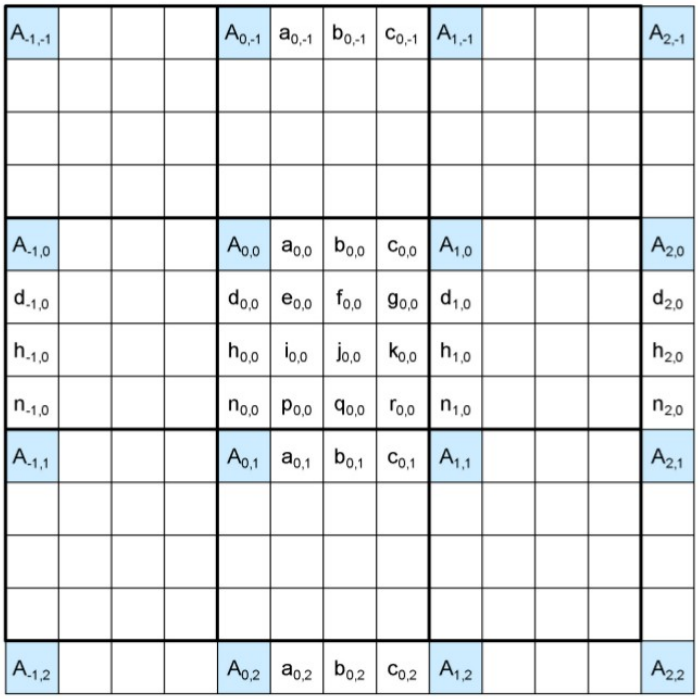
uma transformada baseada na DST (Transformada discreta de senos). Os coeficientes são em seguida quantizados para permitir uma maior compressão.

3.3.3 Estimativa de movimento

Conforme já citado, no HEVC exista a predição de movimento usando PUs de formato quadrado ou retangular. Tal técnica permite uma compressão ainda maior pois uma imagem possui uma grande redundância entre os quadros temporalmente justapostos.

No HEVC também existe a predição de movimento fracionário com precisão de um quarto. A figura 3.3 mostra como são localizados os pixels em posição fracionária.

Figura 3.3 - Localização dos pixels em posição fracionária



Fonte: (AFONSO 2012)

Na figura os quadrados sombreados são os pixels em posição inteira tal como eles existem na imagem original. Os quadrados em branco são os pixels em posição fracionária. Como tais pixels não existem na imagem original, eles precisam ser calculados usando um filtro de interpolação. No HEVC são usados um filtro de 8-tap para os pixels em posição de b, h e j e 7-tap para os restantes. O número de taps do filtro significa quantos pixels de posição inteira são necessários para calcular o pixel em posição fracionária. A tabela 3.1 mostra os coeficientes de cada filtro:

Tabela 3.1 - Coeficientes dos filtros de interpolação em HEVC

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

As equações que implementam cada filtro são as seguintes:

$$a_{0,j} = (\Sigma_{i=-3..3} A_{i,j} qfilter[i]) >> (B-8)$$
$$b_{0,j} = (\Sigma_{i=-3..4} A_{i,j} hfilter[i]) >> (B-8)$$
$$c_{0,j} = (\Sigma_{i=-2..4} A_{i,j} qfilter[1-i]) >> (B-8)$$
$$d_{0,j} = (\Sigma_{j=-3..3} A_{0,j} qfilter[j]) >> (B-8)$$
$$h_{0,0} = (\Sigma_{j=-3..4} A_{0,j} hfilter[j]) >> (B-8)$$
$$n_{0,0} = (\Sigma_{j=-2..4} A_{0,j} qfilter[1-j]) >> (B-8)$$

Na equação B é o número de bits para cada pixel da imagem original, geralmente 8, podendo ser também 10 ou 12 em perfis mais avançados. Após o calculo destas posições iniciais são calculadas as posições finais a partir das primeiras:

$$e_{0,0} = (\Sigma_{v=-3..3} a_{0,v} qfilter[v]) >> 6$$
$$f_{0,0} = (\Sigma_{v=-3..3} b_{0,v} qfilter[v]) >> 6$$
$$g_{0,0} = (\Sigma_{v=-3..3} c_{0,v} qfilter[v]) >> 6$$
$$i_{0,0} = (\Sigma_{v=-3..4} a_{0,v} hfilter[v]) >> 6$$
$$j_{0,0} = (\Sigma_{v=-3..4} b_{0,v} hfilter[v]) >> 6$$
$$k_{0,0} = (\Sigma_{v=-3..4} c_{0,v} hfilter[v]) >> 6$$
$$p_{0,0} = (\Sigma_{v=-2..4} a_{0,v} qfilter[1-v]) >> 6$$
$$q_{0,0} = (\Sigma_{v=-2..4} b_{0,v} qfilter[1-v]) >> 6$$
$$r_{0,0} = (\Sigma_{v=-2..4} c_{0,v} qfilter[1-v]) >> 6$$

3.3.4 Estado da arte

Existem várias abordagens que tentam maximizar o desempenho do algoritmo de busca dos vetores de movimento de precisão sub-pixel. Foram estudados alguns trabalhos publicados para se ter uma idéia do estado da arte. Alguns destes trabalhos foram feitos baseados no padrão H.264/AVC, mas, devido a similaridade destes codecs, as propostas também se aplicam ao HEVC.

O método proposto por (SUH 2004) e aperfeiçoado por (KAO 2006) tenta estimar o vetor sub-pixel a partir dos oito vetores de posição inteira na vizinhança. Este método não precisa do cálculo da interpolação do bloco, o que resulta em uma grande economia em acessos a memória, embora ele requisite o cálculo custo de movimento para os vetores em posição inteira. Alguns métodos de busca rápida como o EPZS que não calculam diretamente todos os vetores de movimento na posição inteira (TOURAPIS 2002), logo não é compatível com este modelo.

O modelo proposto por (HE 2013) simplifica a etapa de busca de vetores de precisão de um quarto usando um filtro bilinear para calcular estes sub-pixels. Além disso, devido a linearidade da operação de interpolação bilinear ele aproveita o fato de que é possível calcular o SAD dos blocos de posição de um quarto de pixel diretamente a partir dos blocos em posição meio pixel.

No trabalho de (DAI 2012) só são calculados os vetores nas direções retas a partir do bloco atual (o que corresponderia aos pixels a, b, c, d, h e n na figura 3.3). Isto simplifica a complexidade do algoritmo pois os sub-pixels nas posições na diagonal requerem cálculos mais complexos. Após esta etapa é feita uma estimativa de quais candidatos são os melhores e é calculado o custo de movimento somente para os melhores candidatos.

Existe ainda o trabalho de (AFONSO 2012) que propõe uma arquitetura em dois estágios, primeiro buscando o bloco com o menor SAD nos oito vizinhos deslocados em meio pixel e depois refinando a busca sobre os oito blocos vizinhos deslocados em um quarto de pixel. Esta é a mesma abordagem usada pelo software de referência, mas com a inovação de que foi implementada em FPGA.

4 IMPLEMENTAÇÃO

A primeira implementação de qualquer codec normalmente é feita em software. Geralmente estas implementações são feitas para poder ter um demonstrador da tecnologia rapidamente e para servir de referência para futuras implementações. Estas implementações habitualmente são muito lentas para o uso prático, embora existam formas de se melhorar substancialmente o desempenho em CPU, como o uso de instruções SIMD e o paralelismo multi-núcleo das arquiteturas mais modernas. Mesmo o H.264/AVC, que originalmente era um codec de alta exigência computacional, atualmente tem uma implementação que pode ser executado em tempo real em uma CPU moderna com pouco prejuízo para as outras qualidades do vídeo (nível de compressão, qualidade visual, etc.).

Uma maneira de se programar um codec de alto desempenho usando apenas materiais facilmente encontradas no mercado é usar a GPU do computador para executar tarefas de maior carga computacional. As GPUs são unidades de processamento gráfico e podem ser encontradas em qualquer computador pessoal. Inicialmente elas usadas para renderizar imagens em 3D, em especial para entusiastas de jogos. As GPUs mais modernas são cada vez mais programáveis e podem ser configuradas para executar tarefas que precisam de alto poder computacional e que podem ser paralelizadas, como a quebra de chaves de encriptação, análises estatísticas ou computação gráfica. Neste caso, a GPU é uma excelente ferramenta para ser programada para codificar video em alta definição pois grande parte do trabalho de codificação de video é repetitivo e paralelizável.

Usar GPUs para criar codificadores de alto desempenho é favorável quando se deseja criar um codec de alto desempenho relativamente rápido e com baixo custo sem que o custo de aquisição de equipamentos nem o tamanho ou consumo de potência seja um problema. Quando é necessário integrar um codificador em um sistema embarcado o método mais comum empregado atualmente é o de desenvolver uma ASIC (Application Specific Integrated Circuit). Desenvolver uma ASIC custa muito mais caro e toma muito mais tempo para

desenvolver do que um software para GPU mas o custo unitário e consumo de energia são muito mais baixos.

Quando tanto o consumo de potência e espaço ocupado por uma GPU quanto o custo de desenvolvimento de uma ASIC são impraticáveis, a solução intermediaria é implementação usando FPGAs. Uma FPGA é um chip que possui portas lógicas programáveis. Uma FPGA pode se comportar de maneira muito similar a uma ASIC, exceto pelo fato de que o custo unitário de uma FPGA é superior a uma ASIC em grande escala e o desempenho de uma FPGA normalmente é inferior além da dificuldade de se comprar FPGAs no mercado.

4.1 Método usado

Para este trabalho, a tecnologia usada será a implementação em FPGA. Inicialmente a implementação será feita totalmente em computador usando simuladores. Posteriormente será feito a sintetização e, de acordo com as disponibilidades de material, o teste em uma FPGA.

O código para este trabalho pode ser encontrado digitando em um terminal Linux onde a ferramenta git estiver instalada o seguinte comando:

```
$ git clone https://github.com/gabrieldiego/tg.git
```

4.2 Software de referência

O software de referência é a maior fonte de referência sobre o HEVC a ser usada na implementação deste trabalho. O código fonte é livremente acessível a partir do repositório svn https://hevc.hhi.fraunhofer.de/svn/svn_HEVCSoftware/. Isso permite que seja facilmente feita a análise do código fonte.

4.3 Análise do software de referência

4.3.1 Download do código fonte do software de referência e compilação

O primeiro passo neste trabalho foi o de baixar o código do software de referência usando do repositório svn usando o seguinte comando:

```
$ svn co https://hevc.hhi.fraunhofer.de/svn/svn_HEVCSoftware/trunk
```

A versão usado ao longo do trabalho é a versão 13.0 que corresponde a revisão 3800 publicada no dia 22 de Janeiro de 2014 neste repositório.

O passo seguinte foi compilar o código. A máquina usada tinha instalado o Ubuntu 12.04, assim foi usado o script para compilar no sistema Linux. Isso pode ser feito acessando o diretório build/linux e executando o make com os comandos:

```
$ cd build/linux
```

```
$ make -j4
```

O parâmetro -j4 permite a execução de 4 processos de compilação ao mesmo tempo, o que traz ganhos no tempo de compilação em uma máquina com 4 núcleos de processamento, o que era o caso pois a máquina usada possui um processador Intel Core i5-3427U, que possui 2 núcleos físicos, cada um funcionando como 2 núcleos lógicos (tecnologia chamada de Hyper-Threading pela Intel).

Os binários resultantes podem ser encontrados no diretório bin. No caso do codificador o executável é o TAppEncoderStaticd. O Static no nome do arquivo diz que os binários estão compilados com linkagem estática e o d no final significa que os binários contêm símbolos para debugagem.

Quando este binário é executado sem nenhum parâmetro de entrada ele imprime uma lista com os parâmetros que podem ser usados. Como forma de facilitar o uso do software, na pasta cfg existem vários arquivos com uma lista de configurações pré-redigidas para alguns casos típicos de codificação. Para usar um destes arquivos basta executar o codificador usando o seguinte comando:

```
$ ./TAppEncoderStaticd -c ../cfg/encoder_lowdelay_P_main.cfg
```

Substitua o arquivo encoder_lowdelay_P_main.cfg pelo que for mais adequado. Note que este arquivo não possui informações sobre o arquivo de vídeo de entrada. Para isso é necessário passar as configurações do arquivo de vídeo de entrada que deve estar no formato yuv. Algumas seqüências tipicamente usadas em HEVC já possuem um arquivo de configuração pronto na pasta cfg/per-sequence. Estes arquivos podem ser usados diretamente usando o seguinte comando como exemplo:

```
$ ./TAppEncoderStaticd -c ../cfg/encoder_lowdelay_P_main.cfg
-c ../cfg/per-sequence/ChinaSpeed.cfg
```

Caso for necessário usar parâmetros diferentes também é possível enviar cada parâmetro pela linha de comando diretamente (o que pode resultar em uma linha de comando muito longa), modificar um dos arquivos existentes ou inserir a opção após a inclusão do arquivo de configuração, o que anula a opção anterior em favor da inserida depois.

4.3.2 Estrutura do código fonte

Uma vez que o código foi compilado e o binário testado, foi feita a análise do código fonte. O código fonte fica na pasta source. Dentro desta pasta existem duas sub-pastas: App e Lib. A pasta App contém o código fonte da aplicação que executa o codificador ou o decodificador e a pasta Lib possui o código fonte da implementação do codec (que é o objeto

de interesse, neste caso). Dentro da pasta Lib existem 6 pastas, sendo as de maior interesse são as pastas TLibDecoder, TLibEncoder e TLibCommon. A pasta TLibDecoder possui o código específico do decodificador, a pasta TLibEncoder possui código específico do codificador e a pasta TLibCommon possui todo o código que é usado tanto pelo codificador como pelo decodificador. Conforme mostrado na figura 3 um codificador obrigatoriamente possui partes um decodificador integrado, logo seria problemático ter duas implementações de um decodificador no mesmo projeto, por isso o código comum ao codificador e o decodificador é colocado separadamente em uma só pasta. É importante notar também que somente as pastas TLibDecoder e TLibCommon possuem código especificados pela norma pois implementam o decodificador e a pasta TLibEncoder possui um exemplo apenas para facilitar o trabalho de implementação de um codificador.

O código foi escrito inteiramente em C++, o que significa que o código é escrito usando orientação a objetos. Isso facilita o estudo do código fonte pois permite o uso de ferramentas facilmente encontradas na rede, notadamente o gcc. Usando o gcc como compilador, é possível ter acesso a diversas outras ferramentas muito úteis para o desenvolvimento como o gdb para debugagem e o gprof para a análise de desempenho.

4.3.3 Classes e métodos do módulo de pesquisa de vetores de movimento

Nesta seção será apresentado uma breve descrição das classes e métodos do módulo de pesquisa de vetores de movimento e como eles se comportam durante a pesquisa de vetores de movimento com precisão sub-pixel. Embora o trabalho esteja limitado à parte de pesquisa de vetores sub-pixel, esta parte está integrada à pesquisa de vetores de movimento. As classes correspondentes a outros módulos não serão descritas devido ao tamanho do projeto (cerca de 60 mil linhas em C++, calculado usando o utilitário wc) e à baixa relevância destes módulos para este trabalho.

O método TEncSearch::predInterSearch é o que chama os métodos necessários para realizar a predição inter-quadros incluindo o método TEncSearch::xMotionEstimation, que chama os métodos que calculam o menor custo entre os 8 vizinhos imediatos (acima, abaixo, a direita, a esquerda e nas diagonais). O mesmo processo é repetido para refinar o vetor de movimento em 1/4 de pixel usando o método TEncSearch::xExtDIFUpSamplingQ. Os métodos TEncSearch::xExtDIFUpm a busca inter-quadros bi-direcional ou fracionária. Neste caso estamos interessados na busca fracionária que é decidida pela chamada ao método TEncSearch::xPatternSearchFracDIF. Este método por sua vez chama o métodos TEncSearch::xExtDIFUpSamplingH e TEncSearch::xExtDIFUpSamplingQ que aplicam os

filtros que geram um bloco com precisão de 1/2 pixel e 1/4 pixel, respectivamente, e em seguida chama o método TEncSearch::xPatternRefinement que busca o vetor de movimento com o menor custo de movimento. Os filtros são implementados pelos métodos TComInterpolationFilter::filterVerLuma e TComInterpolationFilter::filterHorLuma (implementados usando o template TComInterpolationFilter::filter). É importante notar aqui que a busca dos vetores de movimento é feita exclusivamente usando somente os blocos do canal Y (também chamado de luma), os blocos do canal U e V (chamados de chroma) usam o mesmo vetor de movimento que o do canal Y.

4.4 Implementação em software

Antes de proceder à implementação em hardware, é importante implementar em software a arquitetura que será feita em hardware. Isso permite a validação dos resultados e fácil verificação da implementação do RTL.

A implementação em software foi feita diretamente no código fonte do software de referência. Isso permite que sejam feitos testes usando amostras reais.

No software de referência a busca por vetores de movimento com precisão sub-pixel é realizada em duas etapas. Primeiro é calculado ao redor do bloco encontrado previamente pelo algoritmo de busca de vetores de posição inteira os oito blocos deslocados meio pixel em cada direção (acima, abaixo, esquerda, direita e as diagonais) e em seguida o custo de movimento (SAD ou Hadamard) é calculado para cada candidato mais o próprio bloco e o que corresponde com o menor custo é selecionado. Por último o mesmo processo é repetido usando deslocamentos de 1/4 de pixel.

A abordagem usada pelo software de referência é simples, mas impões custos altos à implementação de hardware, em especial para o cálculo de cada bloco, já que ele usa os mesmo filtros usados para a decodificação de 7 ou 8 taps. Isso implica no preenchimento do bloco em 4 pixels em cada direção pois o filtro precisa de pixels que não estão presentes no bloco. Assim para calcular um bloco 8x8, é necessário enviar para o filtro um bloco 16x16.

A implementação proposta simplifica alguns pontos. O algoritmo se limita a calcular somente blocos de tamanho 8x8 embora a especificação permita PUs de tamanho 4x4 até 32x32 mais vários formatos retangulares.

Ao invés de usar filtros de 7 e 8 taps, será usado uma média aritmética simples para o cálculo dos pixels de deslocamento de meio pixel e média ponderada para os de deslocamento de um quarto. As equações abaixo correspondem às da implementação feita (usando os pixels da figura 6 como referência):

$$a_{0,j} = 3 * A0,j + A1,j + 2>>2$$
$$b0,j = A0,j + A1,j + 1>>1$$
$$c_{0,j} = A0,j + 3 * A1,j + 2>>2$$
$$di,0 = 3 * Ai,0 + Ai,1 + 2>>2$$
$$hi,0 = Ai,0 + Ai,1 + 1>>1$$
$$n_{i,0} = Ai,0 + 3 * Ai,1 + 2>>2$$

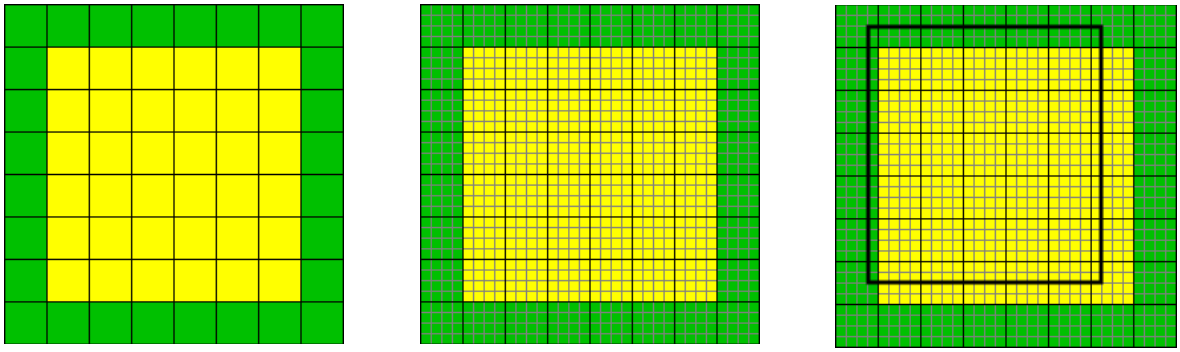
Os pixels seguintes dependem do cálculo de pixels anteriores:

$$e_{0,0} = 3 * a_{0,0} + a_{0,1} + 2>>2$$
$$f_{0,0} = 3 * b0,0 + b0,1 + 2>>2$$
$$g_{0,0} = 3 * c_{0,0} + c_{0,1} + 2>>2$$
$$i0,0 = a_{0,0} + a_{0,1} + 1>>1$$
$$j_{0,0} = b0,0 + b0,1 + 1>>1$$
$$k0,0 = c_{0,0} + c_{0,1} + 1>>1$$
$$p_{0,0} = a_{0,0} + 3 * a_{0,1} + 2>>2$$
$$q_{0,0} = b0,0 + 3 * b0,1 + 2>>2$$
$$r_{0,0} = c_{0,0} + 3 * c0,1 + 2>>2$$

É possível observar nas equações a soma por 1 ou 2 que serve para que o arredondamento das divisões (implementadas usando shifts a direita) seja para o inteiro mais próximo. Sem esta soma, o arredondamento seria sempre para baixo, o que pode inserir um pouco de ruído na imagem.

Outra simplificação é a eliminação do preenchimento dos blocos. Como isso impede o cálculo dos pixels próximos a borda, só é usado para o cálculo do custo de movimento o bloco 6x6 no interior no bloco 8x8 que não toca nas bordas. A figura 4.1 ilustra esta abordagem. Isso além de eliminar o preenchimento permite diminuir o número de pixels a calcular soma das diferenças absolutas de 64 para 36. Ao invés de calcular os vetores em dois passos (primeiro com precisão 1/2 e depois com precisão 1/4), os subpixels de precisão 1/2 e 1/4 são gerados todos ao mesmo tempo. Embora isto implique no cálculo de 24 blocos ao mesmo tempo, ao invés de 18 no método original, isto simplifica a implementação em hardware além de melhorar o desempenho.

Figura 4.1 - Exemplo do método de busca de vetores de movimento



(a) (b) (c)

Fonte: Trabalho próprio

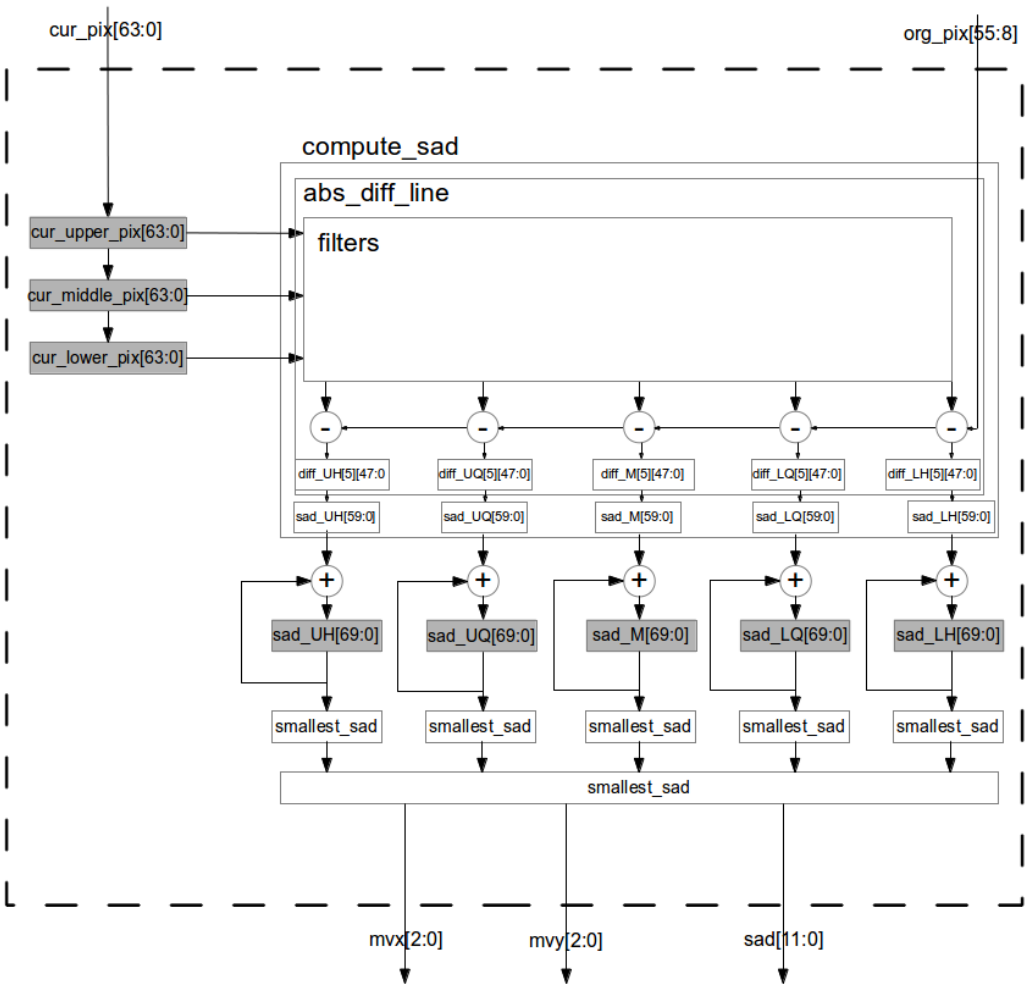
Na implementação original são efetuadas no total (8×8+27×7)×64 = 16192 multiplicações mais (8×7+27×6)×64 = 13952 somas e subtrações para o cálculo de cada bloco sub-pixel 8×8 enquanto no método proposto são necessárias somente (2×8+27×2)×36 = 2520 multiplicações e (2×7+27)×36 = 1066 somas por bloco o que gera uma economia na área do circuito e melhora o desempenho.

4.5 Implementação em hardware

O hardware foi implementado usando o Icarus Verilog como simulador. Embora este simulador seja limitado em termos de características e contenha um certo número de bugs ele é muito mais leve do que os softwares de simulação comerciais existentes e de fácil uso. Isso permite um rápido ciclo de implementação e testes.

A figura 4.2 mostra um diagrama da arquitetura de hardware proposta.

Figura 4.2 - Diagrama da arquitetura proposta



Fonte: Trabalho próprio

Para calcular os sub-pixels para cada linha é necessário ter sempre as linhas de pixels acima e abaixo da atual para poder calcular os sub-pixels na direção vertical, que são guardados em três registradores chamados cur_[upper|middle|lower]_pix. Estes registradores formam uma fila que recebe uma linha a cada ciclo e descartam a linha contida no último registrador. O conteúdo destes registradores é enviado a um bloco de filtros que calcula cada um dos sub-pixels. O resultado deste bloco é enviado para a o bloco que calcula o SAD entre estes sub-pixels e os pixels da linha correspondente no bloco original notado no diagrama pelo nome org_pix. O resultado do SAD para cada sub-pixel é salvo nos registradores sad_[UH|UQ|M|LQ|LH], que correspondem respectivamente aos halfpixels acima,

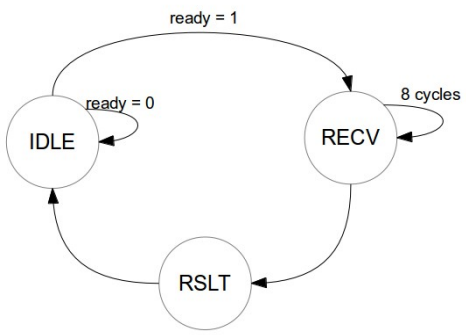
quarterpixels acima, subpixels na linha atual, quarterpixels na linha inferior e halfpixels na linha inferior. Ao final da entrada de todas as linhas é escolhido o menor SAD pelo bloco smallest_sad, primeiro na direção horizontal e depois na direção vertical e o que permite determinar o vetor de movimento e o SAD deste vetor de movimento. Se houver mais de um bloco com o valor de SAD mais baixo, por simplicidade na implementação é escolhido o primeiro bloco na ordem raster que contém este valor. A ordem na qual são escritas cada linha no bloco de hardware está notada a seguir:

```
cur_pix | org_pix |
1ªlinha |    X    |
2ªlinha |    X    |
3ªlinha | 2ªlinha |
4ªlinha | 3ªlinha |
5ªlinha | 4ªlinha |
6ªlinha | 5ªlinha |
7ªlinha | 6ªlinha |
8ªlinha | 7ªlinha |
   X    |    X    | --> mvx, mvy e sad prontos
```

Note que as linhas do bloco org_pix, que corresponde ao bloco original, só começam a serem enviadas dois ciclos após o envio da primeira linha do bloco cur_pix, que corresponde ao bloco atual (baseado no qual serão calculados os sub-pixels). Note também que a primeira e última linhas do bloco org_pix assim como os pixels da coluna da esquerda e da direita não são enviados pois o movimento só será calculado sobre o bloco interior 6x6.

Esta arquitetura é controlada por uma máquina de estados relativamente simples com apenas três estados. A figura 4.3 abaixo ilustra o diagrama de estados:

Figura 4.3 - Diagrama de estados



Fonte: Trabalho próprio

O estado IDLE é o estado padrão no momento em que é feito o reset. Quando o sinal de ready é enviado ao bloco as linhas são enviadas na ordem indicada anteriormente durante 8 ciclos no estado RECV. Ao final destes 8 ciclos a máquina de estados vai para o estado RSLT onde são exibidos os resultados e volta automaticamente no ciclo seguinte para o estado IDLE.

4.6 Resultados da síntese em FPGA

A implementação foi sintetizada usando o software Quartus II da Altera. O dispositivo escolhido foi o EP4CE15F17A7 da família Cyclone IV E. A escolha por este dispositivo foi pelo fato de que os chips desta família são de baixo custo e existem várias plataformas de desenvolvimento voltadas para estudantes.

O sintetizador indicou que foram usados 9181 dos 15408 elementos lógicos do dispositivo, o que corresponde a 60% de sua área interna e o clock máximo obtido foi de 42.65 MHz usando o modelo Slow 1200mV 125C.

O sintetizador foi executado usando o dispositivo Aria II GX EP2AGX45CU17C4. Este é um dispositivo de média gama que tem um custo relativamente alto ($462 na Digikey em 10 de Setembro de 2014), mas que é mais adequado para ser usado em aplicações de alto desempenho como um codificador de vídeo. Neste dispositivo foram usados 7343 ALUTs das 36100 disponíveis (20% de sua área) e o clock obtido foi de 79.88 MHz para o modelo Slow 900mV 85C.

A tabela 4.1 compara estes resultados:

Tabela 4.1: Resultados da síntese

Dispositivo	Clock e modelo	LUTs/ALUTS usados
Altera Cyclone IV E EP4CE15F17A7	42.65 MHz Slow 1200mV 125C	9181/15408 (60% do total)
Altera Aria II GX EP2AGX45CU17C4	79.88 MHz Slow 900mV 85C	7343/36100 (20% do total)

Levando em conta que cada PU 8x8 é processada em 10 ciclos foi feita uma estimativa de clock mínimo necessário para codificar em tempo real usando uma instância deste bloco. Para todos os casos foi considerado um quadro YUV 4:2:0. A formula usada para todas as estimativas foi:

frequencia = largura x altura x fps x 3 / 2 / 64 / 10

Esta estimativa não leva em conta tempo perdido para o bloco colocado em estado de espera ou o uso de múltiplas instâncias trabalhando em paralelo. A tabela 2 mostra o clocks mínimos necessários para processar vídeo para cada resolução e taxa de quadros:

Tabela 4.2: Clocks mínimos necessários para processar vídeo

Taxa de quadros / Resolução	30 fps	60 fps	120 fps
720x480	2.43 MHz	4.86 MHz	9.72 MHz
1920x1080	14.58 MHz	29.16 MHz	58.32 MHz
3840x2160	58.32 MHz	116.64 MHz	233.28 MHz

É possível observar na tabela que o clock obtido é o suficiente para processar vídeos de alta definição inclusive vídeos de resolução 3840x2160 caso for usado o dispositivo Aria II GX. Caso forem realizadas outras melhorias tanto no RTL desenvolvido como o uso de dispositivos de maior desempenho o desempenho deste bloco pode aumentar ainda mais.

5 RESULTADOS E ANÁLISE

Neste capítulo serão apresentados os resultados obtidos da solução proposta. Primeiro serão apresentados os resultados dos testes feitos usando um vídeo de ensaio padrão com uma implementação do algoritmo proposto. Em seguida serão apresentados os resultados da sintetização da implementação em hardware em uma FPGA.

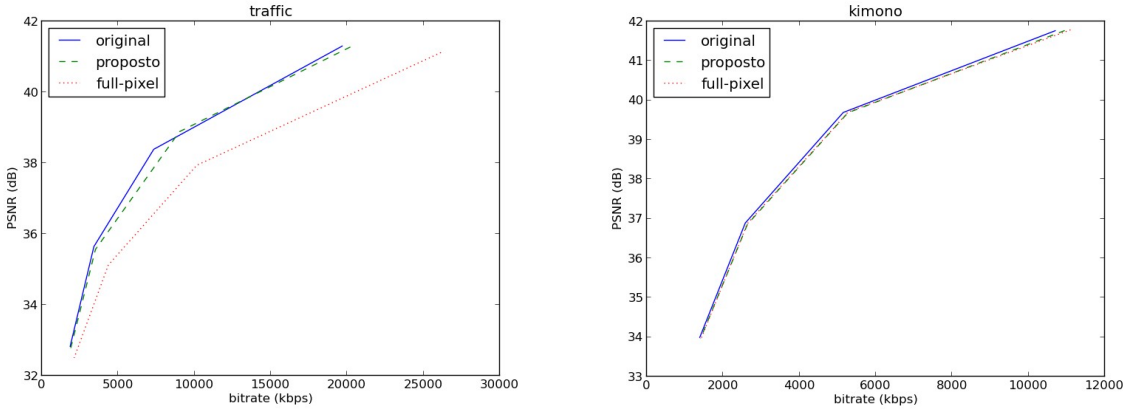
5.1 Testes com bitrate e PSNR

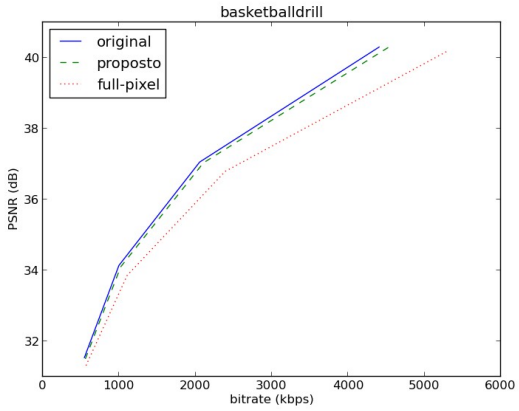
Por questões de desempenho e facilidade no desenvolvimento, esses ensaios foram feitos usando uma implementação em software da proposta comparando com a implementação original do software de referência. O software de referência foi modificado para só usar Prediction Units de tamanho 8x8 para permitir a comparação. As duas últimas colunas mostram os resultados para a predição somente usando vetores de movimento full-pixel, para mostrar que o método proposto possui resultados superiores a nenhuma busca sub-pixel. A tabela 4.1 mostra os resultados obtidos e na figura 4.4 estão plotados os gráficos das tabelas:

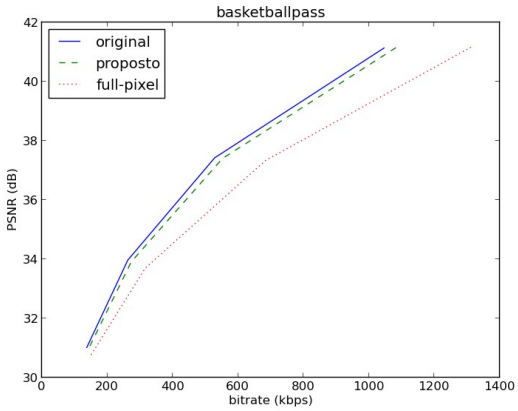
Tabela 5.1 - Resultados com PSNR dos ensaios

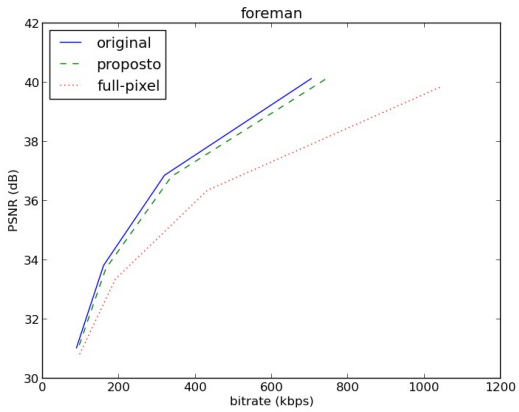
Seqüência	Resolução	QP	Bitrate método original	PSNR método original	Bitrate método proposto	PSNR método proposto	Bitrate full-pixel	PSNR full-pixel
Traffic	2560x1600	22	19671.46 kbps	41.30 dB	20204.93 kbps	41.28 dB	26137.57 kbps	41.12 dB
		27	7326.17 kbps	38.39 dB	9014.84 kbps	38.89 dB	10149.08 kbps	37.94 dB
		32	3407.45 kbps	35.64 dB	3510.18 kbps	35.56 dB	4353.90 kbps	35.12 dB
		37	1852.11 kbps	32.83 dB	1886.86 kbps	32.77 dB	2112.59 kbps	32.50 dB
Kimono	1920x1080	22	10699.45 kbps	41.76 dB	10946.27 kbps	41.77 dB	11098.41 kbps	41.79 dB
		27	5145.66 kbps	39.69 dB	5268.58 kbps	39.69 dB	5275.58 kbps	39.71 dB
		32	2578.30 kbps	36.89 dB	2634.73 kbps	36.86 dB	2630.11 kbps	36.87 dB
		37	1385.46 kbps	33.99 dB	1406.39 kbps	33.96 dB	1409.82 kbps	33.94 dB
Basketball Drill	832x480	22	4405.01 kbps	40.30 dB	4523.65 kbps	40.29 dB	5284.27 kbps	40.17 dB
		27	2054.04 kbps	37.06 dB	2103.00 kbps	37.03 dB	2389.56 kbps	36.79 dB
		32	996.04 kbps	34.14 dB	1019.00 kbps	34.10 dB	1103.95 kbps	33.86 dB
		37	543.51 kbps	31.54 dB	550.43 kbps	31.47 dB	563.55 kbps	31.31 dB
Basketball Pass	416x240	22	1045.93 kbps	41.14 dB	1082.21 kbps	41.15 dB	1311.37 kbps	41.16 dB
		27	528.05 kbps	37.42 dB	551.75 kbps	37.40 dB	683.41 kbps	37.33 dB
		32	262.68 kbps	33.97 dB	273.44 kbps	33.91 dB	315.99 kbps	33.70 dB
		37	137.20 kbps	31.02 dB	140.81 kbps	30.93 dB	149.45 kbps	30.75 dB
Foreman	352x288	22	703.10 kbps	40.13 dB	739.59 kbps	40.11 dB	1039.66 kbps	39.84 dB
		27	318.54 kbps	36.87 dB	337.18 kbps	36.83 dB	431.16 kbps	36.36 dB
		32	159.20 kbps	33.83 dB	165.09 kbps	33.72 dB	189.87 kbps	33.36 dB
		37	87.99 kbps	31.04 dB	90.23 kbps	30.94 dB	94.22 kbps	30.77 dB

Figura 4.4 - Gráficos dos resultados









Fonte: Trabalho próprio

Com o intuito de simplificar os testes todas as seqüências supracitadas foram codificadas somente os 30 primeiros quadros. O arquivo de configuração usado para todos os testes foi o encoder_lowdelay_P_main.cfg (com modificações para limitar o uso de PUs a somente de tamanho 8x8).

É interessante notar que na seqüência kimono o método proposto não possui vantagem sobre o uso de vetores full-pixel sem o cálculo de vetores sub-pixel embora em todos os outros casos, embora exista uma perda é notável a melhoria em relação ao uso de vetores full-pixel somente.

A tabela 2 mostra os valores de BD-Rate que quantificam a diferença entre a curva de bitrate e PSNR usando o método original e o método proposto para cada amostra, calculado usando o utilitário avsnr:

Tabela 5.2: Valores de BD-Rate para cada seqüência

Seqüência	BD-Rate
Traffic	3.88 %
Kimono	3.56 %
Basketball Drill	2.21 %
Basketball Pass	1.31 %
Foreman	0.88 %

Nesta tabela é possível notar que as curvas possuem diferença maior usando o método proposta (ou seja, o método proposto tem o desempenho mais fraco) nas amostras com resolução maior. Os testes mostram que a perda de PSNR e o ganho em bitrate é insignificante comparado ao método realizado pelo software de referência, que calcula as posições sub-pixel usando o mesmo filtro de 7 ou 8 taps do decodificador o que gera uma alta precisão, embora um alto custo para realizar o cálculo. Outro agravante é que o método original considera também o custo de codificação dos vetores de movimento no cálculo do custo de movimento e o método proposto não, o que dá uma maior vantagem ao método original. A linha mostrando os resultados usando somente vetores full-pixel mostra que as perdas no método proposto não são piores do que o não uso de vetores com precisão sub-pixel.

5.2 Comparação com outros trabalhos

A tabela 5.3 mostra um comparativo com os outros trabalhos citados:

Tabela 5.3: Comparativo com outros trabalhos

	Proposto	(AFONSO 2012)	(HE 2013)	(DAI 2012)	(KAO 2006)
FPGA ou processo	Altera Aria II GX EP2AGX45CU17C4	Altera Stratix III EP3SE50F484C2	E-Shuttle 65nm CMOS	Software somente	TSMC 130nm
Número de LUTS ou gates usados / memória	7343 LUTs	152 ALUTs 157 registradores	1183k (2-input NAND eq.) / 19.2 kB		56,539
Padrão usado	HEVC	HEVC	HEVC	HEVC	H.264/AVC
Resolução máxima em tempo real	3840x2160 @ 30fps	3840x2160 @ 64fps	7680x4320 @ 30fps		3200x2400 @ 30 fps
Queda na qualidade	0.88-3.88% BD-Rate	Nenhuma (igual ao SW referência)	Não aferido como um todo	0.00-0.03 dB 0.4-0.8% bitrate	0.15 dB

Na tabela é possível observar uma variadade de tecnologias e casos diferentes de testes que não permitem uma comparação conclusiva, mas que podem dar uma noção do estado da arte das técnicas. O trabalho de (AFONSO 2012) é o que tem o melhor resultado em termos de PSNR e bitrate pois faz uma implementação precisa conforme a especificação, mas que custa mais em termos de hardware. Os trabalhos de (HE 2013) e (KAO 2006) foram feitos em hardware o que leva a ganhos expressivos no desempenho do bloco, embora no caso de (KAO 2006) a técnica usada é antiga e não tem um ganho significativo em relação a FPGAs. No trabalho de (HE 2013) não é feita uma avaliação global do impacto no PSNR do resultado final, apenas em cada técnica usada, alegando impactos entre 0.02 dB e 0.1 dB para cada sem afirmar se estes impactos se acumulam linearmente. No trabalho de KAO (2006) há uma tabela que mostra que os experimentos foram feitos usando amostras tipicas para o H.264/AVC (como Foreman e Carphone), sem indicar qual bitrate foi usado ou obtido. O trabalho de (DAI 2012) só foi implementado em software e só é possível fazer uma comparação em termos de qualidade, que é ligeiramente superior ao do trabalho proposto.

Estes trabalhos acima citados possuem implementações e casos de teste muito diversificados o que leva a necessidade da existência de um artigo que compare todas estas arquiteturas usando um método padrão.

6 CONCLUSÃO

Este trabalho apresenta uma arquitetura de estimativa de vetores de movimento sub-pixel de baixa complexidade e com um impacto na qualidade comparável ao estado da arte.

Este trabalho permite a melhoria em vários pontos. Uma delas é considerar o custo de cada vetor de movimento o que permite uma melhor compressão no vídeo. Outra melhoria é o de usar filtros mais precisos para os pixels mais próximos do centro da PU, pois os pixels mais interiores possuem mais pixels vizinhos, logo permitem o uso de um filtro mais largo (com mais taps) sem precisar de preenchimento (padding) o que melhora a precisão do cálculo do custo levando a uma qualidade superior do vídeo resultante, embora em troca de um custo mais alto de implementação em termos de área de circuito e clock menor. Em outra direção também pode se optar por fazer uma simplificação ainda maior do algorítmo usando somente o bloco 4x4 no interior do da PU 8x8. Isto pode ter um impacto negativo na qualidade da saída do codificador que deve ser avaliada. Outra melhoria possível seria a implementação deste algorítimo também para PUs de diferentes tamanhos como 16x16 ou 16x8.

REFERÊNCIAS

AFONSO, V.; **Desenvolvimento de arquiteturas para estimação de movimento fracionária segundo o padrão HEVC**. 2012, 56 f. Trabalho Individual II, UFPel, Pelotas, 2012.

BELMUDEZ, B.; MOELLER, S.; LEWCIO, B.; RAAKE, A.; MEHMOOD, A. Audio and video channel impact on perceived audio-visual quality in different interactive contexts. **IEEE International Workshop on Multimedia Signal Processing**, Rio De Janeiro, p. 5-12, IEEE, out. 2009.

BOSSEN, F.; BROSS, B.; SÜHRING, K.; FLYNN, D.; HEVC Complexity and Implementation Analysis. **IEEE Transactions on Circuits and Systems for Video Technology,** Chicago, v. 22, n. 12, p. 1685-1696, IEEE, dez. 2012.

Converting Between YUV and RGB. [S.l.:s.n] Disponível em < http://msdn.microsoft.com/en-us/library/ms893078.aspx>. Acesso em 7 ago. 2014.

DAI, W.; AU, C. O.; LI, S.; SUN, L.; ZOU, R.; Fast sub-pixel motion estimation with simplified modeling in HEVC, **Circuits and Systems (ISCAS), 2012 IEEE International Symposium on,** Seoul, p. 1560-1563. IEEE, mai. 2012.

HE, G.; ZHOU, D.; CHEN, Z.; ZHANG, T.; GOTO, S. A 995Mpixels/s 0.2 nJ/pixel fractional motion estimation architecture in HEVC for Ultra-HD, **In Solid-State Circuits Conference (A-SSCC), 2013 IEEE Asian**, Singapore, p. 301-304. IEEE, nov. 2013.

JACK, K.; **Video Demystified: A Handbook for the Digital Engineer**, 5. ed., Burlington Newnes, 2007.

KAO, C.-Y.; KUO, H.-C.; LIN, Y.-L.; High performance fractional motion estimation and mode decision for H.264/AVC. **Multimedia and Expo, 2006 IEEE International Conference on,** Toronto, p. 1241-1244. IEEE, jul. 2006.

KIM, I.-K.; MIN, J.; LEE, T.; HAN, W.-J.; PARK, J.H. Block partitioning structure in the HEVC standard. **Circuits and Systems for Video Technology, IEEE Transactions on,** Chicago, v. 22, n. 12, p. 1697-1706, IEEE, dez. 2012.

LIU, J.; WANG, J. **JPEG Compression and Ethernet Communication on an FPGA.** [S.l.:s.n] Disponível em < http://people.ece.cornell.edu/land/courses/ece5760/FinalProjects/f2009/jl589_jbw48/jl589_jbw48/>. Acesso em 15 jul. 2014.

OHM, J. SULLIVAN, G. J.; SCHWARZ, H.; THIOW KENG TAN ; WIEGAND, T. Comparison of the Coding Efficiency of Video Coding Standards—Including High Efficiency Video Coding (HEVC). **IEEE Transactions on Circuits and Systems for Video Technology,** Chicago, v. 22, n. 12, p. 1669-1684, IEEE, dez. 2012.

RICHARDSON, I. E.; **The H. 264 advanced video compression standard.** 2. ed. John Wiley & Sons, 2011.

SAYOOD, K.; **Introduction to Data Compression**. 2. ed. San Francisco: Morgan Kaufmann Publishers, 2000.

SUH, J. W.; JEONG, J.. Fast sub-pixel motion estimation techniques having lower computational complexity. **Consumer Electronics, IEEE Transactions on,** Reading, v. 50, n. 3, p. 968-973, IEEE, ago. 2004.

SULLIVAN, G. J.; OHM, J.-R; HAN W.-J; WIEGAND T. Overview of the High Efficiency Video Coding (HEVC) Standard. **IEEE Transactions on Circuits and Systems for Video Technology,** Chicago, v. 22, n. 12, p. 1649-1668, IEEE, dez. 2012.

TOURAPIS, A. M.; Enhanced predictive zonal search for single and multiple frame motion estimation. **Proc. SPIE 4671, Visual Communications and Image Processing 2002**, p. 1069-1079. International Society for Optics and Photonics, jan. 2002.

APÊNDICE – ARTIGO DO TG1

Resumo: Este artigo descreve a proposta do Trabalho de Graduação de Engenharia de Computação realizado por Gabriel Diego Teixeira, o qual consiste no desenvolvimento de um módulo de busca de vetores de movimento com precisão sub-pixel para o codec HEVC em FPGA. Este trabalho tem como motivação a necessidade da aceleração do processamento do codec HEVC, também conhecido informalmente como h.265, devido ao fato de que este é um codec que demanda grande tempo de processamento em software. Uma implementação em FPGA seria uma maneira de se ter uma implementação que executa mais rapidamente.

Introdução

Existe um grande número de aplicações nos dias de hoje que dependem da transmissão ou armazenamento de vídeo digital como a tv aberta, tele-conferências, tele-medicina entre vários outros. Porém, vídeo digital exige uma grande capacidade de banda passante para ser transmitida ou de espaço de armazenamento caso se queira armazenar um vídeo com alta definição [1].

Atualmente existem codecs (algoritmos de compressão e descompressão de sinais digitais) que conseguem uma boa relação entre qualidade do vídeo reconstruído em relação ao original e o volume de dados resultante. Entre tais codecs pode-se citar WebM, H.264/AVC e o HEVC. O poder de processamento necessário para executar um codificador (software que implementa o algoritmo de compressão do codec) em tempo real, isto é, comprimir um vídeo no mesmo ritmo em que ele é adquirido, especialmente no caso do HEVC, pode superar até mesmo a capacidade das CPU's mais rápidas que existem atualmente, quando se trata de vídeos em alta definição. Além disso, mesmo no caso de codificação para armazenamento, onde se prioriza uma melhor qualidade visual, o tempo de codificação ou transcodificação (conversão de um formato de vídeo codificado para outro) não pode ser exageradamente alto [2].

Uma maneira eficiente de codificar de vídeo de alta definição em tempo real é a implementação em hardware do codec, ou pelo menos das partes de processamento mais intensivo como, por exemplo, a busca por vetores de movimento. Na maior parte dos casos só a pesquisa de vetores de movimento representa entre 60% e 90% do tempo de processamento total [3]. Por outro lado implementar um codec (ou qualquer outro algoritmo) em ASIC (*Application Specific Integrated Circuit*, um chip desenvolvido para uma aplicação especifica) é caro e demorado. Para agilizar e aliviar o custo de desenvolvimento FPGAs (*Field Programmable Gate Array*) podem ser usadas. FPGAs são chips com blocos lógicos e switches programáveis que podem se comportar de forma semelhante a um ASIC, embora com desempenho mais limitado e custo unitário superior [4].

Codecs de vídeo

Vídeo digital pode requerer uma grande quantidade de dados para ser guardado em uma mídia digital ou mesmo ser transmitido. Se for considerado, por exemplo, um vídeo de resolução 1920 por 1080 pixels a 30 quadros por segundo no formato de pixel RGB com 24 bits para cada pixel (um byte para cada cor: vermelho, verde, e azul), se teria uma ocupação de mais de 1492 Megabits por segundo de vídeo ou mais de 938 Gigabytes para uma seqüência de 90 minutos. Enquanto isso, uma transmissão de vídeo digital em TV aberta tem uma banda de no máximo 19 Megabits por segundo [5] e um disco Bluray de dupla camada tem no máximo 50 Gigabytes disponível para guardar tanto o vídeo como o áudio [6].

Para poder diminuir o volume de dados que um vídeo precisa para ser guardado, é necessário utilizar um algoritmo de compressão de vídeo que se aproveite da redundância dos dados em uma sequência de vídeo. Como forma de explorar esta redundância e para aumentar a taxa de compressão, alguns algoritmos também provocam uma pequena perda de qualidade que seja pouco ou não perceptível para a pessoa que vai assistir o vídeo [7].

Um codec de vídeo é um programa que implementa um algoritmo de compressão ou descompressão de vídeo. O algoritmo de descompressão normalmente é padronizado por uma especificação ou por um software de referência, o que garante compatibilidade nas implementações. Normalmente só o decodificador é padronizado, cabendo ao desenvolvedor do codificador a tarefa de encontrar um meio adequado de desenvolver um algorítmo que gere um vídeo considerando questões como tempo de processamento, taxa de saída, qualidade desejada entre outras.

Qualidades de um codec de vídeo

A qualidade de um codec de vídeo pode ser avaliada em diversas dimensões entre as quais pode-se citar:

- Volume de dados resultante.
- Qualidade visual do vídeo resultante (no caso de codificação com perdas).
- Atraso entre a entrada do vídeo para ser comprimido e a saída do vídeo descomprimido.
- Desempenho do codificador ou do decodificador que tem os seguintes sub-parâmetros:
    - Tempo de execução.
    - Memória usada.
    - Área de circuito necessária (no caso de uma implementação em hardware).

Tais dimensões podem ser dependentes tanto do codec quanto da implementação do mesmo. Como neste trabalho de graduação será usado um codec padrão a maximização destes parâmetros será feita pela qualidade da implementação, visto que a implementação de um codificador tem-se a liberdade de fazer tal implementação de várias maneiras, mas apenas uma pequena parte delas são de real interesse. Além disso, alguns destes parâmetros podem ser interdependentes, como o volume de dados e a qualidade visual. Neste caso é necessário encontrar o melhor compromisso entre todos os parâmetros para se ter um equilíbrio ótimo.

Padrões de codec modernos

Entre os codecs mais modernos existentes pode-se destacar o WebM, o H.264/AVC e o recente HEVC, que deve ser ratificado em janeiro de 2013[8].

O WebM era um codec proprietário desenvolvido pela empresa On2 (na época ele era chamado de VP8). A On2 foi comprada pela Google, que decidiu colocar o codec VP8 a disposição da comunidade sem cobrar royalties pelo seu uso, diferentemente da grande maioria dos codecs modernos que normalmente tem uso restrito por patentes. Como parte dessa iniciativa a Google renomeou este codec para WebM para refletir o apoio de tecnologias livres para uso na internet já que o codec mais usado para se transmitir vídeos na internet, o H.264/AVC, possui patentes que restringe seu uso no âmbito do software livre. Por outro lado o WebM é criticado por não ter uma qualidade compatível com o H.264/AVC além da baixa adoção por parte de várias empresas. Algumas empresas, como a Apple, já afirmaram que não vão permitir tais implementações do WebM [9].

O H.264/AVC é provavelmente o codec mais popular em uso atualmente com várias implementações em hardware e software e uma penetração de mercado muito significativa. O H.264/AVC também é o codec com uma das melhores qualidades no mercado em termos de qualidade visual e volume de dados resultante [9], embora o tempo de processamento e memória usada seja bastante alto, compensado pelo fato de que existem várias implementações em hardware e a existência de perfis de baixo nível que sacrificam a qualidade visual e de compressão por uma execução mais eficiente.

O HEVC, também popularmente conhecido como h.265, é um codec de próxima geração que deve substituir o H.264/AVC como codec de melhor qualidade visual e compressão. Por outro lado este codec é ainda mais exigente em termos de processamento comparado aos demais codecs existentes. Além disso, o HEVC ainda não está com a sua especificação pronta, que está prevista para ser divulgada em janeiro de 2013, mas já existe um rascunho que permite a implementação de praticamente todas as características que este codec deve ter no lançamento oficial [5].

Estrutura de um codec de vídeo

Embora possam existir várias arquiteturas de um codec de vídeo, pelo menos nos codecs já citados segue-se uma estrutura básica como a mostrada na figura seguinte [10]:

Estimativa de movimento usando pesquisa de vetores de movimento

Muitos dos codecs usados hoje em dia, incluindo aí os já citados, são baseados em blocos, isto é, as imagens de entrada são particionadas em pequenos conjuntos quadrados de pixels de tamanho variando de 4x4 a 32x32, dependendo do codec. De maneira simplificada cada bloco passa por um processo de transformada, que serve para concentrar as informações do bloco em um numero menor de coeficientes, quantização, que serve para diminuir a variedade de coeficientes possíveis e facilitando a correlação durante a codificação entrópica, mapeado em zig-zag, o que deixa os coeficientes numa ordem que é mais conveniente para a compressão, e por fim comprimindo usando um método que busca a correlação entre os coeficientes quantizados usando o menor número de bits possível [10].

Em cada quadro seguinte da seqüência, ao invés de se fazer as mesmas operações sobre os pixels dos blocos do novo quadro, busca-se representar o novo bloco com a cópia do bloco do quadro anterior na mesma posição. Como a imagem contém sequências em movimento, o bloco que possui a melhor correlação com o atual pode estar numa posição diferente, logo é necessário buscar no quadro anterior o bloco que mais se parece com o bloco

atual. Tal busca envolve comparar vários blocos usando uma métrica (geralmente a soma do valor absoluto das diferenças entre cada pixel dos blocos comparados chamado de SAD) até se encontrar o bloco que tem o valor mais adequado [11].

Como o movimento na sequência frequentemente é desalinhado em relação à mascara de pixels, o movimento de um objeto pode acabar caindo entre dois pixels. Como forma de diminuir o custo de codificação do bloco, pode-se incluir a pesquisa por vetores de movimento em pixels imaginários que ficam entre dois pixels vizinhos. Isso é feito fazendo a pesquisa por vetores de movimento em uma imagem alongada por um fator de dois (ou até de quatro ou oito, como fazem alguns codecs), e o vetor de movimento acaba assumindo um valor fracionário. Quando o fator de escala é de dois, isto é chamado também de movimento com precisão half-pixel e se o fator for de quatro, quarter-pixel ou de uma maneira genérica, sub-pixel [10].

A busca por vetores de movimento é computacionalmente muito intensiva pois envolve calcular a diferença entre muitos pixels para cada bloco em diferentes quadros que possuem vários blocos cada. Para compensar o grande número de operações cada comparação pode ser implementada usando instruções em assembly altamente otimizadas em codificadores baseados em software ou usando uma implementação em hardware que faça o máximo possível de operações em paralelo. Mesmo tais otimizações podem resultar em uma grande quantidade de processamento o que leva a necessidade de se otimizar os algoritmos que fazem a busca. Para diminuir o número de comparações usa-se algoritmos que tentam estimar os vetores de movimento reusando os vetores de movimento de blocos vizinhos ou de blocos do quadro anterior [11].

Desenvolvimento de hardware

Um ASIC e capaz de obter o máximo desempenho possível usando as tecnologias atuais de sistemas de computação quando é necessário a implementação de um algorítmo de alto desempenho. Como um ASIC pode custar muito caro para desenvolver e tal desenvolvimento toma um tempo muito longo, uma alternativa para desenvolver hardware é a de desenvolver em FPGA ainda que aceitando uma perda no desempenho e o ganho no custo unitário de cada unidade produzida [13].

FPGAs são chips com blocos lógicos e switches programáveis que podem se comportar de forma semelhante a um ASIC, embora com desempenho mais limitado e custo unitário superior. Para programar uma FPGA usa-se uma linguagem de descrição de hardware (como Verilog ou VHDL) de uma maneira muito semelhante a um ASIC. Isso vem em

contraste ao desenvolvimento em software onde o paradigma é o de uma unidade de processamento que lê o valor de uma ou mais posições de memória ou registradores, calcula um resultado e escreve ele de volta na memória ou registradores, sendo estes passos realizados ordenadamente, exceto no caso de múltiplos processadores ou processadores gráficos. Em FPGAs o processamento da informação é por definição paralelo através de tantas células programáveis quanto o algoritmo (e a implementação) consegue utilizar.

FPGAs também possuem vários módulos de hardware frequentemente usados embutidos como módulos DSP, blocos de RAM, multiplicadores e até mesmo microprocessadores inteiros, entre outros. Isto evita a implementação repetitiva de blocos de hardware muito usados, visto que a implementação de tais blocos usado portas lógicas básicas ocupam uma quantidade substancial da lógica além de serem menos eficientes em termos de consumo de potência e desempenho comparado ao uso de blocos prontos.

Proposição do projeto

Este projeto visa acelerar o algorítimo de pesquisa de vetores de movimento do codec HEVC com precisão sub-pixel em Verilog. O objetivo até o final do semestre é o de obter uma descrição em hardware desse algoritmo e de testar a mesma implementação em FPGA.

Trabalhos existentes

Vladmir Afonso, mestrando da UFPel em Ciência da Computação, está trabalhando também em uma arquitetura de estimativa de vetores de movimento fracionários no padrão HEVC [12]. Neste trabalho ele propõe uma simplificação das equações do filtro de amostras sub-pixel como forma de diminuir o uso do hardware. Como forma de aumentar o desempenho desta arquitetura, ele propõe também a separação do fluxo de cálculo deste filtro em dois estágios formando um pipeline.

Existem alguns trabalhos que propõem a implementação do algoritmo de pesquisa de vetores de movimento em software [13][14]. Isto está fora do escopo deste trabalho que propõe a implementação em hardware onde o paralelismo disponível leva a outros tipos de otimizações.

Metodologia

Softwares usados

O hardware será desenvolvido em Verilog usando o software iverilog para simular o código existente e o software gtkwave para a visualização das formas de onda. O iverilog

(abreviação de Icarus Verilog) é uma ferramenta que é capaz de simular e sintetizar um hardware descrito em Verilog seguindo o padrão IEEE-1364. Essa ferramenta foi escolhida por ser de uso livre, não depender de esquemas de licença complicados e pouco confiáveis e por ser fácil de instalar na maioria das máquinas que rodam Linux, Windows ou MacOS. O problema com essa ferramenta é que a implementação do Verilog está incompleta, embora a ferramenta seja usável. O software gtkwave complementa o iverilog para a visualização das formas de onda produzidas pelo iverilog. De maneira similar ao iverilog, esta ferramenta é de uso livre, fácil instalação e confiável, requer poucos recursos computacionais para executar, além de ter todos os recursos necessários para o projeto.

Para implementar em hardware, o software usado será o Xilinx ISE, para poder compilar e executar o código na FPGA disponível para o desenvolvimento no laboratório. Esta ferramenta só será usada para a síntese na FPGA e eventual correção de erros que só surgem quando se compila o código em uma FPGA.

Comparação

A implementação em hardware será comparada com a implementação do software de referência com o objetivo de verificar se a implementação está conforme a especificação. Além dessa comparação de conformidade, os vetores de movimento gerados pela implementação em hardware serão inseridos no software de referência para verificar a diferença na qualidade resultante. Como a implementação em hardware é apenas parcial, não será verificado o desempenho pois a implementação híbrida teria uma penalidade muito grande de desempenho o que invalidaria a comparação a implementações puramente em hardware ou em software.

Ainda assim pode-se avaliar o desempenho do bloco separadamente. Nesse caso será avaliado o tempo que bloco leva para executar sobre cada *treeblock*. Tendo em mãos o tempo de execução de cada *treeblock*, será possível determinar a resolução máxima obtida pela implementação. A seguinte equação descreve o desempenho máximo da implementação:

$$fps = TreeBlockSize * TreeBlockSize / (timeTreeBlock * frameWidth * frameHeight) \quad (1)$$

Onde *fps* é o número de quadros processados por segundo, *TreeBlockSize* é a largura ou altura de cada *treeblock* (análogo ao *macroblock* do H.264/AVC, mas que pode ser 16, 32 ou 64), *timeTreeBlock* é o tempo que cada *treeblock* leva para ser processado, *frameWidth* é a

largura do quadro e *frameHeight* é a altura do quadro. *timeTreeBlock* pode ser representado alternativamente por:

$$timeTreeBlock = cyclesTreeBlock * periodClock \quad (2)$$

Onde *cyclesTreeBlock* é o número de ciclos para processar cada *treeblock* e *periodClock* é o período do *clock* máximo possível de ser obtido no hardware onde foi implementado. Caso supormos que temos 1000 ciclos para processar cada *treeblock* de tamanho 64 a um *clock* de período de 10ns (ou seja, 100MHz) um vídeo de largura 3840 e altura 2160, temos aí um número de quadros por segundo de 49,38 *fps*.

Cronograma preliminar

Janeiro/2013: Avaliação do desempenho da implementação de referência do HEVC em CPU.

Fevereiro/2013: Implementação dos filtros de interpolação em Verilog.

Março/2013: Implementação do módulo de pesquisa de vetores de movimento em Verilog.

Abril-Maio/2013: Comparação com o software de referência e correção de bugs na implementação.

Maio/2013: Testes de desempenho da implementação final.

Junho/2013: Redação do relatório final.

Referências

[1] HUSEMANN R. et al. Aumento de Desempenho de Codificação Intra H.264 Usando Arquitetura CUDA In: Webmedia'11. Florianópolis, Brasil. 2011. 4 p.

[2] PENG W.-H. Recent Advances of High Efficiency Video Coding Report of NCTU. China. 2012. 21 p.

[3] KUHN P., Algorithms, Complexity Analysis and VLSI Architectures for MPEG-4 Motion Estimation, Boston, MA: Kluwer Academic, 1999

[4] CHU P. P. “FPGA Prototyping by VHDL examples” , 2008 John Wiley & Sons, Ltd. ISBN 978-0-470-18531-5

[5] FARIAS, M. C. Q. O Padrão de Televisão Digital Nacional. In: Marcelo Sampaio Alencar. (Org.). Televisão Digital. 1ª ed. São Paulo: Érica, 2007, p. 237-266.

[6] BUTLER, H. Pioneer BDXL BDR-206MBK Review

[7] JACK K. Video Desmystified: A Handbook for the Digital Engineer. Newnes. 5th edition. 944 p.

[8] “High Efficiency Video Coding (HEVC) achieves first formal milestone toward completion". JCT-VC. 2012-02-10. Acessado em 2012-08-29

[9] RICHARDSON I. E. G., “H.264 and MPEG-4 Video Compression: Video Coding for Next-generation Multimedia”, 2003 John Wiley & Sons, Ltd. ISBN: 0-470-84837-5

[10] RICHARDSON I. H.264 and MPEG-4 Compression: Video Coding for Next Generation Multimedia. Wiley. 1th. edition. 2003. 320 p.

[11] VANNE J. Design and Implementation of Configurable Motion Estimation Architecture for Video Encoding Thesis of Science in Technology. Tampere, Finland. 2011. 155 p.

[12] AFONSO, Vladimir. Desenvolvimento de Arquiteturas para Estimação de Movimento Fracionária Segundo o Padrão HEVC. 2012. 53 f. Trabalho Individual(Mestrado em Ciência da Computação). Universidade Federal de Pelotas, Pelotas.

[13] DAI, Wei, "Fast sub-pixel motion estimation with simplified modeling in HEVC," Circuits and Systems (ISCAS), 2012 IEEE International Symposium on , vol., no., pp.1560-1563, 20-23 May 2012

[14] DAI, Wei; Au, Oscar C.; Li, Sijin; Sun, Lin; Zou, Ruobing; , "Fast sub-pixel motion estimation with simplified modeling in HEVC," Circuits and Systems (ISCAS), 2012 IEEE International Symposium on , vol., no., pp.1560-1563, 20-23 May 2012