Investigação de Algoritmos e Desenvolvimento Arquitetural para a Estimação de Movimento em Compressão de Vídeo Digital

Marcelo Porto, Luciano Agostini
Universidade Federal de Pelotas (UFPel) – DInfo
Grupo de Arquiteturas e Circuitos Integrados (GACI)
Pelotas, Brasil, *Caixa Postal 354 – CEP. 96010-900*
(porto, agostini)@ufpel.edu.br

e

Sergio Bampi
Universidade Federal do Rio Grande do Sul (UFRGS) - II
Grupo de Microeletrônica (GME)
Porto Alegre, Brasil
bampi@inf.ufrgs.br

Abstract

This work investigates some motion estimation algorithms for video compression and it presents results from software implementations for the investigated algorithms. These results were used to define the motion estimation architecture presented in this paper. The designed architecture uses the 4:1 *Pel Decimation* algorithm with SAD, in blocks with 16x16 samples. The search area was defined as 64x64 samples. The synthesis results show that this architecture is able to processes more than 60 SDTV frames (720 x 480 *pixel*s) per second. This result shows that the designed architecture is able to compress SDTV videos in real time.

Keywords: Motion Estimation, Video Compression, VHDL design.

Resumo

Este trabalho investiga algoritmos para a estimação de movimento em compressão de vídeo e apresenta resultados de implementações em software para estes algoritmos. Estes resultados serviram de base para o desenvolvimento da arquitetura para a estimação de movimento que está apresentada neste artigo. A arquitetura desenvolvida utiliza o algoritmo de busca Pel Decimantion 4:1 com SAD, sobre blocos de 16x16 amostras. A área de busca foi definida em 64x64 amostras. Os resultados de síntese indicam que a arquitetura é capaz de processar mais de 60 quadros SDTV (720 x 480 *pixels*) em um segundo. Este resultado indica que a arquitetura desenvolvida é capaz de comprimir vídeos SDTV em tempo real.

Palabras claves: Estimação de movimento, Compressão de vídeo, Projeto VHDL.

1. INTRODUÇÃO

A compressão de vídeos digitais possui grande relevância na atualidade. O interesse da indústria em codificadores de vídeo conduziu à criação de diversos padrões para compressão de vídeo, como o MPEG-2 [1] e o H.264/AVC [2]. Estes padrões reúnem um conjunto de algoritmos e técnicas que reduzem drasticamente a quantidade de informações necessárias para representar um vídeo digital.

Em um codificador, a etapa que gera a maior parte do ganho obtido na compressão é o estimador de movimento (ME – *Motion Estimation*). No entanto, a ME é uma tarefa complexa e, em muitos casos, é impossível sua implementação em software quando se processam vídeos de alta resolução em tempo real.

Este trabalho apresenta uma investigação sobre alguns dos algoritmos de estimação de movimento mais utilizados, bem como o desenvolvimento de uma arquitetura para realizar esta tarefa usando o algoritmo *Pel Decimation* 4:1. Os resultados de síntese da arquitetura indicam que ela é capaz de atingir, com folga, tempo real para resoluções como SDTV (720 x 480 *pixel*s).
A seção dois do artigo apresenta alguns princípios da estimação de movimento. A seção três apresenta os algoritmos de busca investigados e a seção quatro apresenta a função SAD. A seção cinco mostra os resultados das implementações em software. A seção seis apresenta a arquitetura desenvolvida. A seção sete apresenta os resultados de síntese e, na seção oito, estão apresentadas as conclusões e os trabalhos futuros.

2. PRINCÍPIOS DA ESTIMAÇÃO DE MOVIMENTO

Imagens vizinhas em um vídeo tendem a ser muito semelhantes, isto gera uma grande quantidade de informações redundantes. O objetivo principal da estimação de movimento é reduzir a redundância temporal entre imagens vizinhas. A técnica consiste em identificar estas redundâncias e mapear as diferenças através de vetores denominados de vetores de movimento. A partir de um quadro, denominado de quadro de referência, o quadro atual é estimado a partir de vetores de movimento. Para cada bloco do quadro será gerado um vetor de movimento que corresponderá à posição onde este bloco obteve a melhor relação de similaridade no quadro de referência. Uma área de pesquisa será determinada no quadro de referência para cada bloco do quadro atual. Um algoritmo de busca determina a maneira como esse bloco se desloca dentro desta área de pesquisa para que a melhor relação de similaridade (melhor *matching*) seja encontrada. Este algoritmo é chamado de algoritmo de busca e existem diversos algoritmos com características distintas, publicados na literatura. A melhor relação de similaridade é definida por um algoritmo que compara os blocos de acordo com algum critério pré-determinado. Este critério é chamado de critério de distância e existem diversas alternativas de critérios de distância publicados na literatura. A informação do vetor de movimento é gerada para cada bloco de luminância do quadro atual. Esta operação implica em uma grande complexidade computacional, mas, no entanto, proporciona uma redução significativa da quantidade de informação necessária para formar o vídeo. Além dos vetores, também são enviados no *bitstream* do vídeo codificado, os resíduos, que resultam da subtração entre o quadro original e o quadro remontado após o processo de estimação e compensação de movimento [3].

3. ALGORITMOS DE BUSCA

Os algoritmos de busca na estimação de movimento determinam a forma como será realizada a busca dentro da área de pesquisa. O algoritmo de busca tem influência direta na complexidade computacional da estimação, bem como na qualidade dos vetores gerados. Este trabalho irá investigar os seguintes algoritmos de busca: *Full Search* [4], *Three Step Search* (TSS), *One at a Time Search* [3], *Diamond Search*, e *Pel Decimation* 2:1 e 4:1 [5]. Nesta seção serão apresentados os algoritmos estudados, bem como os resultados das implementações em software para as implementações destes algoritmos.

3.1 *Full Search*

O algoritmo *full search* procura a melhor relação entre as áreas de pesquisa (melhor *matching*) comparando o bloco que está sendo pesquisado com todas as posições possíveis dentro da área de pesquisa. O bloco é deslocado de um *pixel* dentro da área de pesquisa, começando do canto superior esquerdo, até que todas as posições tenham sido comparadas e a menor diferença tenha sido registrada. Ao final, será gerado um vetor de movimento referente ao deslocamento do bloco para a região de melhor *matching* [4]. A Figura 1 ilustra o processo de busca completa.
A Figura 1 ilustra uma região de pesquisa de +/- 7 *pixels* em torno do bloco corrente (supondo que a posição original do bloco seja a posição central da área de pesquisa). Cada ponto na figura representa um vetor de movimento, sendo que o vetor que resultar no melhor casamento será o escolhido. Para cada ponto marcado na figura uma função de similaridade será aplicada para cada *pixel* do bloco, em comparação com o *pixel* da área de referência. Logo após, o algoritmo desloca o bloco atual um *pixel* dentro da área de pesquisa e re-calcula a diferença. Esta operação é repetida para todas as posições possíveis dentro da área de pesquisa.
Alguns critérios de parada podem ser adotados para acelerar o processo do algoritmo de busca completa como, por exemplo, determinar uma diferença mínima, quando o algoritmo encontrar um valor abaixo desse mínimo ele pára sua execução. Vale lembrar que todas as técnicas de redução do número de cálculos do algoritmo de busca completa podem implicar na geração de um vetor de movimento que pode não ser o vetor ótimo.
A complexidade computacional do algoritmo de busca completa dificulta a sua aplicação em CODECs (codificadores/ decodificadores) que necessitam operar em tempo real para vídeos de alta resolução. Implementações em hardware podem minimizar a complexidade do algoritmo ao paralelizar as operações de comparação. Isto acelera o processo de estimação tornando viável a utilização deste algoritmo, que é uma ótima opção quando se deseja alta qualidade e taxa de compressão do vídeo comprimido.

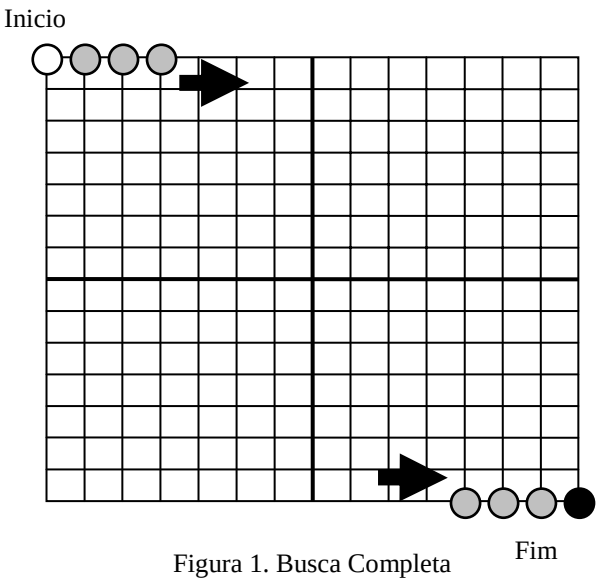


Figura 1. Busca Completa

Os próximos algoritmos que serão apresentados neste trabalho são considerados algoritmos rápidos (sub-ótimos), ideais para implementações de estimadores de movimento que operem em tempo real, principalmente em software. É importante salientar que estes algoritmos usam técnicas para diminuir o número de comparações dentro da área de pesquisa, o que pode desviar o rumo da pesquisa para uma região que não leve ao vetor ótimo. Mesmo assim, estes algoritmos podem encontrar regiões com erros muito próximos aos encontrados pelo algoritmo de busca completa, mesmo que gerando vetores de movimento bem diferentes.

3.2 *Three Step Search*

Este algoritmo é mais difundido com o nome de busca em três passos. No entanto, ele pode ser estendido para n passos [3]. Para uma área de pesquisa de +/- (2n – 1) *pixel*s, o passo inicial S deve ser inicializada com S = 2n-1.

O primeiro passo consiste em posicionar a busca no centro da área de pesquisa e calcular o erro para esta posição. Em seguida, calcular mais oito valores +/- S *pixel*s em torno da posição (0,0) (centro da área de pesquisa). Comparar os nove valores de erro obtidos e determinar como nova posição de origem a posição de menor erro.

No segundo passo a variável S é dividida por dois e novamente oito valores +/- S *pixel* em torno da origem são calculados. Desta vez, oito valores são comparados e, novamente, a origem será substituída pela posição com o menor erro. A Figura 2 ilustra o algoritmo *Three Step Search* para uma área de pesquisa de +/- 7 *pixel*s. Os pontos de melhor *matching* escolhidos a cada passo do algoritmo estão destacados em cinza.

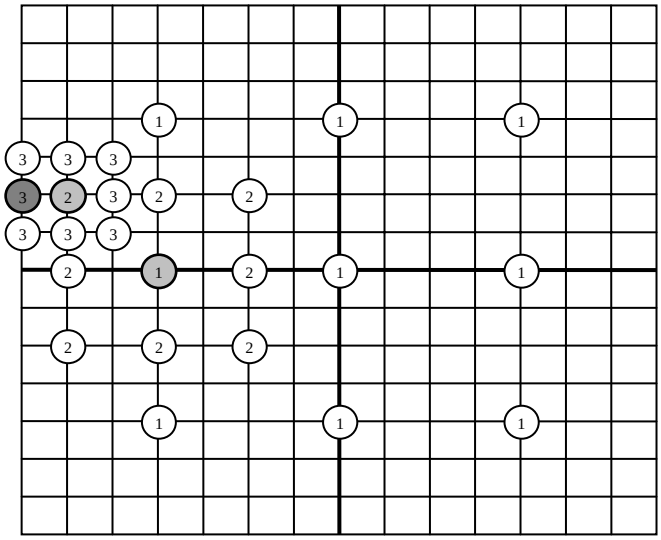


Figura 2. *Three Step Search*

No terceiro e último passo, mais uma vez a variável S é dividida por dois. Desta vez ela irá conter o valor um, o que indica o último estágio de busca. Mais uma vez, os oito valores de erros são comparados e o vetor de movimento será gerado para a posição com o menor valor de erro.
Após os três passos do algoritmo, 25 posições são comparadas (9 + 8 + 8). Em geral 8n + 1 comparações são necessárias para uma busca em uma área de pesquisa de +/- (2n-1). Se utilizássemos o algoritmo de busca completa, para a mesma área de pesquisa teríamos um total de 225 (15 x 15) comparações.

3.3 *One at a Time Search*
Este algoritmo é bastante simples e consiste na idéia de se encontrar primeiro um mínimo horizontal e, em seguida, um mínimo vertical [3]. O algoritmo começa calculando o erro para a posição central da área de pesquisa. Em seguida, mais dois valores, um imediatamente à direita e outro imediatamente à esquerda, são calculados. Caso o valor do centro seja o menor, o algoritmo passa para o estágio de busca vertical. Caso contrário, redefine-se o centro com a posição do menor erro e calcula-se o valor para o vizinho, seja ele à direita ou à esquerda (dependendo do valor escolhido no passo anterior). O estágio de cálculo dos valores verticais é muito semelhante, variando a busca para cima e para baixo. A Figura 3 ilustra uma busca de um algoritmo *One at a Time Search* em uma área de pesquisa de +/- 7 *pixels*.

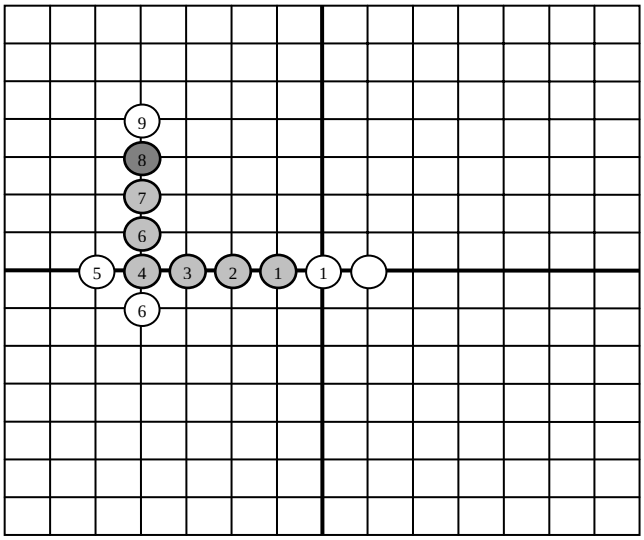


Figura 3. One at a time Search

No ponto número 5 o resultado do erro é maior que no ponto “4”, então o algoritmo encerra a busca horizontal e inicia a busca vertical. São calculados dois valores, um acima e outro abaixo da posição número “4”. O menor erro é encontrado acima. O algoritmo segue deslocando um *pixel* acima até que a posição número “9” resulta em um erro maior que a posição “8”. O algoritmo encerra a busca e gera o vetor para a posição “8”.
Para este algoritmo, não é possível definir o número exato de operações, pois isso dependerá diretamente das informações contidas na área de pesquisa e no bloco. Estas informações determinarão o número de cálculos na horizontal e na vertical.
Pode-se perceber, pelo reduzido número de operações, que este algoritmo é bastante rápido, no entanto, está muito suscetível a encontrar mínimos locais, ou seja, determinar o vetor de movimento para regiões bem próximas ao início da pesquisa, podendo desviar o rumo da pesquisa numa direção contrária a região de menor erro.

3.5 *Diamond Search*
O algoritmo *Diamond Search* possui dois padrões diamante que são usados na etapa inicial e final do algoritmo. A Figura 4 ilustra os padrões *Large Diamond Search Pattern* (LDSP) e o padrão *Small Diamond Search Pattern* (SDSP). O padrão LDSP consiste em 9 comparações e é utilizado na etapa inicial da pesquisa. Já o padrão SDSP consiste em 4 comparações e é utilizado na etapa final da pesquisa, com o intuito de refinar o resultado obtido na etapa anterior [5].

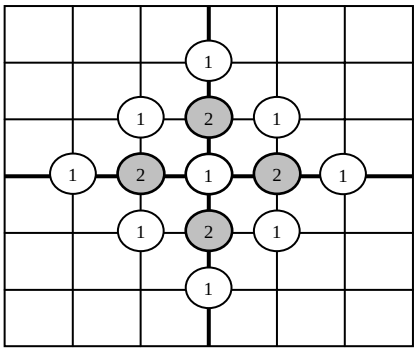


Figura 4. Large Diamond (LDSP) (1) e Small Diamond (SDSP) (2)

O algoritmo começa aplicando o padrão LDSP ao centro da área de pesquisa. Caso o valor de menor erro seja encontrado no centro, o algoritmo aplica o padrão SDSP para refinar o resultado obtido. Caso contrário, um novo padrão diamante é aplicado à posição de menor erro da etapa anterior. Esta posição pode pertencer a uma aresta ou a um vértice do diamante. As Figuras 5 e 6 representam, respectivamente, a busca por uma aresta e a busca por um vértice.

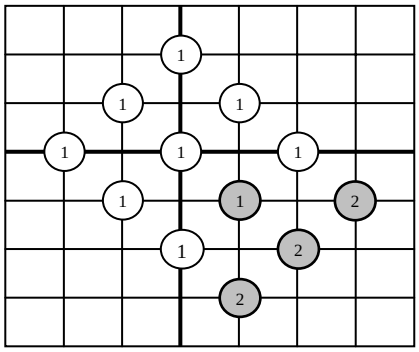


Figura 5. Busca por uma aresta

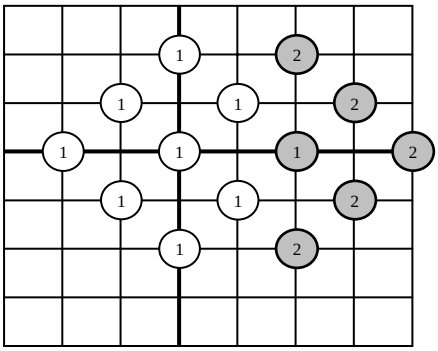
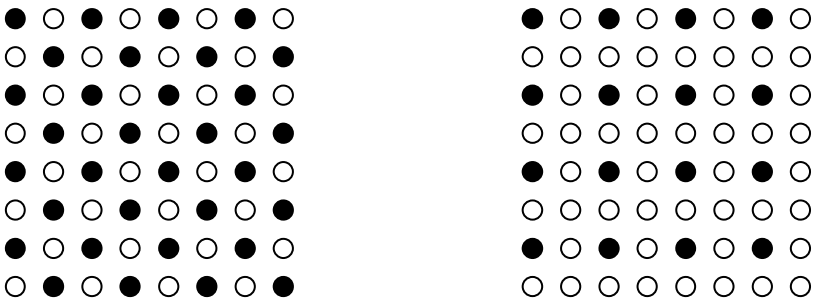


Figura 6. Busca por um vértice

No caso da busca por uma aresta, mais 3 valores são calculados para formar um novo diamante em torno da nova origem. Quando o novo centro é um vértice do diamante, mais 4 valores são calculados para formar o novo diamante em torno do centro. Caso o menor erro não seja encontrado no centro do novo diamante, a etapa de busca será repetida, seja ela por uma aresta ou por um vértice. Quando o menor erro for encontrado para o centro do diamante o padrão SDSP é aplicado. Então mais quatro valores imediatamente ao redor do centro serão calculados e a posição com o menor erro será a escolhida.
Novamente não se pode determinar o número de operações do algoritmo. Assim como o algoritmo de busca logarítmica, o algoritmo *Diamond Search* pode começar a sua busca em uma direção e desviar a pesquisa ao longo do processo, o que pode evitar que o algoritmo caia em um mínimo local.

3.6 *Pel Decimation*

A técnica de *Pel Decimation* (também chamada de *Pel Subsampling*) consiste em reduzir o número de cálculos realizados pelo critério de distorção, a cada comparação [5]. O *Pel Decimation* é baseado no algoritmo de busca completa, no entanto, a distorção não é calculada para todos os *pixel*s do bloco. Isto diminui o tempo de processamento e a complexidade computacional do algoritmo. A técnica de *Pel Decimation* geralmente é aplicada nas proporções 2:1 e 4:1. A Figura 7 ilustra a técnica para as proporções 2:1(a) e 4:1(b) para um bloco de 8x8 *pixel*s. Apenas as posições em preto são calculadas, as posições em branco são desconsideradas.
Com o algoritmo *Pel Decimation* pode-se reduzir significativamente o número de operações do algoritmo de busca completa. Neste caso, para um bloco de 8x8 *pixels*, a cada comparação, o algoritmo de busca completa deve realizar 64 operações (8x8). Para o exemplo da figura 8(a) o algoritmo deve realizar 32 operações (4x8) e para o exemplo da figura 8(b) o algoritmo deve realizar apenas 16 operações (4x4).



a) *Pel Decimation* 2:1 b) *Pel Decimation* 4:1

Figura 7. Técnica de *Pel Decimation*

Esta técnica reduz a complexidade computacional do algoritmo e o tempo de operação. No entanto, os vetores resultantes podem não ser os vetores ótimos. Mesmo comparando todas as posições da área de pesquisa, ao desconsiderar alguns elementos do bloco, o algoritmo pode conduzir a escolha de uma região que não gera o resultado ótimo.

4. FUNÇÃO DE SIMILARIDADE SAD

A função similaridade é a maneira como as diferenças entre as regiões comparadas são avaliadas. Os critérios mais utilizados são: o menor erro quadrático (MSE), o menor erro absoluto (MAE) e a soma das diferenças absolutas (SAD) (Richardson, 2002). Este trabalho apresenta resultados utilizando o SAD como função de similaridade, por sua maior simplicidade. A função para o cálculo do SAD é dada por (1), onde **Erro(x,y)** representa o erro para a posição (x,y); **R** representa o ponto da área de referência, **P** representa o ponto da área de pesquisa e **N** é o tamanho do bloco.

$$Erro(x,y) = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1}\left|R_{i,j} - P_{i+x,j+y}\right| \qquad (1)$$

5. RESULTADOS DAS IMPLEMENTAÇÕES EM C PARA OS ALGORITMOS ESTUDADOS

Neste trabalho foram implementados os algoritmos: *Full Search*, *Three Step Search*, *Diamond Search*, *One at a Time Search* e os algoritmos *Pel Decimation* 2:1 e 4:1. A linguagem escolhida foi C [Kernighan 99] e os algoritmos foram combinados com o critério SAD. Para todas as implementações, a área de pesquisa foi delimitada em +/- 15 amostras em torno do bloco de 16x16 amostras, ou seja, a área de pesquisa possui um tamanho total de 46x46 amostras. O vídeo utilizado como entrada foi o "src21_ref__525_480i@30.yuv" [Fedora 2005] que possui uma resolução de 720x480 *pixels*. Todos os algoritmos foram compilados no compilador DEV-C++ 4.9.9.2 [Bloodshed 2005] em um processador Intel Pentium 4, 3.2GHz com 1GB de RAM.

Os resultados de erro, apresentados neste artigo, são comparados com o valor do somatório da diferença *pixel* a *pixel* entre os primeiros 100 quadros da amostra. A Tabela 1 apresenta os resultados de erro utilizando o critério SAD. Como era esperado, todos os algoritmos geram um erro menor que o erro absoluto gerado entre os quadros. No entanto, deve-se destacar o resultado obtido pelo algoritmo *Full Search*, que proporcionou uma diminuição do erro da ordem de 65%. As duas versões do algoritmo *Pel Decimation* também obtiveram um bom resultado, com percentuais de diminuição acima de 59%. Os piores desempenhos, como também já era esperado, ficaram para os algoritmos que não realizam a busca em todas as posições da área de pesquisa.

Tabela 1. Resultados de erro para os algoritmos estudados

Algoritmo de Busca	Erro	Erro por Bloco	Diminuição do Erro(%)
Full search	74.282.817	550,24	65,15
Diamond search	124.357.591	921,17	41,77
Three Step search	92.714.430	686,77	56,50
One at a Time search	194.984.582	1.444,33	8,53
Pel Decimation 2:1	75.068.732	556,06	64,78
Pel Decimation 4:1	87.351.770	647,05	59,02

Estimação para 100 quadros de 720x480 *pixel*s – Pentium 4, 3.20 GHz, 1Gb RAM

A Tabela 2 apresenta os resultados de tempo de execução destas implementações. Mais uma vez, o algoritmo *Full Search* merece destaque, pois ele possui o maior tempo de execução entre todos os algoritmos apresentados. Entre os algoritmos mais lentos estão, também, as duas versões do algoritmo *Pel Decimation*. Os demais algoritmos possuem tempos de execução muitas vezes inferior, sendo que o melhor desempenho está com o algoritmo *One at a Time Search*.

Tabela 2. Resultados de tempo de execução

Algoritmo de Busca	Tempo (s)	Tempo por Bloco (ms)
Full search	419,9	3,11
Diamond search	8,9	0,065
Three Step search	7,7	0,057
One at a Time search	2,5	0,018
Pel Decimation 2:1	310,2	2,30
Pel Decimation 4:1	108,6	0,80

Estimação para 100 quadros de 720x480 *pixel*s – Pentium 4, 3.20 GHz, 1Gb RAM

Comparando os resultados obtidos nas Tabelas 1 e 2, é possível perceber que os algoritmos que resultam no menor erro tendem a ser os que possuem o pior desempenho. É importante destacar que estes resultados foram gerados para software rodando sobre um processador de propósito geral. Isso implica em uma execução seqüencial das instruções. Como os algoritmos *Full Search* e *Pel Decimation* não possuem dependência de dados, presente nos demais algoritmos [Kuhn 99] sua implementação em hardware pode explorar massivamente o uso de paralelismo quando da sua implementação em hardware. Além disso, a implementação destes algoritmos em hardware é facilitada em função da homogeneidade do processamento realizado. Considerando a possibilidade de explorar o paralelismo e considerando a pequena diferença de erro gerada pelos algoritmos *Full Search* e *Pel Decimation* 4:1, optou-se, neste trabalho, pelo desenvolvimento de uma arquitetura para o algoritmo *Pel Decimation* 4:1, utilizando SAD.

6. ARQUITETURA PROPOSTA

A arquitetura que foi desenvolvida neste trabalho é baseada em uma arquitetura desenvolvida na UFRN e que ainda não foi publicada. A arquitetura da UFRN foi desenvolvida utilizando o algoritmo *Full Search* e usando o SAD como critério de distância. Na arquitetura proposta neste artigo, o algoritmo de busca escolhido foi o *Pel Decimation* 4:1, também utilizando o critério SAD. A arquitetura desenvolvida na UFRN considera uma área de pesquisa de 32x32 amostras, com blocos de 16x16 amostras. A solução apresentada neste trabalho considera uma área de pesquisa de 64x64 amostras, ou seja, quatro vezes maior, com um bloco também de 16x16 amostras. A arquitetura desenvolvida na UFRN é capaz de atingir tempo real para resoluções elevadas, mas, para tanto, utiliza uma quantidade excessiva de recursos de hardware. Utilizando o *Pel Decimation* ao invés do *Full Search*, é possível diminuir a quantidade de recursos de hardware utilizados, mantendo o desempenho elevado. Além disso, é possível explorar características adicionais, como o aumento na área de busca.

O diagrama em blocos da arquitetura desenvolvida neste trabalho está apresentado na figura 8. A memória interna está organizada em 5 memórias distintas como está apresentado na figura 8. Uma memória é destinada ao armazenamento do bloco do quadro atual (com 8 palavras de 64 bits) e as outras 4 memórias são usadas para armazenar a área de pesquisa (cada uma com 32 palavras de 256 bits).

Quando o ME inicia seu funcionamento, o gerenciador de memória realiza a leitura de uma palavra de cada uma das memórias. Com isso, são acessadas uma linha da área de pesquisa e uma linha do bloco atual. Essas linhas são armazenadas nos registradores da área de pesquisa (RLP na figura 8) e nos registradores de bloco (RLB na figura 8). Nesse momento, o gerenciador envia um sinal que habilita o início das atividades do controle e dos seletores.

A unidade de processamento (UP na figura 8) calcula a distorção entre uma amostra do bloco do quadro atual e uma amostra da área de pesquisa. Cinco UPs são combinadas, formando uma linha de SAD. Um conjunto de 25 linhas de SAD forma uma matriz de SAD, como está apresentado na figura 8, realizando a busca completa sobre a área de busca do quadro de referência. O controle gerencia as operações e define em que estágio do pipeline cada uma das linhas da matriz de SADs está operando. Depois de finalizado o processamento de toda a região de busca, o comparador recebe os valores de distorção de todos os blocos candidatos e escolhe aquele com menor valor, gerando o vetor de movimento para este bloco.

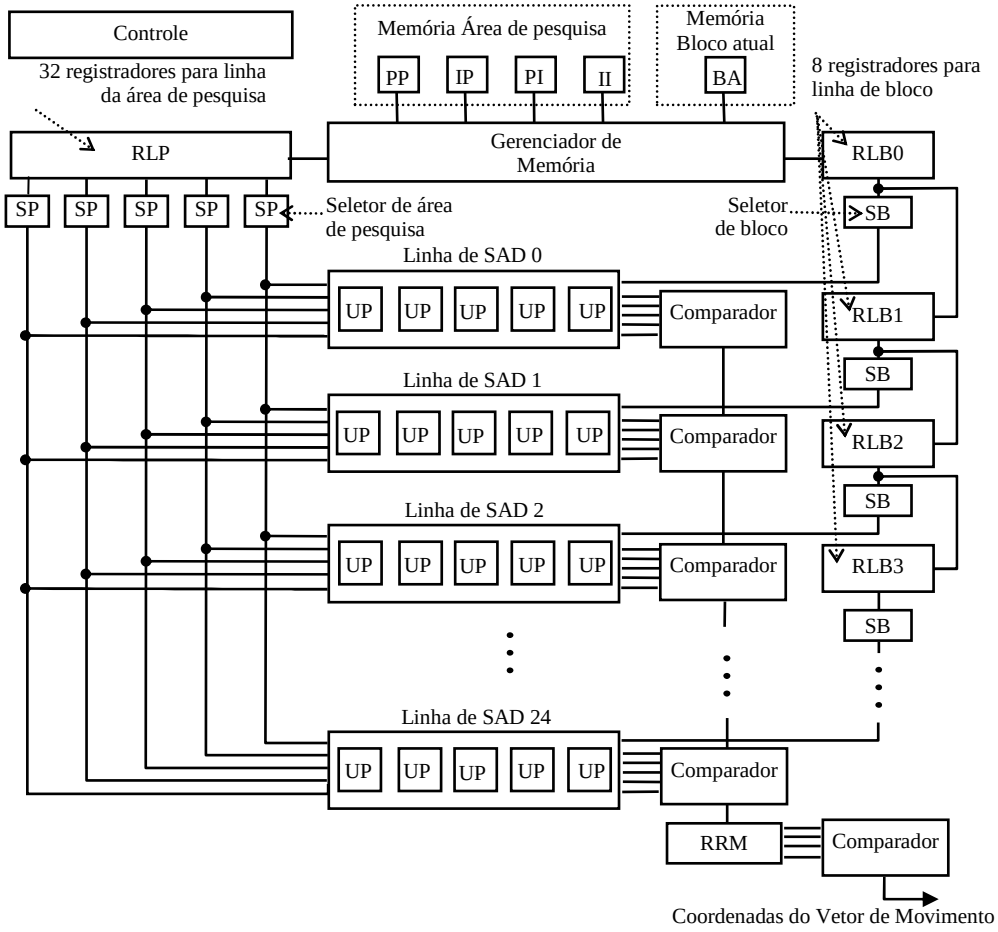


Figura 8. Diagrama em blocos da arquitetura do estimador

A arquitetura para o cálculo do SAD foi construída hierarquicamente. A instância de mais elevado nível hierárquico é a matriz de SADs, que é formada por 25 linhas de SAD, que, por sua vez, são formadas por cinco unidades de processamento (UPs), como está apresentado na figura 8.

A Figura 9 apresenta a arquitetura simplificada de uma UP. Cada UP recebe como entrada duas amostras do bloco atual (B0 a B1 na figura 8) e duas amostras do bloco candidato (R0 a R1 na figura 8). Isso significa que cada UP calcula, paralelamente, a similaridade de ¼ de linha do bloco candidato em relação ao bloco atual. Os módulos dos resultados das subtrações são somados para gerar um valor único. O resultado representa o módulo do erro entre os dois blocos para as duas primeiras amostras, ou seja, para ¼ da primeira linha dos blocos.

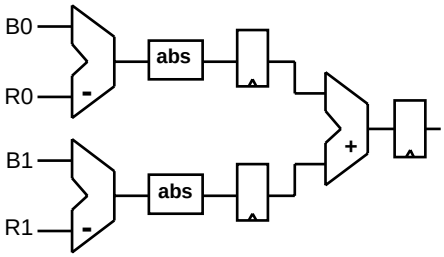


Figura 9. Diagrama em blocos de uma UP

Na arquitetura original, cada UP processava ½ linha, para o bloco candidato. Esta opção de projeto acelera o cálculo do SAD para o bloco, pois apenas duas realimentações devem ser realizadas para concluir o SAD de uma linha, no entanto, uma grande quantidade de recursos de hardware são necessários para sua implementação. Ao utilizar o algoritmo *Pel Decimation* 4:1, o tamanho da linha cai pela metade (8 amostras devem ser comparadas ao invés de 16), o que reduz o tamanho da arquitetura da UP. Nesta proposta optamos por processar ¼ da linha de cada bloco candidato,

para diminuir ainda mais o consumo de recursos. Então, são necessárias duas vezes mais realimentações para finalizar o cálculo do SAD.
O SAD parcial do bloco candidato (¼ de linha) gerado pela UP deve ser armazenado e adicionado aos SADs das demais partes do bloco candidato para gerar o SAD total deste bloco, que é formado por 8 linhas com 8 amostras em cada linha (64 SADs devem ser acumulados). A linha de SADs, apresentadas na figura 4, agrupa cinco UPs e realiza a acumulação para gerar o valor final do SAD dos blocos candidatos. Cada UP é responsável pelo cálculo, em tempos distintos, da similaridade de diferentes blocos candidatos. Com esta solução, as UPs processam os SADs relativos a cinco diferentes blocos candidatos. Então, uma linha de SADs calcula, em paralelo e com pipeline, o SAD de 25 diferentes blocos candidatos.
Na Figura 10 estão apresentadas as quatro UPs (destacadas em cinza) e estão presentes os acumuladores utilizados para guardar o valor parcial e final dos cálculos do SAD de cada bloco. Como cada UP processa uma linha do bloco em quatro etapas, então cada UP gera 32 resultados parciais de SAD para cada bloco processado. Estes 32 resultados parciais devem ser somados para gerar o SAD final do bloco.

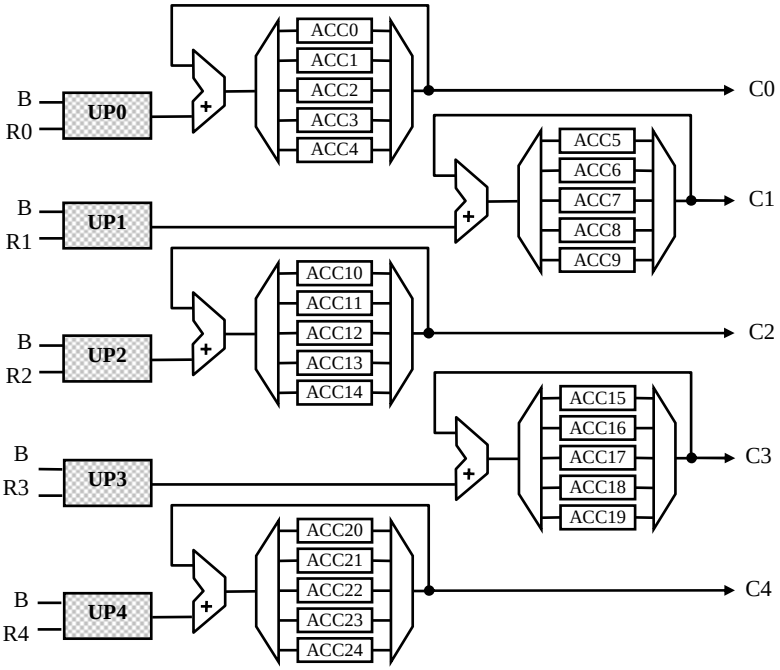


Figura 11. Arquitetura para uma linha de SAD

Como os resultados de um bloco não são completamente gerados em paralelo, uma simples estrutura de somador e registrador de acumulação é suficiente para fazer esta totalização. Como as UPs calculam, em paralelo, o SAD de mais de um bloco, então um registrador é utilizado para armazenar o SAD de cada bloco (ACC0 a ACC24 na figura 10) e um par demultiplexador e multiplexador é necessário para controlar o acesso correto aos registradores, como está apresentado na figura 4. Quando uma linha de SADs concluir seus cálculos, então os registradores ACC0 a ACC24 da figura 10 irão conter os SADs finais dos 25 blocos candidatos.
O comparador recebe a saída das linhas de SAD, e pode realizar cinco comparações de SADs em paralelo, em um pipeline de cinco estágios. As cinco saídas da linha de SADs (C0 a C4 na Figura 10) entregam para o comparador cinco valores de SADs de blocos candidatos em cada ciclo de *clock* e, em cinco ciclos, todos os 25 valores de SAD de uma linha de SAD já foram entregues ao comparador. Quando o menor valor é encontrado, então o comparador compara este valor de SAD com o menor valor dentre os SADs da linha de SAD anterior. A ligação entre os diversos comparadores pode ser observada na figura 8. Então, o menor SAD dentre todos os SADs calculados até então, e o vetor de movimento do bloco candidato que gerou este menor SAD, são disponibilizados na saída para o comparador da próxima linha de SADs.

7. RESULTADOS DE SÍNTESE

A arquitetura do estimador de movimento foi descrita em VHDL [9] e sintetizada no FPGA Virtex-II Pro XC2VP70 da Xilinx [10], utilizando a ferramenta ISE, também da Xilinx [10]. A Tabela 3 ilustra os resultados de síntese. Um resultado interessante é que a matriz de SAD utiliza mais elementos lógicos (LUTs para o Virtex-II Pro) do que o

estimador completo. Isto se deve ao fato de que a matriz de SAD possui centenas de pinos de entrada a mais que o estimador, pois neste bloco as memórias não estão presentes e, então, as entradas das amostras da área de pesquisa e do bloco atual passam a ser pinos externos, exigindo uma alocação de LUTs para fazer o roteamento necessário.

Tabela 3. Resultados de síntese do estimador de movimento

Bloco	LUTs	Freq. (MHz)	Mem. Bits
Controle Global	157	267,2	-
Controle Local	23	290,6	-
UP	45	381,9	-
Linha de SAD	350	357,0	-
Comparador	235	224,6	-
Gerenciador de Memória	605	284,0	32.286
Matriz de SAD	27.265	173,9	-
Estimador	19.695	228,0	32.286

Dispositivo Virtex-II Pro XC2VP70

Um dos resultados mais importantes da Tabela 3 é a freqüência de operação da arquitetura do estimador. Com a freqüência de 228MHz, e gerando um novo vetor a cada 2.615 ciclos, a arquitetura pode gerar mais de 87 mil vetores de movimento por segundo. Considerando a relação de sub-amostragem 4:2:0 [3], muito comum em vídeo digital, para cada quadro informações de luminância (Y), existe uma informação de crominância Cb e outra de crominância Cr. Considerando esta relação e considerando que cada vetor de movimento se refere a um bloco de 16x16 amostras de luminância e dois blocos de 8x8 amostras de crominância, mais de 33 milhões de amostras podem ser processadas a cada segundo por esta arquitetura.

Para melhor visualizar estas informações, a Tabela 4 mostra a taxa de processamento do estimador para alguns dos padrões de vídeo existentes. Os resultados mostram que a arquitetura desenvolvida neste trabalho pode atingir tempo real para a maioria dos padrões de vídeo existentes, exceto para HDTV (1920 x 1080), onde obteve uma taxa de processamento inferior a 30 quadros por segundo.

Tabela 4. Taxa de processamento da arquitetura

Padrão	Quadros por Segundo
QCIF (176 x 144)	880,3
CIF (352 x 268)	220,0
VGA (640 x 480)	72,6
SDTV (720 x 480)	64,7
HDTV (1920 x 1080)	10,7

Outro resultado importante foi que a arquitetura completa do estimador utilizou 19,7 mil elementos lógicos, isto implica em cerca de 28% dos elementos lógicos do dispositivo alvo. Este é um bom resultado, tendo em vista que a arquitetura opera sobre uma área de pesquisa de 64x64 amostras.

A arquitetura desenvolvida na UFRN (referência para este trabalho) pode operar a uma freqüência de 172,1MHz, podendo processar mais de 132 milhões de amostras por segundo. Este resultado possibilita a utilização da arquitetura para aplicações HDTV em tempo real. No entanto esta arquitetura opera sobre uma área de pesquisa de 32x32 amostras e, mesmo assim, utiliza mais de 37,5 mil elementos lógicos, o que resulta em mais de 50% dos recursos de hardware do FPGA alvo. A arquitetura apresentada neste trabalho possui uma área de pesquisa quatro vezes maior do que a arquitetura da UFRN, mas em função da utilização do algoritmo *Pel Decimation* 4:1 e de simplificações na UP, foram utilizadas 19,7 mil LUTs, quase metade do que foi utilizado na arquitetura da UFRN. A arquitetura desenvolvida neste artigo atingiu uma freqüência de operação de 228MHz e, em função das realimentações adicionais que foram inseridas nas UPs (duas vezes mais realimentações) e na matriz de SAD (quatro vezes mais realimentações), esta arquitetura atingiu uma taxa de processamento de 33 milhões de amostras por segundo, significativamente inferior a taxa atingida na arquitetura da UFRN.

Cabe destacar que o acréscimo de quatro vezes na área de pesquisa, mantendo o tamanho de bloco em 16x16, gerou um acréscimo significativo no número de blocos candidatos da área de pesquisa e, por conseqüência, no número de cálculos realizados. Na arquitetura apresentada neste artigo são 2401 blocos candidatos, enquanto que na arquitetura da

UFRN, são 289 blocos candidatos. Isso significa que, para cada bloco do quadro atual, são 8,3 vezes mais blocos candidatos que devem ser calculados.
Do ponto de vista de taxa de processamento, é importante salientar que mesmo com uma taxa de processamento inferior, a arquitetura desenvolvida neste trabalho atinge um desempenho elevado, sendo suficiente para codificar vídeos SDTV (720 x 480 *pixel*s) em tempo real. Além disso, a área de busca foi quadruplicada e mesmo assim os recursos de hardware utilizados foram reduzidos quase pela metade em relação à solução base. Utilizando outras possibilidades de implementação, como ASIC por exemplo, ou até mesmo uma nova geração de FPGAs, esta mesma arquitetura pode alcançar a taxa necessária para processar vídeos HDTV em tempo real.

8. CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho teve como objetivo principal investigar algoritmos de estimação de movimento e implementá-los em software para avaliar as suas características e desenvolver uma arquitetura de hardware. A arquitetura desenvolvida neste trabalho foi baseada em uma arquitetura desenvolvida na UFRN e utilizou o algoritmo *Pel Decimation* 4:1 com SAD. A arquitetura desenvolvida trabalha com blocos de 16x16 amostras e opera sobre uma área de pesquisa de 64x64 amostras. Os resultados de síntese indicam que esta arquitetura é capaz de gerar mais de 87 mil vetores de movimento por segundo, o que implica no processamento de mais de 33 milhões de amostras a cada segundo. Com estes resultados de desempenho a arquitetura desenvolvida pode codificar com folga vídeos SDTV em tempo real. Esta taxa de processamento é muito inferior do que a obtida na arquitetura de referência (UFRN) e esta diferença é causada, principalmente pela quadruplicação da área de pesquisa e pelo aumento no número de realimentações de dados, que foram geradas no intuito de diminuir a quantidade de recursos de hardware utilizados. Este objetivo foi atingido, pois houve uma redução de mais de 47% de recursos em relação à arquitetura da UFRN.
Como trabalho futuro, pretende-se aumentar o paralelismo da UP para que processe ½ da linha do bloco, ao invés de ¼. Esta modificação permitirá dobrar o desempenho da arquitetura e, com mais alguns ajustes arquiteturais, é provável que esta solução atinja o desempenho exigido para HDTV (1920 x 1080 *pixels*) em tempo real.

Referencias

[1] ISO/IEC JTC1. Generic coding of moving pictures and associated audio information – Part 2: Video. ISO/IEC 13818-2 (MPEG-2). Nov. 1994.

[2] ITU-T. Advanced Video Coding for Generic Audiovisual Services. ITU-T Recommendation H.264. May, 2003.

[3] Richardson, I. Video Codec Design – Developing Image and Video Compression Systems. Chichester: John Wiley and Sons, 2002.

[4] Lin, C.; Leou, J. An Adaptative Fast Full Search Motion Estimation Algorithm for H.264. In: ISCAS 2005 - IEEE INTERNATIONAL SYMPOSIUM CIRCUITS AND SYSTEMS. Proceedings… Kobe: IEEE, 2005, p. 1493-1496.]

[5] Kuhn, Peter et al. Algorithms, Complexity Analysis and VLSI Architectures for MPEG-4 Motion Estimation. Boston: Kluwer Academic Publisher, 1999. 239p.

[6] Kernighan, Brian W.; Ritchie, Dennis M. C: a Linguagem de Programação Padrão Ansi. Rio de Janeiro: Campus, 1999.

[7] Fedora Core Test Page, http://ginfo05.dee.ime.eb.br/, Acesso em out 2005.

[8] Bloodshed Software – DEV-C++ , “Providing Free Software to the Internet Community” <http://www.bloodshed.net/devcpp.html>, Acesso em out. 2005.

[9] Ashenden, Peter J. The Student’s Guide to VHDL. San Francisco: Morgan Kaufmann, 1998. 312p

[10] Xilinx INC. Xilinx: “The Programmable Logic Company”. Disponível em: <www.xilinx.com>. Acesso em: jan. 2006.