UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

EDUARDA RODRIGUES MONTEIRO

Implementação e Análise de Algoritmos para Estimação de Movimento em Processadores Paralelos tipo GPU (*Graphics Processing Units*)

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Prof. Dr. Sergio Bampi
Orientador

Porto Alegre, abril de 2012.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Monteiro, Eduarda Monteiro

Implementação e Análise de Algoritmos para Estimação de Movimento em Processadores Paralelos tipo GPU (*Graphics Processing Units*) / Eduarda Rodrigues Monteiro – Porto Alegre: Programa de Pós-Graduação em Computação, 2012.

114p.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2012. Orientador: Sergio Bampi.

1. Compressão de Vídeo 2. Estimação de Movimento 3. Processamento Paralelo. 4. CUDA I. Bampi, Sergio. II. Título.

AGRADECIMENTOS

Meus primeiros agradecimentos vão para minha família, principalmente para meus pais e minhas irmãs que, incondicionalmente, me apoiaram nas minhas decisões e sustentaram minha formação durante todos estes anos. Mesmo que às vezes eu não tenha demonstrado isso a eles, queria que soubessem que sou muito grata por tudo. Aos demais familiares, que de alguma forma, contribuíram para minha formação, também vai um agradecimento.

Gostaria de agradecer ao meu orientador Sergio Bampi todas as oportunidades e pelos conhecimentos compartilhados comigo durante todos este período. Em especial, agradeço pela confiança depositada quando ingressei no mestrado, esta oportunidade vai gerar diversos frutos na minha caminhada.

Agradeço também aos meus amigos e colegas que contribuíram para minha formação com os momentos de descontração e de companheirismo foram muito importantes. Aos colegas: Bruno Vizzotto, Bruno Zatt, Cláudio Diniz, Cristiano Thiele, Daniel Palomino, Felipe Sampaio, Kleber Hugo, André del Mestre, Eduardo Chiele, Lenna, Leonardo Soares, Eduardo Cruz, João Lima, Carlos Bruno, Antônio Lucas e Catia Souza. Muito obrigado a todos, cada um teve uma participação especial nesta jornada e o apoio deles foi fundamental para a realização deste trabalho.

Tenho que agradecer também ao grupo de Processamento Paralelo e Distribuído da UFRGS, o GPPD, o qual me proporcionou os recursos computacionais que viabilizaram o desenvolvimento do meu trabalho.

Enfim gostaria de agradecer a todos os meus amigos, em especial Alexandre Hubner, Elizabeth Farias e Felipe Munhoz que também tiveram um papel fundamental nesta formação.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
ASIC	*Application Specific Integrated Circuit*
AVC	*Advanced Video Coding*
BITSTREAM	Sequência de bits do vídeo codificado
Cb	*Chrominance Blue*
CIF	*Common Intermediate Format*
Cr	*Chrominance Red*
CUDA	*Compute Unified Device Architecture*
DCS	*Dual Cross Search*
DS	*Diamond Search*
FPS	*Frames per Second*
FS	*Full Search*
FSS	*Five Step Search*
GB	*Gigabytes*
GOPS	*Giga Floating Point Operations per Second*
GPU	*Graphics Processing Unit*
GPGPU	*General-Purpose Computing on Graphics Processing Units*
GPPD	Grupo de Processamento Paralelo e Distribuído
HD	*High Definition*
HEVC	*High Efiiciency Video Coding*
HDTV	*High Definition Digital Television*
HIS	*Hue, Saturation, Intensity*
HS	*Hexagon Based Search*
II	Instituto de Informática
INTRA	Predição Intra-Quadro
INTER	Predição Inter-Quadros
INRIA	*Institut National de Recherche em Informatique et em Automatique*

ISO	*International Organization for Standardization*
ITU	*International Telecommunication Union*
JPEG	*Joint Photographic Experts Group*
JVT	*Joint Video Team*
LDSP	*Large Diamond Search Pattern*
MAE	*Mean Absolute Error*
MC	*Motion Compensation*
ME	*Motion Estimation*
MIMD	*Multiple Instruction Multiple Data*
MPEG	*Motion Picture Experts Group*
MPI	*Message Passing Interface*
MS	Milissegundo
MSE	*Mean-Squared Error*
OPENMP	Open Multi-Processing
OTS	*One at a Time Search*
PSNR	*Peak Signal-to-Noise Ratio*
RGB	*Red, Green, Blue*
QCIF	*Quarter Common Intermediate Format*
QP	*Quantization Parameter*
SAD	*Sum of Absolute Differences*
SATD	*Sum of Absolute Transformed Differences*
SDSP	*Small Diamond Search Pattern*
SIMD	*Single Instruction Multiple Data*
SIMT	*Single Instruction Multiple Threads*
SMPUMHEXAGONS	*Simplified Unsymmetrical Multihexagon Search*
SSS	*Six Step Search*
SVC	*Scalable Video Coding*
VGA	*Video Graphics Array*
VLC	*Variable Length Coding*
VLSI	*Very Large Scale Integration*
TSS	*Three Step Search*
TV	*Digital Television*
UFRGS	Universidade Federal do Rio Grande do Sul
WQXGA	*Wide Quad eXtended Graphics Array*
Y	*Luminance*

YCbCr *Luminance, Chrominance Blue, Chrominance Red*

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A demanda por aplicações que processam vídeos digitais têm obtido atenção na indústria e na academia. Considerando a manipulação de um elevado volume de dados em vídeos de alta resolução, a compressão de vídeo é uma ferramenta fundamental para reduzir a quantidade de informações de modo a manter a qualidade viabilizando a respectiva transmissão e armazenamento. Diferentes padrões de codificação de vídeo foram desenvolvidos para impulsionar o desenvolvimento de técnicas avançadas para este fim, como por exemplo, o padrão H.264/AVC. Este padrão é considerado o estado-da-arte, pois proporciona maior eficiência em codificação em relação a padrões existentes (MPEG-4). Entre todas as ferramentas inovadoras apresentadas pelas mais recentes normas de codificação, a Estimação de Movimento (ME) é a técnica que provê a maior parcela dos ganhos. A ME busca obter a relação de similaridade entre quadros vizinhos de uma cena, porém estes ganhos são obtidos ao custo de um elevado custo computacional representando a maior parte da complexidade total dos codificadores atuais. O objetivo do trabalho é acelerar o processo de ME, principalmente quando vídeos de alta resolução são codificados. Esta aceleração concentra-se no uso de uma plataforma massivamente paralela, denominada GPU (*Graphics Processing Unit*). Os algoritmos da ME apresentam um elevado potencial de paralelização e são adequados para implementação em arquiteturas paralelas. Assim, diferentes algoritmos têm sido propostos a fim de diminuir o custo computacional deste módulo. Este trabalho apresenta a implementação e a exploração do paralelismo de dois algoritmos da ME em GPU, focados na codificação de vídeo de alta definição e no processamento em tempo real. O algoritmo *Full Search* (FS) é conhecido como algoritmo ótimo, pois encontra os melhores resultados a partir de uma busca exaustiva entre os quadros. O algoritmo rápido *Diamond Search* (DS) reduz significativamente a complexidade da ME mantendo a qualidade de vídeo próxima ao desempenho apresentado pelo FS. A partir da exploração máxima do paralelismo dos algoritmos FS e DS e do processamento paralelo disponível nas GPUs, este trabalho apresenta um método para mapear estes algoritmos em GPU, considerando a arquitetura CUDA (*Compute Unified Device Architecture*). Para avaliação de desempenho, as soluções CUDA são comparadas com as respectivas versões *multi-core* (utilizando biblioteca OpenMP) e distribuídas (utilizando MPI como infraestrutura de suporte). Todas as versões foram avaliadas em diferentes resoluções e os resultados foram comparados com algoritmos da literatura. As implementações propostas em GPU apresentam aumentos significativos, em termos de desempenho, em relação ao software de referência do codificador H.264/AVC e, além disso, apresentam ganhos expressivos em relação às respectivas versões *multi-core*, distribuída e trabalhos GPGPU propostos na literatura.

Palavras-Chave: Estimação de Movimento, *Full Search*, *Diamond Search*, GPU, CUDA**.**

Implementation and Analysis of Algorithms for Motion Estimation onto Parallels Processors type GPU

ABSTRACT

The demand for applications processing digital videos has become the focus of attention in industry and academy. Considering the manipulation of the high volume of data contained in high resolution digital videos, video compression is a fundamental tool for reduction in the amount of information in order to maintain the quality and, thus enabling its respective transfer and storage. As to obtain the development of advanced video coding techniques, different standards of video encoding were developed, for example, the H.264/AVC. This standard is considered the state-of-art for proving high coding efficiency compared to previous standards (MPEG-4). Among all innovative tools featured by the latest video coding standards, the Motion Estimation is the technique that provides the most important coding gains. ME searches obtain the similarity relation between neighboring frames of the one scene. However, these gains were obtained by the elevated computational cost, representing the greater part of the total complexity of the current encoders. The goal of this project is to accelerate the Motion Estimation process, mainly when high resolution digital videos were encoded. This acceleration focuses on the use of a massively parallel platform called GPU (Graphics Processing Unit). The Motion Estimation block matching algorithms present a high potential for parallelization and are suitable for implementation in parallel architectures. Therefore, different algorithms have been proposed to decrease the computational complexity of this module. This work presents the implementation and parallelism exploitation of two motion estimation algorithms in GPU focused in encoding high definition video and the real time processing. Full Search algorithm (FS) is known as optimal since it finds the best match by exhaustively searching between frames. The fast Diamond Search algorithm reduces significantly the ME complexity while keeping the video quality near FS performance. By exploring the maximum inherent parallelism of FS and DS and the available parallel processing capability of GPUs, this work presents an efficient method to map out these algorithms onto GPU considering the CUDA architecture (Compute Unified Device Architecture). For performance evaluation, the CUDA solutions are compared with respective multi-core (using OpenMP library) and distributed (using MPI as supporting infrastructure) versions. All versions were evaluated in different video resolutions and the results were compared with algorithms found in the literature. The proposed implementations onto GPU present significant increase, in terms of performance, in relation with the H.264/AVC encoder reference software and, moreover, present expressive gains in relation with multi-core, distributed versions and GPGPU alternatives proposed in literature.

Keywords: Motion Estimation, Full Search, Diamond Search, GPU, CUDA.

1 INTRODUÇÃO

Atualmente, o mercado de aparelhos multimídia que processam vídeos digitais está em crescente expansão. Exemplos disso são as mais novas tecnologias para diversas aplicações, como TV digital, videoconferência e videovigilância. Estes tipos de aplicações, frequentemente voltadas para vídeos de alta definição, tem se tornado cada vez mais presentes na comunidade a partir de diferentes dispositivos, desde reprodutores de vídeos portáteis e telefones celulares até televisores digitais. Visando a boa qualidade visual destas aplicações, onde há uma grande quantidade de dados a serem transmitidos e/ou armazenados, a codificação de vídeo torna-se indispensável e é um assunto em evidência, tanto no meio acadêmico/científico quanto na indústria.

Desde o primeiro padrão de codificação de vídeo, o ITU-T H.261 (ITU, 1990), até o padrão de codificação mais recente, ITU-T H.264/ISO MPEG-4 *Part 10 Advanced Video Coding* (AVC) (ITU, 2010), uma considerável eficiência[1] neste processo tem sido atingida a partir da inclusão de diferentes técnicas de codificação avançadas. O padrão H.264/AVC, desenvolvido pelo JVT (*Joint Video Team*) (JVT, 2003) é considerado o estado da arte em codificação de vídeo digital e introduz ganhos significativos em relação à taxa de compressão quando comparado com outros padrões existentes (MPEG-2, entre outros).

O interesse de empresas e grupos de pesquisa por sistemas capazes de fornecer/transmitir vídeos de alta resolução em maior qualidade de vídeo, associados a requisitos de tempo real[2] e baixo consumo de potência é crescente. Neste cenário, o desenvolvimento de arquiteturas específicas para codificação de vídeos tem sido impulsionado considerando que os processadores de propósito gerais não são mais capazes de lidar com todos estes requisitos citados. Assim, diferentes plataformas vêm sendo utilizadas para implementação de codificadores de vídeo: ***(i)*** ASICs (*Application Specific Integrated Circuits*); ***(ii)*** FPGAs (*Field Programmable Gate Array*; ***(iii)*** ASIPs (*Application Specific Instruction-Set Processor*) e DSPs (*Digital Signal Processor*).

1.1 Escopo do trabalho

Dentre todas as técnicas, arquiteturas e ferramentas introduzidas pelos recentes padrões de codificação de vídeo, a Estimação/Compensação de Movimento (ME/MC), provêm os ganhos mais significativos na compressão do vídeo. Estes ganhos são obtidos a partir da exploração da redundância temporal entre quadros, a qual tem como objetivo

[1] Relação entre o número de bits de um vídeo cru (vídeo não codificado) e o número de bits do vídeo codificado.

[2] Uma taxa de reprodução de quadros para aplicações de codificação de vídeos de alta resolução é de 30 quadros por segundos (*frames per second* - fps), ou mais.

reduzir informações semelhantes entre os quadros vizinhos que constituem um vídeo digital. A Estimação de Movimento, presente nos codificadores de vídeo, emprega o uso de algoritmos de busca para encontrar em quadros de referência (quadros previamente codificados) uma região que mais se assemelha com uma região no quadro atual. Como resultado desta técnica, têm-se vetores de movimento os quais apontam para as regiões mais similares no quadro de referência. Por outro lado, a Compensação de Movimento, presente tanto nos codificadores quanto nos decodificadores[3], recebe o vetor de movimento e o número do quadro de referência para obter a região.

No processo de codificação de vídeo, os ganhos obtidos pela ME são atingidos ao custo de uma significativa complexidade computacional, quando comparada à complexidade computacional de um codificador completo. Na Figura 1.1 são apresentados dados de tempo de execução da ME (em porcentagem, referentes ao tempo de execução completo de um codificador de vídeo), utilizando o *software* de referência do padrão H.264/AVC, o JM 18.2 (JM, 2012). A ME foi simulada tanto com um algoritmo de busca exaustivo (Figura 1.1- A), quanto com um algoritmo rápido de busca (Figura 1.1 - B) para o perfil *Baseline*[4]. As simulações realizadas envolveram três sequências de vídeo (formato YUV) em diferentes resoluções: *Foreman* - CIF (*352x288*), *Mobcal* - HD720p (*1280x720*) e *Blue Sky* - HD1080p (*1920x1080*).

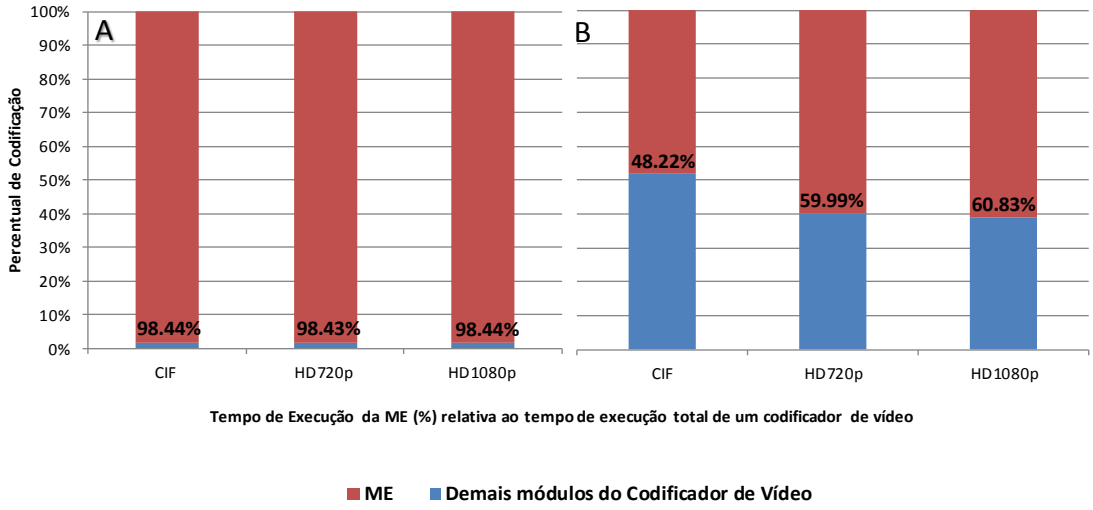


Figura 1.1: Custo Computacional da ME *x* Outras técnicas – Algoritmo Exaustivo (A) e Algoritmo Rápido(B).

O elevado custo computacional introduzido pela etapa de Estimação de Movimento (em vermelho) em relação às demais técnicas (em azul) que compõe o codificador de vídeo considerado pode ser observado na Figura 1.1. Para o algoritmo exaustivo (A) e para o algoritmo rápido (B) a complexidade computacional da ME se mantém significativa, representando mais de 98% e 40% do tempo da codificação, respectivamente. Este custo considerável é devido ao processo de busca pelo melhor

[3] Isto também é presente nos codificadores (o par MC/ME), para evitar possíveis diferenças entre operações de codificação e decodificação.

[4] Perfil do padrão H.264/AVC projetado para minimizar a complexidade proporcionando elevada robustez e flexibilidade.

casamento (do inglês, *best match*) entre os blocos que compõem os quadros atual e de referência.

Em geral, todos os padrões de codificação de vídeo restringem suas padronizações na definição do processo de decodificação de vídeo e a sua sintaxe do *bitstream* (fluxo de bits que sinalizam os quadros codificados), de modo a incumbir ao codificador a responsabilidade de gerar o *bitstream* de acordo com o padrão em questão. Por esta razão, os algoritmos de busca não são objetos de padronização, fato este que tem sido um tema de pesquisa muito ativo na comunidade nos últimos anos. Esta questão continua em aberto, uma vez que o algoritmo de busca deve apresentar um equilíbrio (custo/benefício) entre fatores muito importantes: qualidade de vídeo, complexidade (custo) computacional e taxa de *bits*.

Dentre diferentes algoritmos existentes para Estimação de Movimento, o algoritmo *Full Search* (FS) é conhecido como algoritmo ótimo por sempre encontrar o melhor casamento, visto que todas as possibilidades de casamento (com a granularidade de *pixel*) são analisadas. Esta busca, realizada em um ou mais quadros de referência, pode ser restringida a uma região da imagem (denominada área de busca), assumindo que os movimentos de objetos pertencentes a uma cena tendem a ser restritos dentro de um determinado limite de um quadro para o outro. O uso de uma área de busca no algoritmo FS pode apenas diminuir a sua complexidade, mas ainda assim mantendo um custo computacional elevado. Por exemplo, considerando uma área de busca de *128x128 pixels* divididos em blocos de tamanho *4x4 pixels*, existem 15.625 blocos a serem avaliados (chamados de blocos candidatos).

O algoritmo FS é considerado um algoritmo altamente regular o qual apresenta um elevado potencial para o processamento paralelo e distribuído, onde o processamento da ME para um bloco atual não possui dependência de dados com blocos vizinhos. Ao longo dos anos, a ME (e demais módulos de codificação de vídeo em geral) tem sido frequentemente implementada em circuitos VLSI (*Very Large Scale Integration*) a fim de alcançar uma alta taxa de processamento voltada para aplicações de tempo real, alta resolução e baixo consumo de potência. Esta é a solução mais eficiente em termos de desempenho, mas este fato envolve um considerável custo para projetar e fabricar circuitos integrados, o que se torna rentável apenas para um elevado volume de produção.

Por todas estas razões mencionadas, algoritmos rápidos têm sido propostos (KUHN, 1999), (PORTO, 2012) na literatura para diminuir a complexidade computacional em relação a um algoritmo exaustivo, ao custo de uma redução na qualidade de vídeo e eficiência na codificação. Assim, um dos principais objetivos de algoritmos rápidos é tentar prover uma boa relação entre qualidade de vídeo (próxima da qualidade apresentada pelo FS) e complexidade computacional (significativamente menor que o custo apresentado pelo FS). Um dos algoritmos rápidos mais conhecidos é o *Diamond Search* (DS), classificado como algoritmo sub-ótimo. Este algoritmo possui dois padrões de busca em forma de diamante, o *Large Diamond Search Pattern* (LDSP) e o *Small Diamond Search Pattern* (SDSP), usados na etapa inicial e final, respectivamente. O LDSP possui nove comparações no total ao redor do centro da área de busca. Por outro lado, o SDSP é usado para refinar os resultados, consistindo em quatro comparações. Primeiro o LDSP é aplicado, e somente quando o melhor casamento corresponder ao ponto central do diamante o refinamento (SDSP) é aplicado para finalização do algoritmo.

A elevada complexidade da ME é um dos principais fatores que impactam no desempenho dos codificadores atuais de vídeo. Para aplicações multimídias com restrições de processamento em tempo real, questões como desempenho e eficiência energética são críticas no projeto dos codificadores, principalmente para a etapa de ME. Para demonstrar esta questão, dois cenários distintos (um com o algoritmo FS e o outro com o algoritmo DS) foram executados sequencialmente e os resultados obtidos, em termos de quadros por segundo (@fps), foram posteriormente analisados. Os dois cenários foram executados sob as mesmas condições de teste: JM 18.2 (A) e o *x264* (B) (X264, 2012). Em resumo, os parâmetros utilizados foram os mesmos em ambos os codificadores de vídeo apresentados: ***(i)*** Algoritmos de Busca FS e DS; ***(ii)*** Área de Busca *32x32 pixels*; ***(iii)*** Sequências de vídeos de resoluções CIF (300 quadros), HD720p (50 quadros) e HD1080p (50 quadros); ***(iv)*** Perfil *Baseline.*

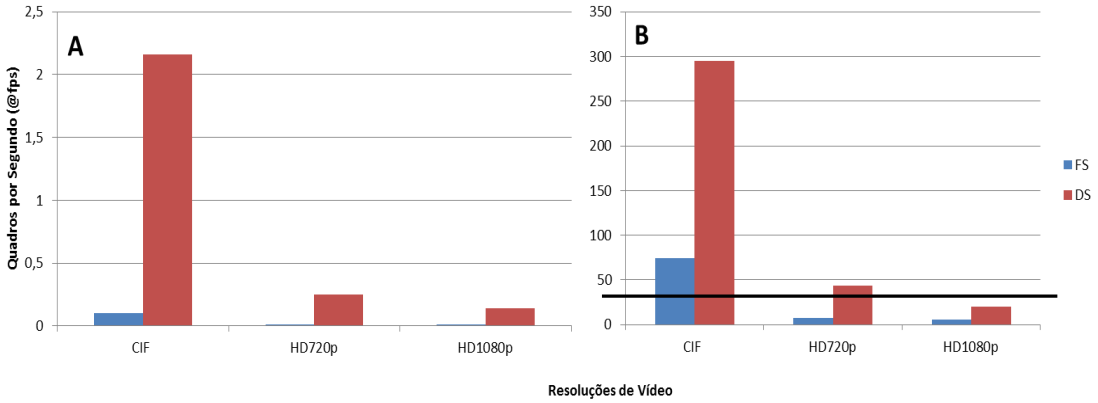


Figura 1.2: JM (A) *x x264* (B) – Quadros por Segundo (@fps).

Com base na Figura 1.2 se pode observar que nenhuma das implementações (FS e DS) atinge codificações em tempo real (taxa ideal de 30 quadros por segundo) para vídeos de alta resolução. No entanto, conforme destaque na Figura 1.2, apenas o codificador *x264* (B) com as resoluções CIF (*352x288*) e HD720p (*1280x720*) atingiu este requisito. Além disso, a resolução CIF é baixa com pouca aplicação nos produtos atuais e o desenvolvimento do *x264* foi projetado visando principalmente um bom desempenho em termos de tempo de execução. Por sua vez, no *software* JM, o qual é um codificador mais completo (pois implementa todas as funcionalidades de um codificador H.264) voltado para uso científico e não para prover uma implementação eficiente, não obteve taxa para processamento aceitável (por exemplo, acima de 30 fps) em nenhuma das resoluções apresentadas.

O objetivo do trabalho é acelerar o processo de Estimação de Movimento, visto que as análises apresentadas mostraram que este é o módulo do codificador de vídeo que apresenta o maior custo computacional, principalmente quando vídeos de alta resolução são processados. Esta aceleração concentra-se no uso de uma plataforma massivamente paralela, a qual é disponível em larga escala nos computadores pessoais da atualidade, denominada GPU (*Graphic Processing Units).*

1.2 Contribuição deste trabalho

Neste contexto, trabalhos recentes têm apresentado diferentes soluções para a paralelização de ME focando plataformas que dispõe de múltiplas unidades de

processamento (do inglês, *multi-core*) em um único dispositivo. Recentemente, a comunidade acadêmica e industrial de processamento paralelo voltou sua atenção para Unidades de Processamento Gráfico (GPU). As GPUs foram originalmente projetadas para processamento gráfico e renderização 3D, porém devido ao seu elevado potencial de paralelismo, as empresas fabricantes de placas gráficas têm estendido as capacidades e flexibilidades delas para o processamento de propósito geral, cunhando o termo GPGPU (*General Purpose* GPU) (GPGPU, 2012). Com as GPGPUs, novas arquiteturas e interfaces de programação foram criadas para mais facilmente programar a funcionalidade das unidades gráficas.

A arquitetura Fermi (NVIDIA FERMI, 2012), proposta pela empresa NVIDIA (NVIDIA, 2012) em 2009, é a mais popular e proeminente solução GPGPU disponível nos dias atuais por introduzir diferentes inovações em relação a arquiteturas anteriores, bem como o incremento de: dois níveis de memória cache, número de núcleos de processamento, precisão, entre outros. A arquitetura CUDA (*Compute Unified Architecture*) (NVIDIA CUDA, 2011) foi proposta pela NVIDIA com o objetivo de viabilizar uma API (*Application Program Interface*) de programação capaz de explorar o elevado grau de paralelismo inerente a estes dispositivos gráficos. O poder computacional provido por este tipo de tecnologia tem feito deste paradigma um grande destaque em diversas áreas, incluindo a comunidade de codificação de vídeo.

A partir da exploração do paralelismo inerente aos algoritmos FS e DS em conjunto com o processamento paralelo disponível nas recentes placas de vídeo, este trabalho apresenta um método eficiente para mapear a Estimação de Movimento (FS e DS) em GPU usando a arquitetura CUDA, visando atingir processamento em tempo real no processo de codificação de vídeo. Para avaliação de desempenho da ME em GPU, as soluções da ME propostas em CUDA são comparadas com implementações *muti-core* e distribuídas. A solução *multi-core* está baseada na API OpenMP (OPENMP, 2012) e, por outro lado, a versão distribuída na biblioteca MPI (*Message Passing Interface*) (MPI, 2012). Os desempenhos destas possíveis soluções são comparados usando sequências de vídeo reais em diferentes resoluções (CIF, HD720p e HD1080p). Além disso, os resultados são comparados com o estado-da-arte os quais apresentam implementações e resultados neste contexto.

1.3 Organização do Texto

O texto da dissertação foi organizado da seguinte forma. No Capítulo 2, uma introdução geral sobre os principais conceitos que envolvem codificação de vídeo e a arquitetura CUDA são apresentados. O Capítulo 3 apresenta uma revisão mais específica sobre a Estimação de Movimento, foco deste trabalho, e os algoritmos de busca considerados: *Full Search* e *Diamond Search*. O Capítulo 4, apresenta a contribuição desta dissertação, detalhando as implementações em CUDA e suas respectivas versões *multi-core* e distribuídas. O Capítulo 5 apresenta a descrição de alguns trabalhos relacionados para comparações com os resultados obtidos deste trabalho. O Capítulo 6 descreve os experimentos realizados e introduz os resultados alcançados bem como comparações com trabalhos relacionados para análise de desempenho do trabalho. Por fim, o Capítulo 7 conclui o trabalho e indica alguns trabalhos futuros.

2 CONCEITOS BÁSICOS DE COMPRESSÃO DE VÍDEO E ARQUITETURA CUDA

Neste capítulo serão brevemente abordados alguns conceitos básicos sobre a compressão de vídeos digitais e a arquitetura CUDA. Estes conceitos serão fundamentais para a compreensão do trabalho desenvolvido nesta dissertação.

2.1 Conceitos Básicos sobre Compressão de Vídeo Digital

O principal objetivo da compressão de vídeos digitais está diretamente relacionado com a redução da quantidade de dados e informações das imagens que constituem um vídeo. Este volume de dados é variável de acordo com a resolução do vídeo em questão podendo dificultar a transmissão e o armazenamento das informações. Por exemplo, considerando um vídeo original com resolução *720x480 pixels* sendo reproduzido a uma taxa de 30 quadros (*frames*) por segundo com 24 bits para representar cada *pixel*, seriam necessários os seguintes recursos: ***(i)*** 249 milhões de bits por segundo para transmissão e, ***(ii)*** 19 Gb para o armazenamento de uma sequência de 10 minutos de vídeo (AGOSTINI, 2007). Sendo assim, pode-se perceber a importância da compressão de vídeo para viabilizar a manipulação, transmissão e armazenamento deste tipo de mídia.

2.1.1 Estrutura de um Vídeo Digital

Vídeos digitais são formados por quadros ou imagens estáticas. A reprodução destas imagens estáticas a uma taxa de *30Hz* induz a percepção de movimento (RICHARDSON, 2003), onde a suavidade dos movimentos em uma cena é dada pela quantidade de quadros reproduzidos por segundo. As imagens que pertencem a uma mesma cena de vídeo e, temporalmente próximas, são chamadas de imagens (quadros) vizinhas.

Considerando que um vídeo digital é constituído de uma sequência de quadros reproduzidos a uma determinada frequência, tipicamente, os codificadores de vídeo dividem estes quadros em blocos. Na figura 4.1 pode-se observar uma ilustração destes conceitos.

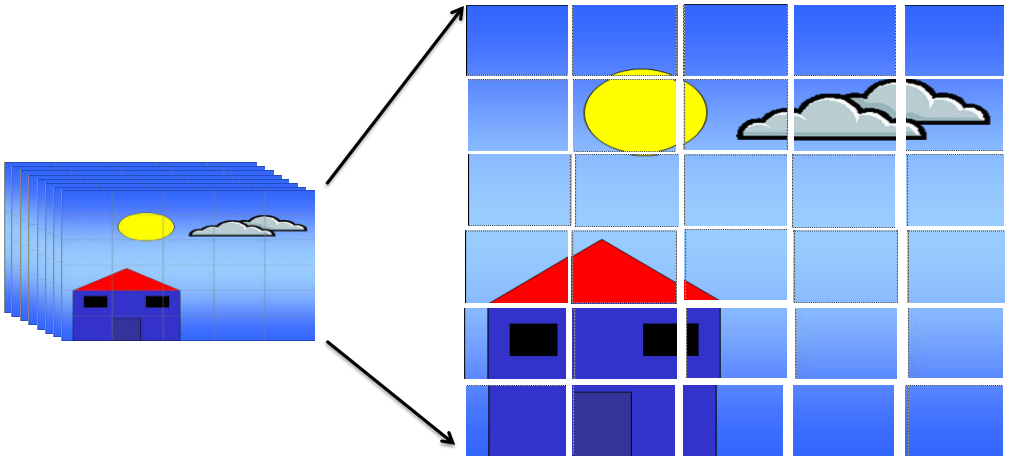


Figura 2.1: Sequência de Quadros e blocos em um vídeo digital.

Como se pode observar na Figura 2.1, cada quadro, composto por uma matriz de blocos. Os blocos que compõe um quadro podem ser subdivididos em outros blocos de diferentes dimensões. O padrão de codificação de vídeo H.264/AVC, que representa o estado-da-arte neste cenário, denomina um bloco de dimensão *16x16 pixels* de *macrobloco*, onde cada macrobloco pode ser divido em blocos de *8x16*, *16x8, 8x8* e *4x4 pixels*, como está apresentado na Figura 2.2. Além disso, salienta-se que a dimensão *8x8 pixels* pode ser subdividida em blocos de *4x8*, *8x4* e *4x4 pixels*, veja Figura 2.3.

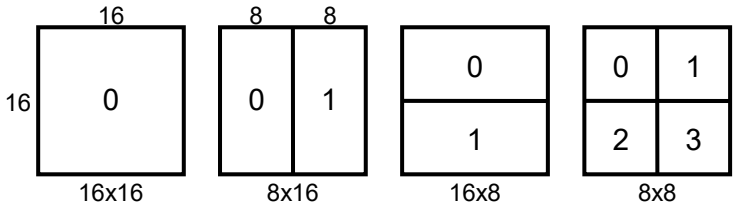


Figura 2.2: Divisão do macrobloco em partições (AGOSTINI,2007).

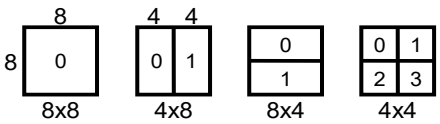


Figura 2.3: Divisão do sub-macrobloco em partições (AGOSTINI, 2007).

Os quadros são formados por um conjunto de pontos, também denominados de *pixels*. De maneira análoga aos blocos, os *pixels* pertencentes à mesma cena, e/ou temporalmente próximos, são chamados de *pixels* vizinhos. Um *pixel* pode ser representado por um número, ou por um conjunto de números os quais representam as componentes de cor (crominância - cor e luminância - brilho), de acordo com o espaço de cor utilizado. Os *pixels* pertencentes à mesma imagem são denominados *pixels* vizinhos (RICHARDSON, 2003).

2.1.2 Espaço de Cores

O sistema visual humano é responsável por distinguir uma ampla gama de cores. A esta interpretação associa-se a representação digital de um vídeo colorido. O olho humano é constituído de elementos sensíveis à luz, denominados bastonetes e cones. Os bastonetes, sensíveis a baixa luz de luminosidade são utilizados para visão noturna de modo a captar imagens com poucos níveis de detalhes. Por sua vez, os cones, sensíveis às cores vermelha, verde, azul e amarelo, são utilizados para visão diurna identificando um maior nível de detalhes, mas, em contrapartida precisam de maior luminosidade para seu funcionamento (POLLACK, 2006). Por estas razões observa-se que o olho humano apresenta dificuldades na distinção de cores de objetos em ambientes com pouca iluminação. Entretanto, este sistema visual é capaz de discernir mais de cinco dúzias de tons de cinza, as quais indicam a intensidade luminosa da imagem (*luminância*) (GONZALES, 2003). Milhares de cores (*crominância*) podem ser identificadas através de combinações de intensidades de cores captadas pelos cones. Sendo assim, pode-se dizer que o olho humano é mais sensível as informações de luminância do que as informações de crominância.

Para representação de cores de forma digital necessita-se de um sistema denominado *espaço de cores*. Dentre os diversos espaços de cores existentes quando da representação de imagens digitais, citam-se: RGB, HSI e YCbCr (SHI, 1999). O sistema RGB, apesar de ser um dos mais conhecidos na literatura, possui um elevado grau de correlação entre seus componentes (R, G e B). Sendo assim, este espaço de cores é considerado inadequado para codificação de vídeo, pois dificulta a manipulação das informações de forma independente (brilho e cor), e assim, inviabiliza a aplicação da *Subamostragem de Cores*, técnica de compressão que será brevemente discutida na Seção 2.6. Por outro lado, nos espaços de cores onde os componentes luminância e crominância são tratados de forma independente, a técnica referente à subamostragem torna-se viável. Nos codificadores de vídeos atuais, comumente utilizados como o padrão H.264/AVC, o espaço de cores YCbCr (luminância e crominância) é considerado. O espaço de cores YCbCr é composto de três componentes básicos: ***(i)*** luminância (Y): define a intensidade luminosa ou brilho, ***(ii)*** crominância azul (Cb) e, ***(iii)*** crominância vermelha (Cr).

2.1.3 Redundância dos Dados na Representação dos Vídeos Digitais

A compressão de vídeos digitais é essencial para tornar viável a transmissão e armazenamento de um vídeo digital. O elevado grau de redundância entre os dados é uma característica explorada pelas técnicas de compressão de vídeo existentes, e considerada fundamental para que o processo seja eficiente. Esta propriedade consiste em descartar um grande volume de dados a partir de uma sequência de vídeo, tornando possível a representação de um vídeo digital com uma menor quantidade de bits e com boa qualidade quando comparada com a versão original.

Os algoritmos de compressão de dados têm como principal objetivo identificar redundâncias entre elementos as quais se manifestam através de algumas propriedades, como por exemplo, repetições e correlações. Os dados que não contribuem com informações novas não são considerados redundantes. Na compressão de vídeo quatro tipos de redundâncias são, geralmente, exploradas: espacial, temporal, entrópica e psicovisual.

2.1.3.1 Redundância Espacial

A redundância espacial, também conhecida como "*redundância intra-quadro*", "redundância *inter-pixel*" ou até mesmo "*redundância intra-frame*" (GHANBARI, 2003), refere-se à correlação que existe entre os *pixels* de um mesmo quadro. Os *pixels* que constituem um mesmo quadro tendem a ser semelhantes quando estão representando um mesmo objeto na imagem. Sendo assim, acrescentam pouca informação visual a cada ponto.

2.1.3.2 Redundância Temporal

A redundância temporal, também conhecida como "*redundância inter-quadros*" (GHANBARI, 2003), é referente à correlação existente entre quadros temporalmente próximos em um vídeo. Este tipo de redundância pode ocorrer em diferentes situações, principalmente quando não há um movimento de um objeto pertencente a uma mesma cena, ou seja, quando um bloco de *pixels* não se desloca de um quadro para outro.

2.1.3.3 Redundância Entrópica

A redundância entrópica ou codificação de entropia (BHASKARAN, 1999) não está diretamente relacionada com o conteúdo da imagem, mas com a maneira de representação dos dados. A redundância entrópica tem como meta representar as informações mais frequentes de um vídeo digital por símbolos de modo a obter um número menor de bits em sua respectiva representação. Desta forma, quando a probabilidade de ocorrência de um símbolo é grande, a quantidade de novas informações a serem transmitidas por este símbolo é menor.

2.1.3.4 Redundância Psicovisual

A redundância psicovisual e considerada apenas por alguns autores, concentra-se no modelo do sistema visual humano, de modo a explorar as características do olho humano (cores e luminosidade) (GONZALEZ, 2003). Este tipo de redundância ocorre pelo fato de algumas informações terem uma menor relevância em relação a outras informações em um processamento visual normal. Assim, os codificadores de vídeo podem explorar esta característica para aumentar a eficiência de codificação através da redução da taxa de amostragem dos componentes de crominância em relação aos componentes de luminância.

2.1.4 Técnicas de Compressão de Vídeo

Com o objetivo de reduzir ao a quantidade de dados que representam vídeos digitais, os padrões de codificação de vídeo aplicam um conjunto de algoritmos de compressão. A Figura 2.4 apresenta um resumo de técnicas de compressão de vídeo existentes organizadas na ordem em que são aplicadas no processo de compressão de vídeo e, além disso, de acordo com o tipo de redundância em que cada qual atua.

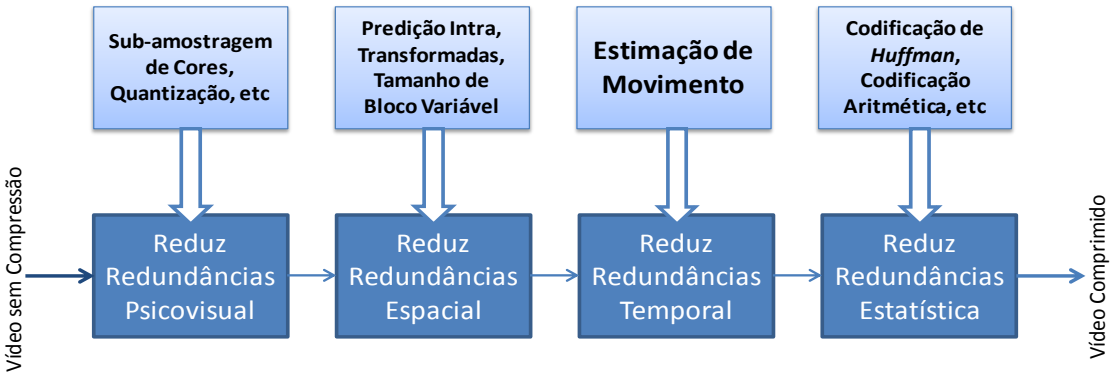


Figura 2.4: Técnicas de compressão de vídeos aplicadas para cada tipo de redundância (modificado a partir de (KALVA, 2006)).

Existem técnicas de compressão de dados sem perdas e com perdas de informação (RICHARDSON, 2003). O principal objetivo das técnicas de compressão sem perdas é codificar os dados de modo a possibilitar uma reconstrução idêntica a sua versão original. Para este fim as redundâncias presentes nos vídeos digitais, descritas anteriormente, são devidamente exploradas.

Os vídeos digitais contêm um grande volume de informações o que torna necessária uma elevada eficiência em relação às taxas de compressão. Isto é obtido através da eliminação de informações redundantes do vídeo. A exploração das limitações do olho humano permite viabilizar a tolerância (aceitação) de perdas de informação (redundância psicovisual), de forma que a imagem reconstruída obtenha uma diferença visual imperceptível em relação à imagem original e utilize um menor volume de dados.

Existem diferentes técnicas de compressão de vídeo com perdas. Um exemplo é a subamostragem de cores, introduzida na Seção 2.6. Outra técnica importante é a quantização. Esta técnica faz uma divisão inteira dos coeficientes gerados pela transformada, reduzindo parte dos coeficientes a zero. Este fator insere perdas no vídeo, que são controladas pelo parâmetro de quantização (QP). Quanto maior o parâmetro de quantização, mais informação é perdida, ou seja, se obtém mais compressão ao custo de uma menor qualidade de imagem.

2.1.5 Subamostragem de Cores

Como já mencionado na Seção 2.3, o olho humano é mais sensível às informações de luminância do que as informações de crominância. Os padrões de compressão de imagens estáticas e vídeos exploram esta característica para prover um aumento na eficiência de codificação. Este ganho é obtido através de uma operação denominada, subamostragem de cores. A subamostragem de cores é realizada sob o espaço de cores nos padrões de compressão de vídeo atuais e consiste na redução da taxa de amostragem dos componentes de crominância em relação aos componentes de luminância (RICHARDSON, 2002).

Por exemplo, considerando um bloco de dimensão *2x2* o qual é constituído de quatro amostras, os formatos mais comuns na literatura para relacionar os componentes de crominância com os componentes de luminância em uma aplicação desta propriedade são: ***(i)*** 4:4:4 – para cada quatro amostras de luminância (Y), existem quatro amostras de crominância azul (Cb) e quatro amostras de crominância vermelha (Cr), deste modo

a subamostragem não é aplicada pois a resolução dos três componentes é igual; ***(ii)*** 4:2:2 - para cada quatro informações de Y, existem apenas duas informações de Cb e duas de Cr; ***(iii)*** 4:2:0 ou 4:1:1 - para cada quatro informações de Y, existe apenas uma informação de Cb e uma de Cr.

Neste contexto, verifica-se que a subamostragem de cores é responsável por um ganho significativo na compressão do vídeo, uma vez que parte da informação da imagem é previamente descartada, sem causar impacto visual perceptível. O formato 4:2:0, por exemplo, onde cada componente de cor possui apenas uma das quatro amostras presentes, irá utilizar a metade das amostras para codificar a cor de um vídeo no formato YCbCr quando comparado com o formato 4:4:4. Este fato proporciona uma redução significativa, mais de 50% de informações, considerando apenas a aplicação da subamostragem, descartando a aplicação de qualquer outro processo de compressão.

O padrão H.264/AVC, estado-da-arte neste cenário, faz uso do espaço de cores YCbCr. E em relação à subamostragem de cores, são considerados os formatos 4:4:4, 4:2:2 e o 4:2:0. O formato 4:2:0 é mais utilizado por proporcionar maior ganho dentre os outros formatos.

2.1.6 Análise da Qualidade de Vídeos Digitais

Considerando as técnicas de compressão de vídeo com perdas de informação, como por exemplo, a subamostragem de cores e quantização mencionadas na Seção 2.1.5, a necessidade de análise da qualidade de um vídeo digital é fundamental. Por este motivo, diferentes critérios foram criados com o objetivo de tornar possível esta avaliação de qualidade das imagens. Estes critérios fazem referência a algoritmos que comparam os *pixels* da imagem original com os *pixels* da imagem reconstruída, ou seja, com a imagem obtida após a decodificação.

Neste contexto, o critério mais comum na literatura é denominado de PSNR (*Peak Signal-to-Noise Ratio*) (GHANBARI, 2003). O PSNR é uma métrica de comparação utilizada para avaliar a qualidade objetiva dos vídeos e é definido pela equação (1).

$$PSNR_{dB} = 20 \cdot \log_{10}\left(\frac{MAX}{\sqrt{MSE}}\right) \tag{1}$$

A comparação por esta métrica é realizada sob dois diferentes quadros (original - O e estimado - R) em uma escala logarítmica utilizando como base o critério de similaridade MSE (*Mean Squared Error*) relativo ao valor máximo (*MAX*) de representação do *pixel*. O MSE é definido em (2), onde *m n* é a dimensão da imagem em *pixels*, *O* e *R* são as imagens original e reconstruída, respectivamente.

$$MSE = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\left[R_{i,j} - O_{i,j}\right]^2 \tag{2}$$

Desta forma, a qualidade do vídeo é determinada de acordo com o valor obtido pelo PSNR, onde os valores altos indicam alta qualidade e valores baixos indicam baixa qualidade.

2.2 Conceitos Básicos sobre a Arquitetura CUDA

A arquitetura CUDA foi proposta pela empresa NVIDIA em 15 de fevereiro de 2007. Esta arquitetura é uma tecnologia voltada para computação de propósito geral (GPGPU) massivamente paralela e de alto desempenho a partir de processadores gráficos. O objetivo desta arquitetura é explorar o alto grau de paralelismo inerente às GPUs, especialmente os *chips* projetados pela empresa NVIDIA. O grande poder computacional oferecido por este tipo de tecnologia tem feito desta arquitetura um grande destaque em diversas áreas, principalmente na comunidade científica. Um dos pré-requisitos para o uso eficiente desta arquitetura é que as aplicações a serem desenvolvidas sejam altamente paralelizáveis.

CUDA é conhecida tanto como a linguagem que viabiliza a programação das placas gráficas, quanto como a arquitetura que compõe as GPUs. Esta API é, tipicamente, uma extensão da linguagem C/C++ constituída de abstrações que facilitam a programação dos processadores gráficos para execução de aplicações com graus mínimos de paralelismo de dados, estruturas síncronas e blocos regulares, também tratadas como aplicações de propósito gerais. A arquitetura CUDA pode ser utilizada a partir da geração 80 (*GeForce 8800*) das placas gráficas da empresa NVIDIA.

A arquitetura CUDA é baseada na execução de *threads* simultâneas de uma mesma instrução. Por esta razão, a empresa NVIDIA introduziu um novo conceito na computação paralela, análogo à organização SIMD (*Single Intructions Multiple Data*), denominado SIMT (*Single Instructions Multiple Thread*). Neste cenário, o gerenciamento dos dados desta tecnologia dá-se no nível de *threads*, onde a arquitetura desempenha funções básicas, como: criar, gerenciar, agendar e executar *threads* automaticamente (NVIDIA CUDA, 2011).

Os conceitos apresentados nesta seção referem-se à API CUDA, em especial à arquitetura Fermi. A Fermi foi utilizada no desenvolvimento deste trabalho e era considerada o estado-da-arte em placas gráficas à época de seu desenvolvimento, proposta pela NVIDIA em 2009. A arquitetura Fermi, é apresentada na Figura 2.5 e segue os princípios da API CUDA, porém com inovações em níveis de software e *hardware*.

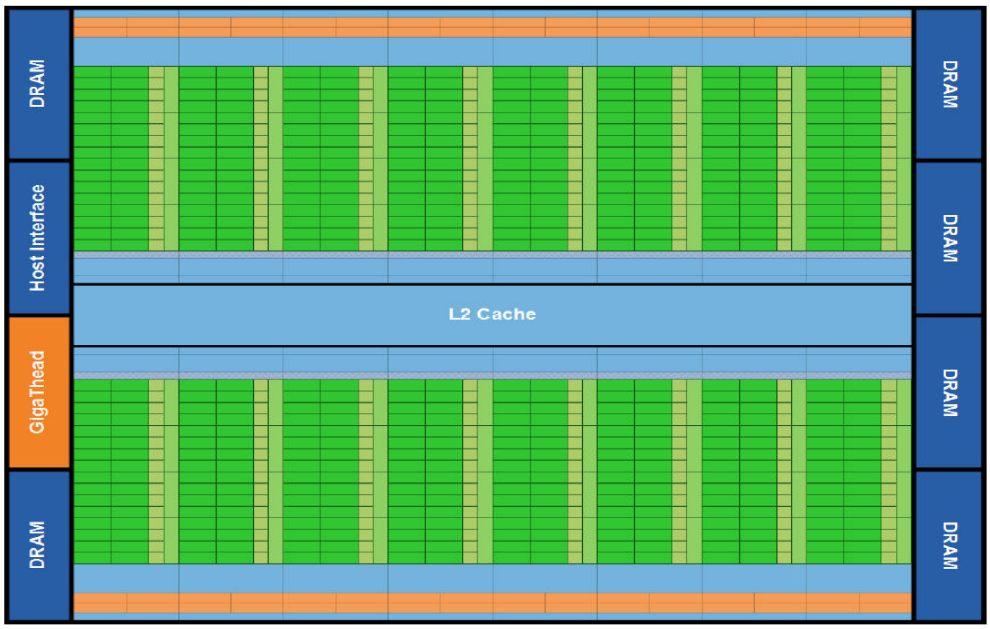


Figura 2.5: Arquitetura Fermi (NVIDIA FERMI, 2012).

As melhorias inerentes a esta arquitetura proporcionam avanços significativos em relação a desempenho e eficiência, como por exemplo, a inclusão de dois níveis de memória cache, elevado poder de processamento a partir do aumento do número de processadores, melhor gerenciamento de instruções e maior precisão. Maiores informações arquiteturas podem ser encontradas em (NVIDIA FERMI, 2012).

2.2.1 Fluxo de Processamento

O fluxo de processamento de uma aplicação, a qual utiliza a arquitetura CUDA como alternativa de co-processamento, é um dos principais pontos a serem abordados neste cenário. Desta forma, tem-se a importância de dois conceitos: processo e *kernel*. O processo é a representação de uma tarefa em CPU (TANENBAUM, 2007). Por sua vez, quando a placa gráfica é requisitada parte deste processo é enviado para GPU, o qual é denominado de *kernel*. Veja a ilustração deste fluxo de execução na Figura 2.6.

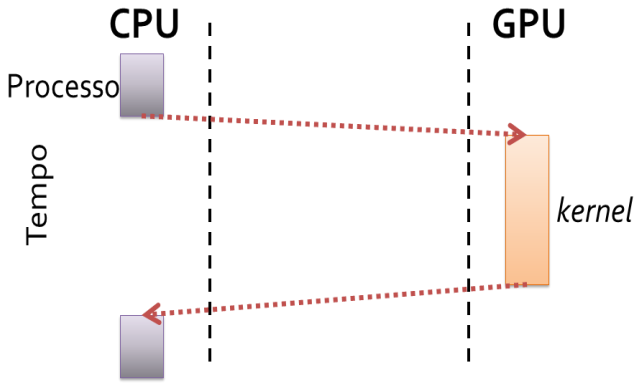


Figura 2.6: Fluxo de Execução – Arquitetura CUDA (PILLA, 2009).

De acordo com a figura apresentada pode-se observar que a execução da aplicação dá início na CPU a partir de um código sequencial, onde neste contexto também é chamada de *host*. Quando o *kernel* é invocado pela CPU, a execução é movida para GPU onde o código executado em paralelo a partir de suas unidades básicas de processamento, denominadas *threads*. No momento em que todas as *threads* terminarem de executar a aplicação, a correspondente função paralela é finalizada e a execução volta para a CPU, como dado(s) resultante(s) definido pelo programador.

2.2.2 Processadores

Os núcleos de processamento que constituem a arquitetura CUDA são denominados *Streaming Multiprocessors* (SM). A arquitetura de um SM pode ser visualizada na Figura 2.7.

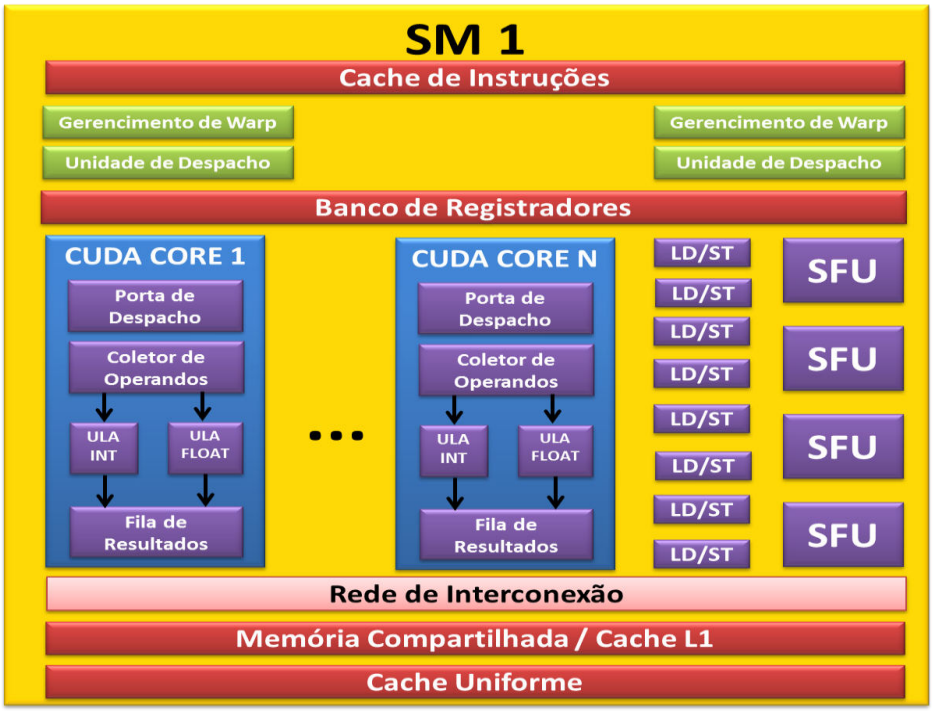


Figura 2.7: Arquitetura de um SM (modificado a partir de (PILLA, 2009)).

Na Figura 2.7 pode-se observar que um SM é composto de: ***(i)*** CUDA *Cores* (CC) ou *SPs (Scalar Processors*): núcleos de processamento os quais são constituídos por *threads*; ***(ii)*** *LD/ST*: realiza o gerenciamento dos CUDA Cores para execução de instruções *Load* / *Store*; ***(iii)*** Banco de Registradores: podem ser acessados pelos CUDA *Cores*; ***(iv)*** SFU (*Super Function Units*): unidades de funções especiais responsáveis por diferentes aplicações, dentre elas operações que introduzem precisão dupla de ponto flutuante; ***(v)*** Cache de Instruções: armazenamento de instruções; ***(vi)*** Gerencimento de *Warp*: unidade de manipulação das *threads*, onde um conjunto de *threads* denomina-se *warp*; ***(vii)*** Unidade de Despacho: responsável pelo gerenciamento das instruções que serão executadas pelos CUDA *Cores.*

Como já mencionado, a hierarquia de processamento da arquitetura CUDA considera *threads* como a unidade básica de processamento. Estas *threads* são devidamente organizadas em *blocos*, que por sua vez, os blocos podem representar até três dimensões e sua organização é dada por um conceito denominado *grid* (NVIDIA CUDA, 2012). Na *grid*, a qual pode ser representada por até duas dimensões, são definidos os recursos que serão utilizados pela GPU para a execução de uma determinada aplicação, como por exemplo, a alocação do número de blocos e *threads* necessários. Esta hierarquia pode ser visualizada na Figura 2.8 onde é apresentado o mapeamento das *threads* na arquitetura CUDA, de acordo com os conceitos mencionados.

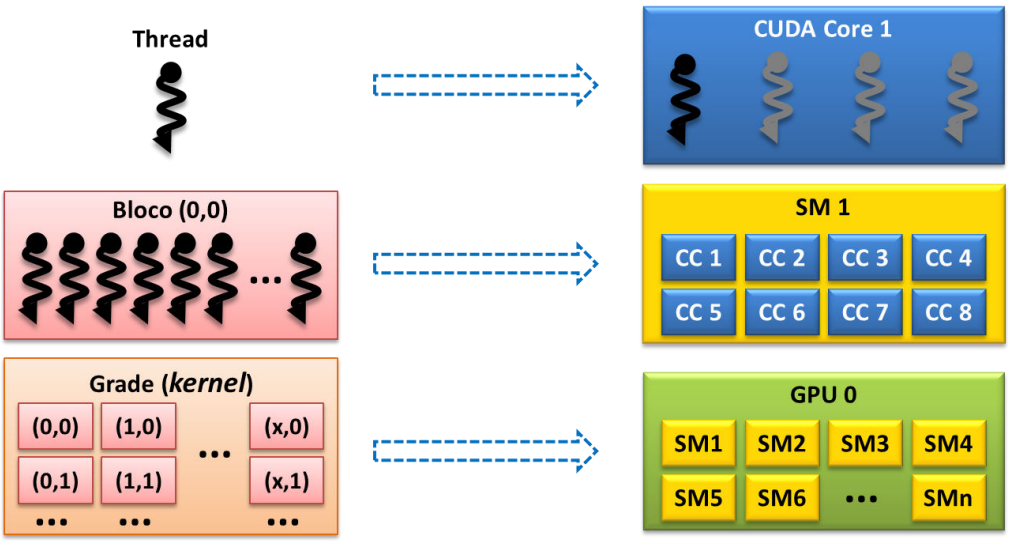


Figura 2.8: Mapeamento das *Threads* na Arquitetura CUDA (modificado a partir de (PILLA, 2009)).

A Figura 2.8 representa o mapeamento das *threads* do nível lógico para o nível físico, onde uma *thread* é mapeada para um *CUDA Core* e, como já mencionando anteriormente, as *threads* são organizadas em w*arps*. Por sua vez, um bloco é mapeado para um SM, e por fim, um *kernel* (ou mais de um) para uma GPU.

Neste cenário, destaca-se como umas das inovações providas pela arquitetura Fermi a possibilidade de utilização de mais de uma *warp* por SM, considerando que em versões anteriores era permitida apenas uma.

2.3 Considerações Finais sobre o Capítulo

Este capítulo introduziu alguns conceitos que serão importantes para a compreensão dos demais capítulos que constituem esta dissertação. A fundamentação apresentada está relacionada com alguns conceitos básicos de compressão de vídeo, bem como algumas técnicas e a estrutura de vídeos digitais. Salienta-se que *pixels* e imagens vizinhas possuem características muito semelhantes e este fator é considerado indispensável na compressão de vídeos digitais.

Além disso, uma breve descrição da arquitetura CUDA foi introduzida, em especial o mapeamento lógico e físico das unidades básicas de execução deste paradigma (*threads*) e a arquitetura inerente a esta tecnologia. O termo "*arquitetura CUDA*" se refere tanto à arquitetura inerente aos dispositivos gráficos, quanto a API que dá suporte a programação dos núcleos de processamento para execução de aplicações de propósitos gerais.

O Capítulo 3 apresenta com mais detalhes os conceitos relacionados à Estimação de Movimento, presente apenas nos codificadores de vídeo, por ser o objeto desta dissertação.

3 ESTIMAÇÃO DE MOVIMENTO

Este capítulo apresenta os principais conceitos relacionados à Estimação de Movimento (ME). O funcionamento deste módulo na codificação de vídeo será descrito, bem como, alguns dos principais algoritmos de busca conhecidos. Além disso, os critérios de similaridade mais utilizados serão brevemente descritos.

3.1 Introdução

A Figura 3.1 apresenta o diagrama de blocos referente a um codificador de vídeo genérico. Os blocos ilustrados na Figura 3.1 têm como objetivo comum a redução de algum dos tipos de redundância existentes nos vídeos digitais. Os principais blocos ilustrados na Figura 3.1 são: ***(i)*** predição inter-quadros: explora a redundância temporal entre quadros vizinhos de um vídeo; ***(ii)*** predição intra-quadros: explora a redundância espacial no quadro atual (quadro em processamento); ***(iii)*** transformadas e quantizações: responsáveis pela redução da redundância psicovisual; ***(iv)*** codificação de entropia: tem como foco a redução da redundância entrópica.

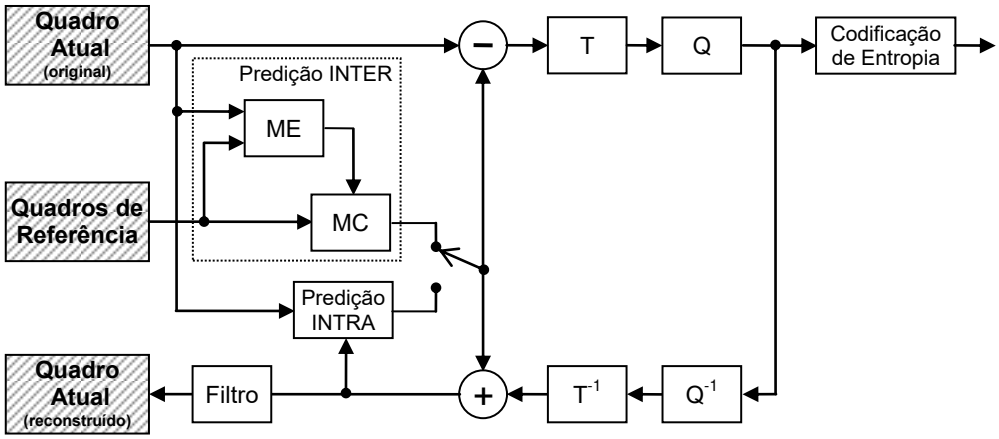


Figura 3.1: Diagrama de Blocos de um Codificador de Vídeo (AGOSTINI, 2007).

A Estimação de Movimento (ME, Figura 3.1) é um dos blocos que compõe um codificador de vídeo e é parte do processo de codificação inter-quadros. A estimação de movimento é responsável por explorar a redundância temporal entre quadros vizinhos. Em resumo, o objetivo da ME é realizar uma busca em um ou mais quadros de referência (quadros previamente codificados e reconstruídos) por um bloco que seja o mais semelhante possível a um bloco do quadro atual. Este módulo está presente apenas nos codificadores de vídeo e é considerado um dos mais importantes neste cenário

provendo a maior parcela de ganhos na compressão (PURI, 2004). Como já mencionado e ilustrado na Figura 1.1 estes ganhos são obtidos ao custo de um elevado custo computacional representando, em geral, mais 80% do total da complexidade dos codificadores de vídeo atuais (H.264/AVC. MPEG, etc) (CHENG, 2009).

Visto que um vídeo digital é composto por uma série de quadros reproduzidos a uma certa taxa de exibição (tipicamente 30 quadros por segundo), a similaridade apresentada entre dois quadros temporalmente vizinhos tende a ser elevada. Desta forma, quando são comparados dois (ou mais) quadros vizinhos tem-se uma grande quantidade de informações redundantes. Neste contexto, para uma melhor compreensão da ME, duas imagens (quadros) temporalmente vizinhas são representadas nas Figuras 3.2 (a) e 3.2 (b).

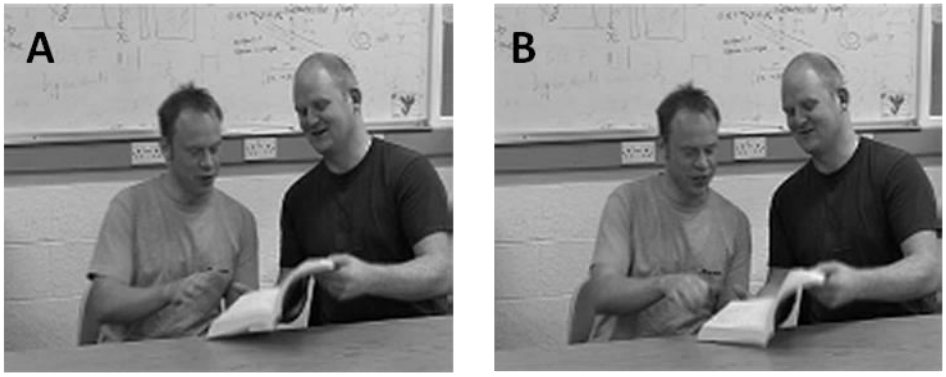


Figura 3.2: (a) Quadro 1 de um Vídeo. (b) Quadro 2 de um Vídeo. (RICHARDSON, 2003).

Analisando as imagens apresentadas na Figura 3.2 verifica-se uma grande similaridade existente entre elas. A diferença entre elas, também conhecida como diferença residual (*pixel* a *pixel*), é apresentada na Figura 3.3 (a).

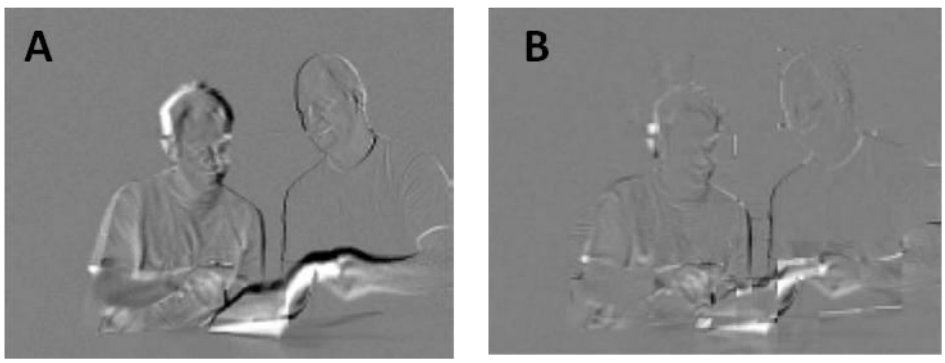


Figura 3.3: (a) Diferença Residual sem aplicação da ME. (b) Diferença Residual com aplicação da ME. (RICHARDSON, 2003).

As principais diferenças, também chamadas de diferenças positivas, concentram-se na área clara, por sua vez as diferenças secundárias, ou diferenças negativas, podem ser visualizadas na área escura. Por fim, toda área em cinza representa regiões onde praticamente não há diferenças. A Figura 3.3 (b) também ilustra a diferença residual existente entre as imagens vizinhas apresentadas na Figura 3.2, porém considerando a aplicação do processo da ME. Deste modo, pode-se observar uma redução nos resíduos da imagem apresentada na Figura 3.3 (b) quando comparada com a Figura 3.3 (a). Esta redução de resíduos está diretamente relacionada com uma considerável compressão do

vídeo. Desta forma, avaliando os quadros resultantes (Figuras 3.3 (a) e 3.3 (b)), salienta-se que a codificação diferencial apresentada (Figura 3.3 (b)) viabilizou uma percepção clara da importância do módulo de ME na codificação de vídeo.

O processo de Estimação de Movimento está baseado na identificação dos movimentos ocorridos entre quadros vizinhos de uma cena e realizar o mapeamento destes deslocamentos para vetores de movimento. Um vetor de movimento (par de números inteiros) corresponde à posição do bloco no quadro de referência que obteve a maior similaridade em relação ao bloco no quadro atual. Desta forma, apenas os vetores de movimentos e os resíduos são transmitidos para as próximas etapas que constituem o codificador de vídeo, ao invés de informações originais referentes a quadros inteiros.

A busca por um bloco do quadro atual em um (ou mais) quadro de referência é delimitada por uma região denominada área de busca ou pesquisa. A área de busca, tipicamente, localiza-se ao redor do bloco do quadro atual a ser codificado. Esta região pode conter o quadro inteiro, mas em geral, seu tamanho é definido por uma fração do quadro, de modo a diminuir a complexidade computacional. O processo referente a ME, para um quadro atual, é ilustrado pela Figura 3.4 e enumerado nos seguintes passos:

1. Definição: um (ou mais) quadro de referência e o tamanho da área de busca;
2. Busca: para cada bloco que constitui o quadro atual, uma busca pelo bloco mais semelhante ao bloco do quadro atual é realizada no quadro de referência, delimitada pela área de busca;
3. Resultado: ao final desta busca, o melhor casamento entre os quadros (*best match*) é localizado e, portanto, um vetor de movimento que indica o seu deslocamento é gerado.
4. Final: todos os vetores de movimento referentes a todos os blocos que compõem o quadro atual são devidamente inseridos junto às informações de codificação de vídeo.

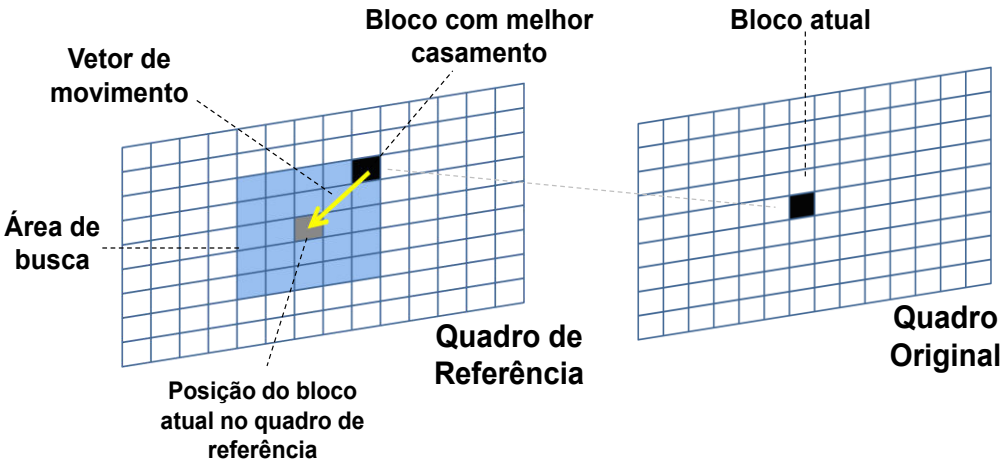


Figura 3.4: Processo referente à Estimação de Movimento (PORTO, 2008a).

Dentre as principais características da ME, salienta-se: ***(i)*** a possibilidade de utilização de mais de um quadro de referência na busca pelo melhor *match* entre os blocos; ***(ii)*** em relação ao espaço de cores apenas a informação de luminância é considerada, visto que as informações de crominância são de pouca relevância no sistema visual humano (RICHARDSON, 2002).

Apesar das vantagens e ganhos que este módulo proporciona na compressão de vídeo, existem muitos parâmetros que podem impactar consideravelmente no resultado final da codificação. Deste modo, destacam-se o tamanho da área de busca e o tamanho dos blocos que serão utilizados. O tamanho da área de busca está diretamente relacionado com a complexidade computacional do problema, pois quanto maior a área, maior será o número de candidatos a serem testados, a fim de encontrar um bom casamento referente ao bloco que está sendo pesquisado. Sendo assim, além do custo inerente à definição do tamanho da área de busca, este critério também influência na qualidade dos vetores gerados, visto que a varredura pode não ser completa dependendo do tamanho da região definida. Por sua vez, em relação ao tamanho dos blocos dos quadros, destaca-se que deve haver um equilíbrio em relação a este parâmetro, pois deve ser pequeno o suficiente para que áreas menores da imagem possam ser comparadas e grandes de modo a evitar que um elevado número de vetores de movimento seja gerado e transmitido.

3.2 Algoritmos de Busca para Estimação de Movimento

No processo de ME, todos os blocos que constituem a área de busca são conhecidos como blocos candidatos. Assim, um algoritmo de busca determina a maneira como os blocos candidatos serão escolhidos dentro da área de busca, sempre tentando encontrar o melhor casamento (similaridade) entre os blocos. Existem diversos algoritmos de busca para Estimação de Movimento na literatura, os quais estão diretamente relacionados com o custo (em termos de computação) exigido pela ME. Além disso, a maneira como os blocos são pesquisados também é responsável pela qualidade dos resultados finais.

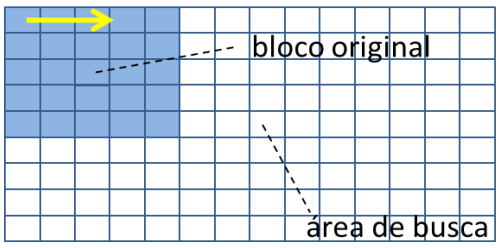
Os algoritmos de busca podem ser classificados em dois tipos: ótimos e sub-ótimos. Os algoritmos sub-ótimos são aqueles que realizam a busca dentro da área de referência guiada por heurísticas, assim somente uma parcela dos blocos candidatos são pesquisados, diminuindo a complexidade computacional desse módulo. Por outro lado, os algoritmos ótimos pesquisam todos os blocos candidatos possíveis, desse modo, os melhores casamentos (matches) são encontrados a um elevado custo computacional. Assim, diferentes algoritmos podem ser citados: ***(i)*** *Full Search* (BHASKARAN, 1999) e (LIN, 2005); ***(ii)*** *Three Step Search* (TSS) (JING, 2004); ***(iii)*** *Diamond Search* (DS) (KUHN, 1999), (ZHU, 2000) e (YI, 2005); ***(iv)*** *One at a Time Search* (OTS) (RICHARDSON, 2002); ***(v)*** *Hexagon Based Search* (HS) (ZHU, 2002); ***(vi)*** *Dual Cross Search* (DCS) (BANH, 2004).

O algoritmo FS é o único algoritmo que pode ser considerado ótimo dentre citados anteriormente, pois busca em todas as posições possíveis na área de pesquisa pelo melhor casamento entre os blocos. Os demais algoritmos podem ser classificados como sub-ótimos, pois adotam diferentes maneiras de busca a fim de reduzir o volume de cálculos na busca pela melhor similaridade, como por exemplo, o algoritmo *Diamond Search.*

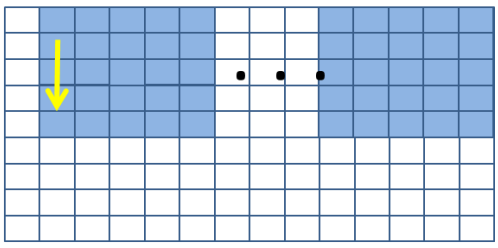
3.2.1 *Full Search* (FS)

O algoritmo *Full Search* é um dos algoritmos considerados neste trabalho. Dentre os algoritmos existentes para ME este algoritmo possui o maior custo computacional, pois todos os blocos candidatos são pesquisados. Desta forma, o algoritmo FS é capaz de gerar vetores ótimos.

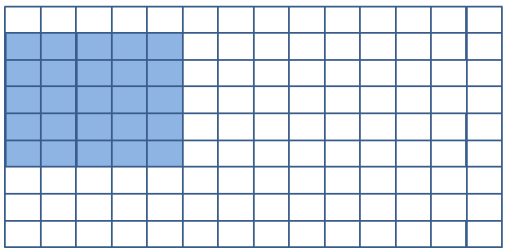
A busca é realizada de maneira com que os blocos se desloquem *pixel* a *pixel* dentro da área de busca, iniciando pelo canto superior esquerdo e terminando no canto inferior direito, como mostra a Figura 3.5. Ao final da busca, um vetor de movimento indica a posição do bloco candidato na área de referência que foi considerado o mais similar ao bloco atual.



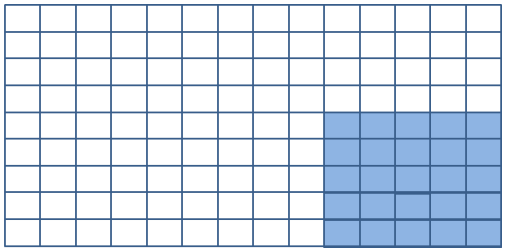
(a) Posição Inicial



(b) O bloco se desloca dentro da área de busca (*pixel* por *pixel*) e é comparado.



(c) Depois de percorrer a primeira linha, a busca pelo bloco se desloca para segunda linha e iniciam-se as comparações.



(d) Posição Final

Figura 3.5: Algoritmo *Full Search* (FS).

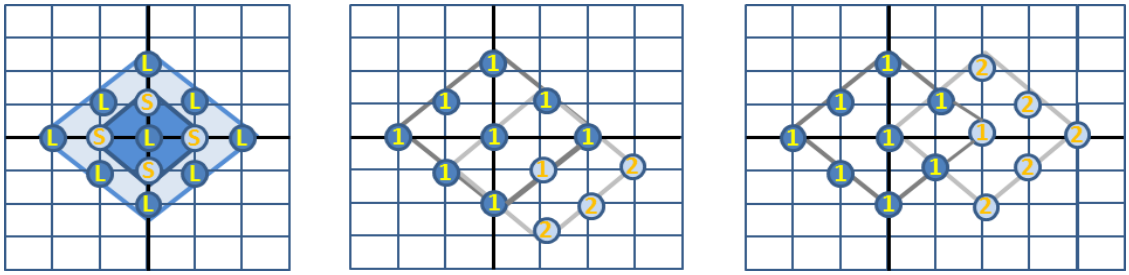
A partir desta descrição pode-se perceber a grandeza da complexidade computacional exigida pelo algoritmo FS, visto que a diferença de todos os *pixels* envolvidos entre o bloco atual e a área de pesquisa é calculada e comparada. Desta forma, como já mencionado, pode-se perceber que o tamanho da área de busca possui um papel fundamental na complexidade deste processo, pois quanto maior a região maior é o número de comparações serão necessárias.

Embora este algoritmo apresente um elevado custo computacional este algoritmo tem como uma das principais características a independência dos dados que o envolvem. Desta forma, visando uma qualidade significativa dos vetores de movimento gerados, este custo pode ser enfrentando a partir da exploração de paradigmas computacionais alternativos.

3.2.2 *Diamond Search* (DS)

O algoritmo *Diamond Search* (DS) é o algoritmo sub-ótimo mais conhecido na literatura. Este algoritmo consiste em dois padrões de busca em forma de diamante, o *Large Diamond Search Pattern* (LDSP) e o *Small Diamond Search Pattern* (SDSP), usados nas etapas iniciais e finais, respectivamente. O LDSP possui nove comparações ao redor da área de busca. O SDSP é usado para refinar os resultados finais gerados pelo LDSP, considerando quatro comparações. A Figura 3.6 (a) apresenta o LDSP e o SDSP. A busca termina quando o melhor casamento é encontrado no centro do maior diamante, do LDSP. Sendo assim, o SDSP é aplicado ao redor do melhor resultado

encontrado pela etapa referente ao LDSP. Quando o melhor casamento for encontrado em uma aresta, mais três blocos candidatos são avaliados a partir de um novo LDSP que será gerado. Por outro lado, quando o melhor casamento for encontrado em um vértice, cinco blocos candidatos adicionais são analisados, como pode ser visto na Figura 3.6 (b) e na Figura 3.6 (c), respectivamente.



(a) LDSP e SDSP (b) LDSP em uma aresta (c) LDSP em um vértice

Figura 3.6: Algoritmo *Diamond Search* (FS).

3.3 Critérios de Similaridade

A avaliação das semelhanças entre um bloco do quadro atual e os blocos que constituem a área de busca no quadro de referência é dada a partir das diferenças existente entre eles. Neste contexto, para encontrar o melhor casamento entre os blocos utiliza-se um critério de similaridade que também pode ser denominado de critério de distorção. A distorção (ou diferença) é inversamente proporcional ao grau de similaridade existente entre os blocos em questão.

Diferentes critérios (ou medidas) podem ser adotados a fim de medir o grau de similaridade aos elementos comparados, como por exemplo, a diferença aritmética entre as regiões. Os critérios comumente utilizados nestas avaliações são: ***(i)*** Menor erro quadrático (*Mean Square Error* - MSE); ***(ii)*** Menor erro absoluto (*Mean Absolute Error* - MAE); ***(iii)*** Soma das diferenças Absolutas (*Sum of Absolute Differences* - SAD) (KUHN, 1999).

3.3.1 SAD (*Sum of Absolute Differences*)

Considerando sua simplicidade, o critério de similaridade mais utilizado para algoritmos de Estimação de Movimento e utilizado neste trabalho é o cálculo de SAD. O SAD calcula as distorções das regiões comparadas a partir da soma das diferenças absolutas. Calculado para cada bloco do quadro atual e cada bloco candidato na área de busca, a partir da soma *pixel* por *pixel* das diferenças absolutas:

$$SAD_{(x,y)} = \sum_{i=0}^{w-1} \sum_{j=0}^{h-1} |BlocoAtual(i,j) - BlocoCandidato(i,j)| \qquad (1)$$

A definição do cálculo de SAD é apresentada em (1), onde *w* se refere à largura e *h* à altura de ambos os blocos, candidato e atual. O bloco candidato é escolhido quando representa o menor valor de SAD, isto é, a menor diferença em relação ao bloco atual. A distância entre as coordenadas *(x,y)* do bloco que representa o menor valor de SAD em relação a posição do bloco atual é o vetor de movimento.

3.4 Considerações Finais sobre o Capítulo

Este capítulo introduziu o principal objeto deste trabalho, a Estimação de Movimento, os algoritmos de busca e o critério de similaridade adotado. A Estimação de Movimento explora de imagens temporalmente vizinhas com o objetivo de identificar as semelhanças existentes entre elas a fim de mapeá-las através de vetores de movimento diminuindo o volume de dados inerente à codificação de vídeo. Tanto o algoritmo ótimo, conhecido como Algoritmo *Full Search*, quanto o algoritmo sub-ótimo, o Algoritmo *Diamond Search* foram brevemente descritos neste capítulo. Para avaliação das redundâncias entre os quadros foi utilizado o critério de similaridade denominado SAD, o qual calcula a soma dos valores absolutos das diferenças entre os *pixels*, e é comumente utilizado pela comunidade pela sua simplicidade.

A Estimação de Movimento é responsável pela maior parcela de ganhos em um codificador de vídeo digital. No entanto, estes ganhos são atingidos ao custo de uma elevada complexidade computacional, a qual dificulta a codificação de vídeos de alta definição em software em tempo real. Neste cenário, a proposta introduzida nesta dissertação trabalha, principalmente, na avaliação da codificação de vídeo em tempo real a partir da utilização de placas de vídeos. O próximo capítulo apresenta soluções paralelas e distribuídas da ME, com ênfase em GPUs.

4 PROPOSTA DE ESTIMAÇÃO DE MOVIMENTO EM GPU

Neste capítulo será descrita a implementação da Estimação de Movimento em GPU desenvolvida durante os dois anos de mestrado para dois algoritmos de busca: *Full Search* e *Diamond Search*. Além disso, apresenta a metodologia de desenvolvimento (ambientes de programação, plataformas de execução, fluxogramas, pseudocódigos, entre outros). Em especial, esta paralelização trata de uma solução para ME baseada em GPU utilizando a arquitetura CUDA. Como base comparativa, foram desenvolvidas soluções alternativas considerando plataformas *multi-core* (OpenMP) e distribuídas (MPI) que também serão brevemente descritas neste capítulo.

4.1 Metodologia de Desenvolvimento

Um pseudocódigo genérico é apresentado no Algoritmo 1 e representa a ME sequencial, independente do algoritmo de busca adotado e do ambiente de execução. Como base comparativa à ME em GPU, foco desta dissertação, diferentes paradigmas computacionais foram também utilizados no desenvolvimento da ME: *distribuído* e *multi-core*. Desta forma, o desenvolvimento das diversas soluções propostas nesta dissertação foi baseado no algoritmo abaixo.

Algoritmo 1: Estimacao_de_Movimento_Generica()

Parâmetros: *YUV_Video, NumQuadros, TamAreaBusca, TamBloco, largura, altura*
Funcionalidade: Estimação de Movimento para diferentes dimensões de área de busca, resoluções de vídeo e algoritmos de busca
Retorno: Menor Valor de SAD e Vetor de Movimento para cada bloco atual

```
1:
2: largura ← LarguraQuadro(YUV_Video);
3: altura ← AlturaQuadro(YUV_Video);
4.
5: AreaBusca : tamanho (TamAreaBusca, TamAreaBusca); //vetor usado para alocar area de busca
6: BlocoAtual : tamanho (TamAreaBusca, TamAreaBusca); //vetor usado para alocar os blocos atuais
7: SAD: tamanho(largura/TamBloco, altura/TamBloco);   // armazena os valores de SAD
8: MV: tamanho (largura/TamBloco, height/TamBloco);   //armazena os vetores de movimento
9: Para cada bloco que compoe o quadro atual
10:                 carrega BlocoAtual;
11:                 carrega AreaBusca;
12:                 [SAD, MV] = Algoritmo_de_Busca(BlocoAtual, TamBloco, AreaBusca);
13: fim Para
14: retorno {SAD, MV}
```

Como se pode observar no Algoritmo 1, tem-se uma sequência de vídeo (formato *.yuv*) obtida como entrada principal da aplicação, da qual são extraídos os parâmetros de altura e largura que serão considerados. A partir do tamanho da área de busca e o tamanho do bloco de vídeo (também obtidos como parâmetros de entrada) são alocados vetores e variáveis que serão utilizados no decorrer da execução da ME. Em resumo, o Algoritmo 1 tem como base principal um laço onde cada iteração se refere à execução

da Estimação de Movimento (independente do algoritmo de busca) para cada bloco atual.

Tendo em vista a complexidade computacional da ME, em especial, na tentativa de explorar o potencial de paralelismo inerente às GPUs, neste trabalho são propostas soluções paralelas e distribuídas para dois algoritmos de busca: para o algoritmo exaustivo *Full Search* (FS) e para o algoritmo rápido *Diamond Search* (DS). Em resumo, a estratégia de paralelização da ME quando da utilização de diferentes paradigmas computacionais e a metodologia utilizada são ilustradas na Figura 4.1.

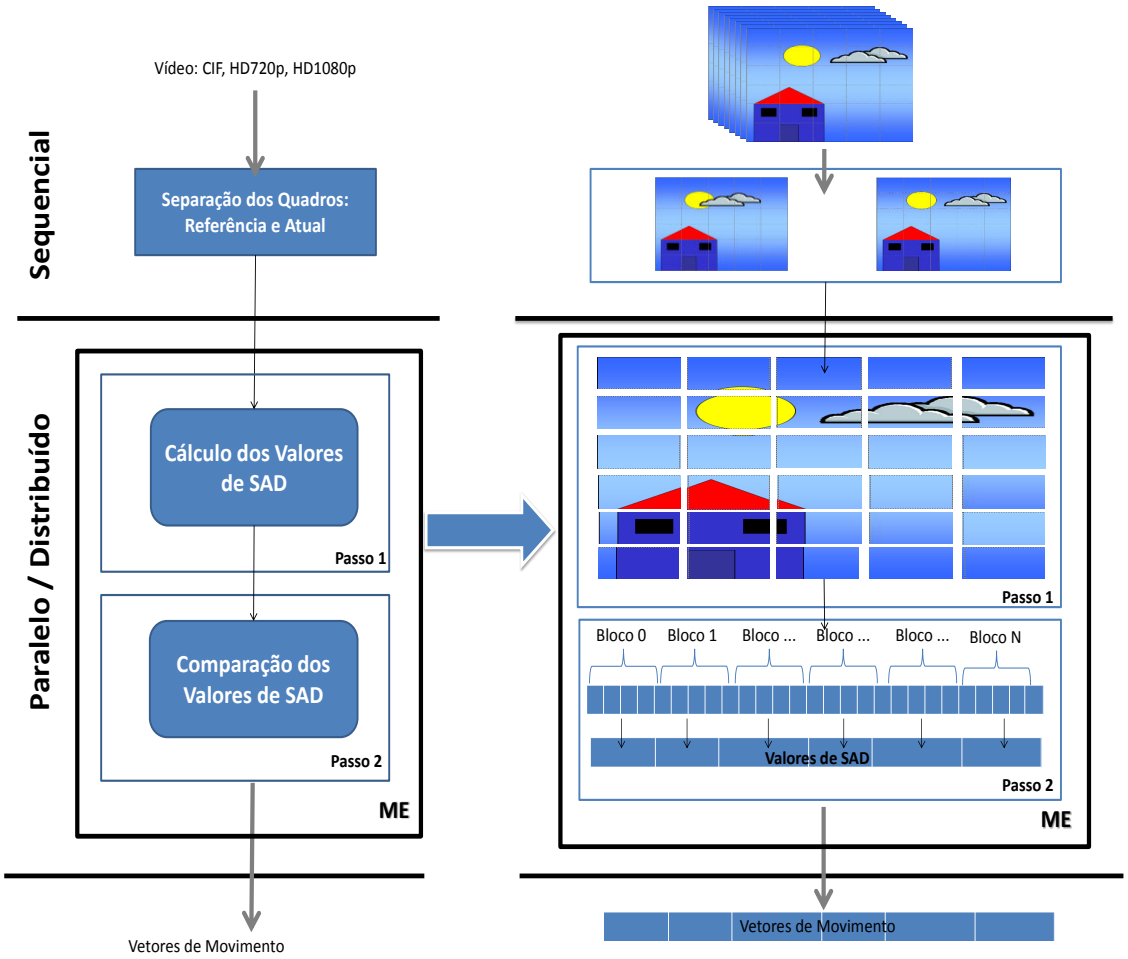


Figura 4.1: Algoritmo proposto referente à Estimação de Movimento paralela/distribuída.

A metodologia proposta para a implementação dos dois algoritmos de busca FS e DS independente do ambiente em que serão executados, é apresentada na Figura 4.1. A metodologia está ilustrada de forma a destacar a função da CPU e, por sua vez, a função da GPU no processo referente à ME, o qual é composto por duas etapas: ***(i)*** cálculo dos valores de SAD para cada bloco atual e ***(ii)*** a busca pelos menores valores de SAD e seus respectivos vetores de movimento. Além disso, a Figura 4.1 proporciona uma visão geral do nível de paralelismo adotado na implementação destas aplicações, onde pode-se perceber que a principal característica trata a execução dos blocos atuais de forma simultânea.

De acordo com a Figura 4.1, o parâmetro de entrada referente à ME proposta, independente do ambiente de execução, é uma sequência de vídeo que pode estar em

diferentes resoluções (CIF, HD720p e HD1080p). Inicialmente, esta sequência é dividida de modo a obter os quadros de referência e atual, considerando que o primeiro quadro do vídeo seja um quadro reconstruído previamente. Nas soluções GPGPU esta etapa é realizada pela CPU e, logo após, estes quadros são enviados para a GPU (no caso da solução distribuída, são enviados os nodos que formam o ambiente, e nas *multi-core* para as *threads* que compõem os núcleos de processamento). De posse de todos os dados necessários, a ME é devidamente executada de acordo com o Algoritmo 1, sendo composta por duas etapas: cálculo dos valores de SAD e comparação deste valores em busca do menor valor. Por fim, como resultado desta aplicação, têm-se os vetores de movimentos.

Além da Figura 4.1 ilustrar a metodologia utilizada, mostrando a ordem de execução das tarefas inerentes à ME proposta nesta dissertação, é possível visualizar com maior clareza conceitos básicos de codificação de vídeo e programação em CUDA. Além disso, em especial, salienta-se o nível de paralelismo proposto em GPU nesta dissertação.

A partir de uma visão clara, de que um bloco de GPU (na Figura 4.1 referentes a Bloco 0, Bloco 1...Bloco N) abrange diferentes e diversos blocos de vídeo (na Figura 4.1 ilustrados a partir da divisão de um quadro de vídeo no Passo 1) , tem-se o nível de paralelismo proposto nesta dissertação. O nível de paralelismo adotado em ambas a implementações GPGPU propostas nesta dissertação é o nível de blocos. A paralelização neste nível consiste em resumo, em processar a ME dos blocos que formam um quadro de vídeo de forma paralela. Ao contrário de muitos trabalhos propostos na literatura, os quais tratam de um paralelismo em nível de *pixels*. Os valores de SAD calculados são armazenados linearmente (de acordo com os blocos atuais) e como segunda etapa, tem-se a busca paralela pelo menor valor de SAD de cada bloco atual.

A biblioteca *Thrust* (THRUST, 2012) foi utilizada para manipulação dos dados entre os dispositivos dispostos, bem como para utilização de alguns algoritmos da biblioteca. A *Thrust* é um container *C++*[5], desenvolvido especialmente para arquitetura CUDA, a fim de viabilizar a criação e o desenvolvimento de aplicações GPGPU que visam a utilização de dispositivos de processamento gráficos como alternativa de co-processamento. Esta biblioteca introduz um elevado nível de abstração na programação dos processadores, por exemplo, provendo a transferência de dados entre CPU e GPU (e vice versa) de forma simples e transparente ao usuário quando comparada com os meios comuns. Além disso, alguns algoritmos existentes na biblioteca foram utilizados nas implementações propostas: ***(i)*** *reduce*: para comparar valores de modo a encontrar o menor SAD; ***(ii)*** *fill*: para inicializar vetores com um determinado valor. Todos os mecanismos da biblioteca *Thrust* utilizados nas versões GPGPU desenvolvidas nesta dissertação foram amplamente avaliados (em termos eficiência em tempo de processamento) e, se mostraram altamente relevantes neste cenário (em termos de desempenho) quando comparados com soluções análogas (métodos clássicos de repetição – *for* – e comparação – *if*), isto é, que mantem a mesmo objetivo na implementação, mas que não incluem os artifícios (ou técnicas) dispostos pela biblioteca.

[5] É uma estrutura de dado existente na linguagem C++ que armazena uma coleção de objetos. Eles são implementados como grandes classes que permitem uma considerável flexibilidade no suporte a tipos de elementos. .

O cálculo de SAD foi considerado como critério de similaridade em todas as soluções apresentadas nesta dissertação, a fim de buscar pelo melhor casamento entre os blocos. O cálculo de SAD é comumente utilizado pela comunidade científica devido à simplicidade imposta na definição que representa este critério de similaridade na comparação entre os blocos candidatos.

Para melhor entendimento das implementações que serão descritas, os algoritmos FS e DS são representados em suas versões sequenciais através dos Algoritmos 2 e 3, respectivamente. Os algoritmos ilustrados a seguir foram utilizados como base para o desenvolvimento das paralelizações apresentadas neste trabalho.

A versão do FS apresentada no Algoritmo 2, é basicamente composta por duas etapas: ***(i)*** cálculo dos valores de SAD para todos os blocos que compõe a área de busca no quadro de referência; ***(ii)*** comparação dos valores de SAD em todos os blocos candidatos, a fim de encontrar o melhor casamento (menor valor de SAD) e gerar os respectivos vetores de movimento.

Algoritmo 2 : Algoritmo_Full_Search()

Parâmetros : *BlocoAtual, TamBloco, AreaBusca*
Funcionalidade: Algoritmo *Full Search* para diferentes dimensões de área de busca e resoluções de vídeo
Retorno: Menor Valor de SAD e Vetor de Movimento

```
1.   SAD ←∞;                                  //armazena valor máximo de SAD
2.
3.  Para todo pixel que compõe a área de busca
4.    acumulador ← 0;
5.    Para todo pixel que compõe bloco
6.       acumulador ←  acumulador + |AreaBusca – BlocoAtual|;          //calcula valor de SAD
7.    fim Para
8.  fim Para
9.
10:  Se acumulador < valor_atual_SAD
11:        SAD ← acumulador
12:        MV ← posicao;        //MV recebe posição do bloco que aponta para o menor valor de SAD
13:  fim Se
14:
15:  retorno {SAD, MV}
```

Por sua vez, o algoritmo *Diamond Search* (Algoritmo 3) é também composto por duas etapas: ***(i)*** a busca pelo SAD no centro do maior diamante (do inglês, LDSP); ***(ii)*** etapa de refinamento, onde é realizada uma busca pelo menor SAD em um diamante menor (do inglês, SDSP). Após a execução destas duas etapas, onde é realizada a busca pelos valores em forma de diamante, LDSP e SDSP, os valores de SAD e seus respectivos vetores de movimento para todos os blocos que compõe o quadro atual são

gerados.

Algoritmo 3 : Algoritmo_Diamond_Search()

Parâmetros : *Video, TamBloco, AreaBusca, BlocoAtual*
Funcionalidade: Algoritmo *Diamond Search* para diferentes dimensões de área de busca e resoluções de vídeo
Retorno: Menor Valor de SAD e Vetor de Movimento

```
1.   SAD ←∞;                                  //armazena valor máximo de SAD
2.
3.   Para todo bloco que compoe o quadro atual
4.        SAD ←calculo_LDSP(TamBloco, BlocoAtual);          //calculo do diamante maior
5.        Enquanto (SAD não corresponder a posição do centro do diamante)
6.             SAD ←calculo_LDSP(TamBloco, BlocoAtual);
7.        fim Enquanto
8.
9.        refinamento ← calculo_SDSP(TamBloco, linha, coluna);   //calculo do diamante menor
10.       Se (refinamento < SAD)
11.            SAD ← refinamento;
12.            MV ← posicao; //MV recebe posição do bloco que aponta para o menor SAD
13.       fim Se
14.  fim Para
15.  retorno {SAD, MV}
```

As soluções *multi-core* dos algoritmos FS e DS utilizam a API OpenMP e, por sua vez, as soluções distribuídas fazem o uso da biblioteca MPI. Ambas as alternativas desenvolvidas serão brevemente descritas a seguir.

A programação paralela trata do gerenciamento de tarefas que operam simultaneamente sob arquiteturas distintas, por exemplo, arquiteturas SIMD e MIMD. Segundo a taxonomia de *Flyn* (FLYNN, 1972), as arquiteturas SIMD (*Single Instructions, Multiple Data*) apresentam um único fluxo de instrução sob os diversos dados, enquanto as arquiteturas MIMD (*Multiple Instruction, Multiple Data*), estão baseadas em diferentes fluxos e múltiplos dados. Desta forma, em cada ambiente de programação paralelo conceitos distintos para uma determinada tarefa se fazem presentes. O ambiente GPGPU se concentra em arquiteturas do tipo SIMD e suas tarefas são representadas por elementos denominados *threads*, as quais são gerenciadas pela API que provê a programação destes dispositivos. Na API OpenMP, também baseada em instruções SIMD (entre outras como MIMD) e no uso de memória compartilhada, estas unidades também são chamadas de *threads*, porém são gerenciadas individualmente pelo escalonador de tarefas do sistema operacional. Sendo assim, pode-se dizer que as *threads* em OpenMP possuem uma sobrecarga relativamente maior que as *threads* em GPU que são gerenciadas pelas *warps*. Por sua vez, em MPI as tarefas são denominadas de processos e estão baseadas na arquitetura MIMD considerando memória distribuída. A comunicação entre os processos ocorre através de troca de mensagens.

A principal diferença entre threads e processos está relacionada com o conceito de cada um destes elementos, onde *threads* são linhas, ou contextos, de execuções escalonadas pela CPU que compartilham o mesmo espaço de memória, e por outro lado, um processo é um conjunto de recursos alocados para executar um programa de forma independente, isto é, com suas próprias variáveis, pilhas e espaço de memória (TANENBAUM, 2007). Sendo assim, quando da comparação dos três paradigmas abordados, salienta-se que os processos utilizados em MPI apresentam a mais alta sobrecarga de gerenciamento (TANENBAUM, 2007), seguidos de OpenMP e GPU é o paradigma que representa o mais leve custo neste contexto.

4.1.1 Estimação de Movimento em OpenMP

Nesta seção, serão descritas as versões paralelas da Estimação de Movimento (FS e DS) utilizando processadores *multi-core*. Em resumo, a paralelização da ME neste paradigma foi baseada no Algoritmo 1 apresentado na Seção 4.1. Para este fim, foi utilizada uma API baseada em memória compartilhada, denominada OpenMP, a qual permite a exploração do paralelismo inerente aos processadores convencionais. A API OpenMP aborda tanto instruções SIMD quando MIMD, ou seja, opera com instruções iguais a um mesmo dado ou instruções diferentes. Neste trabalho quando se refere a API OpenMP, serão abordadas instruções SIMD. A paralelização da ME em OpenMP teve como principal objetivo viabilizar uma base comparativa da versão da ME em GPU quando da utilização de processadores *multi-core*.

A API OpenMP suporta as linguagens C/C++ e, pelo fato de utilizar processadores convencionais, apresenta uma ampla disponibilidade de compiladores. Além disso, viabiliza algumas características importantes para o programador, como por exemplo, um alto nível de abstração, portabilidade e escalabilidade.

A implementação dos algoritmos FS e DS em OpenMP requer os mesmos parâmetros que a ME em GPU: ***(i)*** entrada: uma sequência de vídeo (na extensão *.yuv*) em diferentes resoluções, CIF, HD720p ou HD1080p; ***(ii)*** saída: vetores de movimento. Em resumo ambas as soluções voltadas para este ambiente (FS e DS) consistem em dois grandes laços: ***(i)*** na verificação de todos os quadros atuais e blocos que compõe a sequência de vídeo; ***(ii)*** e na execução da ME, onde são aplicados os algoritmos de busca, para todos os blocos que formam um quadro atual.

Em ambas as implementações propostas (FS e DS) foi utilizada uma diretriz denominada *pragma omp for parallel*, a qual, em resumo, faz com que as iterações da estrutura de repetição (localizada logo abaixo da diretiva) sejam executadas simultaneamente. Estas iterações são distribuídas de forma coerente (equilíbrio entre quantidade de operações e recurso alocado) entre as *threads* que formam o grupo criado na região paralela. Em outra palavras, a API OpenMP cria uma pilha de tarefas (blocos a serem processados) que são executadas por um conjunto finito de *threads*.

4.1.1.1 Algoritmo Full Search em OpenMP

As alterações em relação ao algoritmo sequencial apresentado tem como foco a paralelização do FS em OpenMP a partir do processamento de cada bloco atual por uma *thread*. As modificações realizadas para atingir este objetivo foram baseadas nos Algoritmos 1 e 2 apresentados anteriormente. Sendo assim, no Algoritmo 4 é ilustrado um pseudocódigo que representa a implementação do algoritmo FS em OpenMP.

Algoritmo 4 : Full_Search_OpenMP()

Parâmetros : *BlocoAtual, TamBloco, AreaBusca*
Funcionalidade: Algoritmo *Full Search* para diferentes dimensões de áreas de busca e resoluções de vídeo
Retorno: Menor Valor de SAD e Vetor de Movimento

```
1.  SAD ←∞;                                  //armazena valor máximo de SAD
2.  Num_Threads ← Número Máximo de threads que o processador permitir;
3.
4. [OpenMP] diretriz que cria tarefas simultaneas a partir do Para
5. Para cada bloco que compoe o quadro atual
6.      [SAD, MV] = Algoritmo_Full_Search(BlocoAtual, TamBloco, AreaBusca);
7. fim Para
8. retorno {SAD, MV}
```

Como se pode perceber no Algoritmo 4, a execução da ME está baseada na execução de um laço que percorre todos os blocos que formam o quadro atual. Cada *thread* é responsável por uma iteração deste laço, ou seja, cada uma executa um bloco atual e depois de terminar sua execução é retirada da lista (pilha) de tarefas. Esta característica é dada a partir do *pragma omp parallel for* (já descrito anteriormente). Para cada bloco que compõe o quadro, um bloco atual e sua respectiva área de busca são carregados e assim, a função que realiza a execução do algoritmo FS. *(*veja Algoritmo 2*)* é chamada.

4.1.1.2 Algoritmo Diamond Search em OpenMP

O algoritmo DS em OpenMP foi implementado com base nos algoritmos sequenciais 1 e 3. As principais alterações estão representadas e destacadas no pseudocódigo apresentado no Algoritmo 5.

Algoritmo 5 : Diamond_Search_OpenMP()

Parâmetros : *Video, TamBloco, AreaBusca, BlocoAtual*
Funcionalidade: Algoritmo *Diamond Search* para diferentes dimensões de área de busca e resoluções de vídeo
Retorno: Menor Valor de SAD e Vetor de Movimento

```
1.    SAD ←∞;                                                    //armazena valor máximo de SAD
2.    [OpenMP] diretriz que cria tarefas simultaneas a partir do Para
3.    Para todo bloco que compoe o quadro atual
4.          SAD ←calculo_LDSP(TamBloco, BlocoAtual);             //calculo do diamante maior
5.          Enquanto (SAD não corresponder a posição do centro do diamante)
6.                SAD ←calculo_LDSP(TamBloco, BlocoAtual);
7.          fim Enquanto
8.
9.          refinamento ← calculo_SDSP(TamBloco, linha, coluna);   //calculo do diamante menor
10.         Se (refinamento < SAD)
11.               SAD ← refinamento;
12.               MV ← posicao; //MV recebe posição do bloco que aponta para o menor SAD
13.         fim Se
14.   fim Para
15. retorno {SAD, MV}
```

O algoritmo DS apresentado em OpenMP consiste em um laço que realiza o processamento de um bloco atual por *thread*. Esta paralelização é dada a partir da diretriz *pragma omp parallel for* visando a integridade dos dados a partir de uma proteção das variáveis que envolvem a estrutura de repetição de cada *thread*. Considerando uma estrutura de repetição para cada bloco atual, primeiramente é realizado o cálculo para o bloco candidato localizado na posição central do diamante, logo a seguir, é feita a execução do maior diamante, denominado LDSP (composto por nove blocos candidatos). Enquanto o menor valor de SAD entre dos blocos candidatos não for referente à posição central do diamante o algoritmo itera, de acordo com a posição de menor valor de SAD. Satisfeita esta condição, o refinamento é aplicado no bloco vencedor a partir da função referente ao cálculo do SDSP, menor diamante e constituído por 5 blocos candidatos.

4.1.2 Estimação de Movimento Distribuída utilizando MPI

Nesta seção serão descritas as implementações distribuídas dos algoritmos FS e DS em MPI. Em resumo, MPI provê uma memória distribuída e funcionalidades para que um ou mais processos possam se comunicar entre si a partir de troca de mensagens. Os processos formam os nodos, os quais compõem o ambiente distribuído, também

denominado de *cluster*[6]. O conjunto de instruções que compõe este paradigma computacional é o MIMD. A ME distribuída também está de acordo com o algoritmo proposto na Figura 4.1, sendo caracterizada pelos seguintes parâmetros: uma sequência de vídeo (*.yuv*) em diferentes resoluções como entrada e vetores de movimento como produto final.

O nível de paralelização, granularidade, adotada nas implementações dos algoritmos FS e DS em MPI estão baseadas na execução da ME de um bloco atual por processo. Sendo assim, no início dos algoritmos apresentados a seguir (Algoritmos 6 e 7), os quais ilustram um *pseudocódigo* dos algoritmos FS e DS em MPI, é realizado um cálculo para determinar quantos blocos atuais serão executados por cada processo. Considerando que cada processo é responsável por executar um bloco atual, primeiramente é realizado um cálculo que diz respeito à divisão do número de blocos atuais pelo número de processos disponíveis no *cluster*. O número de blocos que são executados por cada processo varia de acordo com a resolução do vídeo (CIF, HD720p ou HD1080p). A carga que cada processo irá ficar responsável é balanceada de modo a possibilitar que todos os processos realizem uma quantidade de blocos similar. O número máximo de processos disponível no ambiente distribuído é variável e alocado quando da execução da aplicação (por parâmetro). Desta forma, em resumo, os *pseudocódigos* apresentados descrevem as tarefas que cada processo irá executar.

Neste ambiente, um único processo é responsável por realizar a criação, contabilização e finalização do tempo de execução das aplicações, bem como alimentar os nodos com os dados necessários para realização da ME para cada bloco atual. Esta medida foi adotada de modo a diminuir o custo elevado de gerenciamento e comunicação entre os processos, como já mencionado anteriormente. A ME distribuída consiste em duas trocas de mensagens (por quadro atual). Estas trocas de mensagens dão-se entre o processo responsável pelo gerenciamento dos dados necessários para execução da ME (quadro atual e quadro de referência) e os demais processos alocados. A primeira mensagem enviada contém todos os blocos que constituem o quadro atual e, por sua vez, a segunda mensagem é constituída pelo respectivo quadro de referência.

4.1.2.1 Algoritmo Full Search em MPI

A paralelização e distribuição do algoritmo FS em MPI teve como base o Algoritmo 1, apresentado anteriormente. As alterações realizadas nesta versão estão devidamente destacadas no Algoritmo 6, o qual resume as principais alterações no algoritmo FS sequencial que viabilizaram a distribuição das tarefas entre os processos.

---

[6] Um *cluster* é formado um conjunto de computadores que operam simultaneamente em um sistema distribuído para executar uma determinada tarefa, ou mais.

Algoritmo 6 : Full_Search_MPI()

Parâmetros : *TamBloco, AreaBusca, Altura, Largura, Video*
Funcionalidade: Algoritmo *Full Search* para diferentes dimensões de área de busca e resoluções de vídeo
Retorno: Menor Valor de SAD e Vetor de Movimento

```
1.  SAD ←∞;                                                    //armazena valor máximo de SAD
2.
3.  [MPI] Início da rotina MPI;
4.  Se for o processo raiz
5.      carrega Video
6.  fim Se
7.
8.  Se for o processo raiz
9.      Para cada bloco que compoe o quadro atual
10.         carrega BlocosAtuais;
11.     fim Para
12. fim Se
13. [MPI] Envio de BlocosAtuais para demais processos
14. Se for o processo raiz
15.     carrega QuadroReferencia
16. fim Se
17. [MPI] Envio de QuadroReferencia para demais processos
18. Para cada bloco que compoe o quadro atual          //cada iteracao executada por um processo
19.         carrega AreaBusca;
20:         [SAD, MV] = Algoritmo_Full_Search(BlocosAtuais, TamBloco, AreaBusca);
21: fim Para
22: retorno {SAD, MV}
23. [MPI] Fim da rotina MPI;
```

Como já mencionado anteriormente, primeiramente é realizado um cálculo que faz um levantamento do número de processos alocados em relação ao número de blocos que devem ser executados. Este cálculo procura um equilíbrio de tarefas de forma similar entre cada processo, evitando que um ou mais processo fiquem sobrecarregados de tarefas, enquanto os demais podem estar ociosos.

Para garantir a eficiência do paralelismo do FS em MPI, visando o alto desempenho da aplicação, um único processo (processo raiz) é responsável por carregar os parâmetros necessários para a execução do algoritmo. Este processo raiz faz a leitura e o armazenamento da sequência de vídeo obtida como entrada sequencialmente, dos blocos que compõe um quadro atual e do quadro de referência.

Após esta etapa inicial, os dados carregados pelo processo são devidamente enviados aos demais nodos, a partir de duas trocas de mensagens (*broadcasts*). Este protocolo de comunicação é repetido para cada bloco atual que compõe a sequência de vídeo em questão. De posse de todos os argumentos necessários, todos os processos alocados encontram-se aptos a executar a ME FS para cada bloco que compõe o quadro atual.

4.1.2.2 Algoritmo Diamond Search em MPI

O pseudocódigo que representa o DS em sua versão distribuída a partir de MPI é apresentado no Algoritmo 7. Esta descrição é semelhante ao Algoritmo 5, o qual ilustra o pseudocódigo referente à paralelização do DS em OpenMP, com destaque em algumas alterações importantes.

Assim como já mencionado, cada processo é responsável pela execução do de um bloco atual. Inicialmente, é realizado um cálculo que diz respeito à divisão do número de blocos atuais que constituem o quadro em questão de acordo com o número de processos alocados no *cluster*. Este cálculo visa evitar a ociosidade dos recursos computacionais (processos - nodos) alocados. O processo raiz também é responsável

por carregar a sequência de vídeo de entrada e separar os respectivos quadros atuais e referências sequencialmente. Extraídos os dados essenciais para execução do algoritmo, é realizada uma comunicação entre este processo raiz e os demais, enviando os quadros atuais e referências a partir de duas comunicações (dois *broadcasts*): uma com um vetor que compõe todos os blocos atuais, e outra com o respectivo quadro de referência.

Tendo em vista que todos os nodos receberam os quadros de maneira correta, a realização do DS pode ser paralelizada e executada de forma distribuída entre os processos alocados.

A continuidade da execução do algoritmo DS distribuído é semelhante às demais versões descritas, onde todos os blocos atuais são percorridos e executados por cada processo. O primeiro objetivo do algoritmo é encontrar o menor valor de SAD na posição central do diamante (lembrando que o LDSP possui nove posições). Enquanto a posição do menor valor de SAD não for referente à posição central, o algoritmo é iterado. Quando a posição do menor valor de SAD for a posição central, o refinamento do algoritmo (SDSP composto por cinco posições) é aplicado para finalização da execução da ME para o bloco atual em questão.

Algoritmo 7 : Diamond_Search_MPI()

Parâmetros : *Video, TamBloco, AreaBusca, Altura, Largura*
Funcionalidade: Algoritmo *Diamond Search* para diferentes dimensões de área de busca e resoluções de vídeo
Retorno: Menor Valor de SAD e Vetor de Movimento

```
1.  SAD ←∞;                                                    //armazena valor máximo de SAD
2.  [MPI] Início da rotina MPI;
3.
4.  Se for o processo raiz
5.      carrega Video
6.      carrega QuadroAtual;
7.      carrega QuadroReferencia
8.  fim Se
9.
10. [MPI] Envio de QuadroAtual para demais processos
11. [MPI] Envio de QuadroReferencia para demais processos
12.  Para cada bloco que compoe o quadro atual                  //cada iteracao executada por um processo
13.      SAD ←calculo_LDSP(TamBloco, BlocoAtual);              //calculo do diamante maior
14.      Enquanto (SAD não corresponder a posição do centro do diamante)
15.              SAD ←calculo_LDSP(TamBloco, BlocoAtual);
16.      fim Enquanto
17:
18:      refinamento ← calculo_SDSP(TamBloco, linha, coluna);  //calculo do diamante menor
19:      Se (refinamento < SAD)
20:              SAD ← refinamento;
21:              MV ← posicao; //MV recebe posição do bloco que aponta para o menor SAD
22:      fim Se
23:  fim Para
24: retorno {SAD, MV}
25: [MPI] Fim da rotina MPI;
```

4.2 Estimação de Movimento em GPU utilizando a arquitetura CUDA

Entre as diferentes opções arquiteturais já mencionadas, este trabalho considera principalmente a utilização de GPU como alternativa de processamento de propósito geral (GPGPU), em especial a API CUDA é utilizada.

A API CUDA é uma tecnologia comumente aceita e bastante utilizada na comunidade científica. Esta arquitetura viabiliza a programação dos processadores gráficos da empresa NVIDIA para execução de aplicações de propósito geral. CUDA é

baseada em um conjunto de instruções SIMD (*Single Instruction, Multiple Data*), a qual é adequada para algoritmos de busca da ME considerando a exploração máxima de dados independentes (principalmente no algoritmo FS) em conjunto com o massivo paralelismo provido por este tipo de tecnologia. Uma API também flexível e suportada por todas as GPUs do mercado surgiu recentemente, a OpenCL (OPENCL, 2012). No entanto, o status atual da API CUDA encontra-se em estágio avançado de maturação e suporte em nível de ferramentas, por esta razão optou-se pelo uso de CUDA no desenvolvimento deste trabalho.

O hardware utilizado no desenvolvimento das aplicações GPGPU propostas nesta dissertação é composto por: uma CPU e uma GPU da empresa NIVIDA com suporte à arquitetura CUDA. As soluções propostas para ME paralelizada em GPU consistem nas implementações dos algoritmos de busca *Full Search* e *Diamond Search*.

Em resumo, estas aplicações estão baseadas no algoritmo proposto na Figura 4.1, o qual tem as seguintes características gerais: ***(i)*** Parâmetro de Entrada: sequência de vídeo em diferentes resoluções (CIF, HD720p e HD1080p); ***(ii)*** Funcionalidade da Aplicação: execução da Estimação de Movimento para diferentes algoritmos de busca (FS e DS; ***(iii)*** Resultado final: vetores de movimento e respectivos valores de SAD.

Neste cenário, existem duas comunicações importantes entre o hardware considerado, CPU e GPU, em ambas as versões da ME proposta, ME FS e ME DS: ***(i)*** CPU → GPU: uma comunicação para o envio dos quadros de referência e atual, separados em CPU e enviados para a GPU, a partir da sequência de vídeo obtida como parâmetro de entrada; ***(ii)*** GPU → CPU: segunda comunicação a qual consiste no envio de resultados finais da aplicação (vetores de movimento e valores de SAD) da GPU para CPU. Todos os dados que trafegam entre os dois dispositivos são armazenados e carregados da memória global da placa de vídeo.

As comunicações existentes entre o núcleo de processamento central (CPU) e a arquitetura utilizada como alternativa de co-processamento (GPU) em aplicação GPGPU é um fator que requerer extremo cuidado, conforme será verificado no Capítulo 6. Em especial, na codificação de vídeo onde as tarefas são aplicadas para diferentes e diversos quadros digitais. Este cuidado se deve aos custos computacionais inerentes a estas comunicações (tanto de CPU para GPU, quanto de GPU para CPU), a qual pode vir a se tornar o gargalo da aplicação em termos de desempenho. A importância deste fator é justificada a partir da existência de diversos trabalhos que podem ser encontrados na literatura que tratam desta questão, como por exemplo, os trabalhos apresentados por (SCHWALB, 2009) e (CHENG, 2010). Por estas razões, a ME em GPU apresentada neste trabalho concentra suas tarefas em apenas duas comunicações (uma da CPU para GPU e outra da GPU para CPU) para cada quadro atual, contendo os dados necessários para que a funcionalidade da aplicação seja satisfeita.

Conforme os conceitos apresentados no Capítulo 2 se pode observar que em ambos os cenários, API CUDA e Codificação de Vídeo, existe um conceito em comum denominado ‘bloco’, porém com diferentes funcionalidades em cada contexto. Assim, para uma melhor compreensão dos algoritmos implementados em CUDA que serão descritos neste capítulo, foi utilizada a nomenclatura que é apresentada na Figura 4.2.

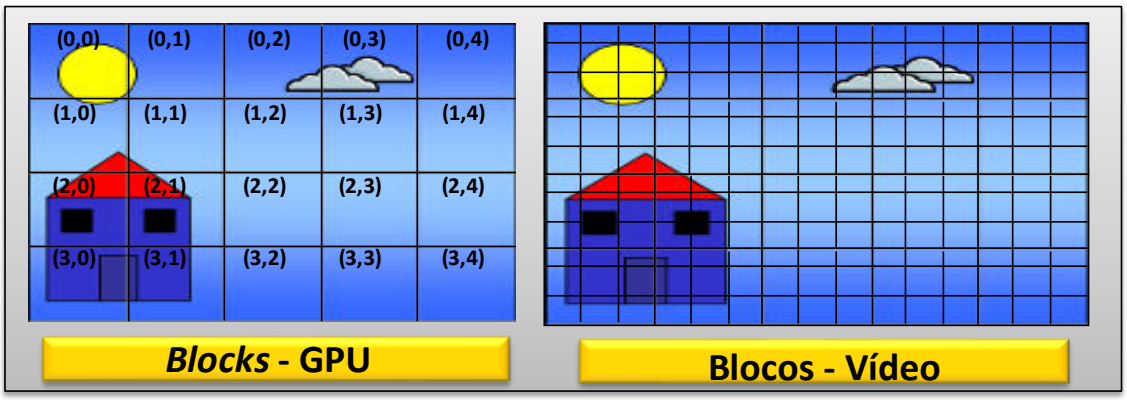


Figura 4.2: Distinção de Bloco de Vídeo e Bloco de GPU: Bloco (conjunto de *4x4 pixels*) *x Block* (conjunto de *threads*).

Na Figura 4.2 pode-se perceber a distinção entre o conceito nos diferentes cenários apresentados. Neste texto, os blocos bidimensionais de GPU compostos de *threads* serão denominados *blocks* e podem ser responsáveis por um ou mais blocos de vídeo que compõe um quadro e, por sua vez, os blocos se referem à divisão do quadro digital, onde neste trabalho se tratam de *4x4 pixels*. Em um contexto geral do trabalho, esta estrutura também pode ser visualizada na Figura 4.1, apresentada anteriormente.

A paralelização da ME GPGPU, tanto no FS quanto no DS, está baseada na hierarquia de processamento de CUDA apresentada no Capítulo 2. Nesta hierarquia citam-se os principais conceitos: ***(i)*** *threads*: unidade básica de processamento em GPU; ***(ii)*** *blocks*: conjunto de threads; ***(iii)*** *grid*: conjunto de *blocks*. Além destes conceitos, salienta-se que a ME em GPU conta com um *kernel* para cada algoritmo proposto, onde um *kernel* é o procedimento responsável pela execução dos algoritmos de busca da ME em CUDA (FS e DS).

4.2.1 Algoritmo *Full Search* em GPU

O algoritmo *Full Search* (FS) é também conhecido como algoritmo ótimo, em termos de qualidade de vídeo e eficiência em codificação, a partir da busca exaustiva no quadro de referência (deslocando-se *pixel* por *pixel* no quadro de referência) pelo melhor casamento entre os blocos (maior similaridade). Este algoritmo sempre encontra o resultado ideal, o qual representa o melhor vetor de movimento, porém para este fim é imposta uma elevada complexidade computacional devido à força bruta da busca exaustiva utilizada. A principal motivação para implementação do algoritmo FS é que apesar de sua exaustividade este algoritmo não apresenta dependência entre os dados, fato pelo qual é de extrema relevância na computação paralela.

Para um compreendimento inicial e geral do algoritmo FS GPGPU proposto, a Figura 4.3 apresenta um diagrama que ilustra o comportamento do FS implementado em CUDA para um bloco atual.

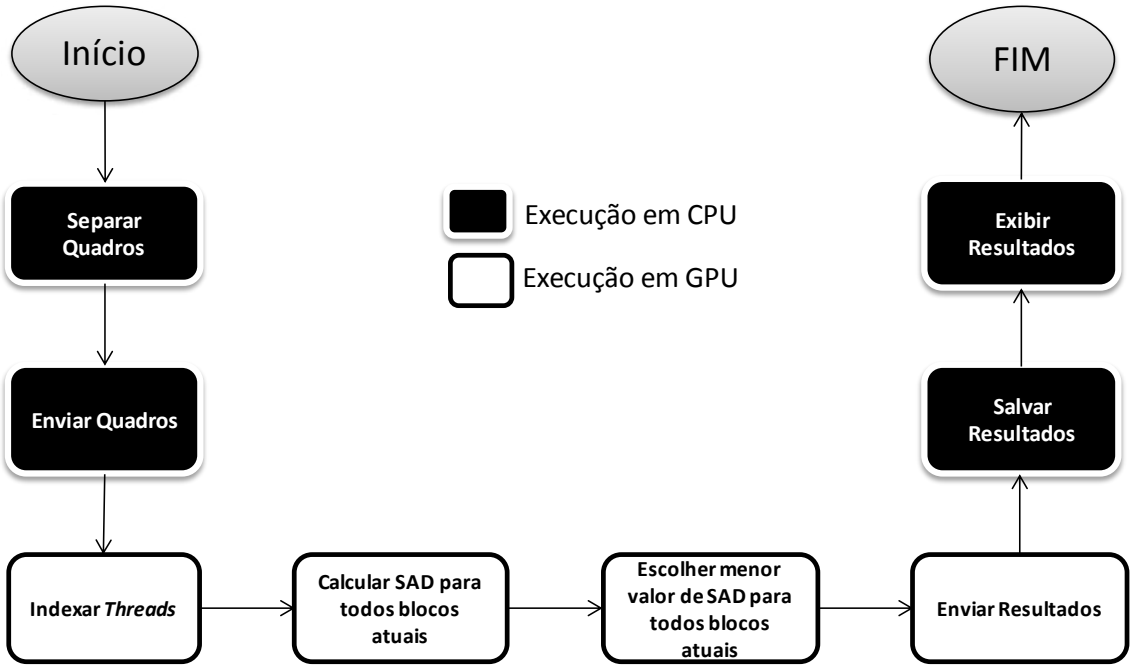


Figura 4.3: Fluxograma do Algoritmo *Full Search* em GPU.

Primeiramente, os quadros são separados em CPU sequencialmente, de modo a obter um quadro atual e seu respectivo quadro de referência por vez. Estes quadros são separados a partir da sequência de vídeo obtida como entrada da aplicação, lembrando que esta sequência pode estar em diferentes resoluções (CIF, HD720p e HD1080p).

A seguir, os quadros são enviados para GPU de modo a obter todos os dados necessários para execução do FS (quadro atual + quadro de referência). Esta comunicação é dada da memória RAM da CPU para a memória global da GPU. Na GPU, as *threads* são posicionadas no início de cada bloco atual para que a execução do quadro possa ser realizada concorrentemente. Este posicionamento das threads no algoritmo FS em CUDA refere-se a um *block* por bloco atual, isto é, um conjunto de *threads* é responsável por executar em paralelo a ME de um bloco atual, onde cada uma delas calcula um valor de SAD para um bloco.

Obtendo as devidas posições no quadro atual, as mesmas são mapeadas para o quadro de referência, onde as buscas pelos menores valores de SAD são realizadas simultaneamente. Todos os blocos candidatos são executados em paralelo e, seus valores de SAD são armazenados linearmente em único vetor (bloco a bloco atual). Este vetor é enviado para CPU a partir de uma comunicação entra a memória global da placa de vídeo e a memória RAM da CPU. Na CPU é acionada a biblioteca *Thrust*, em especial o algoritmo *reduce*, o qual é responsável por realizar a busca pelos menores valores em paralelo, que correspondem a maiores similaridades, para cada bloco atual.

Por fim, é obtido um vetor resultante com todos os valores de SAD vencedores para cada bloco atual e seus respetivos vetores de movimento.

O FS é um algoritmo altamente regular e apresenta um elevado potencial para processamento paralelo, uma vez que a busca pelo melhor casamento entre os blocos vizinhos não apresenta nenhuma dependência de dados. Desta forma, a paralelização do algoritmo FS em CUDA, já resumida no fluxograma, está ilustrada na Figura 4.4.

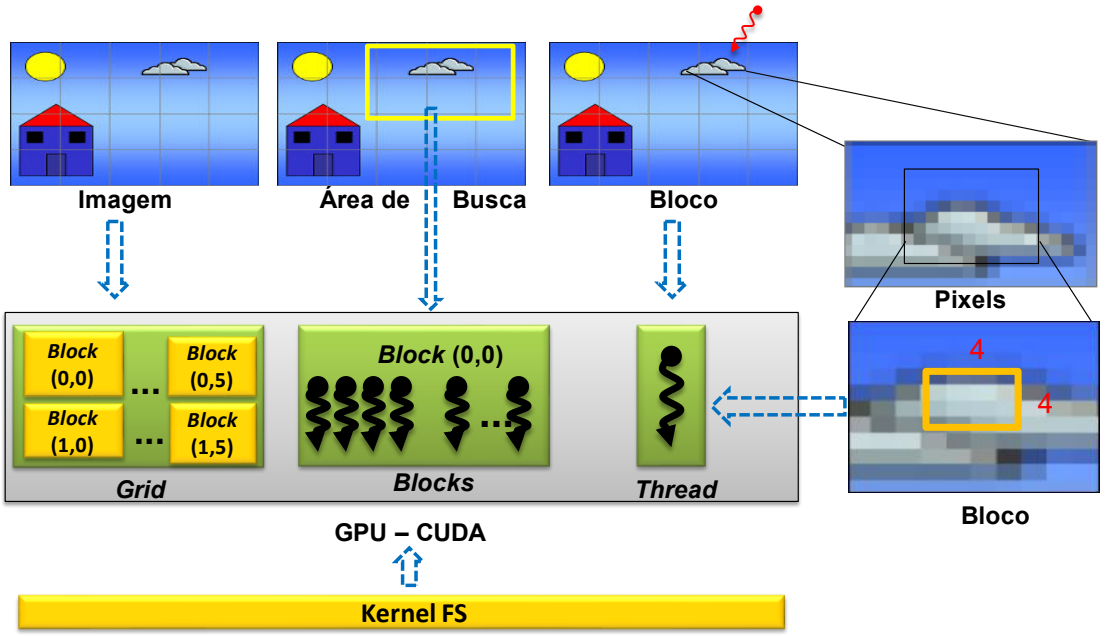


Figura 4.4: Modelo de Programação CUDA – Alocação Algoritmo *Full Search*.

De acordo com Figura 4.4, pode-se perceber o nível de paralelismo adotado na implementação do FS em CUDA. Este nível de paralelismo é em nível de blocos, onde destaca-se que um *block* é responsável por um bloco atual. Sendo assim, esta aplicação está baseada nos seguintes parâmetros: ***(i)*** *threads*: cada *thread* alocada para execução deste algoritmo é responsável por um bloco (vídeo) de dimensão *4x4 pixels* no quadro de referência. Um bloco de vídeo de *4x4 pixel* é a base para operações que envolvem a ME, permitindo um movimento de maior precisão quando comparado com um macrobloco (*16x16 pixels*); ***(ii)*** *block*: o tamanho do *block* na implementação deste algoritmo varia de acordo com a dimensão da área de busca e é responsável por um bloco atual, por fim, ***(iii)*** *grid*: o tamanho da *grid* está diretamente relacionada com a resolução da sequencia de vídeo obtida como parâmetro de entrada e com a quantidade de blocos (*4x4*) atuais inerentes a esta dimensão.

Além disso, pode-se observar que um quadro da sequência de vídeo está diretamente relacionado com a *grid* (que irá ser alocada), em termos de dimensão. Este fato ocorre, pois uma *grid* (ambiente de paralelização em GPU) deve estar apta a suportar toda carga de dados do problema, onde neste caso se refere a um quadro atual por vez.

A relação direta entre a dimensão do *block* e da área de busca da ME em questão também pode ser visualizada na Figura 4.4. A decisão por esta analogia, em termos de dimensão, deu-se porque no FS a complexidade computacional é dependente do tamanho da área de busca. Desta forma, o tamanho do *block* deve conter um número razoável (número mínimo de *threads* disponíveis em um ciclo de relógio/*clock* para executarem a quantidade de blocos candidatos do bloco atual em questão) de recursos computacionais (*threads*) para que o paralelismo dos dados seja realizado com eficiência. Por esta razão, a dimensão do *block* varia de acordo com a dimensão da área de busca, ou seja, quanto maior o tamanho do *block* mais *threads* são alocados e, quanto maior o numero de *threads* maior é o paralelismo disponível para dar suporte a complexidade computacional imposta pelo crescimento da área de busca.

Uma visão de que uma *thread* é responsável pela comparação do bloco atual em questão com um bloco *4x4* no quadro de referência também pode ser observada. Assim pode-se perceber que um bloco é composto de *pixels* (amostras) e que um conjunto de *4x4 pixels* forma um bloco de vídeo.

Os principais parâmetros utilizados no *kernel* em CUDA desenvolvido para o FS e sua respectiva chamada, pode ser visualizada na Figura 4.5.

Definição de Parâmetros

```
BLOCK_SIZE = (TAMBLOCO * AREABUSCA) – (TAMBLOCO -1)
TAMBLOCO = 4x4
Dimensão do Block (BLOCK_SIZE, BLOCK_SIZE);
Dimensão da grid (ceil(LARGURA/TAMBLOCO), ceil(ALTURA/TAMBLOCO));
```

Chamada do *Kernel*

```
full_search<<<grid, threads>>>(quadro_autal, quadro_referencia, TAMBLOCO, AREABUSCA, LARGURA, ALTURA, vetor_total);
```

Figura 4.5: Definição de parâmetros e chamada do *Kernel* referente ao FS.

Na Figura 4.5 é apresentada a configuração e a chamada do *kernel* referente à implementação do algoritmo FS em CUDA.

A dimensão dos *blocks* (conjunto de *threads* que é responsável pela execução de um bloco atual do vídeo) é bidimensional e se refere ao tamanho da área de busca em questão, como exposto anteriormente. O cálculo referente o tamanho do *block* (`BLOCK_SIZE`) leva em consideração: o tamanho do bloco de vídeo considerado (`TAMBLOCO`), a dimensão da área de busca em blocos (uma área de busca *12x12* com blocos de vídeo *4x4* resulta em uma dimensão de *3x3* blocos de vídeo) e por fim, uma garantia de que tamanho do *block* não ultrapasse o limite permitido de *4x4 pixels*, ou seja, garantindo que *threads* não executem dados inválidos e não escrevam em posições de memória inválidas.

Por sua vez, a dimensão da *grid* (quantidade de *blocks* de GPU presentes na execução do *kernel*), a qual é composta com a quantidade de blocos que formam a dimensão da sequencia de vídeo em questão, está relacionada com a resolução do vídeo em questão (CIF, HD720p, HD1080p, entre outros). A fim de garantir que números indevidos não sejam contabilizados, provenientes de divisões inválidas, foi utilizada a função denominada *Ceil (*estrutura de dado da linguagem de programação C/C++*)*, conforme a Figura 4.5, a qual arredonda valores inteiros para cima.

Além disso, a chamada do *kernel* também é descrita e pode-se observar que possui os seguintes parâmetros de entrada: quadro atual, quadro de referência, dimensões dos quadros, área de busca e tamanho do bloco. Por outro lado, como parâmetro de saída considera um vetor que possui os valores de SAD, bem como seus vetores de movimento.

Resumindo, em relação aos parâmetros definidos para execução deste algoritmo salientam-se dois aspectos: ***(i)*** dimensão da *grid*: a definição da dimensão da *grid* em relação ao número de blocos presentes na resolução em questão garante a quantidade de *blocks* suficientes para compreender toda a carga de execução; ***(ii)*** dimensão dos *blocks*: sabendo que a exaustividade apresentada pelo algoritmo FS está diretamente

relacionada como aumento do tamanho da área de busca, a definição da dimensão dos *blocks* de acordo com o tamanho da área de busca foi definida de modo a aumentar o paralelismo dos dados conforme o aumento direto do custo computacional.

Descrita a configuração e a ideia básica do *kernel*, abaixo é apresentado um pseudocódigo através do Algoritmo 8 que representa a implementação do FS em CUDA. Este pseudocódigo é baseado no fluxograma apresentado na Figura 4.2.

Algoritmo 8 : Full_Search_GPU ()

Parâmetros : *quadroAtual, quadroReferencia, TamBloco, AreaBusca, Altura, Largura*
Funcionalidade: Algoritmo *Full Search* para diferentes dimensões de área de busca e resoluções de vídeo em GPU
Retorno: Vetor com menores valores de SAD e suas respectivas posições para todo quadro atual

```
1.  [CUDA] posicao_inicial_bloco_atual = blockID * TamBloco ;                    //indexação das threads
2.  [CUDA] posicao_inicial_quadro_referencia = posicao_inicial_quadro_atual * AreaBusca/2 ;   // indexação das threads
3.
4.  Se (posicao_inicial_bloco_atutal está dentro do quadro && posicao_inicial_bloco_atual está dentro da area de busca )
5.   Se (posicao_inicial_quadro_referencia está dentro do quadro && posicao_inicial_quadro_referencia está dentro da area de busca )
6.      Para cada pixel do bloco:
7.          vetor_final ← calculo_SAD (quadroAtual, posicao_inicial_quadro_atual , quadroReferencia, posicao_inicial_quadro_referencia, TamBloco, Largura )
8.      fim Para
9.    fim Se
10. fim Se
11.
12. Para cada bloco atual
12.     vetor_resultante = encontra_menor_valor_sad(vetor_final);
13. fim for
14. retorno {vetor_resultante}
```

Observando o Algoritmo 8, primeiramente as *threads* que compõe um *block* recebem o índice do início de cada bloco atual (`blockID * Tambloco`), ou seja, recebem a posição inicial de onde devem iniciar a execução. Posteriormente, setada no início de cada bloco que será realizada a comparação entre os blocos, esta posição é mapeada para centro da área de busca no quadro de referência (`posição_bloco_atual * AreaBusca/2`).

Satisfeitos os limites de execução, largura e altura do quadro e dimensão da área de busca, o algoritmo FS é executado a partir do calculo de SAD (`calcula_SAD`) para cada posição inerente ao bloco atual, deslocando-se *pixel* a *pixel* .

Os valores de SAD são armazenados linearmente em um vetor, organizados na ordem de cada bloco atual para cada área de busca. Por fim, os menores valores de SAD e seus respectivos vetores de movimento são gerados.

Para finalizar esta descrição do FS em CUDA, na Figura 4.6 pode-se observar, de maneira resumida, o mapeamento de elementos de vídeo para elementos de GPU.

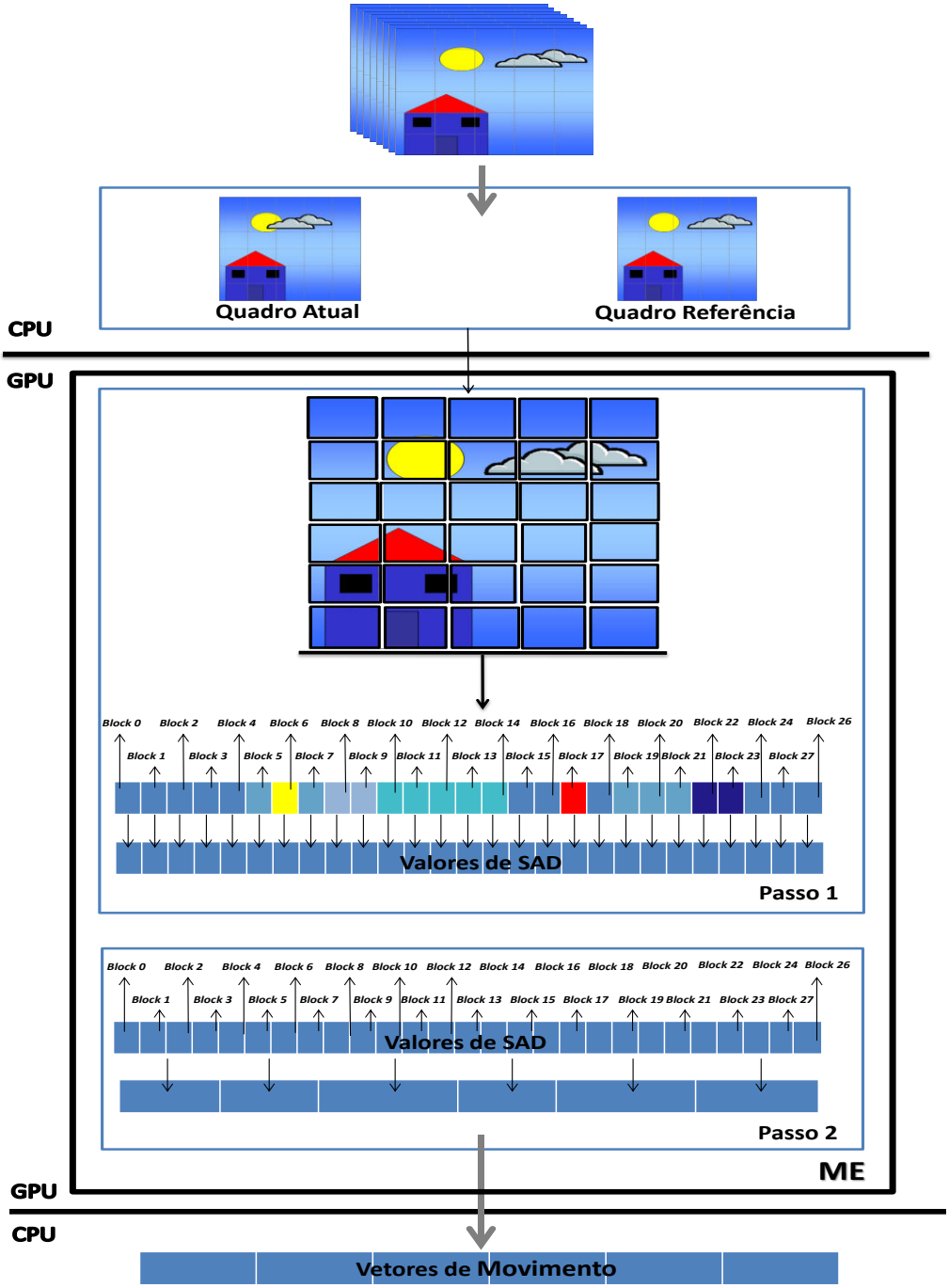


Figura 4.6: Algoritmo *Full Search* – Mapeamento de elementos de vídeo para elementos de GPU.

A divisão da sequência de vídeo é realizada de forma sequencial em CPU, de modo a obter um quadro atual e um respectivo quadro de referência de cada vez. As comunicações entre os dispositivos estão indicadas na Figura 4.6 a partir de linhas horizontais, as quais delimitam a execução dos dois dispositivos, CPU e GPU.

Na GPU é efetivamente executada a ME FS, onde são utilizados dois quadros um atual e um de referência. Estes quadros são tratados em blocos de *4x4 pixels*, assim como se observar na Figura 4.6.

Como já descrito, cada *block* é responsável por executar cada bloco atual do quadro. No entanto, a principal ideia introduzida na Figura 4.6 é ter uma ilustração do nível de paralelismo adotado, onde a base da paralelização consiste na execução da ME para cada bloco atual simultaneamente. A partir disso, pode-se analisar na Figura 4.6 que a execução de cada bloco atual (representado por cores que tentam fazer referência ao quadro ilustrado) está sendo relacionado a um *block* da placa de video. Esta execução resulta em diferentes valores de SAD para cada bloco atual, os quais são armazenados linearmente em um vetor.

No segundo passo, o objetivo é encontrar o menor valor de SAD para cada bloco atual no vetor resultado do primeiro passo. Esta busca é realizada a partir do uso da função "*reduce*" da biblioteca *Thrust*, assim, como produto resultante tem-se os valores de SAD e seus respectivos vetores de movimento, onde suas posições são geradas a partir do índice de cada *thread* o que o executou.

4.2.2 Algoritmo *Diamond Search* em GPU

O algoritmo DS é um algoritmo considerado sub-ótimo e comumente utilizado na literatura. Tipicamente, este algoritmo é composto por duas etapas: LDSP e SDSP, como já mencionado no Capítulo 3. O DS tem como produto final um vetor de movimento para cada bloco atual. Por sua vez, o algoritmo DS também foi paralelizado com base nos conceitos da hierarquia de processamento paralelo da arquitetura CUDA.

A Figura 4.7 apresenta um diagrama que ilustra o comportamento do DS implementado em CUDA para um bloco que compõe o quadro atual.

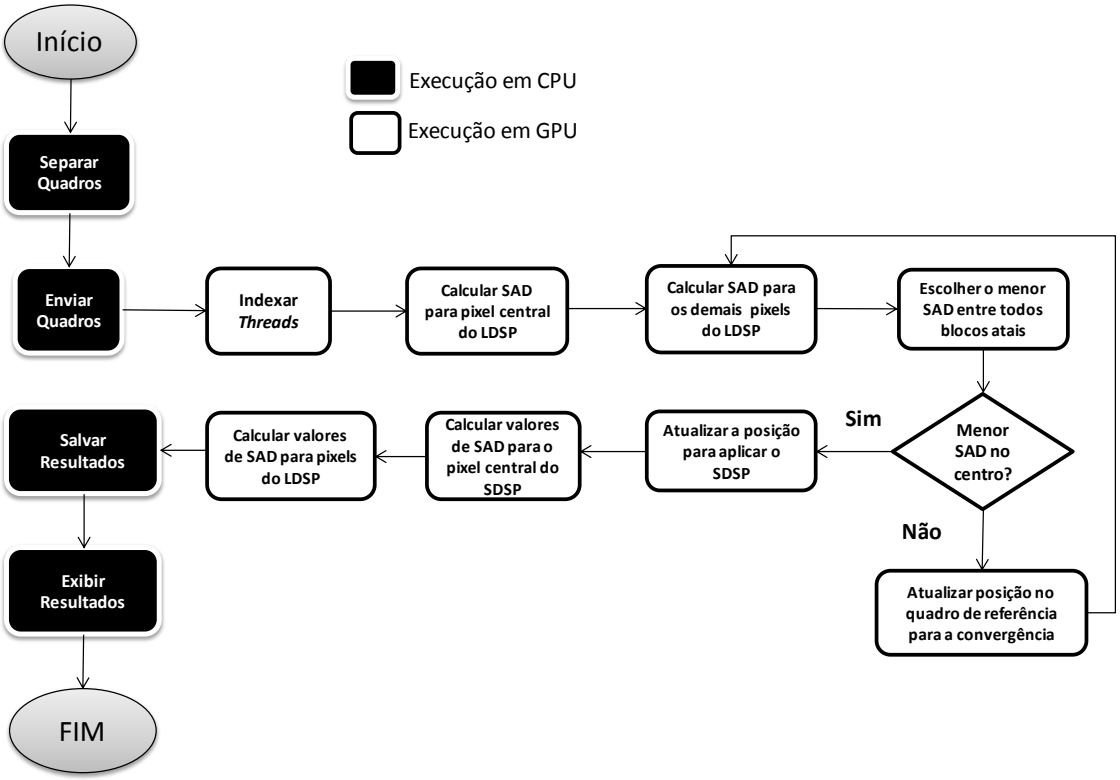


Figura 4.7: Fluxograma do Algoritmo *Diamond Search* em GPU.

Assim como no algoritmo FS em GPU descrito na Seção 4.2.1, inicialmente, os quadros são separados em CPU sequencialmente, de modo a obter um quadro atual e

seu respectivo quadro de referência por vez. Estes dados são enviados da memória RAM da CPU diretamente par memória global da placa de vídeo.

De posse destes dois quadros por vez, a GPU encontra-se apta a realizar a ME para o quadro atual enviado. Neste algoritmo cada *thread* é responsável por executar um bloco atual, diferente do algoritmo FS em CUDA onde um *block* inteiro era responsável pela execução de cada bloco atual. Assim, as *threads* são posicionadas no início de cada bloco atual para que a execução de um quadro atual possa ser realizada simultaneamente.

Obtidas as devidas posições no quadro atual, as mesmas são mapeadas para o quadro de referência, onde as buscas serão realizadas efetivamente. A partir deste ponto, o algoritmo DS é executado na íntegra na placa de vídeo. A execução do DS envolve duas etapas, como já mencionado, a etapa inicial LDSP, tendo como ponto de partida o primeiro pixel do bloco central do diamante. Logo a seguir, os demais *pixels* que fazem parte do LDSP são calculados, a fim de formar os nove blocos candidatos que compõe esta etapa.

Realizada a etapa que consiste no cálculo de SAD para os nove blocos que compõem a etapa LDSP, é feita a escolha pelo menor valor de SAD de cada um dos blocos candidatos. O algoritmo é iterado até que a condição de parada seja satisfeita (valor do menor SAD corresponder ao bloco central do diamante). A direção de convergência do algoritmo é dada de acordo com a posição do diamante em que o menor valor de SAD foi encontrado, podendo estar: acima, abaixo, à direita, à esquerda ou em alguma aresta do elemento central que está sendo avaliado.

Satisfeita esta condição (bloco que representa menor SAD do LDSP se localizar no elemento central do diamante), o refinamento a partir da aplicação da etapa SDSP é realizado a partir do cálculo de SAD para os cincos blocos que compõe o menor diamante. Quando da obtenção dos valores de SAD dos blocos candidatos do refinamento, a busca pelo menor valor é realizada.

Por fim, um vetor com o menor valor de SAD para cada bloco atual e seus respectivos vetores de movimento é enviado para CPU, como produto final do algoritmo proposto. O algoritmo *reduce* da biblioteca *Thrust* não foi utilizado na avaliação dos menores valores de SAD no DS GPGPU, assim como no algoritmo, pois este algoritmo é composto de duas etapas e necessita de duas verificações, uma para a etapa LDSP e outra para a etapa SDSP. Assim, seria necessário um número maior de comunicação entre CPU e GPU (mais duas comunicações, uma para obter o menor valor resultante do LDSP e outra para SDSP), pois os algoritmos que compõe a *Thrust* são invocados somente a partir da CPU, impossibilitando utiliza-los dentro do *kernel* criado.

Como se pode perceber na Figura 4.7, o algoritmo DS possui dependência de dados entre as etapas que o constituem, onde o refinamento (SDSP) é aplicado somente quando o elemento central do LDSP representar a melhor similaridade entre os blocos. A fim de minimizar esta dependência, na versão do DS em GPU proposta neste trabalho concentra-se na execução de blocos atuais simultaneamente, ou seja, uma *thread* é responsável pela execução da ME de um bloco atual

Com base nesta descrição, a paralelização do algoritmo DS em CUDA está baseada ilustrada na Figura 4.8.

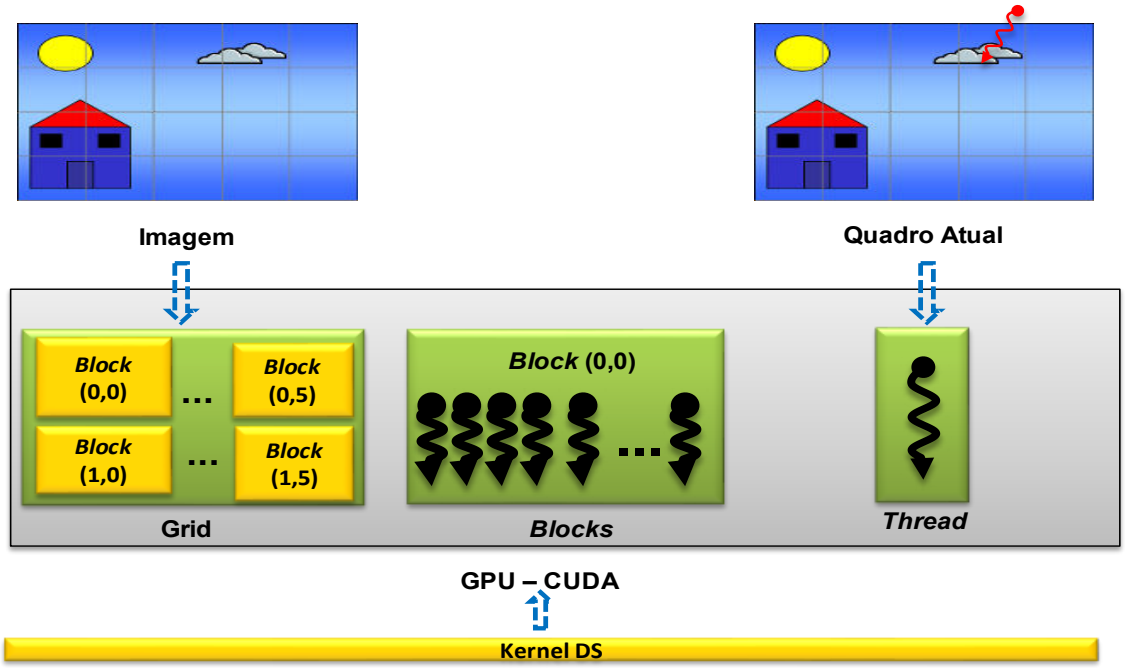


Figura 4.8: Modelo de Programação CUDA – Alocação Algoritmo *Diamond Search*.

A dimensão da *grid*, de acordo com a Figura 4.8, está relacionada de acordo com o tamanho da imagem ou quando atual. Este fato tem a mesma justificativa apresentada para o algoritmo FS em CUDA, onde uma *grid* deve estar apta a suportar o problema em questão, e neste caso, o problema é referente a execução da ME para um quadro atual por vez. Ou seja, o número de *blocks* deve estar de acordo com o número de blocos atuais.

A dimensão dos *blocks* não está relacionada a nenhum elemento específico do contexto de codificação de vídeo assim como os demais conceitos da hierarquia de processamento de CUDA, pois uma dimensão fixa foi adotada para todas as execuções deste algoritmo. A decisão referente à dimensão dos *blocks* utilizada na implementação deste algoritmo foi tomada a partir de algumas simulações e testes de modo a analisar o desempenho da aplicação utilizando dois parâmetros distintos: ***(i)*** dimensão máxima dos *blocks* permitida pela GPU; ***(ii)*** dimensão referente à quantidade de blocos em um quadro. Por exemplo, para um quadro de resolução CIF, composto de *6336* blocos *4x4*, seria necessário alocar blocos de GPU de dimensão *80x80,* o qual compreende *6400 threads*. Na Figura 4.9 podem ser observados os testes realizados considerando os dois parâmetros mencionados, tamanho de bloco máximo (azul) e tamanho de bloco de acordo com a resolução (verde) para diferentes áreas de busca (de *12x12* até *128x128 pixel*) e resoluções de vídeo: CIF (*352x288*) – A; HD720p (*1280x720*) - B e HD1080p – C (*1920x1080*).

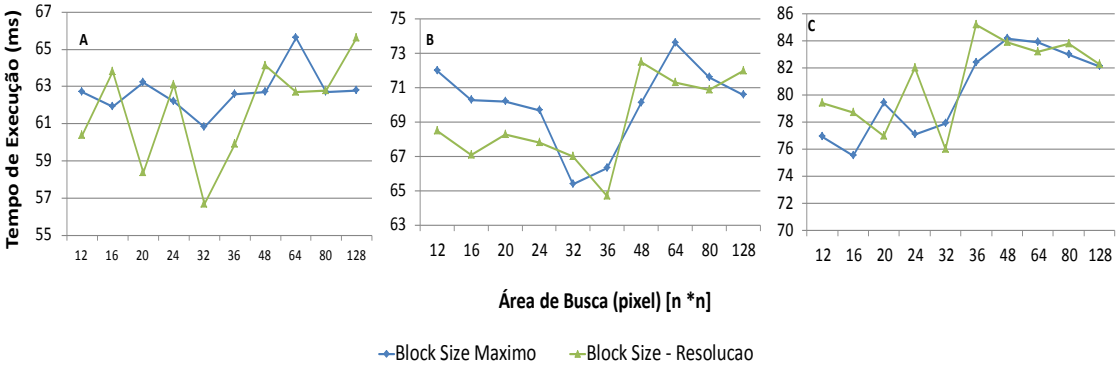


Figura 4.9: Comparativo tamanho de bloco de GPU – Máximo *x* Resolução – CIF (A), HD720p (B) e HD1080p (C).

Analisando os dados apresentados nos gráficos da Figura 4.9 pode-se perceber que não existe uma superioridade estável e visível apresentada por ambos os parâmetros considerados, principalmente em relação ao tamanho de *block* referente à resolução da sequência de vídeo (em verde), o qual para algumas áreas de busca apresenta uma irregularidade considerável em relação às demais dimensões. Desta forma, optou-se por utilizar o máximo de recursos disponibilizados pela GPU, isto é, alocando a dimensão máxima de *block* para este algoritmo.

Dando continuidade a ilustração da Figura 4.8, pode-se perceber o nível de paralelismo adotado nesta implementação, onde, em resumo, cada *thread* é responsável pela execução de um bloco atual. Diferente da implementação FS GPGPU onde diversas *threads* atuam na execução da ME em um bloco atual, visto que um *block* é responsável por um bloco atual no FS em CUDA.

Em resumo, os parâmetros relacionado à alocação de recursos da GPU para implementação e execução do DS são os seguintes: ***(i) thread***: cada *thread* que compõe o bloco de GPU é responsável pela execução de um bloco atual; ***(ii) block***: o tamanho do *block* adotado na implementação deste algoritmo se refere ao número máximo de *threads* que a placa gráfica utilizada permite alocar; ***(iii) grid***: o tamanho da *grid* está de acordo com o numero de blocos atuais que compõe um quadro atual, ou seja, com a resolução do vídeo de entrada.

Considerando a nomenclatura adotada em CUDA, pode-se observar a definição dos parâmetros e a chamada do *kernel* responsável por executar o algoritmo DS em CUDA na Figura 4.10.

Definição de Parâmetros
BLOCK_SIZE = NUM_MAXIMO_PERMITIDO; *TAMBLOCO = 4x4* ***Dimensão do Block*** *(BLOCK_SIZE, BLOCK_SIZE);* ***Dimensão da grid*** *(ceil(LARGURA/TAMBLOCO)/BLOCK_SIZE, ceil(ALTURA/TAMBLOCO)/BLOCK_SIZE);*
Chamada do *Kernel*
diamond_search<<<***grid***, ***threads***>>>(quadro_atual, quadro_referencia, TAMBLOCO, *AREABUSCA*, LARGURA, ALTURA, vetor_total);

Figura 4.10: Definição de parâmetros e chamada do *Kernel* referente ao DS.

Da mesma forma como apresentado no Algoritmo 8 (FS em GPU), na Figura 4.10 é apresentada a configuração do algoritmo DS e a chamada do *kernel* responsável pela sua execução.

A dimensão bidimensional dos *blocks* (número de *threads* presente em um *block*) se refere ao tamanho dos *blocks*, neste caso, ao tamanho máximo que a placa de vídeo utilizado permitir.

Por sua vez, a definição da dimensão da *grid* (número de *blocks* presente na execução do *kernel*) adotada na implementação do DS é dada de acordo com a quantidade de blocos na resolução adotada de modo a garantir que uma *thread* seja responsável pela execução da ME de um bloco atual. O cálculo desta dimensão leva em consideração quantidade de blocos que formam a dimensão da sequencia de vídeo obtida como parâmetro de entrada e o tamanho do *block* (*4x4* pixles) para garantir que cada *thread* irá executar apenas um bloco atual. Para se certificar que números indevidos não sejam contabilizados, provenientes de divisões inválidas, foi utilizada a função denominada *Ceil (*linguagem C*)*, conforme a Figura 4.10, a qual arredonda valores inteiros para cima.

A chamada do *kernel* também é descrita e pode-se observar que possui os seguintes parâmetros de entrada: quadro atual, quadro de referência, dimensões dos quadros, área de busca e tamanho do bloco. Por outro lado, como parâmetro de saída, considera um vetor que possui os valores de SAD, bem como seus respectivos vetores de movimento.

Descrita a configuração e a ideia básica do *kernel*, para uma melhor compreensão da implementação proposta, é apresentado um pseudocódigo através do Algoritmo 9 que representa a implementação do DS em CUDA. Este pseudocódigo é baseado no fluxograma apresentado na Figura 4.7.

Algoritmo 9 : Diamond_Search_GPU()

Parâmetros : *quadroAtual, quadroReferencia, TamBloco, AreaBusca, Altura, Largura,*
Funcionalidade: Algoritmo *DiamondSearch* para diferentes dimensões de áreas de busca e resoluções de vídeo em GPU
Retorno: *menorSad*

```
1. [CUDA] posicao_inicial_bloco_atual = blockID * blockDIM * TamBloco + threadID * TamBloco ; //indexação das threads
2. [CUDA] posicao_inicial_quadro_referencia = posicao_inicial_quadro_atual; //indexação das threads
2. vetor_final[central] ← calcula_SAD_bloco_central_LDSP (quadroAtual, quadroReferencia); //1º pixel do bloco central do diamante
3. for para cada posição do bloco :
4.    vetor_final ←calcula_SAD_demais_pixels_LDSP (quadroAtual, quadroReferencia);
5. fim for
6. menorSad ←encontra_menor_SAD(vetor_final);
7. Enquanto não_convergiu
8.       Atualiza próxima posição do LDSP no quadro de referência
9.       Para cada posição do bloco
10.            vetor_final ←calcula_SAD_demais_posicoes_LDSP (quadroAtual, quadroReferencia);
11.      fim Para
12.      menorSad ← encontra_menor_SAD(vetor_final);
13.      Se o menorSad refere ao bloco central do diamante
14.            convergiu                                                    // calcula LDSP
15.      fim Se
16. fim Enquanto
17.    vetor_final ←calcula_SAD_bloco_central_SDSP (quadroAtual, quadroReferencia);
18.    Para cada posição do bloco :
19.       vetor_final←calcula_SAD_demais_pixels_SDSP (quadroAtual, quadroReferencia);
20.    fim Para
21. menorSad = encontra_menor_SAD(vetor_final);
22. Retorno {menorSAD}
```

De acordo com o Algoritmo 9, primeiramente as *threads* são inicializadas no início de cada bloco atual. Em particular, na indexação que se refere à posição de início do bloco atual tem-se a divisão do quadro atual em *blocks,* de acordo com a dimensão do bloco de *4x4* amostras (`Block_ID * BlockDIM * TamBloco + ThreadID * TamBloco`), em outras palavras, este cálculo situa em que *block* a *thread* está localizada (de acordo com o seu identificador e sua dimensão) e que ela deve executar somente amostra de *4x4 pixels* (tamanho do bloco) por vez.

Posteriormente, esta posição é mapeada para centro da área de busca no quadro de referência, de acordo com o algoritmo DS ilustrado. Assim, a execução efetiva do algoritmo DS pode ser iniciada a partir da aplicação da primeira etapa LDSP quando da execução do elemento central do diamante. Dando continuidade, os valores de SAD de todos os blocos vizinhos do elemento central do LDSP são calculados e armazenados em um vetor. Assim, enquanto a condição principal do algoritmo, onde o menor SAD deve corresponder ao bloco central do LDSP, o algoritmo iterar. As posições são atualizadas para que a nova iteração seja realizada. As iterações do algoritmo DS quando realizada a etapa LDSP variam de acordo com a localização do menor valor de SAD, podendo estar à direita, à esquerda, abaixo, acima ou em alguma das arestas. Caso a condição seja satisfeita, o refinamento do algoritmo é aplicado a partir da execução da etapa SDSP, onde mais cinco blocos candidatos são avaliados e o menor SAD é armazenado. A posição central do diamante foi avaliada individualmente das demais posições de modo a facilitar a condição de parada e iteração do algoritmo.

Por fim, como retorno do algoritmo tem-se um vetor de valores de SAD e seus respectivos vetores de movimento, como já mencionado e descrito.

Para resumir e finalizar a descrição do algoritmo DS em CUDA, a Figura 4.11 é apresentada de modo a deixar claro, mais uma vez, o paralelismo em nível de blocos adotado nesta implementação.

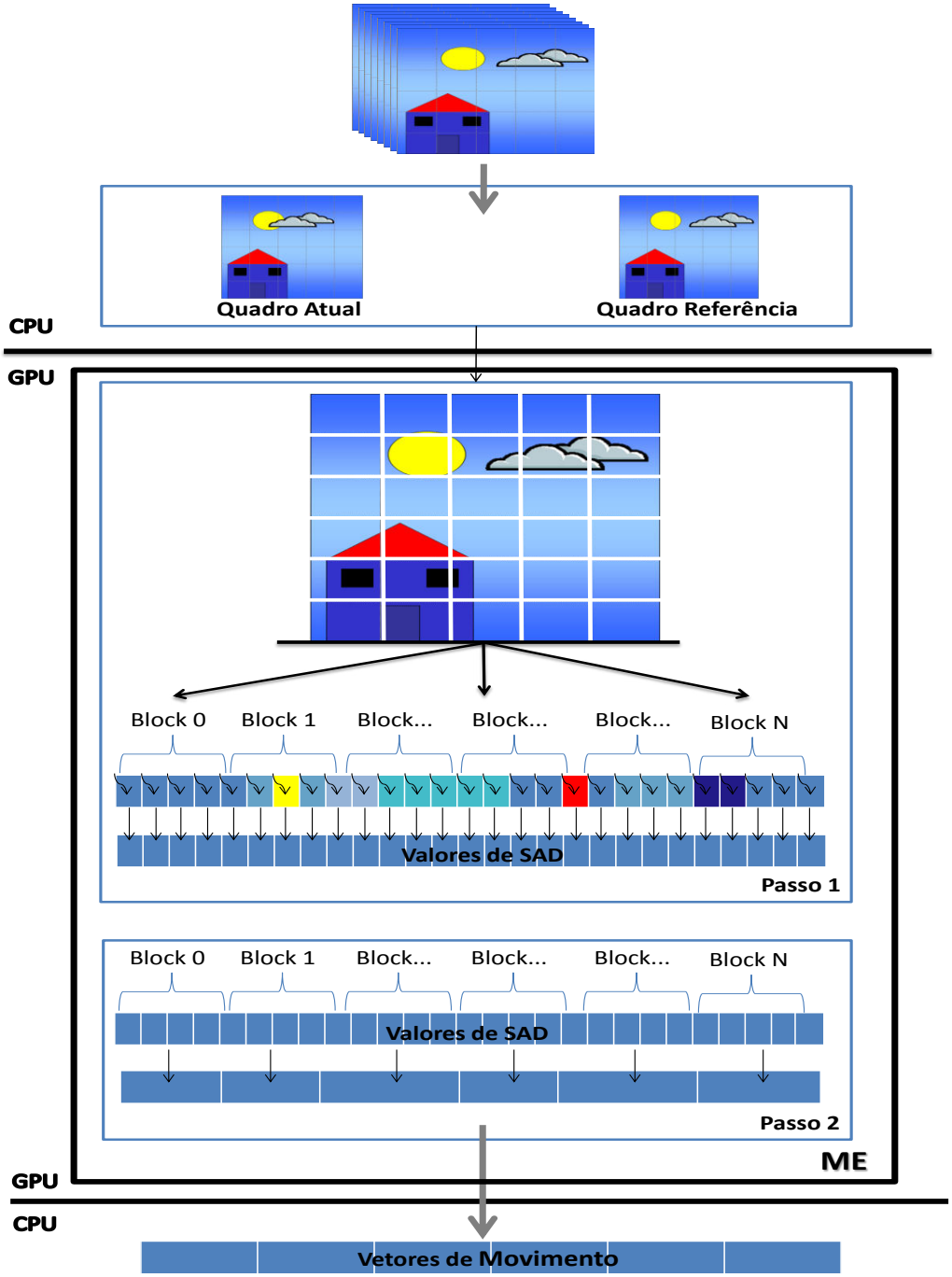


Figura 4.11: Algoritmo *Diamond Search - kernel*.

Primeiramente, tem-se uma visão geral da divisão da sequência do vídeo obtida como parâmetro de entrada em CPU de forma sequencial. Uma vez divido o vídeo de entrada, tem-se dois quadros: atual e referência. Estes elementos são enviados para GPU, onde a comunicação entre os dispositivos CPU e GPU estão devidamente ilustradas na Figura 4.11 a partir duas linhas horizontais que delimitam o que é executado em CPU e o que é processado na placa gráfica.

De posse dos dois quadros para realização da ME, o quadro atual é tratado na GPU em blocos divididos por *4x4* amostras, assim como é meramente ilustrado na Figura 4.11, ainda no passo 1.

A principal característica do *kernel* responsável pela execução do DS em CUDA (na Figura 4.11, ME) é a atribuição de cada *thread* a cada bloco atual. Esta atribuição também é representada na Figura 4.11, atribuindo cada bloco atual e sua respectiva cor que o representa para cada thread que forma um (ou mais) *block*. Além disso, visto que um *block* é formado por um conjunto de *threads*, pode-se perceber que um *block* comporta a execução de mais de um bloco atual, veja passo 1 da Figura 4.11. Para cada bloco atual a ME é executada e os resultados são armazenados linearmente em um vetor.

Terminados os cálculos referentes aos valores de SAD para todos os blocos atuais, são procurados os menores valores e seus respectivos vetores de movimento e armazenados em um vetor resultante o qual é enviado para memória RAM da CPU.

A área de busca adotada neste algoritmo considerou o número de iterações máximo para o encontro do melhor casamento, assim, tem-se uma limitação do algoritmo, evitando que o mesmo obtenha a busca pelos menores valores de SAD em uma só direção. Considerando que a dimensão máxima da área de busca tratada nos experimentos que dizem respeito a este algoritmo é de *240x240 pixels*, pode-se dizer que o número máximo de convergência deste algoritmo é de 20 vezes.

4.3 Considerações Finais sobre o Capítulo

Este capítulo descreveu a metodologia e a implementação utilizada no desenvolvimento deste trabalho. Esta metodologia, em resumo, baseou-se na descrição dos algoritmos sequenciais FS e DS, os quais foram utilizados no desenvolvimento das respectivas implementações paralelas.

Versões paralelas (GPU - CUDA, *multi-core* – OpenMP) e distribuídas (MPI) foram desenvolvidas referente aos algoritmos de busca *Full Search* e *Diamond Search.* Em especial, a paralelização e execução da Estimação de Movimento utilizando GPU como alternativa de co-processamento. As demais implementações em diferentes paradigmas computacionais foram utilizadas como base comparativa das versões GPGPU apresentadas.

Considerando as versões GPGPU da ME, no algoritmo FS a dependência de dados é inexistente e o nível de paralelismo utilizado é em nível de blocos, onde um *block* é responsável pela execução de um bloco atual. Por outro lado, no algoritmo DS a dependência de dados foi tratada a partir do paralelismo em nível de blocos, ou seja, cada bloco atual é executado sequencialmente por uma *thread*.

Para todas as versões da ME descritas (CUDA, OpenMP, MPI e Sequencial), o tempo de execução considera as principais operações que envolvem o algoritmo (em resumo, cálculo e comparação dos valores de SAD), onde o carregamento da sequência de vídeo e a respectiva separação dos quadros (atual e referência) não são contabilizados. Assim, salienta-se que as soluções propostas foram implementadas de forma coerente entre elas, possibilitando comparações justas. O próximo capítulo apresenta alguns trabalhos relacionados encontrados na literatura visando futuras comparações.

5 REVISÃO DE TRABALHOS RELACIONADOS DA LITERATURA

Diferentes soluções de codificação de vídeo em software tem sido desenvolvidas desde a padronização dos codificadores de vídeo. O padrão H.264/AVC, considerado o estado-da-arte em codificação de vídeo, possui múltiplas soluções disponíveis na literatura, como por exemplo, os softwares de referências JM e o *x264*. No entanto, estes softwares não tem suporte para aceleração em GPU, ou quando possuem, tratam apenas de bibliotecas proprietárias, como o caso do *x264*.

Neste cenário, alguns trabalhos visam acelerar a codificação de vídeo em placas gráficas e podem ser encontrados na literatura. Os trabalhos propostos em (LIN, 2006), (LEE, 2007), (CHEN, 2008), (KUNG, 2008), (SCHWALB, 2009), (HUANG, 2009), (YANG, 2009), (COLIC, 2010), (CHENG, 2010), (CHEUNG, 2010) serão descritos neste capítulo e tem como foco principal a implementação de algoritmos da ME em GPU. Salienta-se que não foram encontrados na literatura muitos trabalhos que abordam o paralelismo de algoritmos rápidos em GPU, em especial o algoritmo *Diamond Search*, por esta razão a grande maioria dos trabalhos que serão descritos abaixo apresentam soluções voltadas para o algoritmo FS.

5.1 Trabalho de Chen et al.

O trabalho de *Chen et. al.* (CHEN, 2008) apresenta uma Estimação de Movimento baseada no algoritmo FS em GPU utilizando a arquitetura CUDA. Este trabalho divide o algoritmo proposto para ME em diferentes etapas para atingir um elevado grau de paralelismo considerando um paralelismo em nível de blocos: ***(i)*** cálculo dos valores de SAD para blocos de dimensão *4x4 pixels*; ***(ii)*** cálculo dos valores de SAD para blocos de dimensões variáveis; ***(iii)*** comparações dos valores de SAD considerando precisão inteira; comparações de valores de SAD considerando precisão fracionária; ***(iv)*** refinamento da ME baseada em precisão fracionária.

Este trabalho considera tamanho de bloco variável (*16x8*, *8x16*, *8x8 – 8x4*, *4x8*, *4x4 pixels*) atingindo um alto número de blocos candidatos para serem avaliados. Por esta razão, o desempenho da solução proposta está diretamente relacionado com a tomada de decisão em relação aos diferentes tamanhos de blocos suportados pela aplicação. Assim, pode-se dizer que o trabalho não está focado em atingir uma codificação de vídeo em tempo real, mas em prover visando a qualidade de vídeo a partir de uma ME robusta a qual dá suporte a uma ampla gama de características. Além disso, a versão da ME baseada na precisão fracionária necessita de um refinamento o qual também tem influência na eficiência desta aplicação.

5.2 Trabalho de Lin et. al.

O trabalho de *Lin et. al.* (LIN, 2006) apresenta uma solução paralela para Estimação de Movimento em nível de *pixel* utilizando GPU. Este trabalho não considera o uso da arquitetura CUDA, pois quando da sua publicação a API ainda não tinha sido proposta pela NVIDIA. Desta forma, este trabalho faz o uso da memória de texturas como armazenamento e gerenciamento dos dados e considera precisões inteiras e fracionárias.

Em resumo, o algoritmo proposto é baseada em uma técnica com múltiplos passos considerando o algoritmo FS. Estes passos que implementam a ME em GPU são divididos em quatro principais laços os quais são compostos por duas etapas: ***(i)*** cálculo de valores de SAD para precisão inteira e comparações entre eles; ***(ii)*** comparação entre os valores de SAD entre os mínimos locais a fim de encontrar o mínimo global e cálculo de valores de SAD para precisão fracionária e comparações entre eles.

Considerando os diferentes laços e etapas introduzidos no algoritmo proposto, pode-se observar que não há um equilíbrio referente à divisão das etapas as quais realizam tarefas custosas de cálculo e comparação de valores de SAD em um mesmo passo.

5.3 Trabalho de Lee et. al.

No trabalho proposto por *Lee et. al.* (LEE, 2007) são apresentadas três diferentes soluções para a Estimação de Movimento em GPU considerando o algoritmo FS. Este trabalho utiliza o uso de memória de textura da GPU e o paralelismo apresentado está baseado em nível de *pixel*. As três diferentes alternativas são: ME baseada em precisão inteira, ME com precisão fracionária e ME com precisão inteira considerando três quadros de referência simultaneamente. Este algoritmo é proposto a partir de quatro principais laços que são constituídos de duas etapas: ***(i)*** geração de valores de SAD mínimos locais; ***(ii)*** geração de valores de SAD mínimos globais.

Os melhores resultados alcançados neste trabalho em relação às três soluções propostas se referem à ME com precisão inteira considerando três quadros de referência simultâneos. A solução com precisão fracionária permite um maior refinamento do algoritmo de busca (aumento da qualidade), porém é ineficiente em termos de desempenho em GPU devido à dependência dos dados.

5.4 Trabalho de Cheng et. al.

O trabalho de *Cheng et. al.* (CHENG, 2010) apresenta alternativas baseadas em nível de bloco para paralelização da ME em GPU. Estas soluções implementam distintos algoritmos de busca para ME em CUDA, abordando tanto algoritmos exaustivos, como algoritmos rápidos: *Full Search* (FS), *Three Step Search* (TSS), *Four Step Search* (FSS) e *Diamond Search* (DS).

Estas alternativas exploram todos (ou quase todos) os níveis de memória (Global, Textura, Constante, Compartilhada, Local e Registradores) inerente à arquitetura de uma GPU, em especial à memória de textura responsável pela execução dos algoritmos propriamente dita. A memória compartilhada destina-se para o armazenamento dos vetores de movimento resultantes e por sua vez, na memória global é realizado o acesso coalescido[7] dos dados. Considerando que instruções de controle são impactantes em termo de desempenho em um algoritmo paralelo, no trabalho proposto se fez necessário

[7] Acesso dos dados na memória global da GPU de modo a evitar ociosidade de recursos.

um controle de fluxo dos dados de modo a garantir que as *threads* de uma mesma *warp* não obtenham caminhos divergentes, pois quanto maior é o numero de divergências, maior é o número de instruções a serem executadas pela *warp*.

Além da aceleração da ME proposta por (CHENG, 2010), a solução proposta também abrange outras características da codificação de vídeo, bem como, a etapa posterior à ME no fluxo de codificação de vídeo. Esta etapa está relacionadas a codificação dos vetores de movimento, onde a realização desta tarefa consiste em uma mediana entre três vetores de movimento de blocos vizinhos.

5.5 Trabalho de Schwalb et. al.

O trabalho de *Schwalb et. al.* (SCHWALB, 2009) propõe uma versão rápida da Estimação de Movimento visando a exploração máxima do paralelismo inerente aos dispositivos gráficos em conjunto com o menor comunicação entre CPU e GPU possível. Esta solução concentra-se na paralelização da ME inerente ao software de referência JM 9.0 (software de referência do padrão de codificação de vídeo H.264/AVC mencionado na introdução) setado para o perfil *Baseline* do codificador. A alternativa proposta neste trabalho utilizou o programa *Shader Model* (RANDIMA, 2003), (CG, 2012), o qual é voltado para programação de códigos *assembly* e, concentra-se no algoritmo DS em GPU no menor de busca, *Small Diamond* - SDSP.

A entrada da solução proposta consiste no quadro atual e todos os quadros de referência que dizem respeito ao quadro que será codificado. Como saída, tem-se vetores de movimento e valores de SAD correspondentes. Na ME introduzida neste trabalho são considerados múltiplos quadros de referência (cinco quadros) e o refinamento final do algoritmo proposto, em resumo, é dado de acordo com a especificação do *Diamond Search* - SDSP, onde quatro posições de blocos candidatos são avaliadas ao redor do bloco central. Além disso, o número máximo de iteração atingido no trabalho de (SCHWALB, 2009) é igual a sete, enquanto o trabalho proposto nesta dissertação é de cento e vinte iterações por execução.

A implementação proposta por (SCHWALB, 2009) visa a avaliação de qualidade dos vídeos (não considerado no escopo do trabalho proposto) além da aceleração. Desta forma, os resultados apresentados pelo autor não abordam o desempenho da aplicação e, por esta razão este trabalho não pode ser utilizado como base comparativa de resultados.

5.6 Trabalho de Cheung et. al.

O trabalho de *Cheung et. al.* (CHEUNG, 2010) apresenta diferentes versões da ME, tanto para codificação, quanto para decodificação de vídeo em GPU. Algoritmos de busca exaustivo e rápidos são considerados e a precisão dos dados é tanto inteira, como fracionária.

O algoritmo exaustivo FS considera o cálculo de SAD como critério de similaridade. Por outro lado, o algoritmo rápido está baseado no algoritmo *smpUMHexagonS* (*simplified unsymmetrical multihexagon search*) (YI, 2005b). A ME FS proposta neste trabalho é baseada nos cálculos e comparações dos valores de SAD em um laço que percorre os macroblocos do quadro atual bem como a área de busca correspondente. A justificativa dos autores para o desenvolvimento da ME rápida baseada no *smpUMHexagonS* se deve ao fato do algoritmo viabilizar um equilíbrio entre a complexidade computacional introduzida e a eficiência em codificação apresentada.

Além disso, salienta-se que o trabalho proposto nesta dissertação utiliza tamanho de blocos de vídeo de dimensões *4x4 pixels*, assim, comparações mais precisas podem ser realizadas quando comparado com o trabalho de *Cheung et. al.* que trata de macrobloco (*16x16 pixels*) na codificação dos dados.

5.7 Trabalho de Yang et. al.

O trabalho de *Yang et. al.* (YANG, 2009) propõe uma ME especialmente voltada para vídeos HD1080p considerando o uso de GPU. A versão da ME introduzida neste trabalho propõe uma otimização no uso da memória compartilhada inerente aos processadores gráficos. Além disso, um algoritmo rápido para execução da ME visando codificação em tempo real considerando vídeos de alta resolução é apresentado com base no algoritmo exaustivo FS.

A ME proposta neste trabalho considera o cálculo de SAD como critério de similaridade e a arquitetura CUDA como API de programação. A precisão, em nível de blocos, adotada neste trabalho trata de um macrobloco (blocos de *16x16 pixels*) e cada *thread* alocadas na GPU é responsável por calcular um vetor de movimento.

O algoritmo proposto tem as seguintes características: ***(i)*** primeiramente todas as *threads* trabalham em conjunto de modo a carregar a área de busca e o macrobloco atual na memória compartilhada da GPU; ***(ii)*** logo após, cada *thread* executa um macrobloco e armazena o melhor resultado na memória compartilhada; ***(iii)*** por fim, o é realizada a busca pelo melhor vetor de movimento de cada macrobloco atual e é armazenado na memória global.

O algoritmo apresentado por *Yuang et. al.* foi customizado a partir do algoritmo FS considerando vídeos de resolução HD1080p. Segundo o autor, o custo computacional do FS em conjunto com a resolução do vídeo abordada inviabiliza totalmente o uso do algoritmo exaustivo, na íntegra, neste contexto. Assim, a ME proposta consiste em dois estágios: primeiramente é aplicado o algoritmo FS em uma região subamostrada (a quatro amostras uma é calculada) nos *pixels* do macrobloco (considerando área de busca de *64x32*), assim nem todas as amostras são calculadas. E por fim, no bloco em que o menor valor de SAD é encontrado é feito um refinamento na região de modo a extrair um vetor de movimento mais preciso, considerando macrobloco de *16x16* e área de busca *32x32*.

O trabalho de (YANG, 2009) também visa avaliação de qualidade e não apresenta dados relacionados ao desempenho da aplicação proposta. Os dados que puderam ser comparados com o trabalho proposto nesta dissertação se referem à taxa de processamento atingida pelo autor na execução de sua aplicação.

5.8 Trabalho de Kung. et. al.

O trabalho de *Kung et. al.* (KUNG, 2008) propõe uma ME em GPU de acordo com o padrão de codificação de vídeo H.264/AVC e considerando blocos de dimensão *4x4 pixels*.

Em resumo, como parâmetro de entrada é considerado um bloco atual (*4x4 pixels*) e o vetor de movimento predito por CPU. A memória de textura é utilizada para representar e armazenar os dados, como por exemplo, o quadro de referência.

Em relação aos algoritmos de busca, é utilizado o algoritmo exaustivo FS considerando 16 e 32 *pixels*. Além disso, algoritmos de buscas são utilizados

considerando diferentes etapas: TSS (*Three Step Search*), SSS (*Six Step Search*) e FSS (*Five Step Search*). No entanto, segundo os autores, a utilização de um algoritmo rápido composto de menos etapas que as mencionadas, não é eficiente na implementação em GPU, visto que um dos requisitos básicos para o uso desta tecnologia é a inexistência (ou poucos) de dados dependentes.

Este trabalho também implementa a etapa posterior à ME, a qual considera a codificação de vetores de movimento. Da mesma forma como mencionado anteriormente, é realizada uma mediana de três vetores de movimento de blocos vizinhos. Esta mediana, no entanto, gera uma dependência entre os dados que afeta no desempenho da aplicação.

5.9 Trabalho de Huang et. al.

O trabalho de *Huang et. al.* (HUANG, 2009) está baseado na codificação de vídeo escalável (SVC, do inglês *Scalable Video Coding*) com extensão do padrão H.264/AVC. No SVC as características de codificação de vídeo são exploradas em diferentes camadas. Desta forma, este trabalho apresenta uma paralelização para modelo de computação escalável SVC usando GPU, onde a arquitetura CUDA é utilizada neste contexto.

A metodologia proposta para este fim considera a divisão do padrão H.264/AVC em quatro partes: Transformadas e Quantização, Codificação de Entropia, Decodificação e Predições. As primeiras partes mencionadas (Transformadas e Quantização, Codificação de Entropia e a parte referente à decodificação) são executas em CPU utilizando processadores *multi-core.* A versão paralela do modelo de computação escalável SVC em GPU proposta neste trabalho foi desenvolvida para paralelizar a parte de predição intra e inter-quadros, em especial, algoritmos de modo de decisão.

Visto que a ME em CUDA proposta utiliza a memória compartilhada da GPU e está baseada no padrão de codificação de vídeo H.264/AVC, suporta blocos de diferentes dimensões: *8x8*, *4x8*, *8x4* e *4x4 pixels*. Além disso, tanto os algoritmos rápidos, como o algoritmo exaustivo FS são considerados. A dependência de dados inerente aos algoritmos rápidos é resolvida (ou parcialmente) a partir da utilização de vetores de movimento preditos anteriormente.

Em resumo, o algoritmo paralelo apresentado neste trabalho, primeiramente, calcula os valores de SAD de todas os blocos *4x4 pixels*, logo após as partições *4x4* são juntadas de modo a formar as outras dimensões para a escolha do melhor bloco candidato. Por fim, é aplicado o refinamento em paralelo, quando se consideram *pixels* fracionários.

Os resultados apresentados por (HUANG, 2009) consideram todo o codificador de vídeo é considerando e o uso da versão escalável do padrão H.64/AVC, características pelas quais não são levadas em consideração no escopo do trabalho proposto, o qual visa desempenho da ME apenas. Por esta razão, os resultados deste autor não puderem ser comparados com os resultados obtidos pelo trabalho apresentado nesta dissertação.

5.10 Trabalho de Colic et. al.

O trabalho proposto por *Colic et. al.* (COLIC, 2010) também se refere à exploração de GPUs no contexto de codificação de vídeo, em especial, considerando a arquitetura CUDA. A ME considera macroblocos (*16x16 pixels*) e é paralelizada em duas partes: ***(i)***

cálculo dos valores de SAD e ***(ii)*** busca pelo menor valor afim de gerar os vetores de movimento. Para cada uma destas partes foi utilizado um *kernel* diferente, ou seja, um para cálculo e outro para busca de valores.

Diferentes alocações de recursos quando da utilização da arquitetura CUDA foram considerados: ***(i)*** cada *thread* alocada é responsável por calcular o valores de SAD entre dois *pixels* (um do bloco atual e outro do bloco de referência); ***(ii)*** cada *thread* é alocada para calcular valores de SAD para um bloco candidato na área de busca; ***(iii)*** cada *thread* é responsável por calcular os valores de SAD para um bloco candidato e usar múltiplas threads para cada macrobloco; ***(iv)*** uma abordagem mista, atribuindo dinamicamente mais threads para calcular uma área de busca. Nos resultados apresentados nestes trabalhos foram consideradas as alternativas ***(iii)*** e ***(iv)*** referente às alocações de recursos. Além disso, foi considerada a memória compartilhada em diferentes GPUs.

Um dos principais objetivos do autor é mostrar que o uso de placas gráficas é interessante no contexto de codificação de vídeo, em especial para o módulo mais custo de um codificador, a ME. Desta forma, segundo o autor, somente a apresentação dos resultados obtidos, em termos de desempenho, na execução da primeira etapa do algoritmo proposto (cálculo dos valores de SAD) é suficiente para ilustrar e justificar este fato.

5.11 Considerações Finais sobre o Capítulo

Este capítulo apresentou uma revisão do estado da arte referente à paralelização da Estimação de Movimento em GPU. Todos os trabalhos consideraram o cálculo de SAD como critério de similaridade. A maioria dos trabalhos abrange o algoritmo de busca exaustivo FS pela inexistência de dependência de dados, o que é fundamental para computação paralela. Alguns trabalhos também apresentaram alguns algoritmos rápidos (*Diamond Search*, entre outros) com o objetivo principal de prover uma codificação de vídeo em tempo real. No entanto, outros trabalhos concentram-se em suportar diferentes configurações de codificação de vídeo e qualidade e não ilustram resultados relacionados a desempenho, foco deste trabalho. Além disso, a maioria dos resultados apresentados nos trabalhos trata de sequências de vídeo de baixa resolução, dificultando uma maior comparação de resultados.

Descritos alguns trabalhos presentes na literatura e a proposta desta dissertação, o próximo capítulo apresenta os experimentos realizados na avalição do uso de GPU na Estimação de Movimento, o qual visa a codificação de vídeo em tempo real considerando vídeos de alta definição e comparações com diferentes paradigmas computacionais.

6 REVISÃO DOS RESULTADOS

O capítulo 4 descreveu a principal proposta deste trabalho, a qual concentra-se na aceleração da Estimação de Movimento utilizando placa proprietária com *chip* de processamento gráfico como alternativa de co-processamento. Além disso, o capítulo anterior descreveu alguns artigos existentes na literatura que também atuam neste cenário. A fim de validar essa proposta e comparar com os trabalhos descritos, este capítulo apresenta os experimentos usados na avaliação deste trabalho. Esta avaliação consiste em verificar se o uso de GPUs é uma boa alternativa na codificação de vídeo, bem como, se a paralelização da ME apresentada é eficaz. Além disso, uma análise do desempenho das soluções propostas considerando restrições de processamento em tempo real (idealmente 30 quadros por segundo) resolução é também apresentada. Os experimentos consideram diferentes plataformas, tanto agregados de arquiteturas *multi-core* (CUDA e OpenMP) que proporcionam o paralelismo de dois níveis (processos e *threads*), como arquiteturas distribuídas (MPI).

As discussões presentes neste capítulo abrangem ambos os algoritmos desenvolvidos, *Full Search* e *Diamond Search,* os quais foram descritos no capítulo 2. O critério de similaridade considerado em todos os experimentos foi o cálculo de SAD e a dimensão dos blocos de vídeo adotada foi de *4x4 pixels*.

6.1 Plataformas de Execução dos Experimentos

Os experimentos realizados neste trabalho foram possíveis por intermédio do Grupo de Processamento Paralelo de Distribuído (GPPD) (GPPD UFRGS, 2012) do Instituto de Informática (II) da Universidade Federal do Rio Grande do Sul (UFRGS). Em especial, o ambiente distribuído utilizado considera a plataforma francesa Grid'5000 (GRID5000, 2012) através de um convênio firmado entre o GPPD e o *Institut National de Recherche em Informatique et em Automatique* (INRIA). As plataformas consideradas são as seguintes:

- *Multi-core*: CPU Intel® Core i7 930 2.8GHz (4 núcleos/cores e suporte de 8 threads)
- GPU: NVIDIA GTX 480 1.4GHz conectado via PCI-Express (480 unidades funcionais)
- Distribuído: *cluster Xiru* (XIRU CLUSTER, 2012), modelo Dell PowerEdge 1950, CPU Intel® Xeon E5310 1.6GHz, 2 CPUs por nodo, 4 cores por CPU, total de 28 CPUs e 112 cores, 16GB de memória, rede Gigabit Ethernet.

6.2 Descrição de Experimentos

Esta seção mostra uma descrição dos experimentos realizados na avaliação da Estimação de Movimento em GPU. Os algoritmos de busca utilizados foram descritos no Capítulo 3: *Full Search* e *Diamond Search*. Os parâmetros utilizados foram:

- APIs: CUDA, OpenMP, MPI.
- Resoluções de Vídeo: CIF (*352x288*), HD720p (*1280x720*) e HD1080p (*1920x1080*).
- Dimensões de Área de Busca:
  - FS*: 12x12*, *16x16*, *20x20*, *24x24*, *32x32*, *36x36*, *48x48*, *64x64*, *80x80.*
  - DS*: 12x12*, *16x16*, *20x20*, *24x24*, *32x32*, *36x36*, *48x48*, *64x64*, *80x80*, *128x128*, *240x240*.
- Tamanho de Bloco: *4x4*

Os testes realizados quando da avaliação da Estimação de Movimento em GPU proposta neste trabalho trata de apresentar a versão da ME em CUDA em comparação com as respectivas ME em OpenMP e MPI. Em MPI são considerados diferentes números de nodos alocados e, por sua vez, em OpenMP é considerado o número máximo de *threads* permitido pelo processador utilizado, neste caso 6 *threads*. O tamanho da área de busca é variável de *12x12* até *240x40 pixels* para todas as resoluções consideradas: CIF, HD720p e HD1080p.

Resoluções de baixa definição não foram utilizadas nas simulações apresentadas neste trabalho. A explicação é que vídeos de pequenas dimensões já não encontram muitas aplicações no mercado atual, visto que até mesmo dispositivos portáteis já dão suporte a resoluções mais elevadas, como por exemplo, HD720p.

A resolução CIF considerada neste trabalho refere-se à sequência de vídeo denominada *Foreman*. Para experimentos com resolução HD720p, foi utilizada a sequência de vídeo *Mobcal* e, por fim, a sequência de vídeo *Blue Sky* foi considerada para vídeos HD1080p (comercialmente referida por *FullHD*). Na Figura 6.1 são apresentados os primeiros quadros das sequências de vídeos utilizadas nos experimentos realizados.

(a) CIF: *Foreman*

(b) HD720p: *Mobcal*

(c) HD1080p: *Blue Sky*

Figura 6.1: Primeiros quadros das três resoluções de vídeo consideradas nos experimentos: *Foreman* – CIF, *Mobcal* – HD720p e *Blue Sky* – HD1080p.

Na Figura 6.2 é ilustrado o diagrama de tempo das implementações em CUDA, incluindo as quatro principais etapas da execução do algoritmo em GPU: alocação dos dados, tempo de transferência dos dados da CPU para GPU, execução do *kernel* e tempo de transferência dos dados da GPU para CPU.

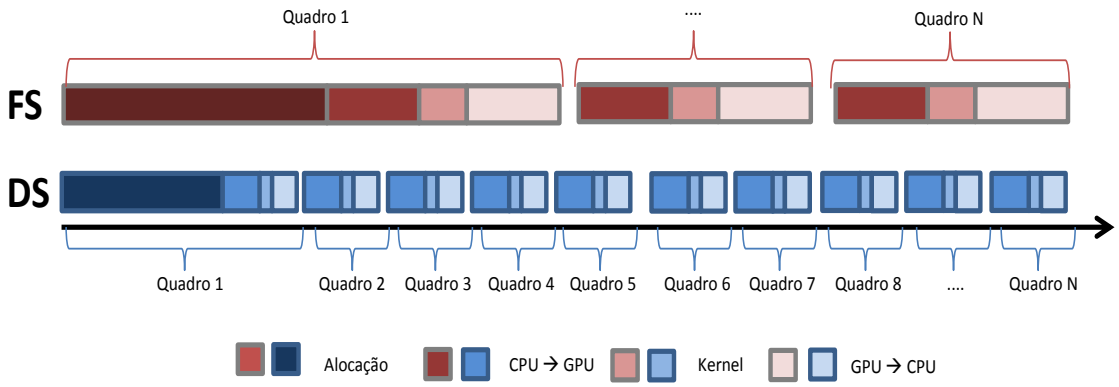


Figura 6.2: Tempo de Execução do FS e DS em CUDA.

É importante notar na Figura 6.2 que o tempo de alocação dos recursos (*threads* e blocos de GPU) em CUDA, para ambos os algoritmos, representa a etapa mais custosa. No algoritmo FS, o tempo de alocação de recursos representa 46.62% do tempo total de execução para codificação do primeiro quadro de uma sequência de vídeo, enquanto para transferência dos dados da CPU para GPU se refere a 26.89%, execução do *kernel* representa apenas 1.93% e quando da transferência dos dados de volta para GPU trata de 26.39%. Por sua vez, a alocação dos dados na execução do algoritmo *Diamond Search* representa 83.78% para a codificação do primeiro quadro que compõe o vídeo digital, 12.87%, 0.89% e 3.22% para as outras etapas, respectivamente.

A alocação dos recursos do GPU para o armazenamento local das amostras necessárias é requerida apenas para o início do processamento do vídeo. Logo, esta etapa representa apenas uma latência inicial, a qual é amortizada ao longo do processamento dos quadros. Sendo assim, neste trabalho o tempo de execução por quadro será representado apenas pelas seguintes etapas: ***(i)*** tempo de transferência dos dados da CPU para GPU, ***(ii)*** tempo de execução do *kernel* e ***(iii)*** tempo de transferência dos dados resultantes da GPU para CPU.

A definição dos experimentos foi feita pelas combinações de todos os parâmetros descritos anteriormente. Cada experimento foi repetido por trinta vezes e, ao final, a média dos resultados foi calculada. No total, cada algoritmo foi executado 3.696 vezes, totalizando 221.760 execuções das versões da Estimação de Movimento propostas e apresentadas neste trabalho.

A fim de comparar a qualidade dos resultados obtidos em relação ao paralelismo desenvolvido, considerou-se a métrica de desempenho denominada *speed-up*. Esta métrica tem como principal objetivo avaliar o desempenho de diferentes paradigmas computacionais, baseando-se nos tempos médios de execução de cada um. Tipicamente, o *speed-up* é calculado como a razão entre o tempo de execução serial e o tempo de execução paralelo:

$$Speed - up = \frac{T_1}{T_p} \qquad (2)$$

A definição de *speed-up* é dada em (2), onde p é o número de processadores, T_1 refere-se ao tempo de execução serial do algoritmo e T_p ao tempo de execução paralelo do algoritmo considerando p processadores envolvidos.

6.3 *Full Search*

Os dados referentes à avalição do algoritmo FS em GPU foram extraídos para as três resoluções mencionadas: CIF, HD720p e HD1080p. Os parâmetros utilizados nos experimentos deste algoritmo consideram *n* variando de 12 a 80, onde *n* * *n* é o número de *pixels* pertencentes no interior da área de busca.

6.3.1 Resultados para a Resolução CIF

A Figura 6.3 ilustra o tempo de execução da Estimação de Movimento em CUDA (vermelho) considerando o algoritmo FS em comparação com as demais versões desenvolvidas no contexto deste trabalho: serial (azul), *multi-core* – OpenMP (verde), bem como, distribuída – MPI (roxo).

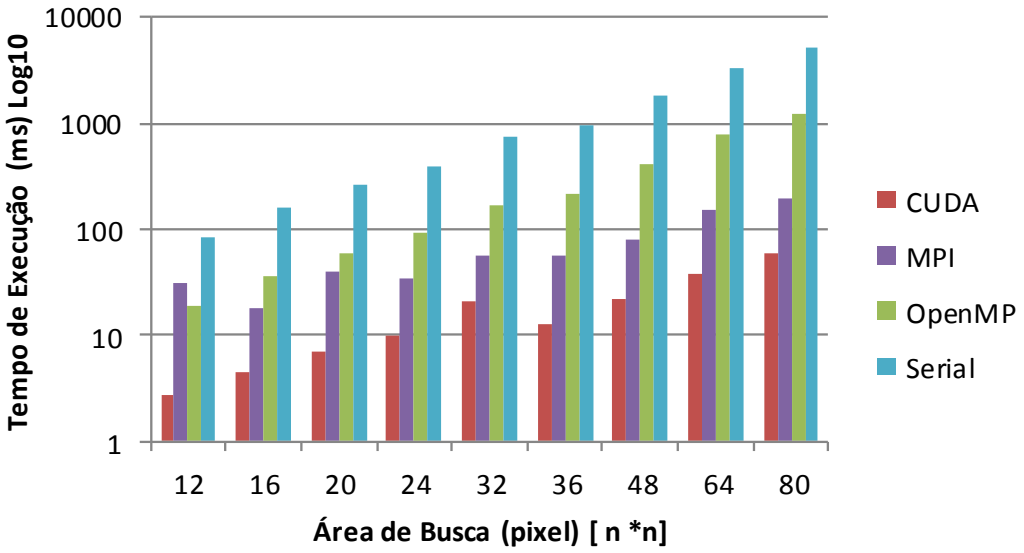


Figura 6.3: Resultado ME FS - CIF - Tempo de Execução: CUDA *x* MPI *x* OpenMP *x* Serial.

Vista a elevada complexidade computacional da Estimação de Movimento considerando o algoritmo FS em sua versão serial executada em software, para uma melhor visualização dos dados os dados ilustrados na Figura 6.3 são apresentados em uma escala logarítmica de base dez.

A versão ME FS em OpenMP (em verde, veja Figura 6.3) apresenta uma melhora no comportamento, em termos de tempo de execução, quando comparado com a versão serial (em azul, veja Figua 6.3) considerando o uso de seis *threads* em sua execução. No entanto, estes resultados não são suficientes quando um dos objetivos a serem alcançados é a codificação de vídeo em tempo real. Sendo assim, a versão ME FS em MPI (em roxo, veja Figura 6.3) apresenta um ganho significativo em relação à versão OpenMP, bem como comparada à versão sequencial. Sobretudo, a versão em CUDA supera todas as versões apresentadas, incluindo MPI, onde um *cluster* é utilizado para a execução do algoritmo.

Considerando que a versão MPI apresenta melhores resultados quando comparado às demais versões apresentadas, na Figura 6.4 é apresentada uma comparação entre as soluções: CUDA e distribuída, para a maior área de busca considerada (*128x128*) variando o número de processos utilizados na execução MPI (de 4 processos até 80

processos, número máximo de processos alocados permitidas pela administração do *cluster* utilizado).

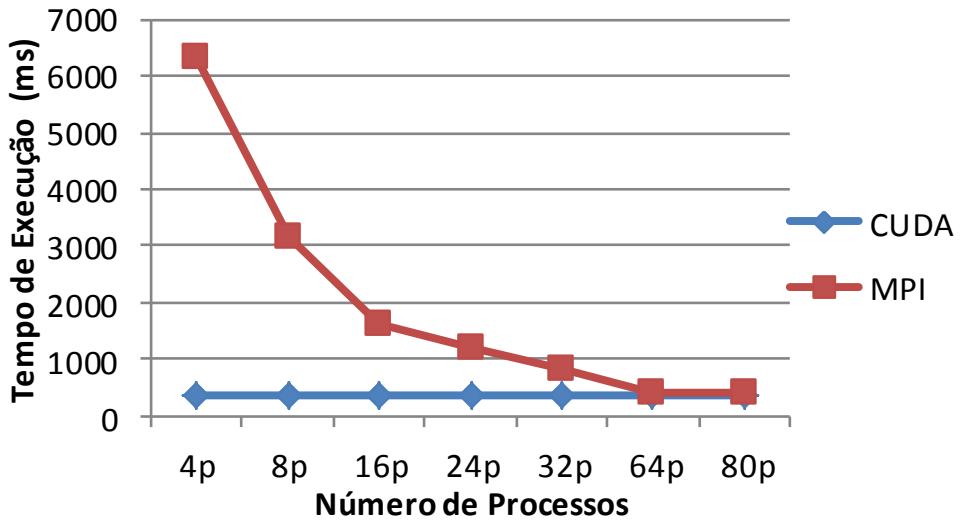


Figura 6.4: Resultado CIF para Área de Busca *128x128* - FS - Tempo de Execução: CUDA *x* MPI (4 - 80 processos).

Pode ser visualizado na Figura 6.4 que aproximadamente 64 processos (equivalente a oito nodos) são necessários para começar a obtenção de resultados semelhantes à versão ME FS em CUDA. Por esta razão, na comparação apresentada na Figura 6.4 foram considerados 64 processos.

Por fim, a Figura 6.5 ilustra um comparativo do desempenho atingido pela arquitetura CUDA (em azul) em relação aos demais paradigmas computacionais considerados (OpenMP – em verde e MPI – em vermelho) em termos de *speed-up* utilizando a resolução CIF.

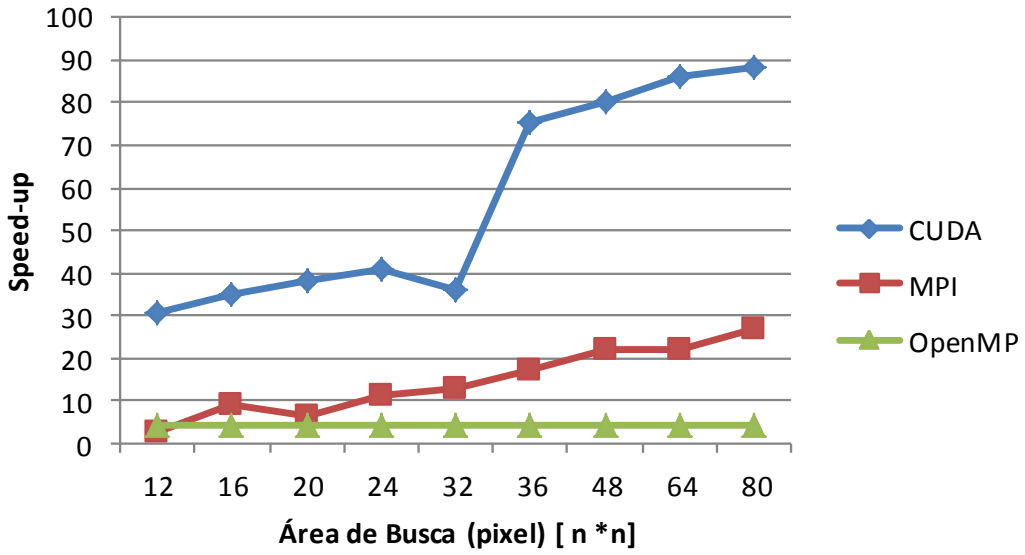


Figura 6.5: *Speed-up* FS - CIF: CUDA *x* MPI (64 processos) *x OpenMP*.

Em termos de *speed-up* (métrica adotada neste trabalho para avaliação do paralelismo proposto em GPU), analisando a Figura 6.6, pode-se perceber a superioridade apresentada pela Estimação de Movimento em CUDA em todas as dimensões de área de busca consideradas. Em especial, o *speed-up*, consideravelmente, crescente apresentado pela versão CUDA a partir da área de busca de dimensão *32x32*. No melhor caso (área de busca *80x80* na Figura 6.5), os valores de *speed-ups* obtidos através das diferentes arquiteturas paralelas propostas atingem ganhos referentes à: *89x* - CUDA, *27x* - MPI e *4x* - OpenMP, quando comparados com a versão serial. Foco deste trabalho, a versão GPGPU atinge um *speed-up 3x* mais rápido que a versão MPI e *22x* que a versão OpenMP.

Em resumo, o comportamento obtido pela ME CUDA é linear. Por outro lado, a versão MPI com 64 processos alocados, a qual apresentou resultados próximos à versão CUDA (veja Figuras 6.3 e 6.4), também possui um comportamento linear, porém com valores significativamente inferiores a versão GPGPU. Por fim, a versão OpenMP mantém-se praticamente constante para todas as áreas de busca analisadas.

6.3.2 Resultados para a Resolução HD720p

Para avaliação da ME FS proposta para vídeos de resolução HD720p, a sequência de vídeo *Mobcal* foi considerada. A Figura 6.6, por sua vez, ilustra uma comparação da versão CUDA (em vermelho) referente à Estimação de Movimento utilizando o algoritmo FS na resolução HD720p entre os demais paradigmas computacionais considerados neste trabalho.

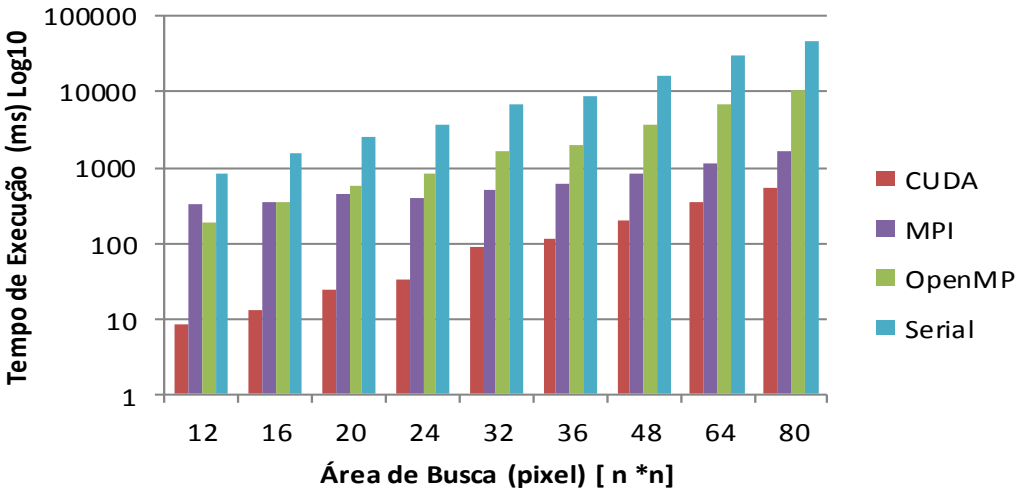


Figura 6.6: Resultado ME FS – HD720p - Tempo de Execução: CUDA *x* MPI *x* OpenMP *x* Serial.

A escala logarítmica também foi utilizada na representação dos dados na Figura 6.6 pela mesma razão mencionada na seção anterior, ou seja, pela grande discrepância dos resultados obtidos a partir da versão serial da Estimação de Movimento (considerando o algoritmo FS) apresentados em relação aos demais ambientes considerados (CUDA, OpenMP e MPI).

Na Figura 6.6, o grande ganho obtido pela versão CUDA, foco desta dissertação, em relação às demais implementações pode ser facilmente visualizado. O tempo de execução da ME FS em CUDA, considerando área de busca *80x80*, é sofreu uma

redução de 68% quando comparado com a versão distribuída MPI e 95% em relação à versão *multi-core* OpenMP.

Na Figura 6.4 foram apresentados os dados que justificaram o uso de 64 processos na versão MPI para vídeos resolução de vídeo CIF. Sendo assim, na resolução HD720p foram alocados também 8 nodos (total de 64 processos) para execução da ME FS em MPI (resultados em roxo, Figura 6.6).

Na Figura 6.7 é apresentada uma ilustração que compara CUDA com MPI, variando o número de processos alocados no cluster e considerando uma área de busca *80x80*. Este gráfico tem como objetivo apresentar o número de recursos distribuídos (processos e nodos) necessários para que os resultados de CUDA começassem a serem alcançados pela versão MPI.

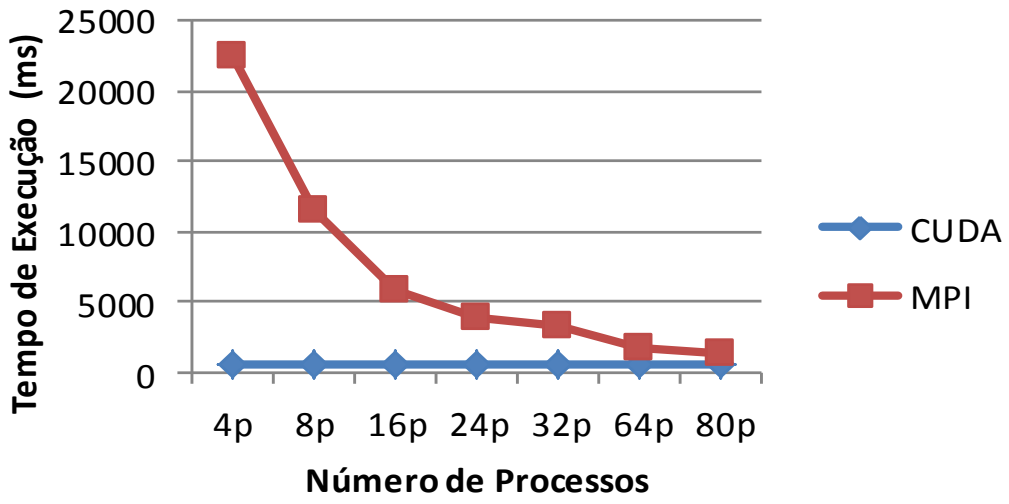


Figura 6.7: Resultado HD720p para Área de Busca *80x80* - FS - Tempo de Execução: CUDA *x* MPI (4 - 80 processos).

Analisando a Figura 6.7 se pode visualizar que os 64 processos (8 nodos), os quais se aproximavam de CUDA na quando vídeos CIF eram processados, já não são suficientes nesta resolução. Assim, para esta resolução, concluiu-se que a alocação de recursos mais apropriada para comparação com a versão ME FS em CUDA deve disparar 80 processos, utilizando 10 nodos de processamento.

Na Figura 6.8 é apresentado um comparativo referente aos resultados de *speed-up* obtidos pelas três versões da Estimação de Movimento paralelas (CUDA, MPI – 64 processos - e OpenMP – 6 *threads*) considerando diferentes dimensões de área de busca (*12x12 – 80x80*) e utilizando resolução HD720p.

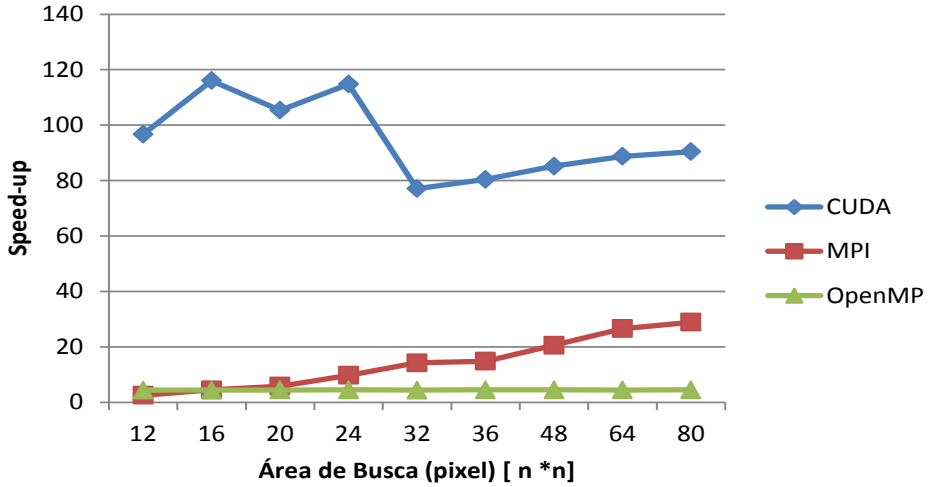


Figura 6.8: *Speed-up* FS – HD720p: CUDA *x* MPI (64 processos) *x OpenMP*.

Conforme a Figura 6.8, mais uma vez pode-se concluir que a arquitetura CUDA se destacou diante dos demais paradigmas computacionais abordados em termos de *speed-up*. A versão da Estimação de Movimento em CUDA apresentou um *speed-up* de até *115x* no melhor caso, enquanto que a versão MPI obteve um *speed-up* de *29x* e OpenMP proporcionou uma aceleração de *4x*. A versão CUDA apresenta resultados mais satisfatórios com relação às demais implementações: *speed-up 12x* maiores em relação à versão distribuída MPI e *28x* melhores do que os atingidos pela versão *multi-core* OpenMP.

6.3.3 Resultados para a Resolução HD1080p

Nesta seção, serão apresentados os resultados obtidos para resolução HD1080p. A Figura 6.9 mostra os resultados gerados com a sequência de vídeo *Blue Sky* a partir de um comparativo entre a Estimação de Movimento considerando o algoritmo FS em CUDA e os demais paradigmas considerados neste trabalho. O número de processos alocados para extração de resultados em MPI é 64 e o número de *threads* utilizadas para versão OpenMP é 6.

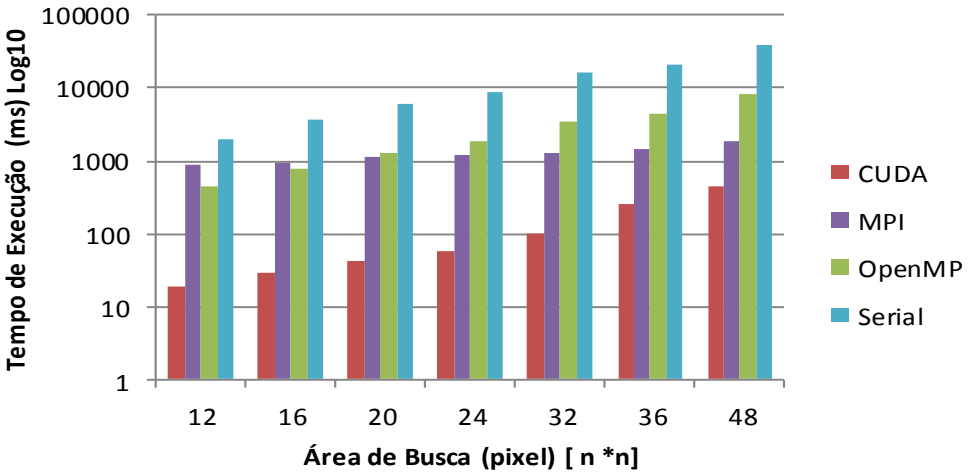


Figura 6.9: Resultado ME FS – HD180p - Tempo de Execução: CUDA *x* MPI *x* OpenMP *x* Serial.

A partir dos resultados apresentados na Figura 6.9, se pode perceber a superioridade da arquitetura CUDA, para um vídeo de alta definição, perante as demais implementações. Para a maior área de busca analisada (*48x48*), a versão ME FS CUDA reduz em 77% o tempo de execução quando comparado com a versão MPI distribuída e 95% quando comparada com a versão *multi-core* OpenMP. Com estes dados, pode-se dizer que o comportamento dos resultados obtidos em CUDA se distancia da versão distribuída quando a resolução do vídeo é acrescida.

Nos resultados apresentados na Figura 6.9 foram considerados 64 processos, equivalentes a 8 nodos alocados. No entanto, pode-se perceber que estes recursos distribuídos também não são suficientes para alcançar os resultados apresentados pela versão ME FS CUDA (bem como os resultados apresentados na resolução HD720p, Seção 6.3.2).

Pela mesma razão mencionada na descrição dos resultados das resoluções anteriores, na Figura 6.10 é ilustrado um gráfico o qual representa um comparativo da versão CUDA com a versão distribuída da Estimação de Movimento considerando o algoritmo FS utilizando vídeo de alta definição.

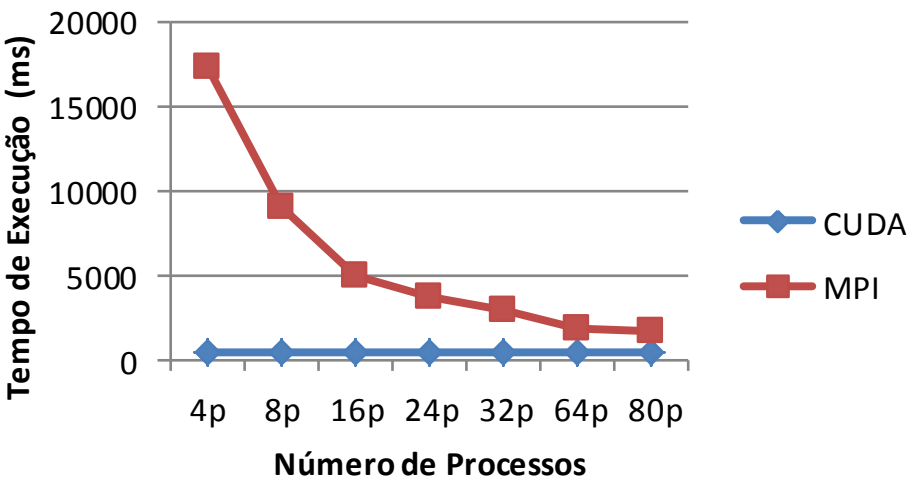


Figura 6.10: Resultado HD1080p para Área de Busca *48x48* - FS - Tempo de Execução: CUDA *x* MPI (4 - 80 processos).

Na resolução HD720p os resultados apresentado em MPI localizavam-se próximos dos resultados atingidos pela versão CUDA (veja Figura 6.8), no entanto, visualizando a Figura 6.10 pode-se perceber que o aumento da resolução do vídeo (HD1080p) faz com que a variação do número de nodos alocados se distancie dos resultados obtidos pela arquitetura CUDA. Nem mesmo o número máximo de nodos alocados, equivalentes a 10 nodos ou 20 CPUs, é suficiente para apresentar resultados iguais, ou até mesmo superiores, que os resultados obtidos através de GPU. Desta forma, quanto maior a resolução do vídeo utilizada, maior a superioridade apresentada pelo paralelismo da arquitetura CUDA.

Para finalizar a descrição de resultados obtidos do algoritmo FS proposto, na Figura 6.11 é apresentado o comparativo de *speed-up* entre a versão CUDA, MPI (64 processos) e OpenMP (6 *threads*). A ME GPGPU apresenta um ganho significativo em relação as demais versões apresentadas.

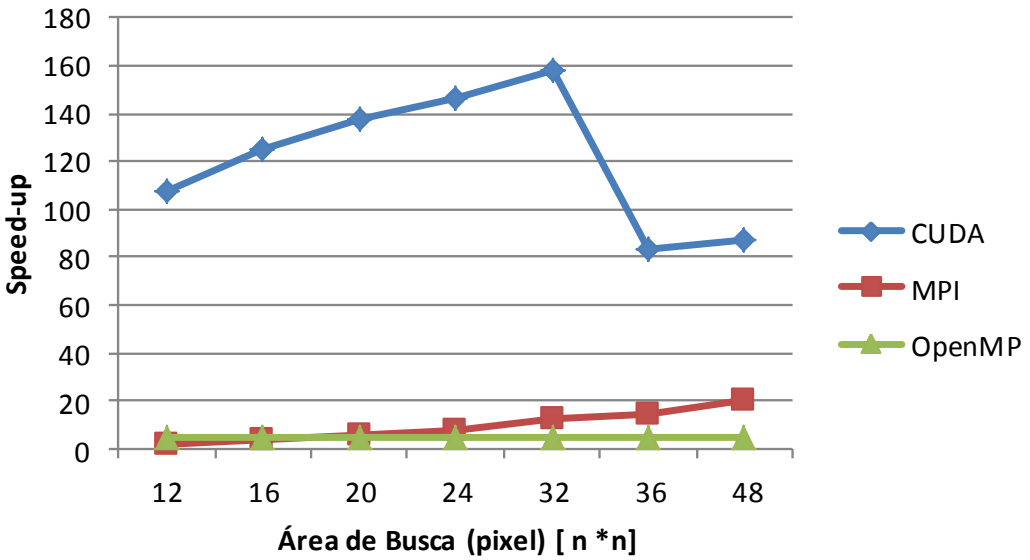


Figura 6.11: *Speed-up* FS – HD1080p: CUDA *x* MPI (64 processos) *x OpenMP*.

O *speed-up* atingido pela versão CUDA atinge, aproximadamente, refere-se à *160x*. A versão MPI, por sua vez, atinge um valor máximo de *20x* e OpenMP *5x*. Os ganhos obtidos com a versão ME GPGPU representam: *13x* quando comparado com a versão distribuída e *31x* com OpenMP.

6.3.4 Comparação entre as Resoluções

Esta seção tem por objetivo comparar e discutir os resultados obtidos em cada uma das resoluções alvos, os quais já foram apresentados nas seções anteriores. Primeiramente, na Tabela 6.1 é apresentado um comparativo referente aos valores de *speed-ups* máximos atingidos pelos diferentes paradigmas computacionais através das três resoluções.

Os valores de *speed-up* apresentados na Tabela 6.1 representam os ganhos atingidos pelas versões da ME em relação a sua respectiva versão serial. Todas as alternativas propostas, em geral, apresentam um acréscimo no valor de *speed-up* com o aumento da resolução do vídeo. Em especial, a versão CUDA, apresenta resultados mais expressivos, com aumentos de 29% da resolução CIF para HD720p e 39% da resolução HD720p para HD1080p.

Tabela 6.1: Relação das versões da ME FS entre Resoluções – *Speed-up*.

Resolução de Vídeo	**CUDA**	**MPI**	**OpenMP**	**Aumento de *Speed-up* CUDA (%)**
CIF	89	27	4	-
HD720p	115	29	4	**+29%**
HD1080p	160	20	5	**+39%**

Na Tabela 6.2 é apresentado um comparativo mais detalhado dos ganhos obtidos pela implementação do FS em CUDA, desenvolvida neste trabalho. Este comparativo relaciona as três resoluções de vídeo comparadas com as versões paralela e distribuída: MPI e OpenMP.

Tabela 6.2: Reduções e Ganhos com Relação a ME FS CUDA.

Resolução do Vídeo	**Redução do Tempo de Execução (%)**		**Aumento do *Speed-up***	
	MPI	**OpenMP**	**MPI**	**OpenMP**
CIF	68%	95%	*3x*	*22x*
HD720p	68%	95%	*12x*	*28x*
HD1080p	77%	95%	*13x*	*31x*

Analisando as primeiras colunas da Tabela 6.2, salienta-se o percentual da redução do tempo de execução atingido pela solução em CUDA em relação à versão distribuída e paralela. Esta redução se refere às maiores áreas de busca consideradas em cada uma das resoluções (CIF, HD720p, HD1080p), *80x80*, *80x80* e *48x48*, respectivamente. Para as resoluções CIF e HD720p, as taxas de redução atingidas por CUDA se mantiveram em comparação com ambos os paradigmas, MPI e OpenMP, com valores referentes à 68% e 95% respectivamente. Por outro lado, para a resolução HD1080p o percentual de redução, em relação a versão distribuída, é ainda maior, representando 77%. Estes resultados são considerados promissores, visto que, o aumento na resolução do vídeo (essencial na codificação de vídeo) em questão implica em ganhos significativos em relação às demais plataformas analisadas. Em relação a versão *multi-core* OpenMP, é possível observar um percentual representado por 95% em todas as resoluções. Assim, todos estes valores mencionados podem ser visivelmente justificados pelos valores de *speed-up* alcançados.

Uma análise mais específica dos valores dos ganhos em *speed-up* (veja Tabela 6.1) da versão do FS em CUDA proposta neste trabalho com relação com as demais versões paralelas é mostrada nas últimas colunas da Tabela 6.2. As menores diferenças em termos de *speed-up* referem-se na comparação entre CUDA e MPI, fato este que se justifica pela massiva utilização de nodos de processamento (chegando a 10 nodos no máximo) e pela extensiva paralelização do algoritmo em até 80 processos executando simultaneamente. Estas diferenças (CUDA *x* MPI) representam um ganho de *3x* da versão CUDA em relação a resolução CIF, *12x* referente a resolução HD720p e, por fim, *13x* com a resolução de alta definição HD1080p. Por sua vez, quando comparados os valores de speed-up atingidos pela versão CUDA em relação a versão OpenMP tem-se os seguintes ganhos: *22x* em relação a resolução CIF, *28x* em referente a HD720p e *31x* para HD1080p.

Em todos os parâmetros da Tabela 6.2 podem-se observar melhores resultados quando a resolução do vídeo a ser codificado é aumentada, tanto no percentual referente ao tempo de execução, quanto na melhoria de *speed-up* apresentada.

A última análise dos resultados referente à implementação GPGPU do algoritmo FS concentra-se em uma comparação dos valores de *speed-up* obtidos entre as três resoluções de vídeo consideradas no escopo deste trabalho: CIF (em azul), HD720p (em vermelho) e HD180p (em verde). Neste cenário, na Figura 6.12 é ilustrado um comparativo considerando uma variação de área de busca de *12x12 pixels* até *48x48 pixels*.

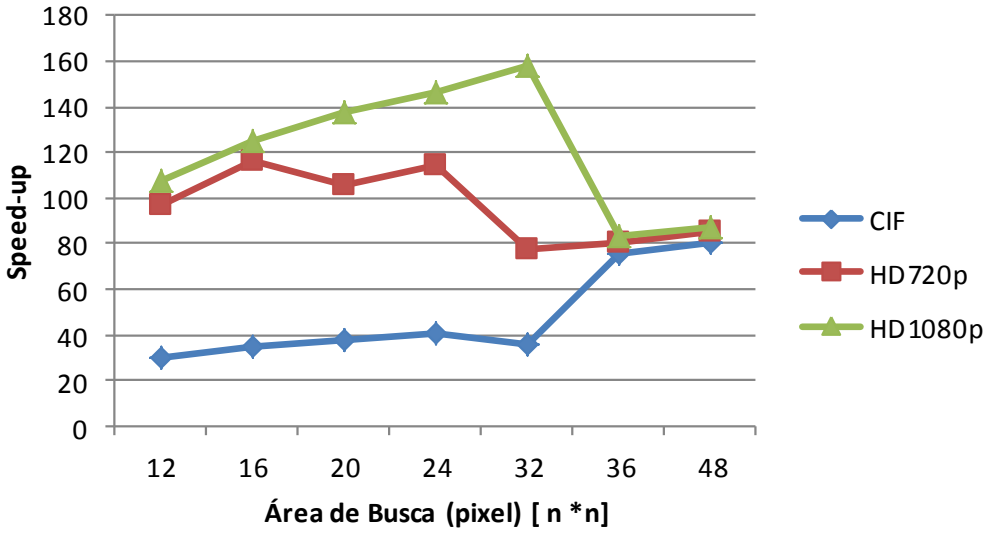


Figura 6.12: *Speed-up* FS CUDA – CIF *x* HD720p *x* HD1080p.

Na Figura 6.12 pode-se observar uma comparação entre as três resoluções variando o tamanho de área de busca. Os valores de *speed-up* atingem maiores resultados quando a resolução do vídeo é aumentada, isto é, para resolução CIF um pico de 80 é atingido, para resolução HD720p o *speed-up* 116 é alcançado e, por fim, para HD1080p o valor é de 159 . Além disso, nota-se que há uma queda de rendimento expressiva nas resoluções de alta definição HD720p e HD1080p entre as áreas de busca *24x24* e *32x32*. Enquanto na resolução CIF um aumento significativo é apresentado.

Considerando que a complexidade computacional do FS é dada de maneira direta pelo aumento da área de busca em questão, a estratégia de paralelização deste algoritmo em GPU concentrou-se em aumentar o paralelismo na medida em que a área de busca cresce. Para este fim, o tamanho do *block* (bloco de GPU) é alocado de acordo com a dimensão da área de busca. Desta forma, quanto maior a área de busca, mais *threads* estarão disponíveis, proporcionando aumento do paralelismo. Contudo, apesar dos expressivos resultados alcançados, em alguns pontos esta estratégia não atingiu os resultados esperados.

Apesar da GPU utilizada na execução destes experimentos viabilizar recursos que suportariam grandes tamanhos de área de busca, a API restringe o número de recursos (*threads* e *blocks*) que podem ser executados de forma paralela no mesmo ciclo de *clock*. Por esta razão há uma queda no rendimento do algoritmo para a codificação dos vídeos de alta definição, visto que o número de recursos alocados para suprir a demanda de processamento ultrapassa o limite permitido pela API, fazendo com que seja necessário um escalonamento das *threads* envolvidas. Este *overhead* tem impacto de maneira bem clara na Figura 6.12, quando o aumento da área de busca causa um decréscimo no desempenho da solução proposta em CUDA.

Esta queda de rendimento varia de acordo com o modelo e a capacidade de processamento de cada placa gráfica. Qualquer que seja o modelo de dispositivo utilizado haverá uma área de busca ótima em que o tamanho do *block* da GPU seja adaptado da melhor maneira possível. Para o dispositivo gráfico utilizado neste trabalho, o melhor tamanho de área de busca que proporciona o melhor desempenho para os vídeos de alta resolução é *32x32*.

6.4 Diamond Search

Nesta seção serão apresentados os resultados obtidos quando da execução em GPU da Estimação de Movimento considerando o algoritmo rápido DS. O DS pode ser considerado um algoritmo de busca sub-ótimo e introduz dependência de dados ao longo do processamento de um bloco, visto que a próxima iteração depende do resultado da iteração atual, e assim por diante. Esta dependência fez com que a estratégia de implementação fosse diferente daquela utilizada para o FS. Consequentemente, todas as conclusões tiradas na seção anterior para o FS não são diretamente transferidas para o DS.

Os testes realizados para o DS consideram três diferentes resoluções de vídeo, CIF, HD720p e HD1080p, para dimensões de área de busca de *12x12 pixel* até *240x240 pixels*, considerando o bloco de vídeo *4x4*.

6.4.1 Resultados para a Resolução CIF

Nesta subseção serão apresentados os resultados obtidos com a resolução de vídeo CIF (*352x288*) na execução do algoritmo DS proposto neste trabalho. A Figura 6.13 ilustra os resultados obtidos pela ME DS em CUDA (em vermelho) em comparação com os demais paradigmas computacionais considerados: Serial (em azul), OpenMP (em verde) e MPI (em roxo).

Na Figura 6.13 pode-se observar os tempos de execução de todas as implementações do DS. O pior resultado, como já esperado, é observado na versão puramente serial. A utilização da API OpenMP conseguiu ganhos na paralelização do algoritmo, mas ainda esteve longe dos resultados atingidos pelas massivas paralelizações utilizadas nas implementações em CUDA e MPI.

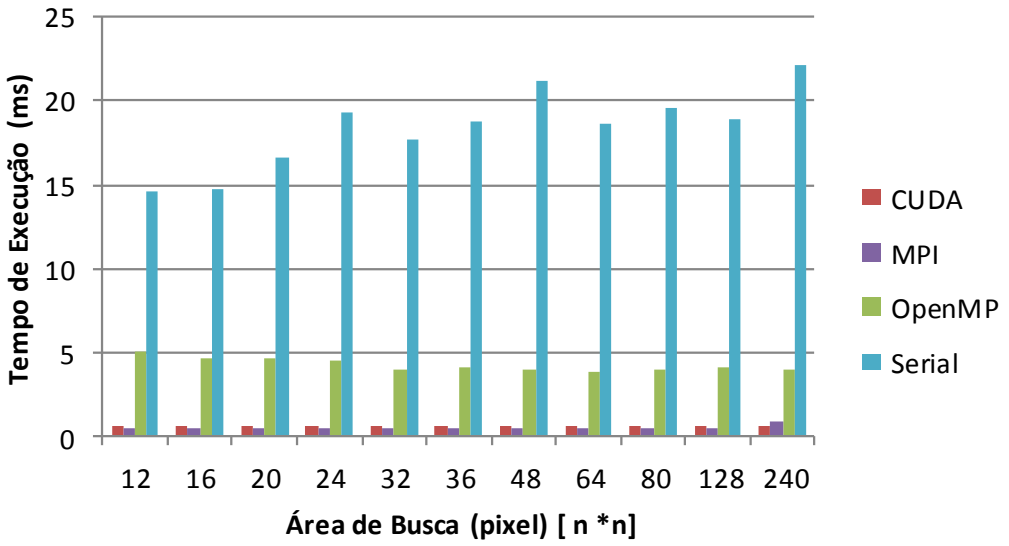


Figura 6.13: Resultado ME DS – CIF - Tempo de Execução: CUDA *x* MPI (64 processos) *x* OpenMP *x* Serial.

A Figura 6.14 apresenta em destaque uma comparação entre as seguintes abordagens: CUDA *x* MPI (64 processos), as quais representam os melhores resultados alcançados em termos de tempo de execução.

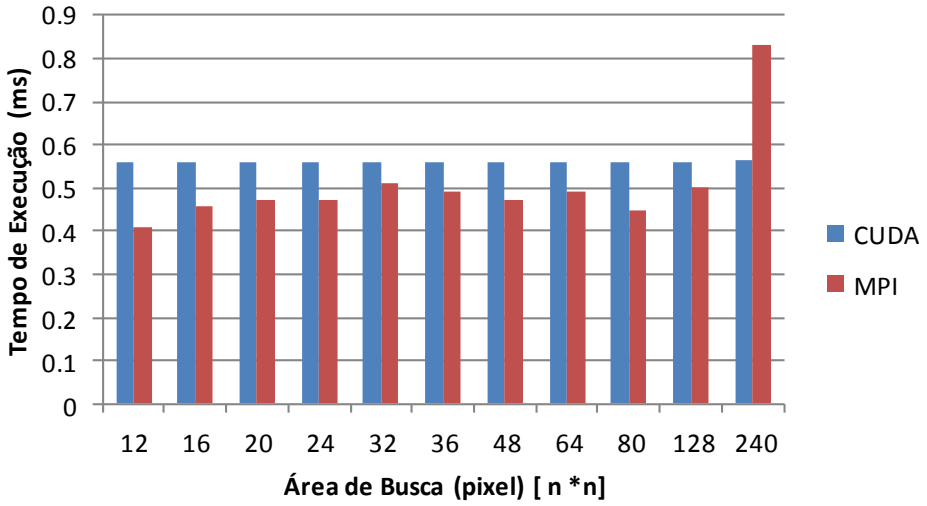


Figura 6.14: Resultado ME DS – CIF - Tempo de Execução: CUDA *x* MPI (64 processos).

Na Figura 6.14 são apresentados os resultados referentes à execução do algoritmo DS considerando resolução de vídeo CIF em comparação entre as versões CUDA e MPI. Nesta ilustração percebe-se que a versão distribuída é superior (menor tempo de execução) em relação à versão GPGPU em quase todas as dimensões de áreas de busca abordadas.

Mesmo que na avaliação destes experimentos o tempo de alocação dos dados na memória da GPU não tenha sido contabilizado (veja Seção 6.5), a comunicação entre os dispositivos CPU e GPU (transferência dos dados entre eles) é um dos fatores responsáveis pelos resultados de CUDA apresentarem-se inferiores em relação à versão distribuída. Este fator é significativo, pois o tempo de comunicação entre os paradigmas (CPU e GPU) representa um custo elevado e, neste caso, não compensa a execução de um algoritmo rápido sob um pequeno volume de dados, isto é, considerando uma baixa resolução de vídeo.

No entanto, na área de busca *240x240 pixels* a versão ME DS CUDA apresenta ganhos consideráveis no tempo de execução quando comparado com a versão MPI. Estes ganhos devem-se a compensação (amortização) do tempo de comunicação entre os dispositivos em relação ao crescimento do volume de dados executado.

A utilização de 64 processos na versão distribuída quando vídeos HD1080p são processados é justificada pela Figura 6.15, onde é ilustrada uma comparação entre a ME DS CUDA e MPI para uma área de busca fixa de *240x240 pixels* e variando o número de processos alocados no *cluster*.

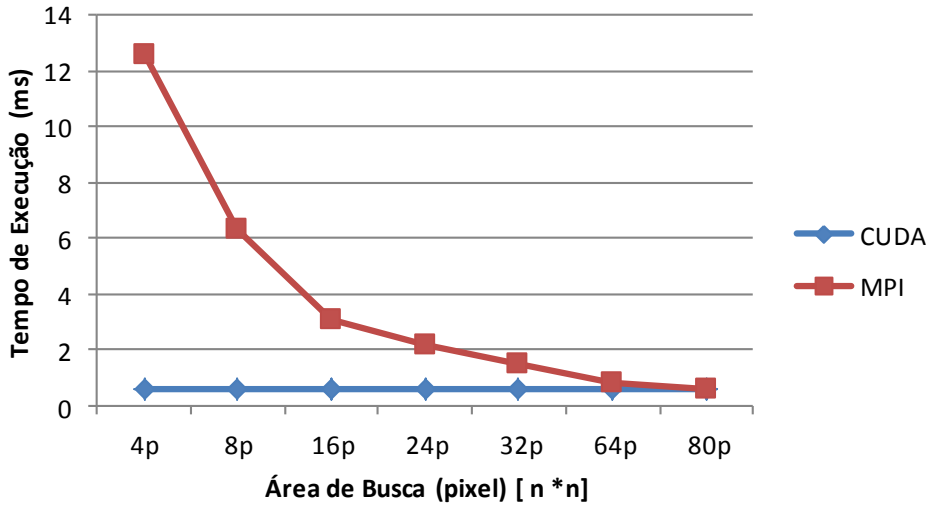


Figura 6.15: Resultado CIF para Área de Busca *240x240* - DS - Tempo de Execução: CUDA *x* MPI (4 - 80 processos).

É possível perceber na Figura 6.15 que o tempo de execução da versão MPI decresce a medida que o número de processos considerados é aumentado. Desta forma, quando o número de processos é igual a 64 os resultados em relação a versão CUDA tornam-se semelhantes fazendo com que este parâmetro seja utilizado nos experimentos desde algoritmo na resolução CIF.

Para uma avaliação do paralelismo proposto para os todos os paradigmas paralelos considerados no escopo deste trabalho, na Figura 6.16 é apresentado um comparativo entre os valores de *speed-up* alcançados.

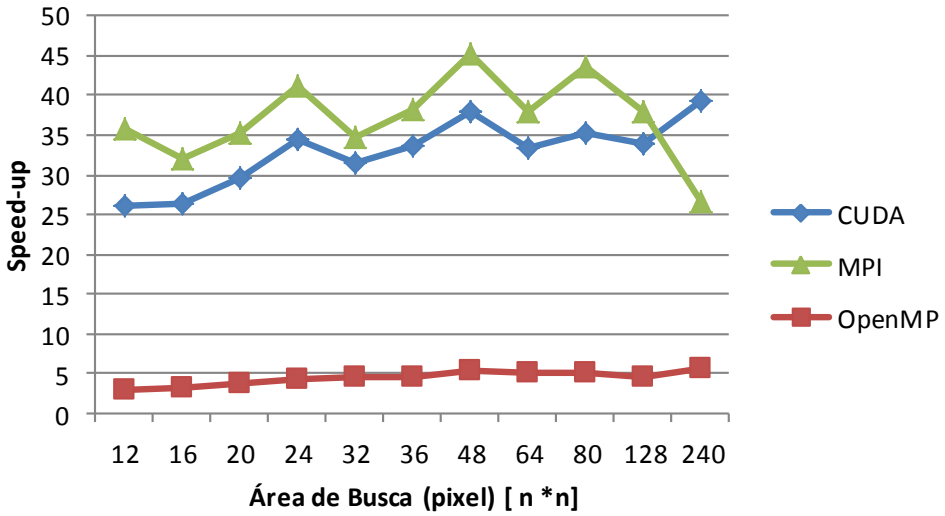


Figura 6.16: *Speed-up* DS - CIF: CUDA *x* MPI (64 processos) *x OpenMP*.

Os resultados apresentados na Figura 6.16 ilustram os valores de *speed-up* referentes aos tempos de execução representados pelas Figuras 6.13 e 6.14. Os *speed-ups* atingidos pela versão OpenMP não representam grandes resultados pois são comparados com as versões GPGPU e MPI que fazem o uso de maiores recursos computacionais em suas

execuções. Desta forma, a versão DS MPI, em resumo, apresenta resultados de *speed-up* mais expressivos que a versão CUDA, a salvo dos resultados na área de busca *240x240*, onde a versão CUDA atinge melhores resultados, e consequentemente, um maior *speed-up* que a versão MPI, a qual neste parâmetro tem uma queda de rendimento. Os ganhos obtidos em relação à versão DS serial atingem, aproximadamente os seguintes valores: *speed-up* 39 – DS CUDA (área de busca *240x240*); *speed-up* 45 – MPI (área de busca *48x48*) e, *speed-up* 6 – OpenMP (área de busca *240x240).*

6.4.2 Resultados para a Resolução HD720p

Nesta seção serão apresentados os resultados obtidos na execução do algoritmo DS considerando a resolução de vídeo HD720p (*1280x720*). Na Figura 6.17, é apresentada uma comparação dos dados obtidos na versão ME DS CUDA em comparação com as demais tecnologias consideradas: Serial, MPI e OpenMP.

Nesta resolução, a versão do algoritmo rápido DS em CUDA começa a obter ganhos em relação ao ambiente distribuído, o qual considera a alocação e utilização de 64 processos. Por outro lado, a versão Serial manteve seu desempenho inferior aos demais, apresentando um tempo de execução elevado. Desta forma, pode-se analisar uma redução apresentada pela versão OpenMP.

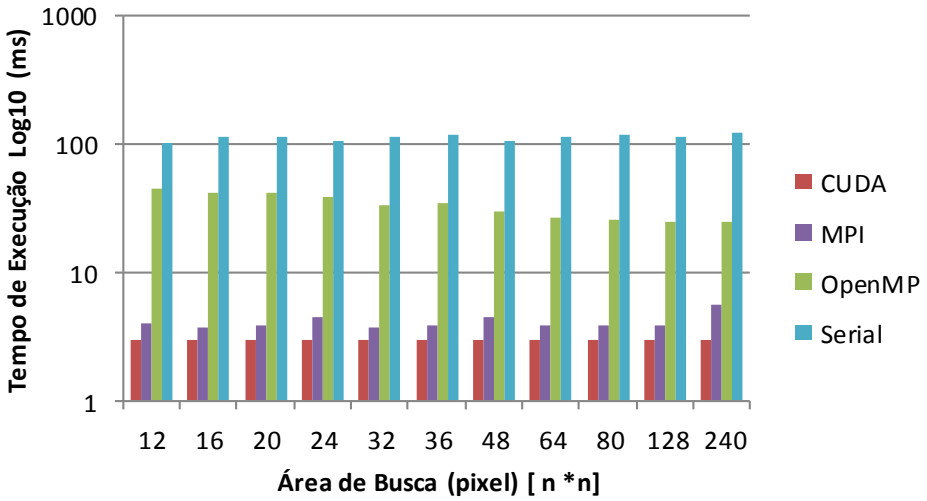


Figura 6.17: Resultado ME DS – CIF - Tempo de Execução: CUDA *x* MPI (64 processos) *x* OpenMP *x* Serial.

Conforme a Figura 6.17, a versão do algoritmo DS GPGPU foi superior a todas as implementações apresentadas, em termos de desempenho. Em especial, considerando a dimensão da área de busca *240x240 pixels*, representa uma redução de 47% em relação à versão distribuída (MPI) e, por sua vez, reduz 88% referente à versão *multi-core* OpenMP.

Da mesma forma como nas demais descrições dos resultados, a justificativa referente à alocação de 64 processos ou 8 nodos na resolução HD720p do DS, é apresentada na Figura 6.18.

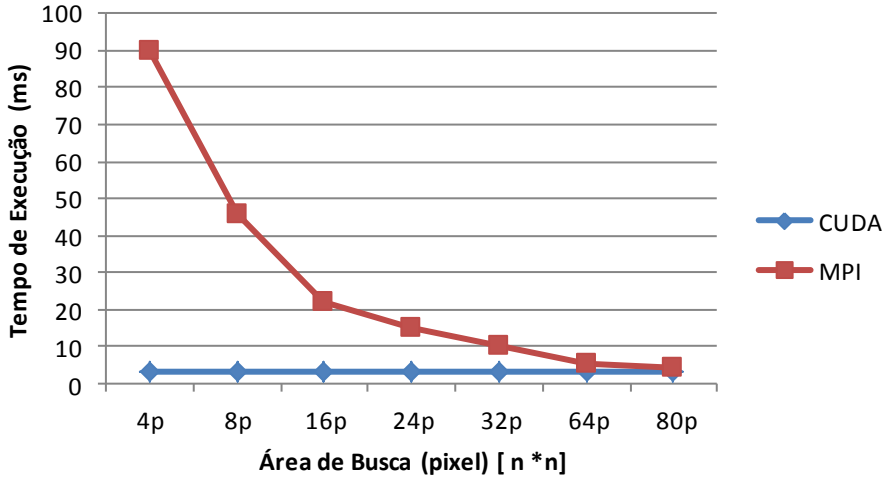


Figura 6.18: Resultado HD720p para Área de Busca *240x240* - DS - Tempo de Execução: CUDA *x* MPI (4 - 80 processos).

Na Figura 6.18 os dados apresentados referem-se à área de busca *240x240 pixels* (maior dimensão considerada na execução do algoritmo DS) e o valor do tempo de execução da versão DS GPGPU foi fixado enquanto o número de processos alocados é variado de 4 até 80. Esta comparação foi realizada para possibilitar a visualização de quantos processos são necessários para alcançar os resultados obtidos na versão CUDA.

Na Figura 6.18, pode-se observar que quando são utilizados 64 processos os comportamentos são aparentemente próximos, embora a versão CUDA ainda tenha um melhor desempenho. Desta forma, este foi o número de processos alocados e utilizados em todas as comparações ilustradas nesta seção. Salientando que 64 processos equivalem a 8 nodos, onde cada nodo é composto por duas CPUs, isto é, são necessários 16 CPUs para que a comparação com 1 GPU seja justa.

Para finalizar os resultados obtidos na execução do algoritmo rápido DS, na Figura 6.19 são apresentados os valores de *speed-up* obtidos na versão CUDA em comparação com os demais paradigmas computacionais abordados.

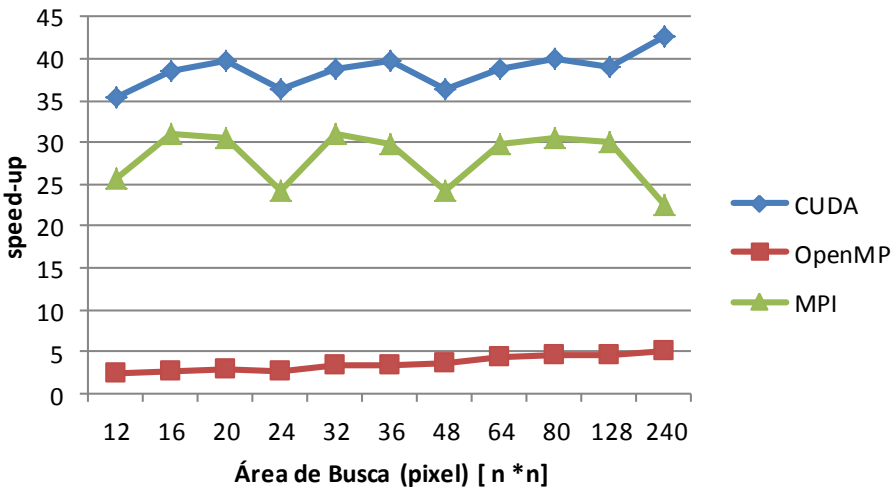


Figura 6.19: *Speed-up* DS – HD720p: CUDA *x* MPI (64 processos) *x OpenMP*.

Na Figura 6.19 o DS GPGPU destaca-se em relação a superioridade dos dados apresentada em termos de *speed-up*. Analisando os resultados em tempo de execução representados pela Figura 6.17, a diferença dos dados obtidos entre CUDA e MPI, aparentemente, é pequena. No entanto, na Figura 6.19 pode-se ter uma percepção clara de que os resultados, em *speed-up*, da versão DS CUDA são consideravelmente superiores à versão distribuída. Além disso, o *speed-up* apresentado por MPI cai consideravelmente quando o aumento do volume dos dados cresce, neste caso na área de busca *240x240*, enquanto a versão CUDA aumenta. Este fato ocorre porque os resultados atingidos em MPI saturam a partir da alocação de 64 processos para execução da ME DS, ou seja, os ganhos obtidos quando da alocação de 16 processos a mais (ou 2 nodos – 4 CPUs) já não são mais significativos com o aumento do número de recursos computacionais.

Em relação à versão Serial, estes resultados representam ganhos de até *43x* considerando a versão CUDA, *31x* em relação a versão distribuída e até *6x* referente a versão *multi-core*.

6.4.3 Resultados para a Resolução HD1080p

Por fim, esta seção finaliza a descrição dos resultados obtidos da Estimação de Movimento em GPU proposta nesta dissertação considerando uma resolução de vídeo de alta definição, HD1080p (*1920x1080*). Os resultados apresentados na Figura 6.20 concentram-se na ilustração dos tempos de execução alcançados pela versão ME CUDA em comparação com a versão Serial, MPI e OpenMP.

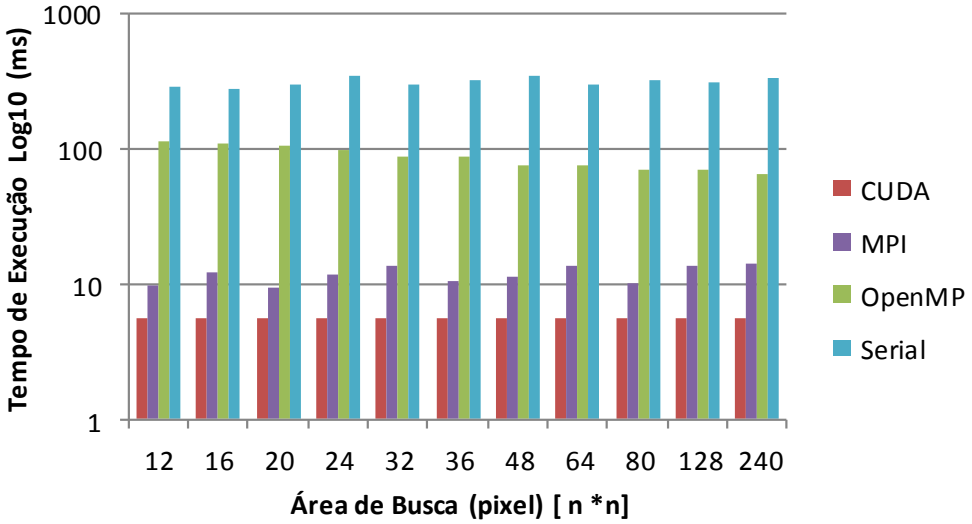


Figura 6.20: Resultado ME DS – HD1080p - Tempo de Execução: CUDA *x* MPI (64 processos) *x* OpenMP *x* Serial.

Na Figura 6.20, pode-se observar que as versões OpenMP e MPI apresentam ganhos em relação a versão Serial, e principalmente, que a ME DS EM CUDA é superior as demais versões. A versão GPGPU na resolução de alta definição (HD1080p) apresenta melhorias ainda maior quando comparado a versão MPI, em relação a resolução apresentada anteriormente (HD720p, veja seção 6.4.2).

Considerando a maior dimensão de área de busca neste experimento (*240x240*), a versão do DS em CUDA reduz o tempo de execução em 60% em relação a versão MPI e 92% referente ao tempo apresentado pela versão implementada em OpenMP.

A justificativa pela utilização de 64 processos na comparação da versão MPI é dada da mesma forma como já mencionado anteriormente. Veja a Figura 6.21.

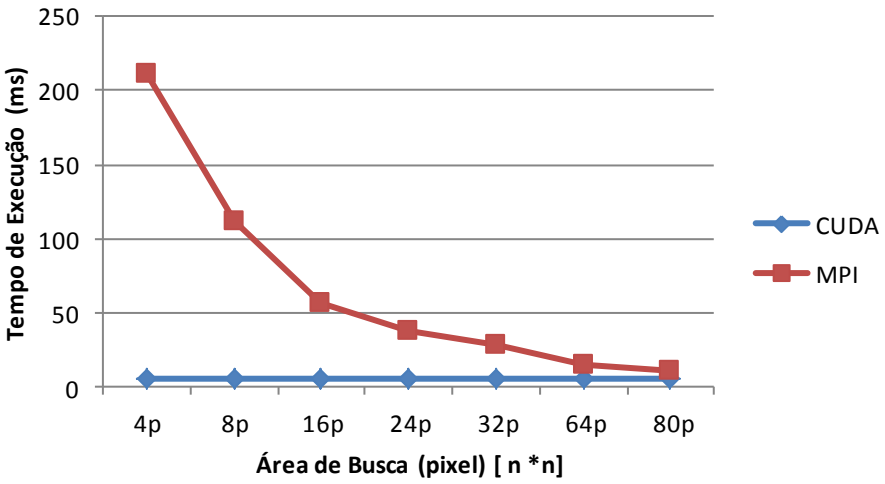


Figura 6.21: Resultado HD1080p para Área de Busca *240x240* – DS – Tempo de Execução: CUDA *x* MPI (4 - 80 processos).

Apesar do comportamento semelhante destes resultados em relação aos demais gráficos apresentados neste contexto, a Figura 6.21 refere-se aos dados obtidos em uma resolução de vídeo de alta definição, HD1080p. Assim, mais uma vez pode-se perceber que os resultados alcançados com 64 processos começam a se tornar semelhantes em relação ao comportamento apresentando por CUDA.

Por fim, a superioridade da versão DS CUDA pode ser visualizada na Figura 6.22, a qual apresenta um experimento que ilustra o comportamento das versões DS (CUDA, MPI, OpenMP) em termos de *speed-up* para a resolução HD1080p.

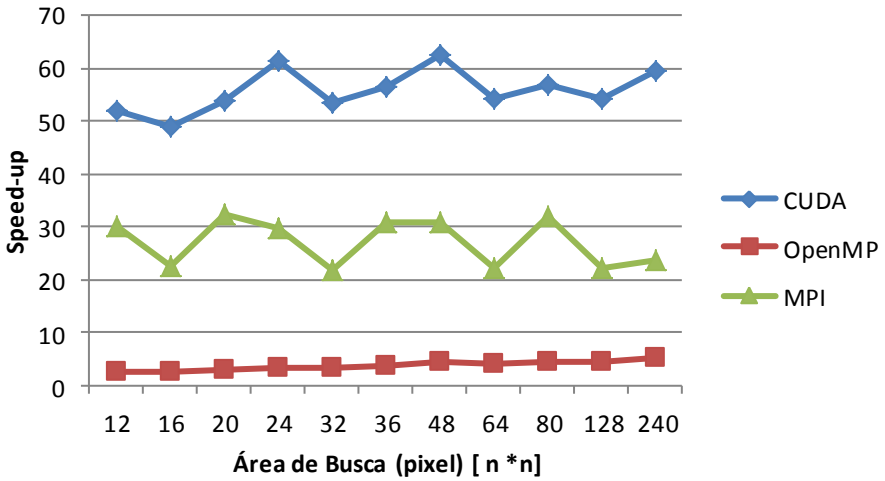


Figura 6.22: *Speed-up* DS – HD1080p: CUDA *x* MPI (64 processos) *x OpenMP*.

Conforme a Figura 6.22 o *speed-up* atingido por CUDA representa um ganho de até *63x* em relação a versão serial, enquanto a versão distribuída apresenta uma melhoria de *32x* em termos de desempenho, e por fim, a versão *muli-core* representa 6 de *speed-up*. Além disso, pode-se perceber, mais uma vez, que o *speed-up* apresentado por MPI satura quando dimensões de áreas de busca maiores são consideradas (*128x128 – 240x240*) enquanto na versão GPGPU apresenta ganhos significativos com o aumento da área de busca.

6.4.4 Comparação entre as resoluções

Após a apresentação e descrição de todos os resultados obtidos na execução do algoritmo DS em GPU em comparação com os demais paradigmas computacionais considerados no escopo deste trabalho, é apresentada uma análise de detalhada do desempenho deste algoritmo com o aumento gradual da resolução do vídeo a ser processado.

A Tabela 6.3 destaca os resultados de *speed-up* atingidos pelo algoritmo rápido DS, na área de busca *240x240,* considerando as três tecnologias paralelas abordadas e, além disso, dá ênfase aos ganhos de *speed-up* obtidos quando a resolução de vídeo é aumentada.

Conforme a Tabela 6.3, a versão DS CUDA apresenta um aumento no valor de *speed-up* de acordo com o aumento da resolução de vídeo utilizada, para a versão MPI este comportamento se repete, porém em uma menor proporção. No entanto a versão OpenMP, como disponibiliza 6 *threads*, atingiu um *speed-up*, aproximadamente constante.

Tabela 6.3: Relação das versões da ME DS entre Resoluções – *Speed-up*.

Resolução de Vídeo	**CUDA**	**MPI**	**OpenMP**	**Aumento de *Speed-up* CUDA (%)**
CIF	39	26	6	**-**
HD720p	43	31	6	**9 %**
HD1080p	63	32	6	**32%**

Pode-se observar que o aumento no *speed-up* alcançado pela versão DS CUDA da resolução CIF para a resolução HD720p é baixo, representando apenas um acréscimo de 9%. Por outro lado, a diferença em *speed-up* da resolução HD720p para resolução HD1080p é satisfatória, representando um aumento de 32%, visto que, nestas resoluções a superioridade de CUDA pode ser claramente visualizada (ver Figuras 6.18 e 6.21).

Na Tabela 6.4 são comparados os resultados atingidos pela Estimação de Movimento em GPU, foco deste trabalho, considerando o algoritmo DS, em comparação com as resoluções abordadas nos experimentos. Estas comparações relacionam a redução no tempo de execução (%) em relação aos demais paradigmas paralelos considerados: MPI e OpenMP. Além disso, destacam-se os fatores (em vezes) em que os *speed-ups* atingidos das versões DS CUDA foram superiores em relação as outras tecnologias para as três resoluções.

Primeiramente, analisando as primeiras colunas da Tabela 6.4, onde estão ressaltados os percentuais referentes à redução no tempo de execução da ME DS CUDA em relação às demais versões paralelas, pode-se perceber que a versão GPGPU do DS apresenta reduções satisfatórias, apresentando valores maiores que 50%. Em particular,

considerando a resolução CIF em relação a MPI, se observa que o percentual de redução foi menor: 37%. Este fato se justifica pela inferioridade da versão DS CUDA apresentada para resolução CIF quando comparado com MPI, onde o volume dos dados executados é baixo, fazendo com que o tempo de comunicação entre os dispositivos não seja compensado o suficiente para obter ganhos maiores que 16 CPUs (64 processos – 8 nodos). No entanto, na medida em que este volume dos dados é acrescido, com o uso de resoluções de vídeo maiores, este percentual é proporcionalmente aumentado chegando a alcançar uma redução de até 60% considerando a resolução HD1080p, a qual é considerada uma referência em termos de alta definição de vídeo atualmente

Tabela 6.4: Reduções e Ganhos com Relação a ME DS CUDA.

Resolução de Vídeo	**Redução do Tempo de Execução (%)**		**Aumento do *Speed-up***	
	MPI	**OpenMP**	**MPI**	**OpenMP**
CIF	32%	86%	*1.5x*	*8x*
HD720p	47%	88%	*2x*	*9x*
HD1080p	60%	92%	*2.2x*	*12x*

Por sua vez, analisando as últimas colunas da Tabela 6.4, onde são apresentados os aumentos em relação aos valores de *speed-up* para as três resoluções comparando a versão CUDA com as demais tecnologias paralelas abordadas, pode-se visualizar que há um aumento no número de vezes em que o *speed-up* de CUDA é maior que MPI e OpenMP a medida em que a dimensão da resolução do vídeo cresce. Além disso, se percebe que a versão CUDA atinge aumentos expressivos com uma superioridade de até *12x*. Estes dados são considerados satisfatórios em todas as comparações propostas, porém onde em relação a comparação entre MPI e CUDA os ganhos não parecem ser tão expressivos vista o número dos fatores alcançados (*8x* - *12x*) referentes a OpenMP. No entanto, na execução da versão distribuída são utilizados um número de recursos significativamente maiores, e, além disso, a obtenção de *speed-up* iguais ou maiores que *2x,* em relação a versão CUDA, para resoluções de alta definição já é considerado um resultado bastante significativo.

Para finalizar esta comparação entre as resoluções de vídeo consideradas da Estimação de Movimento DS proposta neste trabalho, é apresentada na Figura 6.23 uma visão geral do comportamento obtido pela versão ME DS em CUDA a partir de uma ilustração dos *speed-ups* atingidos para as três resoluções variando as dimensões de área de busca.

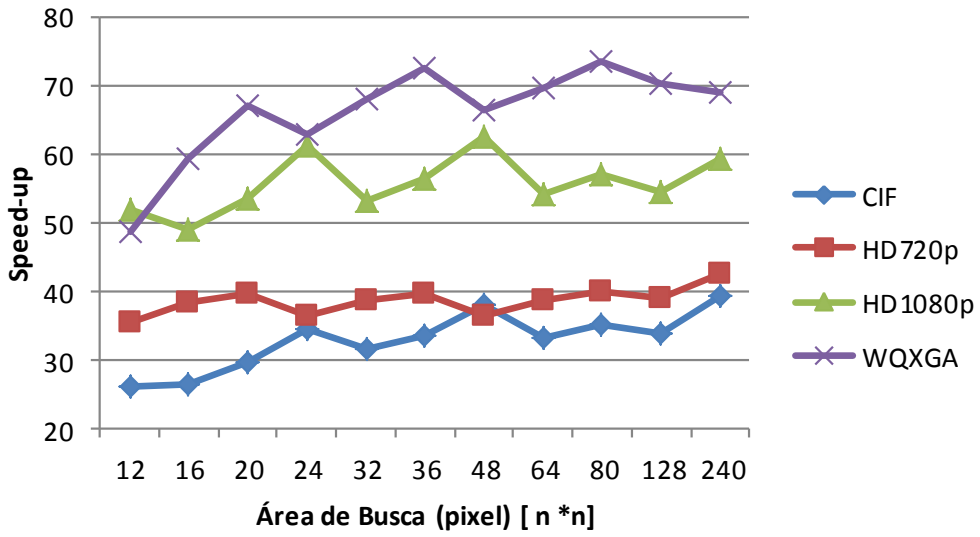


Figura 6.23: *Speed-up* DS CUDA – CIF *x* HD720p *x* HD1080p *x* WQXGA.

Observando a Figura 6.23 se percebe a inclusão de uma nova resolução de vídeo (em roxo), a qual refere-se à WQXGA (*Wide Quad eXtended Graphics Array – 2560x1600*). A utilização desta resolução neste caso específico, é justificada pelo fato do comportamento do algoritmo DS em GPU, apresentar melhores resultados de acordo com o crescimento da resolução de vídeo e da dimensão da área de busca. Desta forma, para visualizar a tendência do comportamento deste algoritmo em CUDA, considerou-se o uso de uma resolução maior que HD1080p. A tendência dos valores de *speed-up* se confirma de acordo com o aumento da resolução do vídeo, onde resultados ainda maiores são apresentados.

6.5 Full Search x Diamond Search

Nesta seção serão realizadas algumas comparações dos resultados em CUDA obtidos entre os algoritmos *Full Search* e *Diamond Search*. Estas comparações incluem uma análise em relação aos valores de *speed-up* atingidos pelos algoritmos propostos neste trabalho em GPU. Além disso, é apresentada uma comparação em relação ao desempenho das duas alternativas (FS e DS) na codificação de vídeo em CUDA em termo de codificação de tempo real.

6.5.1 Speed-up

Os valores de *speed-ups* atingidos pelos dois algoritmos FS e DS, os quais foram apresentados nas Figuras 6.12 e 6.23, respectivamente são devidamente comparados e analisados nesta seção.

Primeiramente, pode-se perceber que os intervalos de *speed-up* alcançados pelo algoritmo FS (entre 30 e 158 – *speed-up*) são maiores que os resultados do DS (entre 26 e 74 – *speed-up*). Esta diferença deve-se ao fato do algoritmo FS ter uma vantagem em relação ao algoritmo DS em relação à computação paralela, ou seja, o FS é totalmente paralelizável não apresentando dependência entre os dados. Enquanto isso, o DS apresenta um comportamento heurístico onde os blocos analisados em cada iteração dependem essencialmente dos resultados das iterações anteriores, acarretando em uma

alta dependência de dados entre os cálculos de cada iteração. Assim, o DS apresenta um menor potencial paralelismo, o que foi verificado pelos piores resultados de *speed-up*.

Outro fator a ser mencionado é a diferença de comportamento em relação aos valores de *speed-up* conforme o crescimento da resolução do vídeo. No algoritmo FS em GPU, os valor de *speed-up,* em geral, aumentam de acordo com o crescimento da resolução do vídeo, porém a partir da área de busca de dimensão *36x36 pixels*, os *speed-ups* das três resoluções tendem a valores semelhantes. No algoritmo DS em GPU, considerando as três resoluções de vídeo, é possível perceber que a implementação em CUDA proposta neste trabalho atinge valores *speed-up* cada vez maiores na medida em que vídeos de maiores resoluções são analisados. A inserção da avaliação de vídeos WQXGA nesse experimento (mostrado na Figura 6.23) prova essa conclusão.

Esta diferença está relacionada com o fato de que na implementação em CUDA do algoritmo FS a alocação do *blocks* (blocos de GPU) é dada de acordo com da área de busca com o objetivo de aumentar o paralelismo (aumento do número de *threads* envolvidas) quando acrescida a dimensão da área de pesquisa. No entanto, quando os recursos computacionais da GPU atingem seu limite de alocação por ciclo de *clock* é necessário o escalonamento das *threads* (realizado pela API CUDA) e, como consequência, o nível do paralelismo diminui. Este fato ocorre nas três resoluções a partir de uma determinada área de busca (na placa de vídeo utilizada *36x36*). Por outro lado, na implementação do algoritmo DS em CUDA a alocação do número de *blocks* considerada é o valor máximo permitido pela GPU utilizada, onde cada *thread* é responsável pela execução da ME de um bloco atual. Desta forma, quanto maior a resolução do vídeo o padrão de *speed-up* é mantido elevado, pois o paralelismo cresce a partir da existência de mais blocos atuais, isto é, um maior número de *threads* na execução do algoritmo.

6.5.2 Codificação em Tempo Real

A Tabela 6.5 apresenta uma comparação referente aos desempenhos das implementações propostas em GPU, FS e DS, para as três resoluções de vídeo, em termos de codificação em tempo real, o qual exige um mínimo de 30 quadros por segundo. É importante notar que o tempo de alocação não foi considerado na contabilização dos tempos de execução, cujos dados e a justificativa são descritas na Seção 6.2 (ver Figura 6.2).

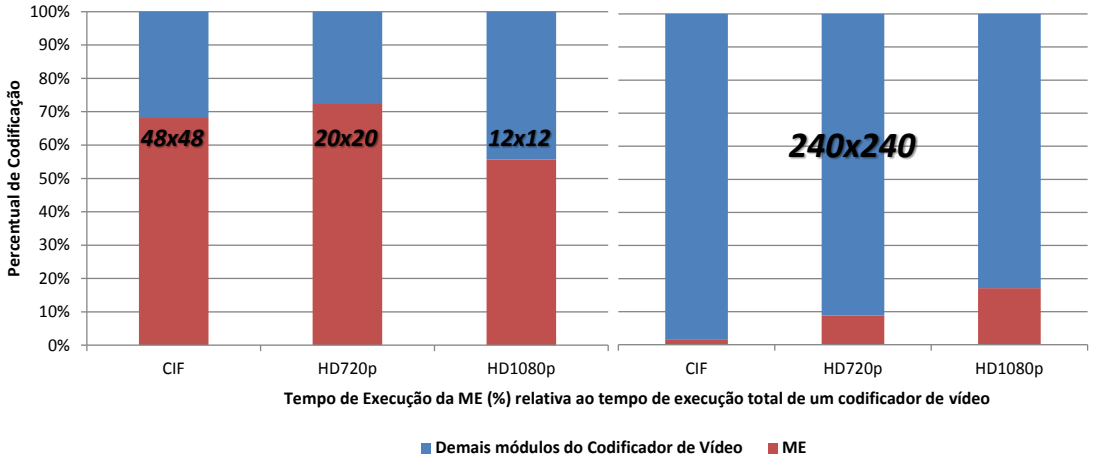
Tabela 6.5: Desempenho da ME em GPU – Taxa de Processamento (@fps).

Algoritmo	**CIF (fps)**	**HD720p (fps)**	**HD1080p (fps)**
DS	1174	340	177
FS	44	41	54
	48x48	*20x20*	*12x12*

A dimensão da área de busca quando para o algoritmo FS impacta diretamente na complexidade de execução da ME, visto que todos os blocos candidatos da área de busca são processados. O algoritmo FS em GPU alcança codificação em tempo real devido ao alto grau de paralelismo explorado pela separação cálculos dos blocos candidatos existentes dentro da área de busca, em diferentes *threads* da GPU, as quais são executadas de maneira paralela. Por sua vez, o desempenho atingido pelo algoritmo

rápido DS ultrapassa o tempo de codificação em tempo real para todas as dimensões de área de busca consideradas e para todas as resoluções de vídeo. Os dados referentes ao algoritmo DS, apresentados na Tabela 6.5, referem-se ao número máximo de iterações necessárias para o algoritmo convergir, isto é, uma área de busca de *240x240 pixels* é considerada.

A Figura 6.24 ilustra uma comparação entre os algoritmos FS (a) e DS (b), mostrando o impacto de ambas as implementações em CUDA dentro das restrições de tempo real. O valor 100% na Figura 6.24 representa o tempo máximo que o codificador pode levar para processar um quadro (*33ms*), de modo a atingir o processamento de 30 quadros por segundo. Dentro deste orçamento de tempo, em vermelho está mostrada a fatia de tempo ocupada pelas implementações FS e DS em CUDA desenvolvidas neste trabalho. Enquanto que a parcela em azul representa o tempo disponível para os demais módulos da codificação. As dimensões de áreas de busca consideradas neste experimento estão de acordo com a Tabela 6.5.



(a) *FS* *(b) DS*

Figura 6.24: Linha do Tempo – Tempo de Execução FS *x* Tempo de Execução DS.

Se a ME em GPU proposta neste trabalho for utilizada em combinação com uma arquitetura convencional (CPU), visando o codificador de vídeo H.264/AVC que é considerado o estado-da-arte neste contexto, a Estimação de Movimento representa os seguintes percentuais do tempo de execução total necessário para atingir processamento em tempo real: ***(a)*** FS: CIF 66% (*48x48*), HD720p 69% (*20x20*) e HD1080p 54% (*12x12*); ***(b)*** DS: CIF 1%, HD720p 8% e HD1080p 16%.

6.6 Comparações com Software de Referência

Nesta seção serão discorridos alguns comparativos entre a Estimação de Movimento em GPU (FS e DS) proposta nesta dissertação e o software de referência do codificador H.264/AVC, JM 18.2, o qual é executado em CPU e foi brevemente introduzido no Capítulo 1. Esta comparação é apresentada na Tabela 6.6, visando o número de quadros por segundo que cada paradigma computacional consegue atingir. O comparativo aborda os dois algoritmos considerados no escopo desde trabalho, FS e DS, sob uma dimensão de área de busca de *32x32 pixels* e as três resoluções de vídeo: CIF, HD720p e HD1080p.

Tabela 6.6: ME GPU *x* ME JM CPU – Taxa de Processamento (@fps).

Algoritmo	ME GPU (fps)			ME JM (CPU) (fps)		
	CIF	**HD720p**	**HD1080p**	**CIF**	**HD720p**	**HD1080p**
DS	1818	341	176	3,2	0,41	0,22
FS	48	11	10	0,9	0,09	0,05

De acordo com os dados apresentados na Tabela 6.6, considerando o algoritmo *Full Search,* a versão da ME em GPU é: *53x*, *122x*, *200x* mais rápida que o software de referência para as resoluções CIF, HD720p e HD1080p, respectivamente. Por outro lado, o algoritmo *Diamond Search* apresenta ganhos de *568x*, *831x*, *800x* para as mesmas resoluções. Observa-se ainda, que os ganhos obtidos em GPU em relação ao software são, em geral, acrescidos de acordo com o aumento da resolução do vídeo, fazendo com que os comportamentos descritos nas Seções 6.3.4 e 6.4.4 sejam justificados.

A Tabela 6.7 apresenta os resultados do uso dos algoritmos em CUDA desenvolvidos neste trabalho em conjunto com a execução do software de referência do H.264/AVC. A ideia é avaliar o uso da GPU como co-processador responsável pela execução ME paralela dentro da codificação completa. Embora este não seja o escopo efetivo do trabalho, esta alternativa é abordada a partir dos dados apresentados na tabela abaixo. A Tabela 6.7, representa uma comparação entre GPU e JM (CPU), com foco no percentual do tempo de execução que a Estimação de Movimento ocupa em relação ao tempo total de codificação de vídeo completa, considerando as demais etapas do codificador.

Tabela 6.7: Percentual da ME em GPU x Percenutal da ME JM.

Resolução de Vídeo	*Full Search*		*Diamond Search*	
	GPU	**JM**	**GPU**	**JM**
CIF	**11%**	98%	**0.17%**	48%
HD720p	**5%**	98%	**0.18%**	60%
HD1080p	**3%**	98%	**0.19%**	61%

Para a obtenção destes dados o tempo total de execução para codificação completa de um quadro foi extraído do JM. Ambos os algoritmos FS e DS foram executados considerando uma área de busca *32x32*. A partir deste dado, o tempo total foi decomposto de modo a obter o tempo de execução da ME independente do tempo de execução dos demais blocos que formam o codificador. Os tempos de execução da ME na GPU e na CPU são mostrados na Tabela 6.7 através do percentual referente ao tempo total de codificação.

Com o uso dos algoritmos em CUDA desenvolvidos neste trabalho, o impacto da etapa de ME dentro do codificador de vídeo é significantemente reduzido quando comprados às implementações originais do software de referência JM, como pode ser observado nos resultados apresentados na Tabela 6.7. Para o FS, com o aumento da resolução do vídeo a ser codificado, maior é o impacto da massiva paralelização do algoritmo proposto em CUDA, visto que mais *threads* serão alocadas e executadas de forma paralela dentro da GPU.

A possibilidade da utilização da GPU como elemento de co-processamento dedicado a executar a ME traz grandes benefícios na aceleração do processo de codificação como um todo, visto que em ambos os algoritmo há reduções significativas: de 98% para 11% no pior caso do algoritmo FS, e de 61% para 0,19% para o algoritmo DS no pior caso.

6.7 Comparações com Trabalhos Relacionados

Por fim, nesta seção serão apresentadas algumas comparações entre os resultados alcançados com a ME em GPU e alguns trabalhos encontrados na literatura, os quais foram brevemente descritos no Capítulo 5.

Todos os trabalhos mencionados consideram a precisão inteira dos *pixels* e o cálculo de SAD como critério de similaridade. As placas gráficas da empresa NVIDIA são utilizadas como alternativa de co-processamento para a ME. As resoluções de vídeo consideradas abordam CIF, HD720p e HD1080p. As comparações, tanto para o algoritmo FS, quanto para o DS, referem-se aos valores de *speed-up* alcançados e a taxa de processamento (@fps), considerando diferentes áreas de busca: *9x9*, *16x16*, *32x32*, *64x64*, *80x80 pixels*. As características e recursos computacionais (número de cores, memória, algoritmo de busca, nível de paralelismo) utilizados nestes trabalhos, em comparação com a ME em GPU proposta, são apresentadas na Tabela 6.8.

Tabela 6.8: Comparativo de recursos computacionais e características dos trabalhos.

Trabalho	**Número de *Cores***	**Número de *Gops***	**Memória**	**Algoritmo de Busca**	**Linguagem/ API**	**Nível de Paralelismo**
(LIN, 2006)	-	66	Textura	FS	OpenCL	*Pixel*
(LEE, 2007)	-	156	Compartilhada	FS	OpenCL	Bloco
(CHEN, 2008)	128	518	Textura	FS	CUDA	*Pixel*
(KUNG, 2008)	128	624	Textura	FS	OpenCL	Bloco
(HUANG, 2009)	128	-	Compartilhada	FS	CUDA	Bloco
(YANG, 2009)	112	-	Compartilhada	FS	CUDA	Bloco
(CHENG, 2010)	192	518	Global/ Compartilhada/ Textura	FS/DS	CUDA	Bloco
(COLIC, 2010)	285	1063	Global/Compartilhada	FS	CUDA	*Pixel*/ Bloco
Este Trabalho	480	1345	Global (Fermi)	FS/DS	CUDA	Bloco

Na Tablela 6.8, pode-se analisar que todas as soluções propostas utilizam placas gráficas da empresa NVIDIA, de diferentes configurações. Onde são salientados os parâmetros relacionados ao número de cores (quantidade unidades funcionais de cada placa de vídeo utilizada) e ao número de *Gops* (Giga Point Operations per Second – indicado pelo fabricante). Se observa também, que a API/Linguagem de programação das placas gráficas varia entre os trabalhos, considerando OpenCL – Cg (do inglês, *Open Computing Language*) (OPENCEL, 2012), (RANDIMA, 2003), (CG, 2012), CUDA e *Shader Model* – Cg (RANDIMA, 2003), (CG, 2012). Estas três linguagens podem ser utilizadas para programação de placas gráficas. A OpenCL, por exemplo, é uma linguagem aberta que pode ser usada em diferentes plataformas e permite a

programação paralela de processadores encontrados em computadores pessoais, servidores ou embarcados. O *Shader Model* é um programa utilizado principalmente no cálculo de efeitos de processamento gráficos em nível de *pipeline*. E por fim, CUDA é uma API de programação mais atual voltada especificamente para programação de dispositivos gráficos da empresa NVIDIA, a qual proporciona um elevado nível de abstração ao programador e comumente utilizada para este fim atualmente. Considerando o histórico da arquitetura CUDA e, salientando que ela foi criada em meados de 2007, pode-se perceber que depois deste ano a maioria dos trabalhos citados na Tabela 6.8, fizeram o uso de CUDA, com excessão de (KUNG, 2008) que optou pela linguagem OpenCL, a qual também é bastante difundida e utilizada neste contexto.

No algoritmo FS GPGPU proposto neste dissertação, cada *thread* é responsável por um bloco de vídeo no quadro de referência (lembrando que um *block* – bloco de GPU – refere-se à área de busca), e por outro lado, no algoritmo DS GPGPU uma *thread* é responsável por um bloco atual. Sendo assim, pode-se dizer que a ME em GPU introduzida neste trabalho, concentra-se no paralelismo dos dados em nível de bloco, onde estas unidades básicas de paralelização referem-se a blocos de vídeo que são processados em simultaneamente. No trabalho de (COLIC, 2010), foram propostos ambos os níveis de paralelismo (bloco e *pixel*), a partir de diferentes configurações/alocações dos recursos da placa de vídeo: uma *thread* para cada cálculo de SAD entre dois *pixels*, cada *thread* trata de um bloco candidato e cada *thread* responsável por um bloco candidato porém multiplas *threads* em um macrobloco. Por sua vez, nos trabalhos introduzidos por (LIN, 2006) e (CHEN, 2008) somente o paralelismo em nível de *pixel* foi utilizado. O trabalho de (CHENG, 2010) propõe um paralelismo em nível de blocos onde uma *thread* trata de 112 *pixels* simultaneamente. Por fim, o trabalho proposto por (YANG, 2009), também utiliza o paralelismo em nível de blocos, onde cada *thread* busca o melhor vetor de movimento de um macrobloco (blocos de vídeo compostos de 16x16 *pixels*).

O nível de paralelismo, torna-se uma característica importante pois está diretamente relacionada com o desempenho da aplicação, em termos de tempo de execução. Os trabalhos que paralelizam a Estimação de Movimento em nível de *pixel* necessitam de uma maior sincronização mais frequente entre as *threads* do que as aplicações que utilizam nível de blocos. Além disso, devem garantir uma eficiência na concorrência dos recursos entre as *threads*, de modo a garantir que o acesso de diversas *threads* simultaneamente não modifique o valor de um registrador erroneamente. Em especial, nas versões da ME propostas por (CHEN, 2008) e (COLIC, 2010), os quais utilizam a arquitetura CUDA em suas implementações, o nível de paralelismo abordada acarreta em um barreira adicional, além das já mencionadas, onde os tamanhos de *blocks* alocados devem que ter o número mínimo de *threads* necessárias para comportar a execução de todo quadro atual paralela, ou seja, considerando o estado-da-arte em GPUs da NVIDIA, para esta condição vir a ser satisfeita e eficiente se considerariam apenas baixas resoluções de vídeo nos experimentos (um número razoalvemente baixo de *pixels* se comparado com vídeos de alta definição).

Em resumo, todos os trabalhos considerados tem como base o codificador de vídeo H.264/AVC em seus desenvolvimentos, a implementação do algoritmo FS, a utilização de maiores resoluções de vídeo nos experimentos apresentados (HD720p e HD1080p), e pequenas dimensões de ára de busca. Considerando a dependência de dados imposta por algoritmos sub-ótimos, poucos algoritmos encontrados na literatura apresentam soluções voltadas para algortimos rápidos para a ME. Além disso, os trabalhos não só

tem como foco a aceleração da Estimação de Movimento, como também de manter a qualidade do vídeo digital e prover diferentes caracteristicas inerentes ao padão H.264/AVC, como por exemplo, tamanho de bloco variável e módulos posteriores a ME – codificação do vetor de movimento. Assim, na medida do possível, os resultados de *speed-up* referentes ao algoritmo FS são apresentados na Tabela 6.9 em comparação com os obtidos neste trabalho, considerando diferentes resoluções de vídeo e dimensões de área de busca.

Os resultados apresentados na Tabela 6.9, mostram que a ME em GPU proposta nesta dissertação par ao algoritmo FS, alcança os maiores valores de *speed-up* do que, praticamente, todos os trabalhos analisados para diferentes resoluções de vídeo e dimensões de área de busca.

Tabela 6.9: Speed-up FS - ME em GPU Proposta x Trabalhos Relacionados.

Trabalho	**CIF**					**HD720p**		**HD1080p**		
	16x16	***32x32***	***48x48***	***64x64***	***80x80***	***16x16***	***32x32***	***9x9***	***16x16***	***32x32***
(LIN, 2006)	1,79	2.18	1,87	1,45	1,29	**-**	**-**	**-**	**-**	**-**
(LEE, 2007)	12,08	26,76	-	-	-	-	-	-	-	-
(CHEN, 2008)	-	10,38	-	-	-	-	-	-	-	-
(KUNG, 2008)	9,35	10,14	-	-	-	28,19	31,43	-	40,4	45,55
(CHENG, 2010)	-	-	-	-	-	-	-	8,74	-	-
(COLIC, 2010)	135,26	-	-	-	-	-	-	-	279,03	-
Este Trabalho	**35,12**	**35,75**	**80,17**	**85,98**	**88,55**	**116,04**	**77,08**	**96,36**	**124,8**	**158,29**

Na resolução CIF os trabalhos que atingem os maiores valores de *speed-up* são os de (LEE, 2007) para áreas de busca de dimensões *16x16* e *32x32 pixels* e, o trabalho de (COLIC, 2010) para área de *16x16 pixels*. Primeiramente, quando comparado com (LEE, 2007), a ME em GPU FS proposta atinge ganhos de *2,9x* e *1,3x* em termos de *speed-up* para as dimensões de área de busca de *16x16* e *32x32 pixels*, respectivamente. Estes resultados refletem a eficiência deste trabalho em relação ao mapeamento do algoritmo FS para os recursos computacionais da arquitetura CUDA (*threads*, *blocks* e *grid*). Salienta-se que o ME proposta por (LEE, 2007) implementa diferentes configurações da ME, como por exemplo, considerando mais de um quadro de referência, precisão dos dados fracionária, entre outras. Estas características demonstram que o trabalho não visa somente a acelaração da ME em GPU, bem como dar suporte a diferentes características do codificador H.264/VC. Além disso, a dimensão dos blocos de vídeo considerada neste trabalho é de *4x4 pixels*, enquanto a dimensão adotada em (LEE, 2007) é referente a um macrobloco, ou seja, *16x16 pixels*. Este fator faz com que a qualidade dos resultados da ME proposta neste trabalho seja superior, visto que o uso de blocos *4x4* permite comparações mais precisas do que uma

implementação que faz o uso de um bloco de uma maior dimensão de bloco. Por sua vez, em relação ao trabalho de (COLIC, 2010), pode-se perceber que os resultados apresentados são superiores a ME GPGPU proposta neste dissertação. No entanto, os resultados descritos no trabalho de (COLIC, 2010) não consideram todo o algoritmo FS, apenas aos cálculos de valores de SAD são contabilizados, desconsiderando a etapa responsável pela busca do menor elemento para cada bloco atual. Segundo o autor, o objetivo do trabalho é apenas mostrar que a GPU é uma boa alternativa para codificação de vídeo e, para este fim, a ilustração de apenas uma das etapas já é o suficiente para representar esta idéia. Em relação aos demais trabalhos, a ME em GPU FS mantem uma superioridade elevada para as diferentes áreas de busca consideradas. Em relação ao trabalho de (LIN, 2006), concentram-se os maiores ganhos apresentados por este algoritmo para resolução CIF, atingindo: *19,6x*, *16,39x*, *42,87x* e *64,64x*, para as dimensões de área de busca de *16x16*, *32x32*, *48x48*, *64x64* e *80x80 pixels*, respectivamente. Um dos fatores que justificam estes ganhos está relacionado com o nível de paralelismo adotado, onde este trabalho utiliza o nível de bloco (um *block* é responsável por um bloco atual) e (LIN, 2006) considera o nível de *pixel*, cujas as desvantagens foram mencionadas anteriormente. Esta mesma justificativa é levada em consideração quando comparado a ME proposta neste dissertação com o trabalho de (CHEN, 2008), onde tem-se um ganho de até *3,5x*, quando da utilização de um paralelismo em nível de blocos. Por sua vez, a versão da ME apresentada por (KUNG, 2008) trata do paralelismo em nível de blocos é. Contudo, o autor considera uma etapa posterior a ME. Esta etapa consiste na codificação de vetores de movimento que é realizada a partir de uma mediana entre três vetores de movimento de blocos vizinhos. Assim, para a codificação de um vetor de movimento tem-se uma dependência de dados, pois para esta etapa ser efetivamente realizada são necessários vetores imediatamente adjacentes (anteriores). Desta forma, em uma comparação entre os resultados apresentados por (KUNG, 2008) e o trabalho prospoto é possível visualizar ganhos de até *3,8x* e *3,5x* para as áreas de busca de *16x16* e *32x32*, respectivamente.

Dos trabalhos apresentados na Tabela 6.9, o trabalho de (KUNG, 2008) é o único que fornece resultados para resolução de vídeo HD720p. Assim, pode-se perceber a superioridade imposta pela ME em GPU proposta neste dissertação. Os dados apresentados representam ganhos de *4,11x* (área de busca *16x16)* e *2,45x* (área de busca *32x32)*, quando comparado com os resultados atingidos por (KUNG, 2008). Apesar deste trabalho considerar o nível de paralelismo de blocos, a justificativa mencionada para os ganhos obtidos na comparação referente a resolução CIF é a mesma mencionada anteriormente para resolução CIF, pois a implementação da ME é a mesma (não considerando a codificação de vetores de movimento) variando apenas os parâmetros dos experimentos, neste caso a resolução do vídeo.

Por fim, três trabalhos apresentam resultados para a resolução HD1080p. Onde o trabalho de (COLIC, 2010) atinge maior valor de *speed-up* para área de busca *16x16*, (KUNG, 2008) para *32x32 pixels* e (CHENG, 2010) para área de busca de *9x9*. Os resultados apresentados por (COLIC, 2010), como mencionado anterior mente, não contabilizam a execução total de ME do algoritmo FS, apenas o cálculo dos valores de SAD são considerados. Por esta razão, o trabalho de (COLIC, 2010) apresenta ganhos em relação a ME em GPU FS proposta nesta dissertação. Na comparação, em relação a área de busca *32x32* pixles, entre o trabalho de (KUNG, 2008) e os resultados atingidos por este trabalho tem-se uma superioridade imposta pela ME apresentada nesta dissertação, apresentando ganho de até *3,48x* em termos de *speed-up*. Da mesma forma,

como os demais parâmetros, a justificativa mencionada para as resoluções CIF e HD720p (referente a comparação entre este trabalho e (KUNG, 2008)) é válida para esta resolução, onde o trabalho de (KUNG, 2008) concentra-se na codifição dos vetores de movimento além da aceleração da ME. Para área de busca *9x9 pixels*, o trabalho de (CHENG, 2010) apresenta resultados para resolução de alta definição HD1080p. A superiodade da ME introduzida nesta dissertação refere-se a um ganho de *11x* quando comparado com (CHENG, 2010). O comportamento apresentado por (CHENG, 2010) é inferior a esta ME em GPU proposta pois este trabalho também implementa a etapa de codificação de vetores de movimento, a qual não é o foco deste trabalho. No entanto, o custo desta codificação dos vetores não é amortizando de forma efetiva no tempo de execução total atingido, desta forma, este etapa é considerada o gargalo do trabalho proposto por (CHENG, 2010).

Para finalizar os comparativos entre a ME proposta e os trabalhos relacionados que implementaram o algoritmo de busca FS em GPU, na Tabela 6.10 são apresentados dados relacionados à taxa de processamento em que os vídeos podem ser codificados. A Tabela abaixo apresenta dados para diferentes trabalhos relacionados, variando a resolução do vídeo e as dimensões de áreas de busca, em comparação com este trabalho.

Tabela 6.10: Taxa de Processamento (@fps) FS – CIF e HD1080p: ME em GPU Proposta *x* Trabalhos Relacionados.

Trabalho	**CIF (fps)**					**HD1080p (fps)**
	16x16	***32x32***	***48x48***	***64x64***	***80x80***	***9x9***
(LIN, 2006)	11,57	3,70	1,49	0,67	0,40	**-**
(LEE, 2007)	13,48	8,89	-	-	-	-
(CHEN, 2008)	-	31,54	-	-	-	-
(YANG, 2009)	-	3,49	-	-	-	-
(CHENG, 2010)	-	-	-	-	-	4,73
Este Trabalho	**219,55**	**47,55**	**44,50**	**26,06**	**16,86**	**688,75**

Analisando os dados apresentados na Tabela 6.10, pode-se perceber que em todos os parâmetros avaliados, a ME em GPU FS proposta neste trabalho atinge um número maior em relação a taxa de processamento dos dados. Além de ser superior em relação a todos os trabalhos encontrados na literatura que forneceram este dado, a ME FS em CUDA atinge codificação em tempo real (30 quadros por segundo) nas duas resoluções de vídeo e para todas as dimensões de áreas de busca abordadas.

Na resolução CIF, os ganhos máximos neste aspecto representam: ***(i)*** *17x* em relação a área de busca *16x16* quando comparado com o trabalho proposto por (LEE, 2007), o qual apresenta os melhores resultados nesta dimensão área de busca, ***(ii)*** *2x* em relação a área de busca *32x32 pixels* quando comparado com (CHEN, 2008), o qual é trabalho mais rápido (em fps) aos demais considerados neste parâmetro; ***(iii)*** *30x*, *43x* e *40x* mais rápidos que (LIN, 2006) nas dimensões de áreas de busca, *48x48*, *64x64* e *80x80*, respectivamente. O trabalho de (LIN, 2006) é o único, dos encontrados na literatura, que apresenta resultados em um maior número de variação na dimensão de área de busca. Apesar a ME em CUDA para o FS proposta neste trabalho considerar ganhos significativos em relação ao (LIN, 2006), o qual utiliza o nível de paralelismo em *pixels*, em nenhuma das dimensões de área de busca mencionadas a codificação em tempo real

foi atingida, salientando a exaustividade do algoritmo FS e do aumento da complexidade computacional de forma proporcional ao crescimento da área de pesquisa.

Na resolução HD1080p, apenas o trabalho de (CHENG, 2010) viabilizou a comparação neste aspecto. No entanto, considerando que a área de busca avaliada por (CHENG, 2010) é pequena (*9x9 pixels*), a ME FS GPGPU proposta nesta dissertação atingiu um ganho de *172x* em relação aos resultados apresentados por (CHENG, 2010) referente à codificação máxima de quadros por segundo. Assim, a ME CUDA FS proposta neste trabalho, considerando o algoritmo FS e uma área de busca *9x9 pixels*, atinge uma codificação de vídeo em tempo real.

A Tabela 6.11 apresenta um comparativo referente aos valores de *speed-up* e a taxa de processamento entre a versão do algoritmo DS em GPU proposto nesta dissertação e os trabalhos encontrados na literatura.

Tabela 6.11: Speed-up e Taxa de Processamento (@fps) DS – HD1080p: ME em GPU Proposta *x* Trabalhos Relacionados.

Trabalho	***HD1080p***	
	9x9	
	Speed-up	***fps***
(CHENG, 2010)	6,86	15,15
Este Trabalho	**49,36**	**177,47**

Poucos trabalhos encontrados na literatura consideram o uso de algoritmos rápidos na paralelização da ME, pois, tipicamente, estes algoritmos sub-ótimos introduzem dependência entre os dados, cuja característica não favorece um desempenho satisfatório em termos de programação paralela. Mais de um trabalho foi neste cenário foi descrito nesta dissertação, no entanto eles não permitiram que fosse realizada uma comparação justa a partir dos poucos dados apresentados na descrição dos resultados. Desta forma, apenas o trabalho de (CHENG, 2010) viabilizou uma comparação do algoritmo DS com os resultados atingidos e apresentados nesta dissertação. A comparação concentra-se uma resolução de alta definição e uma área de busca de *9x9 pixels*, tanto em termos de *speed-up*, quanto em taxa de processamento.

Em termos de *speed-up*, observando a Tabela 6.11 tem-se uma significativa superioridade imposta pela versão CUDA proposta neste trabalho. A ME DS em GPU introduzida nesta dissertação alcança um ganho de *7x* em relação à versão do algoritmo DS descrito em (CHENG, 2010). Por sua vez, em termos de taxa de processamento para a resolução de alta definição HD1080p e área de busca *9x9 pixels* a ME CUDA DS apresenta um ganho de até *12x,* quando comparado com o trabalho de (CHENG, 2010). Além disso, é possível perceber que é possível alcançar codificação em tempo real considerando o algoritmo DS em vídeos de alta definição.

Em resumo, em ambos os aspectos a melhora do trabalho proposto, em tempo de execução, *speed-up* e taxa de processamento é significativa. Esta superioridade é justificada pela mesma razão mencionada na comparação dos resultados referentes ao algoritmo FS. O trabalho de (CHENG, 2010) além da aceleração da ME em GPU, concentra-se na codificação de vetores de movimento já mencionada. Desta forma, tem-

se uma dependência de dados que é acrescida no custo da depêndencia padrão inerente ao algoritmo DS. Para contornar estes problemas o trabalho proposto por (CHENG, 2010) propôs um mecanismo de detecção de depêndencias. No entanto, esta característica comprometeu de forma razoável o desemepenho da aplicação, em tempo de execução total.

6.7.1 Normalização dos Resultados

Analisando a Tabela 6.8 é possível observar que a placa de vídeo utilizada na execução da ME proposta neste trabalho tem maior capacidade de processamento em relação aos dispositivos utilizados nos trabalhos relacionados. Esta superioridade é proveniente de um maior número de *cores*/núcleos que a placa gráfica disponibiliza. Sendo assim, a GPU utilizada nos experimentos que compõe os resultados desta dissertação é: *8.62x* mais rápida do que a placa de vídeo utilizada por (LEE, 2007), *20.37x* quando comparada com o dispositivo gráfico utilizado por (LIN, 2006), *2.59x* em relação aos trabalhos de (CHEN, 2008) e (CHENG, 2010), *2.15x* considerando a GPU utilizada por (KUNG, 2009) e *1.26x* em relação à (COLIC, 2010).

Por esta razão, os resultados apresentados na literatura foram normalizados levando em consideração a capacidade de processamento da GPU utilizada. O objetivo desta análise é proporcionar uma comparação justa e assim, justificar os ganhos atingidos pela Estimação de Movimento desenvolvida neste trabalho. Para realizar esta normalização, foi adotada uma métrica a qual considera o número de operações que cada dispositivo suporta, em termos de *Gops* (do inglês, *Giga Floating Point Operations per Second*). Assim, a métrica utilizada para este fim foi:

$$N = \frac{Speed - up}{Gops} \tag{3}$$

A equação (3) acima define a métrica utilizada para normalização dos resultados, onde *speed-up* é dada em (2) e o número de *Gops* é apresentado na Tabela 6.8, conforme os dados fornecidos pelo fabricante. Sendo assim, esta normalização possibilita que o desempenho atingido (em termos de *speed-up*) pelos autores esteja de acordo com a placa de vídeo utilizada.

Primeiramente, são apresentados os dados referentes à implementação do algoritmo *Full Search* na Tabela 6.12. Estes dados ilustram os ganhos atingidos por este trabalho em relação à literatura. A comparação abrange diferentes resoluções e dimensões de área de busca. Na tabela abaixo se pode visualizar que a versão do algoritmo FS proposta neste trabalho é superior à maioria dos trabalhos relacionados (de acordo com a equação 3), porém algumas perdas também podem ser observadas.

Considerando a resolução CIF, o algoritmo proposto atinge ganhos em relação à (LIN, 2006), (CHEN, 2008) e (KUNG, 2008). Em especial, na comparação com (LIN, 2006) observa-se que os ganhos positivos são obtidos a partir da dimensão de área de busca *48x48*. Este fato mostra que a versão proposta nesta dissertação do algoritmo FS apresenta um melhor desempenho quando se considera um maior volume de dados em processamento, fazendo com que o custo de comunicação seja compensado. Nas dimensões de área de busca *16x16* e *32x32*, o volume de dados envolvido é menor e o custo de comunicação entre os dispositivos (CPU e GPU, vice versa) é impactante no tempo total de execução paralela. Além disso, a versão proposta apresenta perdas para (LEE, 2007) e para (COLIC, 2010). As perdas apresentadas em relação à (LEE, 2007) ocorrem porque o trabalho deste autor incorpora uma exploração de hierarquia de

memória na GPU (não realizada nesta dissertação e proposta como trabalho futuro), assim o custo de comunicação entre os dispositivos é tratado e o tempo de execução é consideravelmente minimizado fazendo com que apresente ganhos em relação ao FS implementado nesta dissertação. Por sua vez, as perdas relacionadas ao trabalho de (COLIC, 2010) se devem ao fato de que os resultados apresentados pelo autor não considera a etapa de busca pelos menores valores de SAD (etapa mais custosa computacionalmente do trabalho) no tempo de execução, contabilizando apenas o processamento do cálculo de similaridade. Desta forma, o *speed-up* apresentado por (COLIC, 2010) é maior do que os dados atingidos por este trabalho.

A normalização dos resultados para resolução HD720p quando da comparação do trabalho proposto para o algoritmo FS com os trabalhos relacionados também é apresentada na Tabela 6.12. Assim, os ganhos deste trabalho são de *2x* e *1,2x* para áreas de busca *16x16* e 32x32 respectivamente, quando comparado com o trabalho de (KUNG, 2009). Estes ganhos são provenientes da técnica de paralelização adotada por este trabalho.

Por sua vez, a Tabela 6.12 introduz os ganhos normalizados considerando a resolução HD1080p onde os ganhos são de *4x*, quando comparado com (CHENG, 2010). A perda apresentada em relação ao trabalho de (COLIC, 2010) possui a mesma justificativa apresentada para a resolução CIF descrita anteriormente.

Tabela 6.12: Normalização dos Resultados - FS: ME em GPU Proposta *x* Trabalhos Relacionados.

Trabalho	**CIF**					**HD720p**		**HD1080p**		
	16x16	*32x32*	*48x48*	*64x64*	*80x80*	*16x16*	*32x32*	*9x9*	*16x16*	*32x32*
(LIN, 2006)	*-1x*	*-1,2x*	*2x*	*3x*	*69x*	-	-	-	-	-
(LEE, 2007)	*-3x*	*-6x*	-	-	-	-	-	-	-	-
(CHEN, 2008)	-	*1,3x*	-	-	-	-	-	-	-	-
(KUNG, 2008)	*2x*	*2x*	-	-	-	*2x*	*1,2x*	-	*1,5x*	*2x*
(CHENG, 2010)	-	-	-	-	-	-	-	*4x*	-	-
(COLIC, 2010)	*-1,6x*	-	-	-	-	-	-	-	*-3x*	-

Conforme mencionado anteriormente, são poucos os trabalhos que apresentam implementações em GPU de algoritmos rápidos para Estimação de Movimento. No entanto, a comparação da versão do algoritmo *Diamond Search* proposta é realizada apenas com (CHENG, 2010). Considerando a resolução HD1080p, a versão ME DS proposta apresenta um ganho de *3x* (área de busca *9x9*) em relação ao trabalho de (CHENG, 2010), de acordo com a normalização proposta (veja equação 3).

6.8 Considerações Finais sobre o Capítulo

Os experimentos apresentados avaliam aspectos essenciais de um ambiente de programação paralela e codificação de vídeo tais como: tempo de execução, *speed-up*, comunicação e taxa de processamento.

Os resultados obtidos e analisados demonstram vantagens significativas da ME em GPU proposta neste trabalho, considerando a arquitetura CUDA, em relação aos demais paradigmas comparados para ambos os algoritmos (FS e DS). No entanto, algumas carências nas implementações propostas podem ser ressaltadas. No algoritmo FS, salienta-se a limitação do paralelismo quando a dimensão da área de busca cresce. Esta limitação ocorre porque o tamanho do *block* (bloco de GPU) é alocado de acordo com o tamanho da área de pesquisa. Desta forma, este requisito está diretamente relacionado com a quantidade de recursos que a placa de vídeo utilizada dispõe. Contudo, esta carência pode ser dissolvida ou minimizada a partir do uso de uma placa gráfica composta por um número maior de CUDA Cores do que a GPU utilizada nestes testes. Em relação ao algoritmo DS em GPU, por ser um algoritmo rápido onde os tempos de execução são consideravelmente menores do que os apresentados pelo FS, a comunicação de memória entre os dispositivos (CPU e GPU, e vice versa) torna-se um aspecto relevante e impactante nos resultados obtidos (veja Figura 6.2) em termos de desempenho. Desta forma, uma estratégia de melhor exploração dos níveis de memória da GPU pode ser realizada de modo a minimizar e até mesmo, solucionar esta carência.

Todas as versões da Estimação de Movimento em GPU para os dois algoritmos FS e DS demonstram ganhos significativos com o uso da arquitetura CUDA. A algoritmo FS tem como vantagem o aumento do paralelismo a medida que a área de busca cresce (até que seja alocado o número de *threads* por ciclo de *clock* permitido pela GPU utilizada). Por sua vez, o algoritmo DS se beneficia do paralelismo em nível de blocos, onde cada *thread* trata de um bloco atual e assim, quanto maior o número de blocos atuais envolvidos (maior resolução do vídeo) um maior nível de paralelismo é considerado.

Em resumo, os ganhos das versões ME GPGPU crescem à medida que a dimensão da área de busca em conjunto com a resolução de vídeo aumenta. Em destaque para a comparação da GPU com MPI (paradigma que mais se aproximou dos resultados de CUDA) onde foram utilizados 64 processos. Neste cenário, pode-se observar que o desempenho da versão distribuída satura a partir da alocação de 64 processos não apresentando ganhos significativos no aumento do número de processos. Por sua vez, este comportamento não é notado nas soluções em CUDA apresentadas, onde a tendência é que os ganhos possam vir a ser beneficiados a partir de um possível aumento no número de CUDA Cores (núcleos de processamento) em experimentos futuros.

Além disso, os resultados em relação à taxa de processamento mostram que o uso de GPU é uma boa alternativa para atingir codificação de vídeo em tempo real. No algoritmo FS, por ser exaustivo, somente considerando áreas de buscas pequenas é possível atingir este dado com resolução de vídeo de alta definição HD1080p. Por outro lado, o algoritmo rápido DS viabiliza a codificação de vídeo em tempo real considerando uma área de busca de dimensão *240x240 pixels* para todas as resoluções de vídeo consideradas no escopo deste trabalho.

A partir destas considerações, foi possível analisar que houve uma redução significativa da Estimação de Movimento, em especial, quando comparado com os custos da ME no software de referência JM, o qual é executado em CPU. Sendo assim,

apesar deste não ser o escopo do trabalho, tem-se a apresentação de uma possibilidade de utilização do software de referência JM em conjunto com uma placa gráfica para que somente a Estimação de Movimento (módulo mais custoso de um codificador de vídeo) fosse executada na GPU, obtendo assim uma melhora significativa em termos de desempenho visando uma codificação de vídeo completa.

Estes resultados foram consolidados a partir das comparações da ME em GPU proposta com trabalhos relacionados para diferentes áreas de busca e resoluções de vídeo. Estes ganhos atingidos foram apresentados e justificados a partir da normalização dos dados realizada, a qual considera a capacidade de processamento de cada GPU e o desempenho atingido. Todos os trabalhos relacionados apresentados nesta dissertação apresentam contribuições relevantes para a comunidade neste contexto, porém são poucos os que apresentam algoritmos rápidos e consideram resoluções maiores que CIF em seus experimentos. Em resumo, como a ME em GPU desenvolvida concentrou-se apenas em avaliação de desempenho, muitas características e estratégias que envolvem qualidade de imagem e codificação de módulos posteriores à Estimação de Movimento não foram desenvolvidas. Assim, as comparações apresentadas ressaltam e justificam ganhos consideráveis em relação aos trabalhos encontrados na literatura tanto em valores de *speed-up*, quanto em quantidade de codificação de vídeo em quadros por segundo. Portanto, conclui-se que o mapeamento dos algoritmos *Full Serach* e *Diamond Search* para execução em placas gráficas propiciam ganhos significativos.

7 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho apresentou diferentes versões da Estimação de Movimento em GPU utilizando a arquitetura CUDA, considerando o algoritmo de busca *Full Search* e o *Diamond Search*. Os experimentos variaram as dimensões de área de busca e as resoluções de vídeo. Para análise de desempenho os resultados obtidos foram comparados com diferentes paradigmas computacionais, *multi-core* e distribuído, trabalhos relacionados e *software* de referência. O requisito inicial, de aumentar/acelerar o desempenho da Estimação de Movimento a partir da paralelização em GPU a fim de codificar vídeos de alta definição em tempo real (mais especificamente vídeos HD1080p a 30 quadros/s) foi atingido. Desta forma, os objetivos do trabalho foram alcançados.

A Estimação de Movimento em GPU proposta para o algoritmo FS é até *80x*, *116x*, e *158x*, mais rápida que a versão serial para as resoluções CIF, HD720p e HD1080p, respectivamente. Se comparado com o software de referência JM apresenta reduções, em termos de desempenho, de até 98,38% para a resolução de vídeo HD1080p (resolução que requer o maior custo computacional). E em relação aos trabalhos relacionados representa ganhos de até *69x* se comparado com (LIN, 2006) pra resolução CIF, *2x* se comparado com (KUNG, 2008) considerando a resolução HD720p e *4x* para HD1080p comparado com (CHENG, 2010).

Por outro lado, a ME em GPU que propõe a implementação do algoritmo rápido DS apresenta melhorias em tempo de execução de até *39x* para resolução CIF, *42x* para resolução HD720p e *62x* para HD1080p em relação a respectiva versão serial. A redução imposta pela ME em GPU DS quando comparado com o software de referência é de até 61% para a maior resolução de vídeo considerada no escopo deste trabalho, HD1080p. O ganho obtido em relação a trabalhos da literatura é de *3x* em relação ao trabalho proposto por (CHENG, 2010).

Portanto, o desempenho apresentado pelas versões da ME propostas mostra que o uso de GPUs é uma boa alternativa para aceleração do processo de Estimação de Movimento. Os resultados provêm uma codificação de vídeo em tempo real, tanto para o algoritmo exaustivo, quanto para o algoritmo rápido, para resoluções de vídeo de alta definição.

7.1 Contribuições do Trabalho

A principal contribuição deste trabalho foi mostrar a redução significativa da Estimação de Movimento quando paralelizada e executada em GPU em relação a diferentes alternativas computacionais. As outras contribuições a partir desta dissertação foram:

- Apresentar características relacionadas à codificação de vídeo, arquitetura CUDA e programação de algoritmos em GPU;
- Constatar as vantagens de utilização do uso de placas gráficas na paralelização da codificação de vídeo;
- Incorporar os conceitos de programação distribuída e *multi-core* no desenvolvimento do trabalho;
- Avaliar a Estimação de Movimento paralela implementada frente a diferentes comparações com outros paradigmas computacionais, software de referência e trabalhos relacionados encontrados na literatura;
- Disponibilizar ao grupo de pesquisa uma nova abordagem a respeito da codificação de vídeo através da paralelização do módulo mais exaustivo da codificação total de um vídeo digital.

Dois artigos completos, um resumo, um resumo expandido e um artigo para *Journal* foram escritos para eventos nacionais, regionais e internacionais durante o desenvolvimento deste trabalho. Foi submetido e aprovado um artigo para o evento nacional da área SBAC-PAD 2011 (do inglês, *23rd International Symposium on Computer Architecture and Hgh Performance Computing*) (MONTEIRO, 2011c). Além desse evento, resumos foram publicados em eventos como SIM 2011 (XXVI Simpósio de Microeletrônica) (MONTEIRO, 2011b) e DEPCP 2011 (do inglês, *Designing for Embedded Parallel Computing Platforms: Architectures, Design Tools and Applications*) (MONTEIRO, 2011a), o qual ocorreu junto ao DATE 2011 (do inglês, *Design, Automation & Test in Europe*). Um artigo completo foi submetido (em Janeiro de 2012) para o ICIP 2012 (do inglês, *IEEE International Conference on Image Processing*) e, um artigo para a edição especial do SBAC-PAD, o *Journal* IJPP (do inglês, *International Journal of Parallel Programming*) também foi submetido (em Dezembro de 2011).

7.2 Trabalhos Futuros

Existem diversas possibilidades de trabalhos futuros associados a esta dissertação, tanto no desenvolvimento algorítmico, quanto na paralelização dos mesmos. Assim, a continuidade do trabalho prevê o desenvolvimento de novos algoritmos de busca para a Estimação de Movimento, tais como, algoritmos rápidos e especulativos. Além disso, serão consideradas algumas variações de codificação de vídeo, tais como o suporte a codificação de vídeo considerando múltiplas vistas e, além disso, considerando o futuro padrão de codificador de vídeo, o HEVC (*High Efficiency Video Coding*), também conhecido como H.265 (SULIVAN, 2010), (JCT-VT, 2011), (DONG,2011).

Não obstante, pretende-se ampliar as avaliações de desempenho e viabilizar diferentes soluções paralelas. Desta forma, espera-se que novos experimentos em outras plataformas tais como programação *multi-core* em *clusters*, visando a utilização de OpenMP em um ambiente distribuído (MPI). Além disso, será avaliada a possibilidade de melhoria da comunicação entre os dispositivos CPU e GPU (e vice versa), a qual é um fator impactante em termos de desempenho, a partir da exploração das hierarquias de memória inerentes a arquitetura FERMI e na avalição de uso de outras tecnologias e dispositivos. No grupo de pesquisa, este trabalho será o ponto de partida para a paralelização de outros módulos inerentes ao codificador de vídeo.

REFERÊNCIAS

AGOSTINI, L. V., **Desenvolvimento de Arquiteturas de Alta Performance Dedicadas à Compressão de Vídeo Segundo o Padrão H.264**. Tese (Doutorado) - Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, RS, 2007.

BANH, X.; TAN, Y. Adaptive dual-cross Search algorithm for Block-matching motion estimation. **IEEE Transactions on Consumer Electronics**, [S.l.], v. 50, n. 2, p. 766-775, 2004.

BHASKARAN, V.; KONSTANTINIDES, K. **Image and Video Compression Standards**:Algorithms and Architectures. 2. Boston: Kluwer Acad. Publishers, 1999.

CHEN, W.; HANG, H. H.264/AVC motion estimation implementation on Compute Unified Device Architecture (CUDA). In: IEEE International Conference on Multimedia and Expo, Hannover, Germany. **Proceedings…** [Hannover: IEEE], 2008, p. 697-700.

CHENG, Y.; CHEN, Z; CHANG, P. An H.264 Spatio- Temporal Hierarchical Fast Motion Estimation Algorithm for High-Definition Video. In: IEEE International Symposium on Circuits and Systems**,** Taipei, Taiwan. **Proceedings...** [Tapei: ISCAS], 2009, p. 880-883.

CHENG, R.; YANG, E.; LIU, T. Speeding up motion estimation algorithms on CUDA technology. In: IEEE Asia Pacific Conference on Postgraduate Research in Microelectronics and Electronics (PrimeAsia), Shangai, China. **Proceedings...** [Shangai: IEEE], 2010, p. 93-96.

COLIC, A.; KALVA, H.; FURHT, B. Exploring NVIDIA-CUDA for video coding. In: ACM SIGMM CONFERENCE ON MULTIMEDIA SYSTEMS, 1., MMSys'10, [S.l.]. **Proceedings...** ACM: New York, USA, 2010, p. 13-22.

DONG, J. LIU, Y. **H265.net Witness the development of H.265**. Disponível em: <http://www.h265.net/>. Acesso em: março 2012.

FLYNN, M. Some Computer Organizations and Their Effectiveness. In: **IEEE Transactions Computer**. [S.l.]. Proceedings… [S.l: IEEE], 1972, C-21: 948.

GHANBARI, M. **Standard Codecs: Image Compression to Advanced Video Coding**. United Kingdom: The Institute of electrical Engineers, 2003.

GONZALEZ, R.; Woods, R. **Processamento de Imagens Digitais**. São Paulo: Edgard Blücher, 2003.

GPGPU – **General Purpose Computation on Graphics Hardware**. Disponível em < http://gpgpu.org/>. Acesso em: março 2012.

GPPD UFRGS – **Parallel and Dristributed Processing Group – GPPD**. Disponível em: <http://gppd.inf.ufrgs.br/new/>. Acesso em: março 2012.

GRID5000 – **Grid5000: Home - Grid'5000**. Disponível em: <www.grid5000.fr/>. Acesso em: março 2012.

HUANG, Y.; SHEN, Y.; Wu, J. Scalable computation for spatially scalable video coding using NVIDIA CUDA and multi-core CPU. In: ACM International Conference on Multimedia (MM '09), New York, NY, USA. **Proceedings...** [New York: ACM], 2010, p. 361-370.

ITU – INTERNATIONAL TELECOMMUNICATION UNION. **ITU-T Recommendation H.264/AVC (03/10)**: advanced video coding for generic audiovisual services. [S.l.], 2010.

ITU – INTERNATIONAL TELECOMMUNICATION UNION. **ITU-T Home**. Disponível em: <www.itu.int/ITU-T/>. Acesso em: março 2012.

ITU – INTERNATIONAL TELECOMMUNICATION UNION. **ITU-T Recommendation H.261**: Video Codec for Audiovisual Services at p×64 kbit/s, Version 1, ITU-T, 1990.

JCT-VT – **JOINT COLLABORATIVE TEAM ON VIDEO CODING: Documentos das reuniões do grupo**. Disponível em: <http://wftp3.itu.int/av-arch/jctvc-site/>. Acesso em: março 2012.

JING, X.; CHAU, L. An efficient three-step search algorithm for Block motion estimation. **IEEE Transactions on Multimedia**, [S.l.], v.6, n.3, p. 35-438, 2004.

JM – **H.264/AMC JM Reference Software**. Disponível em: <http://iphome.hhi.de/suehring/tml/>. Acesso em: março 2012.

JVT – JVT TEAM. **Draft ITU-T Rec. and final draft international standard of joint video specification.** [S.l.]: [s.n.]. 2003.

KALVA, H. The H.264 Video Coding Standard. In: IEEE Computer Society Press, Los Alamitos, CA, USA. **Proceedings...** [Los Alamitos: IEEE], 2006, p. 86-90.

KUHN, P. **Algorithms, Complexity Analysis and VLSI Architectures for MPEG4 Motion Estimation.** Boston: Kluwer Academic Publishers, p. 239, 1999.

KUNG, M.C.; AU, O.C.; WONG, P.H.W.; CHUN HUNG LIU. Block based parallel motion estimation using programmable graphics hardware. In: IEEE International Conference on Audio, Language and Image Processing (ICALIP), Shanghai, China. **Proceedings…** [Shanghai: IEEE], 2008, p. 7-9.

LEE, C-Y; LIN, Y-C; WU, C-L; CHANG, C-H; TSAO, Y-M; CHIEN, S-Y. Multi-Pass and Frame Parallel Algorithms of Motion Estimation in H.264/AVC for Generic GPU. In: IEEE International Conference on Multimedia and Expo (ICME), Beijing, China. **Proceedings…** [Beijing: IEEE], 2007, p.1603-1606.

LIN, C.; LEOU, J. An Adaptative Fast Full Search Motion Estimation Algorithm for H.264. In: IEEE International Symposium Circuits and Systems (ISCAS), Kobe, Japan. **Proceedings...** [Kobe: IEEE], 2005, p. 1493-1496.

LIN Y-C; PEI-LUN LI; CHIN-HSIANG CHANG; CHI-LING WU; YOU-MING TSAO; SHAO-YI CHIEN. Multi-pass algorithm of motion estimation in video encoding for generic GPU. In: IEEE International Symposium Circuits and Systems (ISCAS), Boston, MA, USA. **Proceedings…** [Boston: IEEE], 2006, p. 21-24.

MONTEIRO, E; VIZZOTTO, B.; DINIZ, C.; ZATT, B.; BAMPI, S. Multiprocessing GPU Acceleration of H.264/AVC Motion Estimation under CUDA Architecture. In:

Designing for Embedded Parallel Computing Platforms: Architectures, Design Tools, and Applications, Grenoble, France. **Proceedings…** Grenoble: 2011a.

MONTEIRO, E; VIZZOTTO, B.; DINIZ, C.; ZATT, B.; BAMPI, S. Multiprocessing Acceleration of H.264/AVC Motion Estimation Full Search Algorithm under CUDA Architecture. In: South Symposium on Microelectronics (XXVI SIM). **Proceedings…** Novo Hamburgo: 2011b.

MONTEIRO, E; VIZZOTTO, B.; DINIZ, C.; ZATT, B.; BAMPI, S. Applying CUDA Architecture to Accelerate Full Search Block Matching Algorithm for High Performance Motion Estimation in Video Encoding. In: IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'2011), Vitória, ES, Brazil. **Proceedings…** [Vitória: IEEE], 2011c, p.26.29.

MPI – **The Message Passing Interface (MPI) standard**. Disponível em: < http://www.mcs.anl.gov/research/projects/mpi>. Acesso em: março 2012.

NVIDIA – **NVIDIA Home**. Disponível em: < ttp://www.nvidia.com.br/page/home.html>. Acesso em: março 2012.

NVIDIA CUDA – **NVIDIA CUDA Programming Guide**. [S.l.]: NVIDIA, v4.1, November 2011. Disponível em: < http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf >. Acesso em: março 2012.

NVIDIA FERMI – **NVIDIA FERMI Next Generation CUDA Architecture**. Disponível em: < http://www.nvidia.com/object/fermi_architecture.html>. Acesso em: março 2012.

OPENCL – **OpenCL – The open standard for parallel programming of heterogeneous systems**. Disponível em: < http://www.khronos.org/opencl/>. Acesso em: março 2012.

OPENMP – **The OpenMP API specification**. Disponível em: < http://openmp.org/wp>. Acesso em: março 2012.

PILLA, L, L. **Análise de Desempenho da Arquitetura CUDA Utilizando os NAS Parallel Benchmarks.** 2009. 60 f. Projeto de Diplomação (Bacharelado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

POLLACK, J. Displays of a Different Stripe. **IEEE Spectrum**, [S.l.], v. 43, n. 8, p. 40-44, Aug. 2006.

PORTO, M, S. **Arquiteturas de Alto Desempenho e Baixo Custo em Hardware para a Estimação de Movimento em Vídeos Digitais**. Dissertação (Mestrado em Ciência da Computação) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, RS, 2008.

PORTO, R. **Desenvolvimento Arquitetural para Estimação de Movimento de Blocos de Tamanhos Variáveis Segundo o Padrão H.264/AVC de Compressão de Vídeo Digital**. Dissertação (Mestrado em Ciência da Computação) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, RS, 2008a.

PORTO, M, S. **Desenvolvimento Algorítmico e Arquitetural para a Estimação de Movimento na Compressão de Vídeo de Alta Definição**. Tese (Doutorado) -

Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, RS, 2012.

PURI, A.; et al. Video Coding Using the H.264/MPEG-4 AVC Compression Standard. **Elsevier Signal Processing: Image Communication**. [S.l.], n. 19, p.793–849, 2004.

RANDIMA, F.; MARK, J. **The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics**. Addison-Wesley Professional, 2003.

RICHARDSON, I. **Video Codec Design**: Developing Image and Video Compression Systems. Chichester: John Wiley and Sons, 2002.

RICHARDSON, I. **H.264/AVC and MPEG-4 Video Compression – Video Coding for Next-Generation Multimedia**. Chichester: John Wiley and Sons, 2003.

SCHWALB, M.; EWERTH, R.; FREISLEBEN, B. Fast Motion Estimation on Graphics Hardware for H.264 Video Encoding, Multimedia, **IEEE Transactions on Multimedia**, vol.11, no.1, pp.1-10, 2009.

SHI, Y.; SUN, H. **Image and Video Compression for Multimedia Engineering: Fundamentals, Algorithms and Standards**. Boca Raton: CRC Press, 1999.

SULLIVAN, G.; OHM, J. Meeting report of the first meeting of the Joint Collaborative Team on Video Coding (JCT-VC), **Joint Collaborative Team on Video Coding of ITU-T SG16 WP3 and ISO/IEC JTC1/SC29/WG11 (JCTVC-A200)**, Dresden, 2010.

TANENBAUM, A. S. **Modern Operating Systems**. 3. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007.

THRUST – **Thrust - Code at the speed of light**. [S.l.]: Disponível em: < http://code.google.com/p/thrust/wiki/QuickStartGuide>. Acesso em: março 2012.

XIRU CLUSTER – **Xiru Cluster member of Grid'5000**. Disponível em: < http://gppd.inf.ufrgs.br/cms/gppd/?q=en/resources-list>. Acesso em: março 2012.

X264 – **x264 codec**. Disponível em: <http://x264.nl/>. Acesso em: março 2012.

YANG. S.; Lin, T.; CHIEN, S. Real-time Motion Estimation for 1080p videos on graphics processing units with shared memory optimization, In: IEEE Workshop on Signal Processing Systems (SiPS), Tampere, Finland. **Proceedings…** [Tampere: IEEE], 2009, p. 7-9.

YI, X.; LING, N. Rapid Block-matching motion estimation using modified diamond search algorithm. In: IEEE International Symposium on Circuits and Systems (ISCAS), Kobe, Japan. **Proceedings**... [Kobe: IEEE], 2005, p. 5489 – 5492.

YI, X.; ZHANG, J.; LING, N.; SHANG, W. Improved and simplified fast motion estimation for JM, In **Proc**. JVT Meeting, Poland, Tech. Rep. JVT-P021, 2005b.

ZHU, C.; LIN, X.; CHAU, L. Hexagon-based Search pattern for fast Block motion estimation. **IEEE Transactions on Circuits and Systems for Video Technology**, New York, v. 12, n. 5, p. 349 – 355, 2002.

ZHU, S.; MA, K. A New Diamond Search Algorithm for Fast Block-Matching Motion Estimation. **IEEE Transactions on Image Processing,** [S.l.], v. 9, n. 2, p. 287-290, 2000.

APÊNDICE A <ARTIGOS RELACIONADOS COM OS RESULTADOS APRESENTADOS NESTA DISSERTAÇÃO>

Este apêndice lista as principais publicações realizadas pela autora com os resultados dos desenvolvimentos apresentados nesta dissertação.

Os resultados dos desenvolvimentos foram originalmente tratados nas seguintes publicações:

- (MONTEIRO, VIZZOTTO, DINIZ, ZATT, BAMPI) - Multiprocessing GPU Acceleration of H.264/AVC Motion Estimation under CUDA Architecture. Designing for Embedded Parallel Computing Platforms: Architectures, Design Tools, and Applications (DEPCP, 2011).

- (MONTEIRO, VIZZOTTO, DINIZ, ZATT, BAMPI) - Multiprocessing Acceleration of H.264/AVC Motion Estimation Full Search Algorithm under CUDA Architecture. XXVI SIM - South Symposium on Microelectronics (SIM, 2011).

- (MONTEIRO, VIZZOTTO, DINIZ, ZATT, BAMPI) - Applying CUDA Architecture to Accelerate Full Search Block Matching Algorithm for High Performance Motion Estimation in Video Encoding. 23rd International Symposium on Computer Architecture and High Performance Computing (SBACPAD, 2011).

Os artigos listados abaixo foram submetidos e encontram-se sob avaliação:

- (MONTEIRO, MAULE, SAMPAIO, DINIZ, ZATT, BAMPI) - Parallelization of Full Search Motion Estimation Algorithm for Parallel and Distributed Platforms. International Journal of Parallel Programming (IJPP, 2011).
- (MONTEIRO, MAULE, SAMPAIO, DINIZ, ZATT, BAMPI) - Real-Time Block Matching Motion Estimation onto GPGPU. IEEE International Conference on Image Processing (ICIP, 2011).