EXPERIMENTAÇÃO PRÁTICA DO USO DA ARQUITETURA DE PROCESSAMENTO GRÁFICO NVIDIA CUDA PARA AUMENTO DA EFICIÊNCIA DE ALGORITMOS DE ESTIMATIVA DE MOVIMENTO EM UM CODIFICADOR DE VÍDEO H.264

Ronaldo Husemann[1], Augusto L. Lenz[2], Marco A. Gobbi[3], Valter Roesler[4]

Resumo: Atualmente, se observa a adoção de modernos codificadores de vídeo sobre diferentes soluções multimídia, tais como TV Digital, teleconferência, vídeossegurança, entre outras. Internamente a um codificador de vídeo o algoritmo mais crítico é a estimativa de movimento, que é responsável por identificar e eliminar a redundância temporal entre quadros consecutivos. Considerando a relevância deste algoritmo, este trabalho apresenta um método alternativo para a estimativa de movimento, que foi especialmente otimizado para execução paralela, objetivando sua implementação sobre modernas plataformas de processamento gráfico, como a tecnologia NVIDIA CUDA®. Resultados experimentais feitos sobre diferentes vídeos registraram ganhos médios de velocidade de mais de 400% (4 vezes mais rápidos), pelo uso da solução proposta.

Palavras-chave: Codificador de vídeo H.264. Pesquisa SAD. Computação paralela. GPU CUDA.

1 INTRODUÇÃO

Nos últimos anos, o uso de vídeos digitais na área científica ou mesmo em aplicações de entretenimento aumentou muito, sendo comumente encontrados em equipamentos como *smartphones*, câmeras digitais, terminais de acesso (*set top boxes*), entre outros. Parte relevante nesse cenário são os codificadores de vídeo, que são responsáveis por comprimir as informações de vídeo, reduzindo suas demandas por espaço. Sua função é, portanto, fundamental para permitir a implantação de aplicações multimídia sobre bandas de comunicação restrita. Sua implementação prática, entretanto, é bastante complexa, pois requer grandes capacidades de processamento, tornando necessárias técnicas computacionais sofisticadas e exaustivas (RICHARDSON, 2003).

Para se ter uma ideia dessa complexidade, pode-se considerar que um codificador de vídeo H.264, padrão atual de codificação de vídeo adotado pelo Sistema Brasileiro de TV Digital (SBTVD), para operar sobre vídeos de alta definição, chega a exigir até 3,6 TIPS (*Tera Instructions Per Second*) (CHEN et al., 2006).

Basicamente, dentro de um codificador de vídeo, se destacam por sua maior complexidade os algoritmos de estimativa de movimento, que, em casos extremos, podem demandar mais de 80%

1 Doutor em Engenharia Elétrica pela Universidade Federal do Rio Grande do Sul (UFRGS). rhusemann@inf.ufrgs.br

2 Graduado em Engenharia de Controle e Automação pela Univates. augustollenz@gmail.com

3 Bolsista de Iniciação Científica e aluno do Curso Superior de Engenharia de Controle e Automação da Univates. marcogobbi@universo.univates.br

4 Doutor em Ciências da Computação pela Universidade Federal do Rio Grande do Sul (UFRGS). roesler@inf.ufrgs.br

dos recursos computacionais de um codificador de vídeo (RICHARDSON, 2003). O motivo para isso é que esses algoritmos realizam processos extensivos de pesquisa e comparação, analisando pequenos blocos de *pixels* de cada imagem de entrada na busca pela posição mais provável para onde estes possam ter se movido ao longo da sequência de vídeo, considerando-se diferentes imagens de referência (quadros passados ou mesmo futuros).

Assim sendo, uma das formas mais efetivas para se aumentar o desempenho de um codificador de vídeo reside no aprimoramento da implementação do algoritmo de estimativa de movimento adotado.

Uma estratégia recente que vem sendo usada para atender a elevadas demandas computacionais, como é o caso dos algoritmos de estimativa de movimento, se baseia na exploração do uso de processadores gráficos de alto desempenho, presentes nas modernas placas de vídeo atuais. Modernas unidades de processamento gráfico (*Graphical Processing Units* – GPUs) possuem uma arquitetura altamente paralela, composta por centenas de núcleos operacionais, que permitem executar em um mesmo ciclo de operação um grande número de ações simultâneas. Sua forma de organização é particularmente apropriada para realizar o processamento de gráficos em três dimensões (3D), porém, quando devidamente explorada, pode também ser empregada em diversos outros campos (DINH, 2008).

Este conceito de implementação de aplicações mais genéricas sobre plataformas gráficas é chamado de *General Purpose Graphics Processing Unit* (GPGPU) e tem sido explorado em diferentes centros de pesquisa nos últimos anos (HAN; ABDELRAHMAN, 2011).

Considerando este cenário, o presente artigo realiza uma análise investigativa prática do uso da tecnologia de GPGPU NVIDIA CUDA® como plataforma de desenvolvimento para aumento de velocidade para um algoritmo de estimativa de movimento. A solução proposta foi desenvolvida a fim de ser aplicável para qualquer codificador de vídeo compatível com o padrão H.264.

Como resultado deste trabalho, foi produzida uma versão inovadora de estimador de movimento, adequadamente otimizada para suportar um elevado grau de paralelismo, tornando-se assim bastante apropriada para implementação em arquiteturas de processamento paralelo, como é o caso das modernas GPUs.

A validação da solução proposta foi implementada sobre o *software* de referência JSVM, que é disponibilizado pela entidade *Joint Vídeo Team* (JVT), para desenvolvedores de soluções de codificação baseadas no H.264 SVC (*Scalable Vídeo Coding*), versão mais recente do codificador ITU-T H.264 (ITU, 2008). Os resultados obtidos, comparando-se um ambiente de computação convencional (monoprocessador) com a arquitetura proposta de estimativa de movimento, rodando em placa de processamento gráfico comercial, apontam para ganhos de desempenho de mais de 400% para vídeos de alta resolução.

O artigo é organizado da seguinte maneira: a seção 2 traz uma visão geral de mecanismos de estimativa de movimento, a seção 3 apresenta a arquitetura GPGPU NVIDIA CUDA; a seção 4 apresenta alguns trabalhos relacionados; a seção 5 descreve a arquitetura da solução de estimativa de movimento proposta; a seção 6 mostra os dados reais obtidos comparando o método proposto com uma arquitetura tradicional e finalmente a seção 7 apresenta as considerações finais do artigo.

2 ESTIMATIVA DE MOVIMENTO

Dentro de um codificador de vídeo, o módulo de estimativa de movimento deve explorar a redundância temporal existente entre imagens de um mesmo vídeo. Isto é realizado a partir da pesquisa pela movimentação de blocos de *pixels* dentro de uma área de busca de tamanho fixo ou variável. Este tipo de pesquisa deve ser realizada sobre imagens de referência (já previamente

recebidas), buscando-se a maior similaridade entre o bloco procurado e cada um dos possíveis blocos presentes na área de busca da imagem de referência em qualquer sentido (movimentação vertical, horizontal ou diagonal).

Dentre os diferentes mecanismos de cálculo de similaridade entre blocos, o mais comum é o *Sum of Absolute Differences* (SAD), que representa o erro residual entre dois blocos através do cálculo das diferenças absolutas entre *pixels*. Esse algoritmo é bastante utilizado em codificadores de vídeo, pois demanda apenas operações triviais como somas e subtrações, enquanto que outros cálculos alternativos de similaridade como o *Mean Square Error* (MSE) ou *Sum of Square Errors* (SSE), requerem adicionalmente operações de multiplicação ou outras funções ainda mais complexas (RICHARDSON, 2003).

De forma simplificada a expressão usada para o cálculo de SAD entre dois blocos de *pixels* quaisquer é apresentada abaixo.

$$SAD = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \left| (C_{ij} - R_{ij}) \right|$$

Onde:

N é o tamanho do bloco

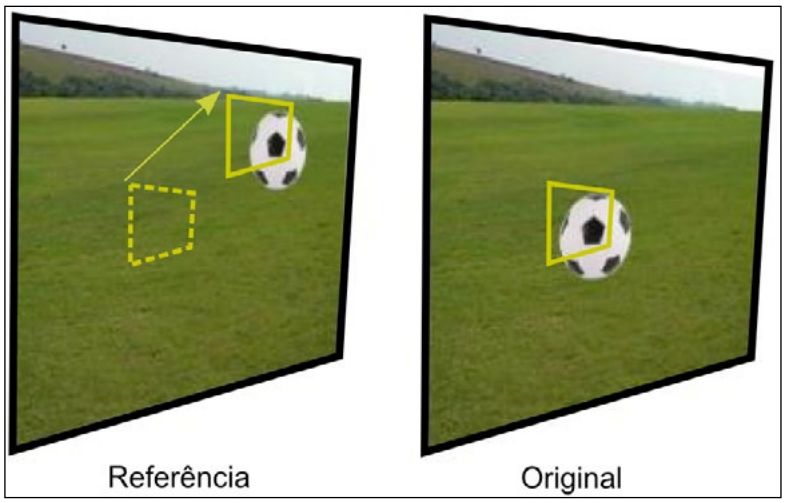
C_{ij} é o *pixel* na posição *ij* do bloco atual

R_{ij} é o *pixel* na posição *ij* do bloco de referência

Durante um processo de estimativa de movimento, o bloco de *pixels* da imagem de referência que apresentar o menor valor de SAD deverá ser selecionado como a melhor estimativa de movimento encontrada. Sua distância em relação ao bloco atual é chamada de vetor de movimento e será salva como resultado do procedimento de busca. Neste caso, o codificador pode apenas armazenar esta informação (vetor de movimento) e passa isto ao decodificador, dispensando a necessidade de retransmitir dados já recebidos, o que contribui para aumentar a taxa de compressão do vídeo.

Um exemplo do resultado deste procedimento é ilustrado na Figura 1, onde se observa a movimentação do objeto bola em duas imagens distintas. No exemplo, a seta apresentada no quadro de referência indica, de forma genérica, o vetor de movimento calculado.

Figura 1 – Localização de bloco similar na imagem de referência



O algoritmo de estimativa de movimento que gera o vetor de movimento mais preciso é conhecido como algoritmo de pesquisa completa (*full search*), uma vez que procura pelo bloco de *pixels* mais similar dentro da totalidade das posições da janela de busca da(s) imagem(s) de referência. Ou seja, o procedimento de busca completa é efetuado considerando-se todos os possíveis blocos da imagem, à procura da movimentação mais correta não importando o quão longe este esteja do ponto original.

Pelo fato do algoritmo de busca completa analisar todas as opções possíveis de deslocamento, ele consegue determinar com precisão o melhor vetor de movimento para aquela condição. Entretanto, na prática o procedimento de busca completa não costuma ser muito utilizado em codificadores comerciais, principalmente devido ao elevado custo computacional demandado. Além disso, também deve-se considerar que dificilmente entre imagens consecutivas de um vídeo ocorrem grandes movimentações relativas, o que indica que a maior parte dos procedimentos realizados por uma busca completa seria desnecessária.

Visando a otimizar este método, diversos algoritmos alternativos de estimativa de movimento foram propostos nos últimos anos. A principal característica explorada por estes algoritmos é de não analisar todos os blocos da janela de busca (como faz o algoritmo *full* seach), mas utilizar-se de um padrão de busca geometricamente distribuído, que consiga inferir de forma relativamente eficiente o caminho da movimentação após um número reduzido de iterações.

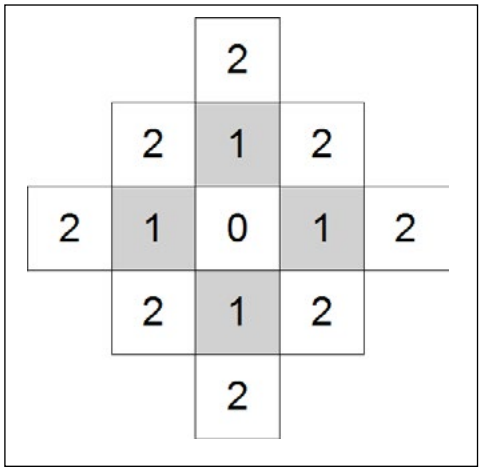
Alguns dos algoritmos mais conhecidos nesta linha são o *Three-Step Search* (KOGA, 1981), o *Four Step Search* (PO, 1996), o *Diamond Search* (DS) (ZHU, 2000) e o *Enhanced Predictive Zonal Search* (EPZS) (TOURAPIS, 2003), que empregam padrões de busca baseados em dois tipos de diamante: grande e pequeno.

Na implementação do presente trabalho, se utilizou também uma topologia de diamante para cálculo da estimativa de movimento. Optou-se pelo padrão de diamante pequeno por este ser mais simples de se implementar, ao mesmo tempo em que se ajusta bem a aplicações que apresentam nível pequeno e médio de movimentação (TOURAPIS, 2001).

Uma ilustração do padrão de busca em diamante pequeno adotado está representada na Figura 2. Neste padrão, as buscas se processam a partir do ponto inicial em relação ao bloco na posição imediatamente superior, imediatamente inferior e aos seus vizinhos laterais (direita e esquerda).

Os números indicados na figura representam o estágio de cada processo iterativo de busca. Isso quer dizer que inicialmente se realiza a avaliação do bloco central (0) em relação aos blocos do primeiro estágio (indicados como 1). Se algum dos blocos avaliados no estágio 1 tiver um valor de SAD menor que o SAD central, deve-se então se reposicionar sobre este o padrão de diamante pequeno e então iniciar o cálculo a similaridade dos blocos do segundo estágio (identificados como 2) localizados no entorno deste. Na prática, o bloco que obtiver o menor resultado de erro absoluto (SAD) indica o caminho para o próximo estágio. A ideia da estratégia é sempre comparar os resultados de um estágio em relação ao estágio anterior. O algoritmo encerra a busca quando encontra o menor erro no centro do diamante (TOURAPIS, 2003).

Figura 2 – Visão geral da topologia de diamante pequeno.



3 TECNOLOGIA GPU NVIDIA

Nos últimos anos, as demandas crescentes por processamento gráfico tridimensional de placas de vídeo, impulsionaram avanços das arquiteturas de processadores dedicados. Além da aplicação típica na área de jogos, empresas fabricantes de processadores gráficos, visando a aumentar sua difusão para outros mercados, começaram a buscar iniciativas para disseminar o uso dessas plataformas também para fins não gráficos, como foi o caso do lançamento, em 2006, da plataforma de *software* e *hardware* CUDA (NVIDIA, 2009).

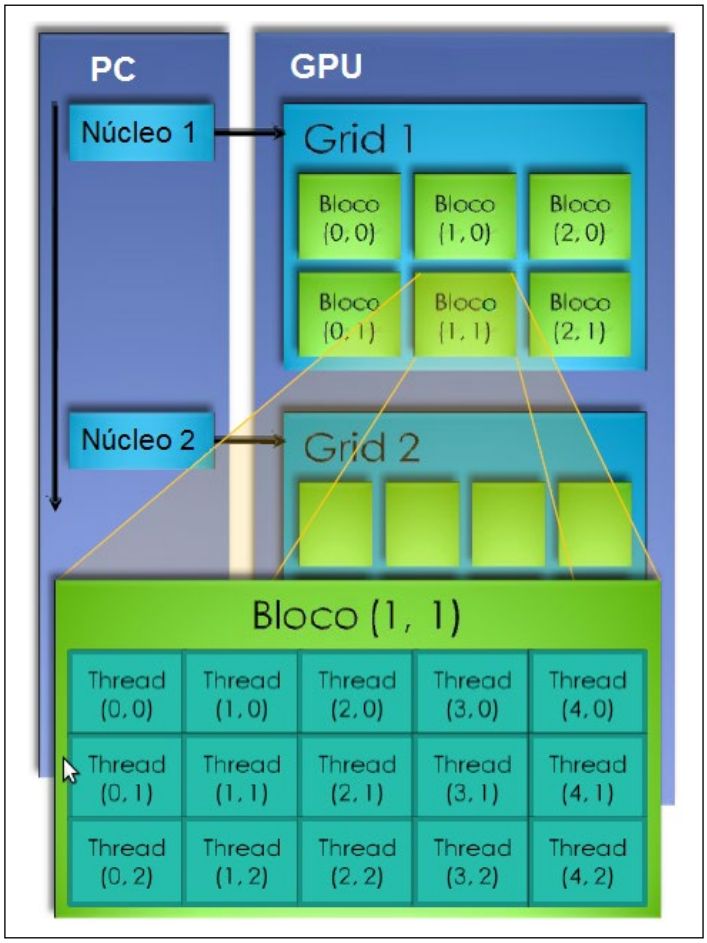
CUDA é uma arquitetura de computação paralela, de propósito geral, que faz uso da capacidade de processamento presente nas GPUs da empresa NVIDIA. Nesta arquitetura o componente principal é chamado *Stream Processor* (SP) (HUANG; SHEN; WU, 2009).

Grupos de *threads* (tarefas independentes previstas pelo algoritmo) podem ser escalonados para execução em um dos núcleos (*kernels*), sequencialmente ou simultaneamente, sem que seja necessário que o programador explicite em qual núcleo o bloco será alocado, tornando assim a solução mais flexível.

Em sua arquitetura as placas de vídeo compatíveis com a tecnologia CUDA possuem um conjunto escalável de multiprocessadores, que são chamados *Streaming Multiprocessors* (SMs). Os SMs são compostos por uma série de processadores escalares (*Scalar processors* – SP), uma unidade de instrução *multi thread* e memória compartilhada.

Os blocos de *threads* criados pelo algoritmo são escalonados automaticamente dentro de *grids* para execução em SMs com capacidade ociosa. As *threads* dentro de um mesmo bloco são executadas concorrentemente, pois cada *thread* é mapeada em um SP distinto, com seus próprios recursos (FIGURA 3) (CHEN, 2008).

Figura 3 – Organização das *threads* em uma arquitetura CUDA



A cada ciclo de operação a mesma instrução pode ser executada em todas as *threads* ativas de um dado bloco (HUANG, 2009). Entretanto, apesar de todas as *threads* começarem a execução no mesmo ponto do código, existe a possibilidade de que haja conflito no acesso a memórias. Neste caso, nem todas as *threads* estarão ativas no mesmo momento, resultando na serialização da execução. Sendo assim, para que seja alcançada eficiência máxima na execução paralela das *threads* é necessário que não hajam ramificações claras no fluxo de execução dentro de um mesmo *grid* (NVIDIA, 2009).

A organização das *threads* se dá na forma de blocos com uma, duas ou três dimensões. A identificação da *thread* que está sendo executada é implementada por meio de índices que identificam a posição da *thread* dentro do bloco (FIGURA 3). Os índices são disponibilizados ao programador por meio de uma variável tridimensional interna. As *threads* que residem no mesmo bloco são executadas no mesmo SM podendo cooperar entre si por meio do compartilhamento de dados, sendo, em alguns casos, necessária a sincronização da execução com funções intrínsecas (barreiras na execução), de forma que a execução prossiga apenas quando todas as *threads* relacionadas tiverem realmente finalizado suas execuções.

Além disso, é importante também observar as características dos diferentes tipos de memória presentes nesta arquitetura, uma vez que sua adequada utilização pode influenciar diretamente o desempenho da solução final. Na prática são cinco tipos de memórias previstos (FIGURA 4):

- Memória local e registradores;
- Memória global;
- Memória compartilhada;
- Memória de textura;
- Memória de constantes.

Os registradores são utilizados para armazenar as variáveis automáticas (variáveis locais). Se não houver espaço suficiente, o compilador alocará as variáveis na memória local. Como a memória local fica fora do *chip* esta apresenta tempo acesso elevado (CROW, 2009).

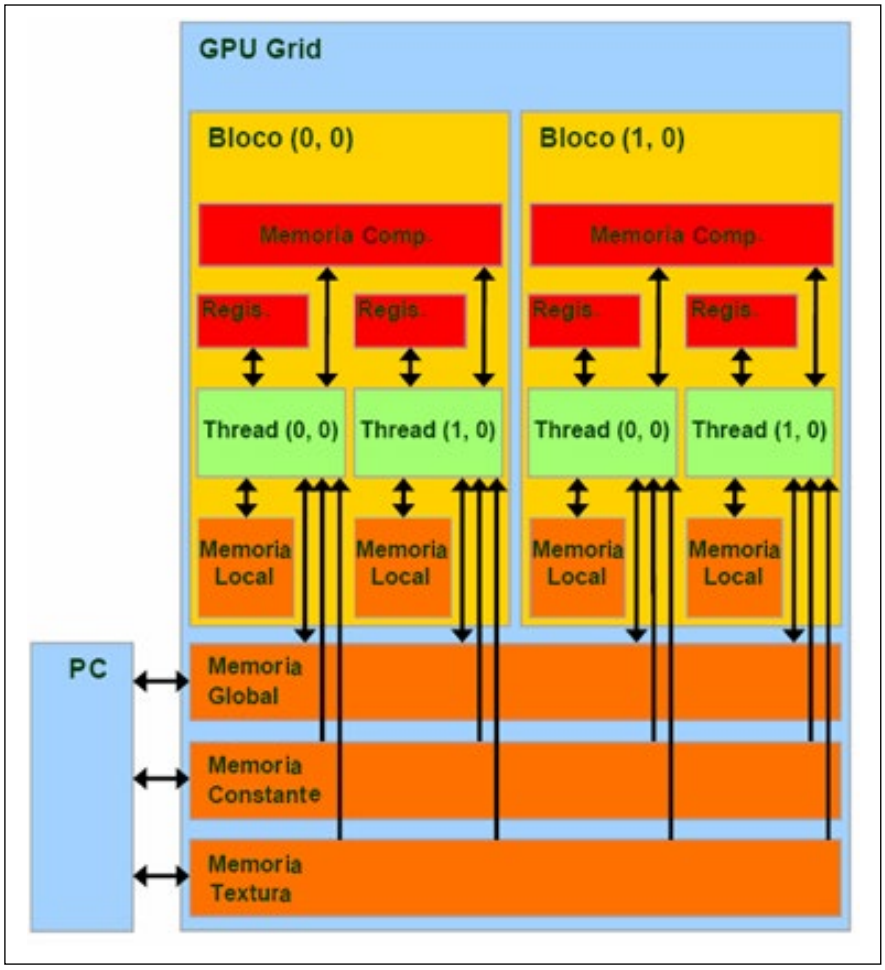
A memória global está localizada na placa de vídeo, no entanto, não sendo integrada ao *chip* da GPU, possui alta latência no acesso. É usada para transações com a CPU externa (PC) alinhadas de 32, 64 ou 128 *bytes*. Na prática, o endereço do primeiro elemento manipulado precisa ser um múltiplo do tamanho do segmento. As requisições de acesso à memória global são combinadas resultando na menor quantidade de transações possível, que obedeça as regras de alinhamento impostas pela arquitetura de *hardware*.

A memória compartilhada está localizada dentro de cada multiprocessador, por isso possui latência cerca de 100 vezes menor do que a memória global ou local, porém o tamanho total desta memória é reduzido (HUANG; SHEN; WU, 2009). A memória compartilhada pode ser utilizada como uma memória *cache* explicitamente gerenciada, organizada em bancos, ou seja, módulos que podem ser acessados simultaneamente, a fim de obter uma alta largura de banda. Essa arquitetura permite que diversas requisições simultâneas, que acessem endereços localizados em diferentes bancos, possam ser atendidas ao mesmo tempo.

A memória de textura (ou memória de superfície) está localizada como memória interna da placa de vídeo e possui uma estrutura de *cache* auxiliar otimizada para acesso a dados que apresentem localidade espacial em duas dimensões. Além disso, esse espaço de memória é projetado de forma a obter fluxos de leitura com latência constante. Dessa forma, uma leitura por meio do *cache* reduz a largura de banda demandada, mas a latência se mantém constante. Quando o dado requisitado estiver presente no *cache* há um ganho significativo no tempo de leitura. Caso contrário o tempo de acesso será o mesmo de uma leitura na memória global convencional. A memória de textura pode ser escrita a partir do computador externo, mas do ponto de vista da GPU é uma memória somente de leitura.

Por fim, a memória de constantes está localizada internamente ao dispositivo gráfico e possui também memória *cache*. Possui acesso (assim como a memória de textura) somente de leitura pela GPU. Esta memória pode ser utilizada pelo compilador por meio de instruções específicas (HARRIS, 2010).

Figura 4 – Hierarquia de memórias da arquitetura CUDA



4 ALGORITMO PROPOSTO

Após a análise destas tecnologias foi desenvolvida uma proposta de algoritmo de estimativa de movimento compatível com o padrão H.264 que alinha a localidade de dados do algoritmo com a estrutura interna dos processadores gráficos da arquitetura CUDA.

Conforme mencionado anteriormente, o algoritmo escolhido se baseia na topologia de busca por diamante pequeno. A ideia proposta está centrada na paralelização do mecanismo de cálculos de similaridade por SAD.

Na prática, o projeto, desenvolvido sob a forma de aplicativo CUDA, é responsável por executar os cálculos de similaridade SAD, valendo-se do conceito de que cada diferença absoluta de *pixel* de uma dada posição pode ocorrer de forma independente e totalmente paralela em relação aos demais *pixels* vizinhos. Essa condição particular permite adequar o algoritmo para explorar um elevado nível de paralelismo.

Conforme visto na seção anterior, para a tecnologia CUDA, esse paralelismo deve ser traduzido em *threads*, que, em última instância, podem ser interpretadas como unidades de execução simultâneas.

Particularmente, na implementação proposta, cada módulo de cálculo de SAD foi estruturado para um *grid* com tamanho básico de 16x8 *threads*. De uma forma geral, o tamanho de bloco de 16x8 elementos permite o uso simultâneo de 128 *threads* para esta implementação. Na prática, o algoritmo de cálculo de SAD utiliza estas 128 *threads* durante os cálculos de diferença absoluta entre *pixels* dos dois blocos analisados, enquanto que as tomadas de decisão do algoritmo de busca são realizadas apenas pela *thread* de índice (0,0).

Na prática, o procedimento implementado de cálculo de SAD consiste em duas operações sequenciais:

- obtenção da diferença absoluta para cada *pixel* dos blocos analisados (bloco de *pixels* do quadro de referência em relação ao bloco atual da imagem que está sendo codificada);
- somatório dos resultados anteriormente calculados.

A implementação dessas operações foi feita utilizando a instrução de baixo nível chamada *usad*, que recebe três argumentos e calcula a soma da diferença absoluta dos dois primeiros argumentos com o terceiro. Essa instrução de baixo nível executa as duas operações (diferença absoluta e soma) com maior velocidade quando comparada com a implementação dessas operações em código de mais alto nível (linguagem C). Além disso, a capacidade da instrução de realizar as duas operações conjuntamente permite acelerar a primeira etapa do algoritmo de redução por meio da junção do cálculo da diferença absoluta e a primeira etapa do somatório de resultados (NVIDIA, 2009).

Para entender melhor esse procedimento, pode-se considerar o caso de um bloco de 16x16 (macrobloco padrão H.264). Nesse caso, inicialmente calcula-se a diferença absoluta dos 128 últimos *pixels* (parte inferior do macrobloco). A seguir, obtém-se o cálculo dos 128 primeiros *pixels* (parte superior) e, na mesma instrução, soma-se esse resultado com as diferenças obtidas na etapa anterior.

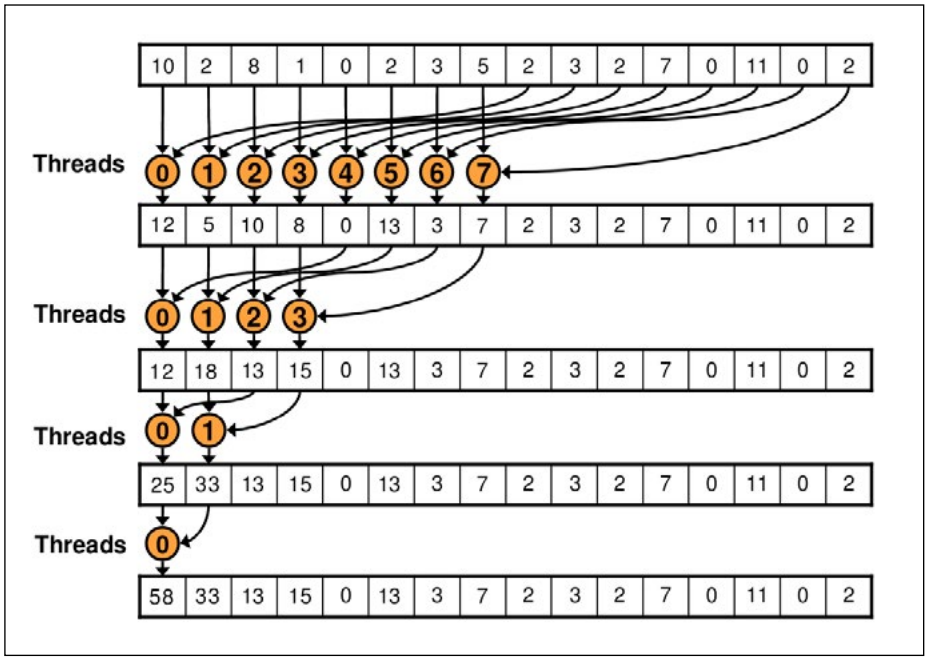
A partir desse resultado (vetor de 128 elementos), dá-se início ao procedimento final de redução (compilação dos dados em uma única variável final).

A primeira etapa da redução requer um número de *threads* ativas igual a 64 (cada qual somando dois elementos vizinhos) e, posteriormente, sendo reduzido para 32. O procedimento se estende reduzindo o número de *threads* à metade a cada estágio novo de execução.

Ao fim do processo, o vetor de valores tem apenas na sua primeira posição o somatório das diferenças absolutas.

A Figura 5 ilustra o procedimento da redução implementado (HARRIS, 2010). Na figura se demonstra como um vetor de 16 elementos seria tratado para permitir o somatório de todos seus elementos em apenas cinco ciclos de operação.

Figura 5 – Procedimento de redução utilizado para o cálculo do valor de SAD



É interessante observar também como, a cada etapa, as *threads* lêem e escrevem simultaneamente na mesma memória, chegando, no último estágio, com apenas uma *thread* ativa para determinar o valor final desejado.

Após o cálculo do SAD para as cinco posições da busca em diamante (centro, cima, baixo, direita e esquerda), são realizados os testes para determinar qual posição possui o menor SAD e, desta forma, decidir se a busca deve ser executada mais uma vez ou se a obtenção do vetor de movimento chegou ao fim.

Além do alinhamento entre *threads* e posições dos elementos internos a serem manipulados no cálculo de SAD, também foi buscado o melhor uso das diferentes memórias da plataforma, de forma a incrementar o desempenho global da solução proposta.

O quadro de referência a ser utilizado pelo algoritmo, por exemplo, foi armazenado na memória de texturas, a qual possui estrutura de *cache* otimizada para dados com localidade 2D (HARRIS, 2010).

Já as variáveis de controle, índices e resultados intermediários foram armazenadas na memória compartilhada, a fim de que possam ser acessados por diferentes *threads* residentes em um mesmo bloco. Essa decisão permitiu uma significativa redução na latência do algoritmo de controle em si.

Por fim, o quadro a ser codificado e os resultados a serem lidos pelo aplicativo no microprocessador da placa mãe residem na memória global. Isso ocorre porque a cada vez que uma transferência de dados é realizada, estes devem ser armazenados na memória global. Esta é uma memória mais lenta, o que motivou que o algoritmo proposto buscasse reduzir o número de acessos à memória global, a qual foi assim usada apenas como interfaces de entrada e saída de dados da placa.

As transferências de dados entre o computador e a placa ocorrem com pacotes que comportam, a cada transferência, múltiplos macroblocos H.264 (16x16 *pixels*), o que também permitiu reduzir as latências de comunicação (CHEN et al., 2006).

5 RESULTADOS EXPERIMENTAIS

A experimentação prática da solução proposta fez uso do *software* de referência JSVM (*Joint Scalable Video Model*) que implementa o codificador H.264/SVC (ITU, 2010). Este *software* JSVM precisou ser adaptado para tornar possível expor à GPU um conjunto de dados suficientemente grande para fazer uso eficiente da sua arquitetura paralela.

Na implementação original do *software* de referência, o módulo de estimativa de movimento recebia um macrobloco, processava-o e devolvia os resultados.

A alteração realizada fez com que o módulo em questão tenha acesso a um quadro inteiro de vídeo, de forma a entregar à placa gráfica, de uma só vez, um conjunto de diversos macroblocos a serem processados em paralelo.

Nos experimentos realizados foram avaliadas duas situações: variabilidade de movimentação (diferentes sequências de vídeo) e diferença na resolução dos vídeos avaliados.

Para se considerar as variações de movimentação foram empregados nos testes três vídeos distintos (City, Crew e Harbour), cada qual com características próprias de variabilidade nas cenas, conforme pode ser visto na Figura 6. O vídeo City (a) apresenta uma visão panorâmica filmada de um helicóptero circundando um arranha-céu. O vídeo Crew (b) traz a movimentação de uma tripulação de ônibus espacial sobre a plataforma de embarque. O vídeo Harbour (c), por fim, mostra o movimento de diversos barcos em um cais.

Figura 6 – Sequencias de vídeo utilizadas nos testes (a) City (b) Crew e (c) Harbour

(a) (b) (c)

O ambiente utilizado no desenvolvimento e experimentação foi composto por um computador com processador Intel Core 2 Quad de 2,66GHz com 2 GB de memória RAM, onde foi instalada uma placa de vídeo NVIDIA GTX 560 Ti. Nos experimentos, foi utilizado o sistema operacional GNU/Linux (distribuição Ubuntu, versão 11) e o *toolkit* CUDA versão 4.2, que contém o compilador e demais ferramentas necessárias ao desenvolvimento de programas que utilizem estas GPUS.

As tabelas a seguir resumem os resultados em termos de ganho de velocidade registrados comparando-se a implementação do algoritmo proposto rodando na placa gráfica em relação ao mesmo algoritmo sendo executado no processador principal do computador (atrasos de comunicação entre computadores e placa gráfica estão inclusos nestas medições).

A Tabela 1 traz os resultados obtidos para a avaliação de vídeos em resolução QCIF (144x176 *pixels*).

Tabela 1 – Ganho de desempenho para vídeos QCIF

Vídeo	Médio	Máximo
City	1,03	1,43
Crew	1,04	1,61
Harbour	0,79	1,06

A análise dos resultados da Tabela 1 aponta para ganhos relativamente pequenos de desempenho nesta condição (inclusive para o vídeo Harbour a solução em GPU foi, em média, mais lenta que a solução em CPU). O fato que explica este baixo desempenho é o tamanho reduzido dos pacotes transferidos. Para volumes pequenos de dados (vídeo QCIF) o tempo de transferência de dados entre computador e placa gráfica se torna significativo, o que limita bastante o desempenho da solução.

Já a Tabela 2 apresenta os resultados obtidos para a avaliação do mesmo algoritmo, porém, neste caso avaliando os mesmos vídeos em resolução CIF (288x356 *pixels*).

Tabela 2 – Ganho de desempenho para vídeos CIF

Vídeo	Médio	Máximo
City	1,85	2,37
Crew	2,22	3,73
Harbour	1,47	2,01

Conforme comentado, o tamanho dos pacotes transferidos afeta o desempenho global do sistema. Para vídeos maiores, com é o caso da resolução CIF, o ganho se apresentou com maior evidência chegando a ser, em média, cerca de duas vezes mais rápido na solução em GPU (vídeo Crew).

A tabela 3, por fim, traz os resultados obtidos da solução proposta, durante a avaliação de vídeos em resolução 4CIF (576x704 *pixels*).

Tabela 3 – Ganho de desempenho para vídeos 4CIF

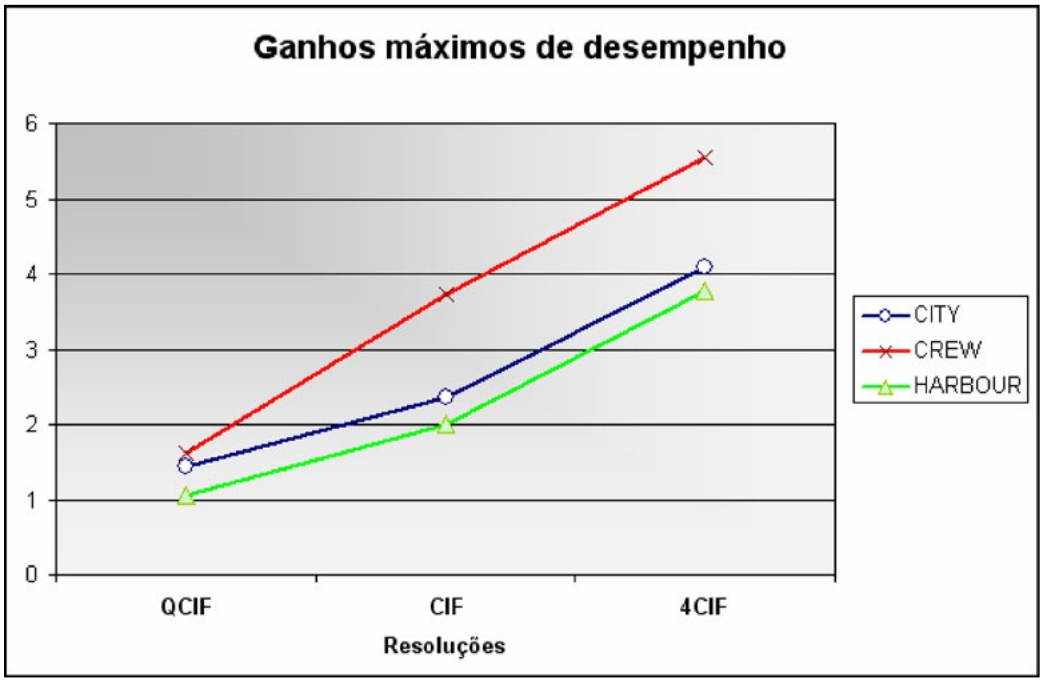
Vídeo	Médio	Máximo
City	3,53	4,09
Crew	4,16	5,54
Harbour	2,9	3,77

Observa-se que para vídeos maiores, o ganho registrado se mostra ainda mais significativo, chegando-se a atingir mais de 350% na média dos três vídeos e mais de 550% no melhor caso (vídeo Crew).

A Figura 7 traz os resultados de ganho de desempenho máximos obtidos para os três vídeos. Os melhores resultados foram obtidos para o vídeo Crew, que apresenta um padrão de movimentação bastante irregular (quando o ganho devido ao paralelismo da tecnologia GPGPU se mostra de fato mais eficiente).

Para todos os vídeos, entretanto, os melhores resultados são obtidos quando resoluções maiores são utilizadas (efeitos da latência de comunicação são menos representativos no tempo total de transferência e execução).

Figura 7 – Ganhos máximos para diferentes vídeos e resoluções



6 CONCLUSÕES

O presente artigo investiga a proposta de uma solução paralela de algoritmo de estimativa de movimento que foi especialmente desenvolvida para explorar recursos das plataformas recentes de processamento gráfico, com ênfase especial para a plataforma NVIDIA CUDA.

Experimentos práticos realizados em laboratório permitiram comprovar o processamento simultâneo de centenas de operações, o que levou a uma redução significativa no tempo de processamento.

Apesar dos ganhos obtidos, vale destacar que para pequenos pacotes de dados, o tempo de transferência de dados entre computador e a placa gráfica pode mascarar os ganhos globais de desempenho da solução.

No geral, a solução proposta apresentou ganhos ponderados de cerca de mais de 350%, em média, para vídeos de resolução 4CIF, atingindo valores máximos de cerca de 450% (mais de 4 vezes mais rápido do que a mesma solução sendo executada em um computador pessoal convencional).

REFERÊNCIAS

CHANG, Yao-Wen; CHEN, Tung-Chieh; CHEN, Huang-Yu. Physical Design for System-On-a Chip. In: LIN, Youn-Long Steve (Ed.). **Essential Issues in SOC Design:** Designing Complex Systems-on-Chip. Springer, 2006. cap. 9.

CHEN, Wei-Nien; HANG, Hsueh-Ming. H.264/AVC motion estimation implmentation on Compute Unified Device Architecture (CUDA). In: IEEE INTERNATIONAL CONFERENCE ON MULTIMEDIA AND EXPO, 2008, Hannover, Alemanha. **Anais...** 2008. p. 697-700.

CHUNG, K.-L., CHANG, L.-C. A new predictive search area approach for fast block motion estimation. **IEEE Transactions on Image Processing,** v. 12, n. 6, p. 648-652, 2003.

CROW, T. S. **Evolution of the Graphical Processing Unit.** 2004.

DO DINH, Minh Tri. **GPUs - Graphics Processing Units.** Institute of Computer Science, University of Innsbruck, July 7, 2008.

HAN, T. D.; ABDELRAHMAN, T. S. Reducing Branch Divergence in GPU programs. In: WORKSHOP ON GENERAL PURPOSE PROCESSING ON GRAPHICS PROCESSING UNITS, 4., mar. 2011, Califórnia, USA. **Proceedings...** Califórnia, 2011.

HARRIS, M. **Optimizing Parallel Reduction in CUDA.** 2010. 38p.

HUANG, Y.-L.; SHEN, Y.-C.; WU, J.-L. **Scalable computation for spatially scalable video coding using NVIDIA CUDA and multi-core CPU MM.09.** China, p. 361-370, Oct., 2009.

ITU (International Telecommunication Unit). **H.264/AVC Reference Software Decoder (version 13.0)**. 2008. Disponível em: <http://iphome.hhi.de/suehring/tml/doc/ldec/html>.

ITU (International Telecommunication Unit). **H.264/SVC Reference Software for H.264 advanced video coding (JSVM version 9.19.9).** 2010.

KOGA, T. et al., Motion compensated interframe coding for video conferencing. In: NATIONAL TELECOMMUNICATION CONFERENCE, 1981, New Orleans. **Proceedings...** New Orleans: NTC, 1981, p. G5.3.1-G5.3.5.

NVIDIA. **NVIDIA's Next Generation CUDA Compute Architecture:** Fermi. 2009. 22 p.

PO, L. M.; MA, W. C. A novel four-step search algorithm for fast block motion estimation. **IEEE Transaction on Circuit Video Technology**, v. 6, p. 313-317, june 1996.

RICHARDSON, I. E. G. **H.264 and MPEG-4 Video Compression.** England: Wiley & Sons, 2003, v. 2.

TOURAPIS, A. M.; AU, 0. C.; LIOU, M. L. Predictive Motion Vector Field Adaptive Search Technique (PMVFAST) - Enhancing Block Based Motion Estimation. In: PROC. OF VISUAL COMMUNICATIONS AND IMAGE PROCESSING, p.883-892, jan. 2001.

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