Implementação da DWT da recomendação do CCSDS de compressão de imagens utilizando GPU

Christofer Schwartz e Marcelo da Silva Pinho

***Resumo*— Este artigo apresenta os resultados de uma implementação parcial da norma de compressão de imagens recomendada pelo CCSDS, utilizando processamento paralelo. Objetiva-se obter uma redução no tempo de processamento da Transformada Wavelet Discreta utilizando o auxílio de uma GPU. Através de imagens de sensoriamento remoto, o algoritmo é testado e apresenta uma aceleração em torno de quatro vezes na execução da DWT de ponto flutuante da recomendação.**

***Palavras-Chave*— Compressão de imagens, CCSDS, GPU, CUDA.**

***Abstract*— This article presents the results of a partial implementation of the image compression standard recommended by the CCSDS, using parallel processing. The objective is to achieve a reduction in processing time of the Discret Wavelet Transform with the aid of a GPU. Through remote sensing images, the algorithm is tested and shows an acceleration around four times the performance of the float DWT used in the recommendation.**

***Keywords*— Image compression, CCSDS, GPU, CUDA.**

I. Introdução

A recomendação do CCSDS (*The Consultative Committee for Space Data Systems*) trazida por [1] e [2] apresenta um sistema de compressão para imagens compostas por 8 bits por *pixel* (256 níveis de cinza) para aplicações aeroespaciais. Como ilustrado pela Figura 1, o sistema recomendado pelo CCSDS faz uso da Transformada Discreta de Wavelet (DWT). A aritmética da DWT pode ser de ponto flutuante (*Float* DWT) ou de valores inteiros (*Integer* DWT). Note que após a etapa da transformada, a utilização de um quantizador não está explicitada. No caso da *Float* DWT, a etapa de quantização é parcialmente substituída por um simples arredondamento dos valores da saída da DWT para o inteiro mais próximo. Na *Integer* DWT suas saídas são multiplicadas por valores inteiros (pesos), que podem ser encontrados em [1]. Em seguida, na etapa correspondente ao codificador de entropia (C.E.), inicia-se a etapa de codificação em plano de bits (BPE). Pode-se dizer que a etapa referente a quantização é feita parcialmente na etapa de arredondamento dos valores dos *pixels* da imagem transformada e parcialmente na etapa de BPE.

Em geral, sistemas embarcados aeroespaciais possuem várias condições no nível de *hardware*. Tomando como exemplo um satélite orbital, o gasto de energia e a velocidade de processamento podem ser um empecilho para se ter um sistema operando em tempo real ou não. Sendo assim, qualquer redução no tempo de processamento, que não implique em um alto consumo de energia, é bem vinda.

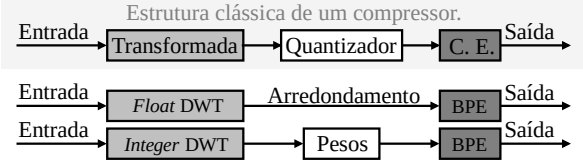

Fig. 1. Comparativo entre a estrutura do compressor recomendado pelo CCSDS e a estrutura clássica de um compressor de imagens.

Como será visto na Seção V, uma referência do gasto de tempo de processamento, para cada uma das etapas acima apresentadas, será obtida utilizando uma implementação da recomendação do CCSDS viabilizada pela *University of Nebraska*. Conclui-se que o tempo médio gasto pelo compressor, na etapa referente a DWT, varia de 24,7% a 27,9% do tempo total.

Em suma, a DWT proposta pela recomendação do CCSDS trata-se de sucessivas operações de convolução nas linhas e colunas das imagens. Uma solução para acelerar o processo de codificação seria realizar tais operações em paralelo. Uma sugestão de como isso pode ser feito e quais resultados o paralelismo de operações pode gerar, serão apresentados neste artigo.

O desempenho de uma implementação paralelizada pode apresentar resultados muito distintos. No fundo, essas variações decorrem de variáveis que aparecem durante a criação do algoritmo e com a escolha da GPU que será utiliza. Os seguintes itens podem contribuir de maneira significante para a variação dos resultados finais:

- A aritmética das operações: inteiros ou ponto flutuante.
- O método de execução da DWT: convolução ou *lifting*.
- O tipo de Transformada Wavelet: Haar, Deslauriers-Dubuc (9,7), Le Gall (5,3), Daubechies (9,7), CCSDS (caso em análise), outros.
- O modelo da GPU: GeForce, Quadro, Tesla ou outros.
- A arquitetura da GPU: Kepler, Fermi ou outras.
- A estratégia de implementação: tipo de memória utilizada (global ou compartilhada), outras.

Essas variações são facilmente encontradas em trabalhos relacionados. Em [3], são feitas algumas implementações em GPU da Transformada Wavelet de Haar. Dentre algumas estratégias descritas e implementadas, a que mais se assemelha com a apresentada por este artigo, tem como resultado ganhos de velocidade de 2 a 5 vezes em comparação com uma execução puramente em CPU. Nota-se que essa não é a me-

Christofer Schwartz e Marcelo da Silva Pinho¸ Departamento de Engenharia Eletrônica, Instituto Tecnológico de Aeronáutica, São José dos Campos-SP, Brasil. E-mails: christofer@ieee.org, mpinho@ieee.org. Os autores agradecem à FINEP/CT - Aeronáutico que, por meio do Projeto AEROSAR (ITA e FUNDEP), deu suporte financeiro a este estudo e ao CNPq, o qual financiou parcialmente este trabalho (143355/2011-2).

lhor estratégia de implementação, porém foi adotada visando minimizar alterações na recomendação.

Outra forma de implementação baseada em *wavelets* é trazida por [4]. Uma Transformada Wavelet é implementada para o processamento em GPU e tem como resultado um ganho de tempo de processamento de 1,7 vezes em comparação com a CPU. Vale comentar que nesse resultado foram contabilizados alguns tempos que serão descartados na análise dos resultados aqui apresentados. Tais tempos podem ser fortemente mitigados em uma aplicação embarcada com *hardware* dedicado.

Trabalhos utilizando GPU para o processamento da *Fast Wavelet Transform* (FWT) de três dimensões (3D-FWT) também podem ser encontrados. Em [5], é feita uma comparação entre duas linguagens de programação voltadas para processamento paralelo: CUDA e OpenCL. Também é feito um comparativo entre duas GPUs com arquiteturas distintas. Ambas as GPUs pertencem a linha de alto desempenho da NVIDIA (Tesla), portanto, os resultados apresentados por [5] serão invariavelmente superiores aos resultados obtidos com GPUs de arquiteturas mais simples. Utilizando uma CPU Intel Core 2 Quad Q6700, [5] apresenta ganhos de tempo que variam de 2,7 a 3,4 vezes (GPU Tesla C870) e 5,3 a 7,4 vezes (GPU Tesla Fermi C2050).

A GPU utilizada para gerar os resultados que serão apresentados neste trabalho é da linha GeForce da NVIDIA. Uma ideia da diferença da capacidade de processamento da linha GeForce e da linha Tesla Fermi (utilizada por [5]) pode ser obtida analisando a Figura 2. Portanto, são esperados resultados inferiores aos apresentados por [5]. Isso não desqualifica o algoritmo criado, pois basta executá-lo em uma GPU com arquitetura superior para que os resultados apresentem ganhos mais significativos.

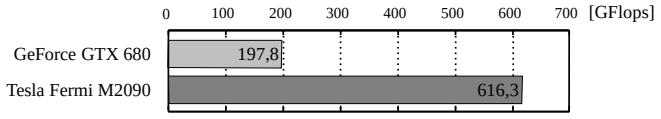

Fig. 2. Performance de ponto flutuante de precisão dupla para duas GPU NVIDIA de linhas distintas.

Neste contexto, podem ser encontrados resultados com valores ainda mais expressivos. É o caso dos resultados trazidos por [6]. Ganhos de tempo que chegam até 40,64 vezes, para imagens de 8192^2 *pixels*, na execução da FWT de duas dimensões utilizando GPU Tesla C870. Nesse caso, também foram dispensados a contabilização de alguns tempos.

Por fim, [7] apresenta um algoritmo para paralelizar a *lifting* DWT, diferente da DWT por convolução recomendada pelo CCSDS. A aplicação do algoritmo é feita em cinco variações da Transformada Wavelet e, utilizando uma GPU da linha GeForce, [7] relata ganhos de velocidade que variam de 10,5 a 13,7 vezes.

Na Seção II será apresentado um resumo da parte da recomendação do CCSDS interessante para o entendimento do problema em pauta. Em seguida, a Seção III dará uma introdução ao processamento utilizando GPU e a linguagem de programação CUDA. Uma vez detalhado o problema, as Sessões IV e V trarão, respectivamente, um detalhamento da solução implementada e os resultados obtidos com a mesma.

II. Visão geral da DWT da recomendação

Descrita em [1] e [2], esta transformada possui três níveis de decomposição. Em cada nível, são aplicados filtros com nove e sete *taps* para os filtros passa-baixas e passa-altas, respectivamente. Os valores dos *taps* para cada um dos filtros, tanto para a DWT quanto para a IDWT, podem ser encontrados nas Tabelas 3-1 e 3-2 de [1].

Enquanto a *Float* DWT gera resultados mais eficientes do ponto de vista de compressão com perdas, somente a *Integer* DWT suporta compressão sem perdas. Neste trabalho será implementada e descrita somente a DWT de ponto flutuante devido aos resultados desejados estarem relacionados a ganhos de desempenho na compressão, mesmo que com perdas.

A. 9/7 Float DWT

Dado um sinal qualquer (x) com $2N$ amostras, sendo $N > 2$, têm-se: $\{x_0,\, x_1,\, x_2,\, \ldots,\, x_{2N-1}\}$. A convolução é feita neste sinal utilizando pares de filtros passa-baixas (C_j) e passa-altas (D_j) com saídas dadas por

$$C_j = \sum_{i=-4}^{4} h_i x_{2j+i}\ ; \quad D_j = \sum_{i=-3}^{3} g_i x_{2j+1+i}\ , \qquad (1)$$

para $j = 0, 1, \ldots, N-1$.

Nota-se que o filtro passa-baixas utilizará as amostras do sinal de entrada com índices pares, enquanto o passa-altas utilizará os valores com índices ímpares. Cada saída tem um comprimento igual a N, logo a união das duas saídas terá um comprimento igual ao do sinal de entrada x.

Outros detalhes triviais com relação a DWT (e.g.: a extensão do sinal por espelhamento das bordas para a realização da convolução) e como é feita a recuperação do sinal transformado podem ser encontrados em [1].

B. A DWT Bidimensional de três níveis

A aplicação da DWT deve ser feita para cada linha da imagem, como também para cada coluna. Essa aplicação sucessiva da DWT entre linhas e posteriormente entre colunas é dita DWT de duas dimensões ou simplesmente DWT 2D. A Figura 3 ilustra a aplicação da DWT 2D.

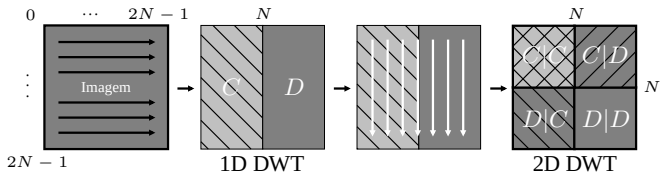

Fig. 3. Ilustração da sequência de aplicação dos filtros passa-altas e passa-baixas para a DWT 2D.

Após aplicar a DWT 2D à imagem, o processo é repetido para parte da imagem transformada. Localizada no canto superior esquerdo, é a parte da matriz que corresponde a aplicação

do filtro passa-baixas no sentido das linhas e colunas. Note que nessa etapa os dados de entrada da DWT 2D correspondem a uma matriz N por N elementos. Por fim, a DWT 2D é aplicada uma terceira vez, também em parte da imagem transformada gerada no processo anterior. Nessa última aplicação, a matriz de entrada da DWT 2D terá $\frac{N}{2}$ por $\frac{N}{2}$ elementos.

Neste ponto é finalizada a etapa da *Float* DWT do compressor. Os valores das saídas da DWT 2D de três níveis são convertidos para valores inteiros antes da aplicação da BPE. Sendo assim, podem ser utilizadas variáveis de precisão simples (*float*) para o cálculo da DWT, não sendo necessária a utilização de variáveis de precisão dupla (*double*).

III. Introdução à GPU

Nos últimos anos, o processamento de dados com o uso de Unidades de Processamento Gráfico vem crescendo bastante. O paralelismo de operações demonstrou, em algumas aplicações, ganhos de desempenho bastante significativos. Um *hardware* bastante acessível para esse fim é a GPU.

No fundo, a GPU por si só não faz nada. Ela precisa ser "gerenciada" por uma CPU. Desta forma, a CPU pode ser chamada de *Host* e a GPU de *Device*. Na prática, o *Host* estará executando um programa de forma serial e, quando exigido pelo algoritmo, ele requisita a execução de um ou mais núcleos (*Kernel*) da GPU. Quando o *Device* termina sua tarefa, o *Host* segue a execução do programa. Em suma, as plataformas trabalham em conjunto, sendo a GPU dependente da CPU.

Caso a GPU utilizada seja da NVIDIA, podem ser utilizadas duas linguagens de programação: CUDA e OpenCL [8]. A implementação descrita neste trabalho utiliza a linguagem CUDA. Isso por ela apresentar melhores resultados e facilidades de implementação [5].

A. Visão geral da linguagem CUDA

A CUDA nada mais é que a linguagem de programação C com extensões mínimas [8]. Ela acrescenta à biblioteca de *runtime* C funções análogas as já existentes: cudaMalloc(), cudaFree(), cudaMemcpy(), entre outras.

A estrutura de execução dos *Kernels* compreende três elementos distintos: grade, bloco e *thread*. Tais elementos são dispostos como mostra a Figura 4.

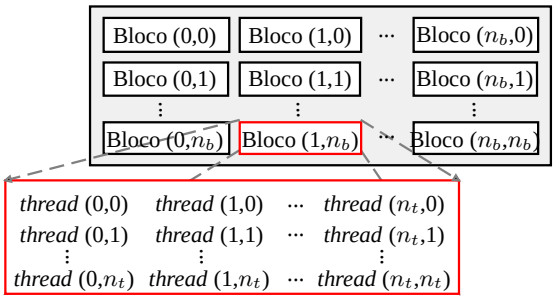

Fig. 4. Distribuição dos blocos e *threads* em uma grade.

Uma grade de execuções é composta por n_b blocos em paralelo e n_b blocos seriais. Cada bloco, por sua vez, pode conter uma matriz de *threads* com dimensões n_t por n_t. Segundo [8], a matriz de *threads* pode ser tridimensional. Contudo, neste trabalho, será utilizada uma estrutura bidimensional. Vale dizer que não são permitidas execuções de duas grades em paralelo. Dessa forma, um código escrito para ser executado por um *kernel*, será executado por outros núcleos em paralelo conforme especificado pela grade.

A CUDA disponibiliza três tipos de memórias: memória global, memória constante, memória compartilhada. A memória constante está vincula à grade. Porém, o *Device* pode apenas ler os dados contidos nessa memória. Diferente da memória constante, a memória global também está vinculada a uma grade, porém o *Device* pode ler e escrever dados nela. A memória compartilhada está vinculada a um bloco. No fundo essa memória tem uma latência inferior a memória global, mas um bloco não tem permissão para ler a memória de outro bloco, utilizada então para a troca de informação entre *threads* de um mesmo bloco.

Outras informações podem ser encontradas em [8].

IV. Metodologia

A CPU hospedeira possui um processador Intel(R) Core(TM) i7-3610QM (terceira geração) com 4 núcleos (8 *threads*) trabalhando a uma frequência de 2241003 KHz. Apesar dessa CPU possuir 4 núcleos de processamento, apenas um será utilizado.

A GPU utilizada é uma NVIDIA GeForce GT 630M (*Streaming Multiprocessor Capability* 2.1). Essa GPU possui dois multiprocessadores (MP) com 48 núcleos cada, totalizando 96 núcleos trabalhando a um *clock* de 950000 KHz.

As duas implementações seguem uma estrutura bem parecida. Isso permite fazer uma análise parcial da implementação, bem como analisar qual o ganho efetivo de se ter o auxílio de um *hardware* com vários núcleos de processamento (mesmo se tratando de núcleos mais simples).

Do ponto de vista de implementação, as operações para realizar a DWT recomendada pelo CCSDS são três. Inicialmente é realizada a etapa de extensão do vetor com o espelhamento das bordas laterais da imagem. Posteriormente são realizadas as operações com os filtros passa-altas e passa-baixas. Neste ponto, pode-se dizer que a DWT já estaria concluída. Contudo, visando repetir o processo para as colunas da imagem, é computada a transposta da imagem transformada.

No fundo, o processo anterior é repetido seis vezes para que seja realizada a DWT 2D de três níveis. Vale lembrar que a quantidade de informação varia para cada nível. De maneira geral, essas seis repetições processam as seguintes quantidades de informação: $(2N)^2$, $(2N)^2$, N^2, N^2, $(\frac{N}{2})^2$, $(\frac{N}{2})^2$.

Ao todo foram feitas duas implementações distintas. Uma utilizando somente a CPU (linguagem C) e outra utilizando a "ajuda" da GPU (linguagem CUDA). Ambos os algoritmos são muito semelhantes e encontram-se ilustrados na Figura 5.

A imagem lida é guardada na memória na forma de um vetor de dados. Esse vetor é copiado para a memória global da GPU. Deste modo, tanto o *Host* como o *Device* terão os dados da imagem à disposição. Caso a opção selecionada no algoritmo seja a execução com GPU, são calculados a quantidade de blocos e *threads* necessários para realizar a execução.

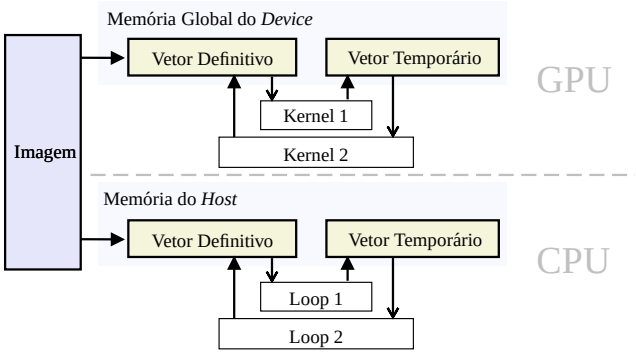

Fig. 5. Diagrama de blocos das etapas de processamento do algoritmo criado.

A estratégia adotada é relativamente simples. Busca-se colocar uma linha inteira da imagem dentro de um bloco. Assim, cada *thread* do bloco iria corresponder a um *pixel* de uma linha (e.g.: $imagem(k2N+j)\ \forall\ j = threadIdx.x\ e\ 0 \leq k \leq 2N$). Porém, o número máximo de *threads* por bloco suportadas pela GPU utilizada é 1024. Uma solução é utilizar mais de um bloco em paralelo para imagens com tamanhos maiores que 1024. Considerando que a quantidade de *threads* por bloco já esteja satisfeita, não é possível adicionar outra linha de *threads* dentro de um mesmo bloco. Portanto, as demais linhas da imagem irão ocupar outros blocos de forma serial dentro da grade.

O *pixel* $imagem(k2N + j)$ será processado quando seus índices corresponderem aos índices de um bloco e de uma *thread*. Os valores de k e j são dados por

$$j = \begin{cases} \underbrace{blockIdx.x}_{0,\ldots,\frac{2N}{1024}} \cdot \underbrace{blockDim.x}_{1024} + \underbrace{threadIdx.x}_{0,\ldots,1024} \\ \qquad\qquad , para\ 2N > 1024 \\ \underbrace{threadIdx.x}_{0,\ldots,1024} \quad , para\ 2N \leq 1024 \end{cases} \tag{2}$$

$$k = \underbrace{blockIdx.y}_{0,1,\ldots,2N} \quad \forall\ N\ . \tag{3}$$

Após o dimensionamento da grade, o primeiro *Kernel* é executado paralelamente entre os núcleos disponíveis. A GPU realiza a cópia dos dados para uma variável temporária, já realizando a extensão do vetor. Ao terminar, o segundo *Kernel* realiza as operações de filtragem, finalizando o processo DWT na horizontal. Note que não existe um bloco para realizar a transposição dos dados. Isso é dispensável pois o segundo *Kernel*, ao devolver a informação processada ao vetor definitivo, já salva os dados nas posições que correspondem à transposta do sinal. Para concluir a DWT 2D, os *Kernels* 1 e 2 são chamados novamente. Para os outros dois níveis restantes, os *Kernels* são chamados mais 4 vezes cada um.

Caso deseje-se processar os dados puramente na CPU, os trabalhos das grades da GPU são substituídos por estruturas de repetição (*looping for*), realizando as operações de forma sequencial.

V. Resultados

Como mencionado na Seção I, uma referência do gasto de tempo de processamento de cada uma das etapas pode ser obtida com a ajuda de uma implementação disponibilizada pela *University of Nebraska*. Junto ao código da implementação disponibilizada, foram adicionados (por esse trabalho) alguns medidores de tempo e os resultados obtidos podem ser vistos na Figura 6. Como fonte de informação foram utilizadas 100 imagens distintas capturadas pelo satélite CBERS-2B. A partir dessas, foram geradas imagens de quatro tamanhos: 512^2, 1024^2, 2048^2 e 4096^2 *pixels*. O programa foi executado 100 vezes para cada uma das 400 imagens e o intervalo de confiança (IC) de 95% foi calculado.

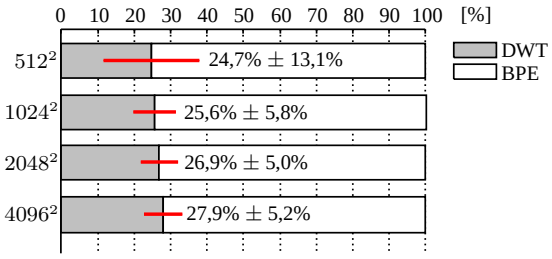

Fig. 6. Porcentagem do tempo total de codificação referente a cada etapa.

Nota-se que para a imagem de menor resolução, o intervalo de confiança é maior. Isso se deve a uma maior flutuação no tempo de processamento. Por ele ser muito pequeno, qualquer variação na ordem de milissegundos pode corresponder a uma grande porcentagem do tempo total de processamento. Conclui-se que o tempo médio gasto no codificador, na etapa referente a DWT, varia de 24,7% a 27,9% do tempo total.

É importante citar que devido a estas medidas de tempo terem sido obtidas utilizando uma implementação pronta, elas podem estar um pouco diferentes do valor que efetivamente se deseja medir. Porém, podem servir como referência para o estudo proposto. Tendo uma referência de tempo gasto pela DWT, os resultados gerados pela implementação descrita neste trabalho podem ser justificados.

Para as mesmas imagens do CBERS-2B, o programa criado realizou 1000 vezes a DWT 2D de três níveis utilizando somente o *Host* e 1000 vezes utilizando o *Host + Device*. Outra simulação foi feita utilizando 32 imagens capturadas por um radar de abertura sintética (SAR) do sensor aeroembarcado E-SAR do Centro Alemão de Pesquisa Aeroespacial (DLR). A Figura 7 expõe os resultados obtidos.

São observados ganhos de velocidade de 3,96 vezes para imagens com 512^2 *pixels*, 3,66 vezes para 1024^2, 3,98 vezes para 2048^2 e 4,19 vezes para 4096^2. De maneira geral, os ganhos para os diferentes tamanhos são muito parecidos. Os resultados são um pouco melhores para imagens com 4096 *pixels*, decaindo levemente a medida que os tamanhos diminuem, com exceção das imagens com resolução de 512^2. Devido a quantidade de *threads* por bloco ser menor que a máxima permitida, a GPU apresentou um resultado um pouco melhor para esse tamanho de imagem.

As imagens de radar contemplam apenas dois tamanhos:

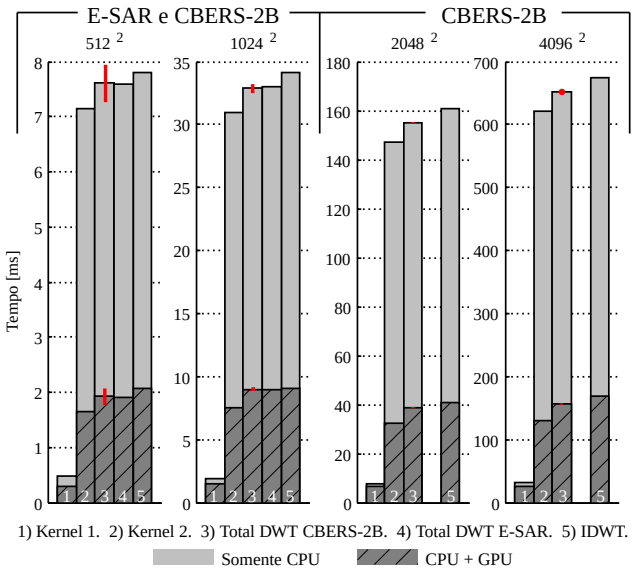

Fig. 7. Desempenho da implementação utilizando GPU.

512^2 *pixels* e 1024^2 *pixels*. Elas apresentaram praticamente os mesmos resultados (ver barras com índice 4 na Figura 7) obtidos com as imagens do CBERS-2B: ganhos de 3,87 vezes para as imagens de 512^2 *pixels* e 3,69 para 1024^2 *pixels*. A variação se dá devido a quantidade de imagens (16 para cada tamanho) não ser grande o suficiente para que o resultado convirja para a média obtida com as imagens do CBERS-2B. Note que as operações envolvidas na DWT e IDWT dependem apenas do tamanho dos vetores de entrada, e não dos seus respectivos valores. Desta forma, os resultados para imagens do CBERS-2B e do SAR devem ser os mesmos. Contudo, na etapa que segue a DWT, isso pode não ser mais verdade. Codificadores que utilizam a dependência estatística dos dados de entrada para realizar a compressão, podem apresentar desempenhos distintos para dados obtidos por fontes distintas.

Os intervalos de confiança para o tempo total da DWT, para cada tamanho de imagem, também foram calculados. Ao analisar a Figura 7, pode-se observar que para as imagens com tamanhos inferiores a 4096^2 o intervalo quase nem aparece. Para as barras referentes ao total de tempo no *Device*, o IC é muito pequeno. Como a GPU não tem outros processos sendo executados em paralelo, os valores das 1000 repetições variam muito pouco, o que não acontece com a CPU, a qual precisa dividir o processamento com outros processos. Vale ressaltar que todos os programas do sistema operacional (SO) foram interrompidos durante as simulações, restando apenas as aplicações essenciais do próprio SO. Existe outro ponto que contribui para a minimização das variações no tempo de processamento do *Host*. A CPU possui quatro núcleos de processamento, sendo assim, o sistema operacional (Windows 8) pode distribuir as aplicações entre os vários núcleos de processamento. Isso evita que outros processos que invariavelmente surjam sobrecarreguem de forma severa o núcleo utilizado para realizar as simulações.

As medidas de tempo foram tomadas no início e no final de cada bloco de operações (ver Figura 5). Os tempos de alocação e desalocação de memória não foram contabilizados em ambos os algoritmos. Lembrando que na prática o sistema estaria implementado em um *hardware* dedicado e tais tempos seriam ínfimos.

Considerando que o sistema a ser embarcado é o de codificação, neste trabalho não foi abordada com detalhes a etapa de reconstrução do sinal. No fundo, a reconstrução ocorre de forma análoga ao que foi visto até aqui. A implementação da IDWT com auxílio da GPU gerou resultados semelhantes aos obtidos com a DWT: 3,76 vezes para imagens de 512^2 e 1024^2, 3,95 vezes para 2048^2 e 4,00 vezes para 4096^2 *pixels* (ver barras com índice 5 na Figura 7).

VI. Conclusões

Diferente de [9], que apresentou diferentes estratégias de implementação objetivando avaliar qual o melhor desempenho do sistema, este artigo paralelizou parte do processamento de compressão de imagens da recomendação [1]. Pode-se concluir que sem realizar grandes mudanças na estrutura recomendada, pode-se migrar o sistema para uma plataforma híbrida (GPU + CPU) e obter ganhos em torno de quatro vezes na parte do processo que corresponde até 27,9% do tempo total de codificação. Os resultados apresentados podem servir como motivação para realizar um estudo de implementação de toda a recomendação, utilizando GPU. Abrindo caminho para um estudo que contabilize o consumo de energia dispendido pelas diferentes plataformas.

Referências

[1] CCSDS, “Image data compression - recommended standard,” CCSDS Recommended Standard for Image Data Compression, Report Concerning Space Data System Standards Blue Book, 2005. [Online]. Disponível em: http://www.ccsds.org

[2] CCSDS, “Image data compression - informational report,” CCSDS Report Concerning Image Data Compression, Report Concerning Space Data System Standards Green Book, 2007. [Online]. Disponível em: http://www.ccsds.org

[3] Z. Wei, Z. Sun, Y. Xie, and S. Yu, “Gpu acceleration of integer wavelet transform for tiff image,” *Parallel Architectures, Algorithms and Programming (PAAP), 2010 Third International Symposium on*, pp. 138–143, 2010.

[4] V. Simek and R. Asn, “Gpu acceleration of 2d-dwt image compression in matlab with cuda,” *Computer Modeling and Simulation, 2008. EMS ’08. Second UKSIM European Symposium on*, pp. 274–277, 2008.

[5] G. Bernabe, G. Guerrero, and J. Fernandez, “Cuda and opencl implementations of 3d fast wavelet transform,” *IEEE Third Latin American Symposium on Circuits and Systems (LASCAS)*, pp. 1–4, 2012.

[6] J. Franco, G. Bernabe, J. Fernandez, and M. Acacio, “A parallel implementation of the 2d wavelet transform using cuda,” *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pp. 111–118, 2009.

[7] W. Van der Laan, A. Jalba, and J. B. T. M. Roerdink, “Accelerating wavelet lifting on graphics hardware using cuda,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 1, pp. 132–146, 2011.

[8] D. B. Kirk and W.-M. W. Hwu, *Programando para Processadores Paralelos: Uma Abordagem Pratica a Programação de GPU*, E. Inc., Ed., 2011.

[9] V. Galiano, M. M. O. López, and H. Migallón, “Improving the discrete wavelet transform computation from multicore to gpu-based algorithm,” *Proceedings of the 11th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE 2011*, p. 544 of 1703, 2011.