Otimização do algoritmo VSTC utilizando programação paralela

Mateus L Assunção, Vinícius G Pereira, Bruno R Campana, Anderson V C Oliveira,
Raquel C G Pinto, Carla L Pagliari
Instituto Militar de Engenharia
Praça General Tibúrcio, 80, 22290-270,
Rio de Janeiro, RJ, Brasil.

RESUMO: *Nesse trabalho, o algoritmo de compressão de imagens "Variable Size Transform Coder" (VSTC) foi paralelizado. Após análise das principais etapas do algoritmo, definimos propostas para implementar a paralelização em rotinas específicas do processo de codificação. Foram geradas duas versões do algoritmo: uma usando MPI para uma plataforma genérica de cluster, e outra usando OpenAcc a ser executada em GPU. Os resultados obtidos levaram a uma redução média de tempo de execução, em relação ao algoritmo original, de 32,3% na versão com OpenAcc e 87,7% na versão com MPI.*

PALAVRAS-CHAVE: *programação paralela, programação em GPU, algoritmo VSTC, compressão de imagens.*

ABSTRACT: *In this work, "Variable Size Transform Coder" algorithm (VSTC) was parallelized. After we analyze the main steps in the algorithm, we defined proposals to implement parallelization on specific routines of the coding process . Two versions of the algorithm were generated: one of them using MPI, for a cluster general platform, and the other one using OpenAcc, to running with GPU. Results led to an average reduction in execution time of 32.3% for OpenAcc version when compared to the original algorithm and 87.7% for MPI version.*

KEYWORDS: *parallel programming, GPU programming. VSTC Algorithm, image compression.*

1. Introdução

O desenvolvimento na computação de alto desempenho tem sido motivado por simulações numéricas de sistemas complexos como a meteorologia, o clima, dispositivos mecânicos, circuitos eletrônicos, processos de fabricação de produtos e reações químicas.

A computação paralela vem sendo explorada nas principais atividades de alto desempenho. É uma forma de computação em que muitos cálculos são realizados de maneira simultânea, que operam segundo o de que problemas de alta complexidade muitas vezes podem ser divididos em partes menores, que são então resolvidos simultaneamente, diminuindo o tempo de execução da computação do problema. Apesar de ser muito explorada atualmente, a computação paralela não é novidade. Paralelismo vem sendo empregado por muitos anos, principalmente em computação de alto desempenho, mas o interesse nele tem crescido ultimamente devido às limitações físicas que limitam a evolução do hardware da CPU (Unidade Central de Processamento).

A integração de computação paralela, redes de alto desempenho, e tecnologias de multimídia estão conduzindo ao desenvolvimento de servidores de vídeo - computadores projetados para servir centenas ou milhares de solicitações simultâneas de vídeo em tempo real. Cada fluxo de vídeo pode envolver tanto taxas de transferência de dados de muitos megabytes por segundo quanto grandes quantidades de processamento de codificação e decodificação.

O algoritmo VSTC [1] é um novo algoritmo de compressão de imagens que tem apresentado bons resultados de compressão, em termos de taxa-distorção, superando algoritmos como o H.264/AVC [2] e MMP [3]. Entretanto, o conjunto de ferramentas utilizados na codificação, como a predição e as transformadas espaciais, associadas à natureza recursiva do algoritmo VSTC, eleva bastante o seu tempo de execução. A computação paralela é uma das alternativas para diminuir o tempo necessário para a realização das tarefas. Para isso, faz-se necessário o estudo das técnicas de paralelismo disponível atualmente, de modo a serem implementadas no código do VSTC.

O objetivo deste trabalho é propor duas versões paralelas do algoritmo VSTC de compressão de imagens. A primeira versão implementa o paralelismo através da comunicação de processos MPI, enquanto a segunda visa à execução do algoritmo em unidades de processamento gráfico (GPU).

Este artigo está organizado da seguinte forma. Na seção 2, abordamos a computação paralela, apresentando conceitos sobre a comunicação MPI e a arquitetura de placas gráficas, assim como os modelos de programação utilizados em cada um, como o CUDA e o *OpenAcc*. O algoritmo de compressão de imagens VSTC é apresentado na seção 3, onde uma visão geral do seu funcionamento é feita seguindo as etapas do algoritmo.

Na seção 4, apresentamos uma análise que nos leva a um levantamento dos possíveis pontos do algoritmo a serem paralelizados, utilizando tanto o MPI, quanto a GPU. Assim, soluções de paralelismo são propostas para o algoritmo VSTC. A seção 5 apresenta os resultados obtidos de cada uma das implementações propostas, enquanto que as conclusões são apresentadas na seção 6.

2. Processamento paralelo

Microprocessadores baseados em uma única unidade central de processamento (CPU) impulsionaram aumentos de desempenho e reduções de custo nas operações lógicas nos computadores por mais de duas décadas. Esses dispositivos trouxeram bilhões de operações de ponto flutuante por segundo (GFLOPS) para o *desktop* e centenas de bilhões para os servidores em *cluster*. A demanda sempre aumentou, criando um ciclo positivo para a indústria de computadores [4].

Durante muito tempo, o aumento da velocidade de processamento dos dispositivos foi alcançada a partir do aumento da frequência do *clock* da CPU [5]. Entretanto, os fabricantes se depararam com restrições de calor e outros limites físicos para que continuassem a evolução do *hardware* por nessa linha [4]. Os fabricantes de microprocessador passaram para o modelo de várias unidades de processamento, conhecida como núcleos, que são usadas em cada chip para aumentar o poder de processamento, exercendo um grande

impacto sobre a comunidade de desenvolvimento de software [4]. A partir de 2005, os principais fabricantes de microprocessadores começaram a oferecer processadores com dois núcleos e na sequência três, quatro, seis e oito [5].

A GPU (*Graphics Processing Unit*), que é unidade de processamento presente nas placas de vídeo, possui na ordem de milhares de núcleos, cada um deles sendo multithread. Processadores com muitos núcleos, especialmente as GPUs, têm liderado a corrida de desempenho em ponto flutuante desde 2003, pelo fato da GPU ser especificamente voltada para a programação paralela, enquanto a arquitetura da CPU ser otimizada para o código sequencial [4].

O processamento paralelo consiste em múltiplos processadores executando partes de um mesmo programa simultaneamente, tendo como objetivo principal reduzir o tempo total de processamento (*wall-clock time*). É diferente do processamento serial onde as expressões são executadas uma após a outra.

Para aferir o desempenho de programas paralelos, as medições mais aceitas são a eficiência e o *speed-up* [6]. A eficiência avalia como o algoritmo paralelo usa o tempo nos vários processadores e é calculada de acordo com a Eq. 1:

$$E(p)=\frac{Ts}{p \cdot Tp} \tag{1}$$

Onde, p é o número de processadores da aplicação, Ts é o tempo de execução do algoritmo sequencial e Tp é o tempo de execução do algoritmo paralelo, executando com p processadores.

Já o *speed-up* avalia o ganho de tempo que o processamento paralelo apresenta sobre o sequencial e pode ser aferido através da equação Eq. 2:

$$speedup=\frac{Ts}{Tp} \tag{2}$$

O ganho de *speed-up* deveria tender a p, que seria o valor ideal. No entanto diversos fatores, impendem que esse ideal seja alcançado, como por exemplo a sobrecarga de comunicação entre os processadores. Dessa forma, a *Lei de Amdahl* é utilizada para encontrar o máximo *speed-up* esperado para um sistema. Essa lei considera que o *speed-up* é limitado pela fração sequencial do programa. A *Lei de Amdhal* é descrita pela Eq. 3:

$$SpeedupTeorico=\frac{1}{(1-f)+\frac{f}{p}} \tag{3}$$

Onde f é a fração do código do programa que, de fato, foi paralelizada e p é o número de processadores utilizado. Dessa forma, a *Lei de Amdhal* considera que a parte serial de execução do programa vai limitar o *speed-up* esperado, independente de quantos processadores são utilizados na execução do programa.

2.1 Programação MPI

O MPI (*Message Passing Interface*) é um padrão de interface para a troca de mensagens em arquiteturas paralelas com memória distribuída. Ele define um padrão de troca de mensagens, através da sintaxe e da semântica de um conjunto básico de rotinas [7]. O padrão MPI é o método de comunicação em computação paralela mais portátil e mais amplamente utilizado, oferecendo robustez, escalabilidade e portabilidade sem comprometer o desempenho da aplicação [8].

A programação usando MPI assume que as estruturas de dados não são compartilhadas, mas divididas entre os múltiplos processadores. Essa divisão de armazenamento requer uma técnica de decomposição do domínio, que vai gerar um custo adicional.

No ambiente MPI, as aplicações são constituídas por um ou mais processos que se comunicam através de funções de envio e recebimento de mensagens entre processos. Um conjunto fixo de processos é criado no início da aplicação, mas eles podem executar diferentes programas. Por isso, algumas vezes o padrão MPI é referido como pertencente à classificação MIMD [10].

Todos os processos que são executados em processadores diferentes têm acesso a sua própria memória local e se comunicam através de uma rede. Assim, é possível que todos executem o mesmo código, pois o MPI cria uma cópia de cada variável para cada processo, executando o programa [11].

A paralelização MPI é vantajosa à medida que permite a execução em máquinas paralelas, reduzindo o tempo de execução, e ainda oferece alocação e armazenamento distribuídos [9].

No MPI, cada processo é identificado por um ranque estabelecido no início da execução do programa e a comunicação entre os processos pode ser ponto a ponto, onde os processos utilizam operações para enviar mensagens de um para outro, ou através de operações coletivas de comunicação [10].

Apesar do padrão MPI ser bastante complexo, um conjunto de seis primitivas básicas é capaz de solucionar uma grande quantidade de problemas [8]. Esse conjunto mínimo de rotinas resumidamente possui funções que incluem: iniciar e terminar o ambiente MPI, identificar processos, enviar e receber mensagens. Um dos tipos de parâmetros dessas primitivas são os comunicadores, que especificam grupos de processos e o contexto das operações que serão executadas. Eles servem principalmente para assegurar que mensagens de diferentes propósitos não sejam confundidas [10].

Outra característica importante do MPI é que a troca de mensagens não é determinística, assim não é garantido que as mensagens cheguem na ordem em que foram enviadas. A responsabilidade de assegurar a execução determinística é do programador, que pode distinguir mensagens através de *tags* ou especificar o processo que deve ser remetente da mensagem na operação de recepção [13].

2.2 Computação em GPU

A arquitetura das GPUs é projetada de tal forma que a maior parte dos transistores são dedicados ao processamento de dados, em vez de cachê de dados e controle de fluxo, como ocorre na CPU. No esquema de utilização de transistores, o espaço destinado ao controle de fluxo e cache de dados na CPU é distribuído na GPU entre as ULAs (Unidades Logicas Aritméticas), buscando esconder a latência do acesso à memória com processamento, ao invés de grandes caches de dados como feito pela CPU [14].

O direcionamento de transistores ao uso específico de processamento de dados, que ocorre na GPU, deve-se à característica paralela das aplicações gráficas que realizam a mesma operação em um grande volume de dados [15].

Neste trabalho, utilizamos a computação em GPU nas

operações de transformada espacial sobre os blocos de imagem do algoritmo VSTC, que possuem tamanho variável. Duas alternativas dentre as ferramentas que facilitam o desenvolvimento de paralelização em GPU são: CUDA e OpenACC.

O CUDA (*Compute Unified Device Architecture*) é uma arquitetura desenvolvida para as placas de vídeo da NVIDIA que permite utilizar a GPU tanto para computação em aplicações de uso geral quanto para as necessidades gráficas tradicionais. No entanto, na plataforma CUDA, o programador deve gerenciar a utilização das *threads* no código, definindo a quantidade de *threads* utilizadas, a quantidade de *threads* por bloco, e a quantidade de blocos. Também deve-se definir como as *threads* estão organizadas nos blocos, e como os blocos estão dispostos na grade. Assim, a implementação se torna mais complexa por causa dessas questões para essa aplicação específica sobre o algoritmo VSTC, e, por isso, nesse trabalho foi preferível a utilização do *OpenAcc* para a realização dos testes, uma vez que, teoricamente, a redução da complexidade é a mesma quando aplicada à rotina do algoritmo onde foi usada.

2.2.1 Hierarquia de memória

A GPU possui uma hierarquia de memória que pode ser utilizada com o objetivo de melhorar o desempenho das aplicações. As placas gráficas compatíveis com CUDA possuem quatro tipos de memórias: a memória global, a memória compartilhada, a memória constante e a memória de textura.

Essas placas gráficas possuem dezenas de multiprocessadores chamados *Streaming Multiprocessors* (SM). Cada multiprocessador possui uma memória compartilhada (*shared memory*), acessível por todas as *threads* de um bloco, além de duas memórias *cache* especiais, que são acessíveis somente para leitura, e de acesso rápido: a memória de textura (chamada de *texture cache*) e a memória constante (chamada de *constant cache*). Adicionalmente, é possível acessar a memória principal da GPU, chamada de *device memory*, onde reside a memória local de cada *thread* e a memória global da aplicação. É a memória com a maior latência de acesso [15].

Cada *thread* possui seus próprios registradores. Os registradores e a memória compartilhada oferecem os menores tempos de acesso na hierarquia de memória, suportando acessos paralelos a altas velocidades. Os registradores são utilizados pelas *threads* para armazenar suas próprias variáveis, chamadas de variáveis automáticas. Elas não são visíveis a outras *threads* e só existem enquanto a *thread* que a criou existir [5].

A memória compartilhada tem a serventia de promover a cooperação entre as *threads* de um mesmo bloco, proporcionando maior velocidade no acesso. Os dados armazenados em memória compartilhada têm o tempo de vida associado ao bloco, sendo visível por todas as *threads* nele contido [17].

A memória de textura proporciona uma otimização para os casos em que o problema possa ser tratado em duas ou três dimensões e que as informações de vizinhança tenham alguma dependência, como, por exemplo, quando o valor de um pixel de uma imagem depende do valor dos *pixels* ao seu redor [5].

O tempo de vida de uma variável está ligado ao tipo de memória onde está armazenada, podendo ser em memória global, constante, compartilhada, ou registradores. A visibilidade de uma variável está ligada ao escopo onde ela é criada, definindo assim quais *threads* poderão acessá-la. Caso uma *thread* utilize uma variável declarada dentro da função *kernel*, essa variável será armazenada em um registrador, e será criada uma cópia privada para cada *thread* que utilizar este *kernel*, ficando o seu tempo de vida restrito à *thread* [17].

2.2.2 OpenACC

CUDA é a abordagem mais madura para programação em GPU, mas é suportada somente em dispositivos NVIDIA. Apesar de ser parcialmente simples para construir um código usando essa ferramenta, alcançar uma boa taxa de desempenho requer geralmente um esforço de codificação e otimização perceptível.

O *OpenACC* (*open accelerators*) representa um esforço para criar uma interface de programação comum para dispositivos heterogêneos de grande adesão. É uma API (*Application Programming Interfaces*) que consiste em diretivas de compilador em regiões de código em C, C ++ e Fortran suscetíveis de serem executadas na GPU, desenvolvida para simplificar a programação paralela de CPU / GPU em sistemas heterogêneos.

As diretivas e o modelo de programação definidos permitem aos programadores criar aplicações capazes de utilizar aceleradores, sem a necessidade de gerenciar explicitamente dados ou transferências de programas entre o *host* (CPU) e o acelerador (GPU). Em vez disso, esses detalhes são implícitos no modelo de programação e são geridos pelos compiladores e os ambientes em tempo de execução.

O modelo de programação permite que o programador forneça mais informação disponível para os compiladores, incluindo a especificação local de dados que serão enviados para o acelerador, orientações sobre o mapeamento de *loops*, e outras informações relacionadas com o desempenho.

A *OpenACC* define uma extensa lista de *pragmas* (diretivas). Na execução de um loop em C, por exemplo, a diretiva (*#pragma acc kernels)* dá início à execução paralela no dispositivo acelerador, fazendo com que cada *loop* seja compilado em um *kernel* GPU separado e ao mesmo tempo, em vez de realizar o *loop* de forma sequencial.

Para o uso eficiente da GPU, transferências de dados entre o *host* e o acelerador devem ser minimizados, por isso a cópia de dados do *host* para o acelerador é feita na maioria dos casos para evitar essa transferências durante o uso da GPU. A diretiva (*#pragma acc data*) é utilizada para a cópia de memória para o acelerador.

3. Algoritmo VSTC

O algoritmo VSTC é um algoritmo de compressão de imagens, proposto em [1], cujos resultados superam algoritmos como o H.264/AVC [2] e MMP [3]. O seu esquema de codificação propõe um sistema multi-escala de blocos, cujas dimensões seguem um arranjo representado por uma árvore binária. O processo de codificação consiste na otimização recursiva dessa árvore binária, fazendo com que um grande número de blocos de várias dimensões sejam testados exaustivamente de forma recursiva até que o algoritmo defina a árvore ótima, através de um critério de otimização taxa-distorção, cujo objetivo é obter o bloco codificado com a menor taxa, porém com a menor distorção em relação ao bloco original.

Primeiramente, a imagem é dividida em blocos de dimensão 64x64, que serão otimizados e codificados sequencialmente. Cada um dos blocos 64x64 são otimizados utilizando-se duas funções recursivas que percorrem toda a árvore binária: a otimização do modo de predição e a otimização do resíduo.

Na otimização do modo de predição, o algoritmo testa dentre 35 modos aquele que fornece o melhor resultado em relação ao critério taxa-distorção. Uma vez escolhido o modo, obtém-se o resíduo do bloco, que é a diferença entre o bloco original e o bloco de predição. Esse bloco de resíduo também é otimizado recursivamente, percorrendo toda a árvore binária, de modo que o algoritmo verifica, por exemplo, se é mais vantajoso codificar para o mesmo conjunto de pixels, um bloco 4x4, dois blocos 2x1, dois blocos 1x2 ou quatro blocos 1x1.

A codificação tem por base a transformada espacial 2D, que, no algoritmo VSTC, consiste em uma combinação da Transformada do Cosseno Discreta [18] e a Transformada do Seno Discreta [19]. Depois da transformada, a maior parte da informação 2D da imagem fica representada por poucos coeficientes. Em seguida, os coeficientes são quantizados, utilizando um quantizador linear [19], e os coeficientes resultantes são codificados usando um codificador aritmético [20].

A natureza recursiva do algoritmo VSTC, que faz com que os blocos de entrada da imagem original sejam exaustivamente testados, seguindo a sequência de segmentação de uma árvore binária, torna o algoritmo um excelente alvo para um trabalho de paralelização, já que ele fornece excelentes resultados quando comparado a outros codificadores, porém tem um custo computacional alto.

4. Propostas de paralelização do algoritmo VSTC

Para realizar a paralelização do algoritmo VSTC, primeiramente analisamos as funções mais demoradas do algoritmo original, e, em seguida, avaliamos as etapas com maior potencial para paralelização.

4.1 Análise da execução do algoritmo VSTC

Para analisar a execução do algoritmo VSTC, foram utilizadas as ferramentas *Valgrind* e *Kcachegrind*. O *Valgrind* consiste em uma máquina virtual que utiliza técnicas de compilação *just-in-time* para executar um programa, o que torna a execução mais lenta. O objetivo da ferramenta é servir para a depuração, encontrando despejos de memória e realizando *profiling*. Já o *Kcachegrind* é uma ferramenta que serve para visualização dos dados gerados pelo *profiling*. Foram executados três testes em modo *debug*, com parâmetro de entrada $QP=28$ (que estabelece o passo de quantização), utilizando essas ferramentas, com as imagens "small" (256x256) [21], "RaceHorses" (416x240) e "Lena" (512x512) [22].

Os resultados do *Valgrind*, mostrados nas Figs. 1, 2 e 3, associam a cada função, listada na coluna *Function*, o número de vezes que essa função foi chamada na coluna *Called*, os tempos em porcentagem do total da execução do programa em que a função estava sendo executada na coluna *Self* e, ainda, o tempo em que a função estava na pilha de execução do programa na pilha *Incl.*

A análise das saídas do Valgrind indica que um conjunto de quatro funções é responsável pela maior parte do tempo de execução. São elas: *opt_blk_and_pred_mod*, *block*, *Transform_Block* e *Inv_Transform_Block*. Juntas, elas corresponderam a 73%, 62% e 75% respectivamente dos tempos de execução de cada teste. Portanto, seria interessante que o paralelismo atuasse sobre esse conjunto de funções.

As quatro funções listadas anteriormente pertencem todas ao mesmo fluxo de chamada de funções, que tem início na função *optimize_block_and_pred_mode*. Portanto, a estratégia de paralelização deve atuar nessa função para obter um resultado eficiente.

As funções *Transform_Block* e *Inv_Transform_Block* são responsáveis pela aplicação das transformadas espaciais, diretas e inversas, respectivamente. Já as funções *optimize_block_and_pred_mode* e *optimize_block* são as duas funções recursivas de otimização do VSTC, que otimizam o modo de predição e o resíduo, respectivamente.

A função de otimização do modo de predição atua sobre cada bloco da imagem e pode ser dividida em duas etapas: na primeira, ela testa todos os possíveis modos de predição, e, para cada um deles, faz uma chamada da função que otimiza o resíduo. Na segunda etapa, a função subdivide o bloco em quatro menores, quando possível, dois através de uma segmentação horizontal e dois através de uma segmentação vertical, e executa o mesmo procedimento de forma recursiva. Em seguida, determina se codificar os blocos menores separadamente resulta em obter uma compressão mais eficiente do que o bloco original, não particionado.

Incl.	Self	Called	Function
100.00	0.00	1	main
99.95	0.07	153	opt_blk_and_pred_mod
94.76	17.95	112 932	opt_blk_and_pred_mod'2
75.87	0.26	1 303 091	optimize_block
68.29	5.63	265 465 788	optimize_block'2
36.83	36.23	190 254 714	Transform_Block
20.93	20.31	190 254 714	Inv_Transform_Block
6.99	0.60	266 768 879	Cabac_Coding_Blk_temp
5.62	4.72	18 406 651 295	free_residue_tree
3.66	3.41	1 732 695 029	biari_encode_symbol
3.20	0.90	115 011 193	write_sig_coeff_tmp
2.08	0.72	89 706 522	write_sig_map_tmp
1.62	0.20	267 250 958	malloc
1.43	1.43	268 518 650	_int_malloc

Fig. 1 – Saída do Valgrind para a imagem "Small".

Incl.	Self	Called	Function
100.00	0.00	1	main
99.96	0.05	476	opt_blk_and_pred_mod
95.67	14.10	351 344	opt_blk_and_pred_mod'2
81.18	0.19	5 614 001	optimize_block
75.01	4.12	1 065 614 900	optimize_block'2
28.25	0.45	1 071 228 901	Cabac_Coding_Blk_temp
26.11	25.68	762 556 092	Transform_Block
20.72	20.72	1 071 228 901	__memcpy_avx_unaligned
18.81	18.35	762 556 092	Inv_Transform_Block
4.30	3.71	79 004 470 121	free_residue_tree
4.18	3.80	10 839 351 387	biari_encode_symbol
3.79	1.11	758 825 893	write_sig_coeff_tmp
2.65	0.94	585 069 909	write_sig_map_tmp
1.40	0.16	942 943 532	unary_exp_golomb_enc

Fig. 2 – Saída do Valgrind para a imagem "Race Horses".

Incl.	Self	Called	Function
100.00	0.00	1	main
99.96	0.05	1 088	opt_blk_and_pred_mod
95.82	14.54	803 072	opt_blk_and_pred_mod'2
80.57	0.20	13 807 348	optimize_block
74.79	4.17	2 586 819 498	optimize_block'2
29.56	0.46	2 600 626 846	Cabac_Coding_Blk_temp
26.35	25.92	1 850 486 139	Transform_Block
24.03	24.03	2 600 626 846	__memcpy_avx_unaligned
16.44	15.98	1 850 486 139	Inv_Transform_Block
4.48	3.82	194 249 883 296	free_residue_tree
2.95	2.73	18 385 315 950	biari_encode_symbol
2.55	0.82	1 551 221 721	write_sig_coeff_tmp
1.93	0.69	1 212 081 717	write_sig_map_tmp

Fig. 3 – Saída do Valgrind para a imagem "Lena".

A função *optimize_block_and_pred_mode* atua de forma sequencial sobre cada bloco 64x64 da imagem de entrada, porém não é possível paralelizar os bloco de entrada para

serem codificados simultaneamente, pois, de acordo com o esquema de predição, o resultado de cada blocos depende dos blocos codificados anteriormente.

Para cada modo de predição, é realizada a otimização do resíduo e determinado o custo do resíduo de forma recursiva, para em seguida determinar qual dentre os modos de predição apresenta o menor custo. Serão discutidas possibilidades de paralelizar essa determinação do bloco de menor custo.

4.2 Implementação do paralelismo utilizando MPI

A partir da identificação de que a primeira parte da função *optimize_block_and_pred_mode* e as funções que esta invoca em sua execução são responsáveis por um grande percentual de tempo de execução do algoritmo, surge a possibilidade de implementar um paralelismo nesta função. A figura 4 ilustra o algoritmo de determinação do modo de predição que resulta no menor custo do bloco, com sua execução de forma sequencial.

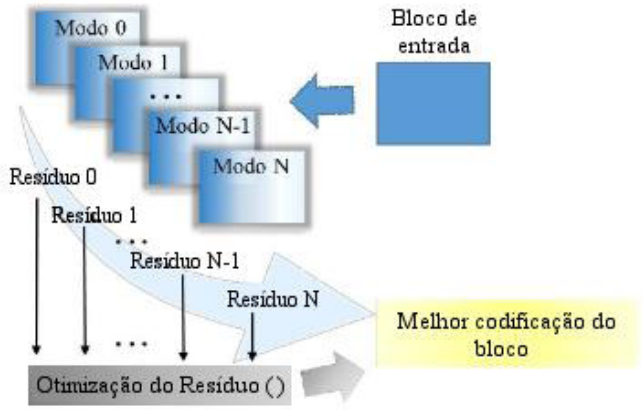

Fig. 4 – Algoritmo sequencial da determinação do melhor modo de intra-predição.

Cada modo de predição em um mesmo bloco é calculado de forma independente dos demais, apenas dependente dos blocos vizinhos decodificados anteriormente, e o resultado de cada um é utilizado no final para determinar o melhor modo de predição. Portanto, uma possibilidade de paralelismo consiste em dividir essa análise de cada modo de predição, com um processo sendo responsável por cada modo de predição. A figura 5 ilustra uma proposta de execução paralela desse algoritmo.

Na figura 5, é exposto que cada processo vai calcular um bloco de resíduo para um modo de predição específico e determinar o seu custo. Em seguida, os processos devem se comunicar para determinar qual deles que obteve o menor custo.

Para realizar essa divisão do algoritmo em processos e comunicação entre eles, utiliza-se o padrão MPI da seguinte forma: após cada processo ter analisado um modo de predição, que é específico para cada processo e depende do *rank* do processo, é obtido um custo do bloco, correspondente à codificação desse modo. Em seguida, os processos se comunicam para determinar qual deles obteve o melhor resultado, através da função *MPI_AllReduce*. A execução dessa função encontra-se ilustrada na figura 6.

A função *MPI_AllReduce*, fornecida pelo MPI, implementa operações de redução. Ela combina valores fornecidos no *buffer* de entrada de cada processo e, usando operações específicas, retorna o valor resultante no *buffer* de saída de todos os processos [10]. A figura 6 ilustra o funcionamento da *MPI_AllReduce* combinada com duas funções distintas. Cada processo inseriu no *buffer* de entrada um vetor com quatro inteiros. Na parte superior, ela foi executada com a operação *MPI_MIN* e a função retornou no *buffer* de saída de cada processo os mínimos para uma mesma posição do *buffer* de entrada. Na parte inferior, ela foi executada com a operação *MPI_SUM*, e retornou, no *buffer* de saída de cada processo, a soma dos valores.

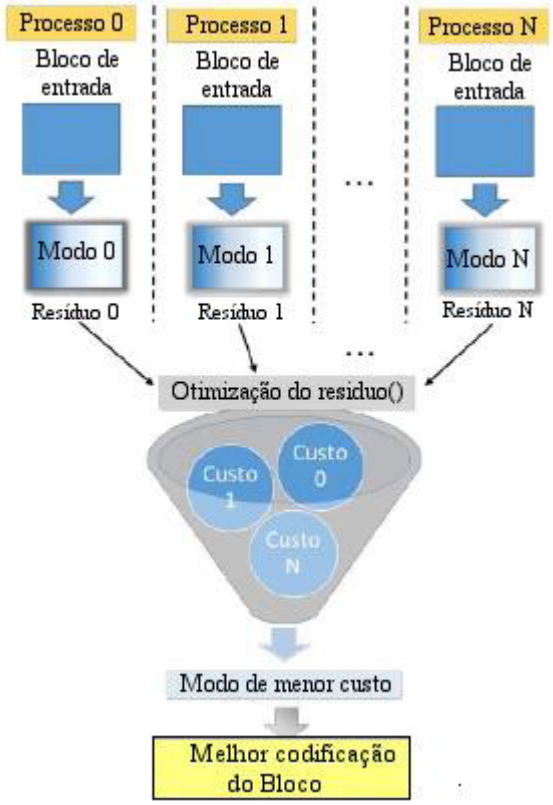

Fig. 5 – Algoritmo paralelo de determinação do melhor modo de intra-predição.

No caso do algoritmo VSTC, a operação *MPI_AllReduce* foi invocada em conjunto com a operação *MIN_LOC*, que retorna qual a posição do menor elemento. Essa posição corresponde ao modo de predição utilizado e, a partir desse modo, cada processo pode determinar a melhor codificação do bloco.

Seguindo a sequência de otimização recursiva do algoritmo VSTC, cada processo precisa recalcular a otimização do bloco de resíduo, fazendo as partições vertical e horizontal do bloco calculado anteriormente, sendo que este último já tem a definição do melhor modo de predição.

Fig. 6 – Exemplo do MPI_AllReduce [10].

Ao todo, são utilizados 35 modos de predição no algoritmo VSTC, porém alguns dos blocos de entrada não possuem

todos os modos de predição disponíveis, como por exemplo aqueles blocos correspondentes às bordas da imagem, os quais não possuem blocos vizinhos de onde seriam feitas as predições.

A função *optimize_block_and_pred_mode* é a principal função do algoritmo, pois, nela, são chamadas todas as outras funções que fazem a codificação do bloco de imagem. Por isso, ao paralelizar essa rotina, espera-se uma redução significativa no tempo total de execução do algoritmo.

4.3 Implementação do paralelismo utilizando a GPU

A partir dos resultados do *Valgrind,* detectou-se que as funções que aplicam as transformadas espaciais, *Tranform_Block* e *Inv_Transform_Block*, consomem bastante tempo da execução do programa. Nos três testes realizados, elas corresponderam a 43,8%, 37,3% e 52,1% do tempo total de execução.

A função *Transform_Block* tem como entrada a matriz *MxN* com os valores de resíduo (a diferença pixel a pixel entre os blocos da imagem original e da predição), e tem como saída uma matriz de mesma dimensão, contendo os coeficientes da transformada. Para uma matriz quadrada, complexidade dessa função é $O(N^3)$ e executa duas multiplicações de matrizes *MxN*: da matriz do bloco de entrada com a matriz da transformada, em seguida, o resultante dessa primeira multiplicação com a inversa da matriz da transformada.

Assim como a função *Transform_Block*, a função *Inv_ Transform_Block Block* tem a mesma complexidade e executa duas transformações de matrizes, mas em ordem inversa do que ocorre na função *Transform_Block*. Ou seja, a entrada é a matriz com os coeficientes da transformada e a saída é o bloco decodificado, que representa uma aproximação do bloco de imagem original.

A plataforma de programação paralela *OpenAcc* permite a introdução de diretivas no código de modo a paralelizar algumas rotinas, utilizando a capacidade de processamento da GPU. A multiplicação de matrizes é um caso em que a introdução do paralelismo na GPU tem resultados expressivos na redução do tempo de execução, de modo que cada elemento da matriz resultante da multiplicação de matrizes é calculado por meio de uma única *thread*, e todos os elementos da matriz resultante são calculados simultaneamente. A implementação do paralelismo *OpenAcc* no código do VSTC ocorreu através da inserção das diretivas nos cálculos das transformadas espaciais. Podemos estimar que essas diretivas vão resultar em um ganho de processamento no cálculo dessas transformadas. Portanto, as acelerações podem variar dependendo de quanto tempo essas funções consomem para imagens distintas.

4.4 Arquitetura do Cray XK7

Para a realização dos experimentos, foi utilizado o Cray *XK7,* uma plataforma de supercomputação, que foi produzida pela *Cray* e lançada em 2012. A plataforma utiliza uma combinação de unidades de processamento central (CPUs) e unidades de processamento gráfico para computação (GPUs). As máquinas *XK7* executam o ambiente de Linux do *Cray*.

O modelo utilizado contém 10 *blades* de 4 nós cada, totalizando 40 nós, que estão interligados através da interconexão *Gemini*. Cada nó possui uma CPU *AMD Opteron 6200 Interlagos* de 16 *cores*, uma GPU *Nvidia Tesla K20 Kepler* e 32 GB de memória.

A programação no *Cray* é dividida por ambientes. Nos experimentos realizados, a versão com MPI foi executada no ambiente GNU e a versão com *OpenAcc* foi executada no ambiente *Cray.*

5. Resultados

O algoritmo VSTC original e as duas versões implementadas foram executadas no conjunto de imagens mostrado na Tabela 1, utilizando como parâmetro de entrada QP = 28. Os experimentos foram realizados no Cray XK7, utilizando o compilador GCC. As imagens utilizadas pertencem ao *test set* do HEVC [22], com exceção da imagem *Small* [21].

A Tabela 1 exibe a dimensão, o número de blocos de entrada 64x64, e o tempo de codificação de cada imagem, e o número final de blocos, resultante das partições feitas pela otimização do algoritmo VSTC. O tempo de execução é diretamente proporcional à dimensão da imagem, ou seja, imagens de dimensões maiores consomem mais tempo para serem codificadas.

Imagem	Dimensão	No Blocos 64x64	Tempo de Execução	No Blocos Final
Small	192x192	9	10min55s	571
RaceHorses	416x240	28	62min45s	6145
BQSquare	416x240	28	68min29s	13655
BlowingBubbles	416x240	28	62min29s	7453
BasketBallPass	416x240	28	53min32s	4891
Keiba	832x480	104	238min33s	10984
FlowerVase	832x480	104	158min42s	5645
Mobisode	832x480	104	120min0s	1452
PartyScene	832x480	104	428min44s	69802
BQMall	832x480	104	284min39s	22917

Tab 1: Resultados dos testes referentes ao algoritmo VSTC original.

A implementação do paralelismo usando o padrão MPI foi testada nas mesmas imagens e nas mesmas condições dos testes da Tabela 1, utilizando também o parâmetro de entrada QP=28.

Da análise dos dados presente nas tabelas 1 e 2, percebe-se que, para todas as imagens, obtemos ganhos significativos de redução do tempo de processamento, e a implementação obteve uma média de 88% de redução do tempo de processamento em relação ao algoritmo VSTC original.

Imagem	Tempo de Execução	Speed-up	Redução
Small	1min21s	8,09	88%
RaceHorses	6min49s	9,21	89%
BQSquare	7min29s	9,15	89%
BlowingBubbles	6min43s	9,30	89%
BasketBallPass	5min50s	9,18	89%
Keiba	26min32s	9,99	89%
FlowerVase	25min39s	6,19	84%
Mobisode	19min34s	6,13	84%
PartyScene	48min10s	8,90	89%
BQMall	31min40s	9,99	89%

Tab 2 - Resultados dos testes da implementação com o MPI.

A média de *speed-up* foi de 8,41, que está condizente com a *Lei de Amdahl*, exposta na seção 2. De fato, considerando

que o *speed-up* resultante é obtido sobre a função *optimize_block_and_pred_mode*, que chama todas as funções de codificação do algoritmo, pode-se observar que, ao paralelizá-la, obtém-se uma redução significativa no tempo total de execução do algoritmo.

O *speed-up* obtido variou mais entre as imagens. Comparando-se as Tabelas 1 e 2, percebe se que não há relação direta entre o *speed-up* e a dimensão da imagem. As duas imagens que apresentaram os piores resultados de aceleração foram *FlowerVase* e *Mobisode*. Na Tabela 1, nota-se que essas duas imagens possuem um número de partição bastante inferior às outras imagens de mesma dimensão. Isso ocorre devido à existência de grandes regiões uniformes na imagem, como podemos ver na figura. 7, que mostra a imagem *FlowerVase*. Ou seja, regiões onde os *pixels* têm valores muito próximos aos *pixels* vizinhos. Nessas regiões, o processo de otimização ocorre mais rápido, pois, como não há diferença entre particionar ou não um bloco, o processo recursivo não se prolonga até os blocos menores. Isso explica porque nessas três imagens o resultado de *speed-up* foi inferior aos das demais.

Fig. 7 – Imagem de teste "FlowerVase".

As principais dificuldades da implementação do MPI consistiram na implementação da comunicação entre os processos, que ocorreu através da função *MPI_AllReduce* e na escrita em arquivos, pois o algoritmo escreve diversas informações da codificação em arquivos de saída, porém como diversos processos estavam executando o mesmo código, eles estavam em concorrência pelo acesso ao recurso. Assim, o código foi alterado para que somente um processo escrevesse a saída em arquivo.

Da mesma forma, a implementação do *OpenAcc* nas funções *Transform_Block* e *Inv_Transform_Block* do algoritmo VSTC foi testada nas mesmas imagens e sob as mesmas condições. A Tabela 3 mostra os resultados obtidos com a paralelização em GPU, utilizando *OpenAcc*.

A redução do tempo na maioria das imagens foi próxima de 30%. A redução média do tempo foi de aproximadamente 32,3%. Podemos notar novamente que a diferença na resolução das imagens não interferiu diretamente na redução do tempo de codificação. Esse resultado é bastante condizente com a análise da execução do algoritmo VSTC exposta na seção 4.1, que indicou que as transformadas espaciais consumiam entre 30% e 50% do tempo total de processamento.

As diferenças de *speed-up* na codificação das imagens não foram muito altas quando comparadas aos resultados obtidos usando o MPI, pois a paralelização com GPU só foi feita nos *loops* das multiplicações de matrizes nas rotinas das transformadas direta e inversa. Destaca-se ainda que que, na implementação com GPU, as imagens *Small*, *FlowerVase* e *Mobisode* (que têm mais regiões uniformes) apresentaram os melhores resultados de *speed-up*.

Imagem	Tempo de Execução	Speed-up	Redução
Small	7min6s	1,54	35%
RaceHorses	43min17s	1,45	31%
BQSquare	48min31s	1,41	29%
BlowingBubbles	43min3s	1,45	31%
BasketBallPass	35min55s	1,49	33%
Keiba	166min29	1,43	30%
FlowerVase	99min56s	1,59	37%
Mobisode	71min45s	1,67	40%
PartyScene	208min6s	1,37	27%
BQMall	191min43s	1,42	30%

Tab 3: Resultados obtidos utilizando OpenAcc.

No desenvolvimento do código paralelizado, ao se copiar as matrizes do *host* (CPU) para o acelerador (GPU), e, ao fim da execução, copiar as matrizes de saída do acelerador para o *host*, pode-se notar um aumento do tempo de execução do algoritmo, devido a matriz representativa do bloco ter dimensões muito pequenas (as dimensões da largura e da altura da matriz são potências de 2 e têm tamanho máximo de 64 *pixels*, e mínimo de 1 *pixel*). Assim, o processamento não paralelizado tem a vantagem de não possuir o tempo de cópia de memória, apenas a execução na GPU. A solução adotada foi não copiar as matrizes a serem usadas no acelerador, de modo que o acesso dos dados fosse feito na memória principal da CPU, o que tornou mais rápida a execução de ambas as funções.

Para validação dos resultados, verificamos que as implementações dos paralelismos não modificaram os valores de taxa e distorção obtidos com o algoritmo VSTC original, ou seja, a relação sinal-ruído (PSNR), o erro quadrático médio (MSE) e a taxa de compressão, em bit por pixel.

Não há outros trabalhos relacionados sobre a paralelização do algoritmo VSTC, de forma que uma comparação com trabalhos de otimização sobre outros algoritmos de compressão de imagem pode não ser uma avaliação eficiente, pois as estruturas de codificação desses algoritmos são diferentes bem como as condições de teste, como o hardware utilizado e as imagens testadas. Porém, destacamos para uma comparação subjetiva o trabalho de paralelização do HEVC publicado em [23], onde foram obtidos resultados de *speed-up* de 8,7 a 10,7. Os valores médios obtidos são próximos àqueles obtidos neste trabalho.

6. Conclusões

Desenvolvemos duas versões paralelizadas do algoritmo de compressão de imagens VSTC com o objetivo de otimizar o seu tempo de execução. Duas abordagens foram estudadas e implementadas, a primeira utilizando MPI, e a segunda utilizando *OpenAcc*.

Ambas as implementações de paralelismo reduziram o tempo de execução do algoritmo. O código resultante da implementação MPI obteve os melhores resultados, o que já era esperado, pois a implementação do paralelismo usando o MPI englobava uma maior parte do código do algoritmo.

A implementação utilizando MPI teve como resultado um aumento de rapidez média no tempo de execução de 8,41 vezes, em relação ao algoritmo original. A implementação utilizando *OpenAcc* resultou em um algoritmo 1,41 vezes mais rápido na média dos testes. Utilizando a *Lei de Amdahl* para analisar os resultados obtidos, foi possível concluir que quase 90% do tempo total de execução do algoritmo VSTC é paralelizável.

O desempenho de codificação do algoritmo nas duas novas implementações foi o mesmo em relação ao código original, validando ambos as implementações, de forma que essas contribuições tornaram o algoritmo VSTC mais rápido, auxiliando o mesmo a competir com os demais padrões de compressão de imagens existentes, possibilitando ainda futuras evoluções.

Agradecimentos

Esse trabalho foi financiado pelo Projeto CAPES/Pró-Defesa 23038.009094/2013-83.

7. Referências Bibliográficas

[1] A. V. C. Oliveira, N. M. M. Rodrigues , S. M. M. Faria , E. A. B. da Silva, C. L. Pagliari, Image coding using variable-size transforms and a full binary tree recursive optimization, International Telecommunications Symposium, 17-20 Aug. 2014, Sao Paulo, Brazil.

[2] ITU-T and ISO/IEC JTC1, Advanced video coding for generic audiovisual services, ITU-T Recommendation H.264 and ISO/IEC 14496-10 (MPEG-4 AVC), 2013.

[3] M. B. Carvalho, E. A. B. da Silva, and W. A. Finamore, "Multidimensional Signal Compression using Multiscale Recurrent Patterns," Signal Processing, Special Edition in Image Coding Beyond Standards, Elsevier, pp. 1559–1580, 2002..

[4] D. Castano-Diez, D. Moser, A. Schoenegger, S. Pruggnaller, A. S. Frangakis, Performance evaluation of image processing algorithms on the GPU. Journal of Structural Biology, 2008.

[5] E. S. COSTA, Implementação em GPU do algoritmo de compressão de imagens baseado em recorrência de padrões multiescala, o MMP, Universidade Federal Fluminense, Niterói-RJ. 2013.

[6] M. Boaventura, Programação paralela usando MPI. UNESP. São José do Rio Preto - SP. 2008.

[7] A. Grama, G. Karypis, V. Kumar, A. Gupta, Introduction to Parallel Computing, Addison Wesley, 2nd Ed.,2003.

[8] D. R. Cheng, V. B. Shah, J. R. Gilbert, A. Edelman, A Novel Parallel Sorting Algorithm for Contemporary Architectures, 2007.

[9] M. F. Su, I. El-Kady, D. A. Bader, S. Y. Lin, A Novel FDTD application featuring OpenMP-MPI Hybrid Parallelization. International Conference on Parallel Processing – ICPP. 2004.

[10] A. A. V. Ignácio, V. J. M. F. Filho. MPI: Uma ferramenta para implementação paralela. UFRJ - Universidade Federal do Rio de Janeiro. Pesquisa Operacional, v22, n.1, 2002.

[11] M. J. Quinn, Parallel Programming in C with MPI and OpenMP. McGraw-Hill Education, 2008.

[12] K. Ganeshamoorthy, Parallel Algorithms on Configurable Hybrid UPC/MPI Clusters. M.Sc. Thesis. University of Colombo School of Computing. 2010.

[13] A. S. Tanenbaum, M. V. Steen, Distributed Systems: Principles and Paradigms. 2. ed. Prentice Hall, 2006.

[14] T. V. Luong, N. Melab, E. G. Talbi, GPU-Based Multi-start Local Search Algorithms. 5th International Conference on Learning and Intelligent Optimization, Rome and Italy, 2011.

[15] NVIDIA, Cuda C Programming Guide. www.nvidia.com: Morgan Kaufmann, 2015.

[16] D. B. Kirk, W. W. Hwu, Programando para processadores Paralelos. Uma abordagem prática à programação de GPU, Elsevier Brasil, 2010.

[17] M. M. Collares, Um resolvedor numérico baseado no método de Lattice-Boltzmann aplicado em unidades de Processamento gráfico. Dissertação de Mestrado. Instituto Militar de Engenharia, 2012.

[18] N. Ahmed, T. NATARAJAN, K. R. Rao, Discrete Cosine Transforn, IEEE Transactions on Computers, C-23(1):90–93, January 1974.

[19] A. K. Jain Fundamentals of Digital Image Processing, Prentice Hall, 1989.

[20] I. Witten, R. NEAL, J. CLEARY. Arithmetic coding for data compression, Communications of the ACM, 30(6):520–540, 1987.

[21] S. M. M. Faria, Very low bit rate video coding using geometric transfom motion, Ph.D. Thesis. , University of Essex, June 1996.

[22] Joint Collaborative Team on Video Coding (JCT-VC), Document: JCTVC-A200, Meeting report of the first meeting of the Joint Collaborative Team on Video Coding (JCT-VC), Dresden, DE, 15-23 April, 2010.

[23] C. C Chi, M. A. Mesa et al.: Parallel Scalability and Efficiency of HEVC Parallelization Approaches. IEEE Transactions on Circuits and Systems for Video Technology. Vol. 22. No. 12, 2012.