MINISTÉRIO DA DEFESA
EXÉRCITO BRASILEIRO
DEPARTAMENTO DE CIÊNCIA E TECNOLOGIA
INSTITUTO MILITAR DE ENGENHARIA
CURSO DE GRADUAÇÃO EM ENGENHARIA DA COMPUTAÇÃO

BRUNO RAMOS CAMPANA
MATHEUS LEAL ASSUNÇÃO
VINICIUS GOMES PEREIRA

OTIMIZAÇÃO DO ALGORITMO VSTC ATRAVÉS DO USO DA ARQUITETURA DO CRAY XK7

Rio de Janeiro
2015

INSTITUTO MILITAR DE ENGENHARIA

BRUNO RAMOS CAMPANA
MATHEUS LEAL ASSUNÇÃO
VINICIUS GOMES PEREIRA

OTIMIZAÇÃO DO ALGORITMO VSTC ATRAVÉS DO USO DA ARQUITETURA DO CRAY XK7

Projeto de Final de Curso apresentado ao Curso de Engenharia de Computação do Instituto Militar (IME) de Engenharia, como parte das exigências do IME.

Orientadoras: Prof ª. Raquel Coelho Gomes Pinto, D.Sc., do IME e Profª. Carla Liberal Pagliari, Ph.D., do IME.

INSTITUTO MILITAR DE ENGENHARIA
Praça General Tibúrcio, 80-Praia Vermelha
Rio de Janeiro-RJ CEP 22290-270

Z34r Campana, B.R., Assunção, M.L., Pereira, V.G.
OTIMIZAÇÃO DO ALGORITMO VSTC ATRAVÉS DO USO DA ARQUITETURA DO CRAY XK7/ Bruno Ramos Campana, Matheus Leal Assunção e Vinicius Gomes Pereira. - Rio de Janeiro: Instituto Militar de Engenharia, 2015.
95 p.: il.
Projeto de Fim de Curso (PFC) - Instituto Militar de Engenharia - Rio de Janeiro, 2015.
1. Compressão de imagens. 2. Algoritmo VSTC. 3. Programação paralela. 4. Programação em GPU.
I. Pinto, Raquel Coelho Gomes.
II. Pagliari, Carla Liberal
III. Instituto Militar de Engenharia.

CDD 627.3902

MINISTÉRIO DA DEFESA
EXÉRCITO BRASILEIRO
DEPARTAMENTO DE CIÊNCIA E TECNOLOGIA
INSTITUTO MILITAR DE ENGENHARIA
CURSO DE GRADUAÇÃO EM ENGENHARIA DA COMPUTAÇÃO

BRUNO RAMOS CAMPANA
MATHEUS LEAL ASSUNÇÃO
VINICIUS GOMES PEREIRA

OTIMIZAÇÃO DO ALGORITMO VSTC ATRAVÉS DO USO DA ARQUITETURA DO CRAY XK7

Projeto de Final de Curso apresentado ao Curso de Engenharia de Computação do Instituto Militar (IME) de Engenharia, como parte das exigências do IME.

Aprovado em 27 de maio de 2015 pela seguinte Banca Examinadora:

_____

Raquel Coelho Gomes Pinto - D.Sc., do IME

_____

Carla Liberal Pagliari – Ph.D., do IME

_____

Anderson Fernandes P. dos Santos - D.Sc., do IME

_____

Julio Cesar Duarte - D.Sc., do IME

RIO DE JANEIRO

2015

AGRADECIMENTOS

A nossa orientadora Raquel Coelho pelo incentivo, apoio e empenho que tornaram possível a conclusão desta monografia.

À Anderson Oliveira, autor do algoritmo VSTC, por seu amplo apoio e compartilhamento de sua pesquisa.

À professora Carla Pagliari pelo suporte no pouco tempo que lhe coube, por suas correções, incentivos e por seu grande bom humor diário.

A nossas famílias que sempre nos apoiaram durante toda nossa formação.

E a todos os nossos amigos que nos incentivaram na realização deste trabalho.

SUMÁRIO

LISTA DE ILUSTRAÇÕES 8

RESUMO 9

ABSTRACT 10

1 INTRODUÇÃO 11

1.1 MOTIVAÇÃO 12

1.2 OBJETIVO 12

1.3 JUSTIFICATIVA 13

1.4 METODOLOGIA 13

1.5 ORGANIZAÇÃO DA MONOGRAFIA 14

2 COMPRESSÃO DE IMAGENS 15

2.1 COMPRESSÃO DE IMAGENS 15

3 PADRÕES DE COMPRESSÃO 29

3.1 PADRÃO H.264/AVC 29

3.2 PADRÃO H.265 (HEVC) 30

3.3 ALGORITMO MMP 31

3.4 ALGORITMO VSTC 32

4 PROCESSAMENTO PARALELO 36

4.1 INTRODUÇÃO AOS PROCESSADORES PARALELOS 36

4.2 PARADIGMA DE PROGRAMAÇÃO MPI 41

4.3 COMPUTAÇÃO EM GPU 44

4.3.1 INTRODUÇÃO CUDA 45

4.3.2 HIERARQUIA DE MEMÓRIA 49

4.3.3 INTRODUÇÃO AO OPENACC 50

5 PROPOSTAS DE PARALELIZAÇÃO DO ALGORITMO VSTC 53

5.1 ANÁLISE DA EXECUÇÃO DO ALGORITMO VSTC 53
5.2 IMPLEMENTAÇÃO DO PARALELISMO UTILIZANDO MPI 57
5.3 IMPLEMENTAÇÃO DO PARALELISMO UTILIZANDO A GPU 61
5.4 ARQUITETURA DO CRAY XK7 62
6 RESULTADOS DAS IMPLEMENTAÇÕES PROPOSTAS 63
7 CONCLUSÃO 68
8 REFERÊNCIAS BIBLIOGRÁFICAS 70
9 APÊNDICES 74
9.1 APÊNDICE A – IMPLEMENTAÇÃO EM CUDA 74
10 ANEXOS 79
10.1 ANEXO A –FIGURA UTILIZADAS NOS TESTES 79

LISTA DE ILUSTRAÇÕES

Figura 2.1 - Primeira imagem digital da história. 15
Figura 2.2 - Protótipo de câmera digital da Kodak 15
Figura 2.3 - Exemplos de compressão de imagem Lena 17
Figura 2.4 - Codificação por *Run Length* 18
Figura 2.5 - Codificação de Huffman 19
Figura 2.6 - Codificação LZW 20
Figura 2.7 - Diferentes formatos de subamostragem de crominância 21
Figura 2.8 - Esquema de compressão por transformada 22
Figura 2.9 - Exemplo de resíduo em codificação de vídeo 26
Figura 3.1 - Modos de predição do padrão H.264/AVC 30
Figura 3.2 - Modos de intra-predição do padrão H.265 31
Figura 3.3 - Sequência de multiescalar de partições do bloco 33
Figura 3.4 - Sequência de varredura de diferentes blocos 34
Figura 3.5 - Novas partições do bloco 35
Figura 4.1 - Modelo computacional SSID 37
Figura 4.2 - Modelo computacional SIMD 38
Figura 4.3 - Modelo computacional MISD 38
Figura 4.4 - Modelo computacional MIMD 39
Figura 4.5 - Gráfico da Lei de Amhdal 41
Figura 4.6 - Modelo de processos no MPI 42
Figura 4.7 - Diferença das filosofias de projeto entre a CPU e a GPU 44
Figura 4.8 - Crescimento diferencial de performance entre CPU e GPU 45
Figura 4.9 - Linguagens e APIs suportadas pelo CUDA 46
Figura 4.10 - Exemplo de um *grid* com seus blocos e *threads* 47
Figura 4.11 - Soma de dois vetores utilizando a forma sequencial e o paralelismo na GPU. 48
Figura 4.12 - Tipos de memórias na arquitetura CUDA 49
Figura 4.13 - Exemplo de utilização do *OpenAcc* na execução paralela de um *loop*. 51
Figura 4.14 - Exemplo de cópia de memória entre *host* e GPU. 52
Figura 5.1 - Resultado da execução no Valgrind do algoritmo para a imagem (a) *Small* 53
Figura 5.2 - Resultado da execução no Valgrind do algoritmo para a imagem (b) RaceHorses 54
Figura 5.3 - Resultado da execução no *Valgrind* do algoritmo para a imagem (b) Lena 54
Figura 5.4 - Sequência de chamadas de funções para teste utilizando a imagem (a) *Small* 55
Figura 5.5 - Ilustração da primeira parte da função *optimize_block_and_pred_mode* 57
Figura 5.6 - Algoritmo sequencial da determinação do melhor modo de intra-predição 58
Figura 5.7 - Algoritmo paralelo de determinação do melhor modo de intra-predição 59
Figura 5.8 - Exemplo do MPI_AllReduce 60

RESUMO

A compressão de imagens tem papel fundamental na economia de memória de armazenamento e de banda de transmissão de imagens, tornando possível a utilização de diversas tecnologias de multimídia atuais que trabalham com imagens de alta-definição.

O algoritmo *Variable Size Transform Coder* (VSTC) é um novo algoritmo de compressão de imagens, que combina o uso de transformadas espaciais e funções recursivas de otimização com um esquema de segmentação adaptativo, similar ao proposto pelo algoritmo MMP. O VSTC tem apresentado bons resultados de compressão, em termos de taxa-distorção, melhores que algoritmos como o H.264/AVC, JPEG e o MMP. No entanto, sua elevada complexidade computacional torna o tempo de execução do algoritmo impraticável.

Com o objetivo de agilizar a execução do algoritmo, esse trabalho visa implementar diferentes formas de paralelização do código do VSTC. O paralelismo está cada vez mais constante nas diversas escalas de computação, com o intuito básico de reduzir o tempo total de processamento. Para possibilitar que o paralelismo obtenha ganhos mais significativos em tempo de execução se faz necessário um uso adequado dos recursos disponíveis e adaptações do software.

Por isso, nesse trabalho foram estudados o padrão de comunicação MPI e as ferramentas CUDA e *OpenAcc*, relacionadas à programação em GPU, para implementação de *software* paralelo. Após a análise do desempenho do algoritmo VSTC foram implementadas versões do algoritmo utilizando MPI e *OpenAcc*. As duas implementações foram testadas em imagens de dimensões variando de 192 x 192 *pixels* a 832 x 480 *pixels*, obtendo 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.

Assim, esse trabalho contribuiu para melhorar o tempo de execução do algoritmo de compressão de imagens VSTC. A abordagem MPI obteve melhores resultados, tornando o algoritmo aproximadamente 8,13 vezes mais rápido, em relação a implementação original.

Palavras chave: Compressão de imagens. Algoritmo VSTC. Programação paralela. Programação em GPU.

ABSTRACT

Image compression has a fundamental role in saving memory storage and image transmission bandwidth, making it possible to use various current multimedia technologies which work with high-definition images.

The algorithm Variable Size Transform Coder (VSTC) is a new image compression algorithm, which combines the use of spatial transforms and a recursive optimization procedure with an adaptive segmentation scheme, similar to that proposed by the MMP algorithm. The VSTC has consistently outperformed algorithms such as H.264/AVC and the MMP in image coding, however, its high computational complexity makes the algorithm runtime impractical.

In order to improve the execution time of the algorithm, this work aims to implement different forms of parallelization of the VSTC code. Parallelism is constant increasinging in different scales of computing, with the basic aim of reducing the total processing time. To enable parallelism to get more significant gains at runtime it is necessary the proper use of available resources and software adaptations.

In this work we studied the communication standard MPI and CUDA and OpenACC tools, related to GPU programming, to implemente the parallel software. After an analysis of the VSTC algorithm performance, different versions were implemented using MPI and OpenACC. Both implementations were tested in images with sizes ranging from 192 x 192 pixels to 832 x 480 pixels, obtaining an average reduction of runtime, from the original algorithm, of 32.3% in the version with OpenACC and 87.7% in the version with MPI.

Thus, this work helped to improve the runtime of the VSTC image compression algorithm. The MPI approach yielded better results, making the algorithm approximately 8.13 times faster, compared to the original implementation.

Keywords: Image Compression. VSTC algorithm. Parallel programming. GPU programming.

1 INTRODUÇÃO

Tradicionalmente, 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.

Atualmente, o desenvolvimento de computadores mais rápidos tem sido motivado por novas aplicações comerciais que requerem um processador capaz de trabalhar grandes quantidades de dados de maneira sofisticada. Estas aplicações incluem as videoconferências, ambientes de trabalho colaborativos, diagnósticos auxiliados por computador na medicina, bases de dados paralelos utilizadas para o apoio à tomada de decisão, processamento gráfico avançado e realidade virtual.

A computação paralela vem sendo explorada nas principais atividades de alto desempenho. É uma forma de computação em que muitos cálculos são realizados em simultâneo, que operam no princípio 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 impedem 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 é um novo algoritmo de compressão de imagens, que tem apresentado bons resultados de compressão, em termos de taxa-distorção, melhores que algoritmos como o H.264/AVC e o MMP [1]. Entretanto, o tempo de execução do algoritmo é muito maior do que os demais. A computação paralela é uma das alternativas para diminuir o tempo de compressão da

imagem. Para isso, faz-se necessário o estudo das técnicas de paralelismo disponível atualmente, de modo a serem implementadas no código do VSTC.

1.1 MOTIVAÇÃO

Devido aos atuais limites físicos de aumento da frequência do *clock* das CPUs, os desenvolvedores de hardware passaram a adotar arquiteturas de múltiplos cores, e dessa forma, a maioria dos computadores vendidos comercialmente já possuem mais de um núcleo.

Nesse contexto, a programação paralela tem ganho destaque como uma nova abordagem de programação que visa explorar a eficiência da arquitetura de múltiplos cores.

Além disso, a melhoria nos processadores gráficos devido ao processamento paralelo nas placas de vídeo, despertou o interesse de pesquisadores em explorar o potencial computacional das GPUs em programas gerais.

A conferência "*Revitalizing Computer Architecture Research*", realizada nos EUA, que aborda os principais desafios da área de arquitetura de computadores destacou a popularização da programação paralela como um dos principais assuntos [2].

Dessa forma, para acompanhar as tendências de hardware, uma melhor compreensão sobre os potenciais e as aplicações da programação paralela se faz cada vez mais necessário.

1.2 OBJETIVO

O objetivo da presente dissertação 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 a execução do algoritmo em unidades de processamento gráfico (GPU).

Serão avaliados os ganhos de desempenho entre as versões paralelas e em relação a versão sequencial, comparando-se os tempos totais de processamento de cada versão.

1.3 JUSTIFICATIVA

O algoritmo *Variable Size Transform Coder* (VSTC) é um algoritmo de compressão de imagens que combina a multiescalaridade do algoritmo MMP com transformadas espaciais [1]. Estudos mostram que o VSTC tem superado tanto o algoritmo MMP quanto o padrão H.264/AVC na codificação de imagens. No entanto, novas propostas de alterações no algoritmo para que este possa competir com o padrão H.265 (HEVC), prolongam o tempo de execução do algoritmo, de forma que a aplicação dessas mudanças torna infactível a utilização do algoritmo.

Com o intuito de reduzir esse tempo de processamento para que o VSTC se torne competitivo discutiu-se a possibilidade de paralelização do algoritmo, pois a programação paralela tem sido bastante utilizada nos últimos anos com o intuito de acelerar o tempo total de processamento de diversos programas.

Dentro do Projeto Estratégico de Defesa Cibernética foi adquirido um supercomputador Cray XK7. Este supercomputador oferece duas arquiteturas de processamento paralelo tendo em vista que cada nó possui uma GPU. Sendo assim, este trabalho oferece a oportunidade de avaliar os dois tipos de paralelismo oferecidos pelo Cray. O uso de GPU pode ser avaliado tendo em vista que o algoritmo realiza o mesmo processamento para cada pixel da imagem e o modelo de programação da GPU é justamente executar em paralelo cada instrução em cima de um dado diferente.

1.4 METODOLOGIA

Para a realização desse trabalho, os seguintes passos foram traçados:

- Estudo do processo de compressão de imagens e do código fonte do algoritmo VSTC.
- Estudo de computação paralela, incluindo a comunicação de processos através do MPI, a programação para GPU e a arquitetura do Cray XK7.
- Proposta e discussão de possíveis paralelismos no código fonte do VSTC.
- Implementação de duas versões paralelas do algoritmo executando com comunicação MPI e processamento em GPU.

- Realização de testes das duas implementações coletando os resultados de tempo de execução e desempenho da compressão.
- Análise dos testes efetuados e comparação com a execução serial do algoritmo.

1.5 ORGANIZAÇÃO DA MONOGRAFIA

Este trabalho encontra-se organizado da seguinte forma: no capítulo 2, são discutidos conceitos básicos sobre a compressão de imagens e, no capitulo 3, são apresentados os padrões de compressão H.264/AVC, H265(HEVC) e os algoritmos MMP e o VSTC.

O capítulo 4 aborda computação paralela apresentando conceitos sobre a comunicação MPI e a arquitetura de placas gráficas, assim como os modelos de programação utilizados em cada um.

Em seguida, o capítulo 5 discute as duas soluções de paralelismo propostas e detalha as alterações do código do VSTC propostas para realizar o paralelismo em cada uma das duas versões.

No capítulo 6, são apresentados os resultados obtidos de cada implementação.

Por fim, as conclusões e considerações finais são expostas no capítulo 7.

2 COMPRESSÃO DE IMAGENS

2.1 COMPRESSÃO DE IMAGENS

Uma imagem digital é uma imagem bidimensional composta de *pixels*, que emprega um código binário para permitir o seu processamento, transferência, impressão ou reprodução. A primeira imagem digital da história foi criada em 1957 por Russell Kirsch, era uma imagem de 176x176 *pixels* de seu filho, como mostra a figura 2.1, e foi feita através de um dispositivo que transformava imagens em matrizes de zeros e uns [3].

Figura 2.1 - Primeira imagem digital da história. Retirado de [3]

A baixa resolução da figura era decorrente da baixíssima capacidade de armazenamento do computador que operava o primitivo escâner. Já o primeiro protótipo de câmera digital foi criado pela Kodak, somente em 1975 e encontra-se ilustrado na figura 2.2:

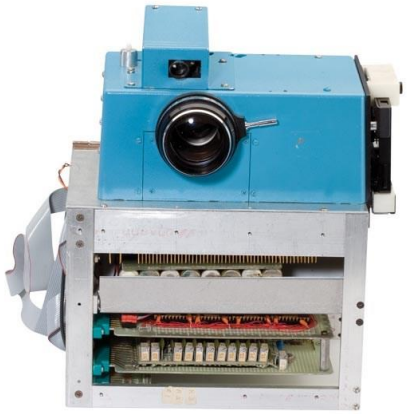


Figura 2.2– Protótipo de câmera digital da Kodak. Retirado de [4]

Conforme a figura 2.2, a primeira câmera digital era baseada num CCD, dispositivo de carga acoplada e pesava cerca de 8 kg. Ela foi construída com um sensor de 100x100 *pixels* e gravava as imagens em uma fita cassete.

A partir de então, o desenvolvimento e a demanda por produtos de multimídia vêm crescendo cada vez mais rápido, contribuindo para problemas de banda de rede insuficiente para a transmissão e de memória insuficiente para armazenamento. Nesse contexto, a teoria de compressão de dados tem se tornado cada vez mais significante para reduzir a redundância de dados [5].

Dessa forma, diversas tecnologias atuais tais como câmeras digitais, televisão digital, DVD, integração de computadores e vídeos, streamings de vídeo pela internet e outros dependem de compressão eficiente de imagens e vídeos [6].

Alguns benefícios da compressão incluem economias de transmissão, armazenamento e processamento de informações, além de redução na probabilidade de erros de transmissão, pois uma quantidade menor de dados é transmitida, e também garante um certo nível de segurança contra espionagens [7].

A Associação de Radiologistas Canadenses reconheceu em um estudo que o crescente volume de informações geradas por novas modalidades de exames como tomografia computadorizada multi-corte e ressonância magnética requisitam o uso de técnicas irreversíveis de compressão que diminuam o custo de armazenamento e melhorem a eficiência de transmissão em rede, com o respaldo de que não ocorra perda de informação clínica relevante [8].

A compressão de imagens tem um papel fundamental na transmissão e no armazenamento dessas informações. O principal objetivo da compressão de imagens é o de representar a imagem com uma quantidade menor de bits sem perder a informação essencial da imagem original. Para atingir esse objetivo, as técnicas de compressão atuam na redução de redundâncias e de informações irrelevantes [9]. No caso de imagens, alguns aspectos que são explorados incluem redundâncias entre *pixels* adjacentes, o emprego de técnicas de codificação mais eficientes e características do sistema visual humano [6].

A compressão de imagens pode ocorrer com perdas (*lossy*) ou sem perdas (*lossless*). Na compressão com perdas é aceitável perder parte da informação para obter uma taxa de compressão maior. Já na compressão sem perdas, após o processo de descompressão a imagem obtida é igual a imagem original [10].

Compressões sem perdas conseguem oferecer reduções de tamanho de em média 3 vezes, as compressões irreversíveis ou com perdas permitem taxas maiores de compressões com perdas da qualidade visual da imagem dependendo da taxa de compressão, podendo ser utilizadas na compressão de imagens médicas. [3] A figura 2.3 apresenta uma imagem original Lena e duas versões comprimidas com diferentes taxas de compressão.

(a) Imagem - Lena original - 12 KB

(b) Imagem - Lena comprimida - 1.8 KB

(c) Imagem - Lena comprimida - 0.56 KB

Figura 2.3 – Exemplos de compressão de imagem Lena – Adaptado de [11]

Analisando a figura 2.3, nota-se que, entre as imagens (a) e (b), as diferenças resultantes do processo de compressão praticamente não interferem na qualidade perceptual da imagem. A imagem (b) apresenta uma compressão de 85%. Já a imagem (c) claramente apresenta uma distorção elevada, isso devido a compressão mais elevada, de 95%. Em geral, na compressão de imagens deve-se equilibrar esses dois fatores: qualidade da imagem resultante e taxa de compressão. Algumas situações requerem taxas de compressão maiores em detrimento da qualidade da imagem final, como por exemplo, onde a capacidade de memória é bastante limitada ou a banda de transmissão é pequena.

Compressões sem perdas são altamente desejáveis em aplicações onde as imagens são posteriormente sujeitas a mais processamentos, intensas edições ou compressões e descompressões repetitivas. Também é usada em geral quando a obtenção da imagem foi bastante custosa ou em aplicações onde a qualidade desejada da imagem final ainda não é conhecida. [12]

Esquemas de compressão sem perdas adotam em geral duas fases distintas e independentes: modelagem e codificação. A fase de modelagem visa coletar informações sobre a imagem, na forma de modelos probabilísticos que serão usados na codificação. Por fim, são aplicadas técnicas de codificação de entropia para comprimir os dados. [12]

Algumas técnicas de compressão sem perdas são: [9]

- Codificação por carreira ou *Run Length Encoding* (RLE):

É um método de compressão simples usada para dados sequenciais e útil no caso de informações repetitivas. Essa técnica substitui sequências de símbolos idênticos, chamados runs, por símbolos menores. Para uma imagem em escala de cinza, o RLE é representado por uma sequência {Ri, Vi}, onde Vi é o valor da intensidade do *pixel* e Ri é o número de vezes consecutivas que esse valor se repete. A figura 2.4 ilustra um exemplo de codificação por carreira:

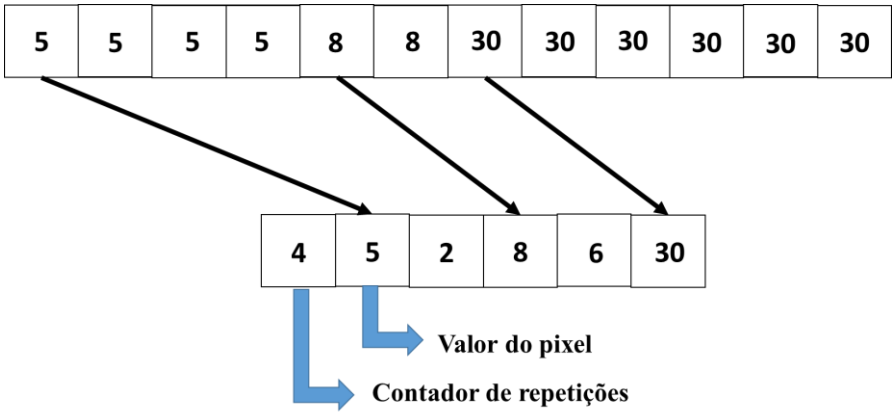
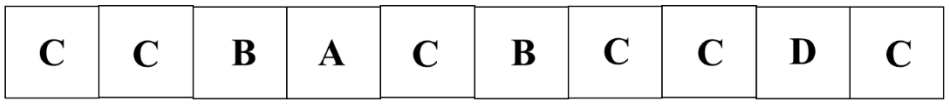


Figura 2.4 – Codificação por *Run Length*

Nota-se na figura que essa codificação gera melhores resultados quando há grandes sequências de valores idênticos. Considerando-se a representação em binários, a representação normal do exemplo da figura necessita de 50 *bits* enquanto a representação codificada utiliza somente 20 *bits*, o que corresponde a uma taxa de compressão de duas vezes e meia.

- Codificação de Huffman ou *Variable Length Code* (VLC):

Essa técnica codifica os símbolos de acordo com suas frequências estatísticas de ocorrências. Os valores que ocorrem com maior frequência são representados com quantidades inferiores de bits, enquanto os símbolos que aparecem com menor frequência são assignados a representações com mais bits. Na figura 2.5, se encontra ilustrado um exemplo de codificação de Huffman, representado pela mensagem original, a tabela de frequência e a árvore de Huffman.



Símbolo	Código	Frequência
A	100	10%
B	11	20%
C	0	60%
D	101	10%

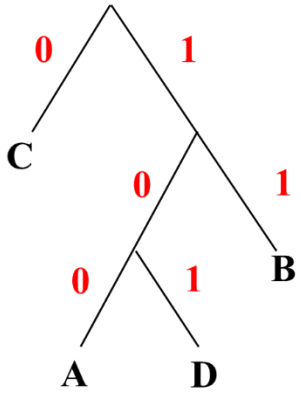


Figura 2.5 – Codificação de Huffman

Na tabela da figura 2.5, observa-se que o símbolo C que apresenta a maior frequência de ocorrência é representado com o menor código binário. Assim, caso todos os símbolos da mensagem original fossem codificados com 2 bits, seriam utilizados 20 bits. Através da codificação de Huffman é possível representar a mesma mensagem com somente 14 bits, o que corresponde a uma redução de 30%.

- Codificação LZW:

LZW (Lempel-Ziv-Welch) é uma codificação baseada em dicionário, podendo ser estática ou dinâmica, dependendo se o dicionário é fixo ou se é atualizado durante a execução do processo. A figura 2.6 ilustra um exemplo de codificação LZW.

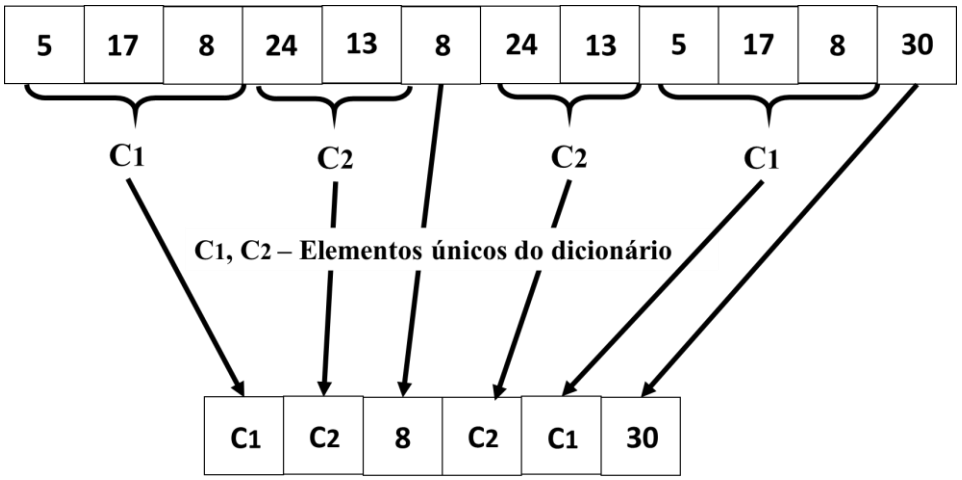


Figura 2.6 – Codificação LZW

Na figura 2.6, observa-se que os símbolos C1 e C2, elementos pertencentes ao dicionário, substituem cadeias de símbolos. Assim, o resultado da compressão dependerá do dicionário utilizado.

Nos três exemplos nota-se que no final o propósito da codificação é de obter uma forma mais eficaz de representar a mesma informação, o que é a base da compressão sem perdas.

Já na compressão com perdas, o objetivo é obter taxas de compressão maiores. Para isso, precisam atuar na eliminação de redundâncias e informações irrelevantes. Informações irrelevantes são aspectos da imagem que muitas vezes não são perceptíveis para a visão humana. Diversos experimentos sobre os aspectos da visão humana concluíram que o olho humano não responde com a mesma sensitividade a todo tipo de informação visual, assim alguns aspectos da informação são mais perceptíveis que outros. Vários algoritmos de compressão de imagens exploram essa característica.

Um exemplo de informação irrelevante decorre do fato de que a visão humana é bem mais sensível as alterações de luminância do que de crominância, assim é possível amostrar os componentes de crominância com taxas inferiores aos componentes de luminância, resultando em boas compressões que não geram perdas nas características de percepção visual da imagem.

A luminância representa o brilho recebido da luz, que é proporcional a energia na banda visível. Já a crominância descreve o tom de cor percebido da luz, que depende da composição do comprimento de onda da luz e é caracterizado por dois atributos: matiz, que especifica o tom da cor e depende do pico do comprimento de onda da luz, e saturação, que descreve quão pura a cor é e depende do espalhamento da banda no espectro da luz. [6]

O modelo RGB primário de representação de cores mescla os atributos de crominância e luminância. Porém, para a compressão é desejável descrever uma cor em termos de sua luminância e sua crominância separadamente para permitir processamento e transmissão mais eficientes. Um sistema de coordenadas pode ser facilmente relacionado ao outro através da transformação linear da equação 2.1.1:

$$\begin{pmatrix} Y \\ Cb \\ Cr \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.334 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} + \begin{pmatrix} 0 \\ 128 \\ 128 \end{pmatrix} \qquad \text{Equação 2.1.1}$$

Onde R, G e B são os valores dos componentes de cada *pixel* no modelo RGB.

Os componentes Y, Cb e Cr de uma imagem definem um sistema de coordenadas de cores, onde Y representa como luminância e Cr, Cb são os componentes de crominância. Para explorar as características do sistema visual humano, existem diversos formatos de taxa de subamostragem de crominância, conforme a figura 2.7.

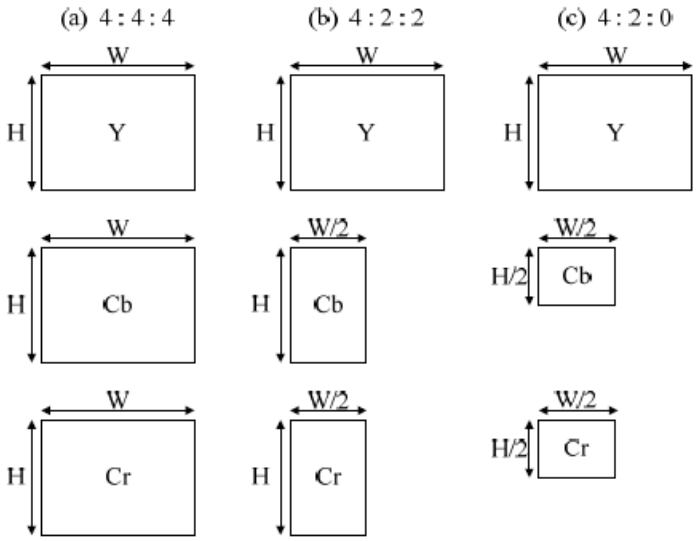


Figura 2.7 – Diferentes formatos de subamostragem de crominância [5]

Conforme a figura 2.7, vemos três formatos principais: [5]

- Formato 4:4:4 – A taxa de amostragem da luminância é a mesma dos componentes da crominância.
- Formato 4:2:2 – Para cada quatro amostras de luminância existem duas para cada componente da crominância. Reduz pela metade o número de *pixels* por linha, mas mantém o número de linhas por frame.

- Formato 4:2:0 – Amostra os componentes de crominância pela metade tanto na vertical quanto na horizontal. Assim é utilizada uma amostra em cada componente da crominância para cada quatro amostras de luminância.

No decodificador, as taxas de amostragem inferiores são convertidas novamente para o formato 4:4:4. O formato 4:2:0 é o mais comum, pois sua utilização gera uma maior taxa de compressão sem grandes perdas na qualidade da imagem.

Já as redundâncias na compressão de imagens estáticas ocorrem para determinadas condições, pois, em geral, os *pixels* de uma imagem não são de fato independentes, pois existe uma correlação entre os *pixels* de uma mesma vizinhança. Essa redundância é dita redundância espacial [7]. Para explorar essas redundâncias os codificadores utilizam diversos métodos como a codificação preditiva e a aplicação de transformadas ortogonais que descorrelatem a informação na imagem. A aplicação de transformadas é bastante utilizada em diversos padrões adotados universalmente, como por exemplo o JPEG [7].

2.2 CODIFICAÇÃO POR TRANSFORMADA

O processo de compressão de imagens na codificação por transformada atua subdividindo a imagem em blocos menores de tamanho N x N e baseia-se na ideia de que, com a aplicação da transformada unitária, a energia da imagem se concentrará em regiões de baixa frequência e em poucos coeficientes da transformada. Assim, é possível descartar informações na representação dos demais coeficientes, sem grandes perdas na qualidade da imagem, em geral essa representação como menos bits ocorre através do processo de quantização. Assim, os compressores de imagens que utilizam transformadas em geral seguem o esquema da figura 2.8.

Figura 2.8– Esquema de compressão por transformada

Na figura 2.8, destaca-se que se pode dividir o processo de compressão de imagens em três etapas principais.

No mapeador ocorre a aplicação da transformada ortogonal, com o objetivo de descorrelatar os dados da imagem para em que seguida, a quantização, aplicada no quantizador, resulte em conjunto mais homogêneo de valores, de forma que a codificação, que será aplicada na sequência, consiga resultar em uma compressão mais eficiente.

A quantização é o processo de reduzir o número de possíveis valores de uma grandeza, assim, reduzindo o número de bits necessário para representá-la. A quantização é um processo que introduz perdas e, portanto, diminui a quantidade de informação associada a imagem. O objetivo da quantização é reduzir a precisão para obter uma taxa de compressão maior [9].

Existem dois modelos de quantização: a escalar e a vetorial. A quantização escalar é a mais comum e consiste basicamente em aplicar uma função que mapeia um escalar (uma dimensão) em um valor de saída. Alguns exemplos simples incluem arredondar valores decimais para o inteiro mais próximo ou para um certo múltiplo de uma dada unidade de precisão.

Já a quantização vetorial é uma técnica clássica de quantização no processamento de sinais que permite modelar funções de densidade de probabilidade. Sua aplicação original foi a compressão de imagens. Ela atua dividindo um grande conjunto de pontos (vetores) em grupos menores, e, através de levantamentos estatísticos determina os vetores mais apropriados para formar a base da quantização. [7]

O codificador de entropia aplica as técnicas expostas de compressão sem perdas, tentando obter uma forma ótima de representar a informação associada ao resultado do processo de quantização. [9]

Então, ficou evidente que o resultado da transformada é fundamental para o resultado final da compressão. Pois quanto mais a transformação conseguir descorrelatar os dados, melhores serão os resultados da quantização e da codificação. Assim, existem várias características desejáveis em uma transformada com o propósito da compressão de imagens. Dentre elas [13]:

- Descorrelação da imagem: a transformada deve descorrelatar a informação contida em um bloco, acumulando uma grande parte da energia em poucos coeficientes.
- Base de funções independente da imagem: Em geral, como ocorrem grandes variações estatísticas entre imagens, uma transformada ótima depende da imagem. Porém, encontrar uma base de funções para essa transformada a partir da imagem é uma tarefa computacionalmente extensiva. Além disso, a transmissão da base de funções compromete a eficiência da compressão. Dessa forma, é desejável um

trade-off entre a performance da transformada e a independência entre a base de funções e a imagem.

- Implementação rápida: o número de operações necessárias para uma transformada de N pontos é em geral da ordem de $O(N^2)$. Algumas transformadas conseguem ser implementadas de forma mais eficiente reduzindo a complexidade de uma transformada de um bloco N x N em duas dimensões de $O(N^4)$ para apenas $O(2N^2 \log N)$.

Algumas transformações importantes na compressão de imagens são [13]:

- Transformada de Karhunen-Loève (KLT):

A KLT é a transformada ótima no quesito de compactação de energia, pois ela consegue reter uma maior parcela de energia em uma quantidade menor de coeficientes do que as outras transformadas. No entanto, a base de funções da KLT é dependente da imagem e obter essas funções é um processo computacionalmente extensivo. Portanto, não existe um algoritmo rápido que implemente a KLT. Por isso, seria interessante enviar a base de funções junto com a imagem codificada para que pudesse ser utilizada no decodificador, no entanto, esse envio de informação extra compromete o resultado final da compressão [13].

- Transformada Discreta de Fourier (DFT):

A DFT é a versão discreta da transformada de Fourier, assim é a versão adotada para cálculos computacionais, sendo utilizada na análise espectral e na filtragem. Para um bloco f de n x n *pixels* a DFT em duas dimensões é definida pela equação 2.2.1.

$$F(u,v) = \frac{1}{n} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} f(j,k) e^{-\frac{2\pi i(uj+vk)}{n}} \quad (2.2.1)$$

Onde *F(u,v)* representa o valor do *pixel* no domínio da transformada, *n* é uma dimensão do bloco, *f(j,k)* é o valor do *pixel* no domínio da imagem e *i* é a unidade imaginária. A transformada inversa, definida na equação 2.2.2, utiliza as mesmas variáveis.

$$f(j,k) = \frac{1}{n} \sum_{u=0}^{n-1} \sum_{v=0}^{n-1} f(u,v) e^{\frac{2\pi i(uj+vk)}{n}} \quad (2.2.2)$$

A DFT essencialmente decompõe o bloco de imagem nos seus componentes espectrais e os índices u e v são chamados frequências espaciais da transformada. A função exponencial pode

ser decomposta em duas parcelas e assim a DFT em duas dimensões pode ser computada como duas DFTs em uma dimensão cada.

É possível implementar uma versão rápida da DFT, que requer O ($N.\log_2 N$). No entanto, os coeficientes da transformada gerados pela DFT são complexos, assim, seu armazenamento e manipulação se tornam mais complicados.

- Transformada Discreta de Cosseno (DCT)

A transformada discreta do cosseno (DCT) é uma técnica para converter um sinal em seus componentes elementares de frequência. Foi desenvolvida por Ahmed, Natarajan e Rao e apresentada em 1974 [14], sendo similar a transformada discreta de Fourier (DFT). Sua aplicação na compressão de imagens foi inicialmente abordada por Chen e Pratt somente em 1984, mas hoje ela é amplamente usada na compressão [15].

A fórmula que define a aplicação da DCT em duas dimensões encontra-se exposta na equação 2.2.3, e a inversa da transformada é descrita na equação 2.2.4. [16]

$$F(u,v) = \frac{4C(u)C(v)}{n^2} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} f(j,k) \cos\left[\frac{(2j+1)u\pi}{2n}\right] \cos\left[\frac{(2k+1)v\pi}{2n}\right] \quad (2.2.3)$$

$$f(j,k) = \sum_{u=0}^{n-1} \sum_{v=0}^{n-1} C(u)C(v)F(u,v) \cos\left[\frac{(2j+1)u\pi}{2n}\right] \cos\left[\frac{(2k+1)v\pi}{2n}\right] \quad (2.2.4)$$

Onde a maioria das variáveis é equivalente as utilizadas na equação 2.2.1 da DFT, exceto pelo mapeamento $C : \mathbb{Z} \rightarrow \mathbb{R}$ que pode ser obtido de acordo com a equação 2.2.5.

$$C(w) = \begin{cases} \frac{1}{\sqrt{2}} & Para\ w = 0 \\ 1 & Para\ w = 1, 2, \ldots, n-1 \end{cases} \quad (2.2.5)$$

A DCT é a transformada utilizada na maioria dos codificadores por bloco, devido a duas características importantes: conservação de energia e compactação de energia. A transformada DCT é unitária e, portanto, conserva a energia da imagem, além disso, ela consegue uma alta compactação de energia para informações altamente correlatadas. Ainda, como a DCT é ortogonal, a implementação de sua inversa é praticamente idêntica à da transformação direta. [16]

Outra propriedade da DCT é que a maior parte da energia da imagem é concentrada nos coeficientes de baixa frequência. [17]

- Transformada de Walsh-Hadamard (WHT) [13]:

A transformada WHT não apresenta grandes resultados de compactação de energia quando comparada, por exemplo, com a DCT. No entanto, ela se tornou popular, especialmente para implementações de *hardware*, por ser de fácil implementação e ainda possuir uma modesta capacidade de descorrelatar dados. A base de funções da WHT pode é obtida através das linhas das matrizes ortonormais de Hadamard. A equação 2.2.6 contém a menor matriz de Hadamard de tamanho 2x2.

$$H_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{2.2.6}$$

Outra técnica bastante empregada na compressão de imagens é a da codificação preditiva, que, em geral, tenta predizer o valor do *pixel* atual a partir dos valores dos *pixels* já calculados. Ela se baseia no fato de que a amplitude de uma amostra é grande, mas a diferença de amplitude entre amostras sucessivas é relativamente pequena. Assim, por exemplo no DPCM (Differential Pulse Code Modulation), ao invés de codificar cada *pixel*, opta-se por codificar a diferença entre o valor do *pixel* e o valor do *pixel* anterior [18]. A essa diferença chamamos de resíduo, normalmente essas técnicas preditivas são bastante utilizadas na codificação de vídeos, onde se é calculado o resíduo entre frames consecutivos conforme pode ser visto na figura 2.9 [19].

Figura 2.9 – Exemplo de resíduo em codificação de vídeo [20]

De acordo com a figura 2.9, a ideia da utilização do resíduo consiste basicamente em transmitir o resíduo e os vetores de movimentos ao invés de todo o frame atual.

Para avaliar o resultado de uma compressão são utilizadas medições como o erro quadrático médio (MSE) e a relação sinal-ruído (PSNR), que são definidos de acordo com as equações (2.2.7) e (2.2.8) abaixo [5]:

$$MSE = \sqrt{\frac{\sum_{x=0}^{W-1}\sum_{y=0}^{H-1}\ [f(x,y)-f'(x,y)]^2}{WH}} \tag{2.2.7}$$

$$PSNR = 20 * \log_{10}\frac{255}{MSE} \tag{2.2.8}$$

Onde, na equação 2.2.7, os valores de *f(x,y)* e *f '(x,y)* são o valor do *pixel* na imagem original e na imagem decodificada, respectivamente, e W e H são os valores de largura e altura da imagem. Na equação 2.2.8, o valor de MSE é o calculado na equação anterior. As duas medições são inversamente proporcionais, para obter uma compressão de melhor qualidade, é desejável reduzir o MSE e consequentemente aumentar o PSNR.

Portanto, o processo de compressão de imagens deve atuar equilibrando dois fatores: taxa de compressão e a qualidade da imagem codificada. Por isso, a teoria da taxa-distorção descreve o *trade-off* entre a taxa de compressão e a distorção resultante.

A teoria da taxa-distorção atua em processos de otimização que visam minimizar a distorção para uma dada taxa de compressão ou minimizar a taxa de compressão para um dado valor de distorção. Esses dois problemas utilizam uma mesma formulação lagrangeana, exposta na equação 2.2.9.

$$J = D + \lambda R \tag{2.2.9}$$

Onde J é o custo lagrangeano, D é a distorção, R é a taxa de compressão e λ é dito o fator lagrangeano, e pode ser manipulado para equilibrar o *trade-off* entre a taxa e a distorção.

Assim, foram estudadas diversas técnicas que são utilizadas na compressão de imagens. Em geral, várias técnicas são combinadas e adaptadas criando os padrões de compressão.

Diversos formatos de compressão foram estabelecidos pela ITU-T e pela ISO/IEC ao longo dos anos, visando melhorar a eficiência da compressão ou obter compressões com menores distorções. Alguns padrões consagrados são o H.264/AVC, o MMP e o H.265, mas, para uso na

internet, os dois formatos de compressão mais comuns são o JPEG – *Joint Photographic Experts Group*, o GIF – *Graphics Interchange Format* e o PNG – *Portable Network Graphics*. [9]

Nas próximas seções serão abordados os padrões H.264/AVC, H.265 e o algoritmo MMP, para em seguida, introduzir o algoritmo *Variable Size Transform Coder* (VSTC) [1], discutindo como os padrões já estabelecidos influenciaram o desenvolvimento do algoritmo.

3 PADRÕES DE COMPRESSÃO

3.1 PADRÃO H.264/AVC

H.264/AVC é um padrão aberto e licenciado que é compatível com as técnicas mais eficientes de compactação de vídeo disponíveis atualmente. Um codificador H.264 pode reduzir o tamanho de um arquivo de vídeo digital em mais de 80% em comparação com o formato Motion JPEG e em até 50% mais do que o tradicional padrão MPEG-4 parte 2. Isso significa menos largura de banda de rede e espaço de armazenamento são necessários para um arquivo de vídeo. Ou visto de outra forma, pode ser atingida uma qualidade de vídeo superior para uma determinada taxa de bits [21].

H.264/AVC contém uma série de novas funcionalidades que o permitem comprimir vídeo com muito mais eficiência do que os padrões mais antigos. Na codificação dos quadros dos vídeos, o H.264 apresenta técnicas de compressão de imagens muito eficientes.

O padrão de compressão de imagem H.264 apresenta um algoritmo para comprimir sem perdas (CABAC – *Context Adaptive Binary Arithmetic Coding),* que comprime os dados de forma mais eficiente, mas requer muito mais processamento para decodificar, baseado em uma aritmética de codificação adaptativa ao contexto. Além disso, implementa uma quantização de maneira mais eficiente. De modo que um novo passo de quantização logarítmica é utilizado. Também é possível usar as matrizes de quantização de frequência personalizadas selecionadas pelo codificador [22]

O padrão H.264 utiliza, também, um método de predição multidirecional para reduzir a redundância espacial usando blocos vizinhos como base. A previsão *intra-frame* oferece nove modos de predição para blocos 4x4, nove modos de predição para blocos 8x8 e quatro modos de predição para blocos 16x16. H.264/AVC usa a técnica de otimização de taxa de distorção (RDO) para obter o melhor modo de codificação dos nove modos de predição em termos de maximização da qualidade da codificação, minimizando a taxa de bits. Isto significa que o codificador tem de codificar a imagem experimentando completamente todos os nove modo de predição. A figura 3.1 ilustra os modos de predição do H.264/AVC.

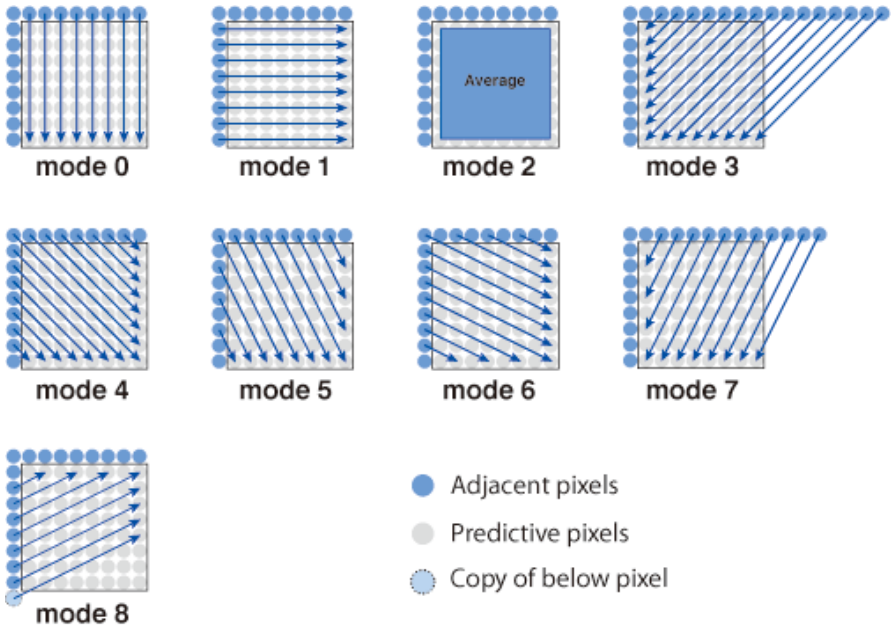


Figura 3.1– Modos de predição do padrão H.264/AVC [23]

O melhor modo é o que tem a taxa de distorção de custo mínimo. A fim de calcular o custo para cada modo, as mesmas operações da transformada, da quantização e da codificação, direta e inversa, são repetidamente realizadas. Todo esse processamento explica a alta complexidade de cálculo do custo. Portanto, a complexidade computacional do codificador é elevada [24].

3.2 PADRÃO H.265 (HEVC)

HEVC (*High Efficiency Video Coding*), ou H.265, é um padrão de compressão de vídeo, um sucessor para H.264/AVC. HEVC foi concebido para melhorar substancialmente a eficiência de codificação em comparação com o H.264/AVC, isto é, reduzir a taxa de bits para metade com qualidade de imagem comparável, à custa do aumento da complexidade computacional. Os principais recursos onde HEVC foi melhorado em comparação com o H.264/AVC foram o suporte para vídeo de resolução superior e melhores métodos de processamento paralelo. HEVC será utilizado em telas HDTV de última geração e sistemas de captura de conteúdo que apresentam taxas de quadros progressivos digitalizados e resoluções de tela de QVGA (320 x 240 *pixels*) até 8192 x 4320 *pixels*, bem como uma melhor qualidade de imagem em termos de nível de ruído, cor, e alcance dinâmico [25].

Diferentemente do H.264/AVC, o H.265 utiliza blocos iniciais de tamanho 64 x 64 e utiliza 36 modos de intra-predição que encontram-se ilustrados na figura 3.2.

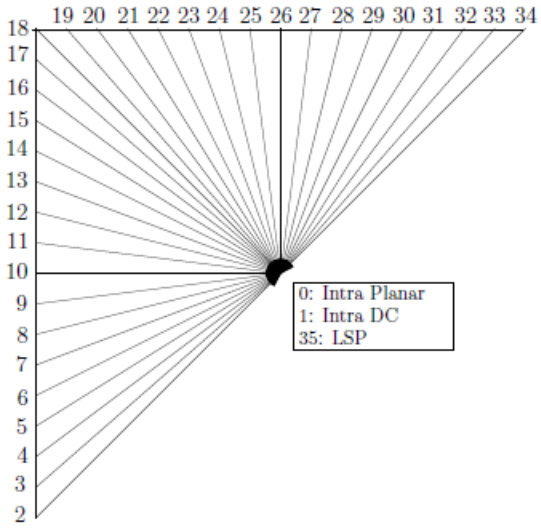


Figura 3.2 - Modos de intra-predição do padrão H.265 [25]

3.3 ALGORITMO MMP

O algoritmo MMP (*Multidimensional Multiscale Parser*) baseia-se no casamento de padrões multiescalas. Ele codifica um sinal de entrada usando um dicionário de padrões que é atualizado de forma adaptativa usando concatenações de segmentos já codificados. O MMP é um método de compressão com perdas. Ele possui um dicionário com vetores de comprimentos diferentes que são usados para codificar segmentos de um sinal de entrada. [26]

Ele foi originalmente desenvolvido com foco na compressão de imagens. Mas diversas pesquisas visam expandir o campo de aplicação do algoritmo para diversos tipos de sinais, como por exemplo sinais de voz, eletrocardiogramas e radares meteorológicos. O MMP é considerado um método bastante eficiente que supera a qualidade de padrões tradicionais como JPEG e H.264/AVC. [27]

O MMP possui a característica de ser universal, pois não requer nenhum conhecimento prévio do sinal. O processo de codificação do MMP divide o sinal de entrada em segmentos que são aproximados a partir dos elementos do dicionário e através de uma estrutura recursiva que analisa se é interessante codificar um bloco inteiro ou particioná-lo e codificar suas partes. Como

o sinal é processado em blocos, o atraso mínimo previsto no processo entrada-saída é proporcional ao tamanho do bloco. [28]

Como critério de desempenho, o MMP tem aplicado a taxa-distorção através do cálculo do custo lagrangeano, pois esse método tem produzido os melhores resultados para diferentes fontes de sinal. O MMP é controlado por um único parâmetro que permite definir o grau de distorção da operação, é o fator lagrangeano exposto na equação 2.2.9. [29]

3.4 ALGORITMO VSTC

O algoritmo *Variable Size Transform Coder*, VSTC, foi desenvolvido por uma parceria entre pesquisadores do Instituto Militar de Engenharia, do Instituto de Telecomunicações (Portugal), do Instituto Politécnico de Leiria (Portugal) e da Universidade Federal do Rio de Janeiro, e foi apresentado em agosto de 2014 no ITS –*International Telecommunications Symposium*, em São Paulo. [1]

Em suma, o algoritmo VSTC se baseia na combinação da utilização da estrutura de otimização recursiva do algoritmo MMP com a aplicação de transformadas espaciais de segmentação flexível, ou seja, que incluem blocos quadrados e retangulares. Assim, a correspondência de padrões adotada pelo MMP é substituída pela aplicação de transformadas 2D.

Inicialmente, o algoritmo foi desenvolvido utilizando blocos de tamanho 16x16 e utilizando o framework de intra-predição definido pelo padrão H.264/AVC e ilustrado na figura.

O procedimento de otimização da taxa-distorção do VSTC é baseado no MMP e consiste basicamente de duas funções recursivas, uma que otimiza o modo de predição e outra que otimiza o resíduo, sendo o bloco de resíduo obtido a partir do modo de predição determinado. A figura ilustra o esquema multiescalar de blocos adotado pelo algoritmo:

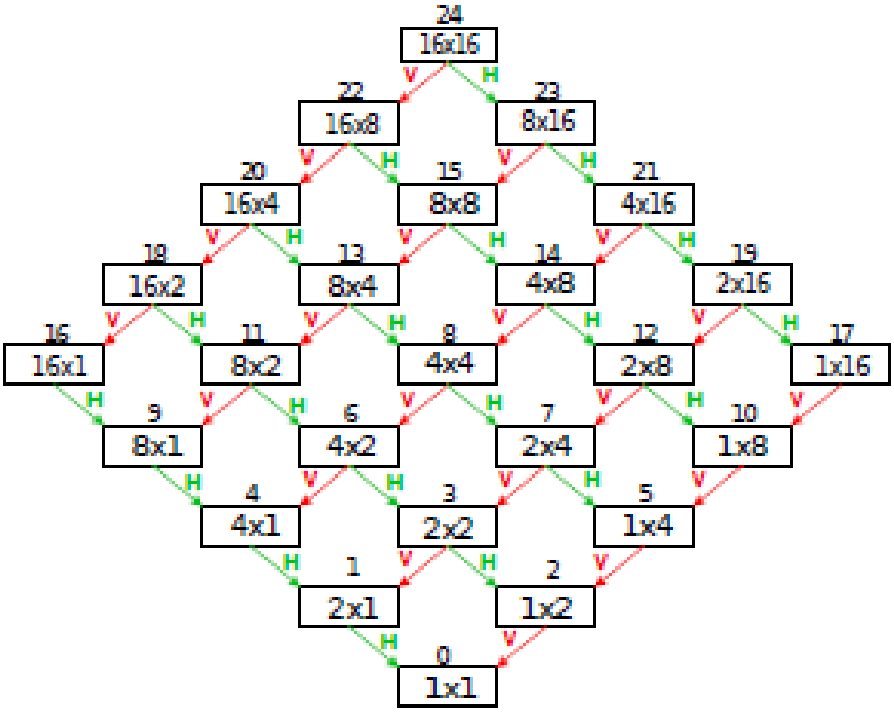


Figura 3.3 - Sequência de multiescalar de partições do bloco [1]

Cada possível tamanho da partição do bloco possui um identificador, chamado *level*, que corresponde ao número em cima de cada bloco na figura. A recursão da otimização que determina o modo de intra-predição é realizada do *level* 24 (16x16) até o *level* 8 (4x4). Já a recursão de otimização do resíduo é realizada até o *level* 0 (1x1).

O algoritmo MMP utiliza um único parâmetro de entrada, o fator Lagrangeano, λ, que controla o trade-off entre taxa e distorção, conforme descrito na equação 2.2.9. Já o algoritmo VSTC, além desse parâmetro, necessita também do parâmetro de quantização QP, que é utilizado no processo de quantização linear. Através de diversos testes combinando esses dois fatores, determinou-se que a melhor relação entre os dois fatores para otimizar o desempenho do algoritmo poderia ser obtida através da equação 3.5.1 [1]:

$$\lambda = a * 2^{\frac{QP-b}{c}} \tag{3.5.1}$$

Onde a, b e c são constantes, com a = 0.92, b=13.74 e c = 3.428. Dessa forma, o algoritmo VSTC utiliza apenas um único parâmetro de entrada, o parâmetro de quantização, QP, e o fator Lagrangeano λ é obtido através da equação 3.5.1.

Para o processo de varredura de um bloco, como o algoritmo VSTC codifica blocos de diversos tamanhos, ele utiliza três métodos distintos: a varredura linear, varredura em forma de z e varredura em em zig-zag, conforme a figura 3.4.

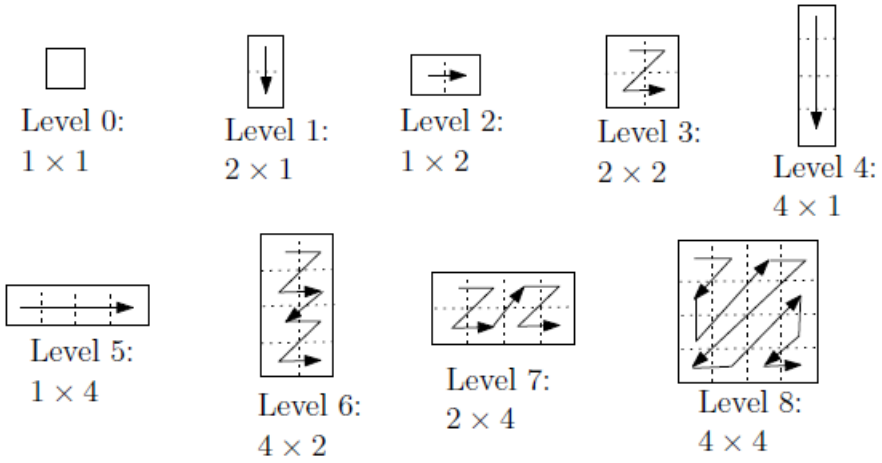


Figura 3.4 - Sequência de varredura de diferentes blocos [1]

Em geral, a varredura linear é aplicada nos blocos linha ou coluna, a varredura em forma de z é aplicada quando uma das dimensões do bloco possui dimensão dois e a varredura em zig-zag é aplicada nos demais casos.

Para codificação de entropia, o VSTC, assim como o H.264/AVC, utiliza o CABAC, que se mostrou mais eficiente para codificar blocos de resíduo, explorando algumas propriedades dos coeficientes das transformadas.

O procedimento de otimização da taxa-distorção do VSTC é baseado no MMP e consiste basicamente de duas funções recursivas, uma que otimiza o modo de predição e outra que otimiza o resíduo, sendo o bloco de resíduo obtido a partir do modo de predição determinado.

Os resultados experimentais mostraram que o algoritmo VSTC apresentou desempenho de compressão melhor que o algoritmo MMP e que o padrão H.264/AVC na compressão de imagens. Além disso, em relação ao MMP ocorreram ganhos significativos nos tempos de compressão, e, principalmente de descompressão.

Atualmente, estão sendo desenvolvidas melhorias no algoritmo que incluem a utilização de blocos iniciais de tamanho 64x64 ou 32x32 (definido como entrada do algoritmo), a utilização do esquema de intra-predição do H.265, que utiliza 35 modos de predição e a aplicação de novos esquemas de transformadas para que o VSTC possa competir com o padrão H.265 (HEVC). Essas

novas características alteram a árvore de partições do bloco da figura 3.3, adicionando novos elementos, conforme ilustrado na figura 3.5.

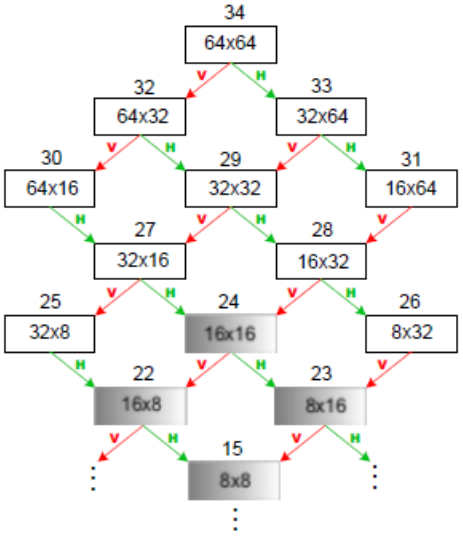


Figura 3.5– Novas partições do bloco

A introdução das novas escalas resultou numa melhora de desempenho quando comparada a versão original que utiliza blocos de no máximo 16 x 16 *pixels*. No entanto, essa introdução combinada com a utilização de um esquema de intra-predição de 35 modos também elevou bastante a complexidade computacional do método, pois como o algoritmo utiliza funções de otimização recursivas, esse aumento eleva o número de iterações de forma exponencial.

Dessa forma, a necessidade de otimizar o tempo de execução do algoritmo instigou a possibilidade de aplicar processamento paralelo na estrutura do algoritmo. Portanto, no próximo capítulo, serão introduzidos conceitos de processamento paralelo e padrões e ferramentas que podem ser utilizados no desenvolvimento de programas paralelos.

4 PROCESSAMENTO PARALELO

4.1 INTRODUÇÃO AOS PROCESSADORES PARALELOS

Microprocessadores baseados em uma única unidade central de processamento (CPU) impulsionaram aumentos de desempenho e reduções de custo nas aplicações de computador por mais de duas décadas. Esses dispositivos trouxeram bilhões de operações de ponto flutuante por segundo (GFLOPS) para o *desktop* e centenas de GFLOPS para os servidores em *cluster*. E o contínuo impulso de melhoria de desempenho tem permitido que o *software* de aplicação ofereça mais funcionalidade, tenha melhores interfaces com o usuário e gere melhores resultados. Os usuários, por sua vez, pedem por ainda mais melhorias, uma vez que, acostumados com essas melhorias, demandem mais capacidades criando um ciclo positivo para a indústria de computadores [30].

Pode-se notar que o aumento da velocidade de processamento dos dispositivos foi durante muito tempo alcançada a partir do aumento da frequência do *clock* da CPU [10]. Entretanto, os fabricantes se depararam com restrições de calor e limites físicos para que continuassem avançando nessa linha de evolução do *hardware* [30]. 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 [30]. A partir de 2005, os principais fabricantes de microprocessadores começaram a oferecer processadores com dois núcleos e na sequência três, quatro, seis e oito [10].

Juntamente com a evolução da CPU, observou-se também que as aplicações começaram a demandar cada vez mais dos gráficos 3D, o que impulsionou uma melhoria nos processadores gráficos em geral, devido ao processamento paralelo em muitos núcleos (*many-core*). A GPU (*Graphics Processing Unit*), que é unidade de processamento presente nas placas de vídeo, possui na ordem de centenas 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 [30].

Agora que todos os novos microprocessadores são computadores paralelos, o número de aplicações que precisam ser desenvolvidas como programas paralelos aumentou drasticamente.

Entretanto, a comunidade de computação de alto desempenho tem desenvolvido programas paralelos há décadas, porém, estas só funcionavam em computadores de grande escala, muito caros, nos quais somente algumas aplicações de elite podiam justificar seu uso, limitando, deste modo, a prática da programação paralela a um pequeno número de desenvolvedores de aplicação [30].

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.

Segundo Flynn, todo processo pode ser interpretado como uma relação entre fluxo de instruções, sequência de instruções executadas em um processador, e fluxo de dados. Assim é possível dividir as arquiteturas de computadores em quatro classes:

- SISD – *Single Instruction Stream/Single Data Stream*

Corresponde ao tradicional modelo de von Neumann, onde um processador executa em sequência um conjunto de instruções sobre um conjunto de dados, conforme a figura 4.1:

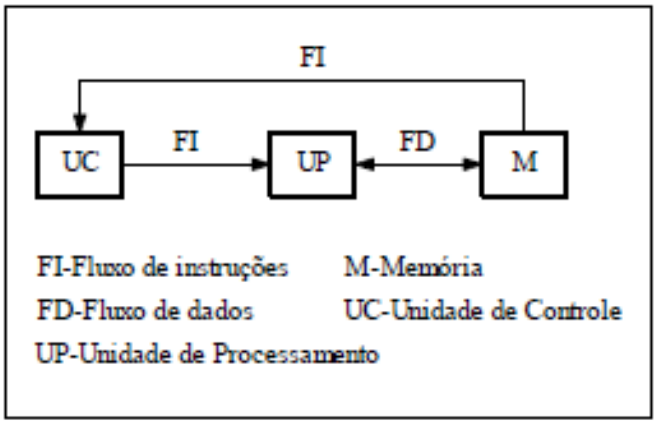


Figura 4.1 - Modelo computacional SSID

- SIMD – *Single Instruction Stream/Multiple Data Stream*

Envolve vários processadores (escravos) atuando sob o comando de uma unidade de controle (mestre) e executando simultaneamente a mesma instrução em diversos conjuntos de dados, conforme ilustrado na figura 4.2. São utilizadas por exemplo na multiplicação de matrizes e no processamento de imagens.

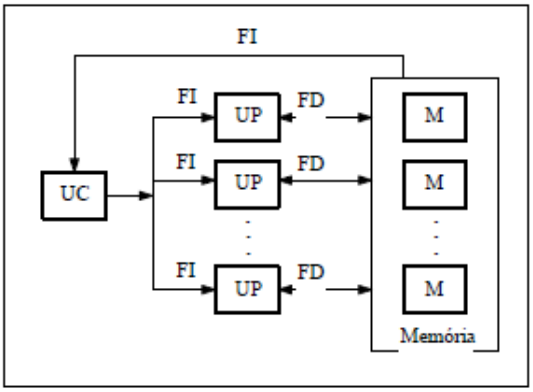


Figura 4.2 - Modelo computacional SIMD

- MISD – *Multiple Instruction Stream/Single Data Stream*

Como ilustrado na figura 4.3, esse modelo envolve diversos processadores executando diferentes instruções em um único conjunto de dados. Em geral, não existem representantes desta categoria, mas alguns autores classificam arquiteturas pipeline como deste modelo.

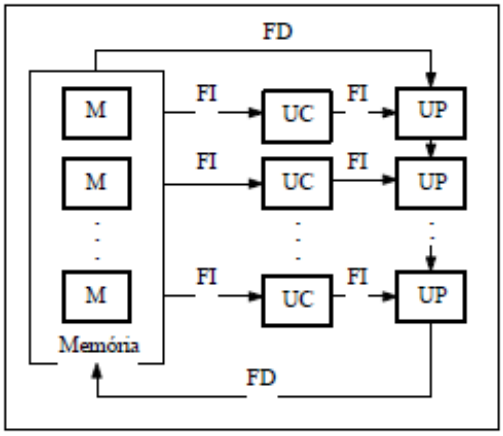


Figura 4.3– Modelo computacional MISD

- MIMD – *Multiple Instruction Stream/Multiple Data Stream*

Envolve múltiplos processadores executando, de forma independente, diversas instruções em vários conjuntos de dados. Esse modelo, que se encontra ilustrado na figura 4.4, engloba a maioria dos computadores paralelos.

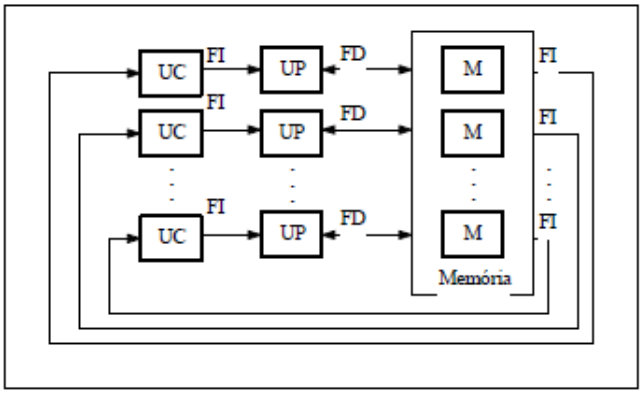


Figura 4.4– Modelo computacional MIMD

As arquiteturas SIMD oferecem facilidades para a programação e depuração de programas paralelos, pois contém um único fluxo de instruções e em geral seus elementos de processamento são simples. Já as arquiteturas MIMD possuem grande flexibilidade para execução de algoritmos paralelos e apresentam bom desempenho, pois, em geral, seus elementos de processamento são assíncronos. Apesar de independentes, em geral, os processos devem cooperar entre si, assim é necessário planejar a comunicação e o sincronismo entre os processos, esses vão depender da organização de memória no sistema. Há dois modelos de organização de memória:

- Memória compartilhada: há uma única memória global, que é acessada pelos diversos processadores. A comunicação entre os processos ocorre através do compartilhamento de posições nesta memória.
- Memória Distribuída: cada processador tem sua própria memória local e, dessa forma, é preciso haver alguma comunicação entre os processos.

As arquiteturas de memória distribuída estão sendo amplamente usadas nos dias atuais. Em geral, nessas arquiteturas, o custo da comunicação é de uma ordem de magnitude maior que o custo da computação. Assim, essas novas arquiteturas demandam novas abordagens de resolução dos problemas para obter boas performances. [31]

Nesse modelo de memória distribuída, a comunicação normalmente ocorre através da troca de mensagens, que consiste basicamente em processos sequencias trocando informações. Para realizar a comunicação entre os processos, existem diversas ferramentas, dentre as quais podemos destacar o MPI (*Message Passing Interface*). [32]

Para aferir o desempenho de programas paralelos, as medições mais aceitas são a eficiência e o *speed-up* [32]. A eficiência avalia como o algoritmo paralelo usa o tempo nos vários processadores e é calculada de acordo com a equação 4.1.1:

$$E(p) = \frac{Ts}{p*Tp} \qquad \text{Equação 4.1.1}$$

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 *speedup* avalia o ganho de tempo que o processamento paralelo apresenta sobre o sequencial e pode ser aferido através da equação 4.1.2:

$$Speedup = \frac{Ts}{Tp} \qquad (4.1.2)$$

O ganho de *speedup* deveria tender a p, que seria o valor ideal. No entanto diversos fatores, impendem que esse ideal seja alcançado, tais como: a sobrecarga de comunicação entre os processadores, há existência de partes do código estritamente sequenciais e o nível de paralelismo utilizado.

Por causa disso, a Lei de Amhdal é utilizada para encontrar o máximo *speedup* esperado para um sistema. Essa lei considera que o *speedup* é limitado pela fração sequencial do programa. A lei de Amhdal é descrita pela equação 4.1.3:

$$Speedup\ teórico = \frac{1}{(1-f)+\frac{f}{S}} \qquad (4.1.3)$$

Onde, *f* é a fração do código do programa que, de fato, foi paralelizada e *S* é o *speedup* dessa seção. Dessa forma, é evidente que a Lei de Amhdal considera que a parte serial de execução do programa vai diminuir o *speedup* esperado. Dessa forma, a Lei de Amhdal limita o o speedup, independente de quantos processadores são utilizados na execução do programa. O gráfico da figura 4.5 ilustra como o *speedup* varia com o número de processadores, para diversos valores fixos de *f*.

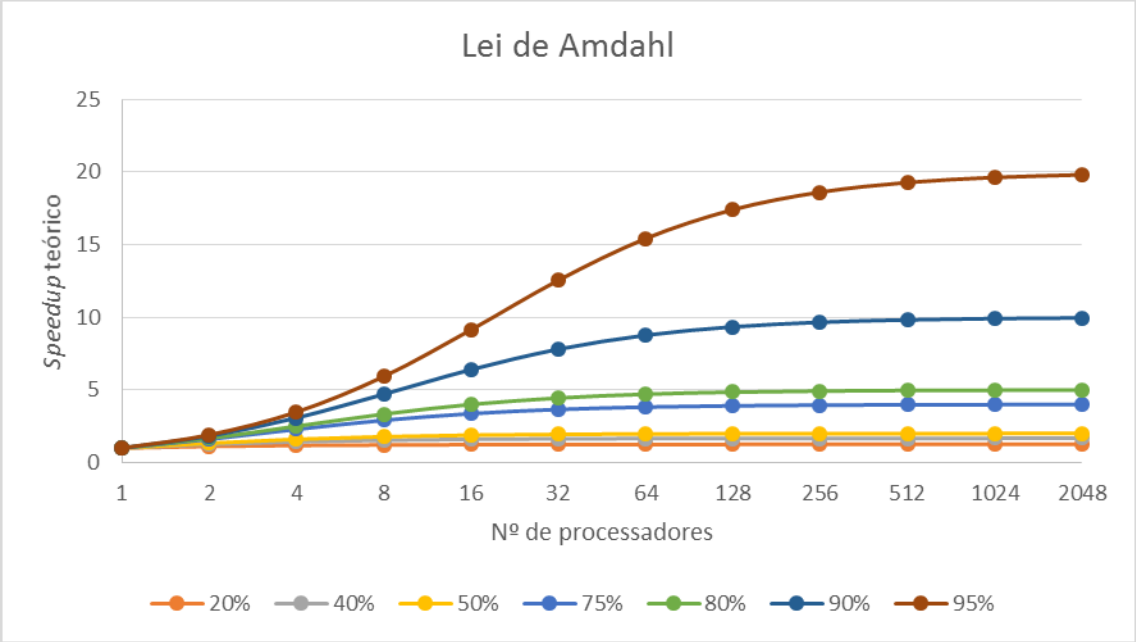


Figura 4.5 – Gráfico da Lei de Amhdal

De acordo com a figura 4.5, temos que, por exemplo, se *f*=95%, ou seja, 95% do código pode ser paralelizado, o máximo *speedup* passível de se obter será de 20 vezes, independentemente da quantidade de processadores utilizados.

Após essa introdução sobre a programação paralela em geral, nas seções seguintes prossegue-se discutindo as duas ferramentas selecionadas para a realização deste trabalho, o paradigma MPI e a programação em GPU.

4.2 PARADIGMA DE 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 [33].

Esse padrão surgiu como resultado do trabalho de aproximadamente 60 pessoas de 40 instituições distintas, incluindo fabricantes de computadores paralelos, laboratórios e autoridades governamentais. O início do MPI se deu em um seminário sobre padronização para troca de mensagens, organizado pelo *Center for Research on Parallel Computing* em abril de 1992. [34]

O projeto do MPI padrão foi apresentado na conferência *Supercomputing* 93, realizada em novembro de 1993 e desse projeto se originou a versão oficial do MPI-1 lançada em maio de 1994. [34]

Foram realizadas mais experiências práticas com o MPI, de forma que após diversas alterações no projeto, foi apresentada, no *Supercomputing* 96, a versão preliminar do MPI-2, que foi unanimemente votado e aceito como padrão em abril de 1997 [34]. Atualmente, o projeto se encontra na versão MPI-3 que foi lançada em 2012.

Atualmente, o padrão MPI é o método de comunicação em computação paralela mais portátil e mais amplamente utilizado. Oferecendo robustez, escalabilidade e portabilidade sem comprometer a performance da aplicação. [31]

O paradigma de programação MPI assume que as estruturas de dados não são compartilhadas, mas divididas entre os múltiplos processadores. Essa ideia de divisão de armazenamento requer uma técnica de decomposição do domínio, que vai gerar um custo de overhead. [35]

No ambiente MPI, os programas paralelos devem ser desenvolvidos de acordo com o modelo SIMD. 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 a classificação MIMD [34]. A figura 4.6 ilustra um esquema do modelo MPI:

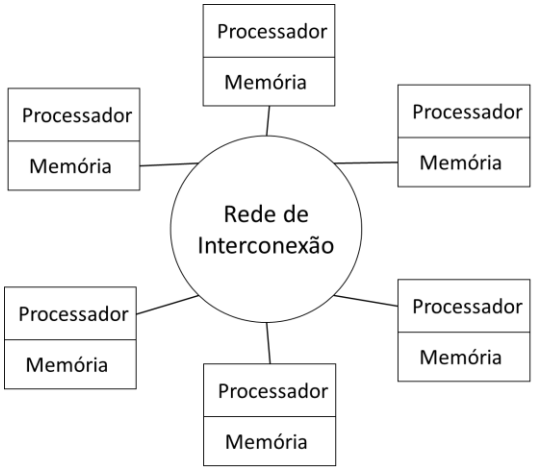


Figura 4.6– Modelo de processos no MPI [36]

Conforme a figura 4.6, todos os processos, que são executados em processadores diferentes tem 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. [36]

Existem diversas versões de bibliotecas MPI disponíveis no mercado, como MPICH2, LAM/MPI e Open MPI. [37]

A paralelização MPI é vantajosa pois permite a execução em máquinas paralelas, reduzindo o tempo de execução, mas também oferece alocação e armazenamento distribuídos. [35]

Uma desvantagem do MPI é que o programa completo deve executar de forma paralela e não somente as partes que consomem tempo de processamento. Na prática, isso acaba sendo ignorado e várias partes do programa são executadas em todos os nós de forma repetitiva. Isso pode, por exemplo, resultar em conflitos no acesso a arquivos. [37]

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 coletivas de comunicação, usadas para executar operações globais. [34]

Apesar do padrão MPI ser bastante complexo, um conjunto de seis primitivas básicas é capaz de solucionar uma grande quantidade de problemas [31]. 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, eles 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. [34]

Outra característica importante do MPI é que a troca de mensagens não é determinística, assim não é garantido que os processos cheguem na ordem em que foram enviados. A responsabilidade de assegurar a execução determinística é do programador, que pode distinguir mensagens através de *tags* ou especificar o processo que deve ser a origem da mensagem na operação de recepção. [38]

4.3 COMPUTAÇÃO EM GPU

A NVIDIA, pensando em extrair todo o potencial computacional da GPU para outras aplicações, desenvolveu uma técnica de programação em que os programadores declarariam explicitamente os aspectos paralelos dos dados de sua carga de trabalho. Assim, surgiu o compilador CUDA *C/C++*, bibliotecas e software de *runtime* para permitir que os programadores rapidamente acessassem o novo modelo de computação paralela de dados e desenvolvessem suas aplicações. Os programadores não precisariam mais usar a API gráfica para acessar as capacidades de computação paralela da GPU [10].

A plataforma foi introduzida formalmente em fevereiro de 2007 [39]; e, atualmente, encontra-se em sua versão 4.1 [40]. A figura 4.7 ilustra a diferença das filosofias de projeto entre a CPU e a GPU:

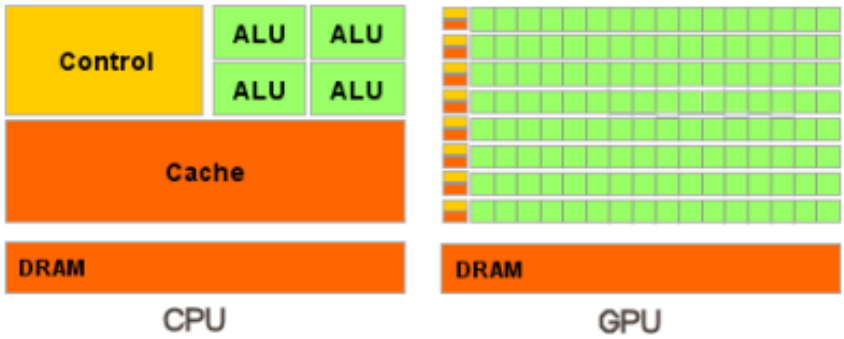


Figura 4.7 - Diferença das filosofias de projeto entre a CPU e a GPU [39].

A arquitetura das GPUs é projetada de tal forma que a maior parte dos transistores são dedicados ao processamento de dados, em vez de cache de dados e controle de fluxo como ocorre na CPU [39]. Essa dedicação da GPU se deve à característica paralela das aplicações gráficas que realizam a mesma operação em um grande volume de dados [40]. Pode-se ver no esquema de utilização de transistores disposto na figura 4.7 que 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.

A figura 4.8 ilustra o crescimento diferencial de desempenho entre CPU e GPU ao longo dos anos.

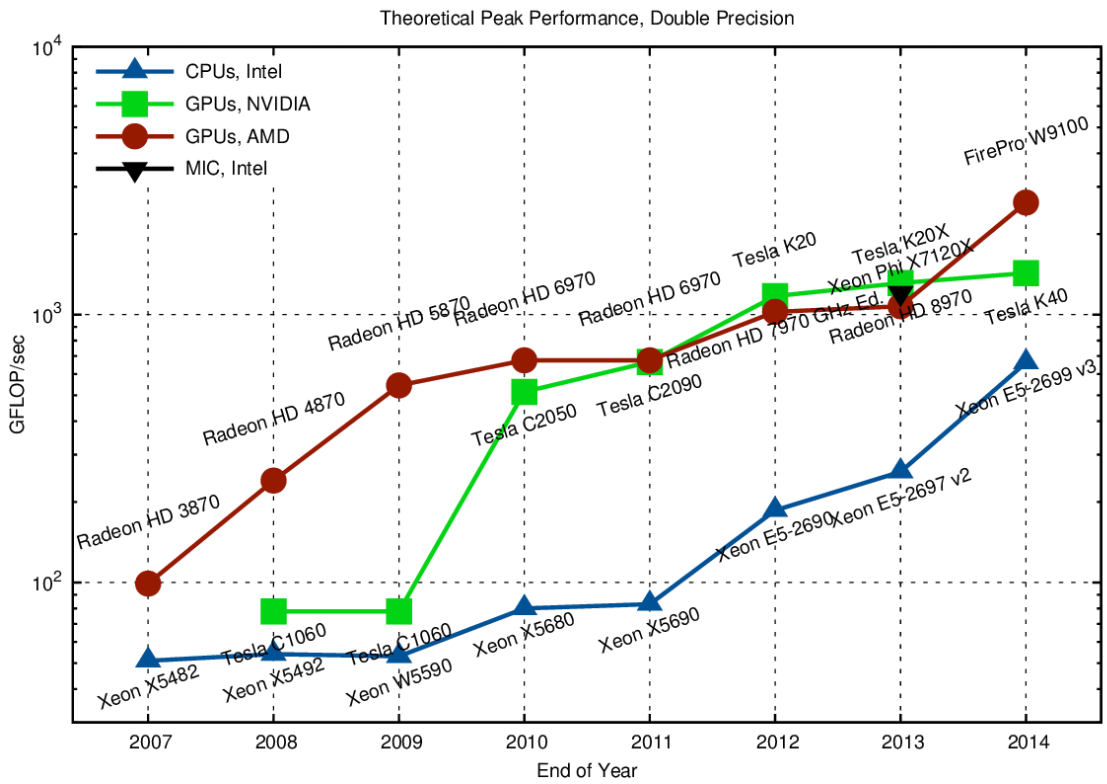


Figura 4.8 - Crescimento diferencial de performance entre CPU e GPU [41]

O rápido aumento no desempenho do *hardware* gráfico, juntamente com as recentes melhorias na sua programação, fizeram a GPU uma plataforma atraente para tarefas computacionalmente exigente, em uma grande variedade de domínios de aplicação.

4.3.1 INTRODUÇÃO CUDA

O CUDA é uma arquitetura desenvolvida para as placas de vídeo da NVIDIA que permite que a GPU possa ser usada tanto para computação em aplicações de uso geral quanto para as necessidades gráficas tradicionais. Quando escrevemos um código para que esse seja executado na placa de vídeo, a esse trecho chamamos de *kernel* [10].

Um programa CUDA consiste em uma ou mais fases que são executadas ou no *host* (CPU) ou em um *device* como uma GPU. As fases que exibem pouco ou nenhum paralelismo são executadas no *host*. Enquanto as que exibem uma quantidade de paralelismo de dados são implementadas no código do *device*. O código do *device* normalmente é escrito em C usando ANSI

C estendido com palavras chaves para rotular as funções com dados paralelos, chamadas *kernels*, e suas estrutura de dados associadas [42].

As funções *kernel* normalmente geram um grande número de *threads* para explorar o paralelismo de dados [42].

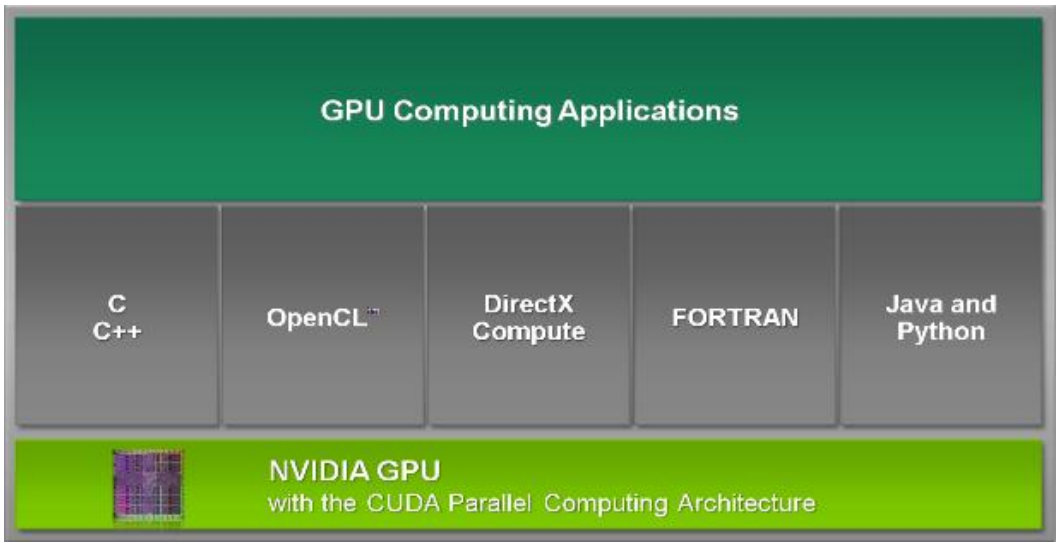


Figura 4.9 - Linguagens e APIs suportadas pelo CUDA [39].

Como mostra a figura 4.9, a arquitetura fornece suporte a várias linguagens e APIs (*Application Programming Interfaces*) para o desenvolvimento de aplicações. Na sua versão 4.1 podem ser utilizadas linguagens de alto nível, como *C*, *C++*, *Fortran*, *Java* e *Python* ou APIs do *driver*, como *OpenCL* e *DirectX Compute*. Para que se possa utilizar a arquitetura CUDA, é necessário primeiramente satisfazer alguns requisitos de *hardware* e *software*. Para satisfazer o requisito de *hardware*, é necessário uma placa gráfica com suporte a arquitetura de *software* CUDA. Basicamente, a partir da série 8 da NVIDIA (NVIDIA *GeForce* 8***), as placas apresentam esse suporte [43]. Para satisfazer os requisitos de *software*, são necessários no mínimo 2 pacotes para instalação: o *driver* especifico para o modelo do hardware, e o CUDA *Toolkit* que contém o compilador juntamente com algumas ferramentas e bibliotecas. Adicionalmente, pode-se instalar um pacote com códigos de exemplo (*code samples*) [44].

O modelo de programação CUDA possui um conceito de execução ordenada e paralela de *threads*. Os *threads* são agrupados em blocos e os blocos agrupados em grades (*grids*). A configuração da quantidade de blocos na grade e a quantidade de *threads* no bloco são definidas antes da chamada do *kernel*. A grade é a estrutura básica para alocação de blocos na qual é

determinado o número máximo de blocos por grade. As grades podem ter uma ou duas dimensões. O bloco é a estrutura básica para alocação de *threads* no qual é determinado o número máximo de *threads* por bloco. Um bloco pode ter uma, duas ou até três dimensões [44].

Na arquitetura CUDA é feita uma cópia do *kernel* e dessa forma permite-se que as instruções associadas a cada bloco sejam executadas em paralelo. Se executarmos um *kernel* com apenas um bloco, significa que apenas uma instância será executada, porém se temos três blocos, podemos dizer que o mesmo código está sendo executado por três entidades diferentes e ao mesmo tempo [41]. Para se executar uma chamada de *kernel*, o mesmo precisa ser informado com quantos blocos e *threads* irá trabalhar. Deste modo antes de efetuar uma chamada de *kernel*, duas variáveis precisam ser configuradas. Uma para a quantidade de blocos e outra para a quantidade de *threads* [44]. A figura 4.10 ilustra a organização das grades (*grids*), blocos e *threads*.

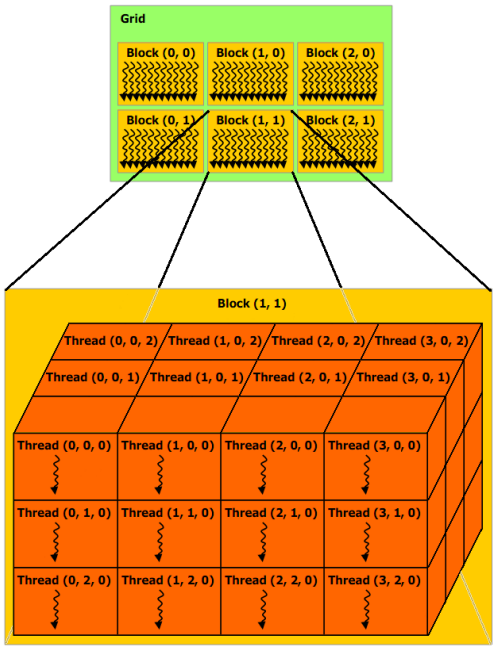


Figura 4.10 - Exemplo de um *grid* com seus blocos e *threads* [39].

Os blocos e *threads* são identificados por índices que permitem a sua distinção dentro do *kernel*, facilidade que auxilia no entendimento de que etapa do processo cada bloco e/ou *thread* estará executando. Essas variáveis que identificam os blocos e *threads* são chamadas, respectivamente, *blockIdx* e *threadIdx*. Para o caso bidimensional, usa-se como forma de identificação as variações de *x* e *y* para ambos os casos [10]. A variável *gridDim* armazena a

configuração da quantidade de blocos em cada dimensão da grade, e a variável *blockDim* armazena a configuração da quantidade de *threads* em cada dimensão dos blocos. Todas as variáveis *Built-in* são do tipo dim3 [44]. A tabela 4.1 organiza as variáveis globais disponíveis na arquitetura CUDA.

Tabela 4.1: Variáveis globais disponíveis na arquitetura CUDA

Variáveis Built-in	Significado	Dimensão	Acesso
threadIdx	Índice do thread no bloco	X	threadIdx.x
		Y	threadIdx.x
		Z	threadIdx.z
blockIdx	Índice do bloco na grade	X	blockIdx.x
		Y	blockIdx.x
		Z	blockIdx.z
blockDim	Dimensão do bloco	X	blockDim.x
		Y	blockDim.x
		Z	blockDim.z
gridDim	Dimensão da grade	X	gridDim.x
		Y	gridDim.y

Para exemplificar, a figura 4.11 ilustra como um *loop* de um código de soma de vetores poderia ser escrito.

```
1 ———————————— DE ————————————————————————
2 for (i = 0; i < 81; i ++{
3             v3[i] = v1[i] + v2[i];
4     }

1 ———————————— PARA ———————————————————————
2 idThread= threadIdx.x + threadIdx.y*blockDim.x;
3 v3[idThread] = v1[idThread] + v2[idThread];
```

Figura 4.11 - Soma de dois vetores utilizando a forma sequencial e o paralelismo na GPU [10].

4.3.2 HIERARQUIA DE MEMÓRIA

Nessa arquitetura representada na figura 4.12 tem-se representadas quatro tipos de memórias: a memória global, a memória compartilhada, a memória constante e a memória de textura. Cada bloco possui uma memória compartilhada e 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 pelos *threads* para armazenar suas próprias variáveis, chamadas de variáveis automáticas. Elas não são visíveis a outros *threads* e só existem enquanto o *thread* que a criou existir. A memória compartilhada tem a serventia de promover a cooperação entre os *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 todos os *threads* nele contido [44]. Tem-se ainda a memória de textura que proporciona uma otimização para os casos em que o problema possa ser tratado em duas ou três dimensões e que as informações de vizinhança possuam alguma dependência. Por exemplo, aplicação de um filtro de média em que o valor de um *pixel* depende do valor dos *pixels* ao seu redor [10].

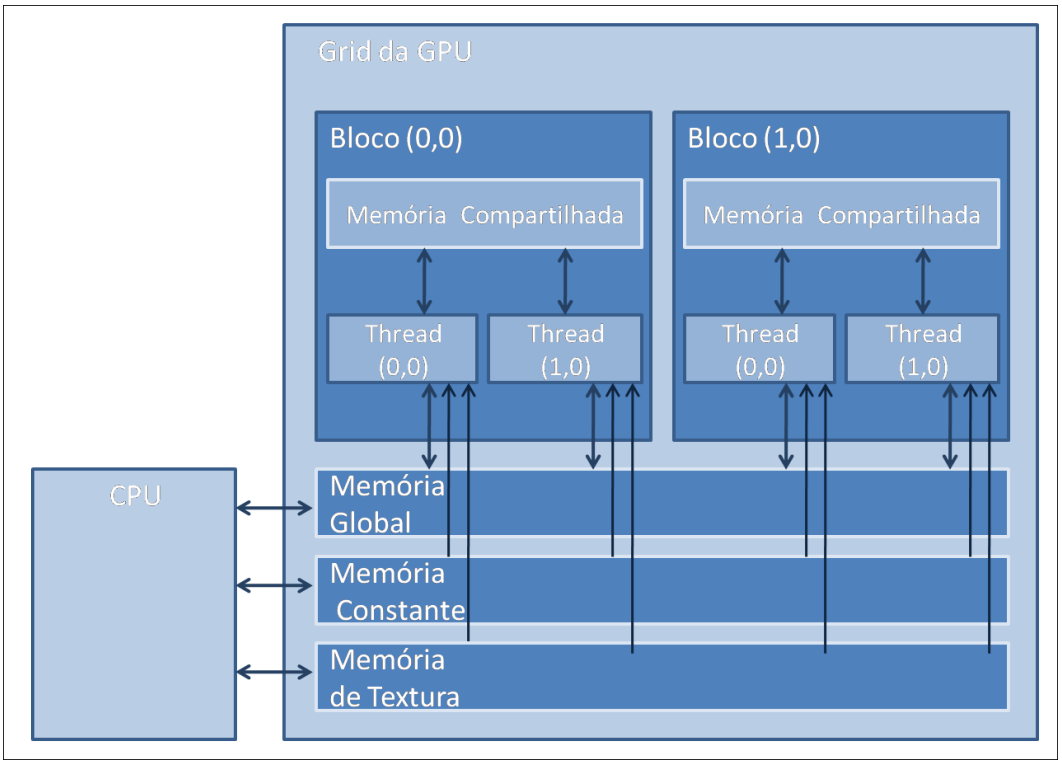


Figura 4.12 - Tipos de memórias na arquitetura CUDA [10].

A GPU possui uma hierarquia de memória que pode ser utilizada com o objetivo de melhorar o desempenho das aplicações. Os qualificadores de variáveis de acordo com os tipos de memória podem ser vistos na tabela 4.2:

Tabela. 4.2: Qualificadores de variáveis em CUDA [44].

Declaração da variável	Memória	Escopo	Tempo de vida
Variável automática	Registrador	*Thread*	*Kernel*
__device__ __shared__ int var	Compartilhada	Bloco	*Kernel*
__device__ int var	Global	Grade	Aplicação
__device__ __constant__ int var	Constante	Grade	Aplicação

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 um *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 ao *thread* [44].

4.3.3 INTRODUÇÃO AO OPENACC

Na tentativa de se aproveitar da GPU sua capacidade de processamento de cálculos matemáticos nas aplicações em sistemas de computação heterogêneos, implicou no surgimento de várias APIs de baixo nível para a programação orientada a esse processador gráfico.

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

O padrão *OpenCL* representa um esforço para criar uma interface de programação comum para dispositivos heterogêneos de grande adesão. No entanto, seu modelo de programação não é simples. As abordagens recentes empregam programação de alto nível baseada em diretiva para aceleradores. Nesse contexto surge a *OpenACC*, uma API que consiste em diretivas de compilador em regiões de código em *C, C ++* e *Fortran* suscetíveis para serem executadas na GPU.

A *OpenAcc* consiste em um padrão de programação de computação paralela 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.

O suporte de *OpenACC* está disponível em compiladores comerciais do PGI a (partir da versão 12.6) , *Cray* e CAPS. O compilador GNU GCC suporta *OpenACC* a partir da GCC versão 5.

A *OpenACC* define uma extensa lista de pragmas (diretivas). As duas diretivas em seguida são fundamentais para o início da execução paralela no dispositivo acelerador, para a linguagem de programação C.

#pragma acc kernels
#pragma acc parallel

A figura 4.13 abaixo ilustra a utilização do *OpenAcc* na execução paralela de um *loop*. Cada *loop* é compilado em um *kernel* GPU separado.

```
#pragma acc kernels
for(i=0;i<max;i++)
    {
        for(j=0;j<max;j++)
        {
            A[i][j]=i+j;
        }
    }
```

Figura 4.13 - Exemplo de utilização do *OpenAcc* na execução paralela de um *loop*.

Para o uso eficiente da GPU, transferências de dados entre o *host* e o acelerador devem ser mantido no mínimo, por isso a cópia de dados do *host* para o acelerador é feita na maioria dos casos. A diretiva (*#pragma acc data)* é utilizada para a cópia de memória para o acelerador, ilustrada na figura 4.14.

```
#pragma acc data copy(A)
#pragma acc kernels
for(i=0;i<max;i++)
{
    for(j=0;j<max;j++)
    {
        A[i][j]=i+j;
    }
}
```

Figura 4.14 - Exemplo de cópia de memória entre *host* e GPU.

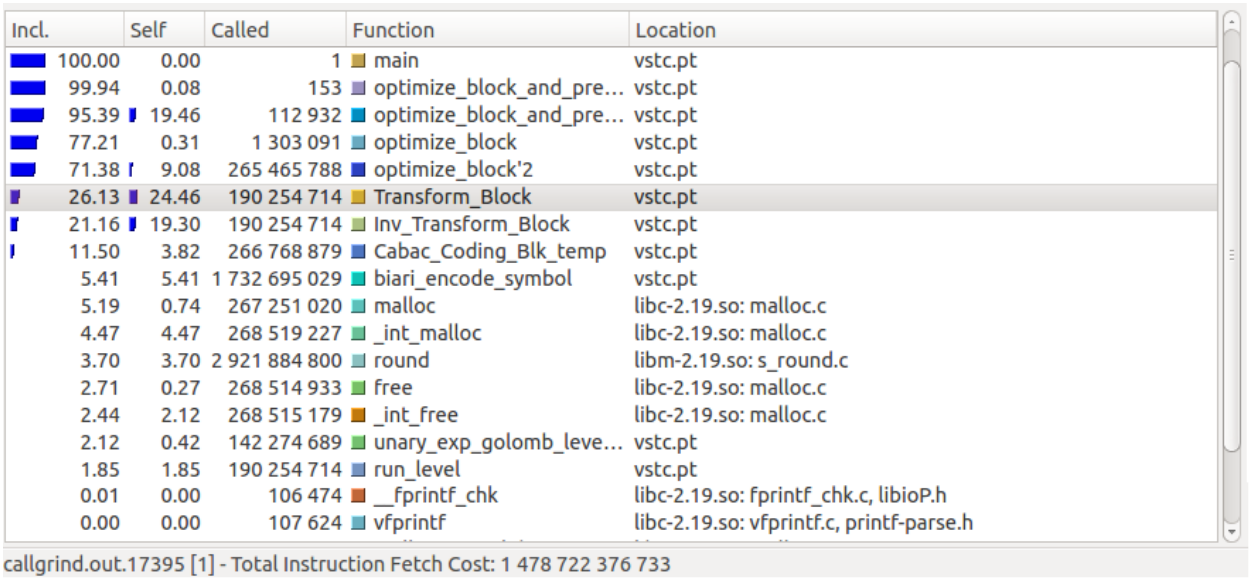
Neste capítulo foi possível compreender diversos conceitos e ferramentas relacionados à programação paralela, no capítulo seguinte, será dado prosseguimento ao trabalho analisando a execução do algoritmo VSTC e propondo implementações do algoritmo que utilizem os conceitos e ferramentas de programação paralela abordados nesse capítulo.

5 PROPOSTAS DE PARALELIZAÇÃO DO ALGORITMO VSTC

5.1 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 do *profiling*.

Foram executados três testes em modo *debug*, com parâmetro de entrada QP=28, utilizando essas ferramentas com as imagens (a) *Small*, (b) *RaceHorses* e (c) Lena, que se encontram no anexo A. As imagens 5.1, 5.2 e 5.3 contém parte dos resultados de cada execução.



Incl.	Self	Called	Function	Location
100.00	0.00	1	main	vstc.pt
99.94	0.08	153	optimize_block_and_pre...	vstc.pt
95.39	19.46	112 932	optimize_block_and_pre...	vstc.pt
77.21	0.31	1 303 091	optimize_block	vstc.pt
71.38	9.08	265 465 788	optimize_block'2	vstc.pt
26.13	24.46	190 254 714	Transform_Block	vstc.pt
21.16	19.30	190 254 714	Inv_Transform_Block	vstc.pt
11.50	3.82	266 768 879	Cabac_Coding_Blk_temp	vstc.pt
5.41	5.41	1 732 695 029	biari_encode_symbol	vstc.pt
5.19	0.74	267 251 020	malloc	libc-2.19.so: malloc.c
4.47	4.47	268 519 227	_int_malloc	libc-2.19.so: malloc.c
3.70	3.70	2 921 884 800	round	libm-2.19.so: s_round.c
2.71	0.27	268 514 933	free	libc-2.19.so: malloc.c
2.44	2.12	268 515 179	_int_free	libc-2.19.so: malloc.c
2.12	0.42	142 274 689	unary_exp_golomb_leve...	vstc.pt
1.85	1.85	190 254 714	run_level	vstc.pt
0.01	0.00	106 474	__fprintf_chk	libc-2.19.so: fprintf_chk.c, libioP.h
0.00	0.00	107 624	vfprintf	libc-2.19.so: vfprintf.c, printf-parse.h

callgrind.out.17395 [1] - Total Instruction Fetch Cost: 1 478 722 376 733

Figura 5.1– Resultado da execução no Valgrind do algoritmo para a imagem (a) *Small*

Incl.	Self	Called	Function	Location
100.00	0.00	1	main	vstc.pt
99.95	0.06	476	optimize_block_and_pre...	vstc.pt
95.99	16.92	351 344	optimize_block_and_pre...	vstc.pt
80.54	0.25	5 614 001	optimize_block	vstc.pt
75.28	7.36	1 065 614 900	optimize_block'2	vstc.pt
25.79	4.44	1 071 228 901	Cabac_Coding_Blk_temp	vstc.pt
20.66	19.28	762 556 092	Transform_Block	vstc.pt
19.59	18.01	762 556 092	Inv_Transform_Block	vstc.pt
13.54	13.54	1 074 787 793	__memcpy_sse2_unalig...	libc-2.19.so: memcpy-sse2-unaligned.S
6.77	6.77	10 839 351 387	biari_encode_symbol	vstc.pt
3.75	0.60	1 072 752 140	malloc	libc-2.19.so: malloc.c
3.17	3.17	1 077 339 703	_int_malloc	libc-2.19.so: malloc.c
3.12	3.12	11 540 842 720	round	libm-2.19.so: s_round.c
2.49	0.50	942 943 532	unary_exp_golomb_leve...	vstc.pt
2.04	0.22	1 077 335 409	free	libc-2.19.so: malloc.c
1.81	1.66	1 077 335 668	_int_free	libc-2.19.so: malloc.c
1.49	1.49	762 556 092	run_level	vstc.pt
0.00	0.00	3 775	arith_code_mode	vstc.pt

callgrind.out.17292 [1] - Total Instruction Fetch Cost: 7 289 256 949 724

Figura 5.2 - Resultado da execução no Valgrind do algoritmo para a imagem (b) RaceHorses

Incl.	Self	Called	Function	Location
100.00	0.00	1	main	vstc.pt.dbg: main_vstcenc.c
99.96	0.06	1 088	optimize_block_and_pre...	vstc.pt.dbg: coder.c
95.44	18.06	803 072	optimize_block_and_pre...	vstc.pt.dbg: coder.c
75.94	0.25	13 791 960	optimize_block	vstc.pt.dbg: coder.c
68.90	5.18	2 585 563 142	optimize_block'2	vstc.pt.dbg: coder.c
32.68	32.20	1 849 611 288	Transform_Block	vstc.pt.dbg: dct.c
20.45	19.89	1 849 611 288	Inv_Transform_Block	vstc.pt.dbg: dct.c
12.61	0.59	2 599 355 102	Cabac_Coding_Blk_temp	vstc.pt.dbg: cabac_encode.c
5.64	5.64	5 198 710 204	__memcpy_sse2_unalig...	libc-2.19.so: memcpy-sse2-unaligned.S
5.49	4.74	194 043 346 396	free_residue_tree	vstc.pt.dbg: coder.c
3.75	3.40	18 383 179 145	biari_encode_symbol	vstc.pt.dbg: cabac_bit_input.c
3.17	1.01	1 550 334 413	write_significant_coeffi...	vstc.pt.dbg: cabac_encode.c
2.42	0.86	1 211 326 167	write_significance_map...	vstc.pt.dbg: cabac_encode.c
1.46	0.22	2 602 831 389	malloc	libc-2.19.so: malloc.c
1.25	1.25	2 612 323 346	_int_malloc	libc-2.19.so: malloc.c
0.04	0.04	28 091 991	getClosestPixels	vstc.pt.dbg: lsp_pred.c
0.04	0.01	804 160	Enc_ReconstructOrigina...	vstc.pt.dbg: coder.c

callgrind.out.9786 [1] - Total Instruction Fetch Cost: 49 464 237 276 817

Figura 5.3 - Resultado da execução no *Valgrind* do algoritmo para a imagem (b) Lena

Os resultados expostos nas figuras 5.1, 5.2 e 5.3 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 em que a função estava na pilha de execução do programa na pilha *Incl.*, e também a localização da função dentro do projeto na coluna *Location*.

A análise das figuras indica que um conjunto de quatro funções é responsável pela maior parte do tempo de execução. São elas: *optimize_block_and_pred_mode*, *optimize_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.

Além disso, o Kcachegring também indica o fluxo de sequência de chamadas para uma determinada função. A figura 5.4 ilustra o resultado desse fluxo para a função *optimize_block* na execução da imagem (a) Small.

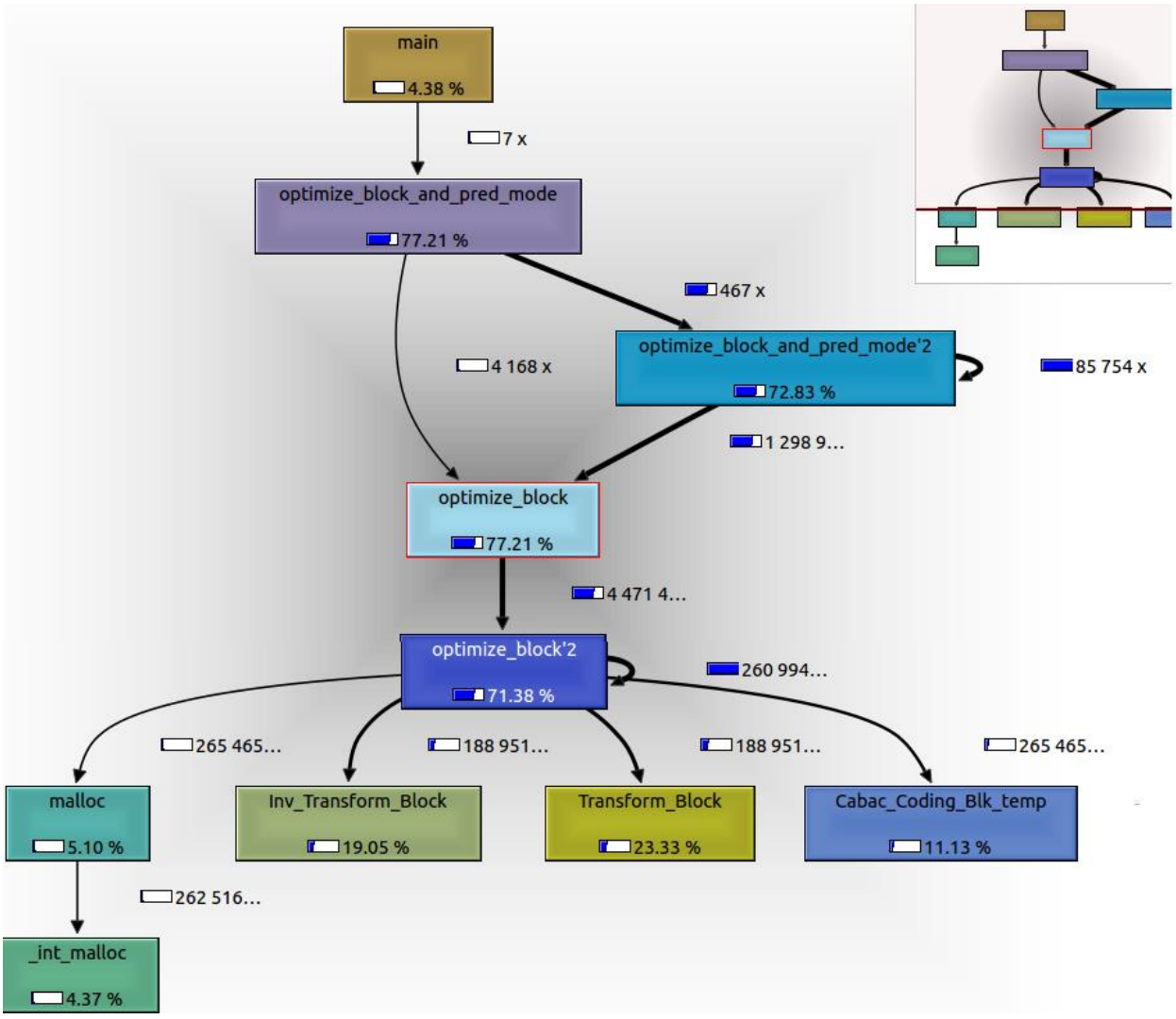


Figura 5.4 – Sequência de chamadas de funções para teste utilizando a imagem (a) *Small*

Pode-se perceber que 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 parcelas, na primeira, ela testa todos os possíveis modos de intra-predição e para cada um faz uma chamada a função que otimiza o resíduo. Na segunda, ela subdivide o bloco em quatro divisões, quando possível, duas horizontais e duas verticais, e executa o mesmo procedimento de forma recursiva, em seguida determina se particionar o bloco resulta em obter uma compressão mais eficiente.

A função atua sequencialmente sobre cada bloco 64 x 64 da imagem de entrada de forma sequencial, mas não é possível paralelizar por blocos pois o resultado de um bloco depende do resultado dos blocos anteriores, pois utiliza-se o esquema de predição. Também é complicado paralelizar a segunda parte da função, pois como nessa parte a função faz chamadas recursivas seria necessário alterar esse funcionamento para limitar o número de chamadas ao número de *threads* que estariam executando em paralelo.

Portanto, a primeira parte da função *optimize_block_and_pred_mode* é a que apresenta melhor potencial para ser paralelizada. O funcionamento dessa primeira parte da função encontra-se ilustrado na figura 5.5 abaixo:

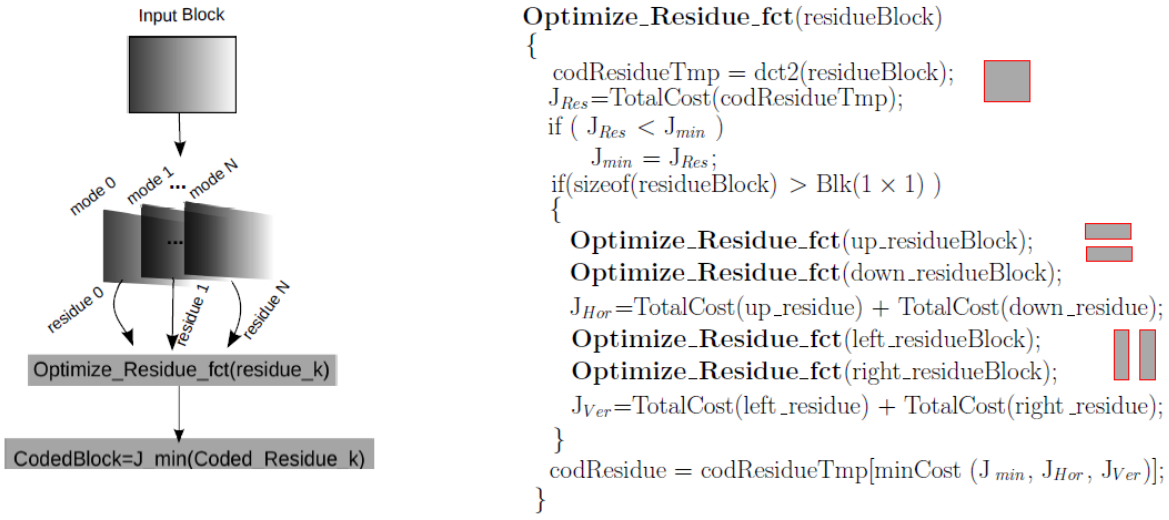


Figura 5.5– Ilustração da primeira parte da função *optimize_block_and_pred_mode* [1]

No lado esquerdo da figura 5.5 está ilustrado o funcionamento da função *optimize_block_and_pred_mode* e no lado direito um pseudocódigo que explicita a execução da função *optimize_block.* Para cada modo de predição, é realizada a otimização do resíduo e determinado o custo do resíduo de forma recursiva, para em seguida se determinar qual o modo de predição que apresenta o menor custo. Serão discutidas possibilidades de paralelizar essa determinação do bloco de menor custo.

5.2 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, discutiu-se a possibilidade de implementar um paralelismo na mesma. A figura 5.6 ilustra o algoritmo de determinação do modo de predição que resulta no menor custo do bloco, com sua execução de forma sequencial.

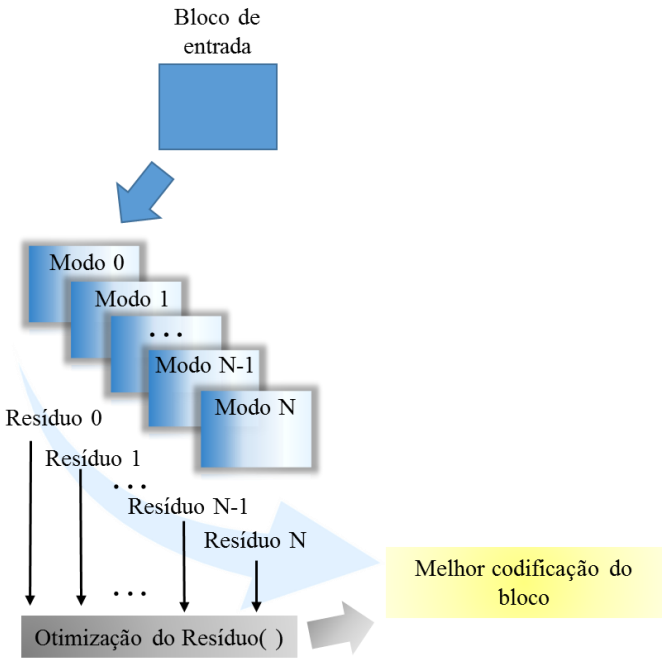


Figura 5.6– Algoritmo sequencial da determinação do melhor modo de intra-predição

Destaca-se que cada modo de predição é calculado de forma independente dos demais, 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 5.7 ilustra uma proposta de execução paralela desse algoritmo.

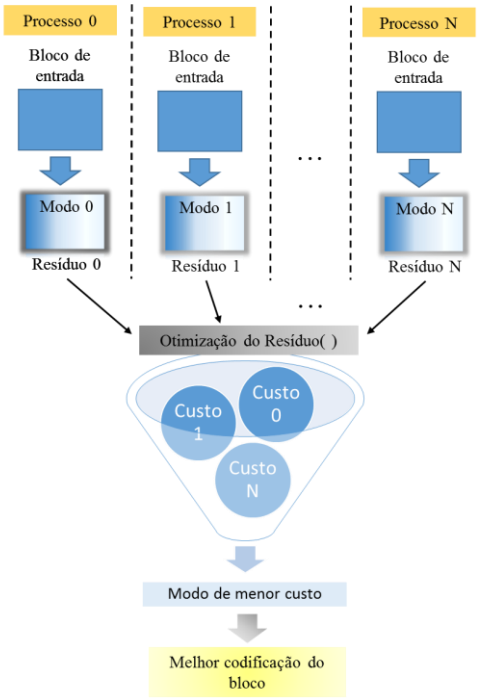


Figura 5.7 – Algoritmo paralelo de determinação do melhor modo de intra-predição

Na figura 5.7, é exposto que cada processo vai calcular para um modo de predição específico um resíduo 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 ranque do processo, do bloco e obtido um custo correspondente a 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 5.8:

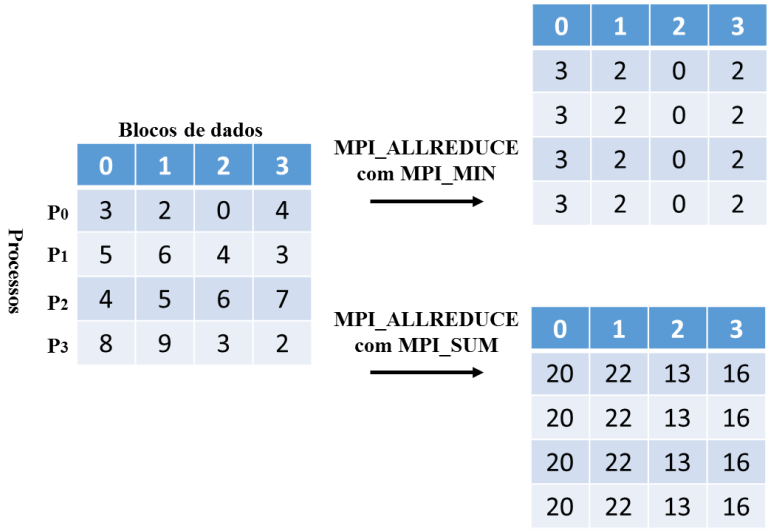


Figura 5.8 – Exemplo do MPI_AllReduce – Adaptado de [34]

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 [34]. A figura 5.8 ilustra o funcionamento da *MPI_AllReduce* combinada com duas funções distintas, cada processo inseriu no buffer de entrada um vetor de 4 inteiros. Na parte superior, ela executou 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 executou com a operação *MPI_SUM* e retornou no buffer de saída de cada processo a soma dos valores.

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. No caso, a posição indica qual o modo de predição que foi utilizado e, a partir desse modo, cada processo pode determinar a melhor codificação do bloco.

Porém, cada processo precisa recalcular o processo de otimização do bloco de resíduo para o modo de predição escolhido por todos os processos, apesar de um dos processos já ter calculado essa informação. Não é possível que esse processo transmita essa informação aos demais pois ela é representada na forma de ponteiros que serão utilizados para particionar o bloco verticalmente e horizontalmente. Caso o processo que detenha essa informação a transmita, ela perdera seu significado, pois, como os processos possuem memória independente e o ponteiro registra um endereço de memória, o valor de um ponteiro em um processo não tem significado para os outros processos.

Assim, são utilizados 36 modos de predição, mas nem todos os blocos possuem todos os modos de predição disponíveis. No entanto, podemos estimar os ganhos com essa implementação: em geral seriam executados 36 modos de predição, mas de forma paralela, cada processo executa no máximo dois modos de predição, o correspondente ao seu modo de predição e o correspondente ao melhor modo para esse bloco, pois, como foi exposto, é preciso determinar os ponteiros para a partição desse bloco. Assim, estima-se que nessa região da função espera-se obter um *speedup* igual a 18, ou seja, espera-se que o código acelere em 18 vezes, já que ao invés de 36 modos de predição serão calculados apenas dois.

Com essa estimativa de aceleração de 18 vezes da região paralela do código e com os resultados da depuração do Valgrind que afirmam que pelo menos 73% do tempo de execução do algoritmo está nessa região paralelizável, então, através da Lei de Amdahl, pode-se esperar que seja obtido um *speedup* aproximado de 3,41 com essa implementação.

5.3 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 representativa do bloco e o modo de predição, e tem como saída o bloco resultante da transformada espacial do cosseno e do seno. A complexidade dessa função é $O(N^3)$ e executa duas multiplicações de matrizes: da matriz do bloco de entrada com a matriz da transformada, e da matriz do bloco de entrada com a matriz do inverso da transformada.

Assim como a função *Transform_Block, a* Inv*_Transform_Block Block* tem como entrada a matriz representativa do bloco e o modo de predição, mas tem como saída o bloco resultante da transformada espacial inversa do cosseno e do seno. Também tem complexidade $O(N^3)$ e executa duas transformações de matrizes, mas em ordem inversa da *Transform_Block*.

A plataforma de programação paralela *OpenAcc* permite a introdução de diretivas no código de modo a paralelizar partes de código, utilizando a capacidade de processamento da GPU. A multiplicação de matrizes é um caso 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 um único *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.

Uma implementação da função *Transform_Block* também foi realizada utilizando o paralelismo na plataforma CUDA. A principal diferença de implementações utilizando *OpenAcc* e CUDA, é que na plataforma CUDA, o programador deve gerenciar a utilização dos *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 os *threads* estão organizadas nos blocos, e como os blocos estão dispostos na grade. A implementação se torna mais complexa por causa dessas questões, e por isso, foi preferível a utilização do *OpenAcc* para a realização dos testes. No Apêndice B se encontra o código desenvolvido em CUDA para a função *Transform_Block*, mas que não foi testado.

5.4 ARQUITETURA DO CRAY XK7

O XK7 é 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 Cray XK7 disponível no laboratório do IME contém 10 lâminas 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 testes realizados a versão com MPI foi executada no ambiente GNU e a versão com *OpenAcc* foi executada no ambiente Cray. A versão com CUDA não foi testada pois atualmente o Cray ainda não encontra-se configurado para utilização do CUDA.

6 RESULTADOS DAS IMPLEMENTAÇÕES PROPOSTAS

O algoritmo VSTC e as duas versões implementadas foram testadas no conjunto de imagens disponível no anexo A, utilizando como parâmetro de entrada QP = 28. Os testes foram realizados no Cray XK7 utilizando o compilador CC.

Para análise comparativa, executou-se o algoritmo VSTC original nas imagens (a) Small, (b) RaceHorses, (d) BQSquare, (e) BlowingBubble, (f) BasketballPass, (g) Keiba, (h) Flowervase, (i) Mobisode, (j) PartyScene, (k) BQMall e (i) BasketballDrill, que se encontram disponíveis no anexo A. Foram coletados o tempo de execução da compressão e dados relativos ao desempenho da compressão em termos de taxa e distorção que se encontram disponíveis na tabela 6.1.

Tabela 6.1 – Resultado dos testes do algoritmo VSTC padrão

Imagem	Tempo de execução	PSNR	MSE	Taxa(bpp)
(a) Small	10min55s	42,06	4,04	0,33
(b) RaceHorses	62min45s	38,91	8,35	1,06
(d) BQSquare	68min29s	38,25	9,74	1,47
(e) BlowingBubble	62min29s	38,02	10,26	1,12
(f) BasketballPass	53min32s	40,00	6,50	0,66
(g) Keiba	238min33s	40,30	6,06	0,48
(h) Flowervase	158min42s	45,11	2,00	0,20
(i) Mobisode	120min00s	45,01	2,05	0,05
(j) PartyScene	428min44s	37,64	11,20	1,61
(k) BQMall	284min39s	39,36	7,53	0,76
(l) BasketballDrill	272min1s	39,23	7,77	0,56

Para melhor compreensão desses resultados, a tabela 6.2 exibe características de cada imagem, como o tamanho da imagem e com quantos blocos de partição ela foi representada. O tamanho contém as dimensões em *pixels* das imagens e o número de blocos de entrada corresponde a quantidade e blocos de tamanho 64 x 64 em que a imagem foi dividida.

Tabela 6.2 – Características das imagens de teste

Imagem	Dimensões	Nº de blocos de entrada	Nº de blocos de partição
Small	192 x 192	9	571
RaceHorses	416 x 240	28	6145
BQSquare	416 x 240	28	13655
BlowingBubble	416 x 240	28	7453
BasketballPass	416 x 240	28	4891
Keiba	832 x 480	104	10984
Flowervase	832 x 480	104	5645
Mobisode	832 x 480	104	1452
PartyScene	832 x 480	104	69802
BQMall	832 x 480	104	22917
BasketballDrill	832 x 480	104	19812

Portanto, comparando os resultados das duas tabelas, nota-se que o tempo de execução tem uma relação direta com as dimensões da imagem, ou seja, imagens de dimensões maiores consomem mais tempo para serem comprimidas. Além disso, também há uma relação entre o tempo de execução e o número de blocos de partição. Quanto maior essa quantidade de blocos, mais tempo foi gasto na estrutura de otimização recursiva do algoritmo.

A implementação do paralelismo usando o padrão MPI foi testada nas imagens (a)Small, (b) RaceHorses, (d) BQSquare, (e) BlowingBubble, (f) BasketballPass, (g) Keiba, (h)Flowervase, (i) Mobisode, (j) PartyScene, (k) BQMall. Os tempos de execução do algoritmo com o MPI se encontram na tabela 6.3.

Tabela 6.3 – Resultados dos testes da implementação com o MPI

Imagem	Tempo de execução (serial)	Tempo de execução (paralelo)	*Speedup*	Redução
(a) Small	10min55s	1min21s	8,09	88%
(b) RaceHorses	62min45s	6min49s	9,21	89%
(d) BQSquare	68min29s	7min29s	9,15	89%
(e) BlowingBubble	62min29s	6 min 43s	9,30	89%
(f) BasketballPass	53min32s	5min50s	9,18	89%
(g) Keiba	238min33s	26min32s	8,99	89%
(h) Flowervase	158min42s	25min39s	6,19	84%
(i) Mobisode	120min00s	19min34s	6,13	84%
(j) PartyScene	428min44s	48min10s	8,90	89%
(k) BQMall	284min39s	31min40s	8,99	89%

Da análise dos dados presente nas tabelas percebe-se que todas as imagens tiveram ganhos significativos de redução do tempo de processamento, e a implementação obteve uma média de 88% de redução do tempo de processamento.

A média de *speedup* foi de 8,41, acima da estimativa de 3,41 vezes que foi realizada no capitulo 5, porém o resultado ainda está condizente com a Lei de Amdahl, exposta no capítulo 4. De fato, considerando que o *speedup* resultante é obtido com uma aceleração de 18 vezes na porção paralela do código, pode-se concluir que mais de 93% do código é paralelizável de acordo com a Lei de Amdahl.

O *speedup* obtido variou mais entre as imagens. Comparando-se as tabelas 6.2 e 6.3 percebe-se que não há relação direta entre o *speedup* e o tamanho da imagem. As duas imagens que apresentaram os piores resultados de aceleração foram *Flowervase* e *Mobisode*. Na tabela 6.2, nota-se que essas duas imagens possuem um número de blocos de partição bastante inferior as demais imagens do mesmo tamanho. Observando as imagens disponíveis no anexo A, nota-se que essas imagens e também a imagem *Small* possuem regiões bastante homogêneas. Nessas regiões, os processos de otimização do modo de predição e otimização do bloco de resíduo são mais rápidos, pois como não há diferença entre particionar ou não um bloco, o processo recursivo não se prolonga até blocos menores. Isso explica porque nessas três imagens o resultado de *speedup* foi inferior ao das demais.

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 compressão em arquivos, 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.

A implementação do *OpenAcc* nas funções *Transform_Block* e *Inv_Transform_Block* foi testada nas imagens (a)Small, (b) RaceHorses, (d) BQSquare, (e) BlowingBubble, (f)BasketballPass, (g) Keiba, (h) Flowervase, (i) Mobisode, (k) BQMall, (k) BasketballDrill, disponíveis no anexo A e gerou os resultados mostrados na tabela 6.4.

Tabela 6.3 – Resultados obtidos utilizando *OpenAcc*

Imagem	Tempo de execução serial	Tempo de execução em GPU	*Speedup*	Redução
Small	10min55s	7min6s	1,54	35%
RaceHorses	62min45s	43min17s	1,45	31%
BQSquare	68min29s	48min31sec	1,41	29%
BlowingBubble	62min29s	43min3sec	1,45	31%
BasketballPass	53min32s	35min55sec	1,49	33%
Keiba	238min33s	166min29sec	1,43	30%
Flowervase	158min42s	99min56seg	1,59	37%
Mobisode	120min00s	71min45sec	1,67	40%
BQMall	284min39s	208min 6sec	1,37	27%
BasketballDrill	272min1s	191min43sec	1,42	30%

A redução do tempo na maioria das imagens foi próxima de 30%. A redução média do tempo foi de aproximadamente 32,3%. Pode-se notar que a diferença de tamanho das imagens não interferiu diretamente na redução do tempo relativo das imagens. Esse resultado é bastante condizente com a análise da execução do algoritmo VSTC exposta no capítulo 5, que indicou que as transformadas espaciais consumiam entre 30% e 50% do tempo total de processamento.

As diferenças de *speedup* entre as imagens não foi muito significante e, comparando os resultados das tabelas 6.2 e 6.4, vemos que não há muita relação entre as características da imagem e o *speedup* obtido. Porém, destaca-se que na implementação com GPU o resultado foi oposto ao do MPI, pois as imagens *Small*, *Flowervase* e *Mobisode* apresentaram os melhores resultados de *speedup*.

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 era realizado com mais rapidez em relação ao tempo de cópia de memória e mais 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 foram comparados os resultados da compressão em termos da taxa-distorção, e, verificou-se que as implementações dos paralelismos não modificaram os valores, indicados na tabela 6.1, de relação sinal-ruído (PSNR), erro quadrático médio (MSE) e taxa de compressão, em bpp.

7 CONCLUSÃO

Neste trabalho foram desenvolvidas duas versões paralelizadas do algoritmo de compressão de imagens VSTC com o objetivo de otimizar o tempo de execução do algoritmo. Três abordagens foram estudadas e implementadas, a primeira utilizando MPI, a segunda utilizando a plataforma CUDA e a terceira *OpenAcc*.

Nos capítulos 2 e 3 foram expostos conceitos básicos sobre a compressão de imagens e padrões de compressão H.264/AVC, H265, MMP e é apresentado o algoritmo VSTC. A importância desse estudo foi o melhor entendimento do algoritmo a ser otimizado.

No capítulo 4, abordou-se 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*. A importância desse estudo possibilitou, junto com a análise do resultado do *Valgrind*, no capítulo 5, o levantamento dos possíveis pontos do algoritmo a serem paralelizados, utilizando tanto o MPI, quanto a GPU.

No capítulo 5, foram propostas soluções de paralelismo propostas para o algoritmo VSTC.

No capítulo 6, foram apresentados os resultados obtidos de cada implementação. A implementação utilizando MPI teve como resultado um aumento de rapidez 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. A implementação do *OpenAcc* foi priorizada, em relação a implementação em CUDA, em virtude do código em CUDA ser mais complexo de gerenciar.

Ainda, utilizando a Lei de Amdahl para analisar os resultados obtidos foi possível concluir que ao menos 95% do tempo total de execução do algoritmo VSTC é paralelizável.

O desempenho da compressão das duas novas implementações foi o mesmo, em relação ao código original, validando ambos os algoritmos. Portanto, conclui-se que o objetivo inicial do trabalho foi atingido, com ambas as implementações de paralelismo reduzindo o tempo de execução do algoritmo.

Dessa forma, esse trabalho contribuiu para ajudar a tornar o algoritmo VSTC de compressão de imagens mais rápido auxiliando o mesmo a competir com os demais padrões existentes e também contribuindo para a compreensão do algoritmo, possibilitando futuras evoluções.

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. No entanto, os tempos de compressão obtidos com essa implementação ainda são maiores que os tempos de compressão do algoritmo H.265 para uma mesma imagem de entrada.

Como continuação dessa pesquisa, sugerimos o estudo da possibilidade de implantação de programação dinâmica no algoritmo VSTC, de modo que, o cálculo do melhor custo do resíduo para um determinado sub-bloco não seja feito mais de uma vez, que torna o tempo de execução maior.

Além disso, ainda existe a possibilidade de combinar os dois tipos de paralelismos que foram utilizados separadamente neste trabalho: MPI e GPU. Os resultados discutidos no capítulo 6 indicam que essa abordagem pode contribuir para uniformizar o *speedup* obtido dado que as duas implementações se comportaram de formas opostas de acordo com as características das imagens de entrada.

8 REFERÊNCIAS BIBLIOGRÁFICAS

[1] Image Coding Using Variable-Size Transforms and a Full Binary Tree Recursive Optimization. *Anderson V. C. de Oliveira, Nuno M. M. Rodrigues, Sergio M. M. de Faria, Eduardo A. B. da Silva, Carla L. Pagliari.2014*

[2] MACIEL, E; MACHADO, F.R.S; JUNIO, L. Ensino de arquitetura paralela ao longo da graduação em computação. Pontíficia Universidade Católica de Minas Gerais, Belo Horizonte. WEAC-2008.

[3] Disponível em: <http://www.oarquivo.com.br/index.php/curiosidades/2751-a-primeira-imagem-digital-da-historia-1957>. Acesso em 20 maio 2015

[4] Disponível em: <http://www.tecmundo.com.br/fotografia-e-design/46477-as-10-cameras-digitais-mais-importantes-da-historia.htm>. Acesso em 20 maio 2015

[5] WEI Wei-Yi, An Introduction to Image Compression, National Taiwan University, Taipei, Taiwan.

[6] GIROD, B. Image and Video Compression, Department of Electrical Engineering, Stanford University, 2000.

[7] VIJAYVARGIYA, G., SILAKARI, S., PANDEY, R. A Survey: Various Techniques of Image Compression, IJCSIS – International Journal of Computer Science and Information Security, v. 11, 2013.

[8] CAR Standards for Irreversible Compression Digital Diagnostic Imaging within Radiology, Canadian Association of Radiologists, 2011, Ottawa, Canada.

[9] DUBEY, R. B., GUPTA R. High Quality Image Compression. WebmedCentral Miscellaneous 2011.

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

[11] Disponível em <http://www.irinfo.org/04-01-2006-colbert/>. Acesso em 25 maio 2015.

[12] CAI Hua, LI Jiang. Lossless Image Compression with Tree Coding of Magnitude Levels. Media Communication Group, Microsoft Research Asia, Beijing, China. 2005.

[13] RABBANI, Majid, JONES, Paul W. Digital Image Compression Techniques, 1991, SPIE-The international Society for Optical Engineering

[14] AHMED, N., NATARAJAN, T., RAO, K. R. Discrete cosine transform. IEEE Transactions on Computers, v.-C-17, n. 1, p. 693-694, 1973.

[15] CHEN, W. H., PRATT, W.K. Scene adaptative coder. IEEE Transactions on Communications COM, v. 32, p. 225-232, 1984.

[16] WATSON, Andrew B. Image Compression Using the Discrete Cosine Transform. NASA Ames Research Center. Mathematica Journal, 4, 1994.

[17] GONZALEZ, Rafael C., WOODS, Richard E. Digital Image Processing. Second Edition. Prentice Hall, New Jersey. 2002.

[18] NELSON, Mark, GAILLY, Jean-Loup. The Data Compression Book, Second Edition. M&T Books. Nova York. 1995

[19] NETO, J.F. Compressão sem perdas de imagens digitais. Universidade Tiradentes, Aracaju, 1999.

[20] Disponível em <http://scholar.harvard.edu/stanleychan/software/subpixel-motion-estimation-without-interpolation>. Acesso em 01 julho 2015.

[21] Disponível em <http://www.axis.com/files/whitepaper/wp_h264_31808_br_0804_lo.pdf>. Acesso em 26 maio 2015.

[22] SULLIVAN, Gary J., TOPIWALA, Pankaj, LUTHRA, Ajay. The H.264/AVC Advanced Video Coding Standard: Overview and Introduction to the Fidelity Range Extensions, 2004.

[23] SARWER, Mohammed Golam, JONATHAN, Q. M. Improved Intra Prediction of H.264/AVC WU University of Windsor, Canada, 2011

[24] Disponível em <http://pro-av.panasonic.net/en/sales_o/p2/ag-hpx370/>. Acesso em 17 maio 2015.

[25] SULLIVAN, G. J.; OHM, J.-R.; HAN, W.-J.; WIEGAND, T. . Overview of the High Efficiency Video Coding (HEVC) Standard. IEEE Transactions on Circuits and Systems for Video Technology. Dezembro, 2012.

[26] TAUBMAN, D. S., MARCELIN, M.W. JPEG2000: Image Compression Fundamentals, Standards and Practice, Kluwer Academic Publishers, 2001.

[27] FILHO, E. B. L., DA SILVA, E. A. B., DE CARVALHO, M. B., JÚNIOR, W. S., KOILLER, S. J. Electrocardiographic signal compression using multiscale recurrent patterns. IEEE Transactions on Circuits and Systems I, v. 52, n. 12, p. 2739–2753, Dezembro, 2005.

[28] DE CARVALHO, M., DA SILVA, E., FINAMORE, W. Multidimensional signal compression using multiscale recurrent patterns, Elsevier Signal Processing, n. 82, p. 1559–1580, Novembro, 2002.

[29] FILHO, E. B. L., DA SILVA, E. A. B., DE CARVALHO, M. B., PINAGÉ, F. S. Universal image compression using multiscale recurrent patterns with adaptive probability model. IEEE Transactions on Image Processing, 2008.

[30] CASTANO-DIEZ, D., MOSER, D., SCHOENEGGER, A., PRUGGNALLER, S. e FRANGAKIS, A. S. Performance evaluation of image processing algorithms on the GPU. Journal of Structural Biology, 164(1):153-160, 2008. ISSN 1047-8477.

[31] CHENG, David R., SHAH, Viral B., GILBERT, John R., EDELMAN, Alan. A Novel Parallel Sorting Algorithm for Contemporary Architectures. 2007

[32] BOAVENTURA, M. Programação paralela usando MPI. UNESP. São José do Rio Preto – SP. 2008

[33] WESLEY, Addison. Introduction to Parallel Computing, 2003.

[34] IGNÁCIO, Aníbal A. Vilcapona, FILHO, Virgílio J. M. Fereira. MPI: Uma ferramenta para implementação paralela. UFRJ - Universidade Federal do Rio de Janeiro. Pesquisa Operacional, v22, n.1, 2002.

[35] SU, Mehmet F., EL-KADY, Ihab, BADER, David A., LIN, Shawn-Yu. A Novel FDTD Application Featuring OpenMP-MPI Hybrid Parallelization. International Conference on Parallel Processing – ICPP. 2004.

[36] QUINN, Michael J. Parallel Programming in C with MPI and OpenMP. The McGraw-Hill Companies

[37] GANESHAMOORTHY, K. Parallel Algorithms on Configurable Hybrid UPC/MPI Clusters. University of Colombo School of Computing. 2010.

[38] TANENBAUM, A. S.; STEEN, M. V. Distributed Systems: Principles and Paradigms. 2. ed. Prentice Hall, 2006.

[39] NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050, 3.1 edition, 5 2009.

[40] NVIDIA. Produtos NVidia. Disponível em <http://developer.nvidia.com/cuda-gpus>. Acesso em 06 maio 2015.

[41] Disponível em <http://www.karlrupp.net/wp-content/uploads/2013/06/gflops-dp.png>. Acesso em 25 maio 2015.

[42] KIRK, David B., HWU, Wen-Mei W. Programando para processadores Paralelos. Uma abordagem prática à programação de GPU. 2010.

[43] RYOO, S., RODRIGUES, C. I., BAGHSORKHI, S. S., STONE, S. S., KIRK, D. B. e HWU, W.-M. W. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. PPoPP 2008 - Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, p. 73-82, Nova York, NY, USA, 2008. ACM.

[44] COLLARES, M. Um resolvedor numérico baseado no método de Lattice-Boltzmann aplicado em unidades de Processamento gráfico. Rio de Janeiro, 2012.

9 APÊNDICES

9.1 APÊNDICE A – IMPLEMENTAÇÃO EM CUDA

```
380  void Transform_Block(int **Block, int **BY,int N, int M, int mode)
381  {
382      int l,c,m,maior,menor;
383      float result;
384      maior=max(N,M);
385      menor=min(N,M);
386
387      //Variáveis que serão usadas na GPU começam com dev
388      int *dev_Block;
389      float *dev_MatFloat_temp, *dev_INV_Float_Transform_Core;
390      float *dev_MatFloat_temp_0;
391      float *dev_Float_Transform_Core;
392      int *dev_BY;
393
394      //Alocação da quantidade de blocos e threads
395      const unsigned int totalthreads = N*M;
396      const unsigned int threadsPerBlock = 512;
397      const unsigned int blocks = ceil(totalthreads/float(threadsPerBlock));
398      dim3 dimGrid(blocks,1,1);
399      dim3 dimBlock(N,M/blocks,1);
400
401      //Alocação das variáveis na GPU
402      cudaMalloc((void**) &dev_Block,N*M*sizeof(int));
403      cudaMalloc((void**) &dev_MatFloat_temp,N*M*sizeof(float));
404      cudaMalloc((void**) &dev_BY,N*M*sizeof(int))
405      cudaMalloc((void**) &dev_MatFloat_temp_0,N*M*sizeof(int));
406      cudaMalloc((void**) &dev_Float_Transform_Core,M*M*sizeof(float));
407      cudaMalloc((void**) &dev_INV_Float_Transform_Core,M*M*sizeof(float));
```

```
408
409     /*Se No linhas > No Colunas -> Matriz alta e magra */
410     if (N>M  || N==M)
411     {
412         //Cópia dos valores do host para GPU
413         cudaMemcpy(dev_Block,Block,N*M*sizeof(int),cudaMemcpyHostToDevide);
414         cudaMemcpy(dev_INV_Float_Transform_Core,INV_Float_Transform_Core,M*M*sizeof(float),cudaMemcpyHostToDevide);
415         cudaMemcpy(dev_Float_Transform_Core,Float_Transform_Core,M*M*sizeof(float),cudaMemcpyHostToDevide);
416
417         //Chamada da função
418         cudaVarreduraHorizontalKernel_1<<<dimGrid,dimBlock>>>(dev_Block,N,M,maior,dev_MatFloat_temp[0][N],dev_INV_Float_T
419
420         //Cópia do valor modificado pela GPU
421         cudaMemcpy(dev_MatFloat_temp_0,MatFloat_temp_0,N*M*sizeof(int),cudaMemcpyHostToDevide);
422
423         //Liberação de memória na GPU
424         cudaFree(dev_Block);
425         cudaFree(dev_INV_Float_Transform_Core);
426
427         //Chamada da função
428         cudaVarreduraVerticalKernel_1<<<dimGrid,dimBlock>>>(dev_BY;,dev_MatFloat_temp[0][N],N,M,maior,dev_MatFloat_temp[1
429
430         //Cópia de memória da GPU para o host
431         cudaMemcpy(BY,dev_BY,maior*M*sizeof(int),cudaMemcpyDeviceToHost);
432
433         //Liberação de memória
434         cudaFree(dev_MatFloat_temp_0);
435         cudaFree(dev_BY);
436         cudaFree(dev_Float_Transform_Core);
437         cudaFree(dev_MatFloat_temp);
438
439     } /* End If */
```

```
__global__ void VarreduraHorizontal_1(int **dev_Block, int N, int M,int maior, int **dev_MatFloat_temp,**d
{

    int Row = blockIdx.y*title_y+threadIDx.y;
    int Col = blockIdx.x*title_x+threadIDx.x;
    float result=0

    if(Col<=M)&&(Row<=N)
    {
        for(k=0;k<M;k++)
            result+=dev_Block[Row*maior+k]*dev_INV_Float_Transform_Core[M*k+Col];
        dev_MatFloat_temp[Row*maior+Col]=result;

    }
}
```

```
__global__ void VarreduraVertical_1(int *dev_BY, float **dev_MatFloat_temp_0, int N, int M,int maior, double Q
int title_x,int title_y)
{

    int Row = blockIdx.y*title_y+threadIDx.y;
    int Col = blockIdx.x*title_x+threadIDx.x;
    float result=0

    if(Col<=M)&&(Row<=N)
{
        for(k=0;k<M;k++)
            result+=MatFloat_temp_0[Row*maior+k]*Float_Transform_Core[M*k+Col];

        dev_MatFloat_temp[Row*maior+Col]=round(result/Qstep);
        dev_BY[Row*maior+Col] = (int) dev_MatFloat_temp[Row*maior+Col] ;
}
}
```

```
    /* Se No linhas < No Colunas -> Matriz baixa e gorda */
    if (N<M)
    {
        //Cópia dos valores do host para GPU
        cudaMemcpy(dev_Block,Block,N*M*sizeof(int),cudaMemcpyHostToDevide);
        cudaMemcpy(dev_INV_Float_Transform_Core,INV_Float_Transform_Core,M*M*sizeof(float),cudaMemcpyHostToDevide);
        cudaMemcpy(dev_Float_Transform_Core,Float_Transform_Core,M*M*sizeof(float),cudaMemcpyHostToDevide);

        //Chamada da função
        cudaVarreduraVertical_2Kernel<<<dimGrid,dimBlock>>>(dev_Block,N,M,menor,dev_MatFloat_temp[1][M],dev_Float_Transform_Core[TM

        //Cópia do valor modificado pela GPU
        cudaMemcpy(MatFloat_temp[1][M],dev_MatFloat_temp,menor*M*sizeof(float),cudaMemcpyDeviceToHost);

        //Liberação de memória
        cudaFree(dev_Block);
        cudaFree(dev_Float_Transform_Core);

        //Chamada da função
        cudaVarreduraHorizontalKernel_2<<<dimGrid,dimBlock>>>(*dev_BY, dev_MatFloat_temp[0][M],N,M,menor,dev_MatFloat_temp[1][M],dev
        1,M/blocks);

        //Cópia de memória da GPU para o host
        cudaMemcpy(BY,dev_BY,menor*M*sizeof(int),cudaMemcpyDeviceToHost);

        //Liberação de memória
        cudaFree(dev_MatFloat_temp_0);
        cudaFree(dev_BY);
        cudaFree(dev_INV_Float_Transform_Core);
        cudaFree(dev_MatFloat_temp);
    } /* End If */
return;
```

```
__global__ void VarreduraVertical_2(int **dev_Block, int N, int M,int menor, int **dev_MatFl
{


    int Row = blockIdx.y*title_y+threadIDx.y;
    int Col = blockIdx.x*title_x+threadIDx.x;
    float result=0

    if(Col<=M)&&(Row<=N)
    {
        for(k=0;k<N;k++)
            result+=Block[Row*menor+k]*Float_Transform_Core[M*k+Col];

        dev_MatFloat_temp[Row*menor+Col]=result;
    }
}
```

```
__global__ void VarreduraHorizontal_2(int *dev_BY, float **dev_MatFloat_temp_0, int N,
int title_x,int title_y)
{

    int Row = blockIdx.y*title_y+threadIDx.y;
    int Col = blockIdx.x*title_x+threadIDx.x;
    float result=0

    if(Col<=M)&&(Row<=N)
    {
        for(k=0;k<M;k++)
            result+=MatFloat_temp_0[Row*menor+k]*Float_Transform_Core[M*k+Col];

        dev_MatFloat_temp[Row*menor+Col]=round(result/Qstep);
        dev_BY = (int) dev_MatFloat_temp[Row*menor+Col] ;
    }

}
```

10 ANEXOS

10.1 ANEXO A –FIGURA UTILIZADAS NOS TESTES

(l) BasketballDrill

(f) BasketballPass

(e) BlowingBubble

(d) BQSquare

(h) Flowervase

(g) Keiba

(i) Mobisode

(j) PartyScene

(b) RaceHorses

(a) Small

(c) Lena

(k) BQMall