Universidade Federal Fluminense
Centro Tecnológico
Escola de Engenharia
Mestrado em Engenharia de Telecomunicações

André Vinícius Pereira de Rezende

Implementação de um algoritmo de compressão de imagens baseado em recorrência de padrões multiescalas em CPU multinúcleo e GPU massivamente paralelas

Niterói-RJ
2010

ANDRÉ VINÍCIUS PEREIRA DE REZENDE

IMPLEMENTAÇÃO DE UM ALGORITMO DE COMPRESSÃO DE IMAGENS BASEADO EM RECORRÊNCIA DE PADRÕES MULTIESCALAS EM CPU MULTINÚCLEO E GPU MASSIVAMENTE PARALELAS

Dissertação apresentada ao Curso de Mestrado em Engenharia de Telecomunicações da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Mestre. Área de Concentração: Sistemas de Telecomunicações.

Orientador: Prof. Dr. MURILO BRESCIANI DE CARVALHO

Niterói-RJ

2010

ANDRÉ VINÍCIUS PEREIRA DE REZENDE

IMPLEMENTAÇÃO DE UM ALGORITMO DE COMPRESSÃO DE IMAGENS BASEADO EM RECORRÊNCIA DE PADRÕES MULTIESCALAS EM CPU MULTINÚCLEO E GPU MASSIVAMENTE PARALELAS

Dissertação apresentada ao Curso de Mestrado em Engenharia de Telecomunicações da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Mestre. Área de Concentração: Sistemas de Telecomunicações.

Aprovada em 07 de 2010.

BANCA EXAMINADORA

Prof. MURILO BRESCIANI DE CARVALHO - Orientador, D.Sc.

UFF

Prof. EDUARDO ANTÔNIO BARROS DA SILVA, Ph.D.

UFRJ

Prof. Dr. ALEXANDRE SANTOS DE LA VEGA, D.Sc.

UFF

Niterói-RJ

2010

A minha esposa Flávia e a nossa querida filha Carolina.

Agradecimentos

Agradeço a minha esposa Flávia pelo carinho e paciência.

Agradeço a meus pais por me ensinarem o caráter que tenho e a força perante as provações.

Agradeço ao meu orientador Murilo pelos incentivos, eterna boa vontade e inestimável ajuda para a finalização desta tese.

Agradeço ao professor Edson pelo incentivo no mestrado e pela paciência ao tirar as minhas dúvidas em Processos Estocásticos.

Agradeço aos grandes amigos Franz, José Luiz e Vagner pelas nossas intermináveis conversas sobre Engenharia. Parte de minha formação foi feita com vocês.

Lista de Figuras

Lista de Tabelas

Sumário

Resumo

O método de compressão de sinais MMP (Multi-dimensional Multiscale Parser) é baseado na recorrência de padrões multiescala onde cada bloco do sinal de entrada é aproximado por um dicionário adaptativo sendo atualizado com versão dilatadas e contraídas de concatenações dos blocos previamente codificados sendo estruturado sob a forma de uma árvore de segmentação binária. Esta classe de algoritmos tem como uma de suas características elevado consumo de recursos de processamento computacionais. Com o surgimento de técnicas e bibliotecas de software para a paralelização de aplicações em CPUs e GPUs, identificou-se a oportunidade de investigar a possibilidade de aceleração desta classe de algoritmos.

Palavras-chave: MMP. GPU. Compressão de imagens. Recorrência de Padrões Multiescala.

Abstract

The method of signal compression MMP (Multi-dimensional Multiscale Parser) is based on recurrency of multiscale patterns, where each block of the input signal is approximated by an adaptive dictionary, which is updated with expanded and compressed versions of concatenation of a previously coded block. This algorithm is structured by a binary segmentation tree. This class of algorithms has the characteristics of a high consumption of computational resources. With the advent of techniques and software libraries for the parallelization of applications on CPUs and GPUs, we identified the opportunity to investigate the possibility of acceleration of this class of algorithms.

Keywords: MMP. GPU. Imaging compression. Multiscale Recurrent Patterns.

Capítulo 1

Introdução

Nos dias de hoje a digitalização de informações tornou-se elemento primordial no funcionamento de boa parte dos equipamentos eletrônicos. Nos mais variados ramos da eletrônica digital moderna os equipamentos processam, interpretam e armazenam informações. Desta forma, o armazenamento eficiente destas informações é fundamental para um eficiente funcionamento deste equipamentos.

O armazenamento de imagens e vídeo é um dos ramos onde as pesquisas relativas ao armazenamento eficiente de informações foram mais intensas. Imagens e vídeos digitalizados, em geral, necessitam de muita memória para o seu armazenamento.

Os padrões atuais de compressão de imagem, áudio e vídeo, utilizados pela indústria, são baseados em codificações híbridas, com utilização de transformadas, tais como a DCT e *Wavelet* como método de codificação na origem com posterior aplicação de métodos de codificação por entropia nos dados resultantes. Estes padrões introduzem uma distorção pouco perceptível ao olho humano na imagem, mas, com um grande ganho de compressão dos dados. Esta distorção é decorrente da eliminação de informações de alta frequência, pois, em geral, a maior parte das informações estão armazenas nas bandas mais baixas. Por outro lado, o desempenho destes compressores degrada notavelmente para imagens híbridas de textos e paisagens, por causa da introdução de mais informações de alta frequência causadas pelo texto e pelas transições rápidas de padrões.

Em [1] e [2] e foi proposto o algoritmo denominado MMP (*Multi-dimensional Multiscale Parser*). Este método, por não utilizar operações de transformação em conjunto com a eliminação das informações de alta frequência, tem melhor adequação à imagens de textos misturadas com paisagens. O MMP é um método baseado na recorrência de

padrões multi escalas, isto é, ele representa um bloco de dados de entrada usando versões dilatadas e contraídas de um dicionário. A medida que o processamento é efetuado, o dicionário é atualizado somente com as concatenações de blocos previamente codificados.

O MMP tem um bom desempenho quando comparado aos métodos de compactação de imagens e vídeo atuais baseados em padrões. Por outro lado, seu processo de compactação exige um elevado poder computacional levando alguns minutos para realizar a compressão de uma imagem qualquer.

Tornou-se muito popular nos computadores pessoais a presença de placas de vídeo aceleradoras 3D. Com a sua evolução e conseguente aumento de complexidade e flexibilidade de aplicação, estes equipamentos passaram a ser chamados de *Graphics Processing Units* ou GPU, em português: Unidades de Processamento gráfico.

As GPUs oferecem um elevado poder computacional para execução de rotinas paralelizáveis, e estão disponíveis em praticamente todas as configurações atuais de computadores pessoais.

O objetivo desta tese é implementar a aceleração do algoritmo MMP pela utilização de uma GPU Nvidia e avaliar os ganhos e problemas desta implementação. Nesta implementação trabalharemos com a versão do MMP com o particionamento flexível.

Nesta tese abordaremos ao MMP e os padrões de compressão correntes, descreveremos a evolução, as características e ferramentas de programação das GPUs atuais, descrevermos o processo de implementação do MMP na GPU com suas respectivas técnicas de programação paralela, os problemas enfrentados e avaliaremos os resultados obtidos.

Capítulo 2

Compressão e Algoritmo MPP

2.1 Compressão de Dados

Em nossa sociedade atual, a digitalização de informações é a principal ferramenta computacional para a solução de problemas e execução de tarefas. Permitindo a transmissão e o armazenamento de informações a representação digital tornou-se parte fundamental de equipamentos eletrônicos, computacionais, de telecomunicação, médicos e de uma parte significativa dos eletrodomésticos das famílias modernas.

2.1.1 Problema

Com a utilização crescente das informações digitais a humanidade busca cada vez mais transportar as informações relativas ao nosso dia a dia e de nosso mundo para o ambiente digital. Com isso a quantidade de informação toma proporções quiméricas. Chegando ao ponto de, determinados aplicativos utilizando como fonte de informações a internet se propor a armazenar fotos de toda a superfície do planeta terra com precisão de metros. Outro exemplo, são os sites de armazenamento de vídeo para a sua exibição sob demanda.

Assim, para buscarmos a viabilização da criação de tais aplicativos e tornar a sua utilização com velocidades de respostas adequadas com as velocidades fornecidas pela internet atual foi inevitável a utilização da ciência da compressão de dados em larga escala nos dias atuais.

Nesta seção apresentaremos de forma breve exemplos de algoritmos de compressão de dados com perdas e sem perdas. Para maiores detalhes e exemplos dos codificadores apresentados consulte [32] e [33].

2.1.2 Sem perdas

As técnicas de compressão de dados sem perdas, como o nome já indica, tratam-se de formas de compressão onde não existe perda de informação. O que significa que, após a descompressão, os dados devem ser recuperados de forma idêntica aos dados originais. Existem inúmeras situações onde este tipo de técnica é fundamental. Por exemplo, citamos aplicações bancárias onde a frase *"O saldo da sua conta é de R$ 1.200,00"*, ao sofrer uma compressão devem ser recuperada sem nenhum erro na tela do usuário [32]. O problema da compressão sem perdas é bem conhecido e profundamente estudado com diversas técnicas já tendo sido propostas para a sua solução.

2.1.2.1 Conceito de Entropia

Intuitivamente nós sabemos o que é informação, pois, constantemente recebemos e enviamos informações na forma de texto, sons e imagens. Além disso, temos o sentimento que informação é uma quantidade não matemática que não pode ser precisamente definida, capturada ou medida [33].

Então, da busca realizada pela Teoria da Informação por uma forma de medir de forma precisa a quantidade de informação de um determinado grupo de dados chegou-se ao Conceito de Entropia. Entropia é a quantidade de bits média necessária para representar um alfabeto de dados, cada um elemento do alfabeto com com sua probabilidade de ocorrência no conjunto da maneira mais eficiente possível. Ela é dada pela fórmula:

$$H(s) = -\sum_{i=1}^{n} P(i) log_2 P(i) \tag{2.1}$$

Onde $P(i)$ é a probabilidade do elemento i do alfabeto e n é o número de elementos do alfabeto.

Os codificadores sem perdas são chamados codificadores de entropia.

2.1.2.2 Codificação de Huffman

A codificação de Huffman é um método de compressão que usa as probabilidades de ocorrência dos símbolos no conjunto de dados a ser comprimido, para determinar códigos de tamanho variável para cada símbolo. Ele foi desenvolvido em 1952 por David A. Huffman na época estudante de doutorado no MIT, sendo publicado no artigo *A Method for the Construction of Minimum-Redundancy Codes* [32].

Uma árvore binária, chamada de árvore de Huffman é construída recursivamente a partir da junção dos dois símbolos de menor probabilidade, que são unidos em símbolos agrupados e estes símbolos agrupados recolocados no conjunto de símbolos. O processo termina quando todos os símbolos foram unidos em símbolos agrupados, formando uma árvore binária. A árvore é então percorrida, atribuindo-se valores binários de 1 ou 0 para cada aresta, e os códigos são gerados a partir desse percurso.

2.1.2.3 Codificador Aritmético

A codificação de Huffman é simples e eficiente e produz a melhor codificação para cada símbolo individualmente. Mas, conforme demonstrado em [33], ele só produz códigos de tamanho ótimo (seus tamanhos na média são iguais a entropia) quando os símbolos tem probabilidades de ocorrência que são potências de 2 negativas (números tais como 1/2, 1/4 ou 1/8). Isso acontece pois não existe forma de atribuir um código que use frações de bit.

A codificação aritmética contorna esta dificuldade atribuindo um único longo código para todo o arquivo de entrada. Abaixo é mostrado de forma resumida como se processa esta codificação:

1. Cria-se um intervalo corrente iniciado com [0,1)
2. Para cada símbolo dos dados a serem codificados:
   (a) Particiona-se o intervalo corrente em subintervalos, um para cada simbolo. O tamanho do subintervalo associado a um símbolo é proporcional à probabilidade de que este símbolo seja o próximo elemento dos dados.
   (b) O subintervalo correspondente ao símbolo que é o próximo símbolo é selecionado como novo intervalo corrente.
3. Codifica-se os dados com o menor número de bits necessário para distinguir o intervalo corrente final de todos os outros possíveis intervalos correntes finais.

2.1.2.4 Codificação LZW

O algoritmo LZW é oriundo de uma família de algoritmos de compressão iniciada nos algoritmos LZ77 e LZ78 criados por Abraham Lempel e Jacob Ziv em 1977 e 1978 respectivamente.

Os algoritmos se LZ77 e LZ78 baseiam na construção interativa de um dicionário com os símbolos encontrados nos dados a serem comprimidos. No início o dicionário é vazio e na medida que o arquivo é lido os símbolos e sequencias de símbolos, ainda não registrados, são inseridos no dicionário [33].

No LZW o dicionário se inicia com com todos os símbolos possíveis, por exemplo, a tabela ASCII para a compressão de textos. Abaixo, é mostrado o algoritmo simplificado do LZW:

Para isso, o dicionário é inicializado com todos os símbolos do alfabeto (por exemplo, ao se usar codificação ASCII são 256 símbolos codificados de 0 a 255). Neste caso, a entrada é lida e acumulada em um *buffer* de caracteres. Sempre que a sequência contida no *buffer* não estiver presente no dicionário emitimos o código correspondente a versão anterior do *buffer* (ou seja, o *buffer* sem o último caractere) e adicionamos o *buffer* ao dicionário. O *buffer* volta a ser inicializado com o último caractere lido (o seu último caractere) e o processo continua até que não hajam mais caracteres na entrada.

1. No início o dicionário contém todos símbolos possíveis e o *buffer* é vazio;
2. Char_atual recebe próximo caractere da sequência de entrada;
3. A string *buffer*+char_atual existe no dicionário?
   (a) Se sim,
      i. *Buffer* recebe *buffer*+char_atual;
   (b) Se não,
      i. Coloque a palavra código correspondente ao *buffer* na sequência codificada;
      ii. Adicione o vetor *buffer*+char_atual ao dicionário;
      iii. *Buffer* recebe char_atual;
4. Existem mais caracteres na sequência de entrada ?
5. Se sim,
   (a) Volte ao passo 2;
6. Se não,
   (a) Coloque a palavra código correspondente ao *buffer* na sequência codificada;

(b) Fim.

2.1.3 Com perdas

A compressão com perdas tem o intuito de contornar os limites atingidos pelas compressão sem perdas explorando características dos sinais a serem compactados e as especificações de utilização do sinal recuperado. Como exemplo, podemos citar a compactação de um sinal na faixa de áudio. Como sabemos, a audição humana se situa na faixa de 20 HZ a 20 KHZ aproximadamente, então, as técnicas de compressão de áudio com perdas exploram esta características e outras tais como o tempo de resposta do ouvido humano a um estímulo sonoro. Quando um sinal que foi recuperado por uma técnica de compressão deste tipo é comparado com o sinal original ele é bem diferente e, portanto, inútil para uma aplicação onde não são aceitas distorções do sinal.

O problema da compressão com perdas ainda não é tão bem conhecido quanto o caso sem perdas e ainda possui um vasto campo de investigação e criação de novas técnicas.

As técnicas de compressão com perdas são frequentemente compostas de 2 ou 3 passos. As de 2 passos são compostas por um passo de quantização com posterior aplicação de um codificador. As técnicas de 3 passos são compostas por passos de transformação, quantização e codificação de entropia. Como exemplo de técnica de compressão de 2 passos podemos citar a codificação PCM utilizada em telefonia [29]. Como exemplo de compressão de 3 passos podemos citar o JPEG utilizado para compressão de imagens [31].

A quantização é o processo de atribuição de valores discretos para um sinal cuja amplitude varia entre infinitos valores. A transformação é uma operação sobre os dados a serem compactados com o objetivo de prepará-los para os processos de quantização e codificação de entropia obterem melhores resultados. A transformação é escolhida de acordo com o tipo dos dados que serão compactados, mas, seu objetivo principal é eliminar as redundâncias de codificação, as redundâncias espaciais ou temporais e as informações não relevantes [31].

2.1.3.1 JPEG

A compressão JPEG, conforme dito anteriormente, se trata de um algoritmo de compressão de 3 passos. Sua aplicação na sua versão básica é na compressão de imagens fotográficas, tanto que, por seus bons resultados na compressão destas imagens se tornou o padrão mais difundido para compressão de imagens na internet. Na figura 2.1 vemos um diagrama de blocos simplificado das partes principais deste padrão de compressão.

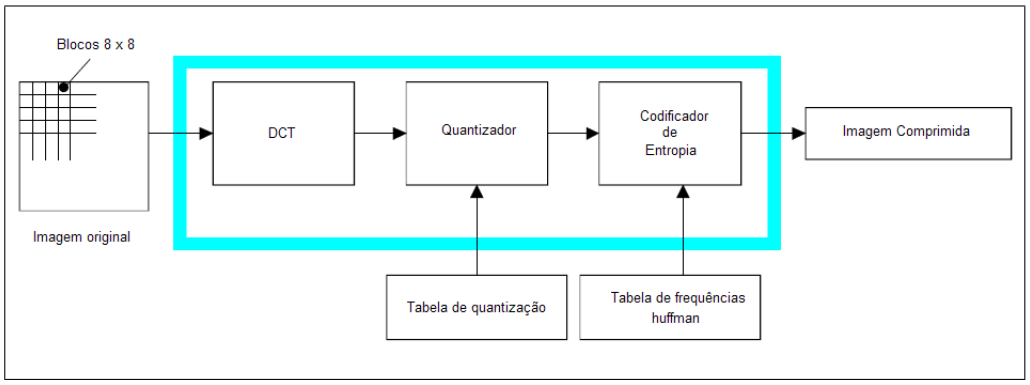

Figura 2.1: *Diagrama simplificado de um codificador JPEG*

A operação de transformação é a DCT - *Discrete Cosine Transform*. A DCT é uma operação de decomposição de um grupo finito de pontos em termos de um somatório de cossenos oscilando em diferentes frequências espaciais. A principal característica desta transformada é deslocar as informações dos dados para primeiras componentes do vetor, dessa forma, os descorrelacionando e preparando para uma melhor aplicação da quantização e codificação por entropia.

A transformada ótima para uma melhor concentração de energia nas primeiras componentes da imagem é a KLT (Karhunen-Loève Transform), que depende de informações estatísticas do sinal. A transformada DCT é mais simples que a KLT e possui resultados muito próximos. A dificuldade na utilização da KLT está no tempo adicional para calcular os autovetores (base) da matriz de correlação, o que muitas vezes inviabiliza a sua utilização em aplicações onde o tempo de resposta é importante [26]. Outra desvantagem da KLT é a obrigatoriedade da transmissão da matriz de transformação, pois, a mesma é dependente do sinal.

O processo de compressão é feito da seguinte forma:

1. A imagem original é dividida em blocos 8x8 e cada um será transformado segundo uma DCT

2. Cada bloco transformado é em seguida quantizado. A quantização é escalar, com um passo por coeficiente obtido chamado de matriz de quantização

3. Nestes blocos quantizados é aplicado um codificador de entropia misto que explora características típicas de disposição das componentes dos blocos.

Conforme recomenda o padrão [30], no processo de codificação as componentes RGB da imagem são agrupadas em blocos 8×8 e cada bloco é transformado plea DCT direta em um grupo de 64 valores chamados de coeficientes DCT. O primeiro é chamado de coeficiente DC e os outros 63 coeficientes AC. Cada um dos 64 coeficientes então é quantizado usando um dos 64 valores da tabela de quantização. A tabela de quantização pode ser padronizada ou determinada de acordo com as características da imagem a ser comprimida, dispositivo de exibição ou condição de exibição. Após a quantização, o coeficiente DC e os 63 coeficientes AC são preparados para a codificação de entropia como é mostrado na figura 2.2.

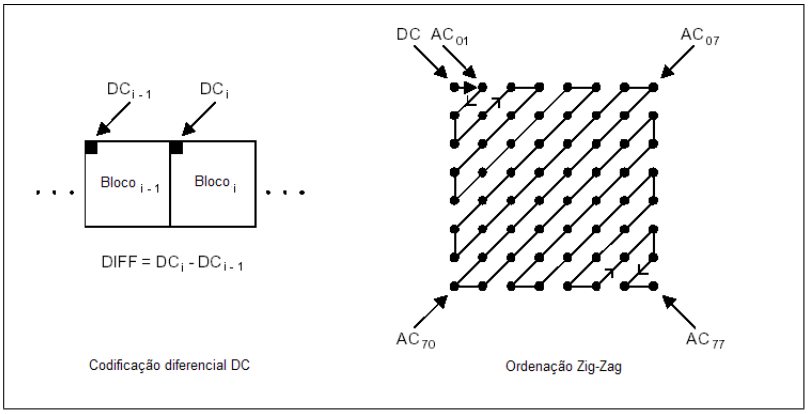

Figura 2.2: *Preparação do bloco a ser codificado*

O coeficiente DC anteriormente quantizado é utilizado para predizer o coeficiente corrente e sua diferença é codificada. Os 63 coeficientes AC quantizados são convertidos em uma sequencia zig-zag como mostrado na figura 2.2. Então, os coeficientes quantizados são passados para um procedimento de codificação de entropia duplo. Primeiro os coeficientes repetidos são agrupados por uma codificação *run-length* e a codificação dos agrupamentos são codificados por uma codificação de *huffman* que se utiliza de uma tabela pré-definida [30].

2.1.3.2 JPEG 2000

De acordo com [28], conforme ocorria a evolução da tecnologia, se tornou claro que o padrão JPEG não estaria evoluindo de forma adequada para atender as necessidades da época. A variedade de áreas de aplicação do JPEG levou a uma confusão de quais seriam os objetivos a serem alcançados na sua evolução. Estava claro o desenvolvimento da tecnologia só seria possível com uma mudança radical no padrão. O grupo de trabalho do JPEG2000 buscou criar um padrão que resolve-se aos seguintes problemas existentes nos padrões da época.

1. Boa qualidade de imagem a baixas taxas bits. O JPEG para médias e altas taxas de bit possui boa qualidade mas a baixas a distorção visual (subjetiva) se torna inaceitável.

2. Compressão com e sem perdas na mesma figura

3. Manipulação de imagens maiores de 64k x 64k sem precisar segmentar a imagem como o JPEG necessita

4. Modo único de compactação em confronto aos 44 modos do JPEG cuja grande maioria não é usada

5. Robustez a ruídos de transmissão ao contrário do JPEG que foi criado antes das transmissões sem fio

6. Boa performance com imagens geradas por computador

7. Documentos compostos por imagem e textos

O padrão JPEG2000 possibilita a utilização de compressão com e sem perdas. O padrão de compressão JPEG2000 é composto dos estágios exibidos na figura 2.3.

No primeiro estágio especificado na figura 2.3 é feito o pré-processamento da imagem original. Este pré-processmento é composto de três sub-estágios mostrados na figura 2.4. Estes passos são necessários para preparar os dados para transformada *wavelet* discreta.

Existem casos onde a imagem a ser codificada é maior do que a memória disponível para o codificador. Para resolver este problema o JPEG2000 disponibiliza uma operação

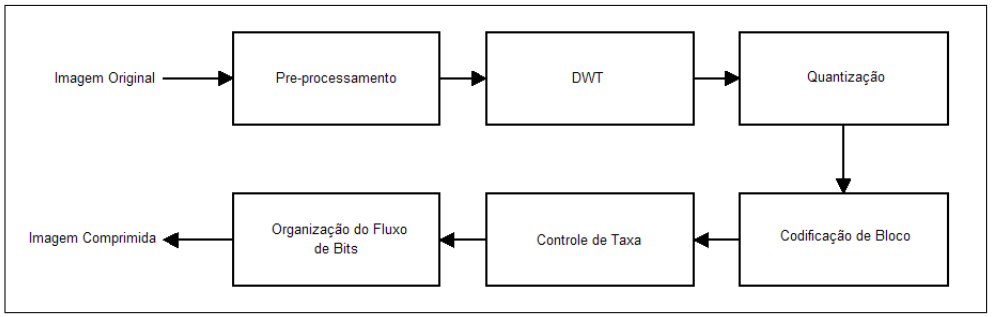

Figura 2.3: *Codificador JPEG2000*

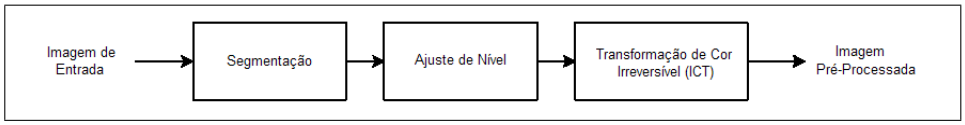

Figura 2.4: *Pré-Processamento de imagem do codificador JPEG2000*

opcional chamada *tiling* que traduziremos como segmentar. Nesta segmentação a imagem é particionada em segmentos de igual tamanho. Cada segmento é comprimido de forma independente com seus próprios parâmetros de compressão.

O JPEG2000 espera que os dados em entrada tenham a sua faixa dinâmica centrada em torno de zero por isso da utilização do bloco de ajuste de nível.

No JPEG2000 o modelo de imagem é composto de um a 214 componentes e cada componente consiste em uma matriz de amostras que representam a luminosidade do componente neste ponto.

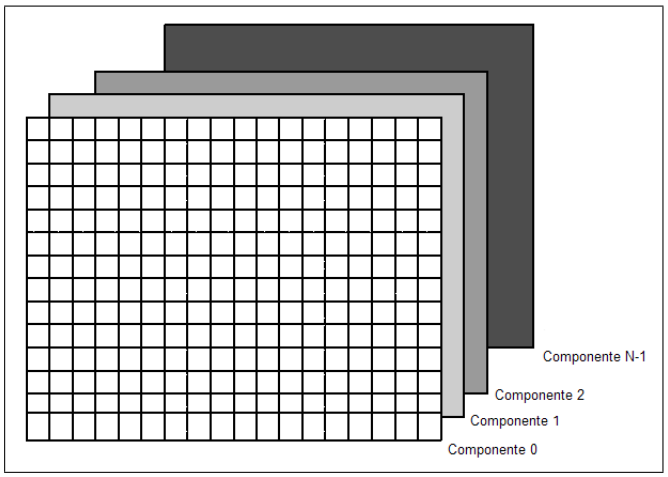

Figura 2.5: *Componentes do modelo de imagem do JPEG2000*

Cada amostra é um valor inteiro podendo ter ou não sinal podendo ter de 1 a 38 bits por amostra.

O JPEG2000 utiliza o modelo de cores Y, C_r e C_b que, apesar de apresentar menos independência estatística que o modelo RGB, essas componentes podem ser comprimidas com maior eficiência. A tranformação do modelo RGB para o modelo YC_rC_b é feita de forma irreversível, por problemas de arredondamento, por uma matriz de transformação descrita na equação (2.2).

$$\begin{bmatrix} Y \\ C_r \\ C_b \end{bmatrix} = \begin{bmatrix} 0.299 & 0.586 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{2.2}$$

O JPEG2000 utiliza a transformada *wavelet* discreta (DWT) para decompor cada segmento de imagem nas suas suas sub-bandas de frequência alta e baixa. A DWT é feita em cada linha e coluna da imagem pré-processada pela aplicação de um filtro passa-alta e um passa-baixa. Como este processo duplica o número de amostras, então, a saída de cada filtro é sub-amostrada por um fator de 2.

No JPEG2000 múltiplos estágios da DWT são realizados. Em geral, nas imagens fotográficas são utilizados de 4 a 0 estágios de DWT. Quando 1 estágio de DWT é realizado a imagem é dividida nas sub-bandas LL, HL, LH e HH. Estas sub-bandas são referentes aos seguintes grupos de filtros:

1. LL: sub-bandas passa-baixa resultantes da filtragem das linhas e colunas
2. HL: sub-banda passa-alta resultante da filtragem das linhas e sub-banda passa-baixa resultante da filtragem das colunas
3. LH: sub-banda passa-baixa resultante da filtragem das linhas e sub-banda passa-alta resultante da filtragem das colunas
4. HH: sub-bandas passa-alta resultantes da filtragem das linhas e colunas

Apenas a sub-banda LL é subdividida no próximo estágio de DWT. Na figura 2.6 são mostradas as bandas resultantes 2 estágios de DWT aplicados sobre uma imagem.

Podem ser utilizadas transformadas DWT reversíveis de números inteiros para inteiros e não reversíveis de números reais para reais. A implementação da DWT é feita através de filtragem de sinais e utiliza-se uma extensão simétrica nas fronteiras da imagem para reduzir artefatos nas bordas da imagem e obter um banco de filtros de reconstrução

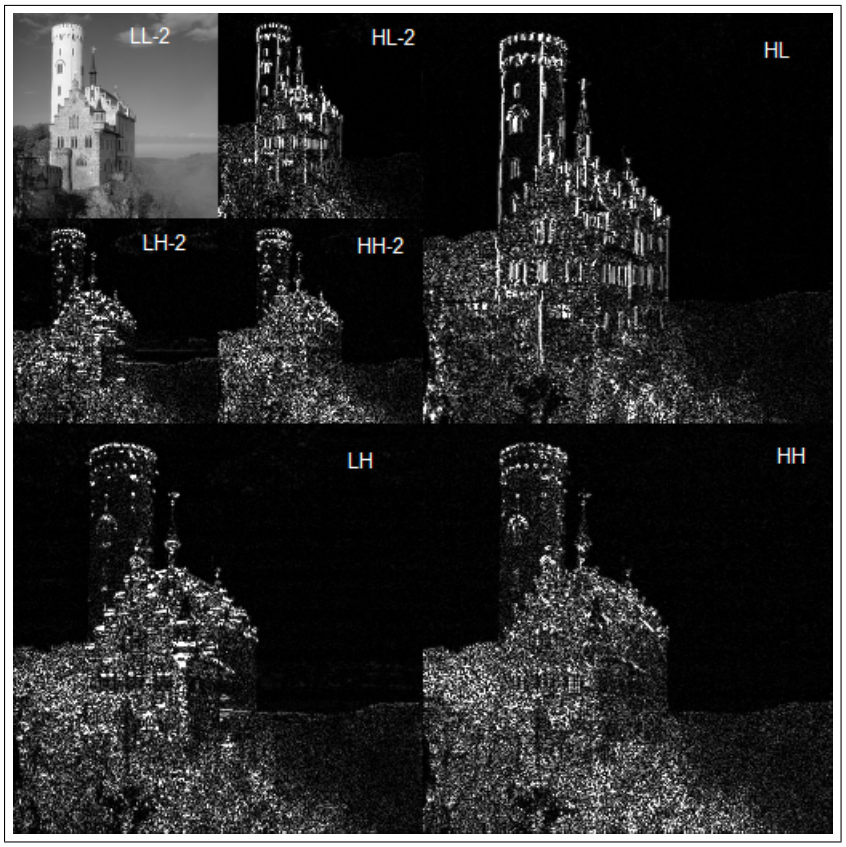

Figura 2.6: *Bandas resultantes da aplicação de 2 estágios de DWT*

quase perfeita. A transformada DWT padrão é implementada através filtros bi-ortogonais [25] que utilizam a família de *wavelets Daubechies* 9-tap/7-tap.

Após a DWT todos os coeficientes são quantizados. A quantização é feita dividindo-se cada coeficiente por um passo de quantização e o arredondando para baixo. Cada passo de quantização pode ser modificado para se obter um determinado nível de qualidade chegando ao ponto de ser sem perdas com a utilização do passo e de uma *wavelet* adequada para esse objetivo.

Após a quantização é feita em cada sub-banda uma operação chamada de *packet partition* [27] que chamaremos de partição de pacotes. Cada partição em pacote possui, sucessivamente, nível de resolução aprimorado de uma imagem ou segmento. Desta forma, inicialmente a imagem é dividida em sua aproximação de baixa qualidade em relação a original que é aprimorada até atingir seu máximo nível de qualidade. Então, os blocos de código são obtidos dividindo cada partição em pacote em retângulos regulares não sobrepostos. Estes blocos de código são utilizados na codificação de entropia.

A codificação de entropia é feita de forma independente em cada bloco de código. Esta codificação é feita com uma codificação aritmética binária dependente de contexto dos planos de bits. Cada plano de bit é codificado em 3 passos. A decisão de qual passo para um dado bit a ser utilizado na codificação é baseado na significância do bit e na significância das regiões vizinhas. Uma região é considerada significante se foi codificado 1 para esta região no plano de bits corrente ou no anterior [27].

O primeiro passo em um novo plano de bits é chamado de passo de propagação de significância. Um bit é codificado neste passo se sua localização não é significante, mas, se pelo menos um de seus oito vizinhos conectados são significantes.

O segundo passo é o passo de refinamento de magnitude. Neste passo, todos s bits que se tornaram significantes no plano de bits anterior são codificados. O terceiro passo é o passo de limpeza, o qual cuida de todos os bits que não foram codificados nos dois passos anteriores.

Após a codificação de entropia a imagem esta pronta para ser armazenada como uma versão comprimida da original [27].

2.1.3.3 Compressão de Vídeo

Os padrões atuais de compressão de vídeo foram construídos a partir da tecnologia de compressão adotada no JPEG. Portanto, adotando na codificação de cada quadro as operações de transformação, quantização e codificação de entropia.

Nas técnicas de compressão de vídeo é utilizado o recurso da subamostragem de croma. Este recurso explora a características da visão humana de perceber menos detalhes de cores comparada à melhor resolução visual para imagens monocromáticas compostas apenas por luminância (Y) [24].

Abaixo são listadas as nomenclaturas utilizadas para as diferentes formas de subamostragem comumente utilizadas em codificação de vídeo:

1. Formato 4:4:4, não há subamostragem, ou seja, para cada amostra Y existe uma amostra C_b e uma amostra C_r correspondentes;

2. Formato 4:2:2, a resolução dos dois sinais de croma é reduzida pela metade horizontalmente;

3. Formato 4:2:0, a resolução dos dois sinais de croma é reduzida pela metade tanto

horizontalmente quanto verticalmente, ou seja, para cada quatro amostras Y existe uma amostra C_b e uma amostra C_r.

Em diversos padrões de codificação de vídeo, a unidade básica de codificação é o macro bloco. Então cada quadro de vídeo é particionado em macro blocos antes de ser aplicado o processo de compressão. Um macro bloco corresponde a uma matriz de 16×16 pixels. Por exemplo, no formato 4:2:0, cada macro bloco consistirá de uma matriz de 16×16 amostras de luminância e duas matrizes de 8×8 amostras crominância. A divisão de cada quadro em macro blocos é a preparação para aplicação dos processos envolvidos na codificação de vídeo, como a compensação de movimento e a transformação. Como cada macro bloco possui características distintas, essa divisão permite que cada diferente bloco seja codificado com os parâmetros e coeficientes mais apropriados [24].

A compensação de movimento busca explorar o fato de que uma sequencia de quadros de vídeo tende a ser composta de quadros muito parecidos. Nesta técnica busca-se em algum quadro anteriormente codificado, blocos que melhor representem cada bloco do quadro atual. Calcula-se então um quadro resíduo, composto pela diferença entre os blocos atuais e aqueles escolhidos, nos quadros de referencia, para aproximá-los. Como estes quadros são semelhantes esta subtração tenderá a zerar todos os pixels. Estes pixels com valores mais próximos de zero melhoram ainda mais o desempenho da compressão realizada nos passos de transformação, quantização de codificação de entropia.

A estimação de movimento busca descobrir movimentos dos blocos de imagem nos quadros anteriores. O resultado desta busca é um vetor de movimento de duas componentes que diz qual bloco de quadros anteriores deve ser subtraído um bloco atual. Nos padrões mais modernos, esta busca podem ser realizadas em blocos de diferentes tamanhos. Além disso, a estimação de movimento é um processo de predição inter quadros. No processo de evolução da técnica, passou a existir também um processo de predição intra quadros que utiliza os pixels de um mesmo quadro na busca do melhor bloco (Padrão H.264).

Baseado nas tecnologias descritas nessa seção foram criados os padrões de vídeo atuais [22]. Em 1988-1990 o ITU-T criou o H.261 com o objetivo de atender aos mercados de videoconferência e videotelefonia sobre redes ISDN. A sua taxa de transmissão é baseada em em multiplos de 64 Kbps até 30x64 Kbps. Os quadros suportados por esse padrão são o CCIR 601 CIF (352 x 288) e o QCIF (176 x 144) utilizando a subamostragem de

croma 4:2:0. Ele utiliza dois tipos de quadros os *Intra-frames* (I) e os *Inter-frames* (P). Os quadros I são basicamente um ponto de sincronização e usam a codificação JPEG. Os quadros P utilizam diferenças com relação ao anterior sendo chamados quadros preditos. Pelo fato do envio de quadros I são evitados erros de propagação. O controle de taxa de bits é feito em um *feedback* feito por um *buffer* que sinaliza que está cheio, fazendo que o codificador diminua o fator de quantização para reduzir a taxa de bits [22].

Em 1993 a ISO/IEC apresentou o MPEG-1 parte 2 que foi muito utilizado nas soluções de Video-CD. MPEG quer dizer *Moving Picture Coding Experts Group* que se estabeleceu em 1988 para criar padrões para distribuição de vídeo e áudio. O MPEG-1 tinha como objetivo exibir qualidade de VHS em um CD-ROM (352 x 288 + CD audio @ 1.5 Mbits/sec). Este padrão é dividido em 3 partes: Vídeo, Áudio e Sistemas (controle de segmentação e empacotamento dos fluxos de vídeo e áudio). O MPEG Vídeo endereça ao problema de codificação de uma informação não disponível no quadro de referência anterior. Como exemplo citamos um macro bloco que no quadro atual está no escuro e não tem uma boa referência no anterior, mas, poderá ser encontrada uma boa referência no próximo quadro. Resolver este problema foi criado o quadro bidirecional B que procura macro blocos de referência nos quadros passados e futuros. Em geral uma típica sequencia de quadros é IBBPBBPBB IBBPBBPBB IBBPBBPBB. Basicamente as suas diferenças ao H.261 são: Grandes lapsos entre os quadros I e P obrigando a aumentar a distância de procura dos vetores de movimento, para melhorar a codificação possibilita a utilização de vetores de movimento de meio pixel, sintaxe do fluxo de bits possibilitando acesso randômico e aceleração para frente e para trás na exibição, criação das fatias para sincronização após perda ou corrupção de dados e macro blocos dos quadros B podendo utilizar dois vetores de movimento um para quadros anteriores e outro para posteriores e o resultado sendo feito pela media dos quadros. Na média o MPEG-1 tem os seguintes desempenhos de compressão: Quadros I compressão de 7 para 1, quadros P compressão de 20 para 1, quadros B compressão de 50 para 1 obtendo uma média de compressão de 27 para 1.

Em 1995 o ISO/IEC e conjunto com o ITU-T apresentaram o o H.262/MPEG 2. Este padrão é o utilizado nas aplicações de DVD Vídeo , Blu-Ray , Digital Video Broadcasting, SVCD. Este padrão foi desenvolvido para a TV digital, atendendo aos requisitos do HDTV e DVD.

Na tabela 2.1 são exibidas as características dos *profiles* (níveis) do MPEG-2.

Nível	Resolução	Pixels/Sec	Mbps	Aplicação
Low	352 x 288 x 30	3 M	4	Equivalente ao VHS
Main	720 x 576 x 30	12 M	15	TV profissional
High 1440	1440 x 1152 x 60	96 M	60	HDTV para consumidor
High	1920 x 1152 x 60	128 M	80	Produção Cinematográfica

Tabela 2.1: *Características dos níveis MPEG-2.*

Em 1996 o ITU-T apresentou o H.263 como um novo padrão para vídeo em baixas taxas, videoconferência, videotelefonia e vídeo em telefones móveis (3GP). Apresentou as seguintes inovações em relação ao H.261: Precisão de meio pixel na compensação de movimento, vetores de movimento sem restrições de área de atuação, codificação aritmética, predição avançada e quadros B e suporte a quadros 4CIF 704 x 576 146 Mbps e 16CIF 1408 x 1152 583 Mbps.

Em 1999 o ISO/IEC apresentou o MPEG-4 parte 2 cujas maiores aplicações foram o vídeo na internet, e os codificadores DivX e Xvid. A Versão 1 foi aprovada em outubro de 1998 e a Versão 2 foi aprovada em dezembro de 1999. O objetivo original do padrão era atender aos requisitos de taxas muito baixas de comunicação (de 4.8 a 64 Kb/s). Atualmente atende às seguintes taxas: Vídeo de 5 Kb até 5 Mb por segundo e áudio de 2 Kb até 64 Kb por segundo. É apresentado de conceito de *Visual Objects* e *Video Object Plane* (VOP). Objetos podem ser de qualquer formato, VOPs podem se sobrepor ou não, suporta escalabilidade de conteúdo, suporta interatividade baseada nos objetos e canais individuais de áudio podem associados com determinado objeto. Padrão adequado para composição de vídeo, segmentação e compressão; linguagem de modelagem de realidade virtual em rede e sistemas de comunicação audiovisual, tais como, interface texto para fala e animação facial. Outros padrões estão sendo desenvolvidos para codificação do formato dos objetos, codificação de movimento dos objetos e textura.

Em 2003 o ISO/IEC e o ITU-T apresentaram o padrão H.264/MPEG-4 AVC. Desenvolvidos nos equipamentos de Blue-Ray, distribuição de vídeo digital, vídeo no Apple iPod e HD DVD.

2.1.4 Quantização

Quantização é o processo de representar um conjunto de amostras de uma função contínua com um número finito de valores. Se cada amostra é quantizada de forma independentemente, então, o processo é chamado de quantização escalar. Um quantizador escalar $Q(.)$ é definido em termos de um grupo finito de níveis de decisão d_i e níveis de reconstrução r_i expresso pela equação (2.3), onde L é o número de estados de saída [22].

$$Q(s) = r_i, ses \in (d_{i-1}, d_i], i = 1, ..., L \tag{2.3}$$

2.1.4.1 Uniforme

Na quantização uniforme os níveis de reconstrução r_i são igualmente espaçados [22].

2.1.4.2 Não uniforme

Na quantização não uniforme os níveis de reconstrução r_i não são igualmente espaçados. Os valores de r_i e de d_i são determinados de modo a otimizar algum critério de desempenho, por exemplo, uma técnica de projeto que minimiza o erro quadrático médio é a técnica de Lloyd-Max [22].

2.1.4.3 Vetorial

A quantização vetorial se refere a quantização de um vetor de amostras em um número finito elementos de um dicionário. Tem sido observado que a quantização vetorial possui inúmeras vantagens na quantização de sinais de fala e imagem com relação à quantização escalar. Pelo fato de fazer um efetivo uso das dependências estatísticas dentre as amostras de dados a serem quantizados, a quantização vetorial provê uma menor distorção para um número fixo de níveis de reconstrução ou um menor número de níveis de reconstrução para um determinado nível de distorção, quanto comparada à quantização escalar. Por outro lado, este melhor desempenho é contrabalançado por um maior custo computacional e de requisitos de memória [22].

Aplicações práticas da quantização vetorial apresentam uma desvantagem: complexidade computacional na busca do vetor referência. O algoritmo tradicional para projetar *codebook* é o algoritmo LBG (*Linde Buzo Gray*) [23], que o faz através de um processo iterativo que otimiza a função custo associada a uma distorção (erro médio quadrático).

Desvantagens do LBG incluem a necessidade de escolher previamente o tamanho do codebook e problemas de mínimos locais durante atualizações do algoritmo.

2.2 Compressão por Recorrência de Padrões Multi Escalas

Os métodos de compressão baseados na recorrência de padrões multi escalas [2] *Multidimensional Multiscale Parser* (MMP), possuem como uma de suas características, a não utilização, a princípio, de transformadas no momento de sua codificação na origem. Esta característica representa uma interessante quebra no paradigma e pressuposto que as imagens ou dados a serem processados são suaves (a maior parte da energia estaria concentrada em frequências mais baixas), ou seja, que possuiriam pouco conteúdo de alta frequência, o que pode limitar a utilização dos métodos baseados em transformada.

Nos diversos trabalhos em que foram implementados protótipos desta nova classe de algoritmos, foi observada a sua aplicabilidade em diversos tipos de dados tais como áudio, imagem, vídeo, imagens de radar, imagens estéreo [11]. e sinais biomédicos. Além disso, como será abordado neste trabalho, algumas variações deste método mostraram ótimo desempenho com imagens chegando a superar padrões tais como o JPEG-2000 [44].

Então, os métodos de compressão baseados na recorrência de padrões multi escalas se apresentam como uma classe completamente nova de algoritmos de compressão, com comportamento universal, adaptativo e não fortemente baseados em asserções de suavidade ou na suposição a respeito de estruturas presentes nos dados a serem comprimidos.

Neste capítulo discutiremos o funcionamento e a aplicação destes métodos na compressão de imagens, apresentando algumas de suas variações e fazendo uma comparação com a tecnologia atual.

2.2.1 Algoritmo Multidimensional Multiscale Parser

Como já dito anteriormente, este método é um algoritmo não baseado na tríade tranformação-quantização-codificação. A ele nos referimos como MMP, pois, é utilizado um dicionário adaptativo de vetores de tamanho variável aproximados.

Este vetores (também chamados de blocos de imagem) são o resultado de uma análise recursiva dos blocos originais da imagem. Em cada bloco é realizada uma segmentação onde o mesmo é dividido em outros dois de mesmo tamanho por uma direção pré-estabelecida. Transformações de escala são utilizadas para redimensionar cada elemento do dicionário durante a sua análise comparada com um bloco de entrada da imagem original.

2.2.1.1 MMP unidimensional

O algoritmo MMP foi pela primeira vez apresentado em [2]. Ele se baseia na aproximação de vetores de dados de uma escala l, utilizando vetores oriundos de um dicionário adaptativo D^l.

Para cada vetor X^l dos dados, o algoritmo inicialmente procura no dicionário pelo elemento S_i^l que minimiza a função de custo Lagrangiano $J(T) = D(X^l, S_i^l) + \lambda R(S_i^l)$, onde $D()$ é a função soma do quadrado das diferenças e $R()$ é a taxa necessária para codificar este vetor aproximado. O índice l indica que o vetor X^l pertence a escala l, que corresponde a um vetor de tamanho (2^L)

O algoritmo então segmenta o vetor original em 2 vetores X_1^{l-1} e X_2^{l-1} , cada um com a metade dos comprimento do vetor original e procura na escala $(l-1)$ pelos elementos $S_{i_1}^{l-1}$ e $S_{i_2}^{l-1}$ que minimizam as funções custo de cada um deles.

Os custos de cada um dos passos anteriores são então avaliados e o algoritmo decide se segmenta ou não o vetor original. Cada vetor não segmentado da escala l é aproximado por um elemento S_i^l do dicionário D^l

Se um vetor é segmentado, então o mesmo procedimento é aplicado recursivamente para cada novo segmento. O particionamento ótimo de um vetor é representado por uma árvore de segmentação binária, que é codificada por *flag* binários: onde o *flag* '0' representa os nós da árvore ou as segmentações de vetores e o *flag* '1' representa as folhas da árvore (sub vetores que não foram segmentados).

No fluxo de bits final, cada *flag* representando uma folha é seguido de um índice que identifica o índice do vetor do dicionário que deve ser utilizado para representar este sub-vetor.

Todas estas informações são codificadas utilizando um codificador aritmético adaptativo com diferentes contextos para cada nível da árvore (que corresponde a uma escala

do dicionário).

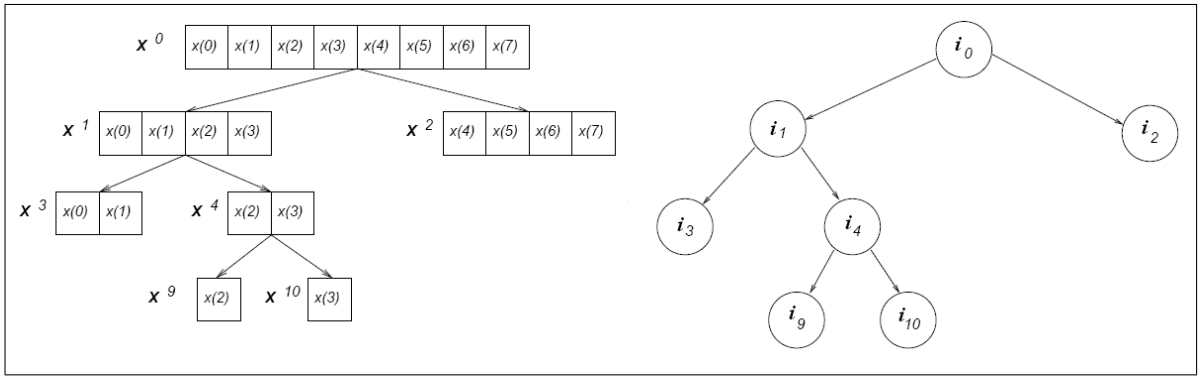

Figura 2.7: *Segmentação de um vetor de tamanho 8 e a árvore binária correspondente*

Na figura 2.7 é representada a segmentação de um vetor e sua correspondente árvore de segmentação. Neste exemplo, $i_0, ..., i_{10}$ são os índices escolhidos para codificar cada sub-vetor. Nele, a lista de símbolos resultantes da codificação é: 0 0 1 i_3 0 1 i_9 1 i_{10} 1 i_2

O uso de um dicionário adaptativo é uma importante característica do MMP. Toda segmentação de um vetor da escala l origina um novo padrão formado pela concatenação de dois vetores do dicionário da escala $l-1$. Este novo vetor é utilizado para atualizar os dicionários de todas as escalas. Com o objetivo de atualizar o dicionário da escala s utilizando o elemento da escala l, é utilizada uma transformação de escala reversível T_l^s para ajustar o vetor a esta escala.

Deve-se notar, que este processo de adaptação não gera mais dados a serem codificados, pois, o decodificador utiliza os *flags* de segmentação e os índices do dicionário para manter uma cópia sincronizada do dicionário, possibilitando uma correta reconstrução dos dados.

2.2.1.2 MMP bidimensional com particionamento flexível

Existem muitas maneiras diferentes de generalizar a segmentação unidimensional para o caso bidimensional. Versões iniciais do MMP utilizavam uma segmentação com uma regra que escolhia uma direção de segmentação preferencial a cada nível da árvore de segmentação. Por exemplo, um bloco 8×8 poderia ser subdividido em dois 8×4, mas não em dois 4×8, e assim sucessivamente resultando nas escalas: 8×8, 8×4, 4×4, 4×2, 2×2, 2×1 e 1×1.

Contudo, resultados experimentais demonstraram que a direção de partição de um bloco (a saber: vertical ou horizontal) pode prover um ganho significante nos resultados da compressão. Ganhos de até 0.4 dB em *peak signal-to-noise ratio* (PSNR) são encontrados em algumas imagens.

Para uma imagem $m \times n$ o valor PSNR é definido por:

$$PSNR = 20log_{10}\left(\frac{MAX_I}{\sqrt{MSE}}\right) \tag{2.4}$$

onde MAX é o valor máximo possível de um pixel e MSE (*Mean Square Error*) é definido por:

$$MSE = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\|I(i,j) - K(i,j)\|^2 \tag{2.5}$$

Estas observações motivaram o desenvolvimento de um novo modo de segmentação para o MMP, onde a direção de partição do bloco de entrada é adaptativamente selecionada. Este novo método se mostrou mais eficiente para aproveitar estruturações dentro da imagem e atingir um bom desempenho de codificação em um ampla variedade de tipos de imagem.

No MMP com particionamento flexível, qualquer bloco pode ser segmentado a direção vertical ou horizontal. O que define esta direção é um critério de taxa distorção $(R - D)$ local. Antes de sua codificação cada bloco de imagem X^l é segmentado nas direções vertical e horizontal. Este procedimento é aplicado recursivamente para cada nó filho até o nível 0 da árvore de segmentação ser alcançado. Isso significa que uma ampla variedade de dimensões de blocos se tornam disponíveis para a compressão. Como exemplo podemos citar o caso de um bloco de tamanho 16×16, no processo de segmentação tradicional apenas são utilizados blocos quadrados e retangulares com as proporções $2:1$, gerando um total de 9 diferentes escalas $(16 \times 16, 16 \times 8, 8 \times 8, ..., 1 \times 1)$.

No novo padrão de segmentação podem ser gerados blocos em 25 possíveis escalas $(16 \times 16, 16 \times 8, 8 \times 16(nova), 16 \times 4(nova), 4 \times 16(nova), ..., 1 \times 1)$, sem restrição de proporções, significando que todos os blocos com as dimensões $2^m \times 2^n$ são disponíveis, para $m, n = 0, ..., 4$.

Na codificação do final ao início da árvore de segmentação o valor da função custo Lagrangiano é avaliada e a opção com o menor custo é a escolhida. Se a decisão de segmentar um bloco em determinada direção é tomada, os nós filhos gerados na outra

direção da árvore de segmentação são podados. Se o menor custo Lagrangiano corresponde a decisão de não segmentar, todos os nós filhos são podados. Um *flag* adicional de segmentação é utilizado para indicar a direção de segmentação.

Este novo método de particionamento também proporciona a utilização da geometria dos blocos em favor do processo de predição, como exemplo, podemos citar a utilização de blocos bem estreitos, tal como, um bloco 16×1 para retratar um detalhe vertical, onde ele tende a gerar um sinal de predição mais preciso, em comparação com um bloco 4×4 (que, por sinal, tem o mesmo número de pixels). Nesta tese, esta foi versão do MMP escolhida para a sua implementação na GPU.

Capítulo 3

GPU e a biblioteca Cuda

3.1 Introdução

A presença das *Graphics Processing Units* ou GPU, em português: Unidades de Processamento gráfico, nos computadores pessoais atuais, se tornou muito comum devido a popularização dos jogos 3D, chegando ao ponto de estarem presentes em praticamente todos os computadores vendidos, mesmo nos modelos onde o circuito controlador de vídeo se encontra embutido na própria placa mãe.

Devido a esta onipresença e levando em conta o elevado poder computacional destes dispositivos, levantou-se a questão de como utilizar estes recursos mesmo nas máquinas onde os usuários não as utilizam para jogos ou aplicações gráficas 3D. Buscando flexibilizar a utilização destes dispositivos surgiu o conceito de *General Purpose computing on GPU* ou Computação de Propósito Geral em GPU [35] que busca implementar recursos de hardware e ferramentas de programação que flexibilizem a utilização das GPUs tornando possível a sua utilização nas mais diversas aplicações.

Um dispositivo GPU possui uma arquitetura muito diferente de uma CPU e isso afeta diretamente a programação do mesmo. Primariamente, GPUs são extremamente paralelos, possuem mais de um núcleo e uma imensa quantidade de ULAs menos sofisticadas que uma CPU, diversas caches e uma memória dedicada que é compartilhada por todos os núcleos.

A GPU também se enquadra em um modelo SIMD de processamento, o que significa que o mesmo conjunto de instruções será executado paralelamente em todos processadores. Isso implica na necessidade de as aplicações desenvolvidas serem paralelizáveis,

ou seja, qualquer solução que não se enquadra nesta especificação simplesmente não é interessante de ser desenvolvida ou portada para essa tecnologia, ou por excesso de complexidade ou por baixo ganho de desempenho. Além disso, a GPU possui um pipeline gráfico, o que significa que o tipo das operações realizadas na mesma devem seguir o mesmo modelo utilizado em aplicações estritamente gráficas.

3.2 Origens

Na década de 80 cada fabricante tinha seu próprio hardware e conjunto de instruções para desenho gráfico 2D e 3D. Construir aplicações para tais tecnologias não padronizadas era uma tarefa árdua e que exigia um alto custo de desenvolvimento. As GPUs desse período tinham o custo de dezenas de milhares de dólares e eram utilizadas em aplicações altamente especializadas tais como visualização cientifica, CAD, simuladores aeronáuticos, visualizações de geologia de poços de petróleo e etc... Como exemplo do auto custo que tinham estes equipamentos, podemos citar um anúncio de 1994 disponível em [37] onde a IBM anuncia o lançamento de novos modelos de sua estação RISC/6000 equipadas com a a sua placa aceleradora 3D POWERgraphics GXT1000 que suportava OpenGl, Phigs e IBMGL 3.2 e tinha o custo, apenas a GPU, de $25.000,00.

A primeira tentativa de implantação de um padrão de programação foi realizada pela *Silicon Graphics Inc.* (SGI) através do PHIGS (sigla de *Programmer's Hierarchical, Graphics System*), que não obteve sucesso devido a sua complexidade. A segunda tentativa da SGI foi o IRIS-GL que teve boa aceitação por ter sido considerado de fácil utilização. Mas, a *Sun Microsystems* e outras grandes empresas continuavam a utilizar o PHIGS. A IRIS-GL possuía muitas licenças e patentes, dificultando o processo de torná-la um padrão da industria e no seu grupo de funções estavam misturadas ao desenho 2D e 3D funções de gerenciamento de janelas, teclado e mouse. Então, em 1992, a SGI liderou a criação do grupo de revisão arquitetural de um novo padrão chamado de OpenGL [36]. Este grupo era composto de empresas de deveriam manter e expandir a sua especificação. Fato que ocorre até os dias atuais [1].

[1] Atualmente mantido pelo *The Khronos Group* fundado em janeiro de 2000 por diversas companhias da área de mídia, incluindo 3Dlabs, ATI, Discreet, Evans & Sutherland, Intel, NVIDIA, SGI e Sun Microsystems, dedicado a criar padrões abertos de APIs para possibilitar autoria e exibição de mídia complexa em uma ampla variedade de plataformas e dispositivos.

Na versão 2.0 lançado em 7 de setembro de 2004 a OpenGL recebeu a adição da OpenGL Shading Language - GLSL, uma linguagem com sintaxe semelhante a C onde os estágios de transformação de vértices e renderização de fragmentos do pipeline (o detalhamento destes elementos será realizado mais a frente) passaram a poder serem programados [42]. Estava marcado o início da flexibilização e programação nas GPUs.

3.3 Vértices e Fragmentos

As GPUs são projetadas para renderizar rapidamente cenas sintéticas. No contexto da computação gráfica renderização é o nome dado ao processo de definição da cor de um pixel. Na computação gráfica 3D esta definição de cor é baseada em parâmetros do ambiente 3D virtual cuja imagem está sendo gerada. Parâmetros tais como: fontes de luz e sua cor, textura e reflexão do objeto a ser retratado em 3D e etc... Em resumo, uma diversidade de parâmetros e cálculos que variam em função da cena a ser retratada e do algoritmo de geração desta imagem.

Este processamento gráfico requer operações sobre grandes conjuntos de dados, mais especificamente vértices e fragmentos. Os vértices são os elementos fundamentais dos poliedros que compõem a cena. Fragmentos equivalem a pixels da imagem final porém, com profundidades variadas. Na renderização ou rasterização, as cores dos fragmentos são combinadas ou o fragmento mais próximo da câmera é escolhido para compor a imagem final.

3.4 Pipelines

O processamento que é realizado sobre os vértices e fragmentos é realizado em uma sequência de operações nos processadores de vértices, também chamados de processadores de geometria, e posteriormente nos processadores de fragmentos. As instruções são executadas no processador como em uma linha de montagem, conhecida na literatura como *pipeline* [43]. Na figura 3.1 vemos a organização e um detalhamento da estrutura de um *pipeline*. Nessa figura os blocos são as primitivas gráficas que estão implementadas em cada *shader*. Para maiores detalhes sobre a função desempenhada por cada bloco integrante de um *pipeline* consulte [36].

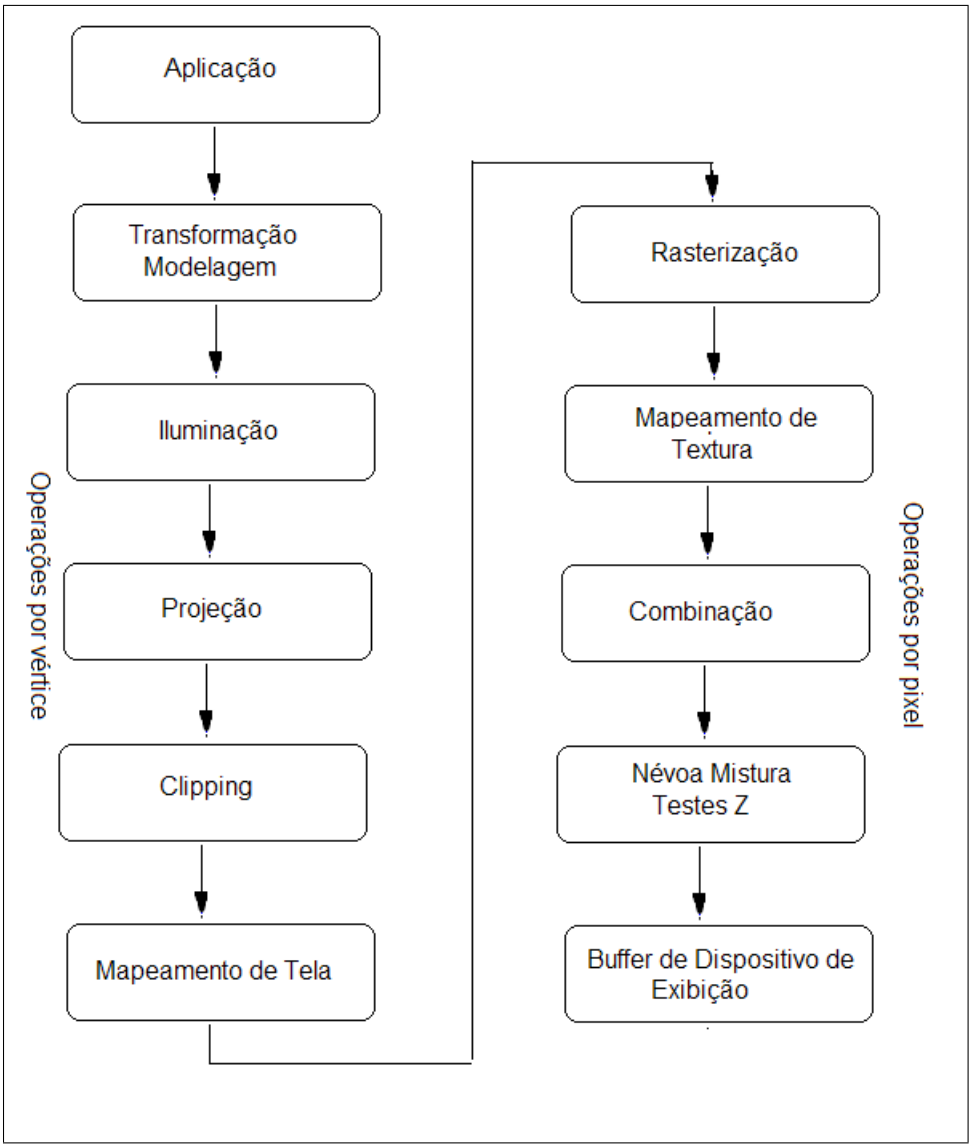

Figura 3.1: *Estrutura de um pipeline*

Nas GPUs atuais são conseguidas grandes acelerações pela implementação em paralelo dos pipelines em uma ordem de milhares. Quando a SGI lançou o release 2.0 da OpenGL a linguagem GLSL implementada tinha como objetivo tornarem programáveis alguns dos estágios dos processadores de vértices e fragmentos [40]. Na figura 3.2 vemos a organização e um detalhamento da estrutura de um *pipeline* OpenGL 2.0 com os processadores de vértice e fragmento programáveis também chamados de *shaders*. *Shader* é a nomenclatura utilizada para os procedimentos de sombreamento ou iluminação personalizado, que permitem aos programadores especificar o modo como um vértice ou pixel deve ser renderizado, oferecendo uma maior liberdade para que eles possam criar um visual e estilo único para seus aplicativos [38].

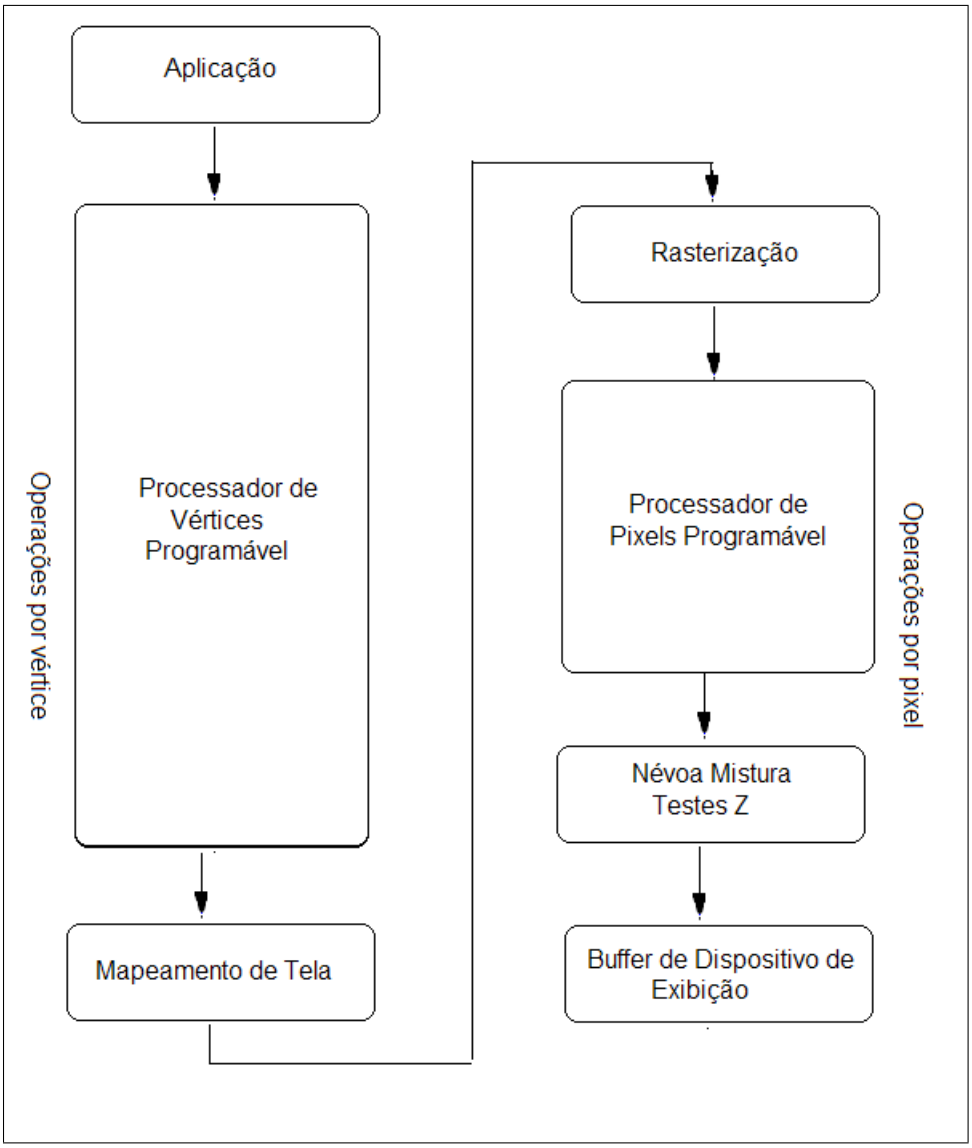

Figura 3.2: *Estruturas programáveis no pipeline*

Em 2005 a ATI e Microsoft apresentaram o conceito de *Unified Shading Architecture* materializado como produto no videogame XBOX 360 [34]. Nesta arquitetura os processadores de vértice e fragmentos são fundidos em um único processador com mais recursos e flexibilidade de programação. Em tese, essa seria uma evolução natural, pois, os processadores de vértice e fragmentos já possuíam precisão de 32 bits para ponto flutuante e compartilhavam das mesmas instruções de loop, decisão entre outras, então, passou a não ter mais sentido a sua separação física.

O poder computacional das GPUs atuais é devido a uma arquitetura altamente especializada, evoluída e ajustada por anos para extrair o máximo de performance das tarefas altamente paralelizáveis da computação gráfica tradicional [35]. O aumento da

flexibilidade das GPUs possibilitou a sua aplicação em tarefas fora das quais as GPU foram originalmente desenvolvidas. Mas, ainda existem muitas aplicações que não são adequadas e muitas em que nunca serão. Tarefas onde a leitura e escrita na memória é disperso e ostensivo são um exemplo desta inadequação. Essa limitação no acesso a memória se deve às limitações impostas aos circuitos de controle de acesso ao barramento de dados e à memória resultantes da liberação de espaço e transistores para implementação de pipelines paralelos. Além disso, só recentemente as GPUs passaram a ter precisão de 32 bits para ponto flutuante, sendo que, não existe previsão para a disponibilização da precisão dupla (64 bits). Dessa forma, levando a impossibilitar a sua utilização em muitas aplicações de computação cientifica.

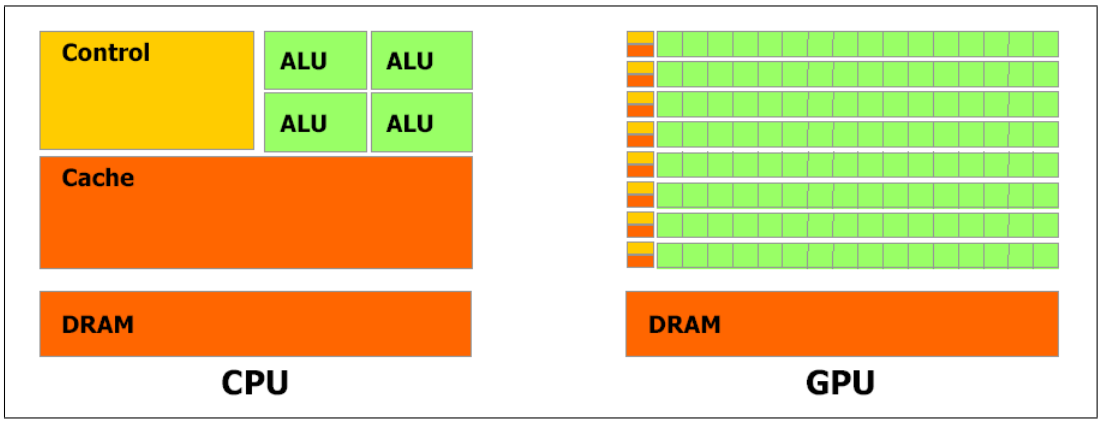

Figura 3.3: *As GPUs disponibilizam mais transistores para os múltiplos pipelines sacrificando os circuitos de controle do acessoa a memória*

3.5 Instruções condicionais

O uso de instruções condicionais (também conhecidas na computação como de ramificação ou desvio) é um dos fundamentos utilizados em programação de processadores considerado imprescindível no desenvolvimento de algoritmos. Tanto que, em arquiteturas em que as instruções ramificação e repetição são limitadas, a implementação de algoritmos se torna altamente complexa. As GPUs atuais suportam as instruções ramificação mas possuem limitações em que o desenvolvedor deve ter atenção para obter os resultados esperados.

Nas GPUs atuais que se baseiam no conceito de unificação do processadores de vértice e fragmentos a implementação paralela das instruções ramificação segue a arquitetura *Single Instruction Multiple Data* (SIMD) onde todos os processadores executam as

mesmas instruções em uma massa de dados variável.

Em uma CPU quando o ocorre uma instrução de desvio dentro de um pipeline a CPU o esvazia e começa a executar as instruções do destino do desvio. Este esvaziamento do pipeline penaliza o desempenho adicionado pelo pipeline na execução das instruções [43].

Dentro de um grupo de instruções SIMD, quando da avaliação de um desvio é idêntico, o único código executado é o do desvio feito por todos os processadores. Mas, se um ou mais processadores executa um desvio diferente, então, os dois desvios devem ser executados por todos os processadores e qual desvio foi executado é informado por uma variável que sinaliza qual o desvio que é válido. Então, quando os códigos executados pelos processadores divergem leva a uma penalização global de desempenho por causa da execução das duas possibilidades de desvio [35].

3.6 Modelo de programação em *streams* (fluxos)

No contexto da programação em GPUs este modelo específica uma linguagem onde diversas operações em paralelo ou assíncronas possam ser executadas em um fluxo de dados. Como exemplo, podemos citar uma rotina de incremento de cada item de um vetor de n posições. Na linguagem C padrão criaríamos um loop onde, passo a passo, incrementaríamos cada posição do vetor. No modelo *streams* poderíamos especificar a função que incrementaria um item do vetor e executaríamos n instâncias desta função em paralelo uma para cada item n do vetor.

3.7 Linguagens e bibliotecas de Programação

Várias linguagens e bibliotecas foram desenvolvidas para programação dos processadores de vértice e de fragmento. Além da já citada *OpenGL Shading Language*s (GLSL) surgiram outras tais como: *C for Graphics* (Cg) e *High Level Shading Language* (HLSL). Com a unificação dos processadores de vértices e de fragmentos, surgiram as linguagem que se propõem a ser ferramentas para desenvolvimento de aplicativos de propósitos gerais. Das quais, citamos como exemplo, as linguagens Brook, Sh, Nvidia Cuda e OpenCL (*Open Computing Language*).

A linguagem Brook é um conjunto de extenções ao ANSI C desenvolvida pela Universidade de Stanford. Seu objetivo é oferecer o modelo de programação em *streams* para GPUs. A Brook pode ser executada em GPUs que suportem OpenGL ou DirectX da Microsoft utilizando-as como interface de acesso ao hardware da GPU. Atualmente a linguagem Brook é vendida comercialmente.

A Sh consiste em uma biblioteca que funciona com a linguagem C++ sobre o suporte da OpenGL. Ela é o resultado de pesquisas do laboratório de computação gráfica da Universidade de Waterloo.

Cuda é o nome dado pela NVidia para sua linguagem de programação em GPUs. Ela consiste em bibliotecas e ferramentas em C e C++, incluindo compilador, para desenvolver aplicações paralelas que executam exclusivamente em placas de vídeo de sua fabricação.

A OpenCl se propõe a ser um conjunto de ferramentas, livre de patentes, para o desenvolvimento de aplicações que explorem paralelismo, que executem em plataformas heterogêneas tais como como CPUs, GPUs, e outros processadores [41]. Baseada na linguagem C99 ela foi proposta pela Apple e atualmente está sendo desenvolvida pelo mesmo grupo que mantém o padrão OpenGL, o *Khronos Group*. A Nvidia e a ATI já disponibilizam, em seus sites, kits de desenvolvimento para a OpenCL o que parece indicar uma tendência a ser adotada como o futuro padrão das ferramentas de desenvolvimento em GPUs.

3.8 Programação Cuda

Em novembro de 2006 a Nvidia apresentou uma nova arquitetura computacional para as suas GPUs e um novo modelo de programação paralela e conjunto de instruções chamado Cuda. A partir da série 8000 de suas GPUs seria disponível, para os desenvolvedores um ambiente de programação baseado na linguagem C [39].

Na figura 3.4 vemos uma comparação da velocidade das GPUs Nvidia em comparação com as CPUs Intel. Podemos observar que os ganhos em velocidade das GPUs Nvidia tem sido muito maior que as CPUs intel. Este aumento se justifica pelas limitações no aumento da velocidade no relógio das CPUs, enquanto, as GPUs baseiam seu aumento de velocidade, prioritariamente, na implementação de mais processadores pa-

ralelizáveis. Atualmente as novas gerações de CPUs buscam a implementação de mais núcleos de execução para contornar as limitações de aceleração de relógio de instruções.

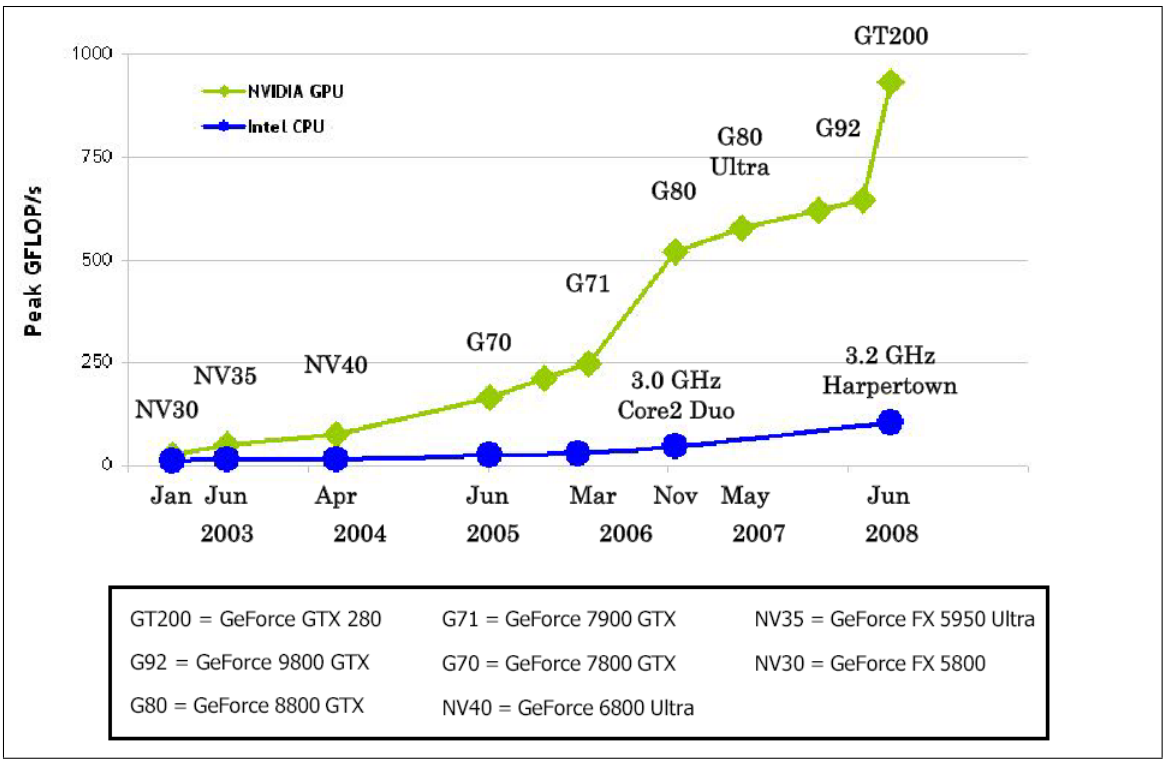

Figura 3.4: *Comparação de performance entre Nvidia GPUs e Intel CPUs em operações de ponto flutuante por segundo [39]*

3.8.1 Estrutura da programação em Cuda

A Cuda oferece ao programador uma definição de função chamada *kernel*. Este tipo de função pode ser executada N vezes em paralelo por N diferentes *threads*, de forma oposta as funções em C regulares, onde uma função é executada apenas uma vez. Um *kernel* é definido utilizando uma declaração `__global__` e o número de *threads* a cada chamada é definido por uma nova sintaxe de chamada de funções: $<<<\ldots>>>$.

```
// Definição da função kernel
__global__ void VecAdd(float* A, float* B, float* C)
{

}

int main()
{
    //  Chamada de uma função kernel
```

```
    VecAdd<<<1, N>>>(A, B, C);
}
```

Cada uma das *threads* que executa um *kernel* é atribuída um número de identificação único que é acessível dentro da função *kernel* através de uma variável chamada *threadIdx*. Como exemplo é mostrado o código abaixo onde são somados dois vetores A e B de tamanho N e o resultado é guardado no vetor C de N dimensões.

```
// Definição da função kernel
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
int main()
{
    // Chamada
    VecAdd<<<1, N>>>(A, B, C);
}
```

3.8.2 Organização das *threads*

Por conveniência, o *threadIdx* é um vetor de 3 componentes, onde cada *thread* pode ser identificada utilizando uma estrutura de uma, duas ou três dimensões. Isso fornece uma forma natural de computar utilizando estes elementos de índice em elementos de domínio de vetores, matrizes e campos. Como exemplo, o código abaixo soma duas matrizes A e B de tamanho $N \times N$ e guarda seu resultado na matriz C $N \times N$.

```
// Definição da função kernel
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    // Chamada
    dim3 dimBlock(N, N);
    MatAdd<<<1, dimBlock>>>(A, B, C);
}
```

As funções na Cuda podem qualificadas como `__global__` que são funções que podem ser executadas na GPU e CPU ou `__device__` que são funções executadas apenas na GPU.

A organização dos *threads* no hardware da GPU é feita em um hierarquia de grids e blocos. Cada grid e bloco pode ser especificado com uma, duas ou três dimensões. No código abaixo é exemplificada a chamada de uma rotina a partir de uma matriz de grids bidimensional de 10×64 onde cada grid é composto por uma matriz de blocos tridimensionais de $256 \times 256 \times 64$.

```
int gridXsize=10;
int gridYsize=64;
int blkXsize=256;
int blkYsize=256;
int blkZsize=64;


dim3 dimGrd(gridXsize,gridYsize);
dim3 dimBlk(blkXsize,blkYsize,blkZsize);

Rotina_GPU_Kernel<<<dimGrd, dimBlk>>>();
```

Neste exemplo anterior a forma do programador acessar o identificador do *thread* atual é pelas variáveis blockIdx e threadIdx. Onde blockIdx.x e blockIdx.y identificam qual bloco do grid 10×64 é o atual e as variáveis threadIdx.x, threadIdx.y e threadIdx.z identificam qual é a *thread* ativa no bloco atual.

Outro termo criado pela Nvidia é o *warp*. Um *warp* é um grupo de 32 *threads* que também é o tamanho mínimo dos dados que podem ser processados no modo SIMD do multiprocessador. A definição de *warp* é utilizada no capitulo de otimização da velocidade do acesso a memória que será citada mais a frente.

Na figura 3.5 se observa um diagrama da Nvidia de como os grids, blocos e *threads* são organizados.

3.8.3 Utilização da memória do dispositivo

Kernels podem executar apenas nas memórias da GPU, então, são disponibilizadas funções para alocar, desalocar e copiar nas memórias de dispositivo bem como transferir dados entre CPU e dispositivo. As memórias de dispositivo podem ser alocadas em blocos de

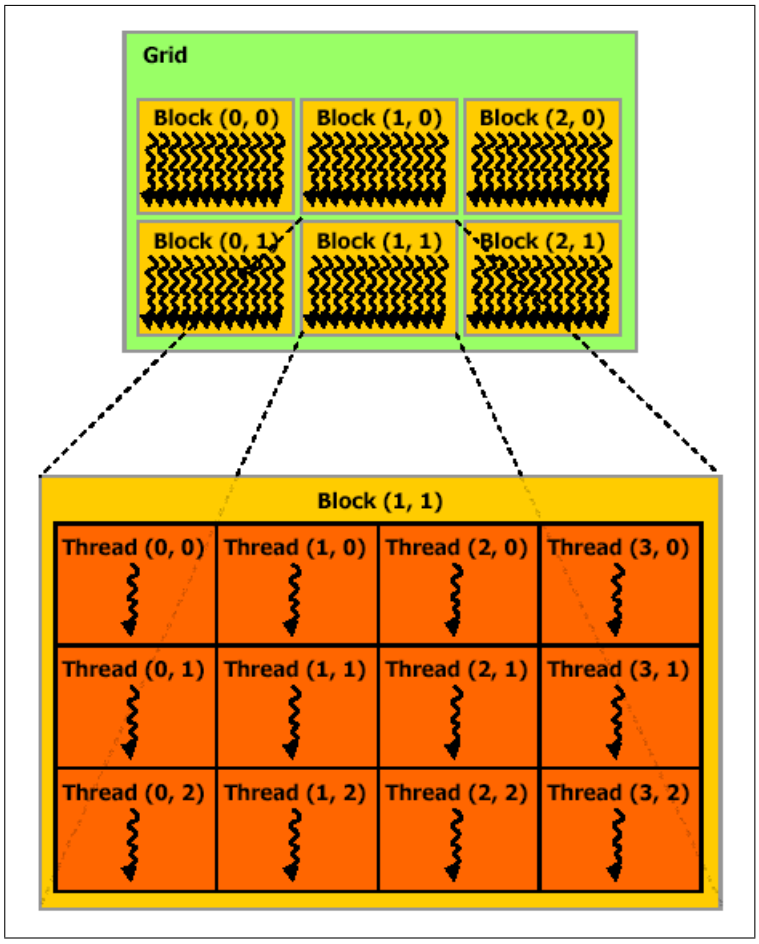

Figura 3.5: *Organização de Grid, Blocos e Threads*

memória lineares ou vetores Cuda. Vetores Cuda são estruturas de memória otimizadas para o uso de texturas.

Blocos de memória lineares podem se auto referenciar com ponteiros [39]. Tipicamente, um bloco de memória é alocado com a função `cudaMalloc()` e liberado com a função `cudaFree()` e a transferência de dados entre a memoria de dispositivo e a memória de CPU é feito com a função `cudaMemcpy()` [39].

Abaixo é mostrado o exemplo de soma de um vetor onde os dados são copiados entre a memória da GPU e CPU:

```
//' Código da função kernel
__global__ void VecAdd(float* A, float* B, float* C)
{
     int i = threadIdx.x;
     if (i < N)
          C[i] = A[i] + B[i];
}
// Código executado na CPU
int main()
```

```
{
    // Aloca os vetores na memória de dispositivo
    size_t size = N * sizeof(float);
    float* d_A;
    cudaMalloc((void**)&d_A, size);
    float* d_B;
    cudaMalloc((void**)&d_B, size);
    float* d_C;
    cudaMalloc((void**)&d_C, size);
    // Copia vetores da memória da CPU para a memória de GPU
    // h_A e h_B são vetores de entrada guardados na memória da CPU
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    // Chamada do  kernel
    int threadsPerBlock = 256;
    int threadsPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<threadsPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);
    // Copia resultados da memória do dispositivo para a memória da CPU
    // h_C contêm o resultado na memória da CPU
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    // Libera memória da CPU
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

As memórias do tipo *shared* são alocada com o qualificador de variável `__shared__` e esta variável é vista, diretamente, apenas no escopo das funções que executam na GPU e podem ser acessadas da CPU com a função `cudaMemcpyToSymbol()`;

As memórias do tipo *constant* são alocadas com o qualificador de variável `__constant__` e esta variável é vista, diretamente, apenas no escopo das funções que executam na GPU e podem ser acessadas da CPU com a função `cudaMemcpyToSymbol()`;

3.8.4 *Compute Capabilities*

A Nvidia distingue a capacidade de cada GPU no ambiente Cuda, pelas capacidades de cada hardware, com uma classificação chamada *Compute Capability*. Esta classificação deve ser bem conhecida pelo desenvolvedor, pois, nela temos informações de limitações para cada GPU no desenvolvimento de aplicações. Como exemplo destas limitações, podemos citar o número de máximo de *threads* ativas suportado pelo equipamento; em uma GPU com *Compute Capability 1.0* que é de 768 por multiprocessador.

Na tabela 3.1 estão listadas a *Compute Capabilities* de algumas GPUs. Avaliando

a placa GeForce 8600GT (que foi utilizada nos experimentos desta tese) vemos que ela tem 4 multiprocessadores e *Compute Capability* 1.1, então, podemos afirmar que ela suporta $4 \times 768 = 3072$ *threads* ativas por chamada de uma rotina *kernel*.

	Number of Multiprocessors (1 Multiprocessor = 8 Processors)	Compute Capability
GeForce GTX 295	2x30	1.3
GeForce GTX 285, GTX 280	30	1.3
GeForce GTX 260	24	1.3
GeForce 9800 GX2	2x16	1.1
GeForce GTS 250, GTS 150, 9800 GTX, 9800 GTX+, 8800 GTS 512	16	1.1
GeForce 8800 Ultra, 8800 GTX	16	1.0
GeForce 9800 GT, 8800 GT, 9800M GTX	14	1.1
GeForce GT 130, 9600 GSO, 8800 GS, 8800M GTX, 9800M GT	12	1.1
GeForce 8800 GTS	12	1.0
GeForce 9600 GT, 8800M GTS, 9800M GTS	8	1.1
GeForce 9700M GT	6	1.1
GeForce GT 120, 9500 GT, 8600 GTS, 8600 GT, 9700M GT, 9650M GS, 9600M GT, 9600M GS, 9500M GS, 8700M GT, 8600M GT, 8600M GS	4	1.1
GeForce G100, 8500 GT, 8400 GS, 8400M GT, 9500M G, 9300M G, 8400M GS, 9400 mGPU,	2	1.1

Tabela 3.1: *Compute Capabilities de algumas GPUs Nvidia*

Então, ao chamar uma rotina *kernel* devemos levar em conta que no máximo 3072 *threads* serão geradas. Um fato que foi observado no desenvolvimento desta tese é que até a versão 2.3 da Cuda SDK uma chamada de *kernel* que ultrapassasse este limite não teria nenhum aviso do compilador ou erro de execução para alertar o desenvolvedor deste fato. Como resultado, alguns *threads* simplesmente não são executados. Já no modo emulador da GPU do SDK os *threads* são executados podendo levar ao programador inexperiente a uma confusão do que está errado no algoritmo em desenvolvimento.

Nas tabelas 3.2, 3.3, 3.4 e 3.5 temos uma listagem detalhada da capacidade ofertada por cada *Compute Capability*.

Especificações da *Compute Capability* 1.0
Número máximo de *threads* por bloco é 512;
Valor máximo das dimensões x, y, e z de um bloco de *threads* é 512,
512, e 64, respectivamente;
O Valor máximo de cada dimensão de um grid de blocos de *threads* é 65535
O tamanho de um *warp* é de 32 *threads*;
O número de registradores por multiporcessador é 8192;
A quantidade de *shared memory* por multiprocessador é de 16KB organizado
em 16 bancos;
A quantidade total de *constant memory* é de 64 KB;
O cache para memórias *constant memory* é de 8 KB por multiprocessador;
O cache para memórias de textura varia entre 6 e 8 KB por multiprocessador;
O número máximo de blocos ativos por multiprocessador é 8;
O número máximo de *warps* ativos por multiprocessador é 24;
O número máximo de *threads* ativos por multiprocessador é 768;
Para uma referencia de textura unidimensional para um vetor
Cuda o maior tamanho possível é 2^{13};
Para uma referencia de textura unidimensional para uma memória linear
o maior tamanho possível é 2^{27};
Para uma textura bidimensional referenciada para uma memória linear
ou um vetor Cuda o valor máximo de largura é de 2^{16} e o valor máximo de
altura é de 2^{15};
Para uma textura tridimensional referenciada para um vetor Cuda
o valor máximo de largura é de 2^{11}, o valor máximo de altura é de 2^{11}
e o valor máximo de profundidade é de 2^{11};
O número máximo de instruções de um *kernel* é de 2 milhões de instruções
PTX (*Parallel Thread Execution*) ;
Cada multiprocessador é composto de 8 processadores, então, um
multiprocessador é capaz de processar os 32 *threads* de um *warp*
em 4 ciclos de clock.

Tabela 3.2: *Capacidade ofertada pela Compute Capability 1.0.*

Especificações da *Compute Capability* 1.1
Suporta funções atômicas em palavras de 32 bits na memória global

Tabela 3.3: *Capacidade ofertada pela Compute Capability 1.1.*

Especificações da *Compute Capability* 1.2
Suporta funções atômicas operando em memória *shared* e em palavras de 64 bits na memória global.
Suporta funções de verificação se ocorreu determinado desvio condicional nos *threads* de um *warp*.
Número de registradores por multiprocessador é de 16384;
O número máximo de *warps* ativos por multiprocessador é 32;
O número máximo de *threads* ativos por multiprocessador é 1024;

Tabela 3.4: *Capacidade ofertada pela Compute Capability 1.2.*

3.8.5 Hierarquia de memória

Um grupo de *threads* pode acessar dados de múltiplos espaços de memória durante a sua execução como podemos observar na figura 3.6.

Cada *thread* possui seu próprio espaço privado de memória local e cada bloco de *threads* possui uma faixa de endereços de memória compartilhada (chamada *Shared Memory*) visível por todas as *threads* deste bloco. Esta faixa de endereços de memória tem o mesmo tempo de vida do bloco da qual ela é derivada. Além disso, todos os *threads* de qualquer bloco pertencente a qualquer grid pode acessar uma mesma memória global chamada de *Global Memory*.

Além dos espaços de memória já citados, existem ainda dois tipos de memória de apenas leitura disponíveis e que podem ser lidas globalmente por todos os *threads* em execução; os tipos constante e textura.

Na figura 3.7 observamos um diagrama onde todos os tipos de memória disponíveis e o sentido de seus fluxos de dados são demonstrados.

As memórias global, constante e textura se mantem inalteradas por entre várias chamadas de diferentes *kernels* para a mesma aplicação. Em geral um acesso à memória global é 150 vezes mais lento no acesso que um registrador ou memória *shared* quando respeitadas as regras de acesso agrupado (*coalesced* este tema será esclarecido mais adi-

Especificações da *Compute Capability* 1.3
Supporte a números de ponto flutuante de precisão dupla.

Tabela 3.5: *Capacidade ofertada pela Compute Capability 1.3.*

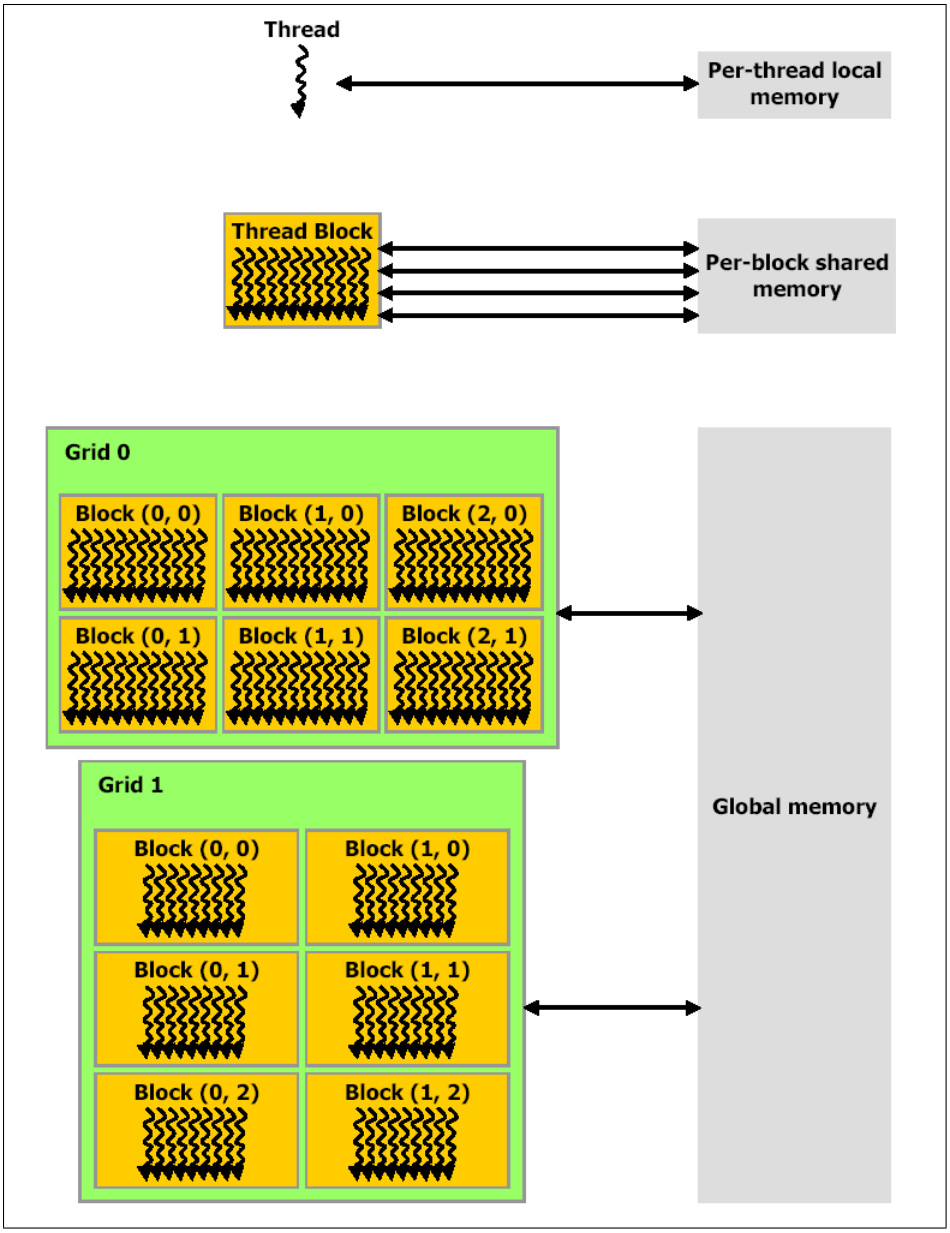

Figura 3.6: *Arquitetura de memória das GPUs NVidia*

ante).

3.8.6 Memória de textura - *Texture Memory*

As memórias de textura são oriundas de uma parte do hardware que a GPU utiliza para gráficos. O acesso a dados oriundos da memória de textura não são sujeitos aos problemas

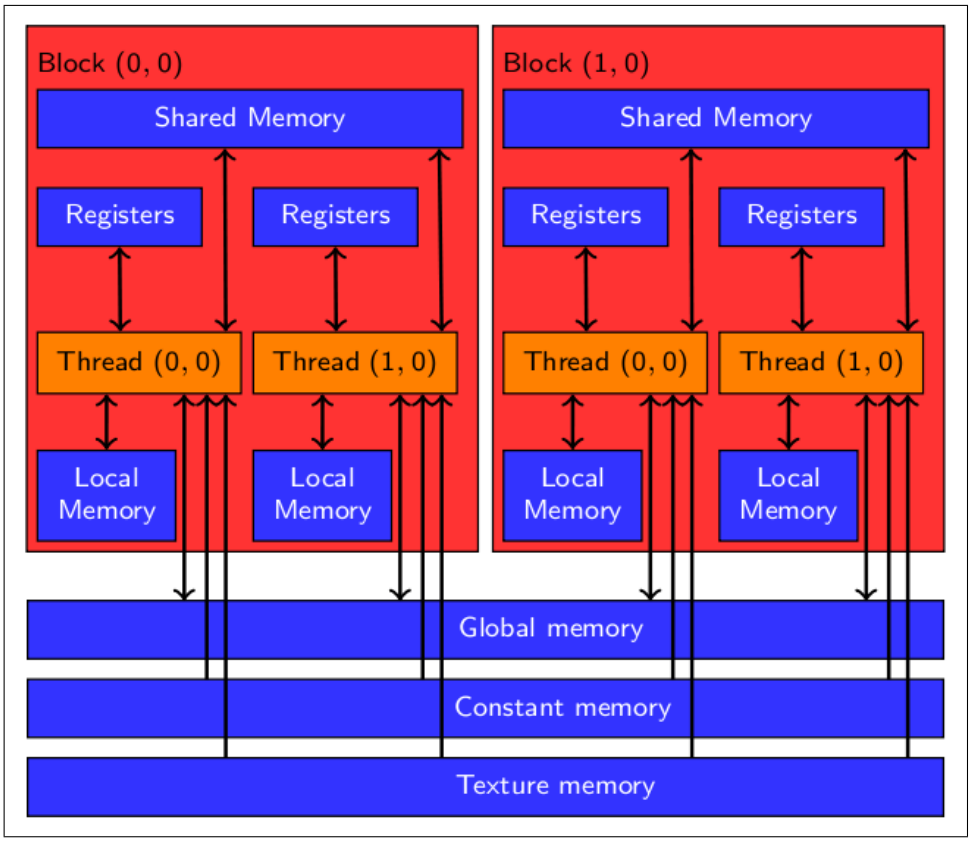

Figura 3.7: *Arquitetura de memória das GPUs NVidia*

que impactam na velocidade dos acessos à memória global.

O acesso a estas memórias por parte das rotinas de *kernel* é realizado, de uma forma indireta,pelo o uso de funções chamadas *texture fetches*.

Os dados que descrevem estas texturas estão organizados em vetores que podem ser de uma, duas ou três dimensões, compostos de elementos onde cada um destes pode ter 1, 2 ou 4 componentes que podem ser números inteiros com ou sem sinal de 8, 16 ou 32 bits ou números de ponto flutuante de 16 ou 32 bits. Esta organização é devido à utilização, originalmente, desta memória para o armazenamento de dados referentes a mapas de bits de imagem que seriam utilizados nas aplicações gráficas para descrever o material da superfície de um determinado objeto.

3.8.7 Declaração das memórias de textura

A declaração de variáveis de textura é semelhante à declaração de *templates* C++ e elas devem ser declaradas no escopo do arquivo onde serão utilizadas, mas, de forma diferente dos *templates*, alguns do atributos devem ser imutáveis e conhecidos durante a compilação.

Uma referência a uma textura tem a seguinte sintaxe:

```
texture<Tipo, Dimensoes, FormaLeitura> texRef;
```

- `Tipo` especifica o tipo do dado que é retornado quando recuperamos esta textura, sendo que, ele deve ser restrito aos tipos inteiros e de ponto flutuante de precisão simples na forma de vetor com já esclarecido na seção anterior
- `Dimensoes` é um valor opcional que especifica o número de dimensões assumindo os valores 1, 2 ou 3
- `FormaLeitura` tem como opções `cudaReadModeNormalizedFloat` para normalizar entre [0.0, 1.0] os números inteiros sem sinal e [-1.0, 1.0] para os sinalizados. Quando utilizamos a opção `cudaReadModeElementType` nenhuma conversão é realizada

3.8.8 Transferência de dados entre CPU e GPU

A velocidade de comunicação entre as memórias da GPU para GPU é muito maior que a velocidade entre a memória da GPU e a memória da CPU hospedeira. Então, uma aplicação que busca otimizar seu desempenho deve ter como premissa a minimização da transferência de dados entre as memórias da GPU e CPU. Além disso, a aglutinação de pequenas transferências de dados em uma grande transferência oferece uma grande melhoria no desempenho.

Podemos ainda, obter altas performances nas tranferências de dados entre CPU e GPU utilizando as memórias em uma configuração *page-locked* na CPU. Além da banda de transferência ser maior, podemos realizar cópias de foma concorrente com a execução da CPU e GPU e em alguns dispositivos estas memórias podem ser mapeadas no espaço de endereçamento da CPU, eliminando a cópia de e para a memória da CPU e GPU.

3.8.9 Execução e copias assíncronas

Com o objetivo de facilitar a execução concorrente entre CPU e GPU algumas funções são assíncronas. Estas funções, quando executadas, não param a execução da CPU ou GPU, sendo executadas de forma concorrente com o código da função que a chamou retornando o controle para a CPU.

- Chamadas de *kernel*

- Funções com o sufixo `Async`
- Funções que fazem copias de memória de GPU para GPU
- Funções que setam valores em memória

Além disso, segundo [39], alguns dispositivos também executam cópias entre memórias *page-locked* e memórias de dispositivo de forma concorrente com a execução de um *kernel*. Uma aplicação pode verificar se um dispositivo tem esta capacidade chamando a função `cudaGetDeviceProperties()` verificando se a propriedade `deviceOverlap` é verdadeira. Atualmente, Esta capacidade é somente suportada para copias de memória que não utilizem dados Cuda.

3.8.10 Otimização do acesso à memória

Qualquer endereço de uma variável residente na memória global ou retornada por uma das rotinas de alocação da API - *Aplication Program Interface* é sempre alinhada em múltiplos de 256 bytes de memória.

Além disso, a velocidade da banda de memória é utilizada de forma mais eficiente quando os acessos a memória feitos pelos *threads* em um meio *warp* (grupo de 16 *threads*), durante a execução de uma instrução de leitura ou escrita em memória, podem ser aglutinadas (na nomenclatura da Nvídia *coalesced*), em uma transação de memória de 32, 64 ou 128 bytes .

Dependendo da *Compute Capability* existem formas diferentes de organizar a aglutinação (*coalescing*) do acesso à memória. Se meio *warp* atende a estes requerimentos de organização, a otimização é atingida mesmo se ocorre divergência de execução em alguns *threads* mesmo que eles não façam acesso a memória [39].

Para conseguirmos sucesso na aglutinação a premissa básica é a memória estar particionada em segmentos de mesmo tamanho de 32, 64 ou 128 bytes e os ponteiros estarem alinhados a esta blocagem.

3.8.11 Aglutinação (*coalescing*) do acesso à memória para dispositivos com a *Compute Capability* 1.0 e 1.1

Para os dispositivos que atendam a *Compute Capability* 1.0 e 1.1 os acessos a memória global para todos os *threads* de um meio *warp* devem ser aglutinados em uma ou duas transações se satisfazem a três condições:

- As *Threads* devem acessar:

  Ambas as palavras de memória de 32-bits, resultando em uma transação de memória de de 64 bytes;

  Ou palavras de memória de 64-bits, resultando em uma transação de memória de de 128 bytes;

  Ou palavras de memória de 128-bits, resultando em duas transações de memória de de 128 bytes;

- Todas as 16 palavras de memória (32 bits) devem estar no mesmo segmento de mesmo tamanho da transação de memória ou duas vezes o tamanho quando acessando palavras de memória de 128 bits;

- As *Threads* devem acessar as palavras de memória em uma sequencia onde a enésima *thread* deve acessar a enésima palavra de memória

Se o meio *warp* não atende aos requisitos acima, uma transação de memória por *thread* é executada e a taxa de transmissão de dados com a memória é significantemente reduzida. Na figura 3.8 são mostrados dois exemplos de acesso a memória que são agrupados (*coalesced*) e as figuras 3.9 e 3.10 são exemplos de acesso à memória não agrupados para as *Compute Capability* 1.0 e 1.1.

Acessos de memória agrupados de 64 bits possuem uma pouco menor velocidade de comunicação do que acessos de memória de 32 bits e acessos de 128 bits possuem uma notável menor velocidade de comunicação do que acessos de memória de 32 bits. Mas, enquanto a velocidade de comunicação para acessos não agrupados é em torno de uma ordem magnitude menor que acesses agrupados quando estes acessos são em blocos de 32 bits, é somente em torno de 4 vezes menor quando eles são de 64 bits e em torno de 2 vezes quando eles são de 128 bits.

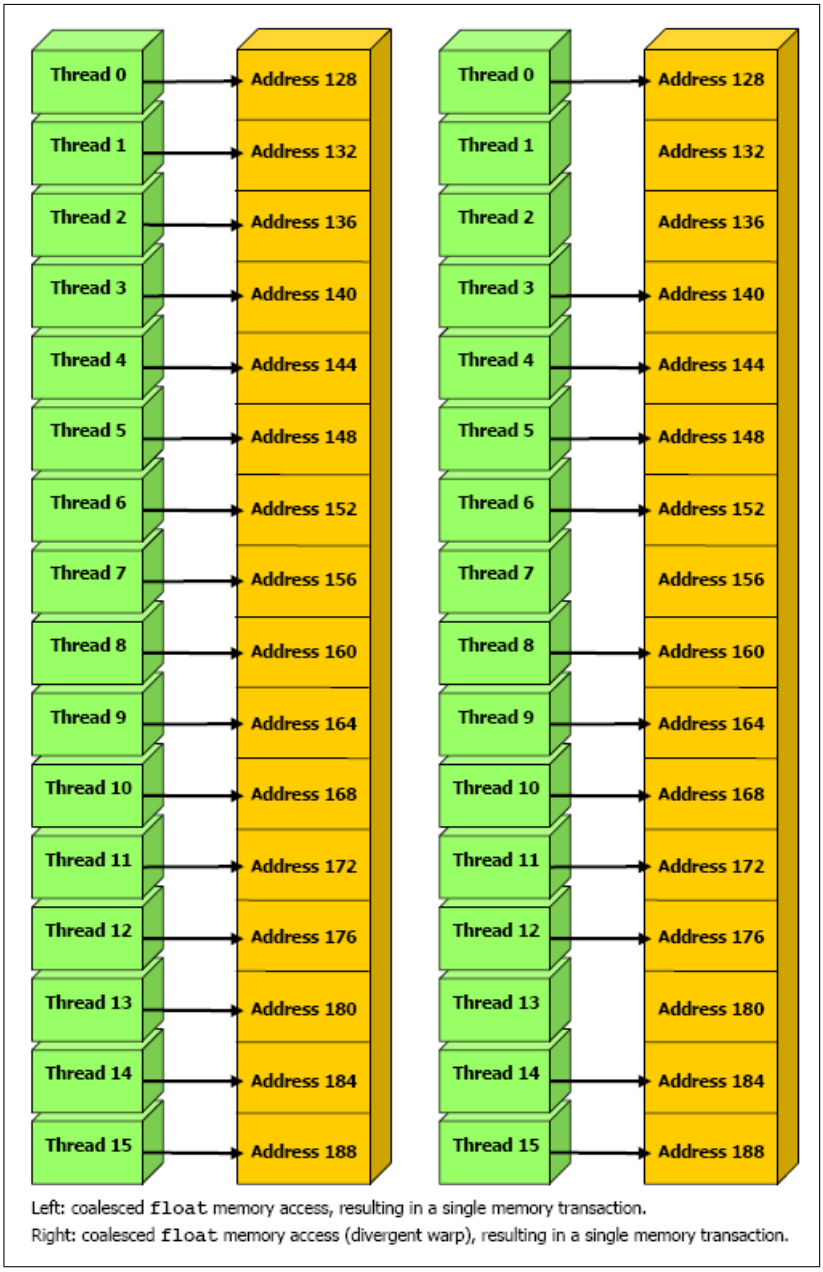

Figura 3.8: *Exemplos de acesso à memória que são agrupados (coalesced)*

3.8.12 Aglutinação (*coalescing*) do acesso à memória para dispositivos com a *Compute Capability* 1.2 ou acima

Para os dispositivos que atendam a *Compute Capability* 1.2 ou maior os acessos a memória global por todos os *threads* de um meio *warp* são aglutinados em apenas uma transação de memória desde que as palavras de memória acessadas por todas as *threads* estejam no mesmo segmento onde seu tamanho seja igual a:

- 32 bytes se todos os *threads* acessam palavras de memória de 8 bits

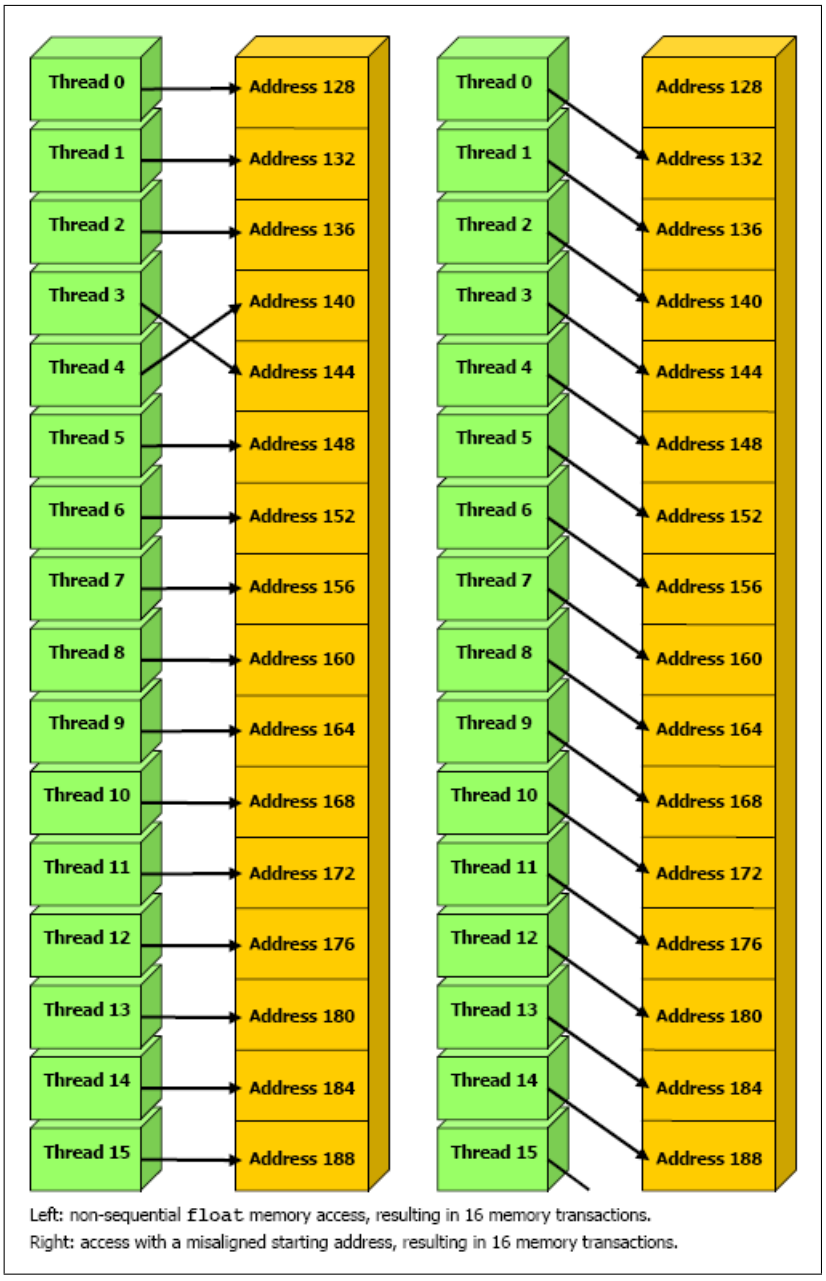

Figura 3.9: *Exemplos de acesso à memória que não são agrupados (coalesced) para as compute capabilities 1.0 e 1.1*

- 64 bytes se todos os *threads* acessam palavras de memória de 16 bits
- 128 bytes se todos os *threads* acessam palavras de memória de 32 bits ou 64 bits

O agrupamento é conseguido para qualquer padrão de endereços solicitados pelo meio *warp* incluindo padrões onde múltiplas *threads* acessam o mesmo endereço. Isso se diferencia dos dispositivos de menores *compute capabilities* onde as *threads* devem acessar os endereços de memória em uma sequência. Na figura 3.11 são exibidos exemplos desta

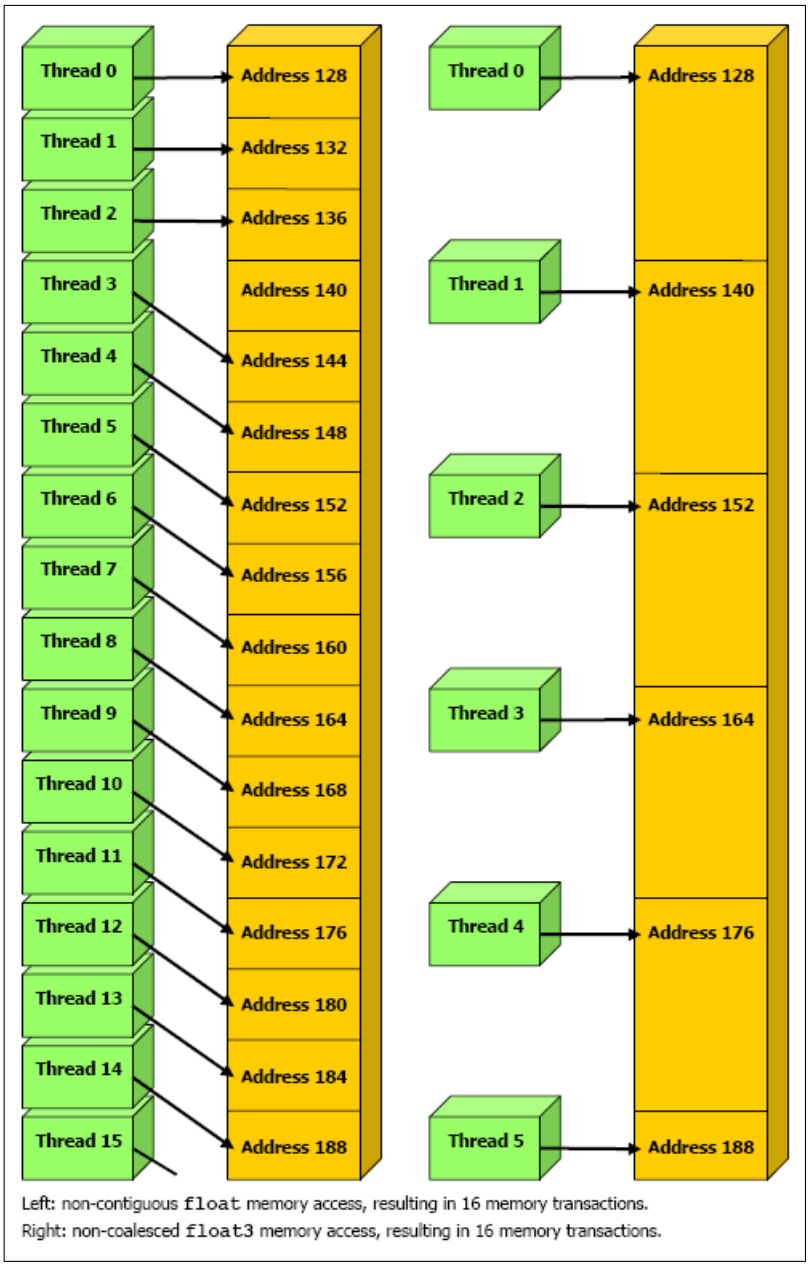

Figura 3.10: *Outros exemplos de acesso à memória que não são agrupados (coalesced) para as compute capabilities 1.0 e 1.1*

nova flexibilidade de endereçamentos.

3.8.13 Planilha auxiliar *Cuda Occupancy Calculator*

A planilha *Cuda Occupancy Calculator* possibilita ao desenvolvedor calcular a ocupação dos multiprocessadores de uma GPU por um *kernel* em execução. Essa ocupação é a taxa de *warps* ativos em relação ao número máximo de *warps* suportados pelo multiprocessador da GPU [19].

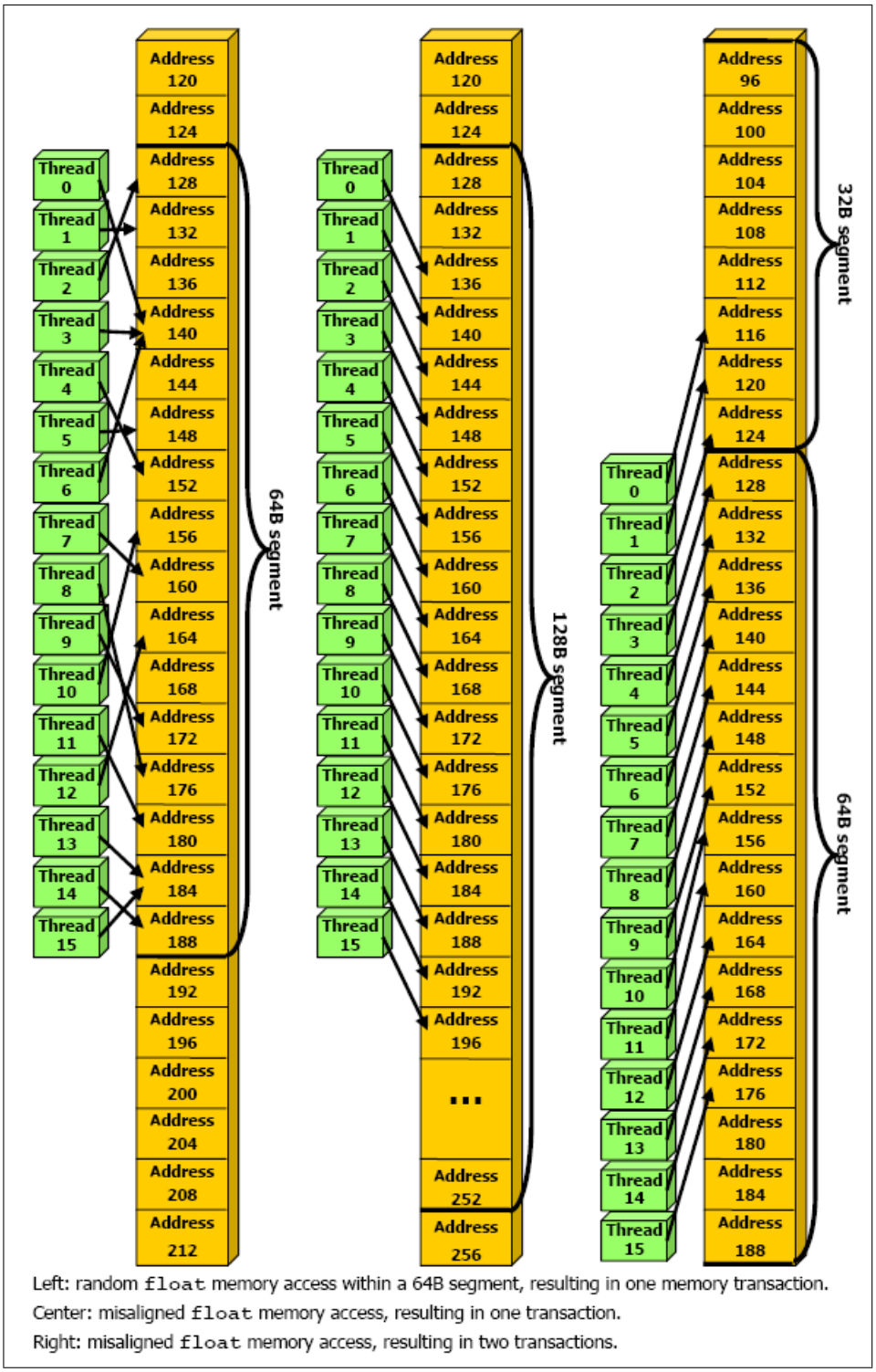

Figura 3.11: *Exemplos de acesso à memória que são agrupados (coalesced) paras as GPU com compute capability 1.2 ou acima*

Cada multiprocessador no dispositivo tem um grupo de N registradores disponíveis para utilização pelos programas. Estes registradores são um recurso compartilhado que é alocado pelos blocos de *threads* que estão em execução no multiprocessador. O compilador

Cuda busca minimizar o uso de registradores para maximizar número de blocos de *threads* ativos simultaneamente. Se um programa tenta lançar um *kernel* para o qual o número de registradores por *thread* vezes o tamanho do bloco de *threads* seja maior que N o lançamento irá falhar.

O número N nas GPUs com *compute capability* 1.0 e 1.1 tem o valor de 8192 registradores de 32 bits por multiprocessador. Nas GPUs com *compute capability* 1.2 e 1.3 o valor de N é 16384.

Para determinar o número de registradores que estão sendo utilizados por um *thread*, devemos compilar o aplicativo passando as opções `-ptxas-options=-v` para o `nvcc`. Com isso o executável irá imprimir a informação de uso de registradores, memória local, memória *shared* e memória constante para cada *kernel* no código fonte. De forma alternativa podemos utilizar a opção `-cubin` na compilação com o `nvcc` está opção irá gerar um arquivo com a extensão `.cubin` onde estarão estas informações em conjunto com seu código fonte.

Capítulo 4

Paralelismo na CPU

4.1 CPU Múltiplos Núcleos ou *Cluster*

No escopo da programação em CPUs existem as opções de programação em múltiplos *threads* em ambientes de CPUs mono processadas, em CPUs com vários núcleos de processador ou até em supercomputadores com milhares de CPUs interligadas por barramentos proprietários ou redes de fibra ótica.

São disponíveis, diversas ferramentas de programação em múltiplos *threads* nessa variedade de ambientes. Em soluções de um computador o padrão proposto foi a biblioteca *Pthreads*. Em 1995, o padrão POSIX (*Portable Operating System Interface*) P1003.1c foi aprovado. Com este padrão, também conhecido como *Pthreads*, aplicações comerciais *multithread* passaram a contar com um padrão de desenvolvimento.

Uma vez que a rede de computadores pode ser considerada como uma excelente opção para a execução dos problemas de processamento massivamente paralelos buscou-se implementar e viabilizar, através de um protocolo baseado na troca de mensagens, o uso de ferramentas que utilizassem os diversos processadores existentes em um rede bem suas respectivas memórias locais. Assim, surgiram, como alternativas a esta forma de execução de aplicações, os ambientes MPI e PVM.

O PVM [9] é mais antigo que o MPI, tendo surgido em 1989 nos laboratórios da Emory University e Oak Ridge National Laboratory, onde nasceu com o objetivo de criar e executar aplicações paralelas em um hardware já existente. Desde a versão PVM 3.4 Beta 6 é oferecida como principal característica a interoperabilidade entre máquinas UNIX e máquinas *Windows*.

O PVM teve grande difusão e foi aceito muito rapidamente contando com milhares de usuários. Assim, tornou-se um padrão de fato devido a sua flexibilidade pois habilita uma coleção de computadores heterogêneos a comportarem-se como se fosse um único recurso computacional expansível e concorrente. Grandes problemas computacionais podem ser resolvidos através da agregação e compartilhamento de processadores e memórias de outros computadores com um custo efetivo menor.

Suporta diversas arquiteturas e redes de trabalho e oferece a capacidade de utilização efetiva de computação paralela com paralelização escalável e dinâmica e a utilização de programação em linguagens como o C e o *Fortran*. O PVM é, finalmente, uma plataforma de computação científica viável tendo sido utilizado para simulações moleculares dinâmicas, estudos de supercondutividade, computações fractais distribuídas, algoritmos matriciais e como base para o ensino de computação concorrente.

Já o MPI teve sua primeira versão publicada em 1994 e atualizada em junho de 1995. Atualmente está sendo discutida uma nova extensão chamada MPI 2. É um produto resultante de um fórum aberto constituído de pesquisadores, empresas, usuários e vendedores que definiram a sintaxe, semântica e o conjunto de rotinas padronizadas para *message passing*.

Como características pode-se citar a eficiência pois foi projetado para executar eficientemente em máquinas diferentes. É especificado somente o funcionamento lógico das operações. A implementação fica a cargo do próprio desenvolvedor que usa as características de cada máquina para gerar um código mais otimizado.

4.2 OpenMP

A *Application Program Interface* - API de desenvolvimento OpenMP suporta uma programação multi plataforma com memória compartilhada em C/C++ e Fortran em todas as arquiteturas, incluindo plataformas Unix e Windows. Conjuntamente definida por um grupo relevante de empresas da área de *hardware* e *software*, o OpenMP é portável, com um modelo de memória escalável que possibilita uma programação de memória compartilhada entre computadores muito simplificada e flexível desde um computador pessoal até um supercomputador [10].

A OpenMP surgiu com a intenção de tornar a programação por memória compar-

tilhada viável e portável. A programação por memória compartilhada é a realizada em *clusters*. Um *cluster*, ou aglomerado de computadores, é formado por um conjunto de computadores, que utiliza um tipo especial de sistema operacional classificado como sistema distribuído. Muitas vezes é construído a partir de computadores pessoais, os quais são ligados em rede e comunicam-se através do sistema, trabalhando como se fossem uma única máquina de grande porte. Como exemplo de um sistema de gerenciamento de *cluster* citamos o sistema livre *Beowulf*, que é constituído por diversos nós escravos gerenciados por um só computador.

O OpenMP visa encapsular a complexidade o detalhamento para cada arquitetura necessário nas linguagens PVM e MPI.

O OpenMP não é uma biblioteca de comunicação. Enquanto as bibliotecas como o PVM e o MPI tinham como objetivo prover comunicação entre máquinas com memória distribuída, o OpenMP tem como objetivo prover a comunicação entre máquinas com memória compartilhada, tipicamente as que simulam memória compartilhada em cima de distribuída.

A necessidade de técnicas deste tipo deve-se a falta de portabilidade de programas escritos para este tipo de arquiteturas, pois cada fabricante tinha sua própria maneira de compartilhar áreas da memória com os outros processadores e isso fez com que muitos passassem a usar ainda as bibliotecas de comunicação nestas máquinas.

Em máquinas multinúcleo ou custers o OpenMP se mostra uma poderosa biblioteca, de muito fácil utilização, para a paralelização de aplicações.

4.2.1 Programação

O OpenMP é suportado por diversos compiladores uma lista detalhada é encontrada em [10]. Nesta tese foi utilizado o compilador GCC. Desde a versão 4.2 o GCC suporta a especificações do OpenMP.

Para compilar um programa com o GCC utilizando os comandos e funções do OpenMP devemos utilizar o parâmetro de linha de comando `-fopenmp`.

4.2.1.1 Controle do paralelismo

Nesta seção serão mostrados alguns exemplos de aplicação das estruturas de controle do paralelismo no OpenMP.

Estrutura: `parallel for`:

Paraleliza um enlace `for` dividindo-o em diversos *threads*.

```
#pragma omp parallel for [shared(vars), private(vars), firstprivate(vars)]
for(i=0;i<N;i++)
{
      C[i]=A[i]+B[i];
}
```

Estrutura: `sections`:

Esta diretiva cria seções paralelas onde cada seção será executada em um *thread* diferente:

```
main ()
{
   int x;
   #pragma omp sections
   {
      #pragma omp section
      {
          // thread 1
      }
      #pragma omp section
      {
          // thread 2
      }
   }
}
```

4.2.1.2 Diretivas

Estas diretivas configuram as estruturas de controle do paralelismo no trato das variáveis dos setores paralelizados.

`shared(vars)` - Compartilha as mesmas variáveis entre todos os *threads*.

`private(vars)` - Cada *thread* obtêm uma cópia privada de cada variável. Onde, apenas a *thread* principal tem estas variáveis inicializadas.

`firstprivate(vars)` - Mesmo funcionamento da anterior, mas, as variáveis tem seu conteúdo inicializado com o valor das variáveis da *thread* principal.

`default(private|shared|none)` - Define o comportamento padrão das variáveis de paralelização. A opção `none` obriga ao programador à definir o comportamento das variáveis.

4.2.1.3 Sincronização e travamento

Estas construções disponibilizam formas de sincronizar e travar a execução dos múltiplos *threads*.

Estrutura: `master`:

Apenas o *thread* principal irá executar neste trecho.

```
#pragma omp master
{
   // Código a ser executado apenas pelo thread principal
}
```

Estrutura: `critical`:

Trava a região de forma mutuamente exclusiva. O nome `name` possibilita a criação de regiões com identificação única.

```
#pragma omp critical [(name)]
{
   // Código a ser executado por apenas um thread por vez
}
```

Estrutura: `barrier`:

Força todos os *threads* a completarem suas operações antes de continuar a execução.

```
#pragma omp barrier
// Operações ou estruturas de apenas uma linha de código

#pragma omp atomic
// Simples operação matemática em tipos de dados primitivos Ex. a +=3
// As operações suportadas são: ++,--,+,*,- ,/,&,^,<<,>>,|
```

4.2.1.4 Configuração e controle

Funções de controle e configuração de execução.

`int omp_get_num_threads()` - Retorna o número de *threads* que são executados na região paralelizada onde esta função é chamada.

`int omp_get_thread_num()` - Retorna o número de identificação do *thread* que está sendo executado.

`int omp_in_parallel()` - Retorna se está em uma região paralelizada.

`int omp_get_max_threads()` - Retorna o número de *threads* que o OpenMP pode gerar.

`int omp_get_num_procs()` - Retorna o número de processadores do sistema.

`void omp_set_num_threads(int)` - Configura o número de *threads* que o OpenMP pode gerar.

Capítulo 5

Implementação

5.1 Introdução

Já foi abordado que o MMP possui um ótimo desempenho na compactação de imagens. Mas, este processo de compactação é de grande intensidade computacional levando na CPU AMD Athlon 64 X2 5200 com memória de 2GB DDR2 667 onde foi desenvolvida esta tese, uma ordem de tempo de 12 minutos para compactar uma imagem Lena monocromática com 512×512 pixels de resolução com 0,56 bits por simbolo. Como o MMP é baseado no processamento de blocos de imagem e possui tarefas de intensa procura em um dicionário de blocos, naturalmente, observou-se um potencial de paralelização. Com o surgimento da linguagem de desenvolvimento Nvidia Cuda e seus numerosos exemplos de aplicações portadas com notáveis ganhos de desempenho [21], observou-se a oportunidade de explorar o potencial acelerador nesta nova tecnologia e ao mesmo tempo começar a avaliar as melhores formas de paralelizar o compressor MMP.

Nesta implementação trabalharemos com a versão do MMP com o particionamento flexível.

Os códigos fonte das rotinas implementadas na GPU estarão disponíveis em cdrom anexo. Não foi possível a inclusão de suas listagens no corpo tese por causa do seu extenso tamanho.

5.2 Ambiente de desenvolvimento

O ambiente de desenvolvimento escolhido o GNU Linux pois o Cuda SDK tem um bom suporte à utilização de *makefiles* do compilador GNU gcc [13] sendo bem integrado as ferramentas de desenvolvimento do Linux. No início foi tentado o desenvolvimento no sistema operacional *Microsoft Windows*, mas, Cuda SDK 1.0 só possuía suporte a uma versão anterior do *Visual C++ Express Edition* (compilador C grátis da *Microsoft*) que não era mais disponível para descarga no site da *Microsoft*.

5.3 Partes paralelizáveis do MMP

Como estratégia de busca de definição de quais partes do algoritmo MMP seriam migradas para execução na GPU, buscamos identificar que partes do algoritmo eram as que tomavam mais tempo no processo de compressão, para então, verificar a possibilidade de as paralelizar. Como ferramenta para fazer esta avaliação de forma precisa foi utilizado o *GNU profiler - gprof*. Este utilitário grava a cada execução de um programa o número de chamadas das suas rotinas e a quantidade de tempo gasta em cada uma delas [12] e os respectivo tempo total.

Para usar o profile devemos compilar e likeditar o programa com o parâmetro -pg (-g +pg na compilação e -pg na linkedição).

Depois devemos rodar o programa normalmente e será criado no diretório atual o arquivo `gmon.out`. Então devemos chamar o programa `gprof` com o nome do executável na linha de comando e será exibido um relatório completo de quais rotinas e sub-rotinas estão tomando recursos de processamento do aplicativo em questão.

Para avaliar o relatório de desempenho obtido utilizamos o utilitário `kprof`. Este aplicativo abre a saída do `gprof` e exibe hierarquicamente as classes e sub-rotinas do aplicativo.

Pela avaliação do relatório do `gprof`, conforme esperado, as funções de maior custo de processamento do MMP são as rotinas `MenorJ` e `Distancia`. Desta forma, a primeira rotina escolhida para execução na GPU foi a `MenorJ` que é a rotina que busca em uma dimensão do dicionário qual o melhor bloco que poderia ser usado para codificar um outro, buscando de forma ponderada o de menor distorção e ao mesmo tempo proporcione a menor taxa de codificação. Além disso, dentro desta rotina está incluída a rotina

`Distancia`. Esta rotina calcula a distancia euclidiana entre um bloco e outro.

Outra rotina candidata era a `CalculaMapa`, pois, a mesma chama a cada execução milhares de vezes a rotina `MenorJ`. Realizando nas configurações iniciais do dicionário 200.704 chamadas à rotina `MenorJ`. Mais adiante, detalharemos a `CalculaMapa` e a solução de implementação de sua versão paralela.

5.4 Organização dos fontes

O SDK Cuda possui suporte à utilização de arquivos de *makefile* com disponibilização de extensões aos usuários do *GNU make*. Para utilização destas extensões devemos incluir no arquivo do projeto cujo nome é `"makefile"` a chamada ao arquivo `"common.mk"`. Abaixo é mostrado o arquivo de projeto utilizado nesta tese:

```
################################################################################
#
# Build script for project
#
################################################################################
# Add source files here
EXECUTABLE := MMP2D_CUDA-FT-DicGPU
# CUDA source files (compiled with cudacc)
CUFILES := MMP2D-FT-DicGPU.cu Dicionario2D-FT-DicGPU.cu
# CUDA dependency files
CU_DEPS := Dicionario2D-FT-DicGPU.h MMP2D-FT-DicGPU.h
# C/C++ source files (compiled with gcc / c++)
CCFILES := Aritmetico.cpp
################################################################################
# Rules and targets
include ../../common/common.mk
```

Os arquivos de fonte que possuem código que é executado na GPU devem possuir a extensão ".cu". Estes arquivos serão compilados com o programa `nvcc` e todo processo de compilação e geração de código *assembly* da CPU e GPU e seu correto escalonamento para execução será feito de forma transparente para o usuário. Em alguns momentos foi necessário a customização do executável com o fornecimento ao compilador *nvcc* de novos parâmetros de linha de comando. Para isso foi necessário obtermos detalhadamente como era feita a compilação e linkedição do executável. Para obtermos este detalhamento, é executado o programa *make* com o parâmetro de linha de comando -n, dessa forma, gerando um arquivo de lote com estas informações. Abaixo é mostrado o conteúdo um arquivo

de lote com a saída do comando `make` sobre o arquivo de projeto `Makefile-ft-DicGPU` (`make -n -f Makefile-ft-DicGPU`).

```
mkdir -p ../../lib

mkdir -p obj/release

mkdir -p ../../bin/linux/release

/usr/local/cuda/bin/nvcc   --compiler-options -fno-strict-aliasing --maxrregcount 8 -I.
-I/usr/local/cuda/include -I../../common/inc -DUNIX -O3   -o obj/release/MMP2D-FT-DicGPU.cu.o
-c MMP2D-FT-DicGPU.cu;

/usr/local/cuda/bin/nvcc   --compiler-options -fno-strict-aliasing --maxrregcount 8 -I. -I/usr/local/cuda/include
-I../../common/inc -DUNIX -O3   -o obj/release/Dicionario2D-FT-DicGPU.cu.o -c Dicionario2D-FT-DicGPU.cu;

/usr/local/cuda/bin/nvcc   --compiler-options  -fno-strict-aliasing --maxrregcount 8 -I. -I/usr/local/cuda/include
-I../../common/inc -DUNIX -O3   -o obj/release/Aritmetico.cu.o -c Aritmetico.cu;

g++ -fPIC -o MMP2D_CUDA-FT-DicGPU_G80   obj/release/MMP2D-FT-DicGPU.cu.o obj/release/Dicionario2D-FT-DicGPU.cu.o
obj/release/Aritmetico.cu.o     -L/usr/local/cuda/lib -L../../lib -L../../common/lib/linux  -lcudart -lcuda
-L/usr/local/cuda/lib -L../../lib -L../../common/lib/linux  -lcutil;
```

Devemos ter atenção ao fato que o Cuda SDK organiza os executáveis em uma árvore de arquivos que os separa por arquitetura de sistema operacional e modo de geração do executável.

Com o executável na sua versão *release* o executável é gerado no diretório:
`/home/usuario/NVIDIA_GPU_Computing_SDK/C/bin/linux/release`
Com o executável na sua versão emulada o executável é gerado no diretório:
`/home/usuario/NVIDIA_GPU_Computing_SDK/C/bin/linux/emu`

5.5 Modo de Emulação da GPU

O Cuda SDK possui um modo de criação do executável, onde a execução do *kernel* é feita na CPU simulando a sua execução em uma GPU. Este modo é muito útil para depuração de uma função *kernel* suportando depuradores e chamadas e funções entrada e saída de terminal (*printf*) facilitando o rastreamento de erros de programação. Por outro lado, este modo deve ser executado com muita atenção pois é muito comum um código que execute corretamente no modo emulado tenha falhas graves de endereçamento e de configuração de execução dos *threads* quando executado na GPU.

Como já citado nos capítulos anteriores, uma GPU com a *Compute Capability 1.0* e 4 multiprocessadores tem capacidade de executar um *kernel* com no máximo 3072 *threads*. No modo emulado não existe esta limitação no número de *threads* fazendo com que os resultados possam ser diferentes do encontrado no modo nativo da GPU.

No modo emulado, operações que causam erros no modo nativo, tais como, alocações de ponteiro para ponteiros (este erro está detalhadamente abordado no próximo capítulo que fala sobre o gerenciamento de memória), não resultam em erro no modo emulado nem de execução e nem de lógica. Isto faz com que o rastreamento destes erros seja bastante complicado dependendo do tamanho do projeto.

Para gerarmos o executável no modo emulado com depuração devemos colocar na linha de comando do utilitário as opções `dbg=1 emu=1`. Abaixo é mostrado um exemplo de linha de comando onde geramos o executável no modo emulado.

```
make -f Makefile-ft-DicGPU dbg=1 emu=1
```

5.6 Gerenciamento de memória

5.6.1 Ponteiros para ponteiros

A rotina de minimização dos custo lagrangiano (busca do menor J), que foi a primeira a ser escolhida para transporte para a GPU, faz intenso acesso ao dicionário de blocos, assim, justificando envio do dicionário. O dicionário está definido dentro de uma classe C++ em conjunto com alguns métodos de acesso a estes dados. Como o SDK Cuda não oferece suporte à classes C++ sendo executadas na GPU, foram migradas para a GPU apenas as rotinas e informações necessárias à execução da rotina de calculo do menor J.

No código fonte que foi portado para execução na GPU, o dicionário é definido como uma estrutura de de dados composta de ponteiros que são alocados para outras estruturas que podem ou não ter ponteiros. Fornecendo assim uma infraestrutura onde os dados e blocos de imagem são alocados dinamicamente.

Para utilizarmos estruturas de ponteiros para ponteiros na GPU devemos nos atentar ao fato que deveremos copiar explicitamente o endereço alocado na GPU para o ponteiro que faz referência a um outro ponteiro. Além disso, conforme afirmado no capítulo 3 página 44 do *Nvidia Cuda^{TM} Programming Guide Version 2.3.* [39] , devemos ter cuidado com o modo emulado do SDK, pois, estes tipos de referenciamento executarão correta-

mente no modo emulado mesmo sem a cópia explicita do endereço do ponteiro, mas, na execução no dispositivo teremos erros imprevisíveis.

Abaixo é mostrado um exemplo de código que cria um um ponteiro para uma lista de ponteiros e faz a sua correta inicialização.

```
00 int** a;
01 cudaMalloc(&a, sizeof(int*) * N));
02 int* ha[N];
03 for(int i = 0; i < N; ++i)
04      cudaMalloc(&ha[i],size));
05 cudaMemcpy(a, ha, sizeof(ha), cudaMemcpyHostToDevice);
```

Neste código observamos na linha 05 a cópia explícita do conteúdo do endereço de cada item do vetor de ponteiros para a variável `a` que é um ponteiro para uma lista de ponteiros.

Além disso, a estratégia de utilização de ponteiros para ponteiros é inevitavelmente ineficiente, pois, a GPU não possui memória cache. Então, cada referência de ponteiro para ponteiro irá incluir uma inevitável penalidade de latência de instruções de 200 ciclos [20]. Na CPU isso não ocorre, pois, cada vetor de ponteiros ira estar na memória cache fazendo este tipo de referência ser relativamente rápido.

Na solução implementada nesta tese buscamos evitar a alocação dinâmica de ponteiros implementando a alocação de todo dicionário no início da execução do programa.

5.6.2 Complexidade da rotina de *kernel*

Durante a implementação da rotina `MenorJ` ocorreu um problema durante a execução deste *kernel* fazendo com que ele não retornasse resultados, gerando a mensagem de erro *Unspecified Launch Failure*. O Cuda SDK não esclarece que tipos de erro poderiam gerar esta mensagem. Mas, pela pesquisa nos fóruns *Cuda Zone* da Nvidia foi afirmado que este problema poderia estar associado a falhas de segmentação ou limitações na complexidade da rotina de *kernel* somado ao número de *threads* colocados em execução.

Deve ser dada muita atenção, ao fato de, no modo emulado este erro não ocorrer sendo que o algoritmo retorna os resultados esperados.

Após uma criteriosa revisão dos ponteiros e verificação de possíveis invasões de memória, buscou-se verificar se a falha de execução era devido à complexidade da rotina de *kernel*.

Como estratégia de determinação do problema comentou-se toda a rotina *kernel* e verificamos que a partir daí não ocorreu mais o erro *Unspecified Launch Failure*. Após a realização de re-inclusões passo a passo de trechos de código, constatou-se que, não existiam erros de execução no código da rotina. Então buscou-se diminuir o tamanho do código da rotina *kernel* segmentando sua chamada duas ou mais rotinas mais simples.

Na rotina `MenorJ` é realizada uma consulta ao codificador aritmético a respeito do custo em bits da compressão do bloco do dicionário do qual está se ponderando a sua distorção em relação ao bloco que está sendo testado. Como o custo de processamento deste cálculo é baixo para CPU em relação ao cálculo da distorção do bloco, buscou-se transportá-la para fora do *kernel* mantendo-a como um atributo de consulta para cada bloco do dicionário na GPU. Assim, conseguiu-se a simplificação do código limitando a complexidade da rotina de *kernel* passando a chamada da rotina `MenorJ_GPU`, a executar corretamente.

5.7 Rotinas de temporização

Para criação das rotinas de temporização não foram utilizadas as chamadas as rotinas de temporização API C padrão. Pois, as mesmas por terem uma precisão de 55 milissegundos não são adequadas para realização destas medidas. Por isso, foram utilizadas rotinas providas pela Cuda SDK que possuem uma resolução em torno de 0.5 microssegundos [39]. Estas rotinas são: `cutCreateTimer`, `cutStartTimer`, `cutDeleteTimer` e `cutGetTimerValue`.

As rotinas básicas criadas estão listadas abaixo, mas, foram criadas rotinas mais complexas que criaram as tabelas de temporização exibidas nos resultados desta tese.

```
void ini_cronometro(int id)
{
   t_cronometro[id]=0;
   CUT_SAFE_CALL(cutCreateTimer(&t_cronometro[id]));
   CUT_SAFE_CALL(cutStartTimer(t_cronometro[id]));
}

float fim_cronometro(int id,char *msg)
{
   CUT_SAFE_CALL(cutStopTimer(t_cronometro[id]));
   float temp = cutGetTimerValue(t_cronometro[id]);
   if(msg!=NULL)
```

```
    printf("Tempo execução função: [%s] - %.3f (ms)\n",msg,temp);
  CUT_SAFE_CALL(cutDeleteTimer(t_cronometro[id]));
  return(temp);
}
```

5.8 Algoritmo de Redução

A busca do menor custo lagrangiano (menor J) é feita por cada dimensão do dicionário. Mas, antes desta busca é realizado cálculo da distorção e do custo de compressão do bloco a ser escolhido, pois, com estes dados calculamos o custo lagrangiano. Apois isso, ela se torna uma rotina de buscar o mínimo em um vetor de números reais (variáveis do tipo *float* na linguagem C). A primeira abordagem utilizada nesta tese foi a de disparar as *threads* utilizando uma variável do tipo global para armazenar o mínimo J encontrado. Esta abordagem não funcionou, pois, os diversos *threads* que foram disparados (número que varia de acordo como número de blocos da dimensão em que se busca, podendo ter de 250 até em torno de 7000 blocos para uma imagem Lena 512 x 512 e $\lambda = 30$), entraram em um conflito de acesso à variável onde deveria ser armazenada o mínimo J. Este tipo de problema é conhecido na literatura de sistemas operacionais como um problema típico de comunicação inter processos chamado de *race condition* [14]. Na arquitetura Nvidia este problema é causado por *threads* fora do mesmo *warp* tentando acessar o mesmo segmento de memória ao mesmo tempo, neste caso, apenas um dos *threads* terá sucesso. Como já foi citado nos capítulos anteriores isso ocorre devido a simplificações nos circuitos de acesso à memória das GPUs.

Existem diversas formas de abordagem à este problema. O Cuda SDK oferece paras as GPUs com a *Compute Capability 1.1* e acima um conjunto de funções atômicas que opera sobre dados de 32 bits ou 64 bits. Este tipo de solução conforme vemos em [17] soluciona o problema da exclusão mutua, mas, desperdiça processamento, pois, cada *thread* perde tempo (esperando a liberação do recurso em um processo chamado de espera ocupada) quando existe um conflito no acesso ao endereço de memória que é acessado pelo outros *threads*.

A solução adotada nesta tese foi o algoritmo de redução, abordado nas obras [16] e [18]. O algoritmo de redução consiste explorar as características massivamente paralelas das GPUs atuais, evitando os conflitos de acesso aos endereços de memória que são

compartilhados buscando organizar o algoritmo de forma a não existirem estes conflitos.

Os algoritmos de redução utilizam um enfoque baseado em árvore binária, para buscar a solução de diversos problemas típicos em computação tais como: Determinação de máximo, mínimo, ordenação e outros onde se adeque a utilização da técnica.

Subdividindo o problema em N pequenos problemas, como exemplificado na figura 5.1 , podemos utilizar múltiplos blocos de *threads* para buscar uma solução. Mantendo o máximo de multiprocessadores da GPU em utilização, com cada grupo de *threads* reduzindo o tamanho do problema a ser solucionado.

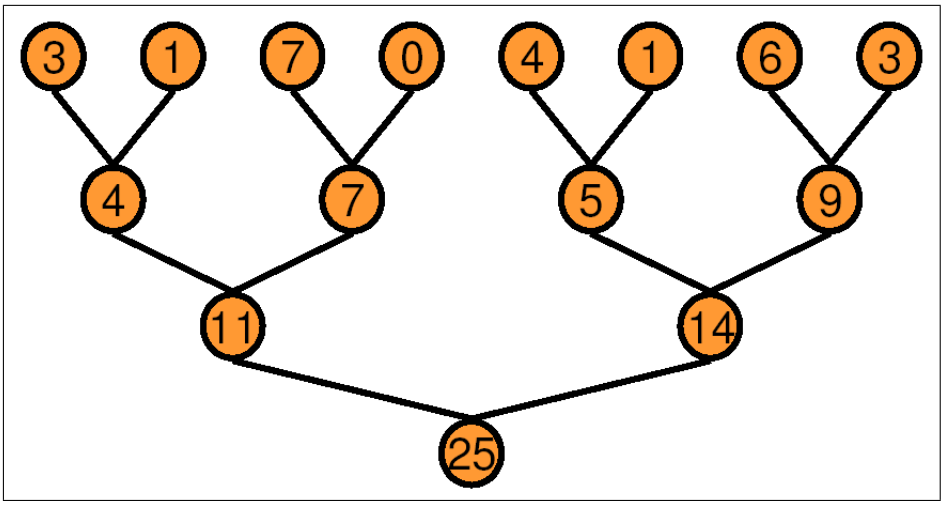

Figura 5.1: *Árvore de redução de oito threads que realiza um somatório*

Mas, para repassar os resultados para o novo bloco de *threads* que iram processar os resultados reduzidos devemos utilizar as rotinas de sincronização do Cuda SDK, tal como a `__syncthreads()`, para garantir que todos os *threads* tenham terminado o seu processamento.

Nas técnicas propostas no estudo da Nvidia [16] são baseadas no conhecimento prévio do número de dados que será processado. Propondo a criação de rotinas que trabalhem com vetores de tamanho fixo. No nosso problema o número de dimensões do dicionário por dimensão varia de 255 até 7000 elementos para a imagem de testes Lena na resolução de 512x512 e $\lambda = 30$. Por isso, foi necessária a criação de uma rotina de redução genérica que não fosse subdividida na chamada múltiplos códigos *kernel* diferentes.

Então, buscou-se a criação de uma rotina de busca do menor J onde a técnica de redução paralela baseada no endereçamento sequencial da memória global, conforme as técnicas descritas em [16], fazendo com que garantíssemos a a maior subdivisão possível

da tarefas em *threads* sem ocorrer conflitos de memória no acesso aos dados mutuamente acessado pelos *threads*. Assim, de dois a dois *threads*, é determinado o menor J entre eles e armazenado este valor na memória global, de forma livre de conflitos, para ser processado em uma próxima iteração até determinarmos o único menor J desta dimensão do dicionário. Na figura 5.2 vemos graficamente como é este algoritmo.

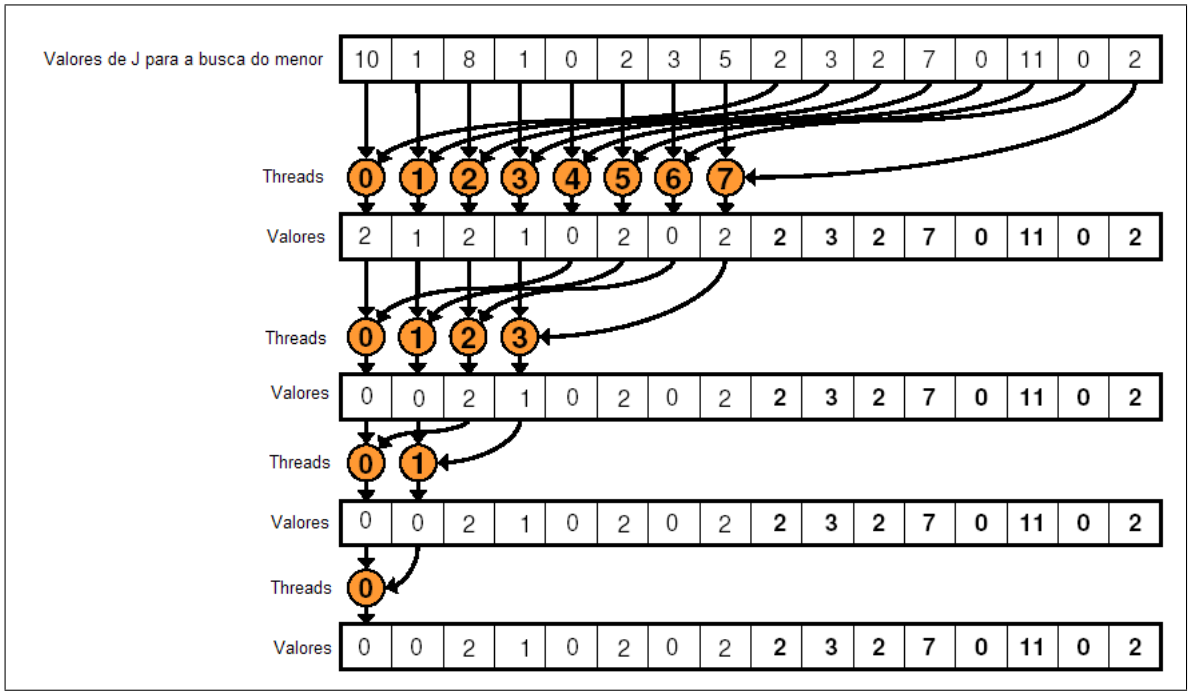

Figura 5.2: *Algoritmo de redução paralela sequencial para a busca do menor J*

É possível avaliar o ganho de processamento gerado por um algoritmo de redução de forma matemática. No algoritmo de busca do menor J em uma dimensão de um dicionário em uma CPU são realizadas n comparações sequenciais nesta busca. No caso de uma GPU com capacidade de execução simultânea de n *threads* são realizadas $n/2$ comparações paralelas e $log_2(n)$ comparações sequenciais. Desta forma poderíamos obter de forma teórica uma aceleração de $n/log_2(n)$ para cada dimensão do dicionário para o caso de uma CPU em que a velocidade de comparação fosse da mesma ordem que a comparação na GPU. Esta conta é teórica, pois, conforme veremos adiante as velocidades da CPU e GPU são diferentes e existem outras latências tais como a de invocação de um *kernel* e as de transferência de dados entre CPU e GPU e vice-versa.

Então, para a uma dimensão com 7000 elementos iríamos obter uma aceleração de $7000/log_2(7000) = 548.03\times$ para a menor dimensão de um dicionário que é de 255 elementos obteríamos uma aceleração de $255/log_2(255) = 31.9\times$.

Para o caso da nossa GPU que possui a limitação de 3072 *threads* foi criada a rotina `MenorJ_CUDA_G80` cujo *kernel* atende a este limite. Para GPUs com maiores recursos existe a rotina `MenorJ_CUDA`.

5.9 Alocação e Atualização Dinâmica do Dicionário e Codificador Aritmético na GPU

Durante a implementação foi observado que a maior penalização no desempenho das rotinas `Menor_J` e `CalculaMapa`, quando implementadas na GPU, era a preparação e envio dos parâmetros destas rotinas. Dessa forma, para distribuir o processamento e assim melhorar o seus desempenhos buscou-se implementar o envio do dicionário e sua atualização sempre que ocorresse esta atualização na CPU. O mesmo foi feito para a informação da taxa do codificador aritmético.

A atualização e envio do dicionário para a GPU foi feita nas seguintes funções da classe dicionário:

```
void Dicionario :: Inicializa
char Dicionario :: IncluiSubDic
```

Durante de envio e atualização dos blocos do dicionário na GPU também é feita a atualização da informação da taxa do codificador aritmético desses blocos.

No momento que o MMP codifica ou segmenta um bloco também é necessária a atualização da taxa dos blocos no dicionário da GPU, portanto, nas respectivas rotinas da Classe Codificador foram feitas atualização do dicionário na GPU.

```
void CodificadorMMPRD2D :: Codifica
void CodificadorMMPRD2D :: SegmentaBloco
```

Na seção onde avaliamos os resultados obtidos veremos que o envio dos dados e sua leitura da GPU é onde encontramos boa parte do tempo gasto no processamento das rotinas de GPU.

5.10 Implementação da rotina `CalculaMapa`

A rotina `CalculaMapa` calcula o custo lagrangiano de todos os nós de uma árvore de segmentação completa. Para um determinado bloco ela armazena o custo de todas as

possíveis segmentações combinadamente até a menor dimensão possível. As possíveis segmentações de um bloco são: não segmentar, segmentar na vertical e segmentar na horizontal. Durante a codificação, por várias vezes, calculamos o custo da segmentação de um bloco, assim, o objetivo desta rotina é acelerar a execução da codificação pré-armazenando estes custos em uma tabela de consulta.

Durante o desenvolvimento desta tese se observou a oportunidade de um grande ganho de desempenho com a implementação desta rotina na GPU. Na sua implementação original esta rotina para um dicionário inicial de 255 blocos 8×8 por dimensão para a montagem do mapa são necessárias 200.704 execuções da rotina `Menor_J`. Conforme observamos na figura 5.3 este valor é correspondente à combinação dos possíveis caminhos de segmentação: 4 possibilidades de segmentação vertical, 4 possibilidades de segmentação horizontal, 7 possibilidades de segmentação vertical e horizontal alternadamente e 7 possibilidades de e segmentação horizontal e vertical alternadamente. Assim sendo teríamos, como número de nós a serem guardados na `CalculaMapa` $4 \times 4 \times 7 \times 7 \times 256 = 200.704$ nós.

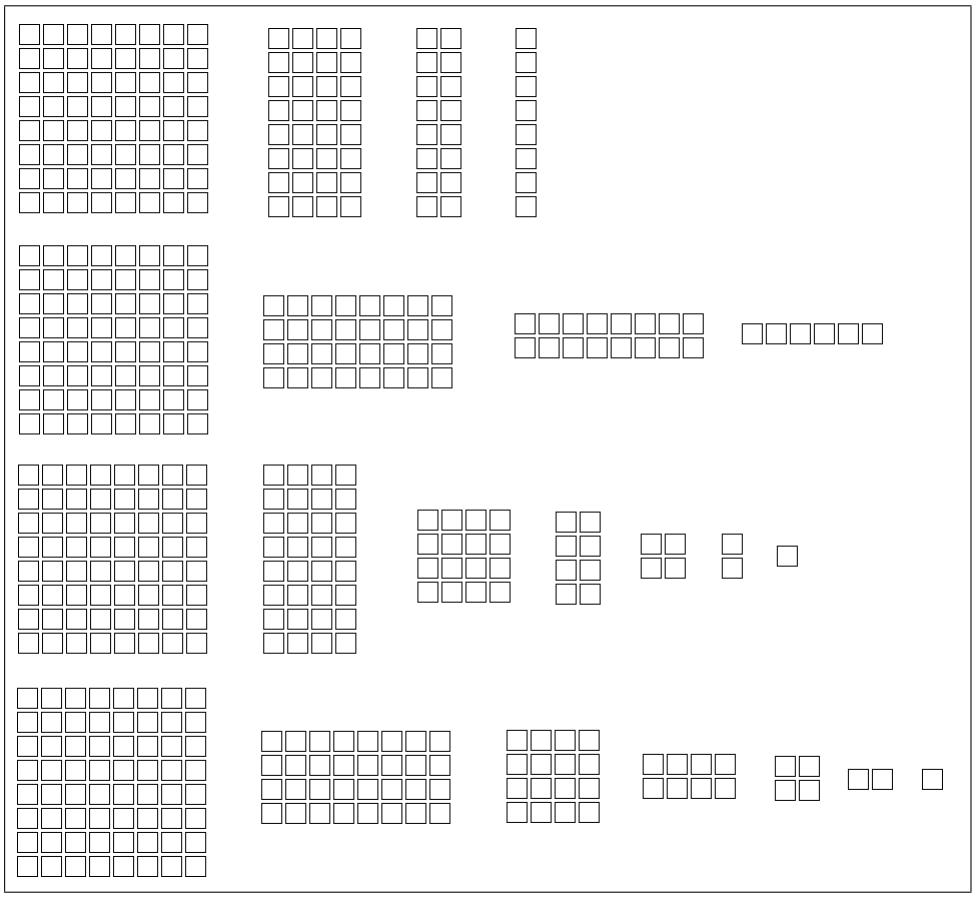

Figura 5.3: *Possibilidades de segmentação de um bloco 8 X 8*

Então, a rotina `CalculaMapa` na CPU realizaria 200.704 chamadas a rotina `Menor_J`. No início da codificação do MMP, com dicionários de tamanho inicial, a rotina `Menor_J` não ocupa totalmente os multiprocessadores, pois, são realizadas apenas 255 operações de cálculo de distância quando o potencial para uma GPU da série G80 seria de 3072. Então, existiria um relevante potencial de ganho de desempenho na implementação da `CalculaMapa` na GPU.

Na época do desenvolvimento desta rotina não sabiamos da limitação da GPU G80 de trabalhar com, no máximo, 3072 *threads*, pois, esta informação está indiretamente indicada no anexo onde está a tabela de *Compute Capabilities*. A leitura da documentação do SDK sem se detalhar na avaliação do anexo nos levou a considerar que as limitações no número de *threads* estavam apenas no tamanho máximo dos grids e blocos informados na saída da rotina `DeviceQuery`:

```
Device 0: "GeForce 8600 GT"
  Major revision number:                         1
  Minor revision number:                         1
  Total amount of global memory:                 536150016 bytes
  Number of multiprocessors:                     4
  Number of cores:                               32
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 8192
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          262144 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    1.19 GHz
  Concurrent copy and execution:                 Yes
```

Pela leitura da listagem acima poderíamos criar um grid de $65535 \times 65535 \times 1$ blocos com cada um 512 *threads* o que proporcionaria a criação de o grupo de 2.198.956.147.200 *threads* o que não é verdade conforme já esclarecido anteriormente.

Durante o desenvolvimento criamos rotinas de *kernel* que não verificavam o limite de 3072 *threads*. Quando isso acontece, não ocorre aviso de erro e os *threads* que seriam acima deste limite de 3072 não são executados alterando o resultados dos cálculos. Mesmo assim, foi criada uma versão da `CalculaMapa` na GPU onde não se impôs nenhum limite no número de *threads* para avaliarmos seu desempenho.

Na implementação da rotina optamos por criar um *thread* para cada nó da tabela gerada pela rotina `CalculaMapa` assim explorando ao máximo os recursos da GPU. Abaixo é mostrado como foram configuradas as dimensões de um grid que suportá-se todos os nós possíveis.

```
int gridXsize=(int)4;
int gridYsize=(int) iMaxDimDic;    // Maior de todas as dimensões do dicionário;
int blkXsize=(int)7;
int blkYsize=(int)4;
int blkZsize=(int)7;

dim3 dimGrd(gridXsize,gridYsize);
dim3 dimBlk(blkXsize,blkYsize,blkZsize);
```

Na rotina `CalculaMapa_GPU` foi necessária a criação de dois *kernels*, um para o calculo e armazenamento de todos os custos lagrangianos (`CalculaMapa_J_GPU_Kernel`) e outra para o calculo dos menores custos por redução (`CalculaMapa_BuscaMenorJ_GPU_Kernel`). Na seção onde avaliamos os resultados obtidos veremos que o envio dos dados e sua leitura da GPU é onde encontramos boa parte do tempo gasto no processamento das rotinas de GPU.

5.11 Versão sem envio do codificador aritmético para GPU

Na elaboração da tese não houve tem hábil para a implementação do codificador aritmético nativo na GPU e a solução encontrada foi o envio das informações e sua atualização. Buscando mais algum ganho no desempenho na GPU, foi implementada uma versão do MMP onde as taxas do codificador aritmético não eram enviadas para a GPU na execução das rotinas Calcula Mapa e Menor J.

Nesta versão a taxa do codificador aritmético era estimada pela log na base 2 do número de elementos do dicionário do bloco. O programa executável destas versões tem a sufixo `"RatesLog"` em seus nomes.

Capítulo 6

Resultados

6.1 Resultados implementação na GPU

Nesta seção avaliaremos os resultados obtidos. Para as rotinas desenvolvidas na GPU não disponibilizaremos a suas listagem devido extenso tamanho das mesmas. A listagem fonte MMP para a GPU está disponível em cdrom anexo ou em contato com autor.

6.1.1 Minimização do Custo Lagrangiano

Conforme já abordado nos capítulos anteriores a minimização do Custo Lagrangiano (busca do menor J) é feita na GPU se utilizando de técnicas de redução. As rotinas que fazem a minimização deste Custo são chamadas de `MenorJ`.

Um detalhamento a respeito dos tempos de execução da rotina `MenorJ_CUDA_G80` está listado na tabela 6.1. Como o próprio nome indica esta rotina calcula menor J na GPU se adequando as limitações das GPU com a arquitetura G80. Na tabela 6.1 são mostrados os tempos de execução da rotina Menor J na CPU e GPU e são detalhados os tempos de leitura e escrita à memória da GPU necessários para a implementação desta rotina. Como podemos observar para um dicionário de 0 a 500 elementos. compostos por blocos com a geometria 8x8, a rotina de menor J na CPU teve um tempo médio de 170 milissegundos e o *kernel* de calculo de menor J (`MenorJ_CUDA_G80`) teve um tempo médio de 286 milissegundos o que significa que, neste caso, a GPU é 1.68 vezes mais lenta. Mas, com o crescimento do dicionário isso se inverte conforme observamos na figura 6.1 que mostra graficamente a tabela 6.2.

Na figura 6.1 observamos que a rotina MenorJ GPU G80 Kernel inicialmente tem

Função	Geometria Num.Blocos	Tempo Médio(ms)
MenorJ CPU	8x8 (0-500)	170
MenorJ GPU G80	8x8 (0-500)	2103
MenorJ GPU G80 Kernel	8x8 (0-500)	286
MenorJ GPU G80 copia dados para GPU	8x8 (0-500)	1368
MenorJ GPU G80 copia OtimoGPU da GPU para CPU	8x8 (0-500)	14
MenorJ GPU G80 libera memoria GPU	8x8 (0-500)	392

Tabela 6.1: *Detalhamento Menor J sem enviar as taxas do codificador aritmetico*

Função	Geometria Num.Blocos	Tempo Médio(ms)
MenorJ CPU	8x8 (0-500)	165
MenorJ CPU	8x8 (500-1000)	492
MenorJ CPU	8x8 (1000-1500)	919
MenorJ CPU	8x8 (1500-2000)	1249
MenorJ CPU	8x8 (2000-2500)	1617
MenorJ CPU	8x8 (2500-3000)	1954
MenorJ CPU	8x8 (3000-3500)	2429
MenorJ CPU	8x8 (3500-4000)	2701
MenorJ CPU	8x8 (4000-4500)	3091
MenorJ CPU	8x8 (4500-5000)	3372
MenorJ GPU G80	8x8 (0-500)	2094
MenorJ GPU G80	8x8 (500-1000)	2397
MenorJ GPU G80	8x8 (1000-1500)	2727
MenorJ GPU G80	8x8 (1500-2000)	3058
MenorJ GPU G80	8x8 (2000-2500)	3414
MenorJ GPU G80	8x8 (2500-3000)	3707
MenorJ GPU G80	8x8 (3000-3500)	4340
MenorJ GPU G80	8x8 (3500-4000)	4631
MenorJ GPU G80	8x8 (4000-4500)	4962
MenorJ GPU G80	8x8 (4500-5000)	5166
MenorJ GPU G80 Kernel	8x8 (0-500)	292
MenorJ GPU G80 Kernel	8x8 (500-1000)	555
MenorJ GPU G80 Kernel	8x8 (1000-1500)	860
MenorJ GPU G80 Kernel	8x8 (1500-2000)	1198
MenorJ GPU G80 Kernel	8x8 (2000-2500)	1554
MenorJ GPU G80 Kernel	8x8 (2500-3000)	1839
MenorJ GPU G80 Kernel	8x8 (3000-3500)	1203
MenorJ GPU G80 Kernel	8x8 (3500-4000)	1355
MenorJ GPU G80 Kernel	8x8 (4000-4500)	1519
MenorJ GPU G80 Kernel	8x8 (4500-5000)	1614

Tabela 6.2: *Menor J G80 sem enviar as taxas do codificador aritmético*

um desempenho semelhante a CPU e a partir dos tamanhos de dicionários na faixa de 3000 a 3500 elementos existe um notável ganho de velocidade (em torno de 32%). Este ganho existe, pois, a partir de 3072 elementos a rotina de *kernel* é chamada novamente para um novo grupo 3072 elementos não existindo a reconfiguração da GPU. Por isso, sua velocidade aumenta consideravelmente. Esse aumento de velocidade existe porque as GPUs são otimizadas para execuções repetitivas das mesmas operações sobre um mesmo grupo de dados, fato que era típico em suas aplicações originárias, tais como, a animação de um objeto 3D.

Outra coisa observada na figura e nas tabelas é o grande impacto que o acesso a memória tem no desempenho das rotinas da GPU anulando os ganhos de velocidade na

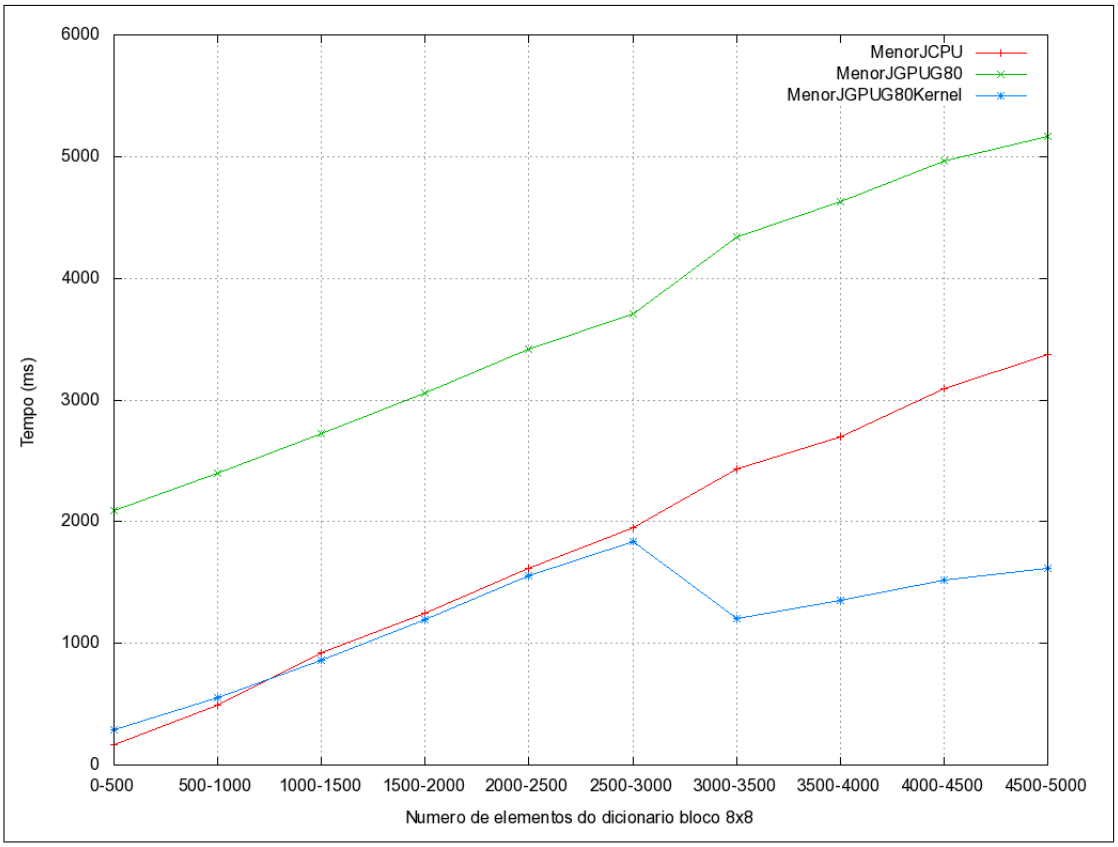

Figura 6.1: *Gráfico Menor J G80 sem enviar as taxas do codificador aritmético*

rotina de *kernel*. Veremos no decorrer deste capítulo que o tempo de transferência dos dados foi o grande impactante no sucesso da aplicação da GPU neste tipo algoritmo de compactação de imagens.

Analisando o desempenho da rotina Menor J GPU quando são variadas as dimensões dos blocos, observamos que não ocorrem grandes diferenças de desempenho como pode ser observado na figura 6.2 e na tabela 6.3. Nesta verificação buscou-se a utilização de dimensões do dicionário com 500 a 1000 elementos, por dimensão, pois, nestas dimensões temos a maior variedade de geometrias com números relevantes de elementos do dicionário.

Na versão com as limitações da arquitetura G80 e onde são enviadas as taxas do codificador foram obtidos os resultados que são enumerados na tabela 6.4 e na figura 6.3. Conforme esperado esta versão tem um menor desempenho devido ao tempo gasto com o envio das taxas do codificador.

Na versão sem as limitações de *threads* existentes na arquitetura G80 e onde não são enviadas as taxas do codificador aritmético foram obtidos os resultados enumerados

Função	Geometria Num.Blocos	Tempo Médio(ms)
MenorJ CPU	1x2 (500-1000)	158
MenorJ CPU	1x4 (500-1000)	185
MenorJ CPU	1x8 (500-1000)	231
MenorJ CPU	2x1 (500-1000)	174
MenorJ CPU	2x2 (500-1000)	177
MenorJ CPU	2x4 (500-1000)	204
MenorJ CPU	2x8 (500-1000)	301
MenorJ CPU	4x1 (500-1000)	204
MenorJ CPU	4x2 (500-1000)	258
MenorJ CPU	4x4 (500-1000)	319
MenorJ CPU	4x8 (500-1000)	368
MenorJ CPU	8x1 (500-1000)	294
MenorJ CPU	8x2 (500-1000)	362
MenorJ CPU	8x4 (500-1000)	448
MenorJ CPU	8x8 (500-1000)	492
MenorJ GPU G80 Kernel	1x2 (500-1000)	138
MenorJ GPU G80 Kernel	1x4 (500-1000)	157
MenorJ GPU G80 Kernel	1x8 (500-1000)	185
MenorJ GPU G80 Kernel	2x1 (500-1000)	138
MenorJ GPU G80 Kernel	2x2 (500-1000)	157
MenorJ GPU G80 Kernel	2x4 (500-1000)	185
MenorJ GPU G80 Kernel	2x8 (500-1000)	247
MenorJ GPU G80 Kernel	4x1 (500-1000)	154
MenorJ GPU G80 Kernel	4x2 (500-1000)	186
MenorJ GPU G80 Kernel	4x4 (500-1000)	246
MenorJ GPU G80 Kernel	4x8 (500-1000)	359
MenorJ GPU G80 Kernel	8x1 (500-1000)	186
MenorJ GPU G80 Kernel	8x2 (500-1000)	254
MenorJ GPU G80 Kernel	8x4 (500-1000)	362
MenorJ GPU G80 Kernel	8x8 (500-1000)	555
MenorJ GPU G80	1x2 (500-1000)	1260
MenorJ GPU G80	1x4 (500-1000)	1343
MenorJ GPU G80	1x8 (500-1000)	1406
MenorJ GPU G80	2x1 (500-1000)	1296
MenorJ GPU G80	2x2 (500-1000)	1321
MenorJ GPU G80	2x4 (500-1000)	1394
MenorJ GPU G80	2x8 (500-1000)	1558
MenorJ GPU G80	4x1 (500-1000)	1335
MenorJ GPU G80	4x2 (500-1000)	1399
MenorJ GPU G80	4x4 (500-1000)	1558
MenorJ GPU G80	4x8 (500-1000)	1852
MenorJ GPU G80	8x1 (500-1000)	1411
MenorJ GPU G80	8x2 (500-1000)	1657
MenorJ GPU G80	8x4 (500-1000)	1860
MenorJ GPU G80	8x8 (500-1000)	2397

Tabela 6.3: *Menor J G80 sem enviar as taxas do codificador aritmético variando a geometria dos blocos*

na tabela 6.5 e na figura 6.4.

Na versão sem as limitações da arquitetura G80 e onde são enviadas as taxas do codificador foram obtidos os resultados enumerados na tabela 6.6 e na figura 6.5.

Podemos observar que, nas duas versões onde configuramos o *kernel* para executar com o máximo de *threads* possível, o número de elementos do dicionário, praticamente, não influencia no desempenho da rotina *kernel*, pois, o tempo de execução que cada *thread*

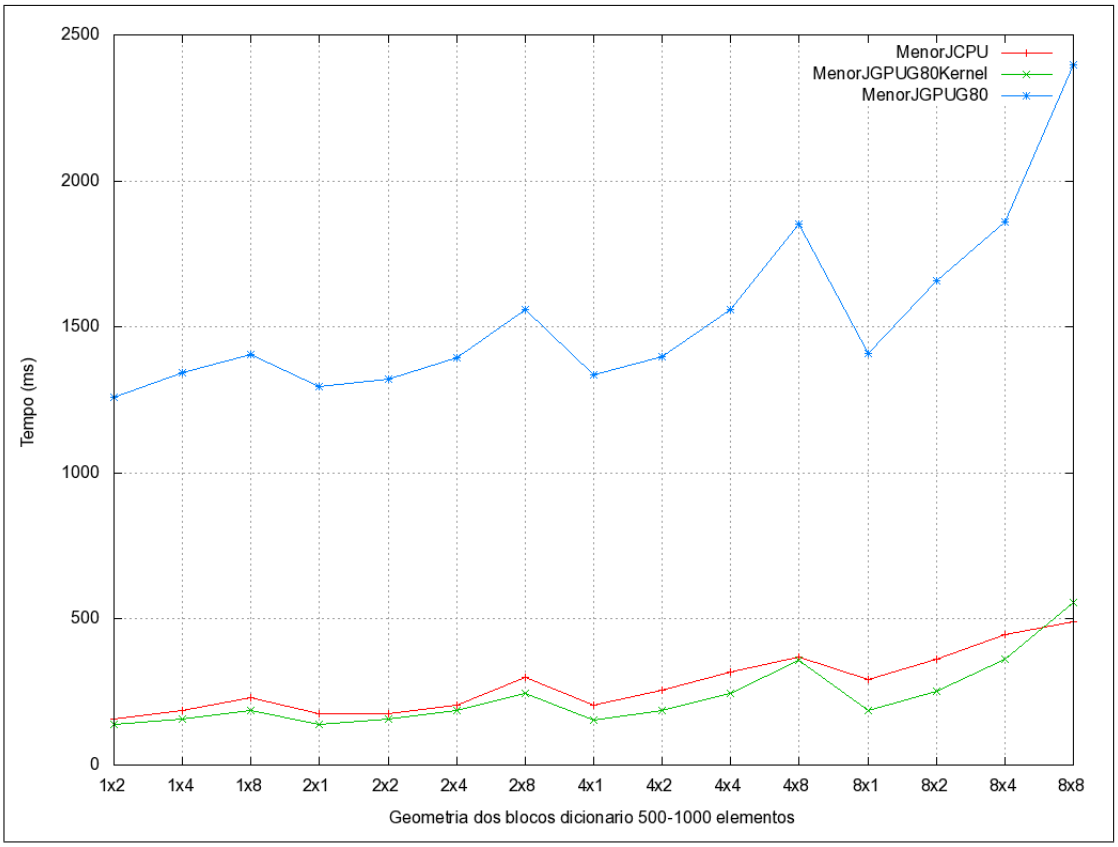

Figura 6.2: *Gráfico Menor J G80 sem enviar as taxas do codificador aritmético variando a geometria dos blocos*

Função	Geometria Num.Blocos	Tempo Médio(ms)
MenorJ CPU	8x8 (0-500)	156
MenorJ CPU	8x8 (500-1000)	437
MenorJ CPU	8x8 (1500-2000)	1419
MenorJ CPU	8x8 (3000-3500)	2950
MenorJ CPU	8x8 (4000-4500)	4981
MenorJ GPU G80	8x8 (0-500)	2177
MenorJ GPU G80	8x8 (500-1000)	2374
MenorJ GPU G80	8x8 (1500-2000)	3192
MenorJ GPU G80	8x8 (3000-3500)	5278
MenorJ GPU G80	8x8 (4000-4500)	7864
MenorJ GPU G80 Kernel	8x8 (0-500)	319
MenorJ GPU G80 Kernel	8x8 (500-1000)	505
MenorJ GPU G80 Kernel	8x8 (1500-2000)	1246
MenorJ GPU G80 Kernel	8x8 (3000-3500)	1209
MenorJ GPU G80 Kernel	8x8 (4000-4500)	2150

Tabela 6.4: *Menor J G80 enviando as taxas do codificador aritmético*

individualmente leva é o tempo de execução da rotina. Mas, na verdade o que acontece é que os *threads* acima do limite de 3070 não são executados dessa forma fazendo que o menor J encontrado não seja o mínimo global.

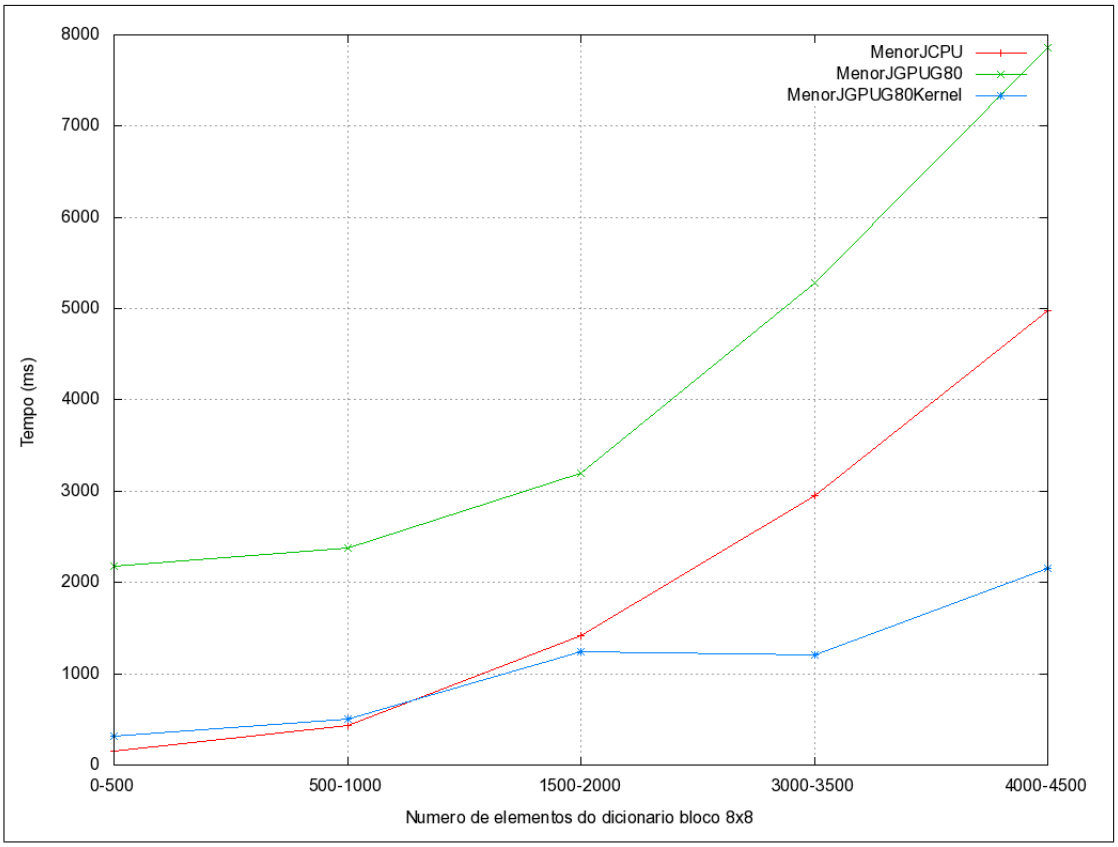

Figura 6.3: *Gráfico Menor J G80 enviando as taxas do codificador aritmético*

6.1.2 Rotina Calcula Mapa

Conforme já foi abordado no capítulo Implementação, se observou a oportunidade de um grande ganho de desempenho com a implementação desta rotina na GPU. Na sua implementação original esta rotina para um dicionário inicial de 255 blocos com a geometria 8×8 por dimensão para a montagem do mapa são necessárias 200.704 execuções da rotina `Menor_J`. Conforme observamos na figura 5.3 este valor é correspondente à combinação dos possíveis caminhos de segmentação: 4 possibilidades de segmentação vertical, 4 possibilidades de segmentação horizontal, 7 possibilidades de segmentação vertical e horizontal alternadamente e 7 possibilidades de e segmentação horizontal e vertical alternadamente. Assim sendo teríamos, como número de nós a serem guardados na `CalculaMapa` $4 \times 4 \times 7 \times 7 \times 255 = 200.704$ nós.

Então o potencial de paralelização da calcula mapa ao se comparar com a menor J é muito maior, pois, para um dicionário inicial a menor J precisa de 255 *threads* para sua execução e a calcula mapa precisa de $255 \times 200.704 = 51.179.520$ *threads*.

Na calcula mapa observamos os mesmos problemas encontrados na menor J: Apesar

Função	Geometria Num.Blocos	Tempo Médio(ms)
MenorJ CPU	8x8 (0-500)	161
MenorJ CPU	8x8 (500-1000)	448
MenorJ CPU	8x8 (1000-1500)	803
MenorJ CPU	8x8 (1500-2000)	1120
MenorJ CPU	8x8 (2000-2500)	1486
MenorJ CPU	8x8 (2500-3000)	1783
MenorJ CPU	8x8 (3000-3500)	2074
MenorJ CPU	8x8 (3500-4000)	2423
MenorJ CPU	8x8 (4000-4500)	2761
MenorJ GPU	8x8 (0-500)	2070
MenorJ GPU	8x8 (500-1000)	2352
MenorJ GPU	8x8 (1000-1500)	2669
MenorJ GPU	8x8 (1500-2000)	3013
MenorJ GPU	8x8 (2000-2500)	3385
MenorJ GPU	8x8 (2500-3000)	3673
MenorJ GPU	8x8 (3000-3500)	3980
MenorJ GPU	8x8 (3500-4000)	4337
MenorJ GPU	8x8 (4000-4500)	4640
MenorJ GPU Kernel	8x8 (0-500)	12
MenorJ GPU Kernel	8x8 (500-1000)	12
MenorJ GPU Kernel	8x8 (1000-1500)	12
MenorJ GPU Kernel	8x8 (1500-2000)	12
MenorJ GPU Kernel	8x8 (2000-2500)	12
MenorJ GPU Kernel	8x8 (2500-3000)	12
MenorJ GPU Kernel	8x8 (3000-3500)	12
MenorJ GPU Kernel	8x8 (3500-4000)	12
MenorJ GPU Kernel	8x8 (4000-4500)	12

Tabela 6.5: *Menor J sem limite de threads sem enviar as taxas do codificador aritmético*

Função	Geometria Num.Blocos	Tempo Médio(ms)
MenorJ CPU	8x8 (0-500)	158
MenorJ CPU	8x8 (500-1000)	445
MenorJ CPU	8x8 (1500-2000)	1451
MenorJ CPU	8x8 (6000-6500)	4938
MenorJ GPU	8x8 (0-500)	2079
MenorJ GPU	8x8 (500-1000)	2314
MenorJ GPU	8x8 (1500-2000)	3151
MenorJ GPU	8x8 (6000-6500)	6065
MenorJ GPU Kernel	8x8 (0-500)	13
MenorJ GPU Kernel	8x8 (500-1000)	14
MenorJ GPU Kernel	8x8 (1500-2000)	15
MenorJ GPU Kernel	8x8 (6000-6500)	70

Tabela 6.6: *Menor J sem limite de threads enviando as taxas do codificador aritmético*

da calcula mapa ter maior ganho de desempenho pelo maior potencial de paralelização estes ganhos são anulados pela preparação e envio dos parâmetros necessários para a GPU. Vale ressaltar que na calcula mapa não é enviado o dicionário que já foi atualizado dinamicamente juntamente com as atualizações do dicionário da CPU. Mas, apenas os parâmetros já impactam enormemente seu desempenho quando comparada a CPU.

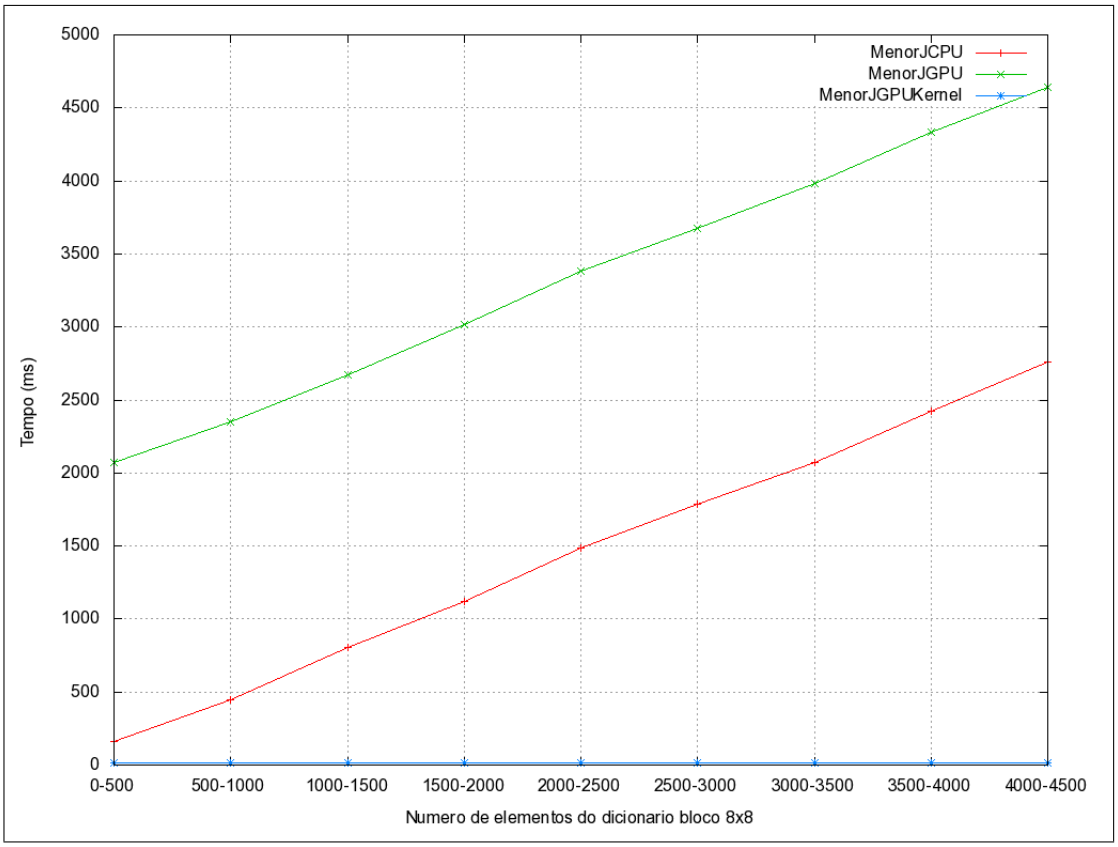

Figura 6.4: *Menor J sem limite de threads sem enviar as taxas do codificador aritmético*

Função	Tempo Médio(ms)	% Tempo Total
CalculaMapa_CPU	39667	6.249219%
CalculaMapa_GPU_G80	767996438	47.042301%

Tabela 6.7: *Tempo total CalculaMapa G80 sem enviar as taxas do codificador aritmético*

Função	Geometria Num.Blocos	Tempo Médio(ms)	% Tempo Total
CalculaMapa_CPU	(0-500)	8288	0.320342%
CalculaMapa_CPU	(500-1000)	17541	0.729528%
CalculaMapa_CPU	(1000-1500)	32903	0.652998%
CalculaMapa_CPU	(1500-2000)	52141	1.207263%
CalculaMapa_CPU	(2000-2500)	80898	1.328005%
CalculaMapa_CPU	(2500-3000)	112488	2.011963%
CalculaMapa_GPU_G80	(0-500)	270724	10.463714%
CalculaMapa_GPU_G80	(500-1000)	285320	11.866748%
CalculaMapa_GPU_G80	(1000-1500)	298697	5.927956%
CalculaMapa_GPU_G80	(1500-2000)	313817	7.266031%
CalculaMapa_GPU_G80	(2000-2500)	328764	5.396944%
CalculaMapa_GPU_G80	(2500-3000)	342246	6.121393%

Tabela 6.8: *CalculaMapa G80 sem enviar as taxas do codificador aritmético*

6.1.3 Envio do dicionário para a GPU

Conforme já abordado anteriormente, durante a implementação foi observado que a maior penalização no desempenho da rotina `Menor_J` e `CalculaMapa` quando implementadas na

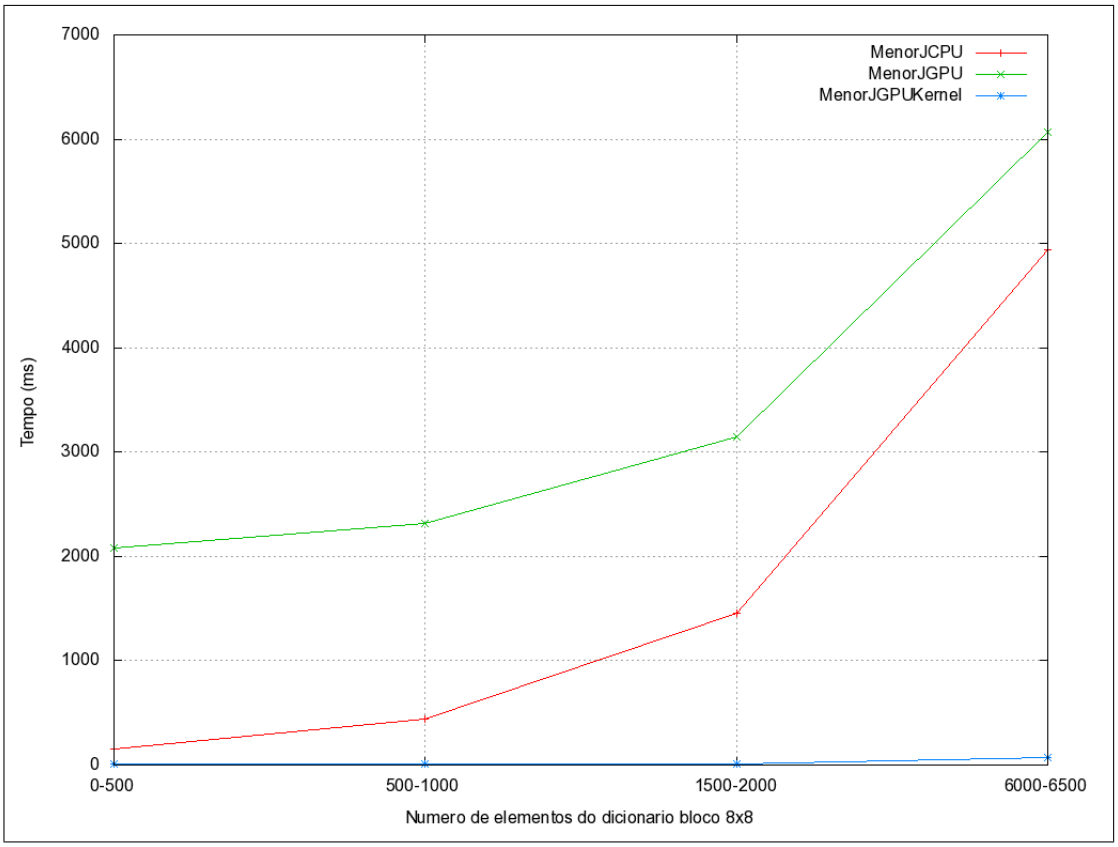

Figura 6.5: *Menor J sem limite de threads enviando as taxas do codificador aritmético*

Função	Tempo Médio(ms)	% Tempo Total
CalculaMapa_CPU	56217	1.167916%
CalculaMapa_GPU_G80	319019	6.627656%

Tabela 6.9: *Tempo total CalculaMapa G80 enviando as taxas do codificador aritmético*

GPU era a preparação e envio do dicionário e parâmetros destas rotinas. Dessa forma, para distribuir o processamento e assim melhorar o seus desempenhos buscou-se implementar o envio do dicionário e sua atualização sempre que ocorre-se esta atualização na CPU. O mesmo foi feito para a informação da taxa do codificador aritmético.

Avaliando a tabela 6.1.3 observamos que a alocação do dicionário na GPU (construtor do dicionário) é enormemente mais lenta na GPU (13.579 vezes mais lenta). Isso ocorre pelo fato do dicionário ser alocado de uma vez só na GPU. Enquanto, na CPU só ocorre alocação de memória no momento da inclusão de novos blocos.

Outro fato observado é que a inicialização dos blocos do dicionário inicial indicada pelas funções "Dicionário Inicializa Bloco na CPU"e "Dicionário Inicializa Bloco na GPU"é 116 vezes mais lento na GPU.

Na inclusão de um elemento (bloco) ao dicionário denotada pelas funções "Dici-

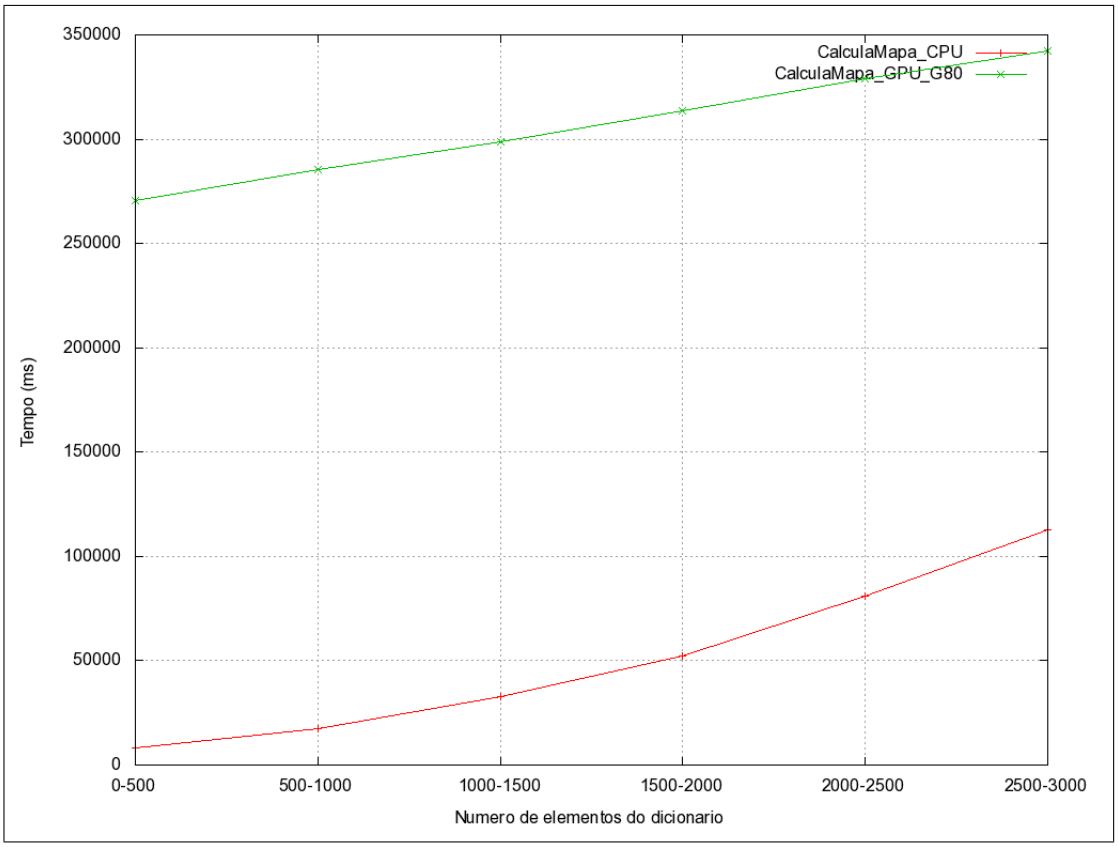

Figura 6.6: *CalculaMapa G80 sem enviar as taxas do codificador aritmético*

Função	Geometria Num.Blocos	Tempo Médio(ms)	% Tempo Total
CalculaMapa_CPU	(0-500)	10230	0.087351%
CalculaMapa_CPU	(500-1000)	19952	0.042926%
CalculaMapa_CPU	(1000-1500)	33148	0.040115%
CalculaMapa_CPU	(1500-2000)	52531	0.070636%
CalculaMapa_CPU	(2000-2500)	72711	0.092883%
CalculaMapa_CPU	(2500-3000)	93211	0.125337%
CalculaMapa_CPU	(3000-3500)	119535	0.168772%
CalculaMapa_CPU	(3500-4000)	138473	0.176890%
CalculaMapa_CPU	(4000-4500)	159530	0.235966%
CalculaMapa_CPU	(4500-5000)	171948	0.127167%
CalculaMapa_GPU_G80	(0-500)	290827	2.483261%
CalculaMapa_GPU_G80	(500-1000)	301469	0.648601%
CalculaMapa_GPU_G80	(1000-1500)	311102	0.376495%
CalculaMapa_GPU_G80	(1500-2000)	323390	0.434852%
CalculaMapa_GPU_G80	(2000-2500)	330610	0.422332%
CalculaMapa_GPU_G80	(2500-3000)	336054	0.451881%
CalculaMapa_GPU_G80	(3000-3500)	353786	0.499511%
CalculaMapa_GPU_G80	(3500-4000)	370363	0.473114%
CalculaMapa_GPU_G80	(4000-4500)	374266	0.553589%
CalculaMapa_GPU_G80	(4500-5000)	384121	0.284083%

Tabela 6.10: *Calcula Mapa G80 enviando as taxas do codificador aritmético*

onário IncluiSubDic na CPU"e "Dicionário IncluiSubDic na GPU"seus tempos são bem semelhantes. Isso ocorre porque, na CPU temos a alocação do bloco nesta inclusão fazendo anular a maior velocidade de acesso a memória na CPU.

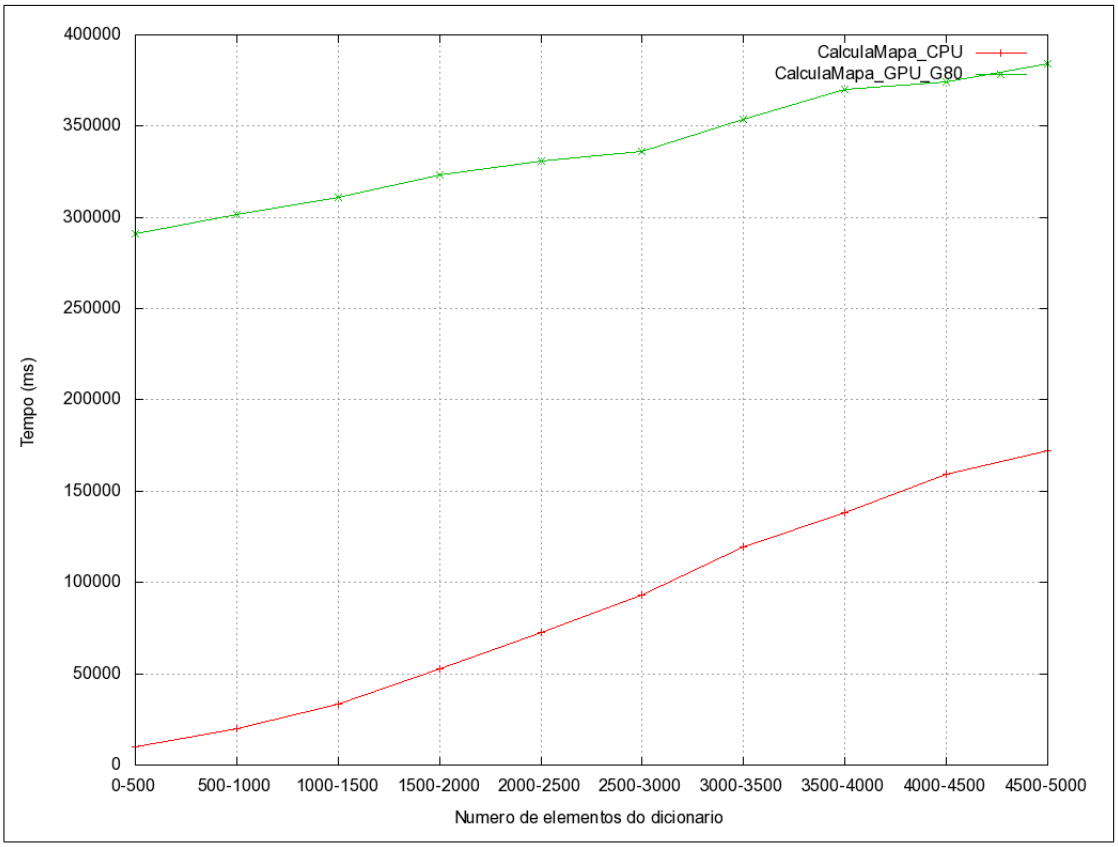

Figura 6.7: *CalculaMapa G80 enviando as taxas do codificador aritmético*

Função	Geometria Num.Blocos	Tempo Médio(ms)	% Tempo Total
CalculaMapa_CPU	Tempo Total	85571	58.765596%
CalculaMapa_GPU	Tempo Total	40888	28.079574%

Tabela 6.11: *Tempo total Calcula Mapa sem enviar as taxas do codificador aritmético*

Função	Geometria Num.Blocos	Tempo Médio(ms)	% Tempo Total
CalculaMapa_CPU	(0-500)	8109	0.857881%
CalculaMapa_CPU	(500-1000)	16624	1.859080%
CalculaMapa_CPU	(1000-1500)	31113	1.731848%
CalculaMapa_CPU	(1500-2000)	50646	3.235208%
CalculaMapa_CPU	(2000-2500)	78007	4.002113%
CalculaMapa_CPU	(2500-3000)	106994	7.014077%
CalculaMapa_CPU	(3000-3500)	137296	10.105532%
CalculaMapa_CPU	(3500-4000)	169820	11.218106%
CalculaMapa_CPU	(4000-4500)	201452	18.745626%
CalculaMapa_GPU	(0-500)	40882	4.325081%
CalculaMapa_GPU	(500-1000)	40892	4.572976%
CalculaMapa_GPU	(1000-1500)	40894	2.276341%
CalculaMapa_GPU	(1500-2000)	40890	2.612029%
CalculaMapa_GPU	(2000-2500)	40890	2.097867%
CalculaMapa_GPU	(2500-3000)	40894	2.680836%
CalculaMapa_GPU	(3000-3500)	40884	3.009230%
CalculaMapa_GPU	(3500-4000)	40897	2.701614%
CalculaMapa_GPU	(4000-4500)	40898	3.805674%

Tabela 6.12: *Calcula Mapa sem enviar as taxas do codificador aritmético*

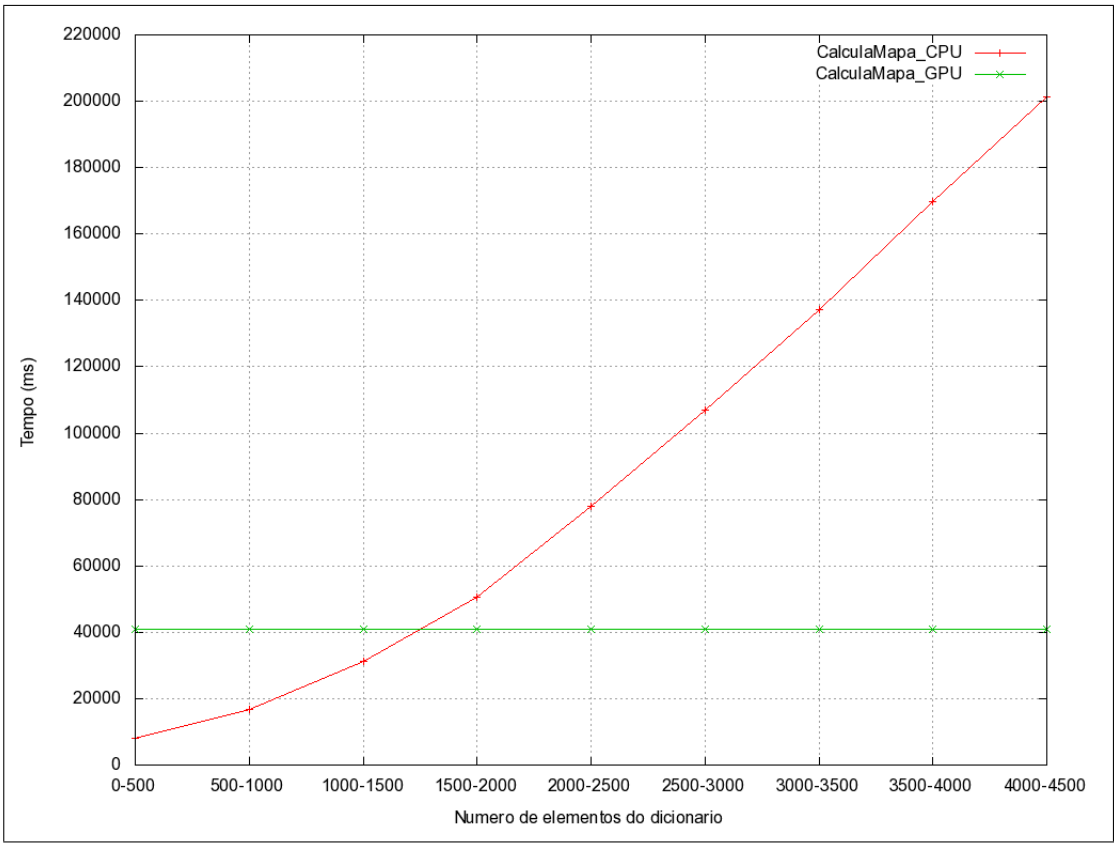

Figura 6.8: *Calcula Mapa sem enviar as taxas do codificador aritmético*

Função	Geometria Num.Blocos	% Tempo Total
CalculaMapa_CPU	Tempo Total	1.288494%
CalculaMapa_GPU	Tempo Total	0.660327%

Tabela 6.13: *Tempo total Calcula Mapa enviando as taxas do codificador aritmético*

Na coluna "% Tempo Total"é indicado qual o percentual do tempo total da execução das tarefas que são executadas na CPU ou GPU. Neste tempo total levamos em conta os tempos de leitura e atualização das memórias e execução das rotinas de menor J e calcula mapa. Deve ser ressaltado, que este tempo total não é o de execução de todo o processo de compactação e sim o tempo das tarefas relacionadas neste parágrafo. No computo do tempo total as tarefas de acesso ao dicionário são contabilizadas conjuntamente nos contadores de tempo total da CPU e da GPU. E as rotinas menor J e calcula mapa, cada uma com sua versão da CPU ou GPU, são contabilizadas nos contadores de tempo total respectivo, pois, as rotinas de CPU só são executadas para podermos comparar os resultados com a GPU.

A maior parte do tempo despendido na comunicação com a memória está localizado na função "GPU_AtualizaRatesBlocosEscala"que realiza o envio da taxa dos blocos de

Função	Geometria Num.Blocos	Tempo Médio(ms)	% Tempo Total
CalculaMapa_CPU	(0-500)	9768	0.052773%
CalculaMapa_CPU	(500-1000)	18429	0.025087%
CalculaMapa_CPU	(1000-1500)	31397	0.024042%
CalculaMapa_CPU	(1500-2000)	49440	0.042064%
CalculaMapa_CPU	(2000-2500)	69138	0.055882%
CalculaMapa_CPU	(2500-3000)	84054	0.071514%
CalculaMapa_CPU	(3000-3500)	104984	0.089322%
CalculaMapa_CPU	(3500-4000)	125711	0.101609%
CalculaMapa_CPU	(4000-4500)	142744	0.133594%
CalculaMapa_CPU	(4500-5000)	158821	0.135127%
CalculaMapa_CPU	(5000-5500)	177467	0.143442%
CalculaMapa_CPU	(5500-6000)	196052	0.183484%
CalculaMapa_CPU	(6000-6500)	214110	0.182168%
CalculaMapa_CPU	(6500-7000)	227930	0.048481%
CalculaMapa_GPU	(0-500)	40328	0.217876%
CalculaMapa_GPU	(500-1000)	40505	0.055140%
CalculaMapa_GPU	(1000-1500)	40415	0.030947%
CalculaMapa_GPU	(1500-2000)	40472	0.034434%
CalculaMapa_GPU	(2000-2500)	40435	0.032682%
CalculaMapa_GPU	(2500-3000)	40687	0.034617%
CalculaMapa_GPU	(3000-3500)	40719	0.034644%
CalculaMapa_GPU	(3500-4000)	40746	0.032934%
CalculaMapa_GPU	(4000-4500)	40716	0.038105%
CalculaMapa_GPU	(4500-5000)	40704	0.034632%
CalculaMapa_GPU	(5000-5500)	40687	0.032886%
CalculaMapa_GPU	(5500-6000)	40739	0.038128%
CalculaMapa_GPU	(6000-6500)	40770	0.034687%
CalculaMapa_GPU	(6500-7000)	40718	0.008661%

Tabela 6.14: *Calcula Mapa enviando as taxas do codificador aritmético*

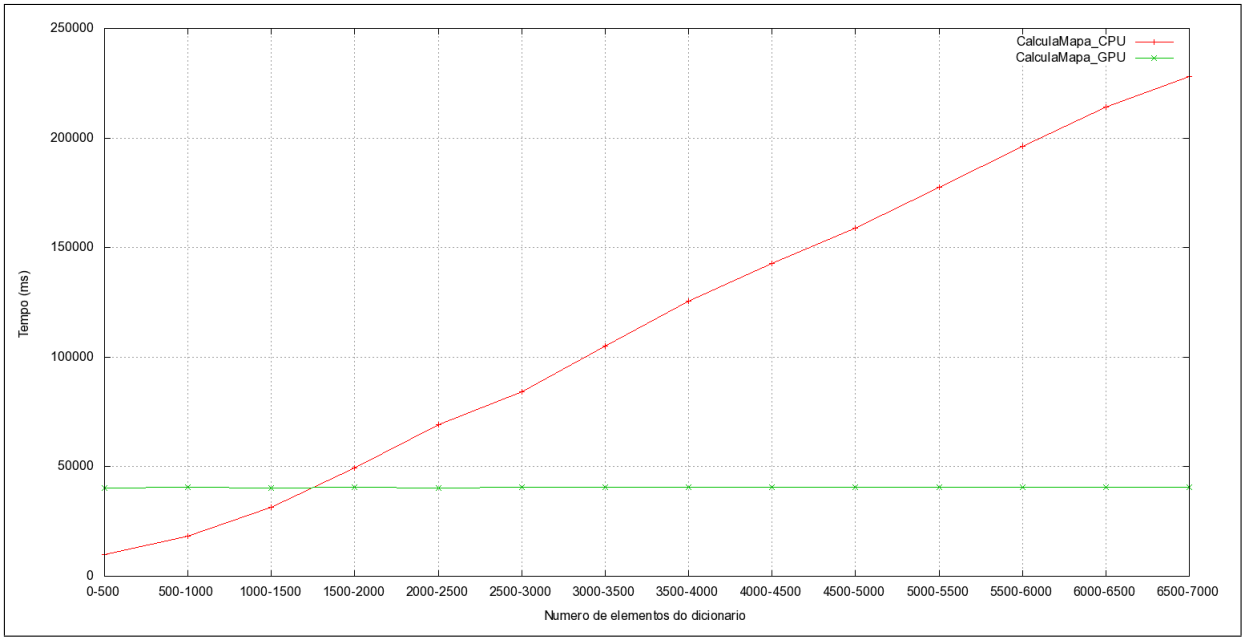

Figura 6.9: *Calcula Mapa enviando as taxas do codificador aritmético*

toda uma escala do dicionário. Esta função é executada quando um bloco é segmentado e ocorre uma atualização do dicionário. O grande tempo dispendido por esta rotina é

decorrente do calculo da taxa de no mínimo 255 elementos de uma dimensão do dicionário e seu respectivo envio para a GPU. Como resultante deste fato esta rotina toma 85% do tempo total do processamento que significam 5,62 minutos de execução.

Função	Tempo Médio(ms)	Vezes Execuçao	Tempo Total Execução (ms)	% Tempo Total
Construtor Dicionario	3	1	3	0.000000%
Construtor GPU_Dicionario	40737	1	40737	0.002739%
Dicionário Dimensiona CPU parte 1	3	1	3	0.000000%
Dicionário Dimensiona na GPU	219	1	219	0.000015%
Dicionário Dimensiona CPU/GPU parte 2	404	1	404	0.000027%
Alocando elementos subdicionarios CPU	3	16	41	0.000003%
Dicionário Inicializa Bloco na CPU	2	4096	6699	0.000450%
Dicionário Inicializa Bloco na GPU	190	4096	777645	0.052284%
Dic. Inic. Resto do dicionário na GPU	168	1	168	0.000011%
GPU Atualiza Rates Blocos	55548	1	55548	0.003735%
Atualiza Rates CPU	23	54320	1240324	0.083391%
GPU Atualiza Rates Blocos Escala	22360	54320	1214586875	81.660736%
Dicionário IncluiSubDic na CPU	526	48671	25618918	1.722445%
Dicionário IncluiSubDic na GPU	416	48671	20245029	1.361141%

Tabela 6.15: *Detalhamentos dos tempos de comunicação no envio do dicionário para a GPU*

6.1.4 Diferenças nos cálculos da GPU

Os resultados dos cálculos do custo Lagrangiano na CPU e GPU, para cada bloco de um dicionário, apresentam pequenas diferenças em valores em torno da quinta ou sexta casa decimal. Estas diferenças ocorrem em poucos casos, algo em torno de 0.13% dos blocos de um dicionário.

Estas diferenças ocorrem apesar dos dois compiladores afirmarem utilizar valores de ponto flutuante seguindo o padrão IEEE-754 [12] [15].

Quando o compilador *gcc* é utilizado em arquiteturas x86 ele transforma as variáveis de ponto flutuante (tanto as do tipo `float` quanto as do tipo `double`), para um formato de precisão estendida nativo dos coprocessadores x87 as convertendo novamente quando são guardadas novamente em memória [12]. Então, as diferenças das duas implementações levam a ocorrer diferenças de comportamento nas situações de arredondamento, *underflow* e *overflow*, pelo fato de, os valores intermediários no processo de calculo terem uma maior precisão relativa e maior valor de expoente quando calculados dentro do coprocessador x87.

6.2 Resultados implementação na CPU com o OpenMP

Para verificar o desempenho do MMP em sistemas multiprocessados, utilizou-se uma CPU Intel I7-860 que possui 4 núcleos cada um com a tecnologia *Hyperthreading* que faz com que osistema operacional perceba cada núcleo como dois. Utilizando o gprof, obtemos a informação de temporização na tabela 6.2:

Função	Tempo (s)	%
Bloco::DistanciaF(Bloco, float)	155.63	35.76
ArithmeticCoder::rate(int, int)	105.41	24.22
Bloco::Compara(Bloco*)	98.13	22.55
CodificadorMMPRD2D::MenorJ(Bloco, int, int)	40.68	9.35
Dicionario::IncluiSubDic(Bloco*, int, int)	22.46	5.16
ArithmeticCoder::add_new_char(int)	5.43	1.25
CodificadorMMPRD2D::ArvoreOtima(Bloco, int, int, char*, int, int)	2.35	0.54
Bloco::Escala(Bloco*)	1.49	0.34
Bloco::Dimensiona(int, int)	1.32	0.30
Bloco::Copia(Bloco*, int, int)	0.93	0.21
Bloco::Bloco()	0.77	0.18
Bloco::Distancia(Bloco)	0.15	0.03
Demais funções	0.41	0.09

Tabela 6.16: *Temporização do MMP-FP em uma CPU intel i7-860 @ 2.88GHz*

Observa-se na tabela 6.2, que que os métodos `Bloco::DistanciaF(Bloco, float)` e `ArithmeticCoder::rate(int, int)` juntos são responsáveis por 59.98 % do tempo gasto no processamento. A função `CodificadorMMPRD2D::MenorJ(Bloco, int, int)` contém todas as chamadas a estas funções. A maior parte das chamadas à a função `CodificadorMMPRD2D::MenorJ(Bloco, int, int)` está na função `CodificadorMMPRD2D::CalculaM A)`, cujo código é:

```
void CodificadorMMPRD2D :: CalculaMapa(Bloco A)
{
   int n,m,k,l;
   Custo J;
   Bloco B;

   for(n = 0; n < Dic.NumEscalasV; n++)
   {
      for(m = 0; m < Dic.NumEscalasH; m++)
      {
         B.Dimensiona(Dic.linhas[n][m], Dic.colunas[n][m]));
         for(k = 0; k < Nmax/Dic.linhas[n][m]; k++)
         {
            for(l = 0; l < Mmax/Dic.colunas[n][m]; l++)
            {
               B.Copia(&A, k*Dic.linhas[n][m], l*Dic.colunas[n][m]);
```

```
                Cotimo[n][m][k][l] = MenorJ(B, n, m);
            }
         }
         B.Dimensiona(0,0);
      }
   }
   delete [] B;
}
```

As chamadas à função `CodificadorMMPRD2D::MenorJ(Bloco, int, int)` podem ser efetuadas em paralelo, se substituirmos um dos comandos de iteração `for` por um comando de iteração `for paralelo`, usando a diretiva OpenMP `#pragma parallel for`. A nova função possui o seguinte código:

```
void CodificadorMMPRD2D :: CalculaMapaMP(Bloco A, int NumThreads)
{
   int n,m,k,l,p, n_thread;
   Custo J;
   Bloco *B;

   B = new Bloco [NumThreads];
   for(n = 0; n < Dic.NumEscalasV; n++)
   {
      for(m = 0; m < Dic.NumEscalasH; m++)
      {
         for(n_thread = 0; n_thread < NumThreads; n_thread++)
            B[n_thread].Dimensiona(Dic.linhas[n][m], Dic.colunas[n][m]);
         omp_set_num_threads(NumThreads);
         #pragma omp parallel for private(p,k,l,n_thread) schedule(dynamic)
         for(p = 0; p < (Nmax/Dic.linhas[n][m])*(Mmax/Dic.colunas[n][m]); p++)
         {
            k = p/(Mmax/Dic.colunas[n][m]);
            l = p%(Mmax/Dic.colunas[n][m]);
            n_thread = omp_get_thread_num();
            B[n_thread].Copia(&A, k*Dic.linhas[n][m], l*Dic.colunas[n][m]);
            Cotimo[n][m][k][l] = MenorJ(B[n_thread], n, m);
         }
         for(n_thread = 0; n_thread < NumThreads; n_thread++)
            B[n_thread].Dimensiona(0,0);
      }
   }
   delete [] B;
}
```

É importante observar que os dois comandos `for` mais internos foram substituídos por um único `for paralelo`, porque o OpenMP não iria, por *default*, implementar o

segundo for de modo paralelo. A variável `B` do tipo `Bloco` foi substituída por um conjunto de variáveis, uma para cada *thread*, para que não houvesse interferencia entre os diferentes *threads*. Para evitar interferência, as variáveis `p, k, l` e `n_thread` foram declaradas como `private`. Finalmente, foi escolhido um esquema de alocação de iterações por *thread* do tipo dinâmico (`schedule(dynamic)`) que neste caso funcionou de modo mais eficiente.

Apesar da rotina `CalculaMapa` ser responsável pela maior parte das chamadas à função `MenorJ`, esta é também chamada pela função `void CodificadorMMPRD2D :: SegmentaBloco(Bloco A, int escalaV, int escalaH, Bloco *AC)`. Por isso foi criada uma versão de execução paralela, `MenorJMP`, para substituir a rotina `MenorJ` original dentro da função `SegmentaBloco`. O código da função `MenorJ` é:

```
Custo CodificadorMMPRD2D :: MenorJ(Bloco A, int escalaV, int escalaH)
{
   //Escolhe o melhor elemento no dicionario para representar o bloco A, segundo um criterio R-D
   Custo Otimo;
   int indice;
   float J, LF;
   Otimo.Imin = 0;
   Otimo.Jmin = Dic.Elemento[escalaV][escalaH][0]->Distancia(A)+Lambda*(Cod.rate(0,2*(escalaV*ProfH+escalaH)+1));
   for(indice = 1; indice < Dic.TamSubDic[escalaV][escalaH]; indice++)
   {
       LF = Lambda*(Cod.rate(indice,2*(escalaV*ProfH+escalaH)+1));
       if(LF < Otimo.Jmin)
       {
           J = Dic.Elemento[escalaV][escalaH][indice]->DistanciaF(A, Otimo.Jmin-LF)+LF;
           if(J < Otimo.Jmin)
           {
               Otimo.Jmin = J;
               Otimo.Imin = indice;
           }
        }
   }
   return(Otimo);
}
```

As chamadas aos dois métodos, `Bloco :: DistanciaF(Bloco, float)` e `ArithmeticCoder :: rate(int, int)`, podem ser paralelizadas se transformarmos o comando de iteração `for` em um `for paralelo` utilizando a diretiva OpenMP `#pragma parallel for`. Contudo devemos ter cuidado, pois a variável `Otimo` não pode ser compartilhada por todos os *threads*, pois isso levaria a inconsistências na otimização. Uma solução possível é:

```
Custo CodificadorMMPRD2D :: MenorJMP(Bloco A, int escalaV, int escalaH, int NumThreads)
{
   // Escolhe o melhor elemento no dicionario para representar o bloco A, segundo um criterio R-D
   Custo Otimo;
   int indice, *Imin, n;
   float J, LF, *Jmin;
   Jmin = new float [NumThreads];
   Imin = new int [NumThreads];
   Imin[0] = 0;
   Jmin[0] = Dic.Elemento[escalaV][escalaH][0]->Distancia(A)+Lambda*(Cod.rate(0,2*(escalaV*ProfH+escalaH)+1));
   omp_set_num_threads(NumThreads);
   #pragma omp parallel for private(indice,LF,J,n)
   for(indice = 1; indice < Dic.TamSubDic[escalaV][escalaH]; indice++)
   {
      n = omp_get_thread_num();
      LF = Lambda*(Cod.rate(indice,2*(escalaV*ProfH+escalaH)+1));
      if(LF < Jmin[n])
      {
         J = Dic.Elemento[escalaV][escalaH][indice]->DistanciaF(A, Jmin[n]-LF)+LF;
         if(J < Jmin[n])
         {
            Jmin[n] = J;
            Imin[n] = indice;
         }
      }
   }
   Otimo.Imin = Imin[0];
   Otimo.Jmin = Jmin[0];
   for(n = 1; n < NumThreads; n++)
   {
      if(Jmin[n] < Otimo.Jmin)
      {
         Otimo.Jmin = Jmin[n];
         Otimo.Imin = Imin[n];
      }
   }
   delete [] Jmin;
   delete [] Imin;
   return(Otimo);
}
```

A terceira função a produzir uma elevada carga computacional é `Bloco::Compara(Bloco*)`. Todas as chamadas a esta função são feitas por `Dicionario::IncluiSubDic(Bloco*, int, int)`. Esta por sua vez é chamada na rotina:

`void CodificadorMMPRD2D::SegmentaBloco(Bloco A, int escalaV, int escalaH, Bloco *AC)`

E o trecho de código responsável por isso é:

```
AS = new Bloco;
for(n = 0; n < Dic.NumEscalasV; n++)
{
   for(m = 0; m < Dic.NumEscalasH; m++)
   {
      AS->Dimensiona(Dic.linhas[n][m],Dic.colunas[n][m]);
      AS->Escala(AC);
      if(Dic.IncluiSubDic(AS, n, m) == 0)
      {
         Cod.add_new_char(2*(n*ProfH+m)+1);
      }
      AS->Dimensiona(0,0);
   }
}
delete AS;
```

Neste caso, podemos usar `#pragma parallel for` para paralelizar o primeiro comando `for`, tomando o cuidado de manter uma variável `AS` do tipo `Bloco` independente para cada *thread*. O código torna-se então:

```
AS = new Bloco [NumThreads];
omp_set_num_threads(NumThreads);
#pragma omp parallel for private(n,m,n_thread)
for(n = 0; n < Dic.NumEscalasV; n++)
{
   for(m = 0; m < Dic.NumEscalasH; m++)
   {
      n_thread = omp_get_thread_num();
      AS[n_tread].Dimensiona(Dic.linhas[n][m], Dic.colunas[n][m]);
      AS[n_thread].Escala(AC);
      if(Dic.IncluiSubDic(&AS[n_thread], n, m) == 0)
      {
         Cod.add_new_char(2*(n*ProfH+m)+1);
      }
      AS[n_tread].Dimensiona(0,0);
    }
}
delete [] AS;
```

Ao implementar esta solução, as chamadas à função `Bloco::Escala(Bloco*)` também serão feitas em paralelo.

Podemos estimar a redução do tempo de processamento em função do grau de paralelismo do seguinte modo:

$$r = \frac{T'}{T} = \frac{\frac{T_p}{N} + T_s}{T} \tag{6.1}$$

onde T é o tempo total sem multiprocessamento, T' é o tempo total com multiprocessamento, T_p é o tempo da parte do programa que será paralelizada, T_s é o tempo da parte que irá rodar sequencialmente e N é o grau de paralelismo.

No caso de substituirmos as funções originais pelas suas equivalentes paralelas, e considerando-se os dados da tabela 6.2 bem como a equação 6.1, esperamos obter uma redução de:

$$r = \frac{\frac{429.23}{N} + 5.93}{435.16} = \frac{0.9864}{N} + 0.0136 \tag{6.2}$$

Para avaliar o desempenho da implementação proposta, realizamos testes em duas condições: No teste 1, a imagem Lena 512×512 foi comprimida com o MMP com parâmetro $\lambda = 30.0$, e tamanho de bloco 16×16, usando 1,2,3,4,5,6,7 e 8 *threads*. No teste 2, a configuração foi a mesma, exceto pelo parâmetro $\lambda = 7.5$.

A tabela 6.2 contém os resultados obtidos para 10 repetições do teste 1.

Tempo (segundos)							
1 thread	2 threads	3 threads	4 threads	5 threads	6 threads	7 threads	8 threads
561.98	322.57	228.36	189.33	165.73	158.60	153.37	152.55
563.04	311.80	228.12	187.10	164.85	157.51	153.24	150.04
563.32	323.05	228.40	189.93	166.34	156.50	152.55	148.78
561.79	319.70	228.46	186.85	165.54	156.88	151.47	150.13
556.17	321.95	228.22	187.10	165.18	157.22	153.11	149.21
580.02	319.64	231.44	188.32	166.56	157.78	152.42	149.61
562.06	315.65	228.50	186.39	165.25	156.34	152.46	149.09
562.79	318.49	228.20	186.81	164.94	156.06	151.87	149.09
587.40	321.79	228.84	189.10	166.58	158.24	153.59	151.82
562.13	315.25	228.48	189.83	165.93	157.84	154.55	152.27
Tempo médio (segundos)							
566.07	318.99	228.71	188.08	165.69	157.3	152.87	150.26
Desvio padrão (segundos)							
9.67	3.71	0.98	1.38	0.65	0.84	0.9	1.42

Tabela 6.17: Tempo de execução para o MMP-2D-FT com blocos 16×16 e $\lambda = 30$ com a imagem Lena.

Tempo (segundos)							
1 thread	2 threads	3 threads	4 threads	5 threads	6 threads	7 threads	8 threads
1443.98	820.87	586.54	506.82	408.19	392.66	384.55	375.24
1454.26	841.24	595.16	504.74	407.99	392.84	384.85	376.00
1466.84	835.91	593.48	502.25	406.56	392.12	386.15	376.33
1385.75	840.94	584.35	506.17	408.75	392.12	382.90	375.60
1384.09	827.14	585.08	506.30	406.83	393.26	382.55	379.00
1384.43	811.52	586.11	506.22	408.92	393.10	384.44	378.31
1450.04	827.71	585.47	499.56	412.96	393.26	383.96	376.59
1389.17	811.47	586.57	497.82	406.49	393.42	386.24	380.67
1343.74	815.25	583.87	506.21	410.47	393.13	382.81	375.37
1385.91	813.61	585.10	504.02	410.76	398.56	387.89	382.46
Tempo médio (segundos)							
1408.83	824.57	587.18	504.01	408.8	393.45	384.64	377.56
Desvio padrão (segundos)							
41.17	11.8	3.89	3.15	2.09	1.85	1.72	2.48

Tabela 6.18: Tempo de execução para o MMP-2D-FT com blocos 16×16 e $\lambda = 7.5$ com a imagem Lena.

A tabela 6.2 contém os resultados obtidos para 10 repetições do teste 2.

A figura 6.10 ilustra a redução teórica, conforme dada pela equação 6.1, e os dados experimentais obtidos com uma CPU Intel i7-860 @2.88GHz rodando o programa MMP-FP comprimindo a imagem Lena de dimensão 512×512 usando blocos 16×16, para $\lambda = 30$ (teste 1) e para $\lambda = 7.5$ (teste 2). O desempenho de compressão foi independente do número de *threads*, como deveria ser.

A diferença observada entre a redução teórica e a obtida possui diversas causas, entre elas:

1. Para permitir a paralelização de um trecho de código são introduzidas etapas adicionais de processamento inexistentes no caso de execução sequencial. Por isso a eficiência nunca chegará a 100%.

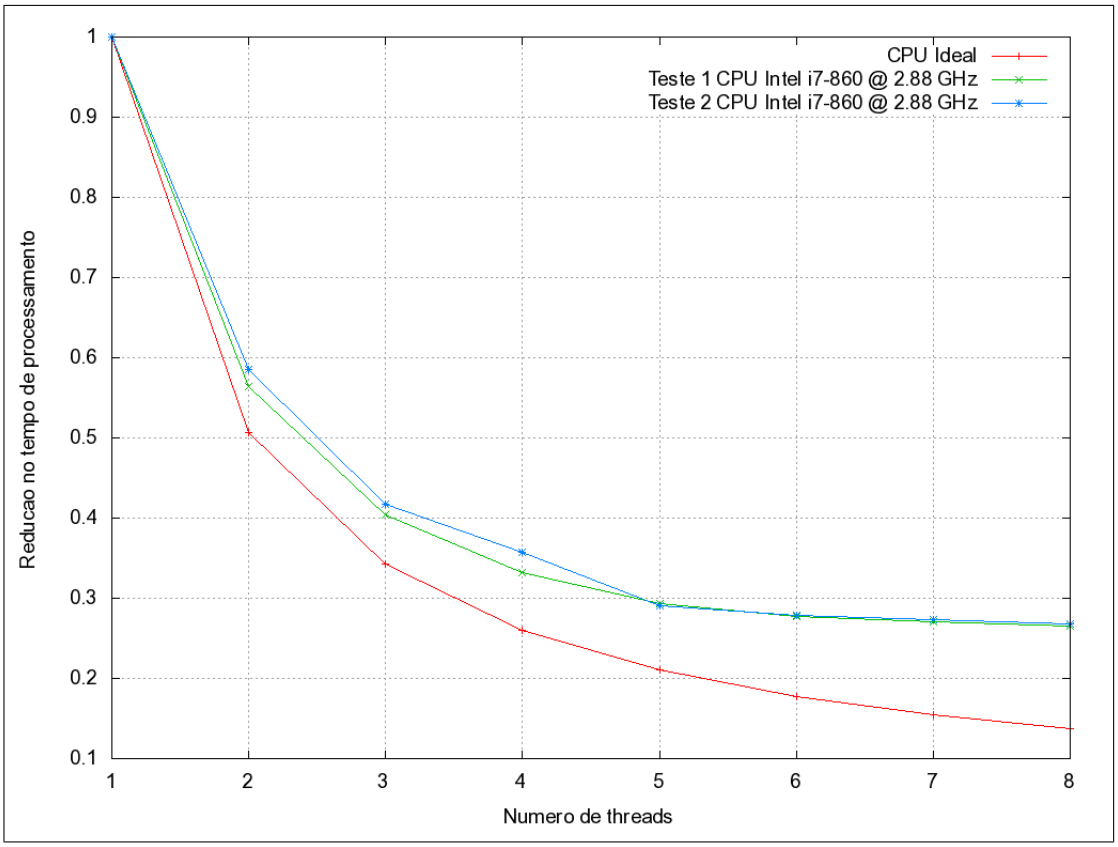

Figura 6.10: *Redução no tempo de processamento com uso de paralelismo*

2. A CPU Intel i7-860 possui uma característica chamada *Turbo Boost* que aumenta a frequência do relógio quando a carga computacional é baixa. Isso mascara a redução inicial obtida ao passarmos de 1 para dois *threads* pois o processador utiliza um relógio mais lento quando operando com 2 *threads*.

3. A CPU Intel i7-860 possui quatro núcleos, cada um com tecnologia *Hyperthreading*. Assim ao passar de 4 *threads*, ainda obtemos aumento de desempenho porém as vezes a uma taxa menor do que a que seria obtida se tivéssemos 8 núcleos.

Capítulo 7

Conclusão

A utilização de GPUs para a aceleração de aplicações cientificas e de uso corrente é bastante promissora. Mas, no estágio de desenvolvimento atual das mesma, aplicações onde existe intensa e fragmentada comunicação entre as memórias da CPU e GPU, classe onde o método MMP se encaixa, existe uma considerável degradação do desempenho. Como vimos no desenvolvimento desta tese a degradação de desempenho causada pela comunicação entre memórias anulou ganhos de aceleração obtidos na nossa adaptação do MMP.

A primeira vista, poderíamos sugerir que o MMP fosse completamente portado para a GPU buscando terminar com a comunicação entre memórias. Mas, existem muitas partes do MMP que não são obviamente paralelizáveis e executariam como apenas uma *thread* mais lenta que a CPU.

Acreditamos que, nas próximas gerações de GPUs estes problemas estejam sanados e possam se obtidos grandes ganhos no desempenho do MMP.

Quanto ao uso do paralelismo em CPUs multi-núcleos, o OpenMP mostrou ser uma solução simples e bastante eficaz para melhorar o desempenho do MMP, chegando-se a ganhos de 3.7 vezes no tempo de processamento com CPUs Intel I7.

Capítulo 8

Sugestões para trabalhos futuros

Com base no trabalho desenvolvido, observamos que é possível paralelizar uma parte considerável do processamento do MMP. Portanto, o mesmo tem grande potêncial de explorar as características das novas gerações de computadores, onde o aumento capacidade paralelismo suportado pelas CPUs e GPUs será cada vez mais relevante como caminho de evolução das arquiteturas.

Partindo desta constatação, existem alguns frentes de pesquisa que tem um bom potencial de obtenção de resultados relevantes.

A conversão do trabalho atual para a execução otimizada nas novas GPUs da Nvídia, onde os recursos de paralelização são maiores, com menos restrições no acesso a memória por parte dos *threads* e com suporte à utilização da memória da CPU como memória de trabalho da GPU.

Do início da implementação desta tese até seu fechamento ocorreu uma sedimentação da linguagem OpenCL [41] no mercado. Além disso, a mesma já é suportada nos SDKs das GPUs Nvídia e ATI. Como esta linguagem se propõem a escalonar os processos de forma otimizada em computadores que possuam CPUs e GPUs com capacidade *multi-thread*, independente suas arquiteturas, é interessante portarmos o código do MMP-Cuda para a OpenCl para o mesmo se tornar independente de arquitetura de CPU e GPU.

Outra vertente interessante, é continuar explorando o potencial do OpenMP em CPUs multinúcleo e verificar o desempenho em Cluster.

Referências Bibliográficas

[1] M. B. de Carvalho, E. B. Silva, E. B., W. A. Finamore, *Multidimensional Signals Compression Using Recurrent Patterns*, In: Elsevier, pp. 1559-1580, 1995.

[2] M. B. de Carvalho *Compressão de Sinais Multi-dimensionais usando Recorrência de Padrões Multiescalas*, Coppe/UFRJ, D.Sc., Engenharia Elétrica , 2001.

[3] Nelson C. Francisco, Nuno M. M. Rodrigues, Eduardo A. B. da Silva, Murilo B. de Carvalho, Sérgio M. M. de Faria, Vitor M. M. da Silva, Manuel J. C. S. Reis, *Multiscale Recurrent Pattern Image Coding with a Flexible Partition Scheme*, Coppe/UFRJ, ICIP, 2008

[4] N. M. M. Rodrigues, E. A. B. da Silva, M. B. de Carvalho, S. M. M. de Faria, and Vitor M. M. Silva, *Universal image coding using multiscale recurrent patterns and prediction*IEEE International Conference on Image Processing, Genova, Italy, September 2005.

[5] Joint Video Team (JVT) of ISO/IEC MPEG & ITU-T VCEG (ISO/IEC JTC1/SC29/WG11 and ITU-T SG16 Q.6), *Draft of Version 4 of H.264/AVC (ITU-T Recommendation H.264 and ISO/IEC 14496-10 (MPEG-4 part 10) Advanced Video Coding)*,March 2005.

[6] N. M. M. Rodrigues, E. A. B. da Silva, M. B. de Carvalho, S. M. M. de Faria, Vitor M. M. Silva, and Frederico Pinagé, *Ef?cient dictionary design for multiscale recurrent patterns image coding*,Island of Kos, Greece, May 2006.

[7] Wikipedia, the free encyclopedia, *Multi-core processor*,http://en.wikipedia.org/wiki/Multi-core_processor, Acessado em 04 de junho de 2010.

[8] William Gropp, Ewing Lusk., *Why Are PVM and MPI So Different?*, Lecture Notes in Computer Science, No. 1332. pp 3-10.

[9] Parallel Virtual Machine Site, *Parallel Virtual Machine (PVM) Version 3.*,http://www.csm.ornl.gov/pvm/pvm_home.html, Acessado em 16 de junho de 2010.

[10] The OpenMP Architecture Review Board, *The OpenMP API specification for parallel programming.*, http://openmp.org/wp/about-openmp/, Acessado em 16 de junho de 2010.

[11] Duarte, Maria Heveline Vieira *Codificação de Imagens Estéreo Usando Recorrência de Padrões*, Coppe/UFRJ, D.Sc., Engenharia Elétrica , 2002.

[12] Gough, Brian J., Stallman, Richard M. *An Introduction to GCC - for the GNU compilers gcc and g++.*, Network Theory Ltd., 2004.

[13] The GNU Project. *GCC, the GNU Compiler Collection.* http://gcc.gnu.org/, Acessado em 01 de abril de 2010

[14] Tanenbaum, Andrew. *Modern Operating Systems 3rd Edition.*, Prentice Hall Press.,Upper Saddle River, NJ, USA, 2007.

[15] NVIDIA Corporation. *Nvidia Cuda^{TM} C Programming Best Practices Guide Cuda Toolkit Version 2.3.*, July 2009

[16] Harris, Mark., *Optimizing Parallel Reduction in CUDA.*, NVIDIA Developer Technology.

[17] Podlozhnyuk, Victor. *Histogram calculation in Cuda.*, Nvidia, 2007.

[18] Shams, Ramtin; Barnes, Nick . *Speeding up Mutual Information Computation Using Nvidia Cuda Hardware.*,Canberra, Australia . The Australian National University. 2007.

[19] Cuda SDK. *Cuda GPU Occupancy Calculator.* CUDA_Occupancy_calculator.xls.,http://developer.download.nvidia.com/compute/cuda/CUD
Disponível também no Cuda SDK

[20] SciGPU.org - Accelerating the Pace of Science With Graphics Hardware . *Data structures in the RTS.* http://scigpu.org/content/data-structures-rts, Acessado em 26 de fevereiro de 2010

[21] R. Clark. *Cuda Zone.* http://www.nvidia.com/object/cuda_home_new.html, Acessado em 31 de março de 2010

[22] A. Murat Tekalp, *Digital Video Processing.*, Prentice Hall, 1995.

[23] Linde, Y., Buzo, A. and Gray, R. M., *An Algorithm for Vector Quantization Design*, IEEE Tr. Commun., v.28, pp. 84-95, 1980.

[24] F. Halsall *Multimedia Communications: Applications, Networks, Protocols, and Standards.*, Addison-Wesley Publishing, 2000.

[25] Mitra, Sanjit K. *Digital Signal Processing a Computer Based Approach*, McGraw-Hill International Editions, 1998

[26] Mello, Ricardo Noé Bretin de *Estudo comparativo da transformada Karhunen-Loève na compressão de imagens*, Universidade Federal do Rio Grande do Sul, M.Sc., Engenharia Elétrica , 2003.

[27] M. W. Marcellin, M. J. Gormish, A. Bilgin, and M. P. Boliek. *An overview of JPEG-2000.* in Proceedings of 2000 Data Compression Conference, pp 523 –544, March 2000

[28] R. Clark. *An Introduction to JPEG2000 and Watermarking.* http://www.jpeg.org/public/jpgintro.pdf, Acessado em 18 de março de 2010

[29] Haykin, Simon,Woods. *Communication Systems 4th edition* United States. John Wiley & Sons, Inc. 2001.

[30] International Telecommunication Union. *ITU Recommendation T.81 JPEG Compression JPEG*, 1993.

[31] Gonzales, Rafael C.,Woods, Richard E. *Digital Image Processing 3rd ed.* United States. Pearson Prentice Hall. 2008.

[32] Sayood, Khalid. *Introduction to data compression 3rd ed.* United States. Morgan Kaufmann. 2006.

[33] David, Salomon. *Data compression - The Complete Reference 4rd ed.* London. Springer-Verlag. 2007.

[34] Victor Moya , Carlos Gonzalez , Jordi Roca , Agustin Fernandez , Roger Espasa. *Shader Performance Analysis on a Modern GPU Architecture.* 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05), Novembro 2005.

[35] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Tim Purcell. *A Survey of General-Purpose Computationon Graphics Hardware.* Computer Graphics Forum, 26(1):80–113, Março 2007.

[36] Shreine, Dave; Woo, Mason; Neider, Jackie; Davis, Tom. *OpenGL programming Guide 2.0. 5 ed.* Estados Unidos. Addison-Wesley.2005.

[37] Computergram *IBM RS6000 ANNOUNCEMENTS Published:10-June-1994*, http://www.cbronline.com/news/ibm_rs6000_announcements_3 Estados Unidos. Acessado em 04 de fevereiro de 2010.

[38] Fosner, Ron. *Real-Time Shader Programming.* San Francisco. Kaufmann Publishers. 2003.

[39] NVIDIA Corporation. *Nvidia Cuda^{TM} Programming Guide Version 2.3.*, August 2009

[40] *OpenGL ES - The Standard for Embedded Accelerated 3D Graphics*, https://www.khronos.org/opengles/2_X/, Acessado em 28 de janeiro de 2010.

[41] *OpenCL - The open standard for parallel programming of heterogeneous systems*, www.khronos.org/opencl/, Acessado em 04 de fevereiro de 2010.

[42] *History of OpenGL*, http://www.opengl.org/wiki/History_of_OpenGL, Acessado em 27 de janeiro de 2010.

[43] Tanenbaum, Andrew S. *Structured Computer Organization (5th Edition)*, Prentice-Hall, Inc., 2005.

[44] Joint Video Team (JVT) of ISO/IEC MPEG & ITU-T VCEG (ISO/IEC JTC1/SC29/WG11 and ITU-T SG16 Q.6) *Draft of Version 4 of H.264/AVC (ITU-T Recommendation H.264 and ISO/IEC 14496-10 (MPEG-4 part 10) Advanced Video Coding)*, March 2005

[45] I. H. Witten, R. M. Neal and J. G. Cleary *Arithmetic Coding for Data Compression*, Communications of the ACM, Vol. 30, No. 6, pp. 520-540, June 1987.

[46] Waldir S. S. Junior *Compressão de Imagens Utilizando Recorrência de Padrões Multiescalas com Segmentação Flexível*, Rio de Janeiro 2004 XIII,145 pp 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia Elétrica, 2004)

[47] Eddie B. L. Filho, Murilo B. de Carvalho e Eduardo A. B. da Silva *Compressão de Imagens Utilizando Recorrência de Padrões Multiescalas com Critério de Continuidade Inter-blocos*, XXI Simpósio Brasileiro de Telecomunicações - SBT'04, 2004.

[48] N. M. M. Rodrigues, E. A. B. da Silva, M. B. de Carvalho, S. M. M. de Faria, and Vitor M. M. Silva *Universal image coding using multiscale recurrent patterns and prediction*, IEEE International Conference on Image Processing, September 2005.

[49] Li-Yi Wei and Marc Levoy, *Fast texture synthesis using treestructured vector quantization*, Proceeding of SIG-GRAPH 2000, pp. 479–488.