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

Érica Souza da Costa

Implementação em GPU do algoritmo de compressão de imagens baseado em recorrência de padrões multiescala, o MMP

Niterói-RJ
2013

ÉRICA SOUZA DA COSTA

IMPLEMENTAÇÃO EM GPU DO ALGORITMO DE COMPRESSÃO DE IMAGENS BASEADO EM RECORRÊNCIA DE PADRÕES MULTIESCALA, O MMP

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: Processamento Digital de sinais.

Orientador: Prof. MURILO BRESCIANI DE CARVALHO

Niterói-RJ

2013

ÉRICA SOUZA DA COSTA

IMPLEMENTAÇÃO EM GPU DO ALGORITMO DE COMPRESSÃO DE IMAGENS BASEADO EM RECORRÊNCIA DE PADRÕES MULTIESCALA, O MMP

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: Processamento Digital de sinais.

Aprovada em MARÇO de 2013.

BANCA EXAMINADORA

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

Prof. EDUARDO ANTONIO BARROS DA SILVA, Ph.D
UFRJ

Prof. TADEU NAGASHIMA FERREIRA, D.Sc
UFF

Niterói-RJ
2013

Ao meu amado marido e aos meus pais que me ensinaram a perseverar.

Agradecimentos

Agradeço ao meu marido Eduardo por estar ao meu lado nos dias mais difíceis.

Agradeço ao meu orientador Murilo por todo o tempo dedicado e sua incansável paciência.

Agradeço aos meus pais que me ensinaram a essência para ser quem eu sou hoje.

Agradeço aos meus amigos que compreenderam as vezes em que não pude estar presente.

Lista de Figuras

Lista de Tabelas

Sumário

Resumo

Este trabalho propõe a implementação de um algoritmo de compressão de dados baseado em recorrência de padrões Multiescalas (MMP) em um ambiente de processamento heterogêneo contendo múltiplos elementos, especificamente CPU multinúcleo e GPU. A proposta visa obter redução no tempo de processamento quando comparado ao obtido por implementações convencionais. Os resultados mostram que a exploração da capacidade de paralelismo tanto na GPU quanto na CPU geram ganhos significativos e relevantes, sendo os melhores resultados obtidos com a utilização da GPU.

Palavras-chave: GPU, paralelismo, MMP, Cuda, processamento multicore.

Abstract

This work proposes an implementation of a data compression algorithm based on recurrent multiscales patterns (MMP) in a heterogeneous environment containing multiple elements, specifically multicore CPU and GPU. The proposal is intended to achieve reduction in processing time as compared to that obtained by conventional implementations. The results show that the utilization of the parallelism capacity of both GPU and CPU leads to significant improvements. The best results were obtained with the use of the GPU.

Keywords: GPU, MMP, paralelism, Cuda, multiCore processing.

Capítulo 1

Introdução

Vem ocorrendo nas últimas décadas uma evolução tecnológica que nos levou à era da Informação. Possuímos nesse panorama voz, dados e imagens trafegando em diversos meios e com os computadores pessoais, a internet, a necessidade de fácil e rápido acesso às informações, cada vez mais queremos que uma maior quantidade de informações transitem pelos meios de transmissão existentes.

Nesse cenário, é possível identificar a importância de que os sinais digitais sejam representados e armazenados de forma eficiente. Uma das soluções empregadas é a compressão dos dados, que permite reduzir a utilização de recursos de armazenamento e economizar banda de transmissão. A técnica visa representar a informação utilizando menos bits que o original.

Podemos citar as aplicações geoespaciais que gerenciam e compartilham grandes quantidades de imagens e que utilizam a compressão para tornar viável o armazenamento dessas informações. A compressão tem um papel importante na medicina devido ao grande volume de dados gerados pelas imagens médicas como ressonância magnética, tomografia computadorizada e ultrassonografia. Imagens são fundamentais no diagnóstico de doenças e planejamento cirúrgico e o seu uso só é possível com os métodos de compressão. Sem a compressão de dados, os atuais meios de transmissão não suportariam a crescente evolução da demanda na Internet. Por isso, a compressão de dados vem sendo alvo de ativa pesquisa.

Por outro lado, tornou-se comum, nos computadores pessoais, uma demanda por placas gráficas cada vez mais eficientes e com essa necessidade os processadores gráficos se tornaram cada vez melhores. A GPU (Graphics Processing Unit), que é a unidade de

processamento presente nas placas de vídeo, se tornou objeto de pesquisa e passível de ser programada e utilizada como unidade de processamento para diversos fins.

Em [1] foi proposto um algoritmo de compressão de imagens baseado em recorrência de padrões mutiescalas, o MMP. Ele possui um bom desempenho quando comparado aos outros métodos de compressão de imagens e vídeos, porém sua execução é demorada e custosa devido ao elevado consumo de recursos computacionais.

Nesse trabalho serão estudadas formas de adaptar o algoritmo MMP de modo a explorar eficientemente a arquitetura de computação paralela disponível nas GPUs.

Essa dissertação está organizada da seguinte forma:

No Capítulo 2 serão apresentadas a definição do problema de compressão de dados tais como tipos de compressão, soluções clássicas e medidas de informação. No Capítulo 3, explica-se o funcionamento e vantagens da utilização do paralelismo na GPU e são apresentados também dados sobre a arquitetura envolvida nessas unidades de processamento.

No Capítulo 4 explica-se de forma resumida e prática o funcionamento do MMP e apresenta-se partes de uma implementação básica em C++.

No Capítulo 5 é feita uma análise dos tempos de execução do código serial de modo a permitir identificar as etapas críticas do ponto de vista do tempo de execução, bem como a forma de implementação na GPU. No Capítulo 6 apresentamos os testes realizados e os resultados obtidos. No Capítulo 7 apresentamos as conclusões e propostas para evolução futura da pesquisa.

Capítulo 2

O problema da compressão

O objetivo da compressão é reduzir o número de bits utilizados para armazenar e transmitir dados. Seu estudo está inserido no ramo da Teoria da informação [5], o campo de estudo criado por Shannon. Neste campo, estudam-se medidas quantitativas da informação, as características das fontes bem como limites de desempenho para compressão e transmissão das mesmas. De modo geral, a compressão de dados é feita por um codificador, que mapeia o sinal emitido pela fonte em uma string binária que será então descomprimida por um decodificador.

Nas seções seguintes iremos apresentar alguns conceitos importantes.

2.1 Tipos de Compressão

A compressão de dados pode ser com ou sem perdas. Ambos os casos estão ilustrados em esquemáticos nas figuras 2.1 e 2.2. Na compressão com perdas, admite-se que parte da informação emitida por uma fonte possa ser perdida em troca de uma compressão maior. Essa perda faz com que o sinal obtido na saída do decodificador seja diferente daquele aplicado à entrada do codificador, o que acarreta uma distorção e demonstra-se que quanto maior o número de bits gasto na representação menor será a distorção e vice-versa. Os dados multimídia (áudio, vídeo), por exemplo, podem tolerar um certo nível de degradação sem que o objetivo, que neste caso é a interpretação por um observador humano, seja perdido.

Já na compressão sem perdas, garante-se que após o processo de compressão e descompressão, a informação seja uma cópia exata do original. Este é o tipo de compressão

usada para armazenar registros em que a perda de um único bit poderia representar um desastre, como por exemplo dados bancários.

Figura 2.1: Esquemático - Compressão sem perdas

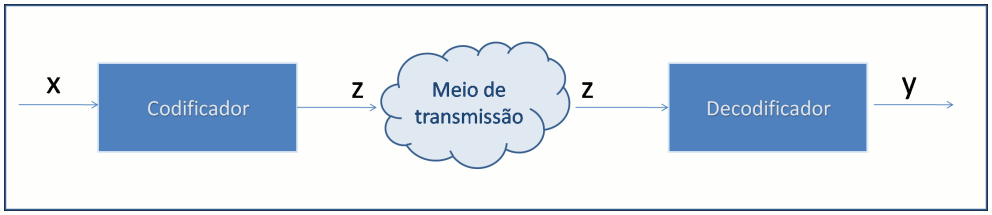


Figura 2.2: Esquemático - Compressão com perdas

2.2 Compressão sem perdas

O nosso objetivo é encontrar uma forma de representar um conjunto de dados de uma forma compacta. A título de exemplo, vamos considerar uma fonte discreta que emita símbolos de acordo com alfabeto abaixo representado.

$$X \in \{a, b, c, d\} \tag{2.1}$$

A nossa necessidade é codificar uma mensagem como $\bar{X} = \texttt{aabaabacabad}$ utilizando o código 1, representado na figura 2.3.

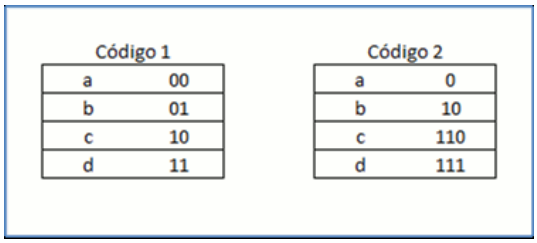


Figura 2.3: Exemplos de códigos

Para este caso, fazendo uma análise caractere a caractere, a string binártia de saída seria conforme a abaixo representada:

$$C_1 = 000001000001001000010011 \tag{2.2}$$

Se considerarmos o código 2, a informação na saída do codificador seria um pouco diferente:

$$C_2 = 0010001001100100111 \tag{2.3}$$

Observamos que a primeira codificação gerou uma sequência de 24 bits, enquanto a segunda uma de 19 bits. A mesma informação poderia ser codificada e transmitida de diversas formas diferentes, porém para que a mensagem original seja recuperada exatamente, o mapa que compõe o código deve ser inversível. É o caso dos dois códigos acima pois cada símbolo da fonte é mapeado em uma string binária diferente.

Além disso, o código deve ser autopontuável, o que significa que devemos ser capazes de separar os códigos após serem concatenados sem nenhuma ambiguidade. Nos dois exemplos isto é possível, pois no caso do primeiro código os símbolos estão agrupados sempre em grupos de 2 bits, enquanto no segundo caso, apesar do comprimento das palavras código ser variável, a escolha foi feita de modo que uma palavra nunca é prefixo de outra. Isso garante que é possível separar as palavras códigos sem a necessidade de nenhuma informação adicional, basta que o decodificador conheça a regra de associação.

Como os desempenhos de códigos diferentes pode ser diferente, é natural perguntarmos se existe um código que possui desempenho ótimo, no sentido de permitir a representação com menor número de bits. Esta questão é estudada na teoria da informação e demonstra-se que o menor número médio de bits necessário para representar cada símbolo emitido pela fonte é dado por:

$$H(X) = -\sum_j p_j \log_2(p_j) \tag{2.4}$$

onde p_j é a probabilidade da fonte emitir o j-ésimo símbolo. Esta medida é chamada de Entropia e demonstra-se que nenhum código de compressão de dados pode usar menos do que $H(X)$ bits por símbolo para representar exatamente os símbolos emitidos por uma fonte X.

A Equação 2.4 pode ser interpretada como sendo a média de $i_j = -\log_2(p_j)$ que é a informação própria do j-ésimo símbolo. Se calcularmos o comprimento médio, medido em bits por símbolo, obtido quando usamos um código particular C cujas palavras código possuem comprimentos dados por l_j, obtemos $\hat{L} = \sum_j p_j l_j$. Comparando-se este resultado com a equação 2.4 pode-se concluir que um código de compressão de dados sem perdas ótimo utiliza $l_j = -\log_2(p_j)$ bits para codificar o j-ésimo símbolo.

Se fôssemos julgar qual dos dois códigos do exemplo seria o melhor deles, o escolhido seria aquele que consegue utilizar a menor quantidade de bits para representar a informação.

Em um processo de compressão, obtém-se benefício da redundância da informação, e para que essa redundância seja identificada é necessário que seja estabelecido um modelo probabilístico para a fonte.

É importante enfatizar que o modelo que define a probabilidade de cada símbolo é determinante para que a compressão seja bem sucedida. O modelo precisa prever corretamente as ocorrências e a um símbolo que ocorra muito, associar uma probabilidade, o que corresponde a ter pouca informação e necessitar de poucos bits para ser codificado.

Esse tipo de compressão é dita como estatística, pois se beneficia das probabilidades de ocorrência de caracteres únicos ou grupos de caracteres, de maneira que códigos menores possam ser usados para representar caracteres (ou grupos de caracteres) frequentemente usados, ao passo que códigos maiores são usados para representar caracteres encontrados com menos frequência.

2.2.1 Soluções clássicas

2.2.1.1 Codificação de Huffman

A codificação de Huffman é um exemplo de implementação de compressão sem perdas e estabelece códigos de tamanhos variáveis e inteiros a partir de uma tabela de probabilidade. A ideia básica está em codificar os símbolos com maior frequência com menos bits e os símbolos com menor frequência com mais bits.

Para que essa atribuição seja feita, constrói-se uma árvore binária baseada nas probabilidades de ocorrência de cada símbolo. Nesta árvore as folhas representam os símbolos presentes nos dados, associados com suas respectivas probabilidades de ocorrência. Os nós intermediários representam a soma das probabilidades de ocorência de todos os

símbolos presentes em suas ramificações e a raiz representa a soma da probabilidade de todos os símbolos do conjunto de dados.

Para exemplificar seu funcionamento, vamos considerar uma sequência de caracteres como sendo $\bar{Y} = AAAAAABBBBBCCCCDDDEEF$. No primeiro passo se determina as frequências de cada símbolo que pode ser observada na tabela 2.4.

Símbolo	Ocorrência
A	6
B	5
C	4
D	3
E	2
F	1

Figura 2.4: Frequência Símbolos - Exemplo Huffman

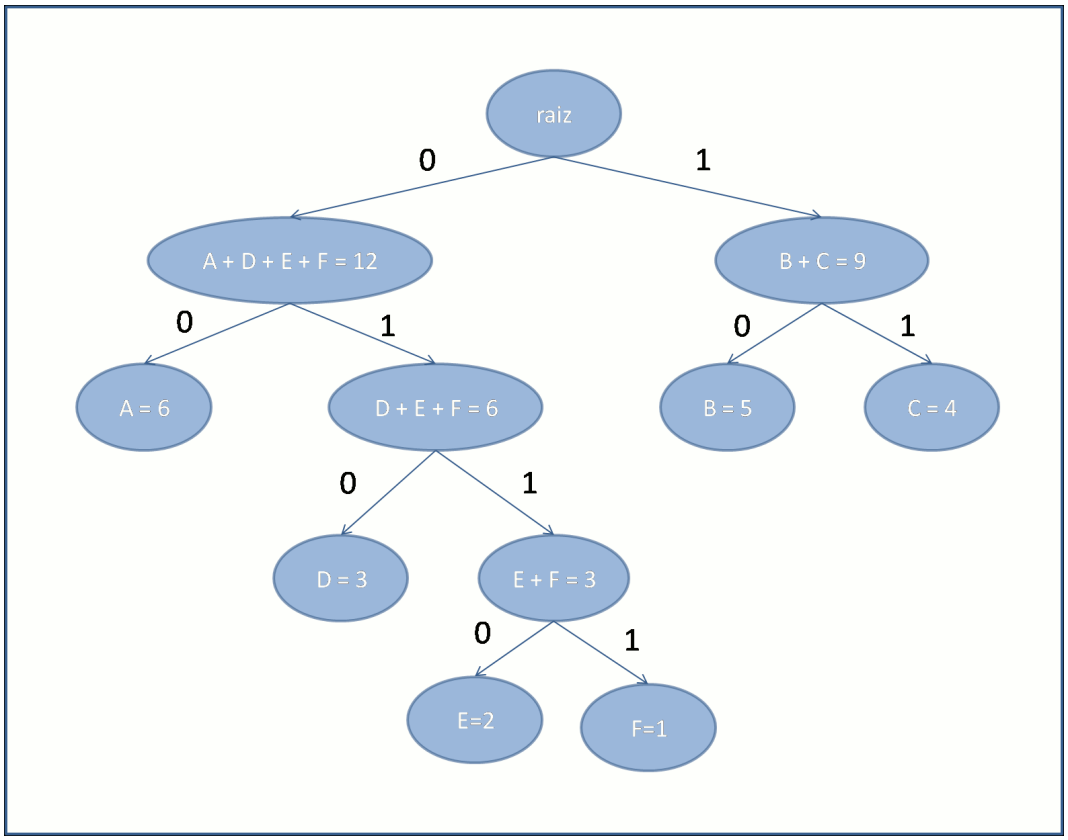


Figura 2.5: Árvore Binária de Huffman

A partir dos símbolos com menor ocorrência se inicia a montagem da árvore, que nesse caso são E e F. Eles são então unidos por um nó intermediário que possui a soma de

suas ocorrência e o passo seguinte é identificar os dois menores valores novamente, porém o nó intermediário que foi criado participa da escolha. Assim é feito até que seja montada a árvore binária conforme está representada na figura 2.5.

A partir da árvore, são determinados os códigos para cada símbolo, pois para cada aresta é atribuído o valor de 0 ou 1 e os códigos são gerais a partir do percursso.

Símbolo	Código
A	00
B	10
C	11
D	010
E	0110
F	0111

Tabela 2.1: Tabela de códigos gerados - Huffman

Na tabela 2.1 estão representados os códigos para cada um dos símbolos de $\bar{Y}$ e a sua codificação seria:

$$C_{huffman} = 000000000000101010101011111111010010010011001100111 \tag{2.5}$$

Apesar de bastante difundida, a codificação de Huffman não é considerada ótima, pois somente utiliza números inteiros de bits para os códigos, ou seja, para uma informação própria de 2.5 bits, dada por $-\log_2(p_i)$, o codificador de Huffman utilizaria 3 bits para codificar o símbolo. Porém podemos dizer que entre os métodos que utilizam códigos fixos e inteiros, a codificação de Huffman possui a melhor aproximação.

2.2.1.2 Codificador Aritmético

Para superar essa deficiência de códigos fixos e inteiros, surgiu posteriormente o codificador aritmético. Ele não produz um único código para cada símbolo, mas sim um código para a mensagem inteira. Cada símbolo adicionado à mensagem altera o código de saída e com ele é possível que um caractere com 2.3 bits de informação própria seja codificado com exatamente 2.3 bits. Isso ocorre pelo fato de não ser baseado em tabela de símbolos. Desta forma ele elimina a associação entre símbolos individuais e códigos e assim é capaz de produzir uma saída codificada com taxa muito próxima da entropia da fonte.

No Codificador Aritmético, a partir de um modelo estatístico, constrói-se uma tabela onde são listadas as probabilidades do próximo símbolo lido ser cada um dos possíveis símbolos. Primeiramente parte-se do intervalo $[0, 1]$ e nele identifica-se o sub-intervalo ao qual corresponde o primeiro símbolo lido. Para cada símbolo subsequente, subdivide-se o intervalo atual em sub-intervalos proporcionais às probabilidades da tabela de intervalos e encontra-se novamente o intervalo que corresponde ao próximo símbolo. Ele consiste em representar a probabilidade de ocorrência de cada caractere de acordo com intervalos cumulativos. No final do processo, é gerado um intervalo que corresponde à probabilidade da ocorrência de todos os símbolos na ordem correta.

A mensagem codificada é valor que esteja contido no intervalo [0,1[e possa ser representado com o menor número possível de dígitos.

Vamos considerar que se quer codificar a mensagem $\bar{Z} = \texttt{acbaab}$ e as probabilidades de ocorrência desses símbolos estejam especificadas na tabela 2.2.

Símbolo	Probabilidade
a	0,4
b	0,35
c	0,25

Tabela 2.2: Probabilidade de Ocorrência - Codificador Aritmético

A figura 2.6 representa a análise realizada e para esse caso e o código deve ser um valor incluído no intervalo $[0.34224, 0.3442[$.

2.2.1.3 Lempel-Ziv-Welch

Tanto a codificação de Huffman quanto o codificador aritmético são métodos que tratam os símbolos individualmente, já o LZW é um método de compressão baseado em dicionário. Um outro diferencial é que ele não necessita conhecer as probabilidades dos símbolos antes de iniciar a compressão. Por vezes este conhecimento é difícil senão impossível de obter, além do que a fonte da informação pode ter uma característica dinâmica e suas estatísticas variarem ao longo do tempo.

Existem duas principais versões dos algoritmos Lempel-Ziv, LZ77 e LZ78, e estes foram criados por Abraham Lempel e Jacob Ziv nos anos de 1977 e 1978. Esses algoritmos se baseiam na construção de um dicionário em tempo de execução com os símbolos encontrados nos dados a serem comprimidos. No início, (LZ77 e LZ78), o dicionário é

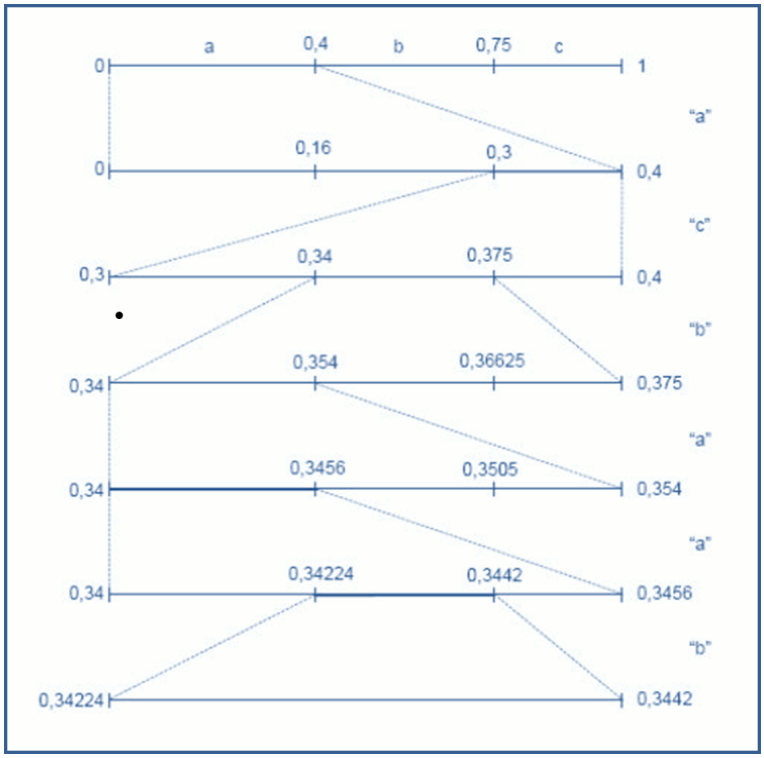


Figura 2.6: Análise Codificador Aritmético

vazio e à medida que o arquivo é lido os símbolos e sequências de símbolos, ainda não registrados, são inseridos no dicionário. No LZ78, sempre se transmite um índice para uma string do dicionário seguido de um símbolo de inovação ASCII. Por exemplo, para codificar "banana", teríamos inicialmente $dic = \lambda$, que representa a string vazia. O algoritmo então prossegue transmitindo 0 (representado por tx0) que é o índice do elemento λ no dicionário, seguido do código ASCII para 'b'. Em seguida o algoritmo evolui como:

1) $dic = \{\lambda, b\}$, tx 0 seguido de "a"

2) $dic = \{\lambda, b, a\}$, tx 0 seguido de "n"

3) $dic = \{\lambda, b, a, n\}$, tx 2 (que é o índice no dicionário para a string "a") seguido de "n"

4) $dic = \{\lambda, b, a, n, an\}$, tx 2

Uma modificação do LZ78 deu origem ao LZW. Nessa versão, o dicionário se inicia com todos os símbolos do alfabeto da fonte. Isso ocorre devido ao fato de agora ser transmitido somente o índice. Assim, o dicionário contém o alfabeto básico e todas as sequências codificadas podem ser representadas somente com índices. Por exemplo a tabela ASCII para a compressão de textos.

A ideia está em inicialmente definir uma variável *String* , que está vazia, e *c* que

Palavra a ser codificada: banana

String	c	String+c	Existe no Dicionário?	Incluir no Dicionário	Código enviado	
	b	b	Sim	-	-	
b	a	ba	Não	256 - ba	98	98 é o índice referente ao símbolo **b** 256 é o próximo índice disponível do dicionário ASCII que foi inicializado de 0 a 255.
a	n	an	Não	257 - an	97	97 é o índice referente ao símbolo **a**
n	a	na	Não	258 - na	110	110 é o índice referente ao símbolo **n**
a	n	an	Sim	-	-	
an	a	ana	Não	259 - ana	257	257 é o índice referente ao símbolo **an**
a		a	Sim		97	97 é o índice referente ao símbolo **a**

Figura 2.7: Exemplo codificação LZW da palavra *banana*

representa o próximo símbolo a ser codificado. A cada interação verifica-se se a concatenação de *String* + *c* existe no dicionário. Caso exista, *String* passa ser *String* = *String* + *c* e *c* é renovado com o próximo símbolo a ser codificado. Caso *String* + *c* não exista, envia-se o índice de *String* no Dicionário, atualiza-se o dicionário com *String* + *c* e *String* passa a ser *c*. Abaixo na figura 2.7 podemos verificar um exemplo ilustrativo dessa codificação para a palavra *banana*.

2.3 Compressão com perdas

Na compressão com perdas, a mensagem recuperada após o decodificador não é exatamente igual à mensagem inicial. Essa representação pode ser observada na figura 2.2. Conforme vimos no caso da compressão sem perdas, a medida básica de informação é a Entropia. Esta medida não pode ser utilizada no caso com perdas. Neste caso, a medida básica de quantidade de informação é determinada pela função R-D, *rate-distortion*, que mede a quantidade de bits média mínima por símbolo emitido pela fonte para obter-se uma representação com distorção igual a *D[referência]*. Uma propriedade desta função é que ela é estritamente decrescente com D e é convexa no intervalo [0, Dmáx]. A figura 2.8 ilustra a curva que representa a função RD de uma fonte Gaussiana, cuja Equação é dada por :

$$R(D) = \frac{1}{2}\log_2\left(\frac{\sigma^2}{D}\right) \tag{2.6}$$

onde σ^2 é a variância da Gaussiana.

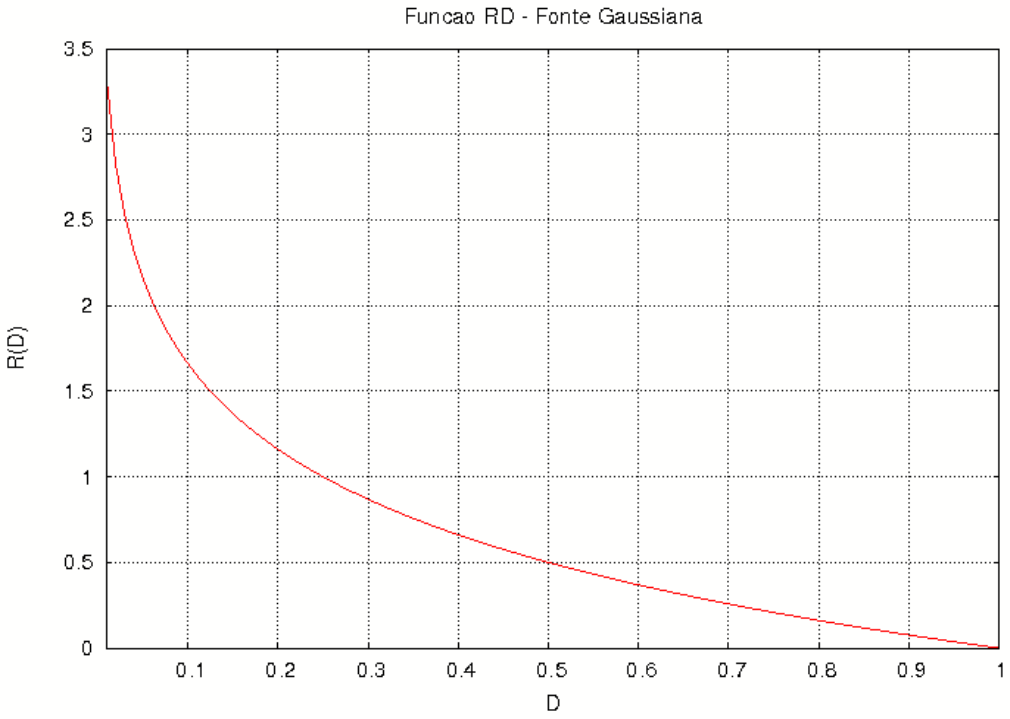


Figura 2.8: Função RD para uma fonte Gaussiana de variância unitária

2.3.1 Implementação em três estágios

Estudando algumas soluções de compressão com perdas, identificamos que muitas delas fazem uso de três etapas no processo: Transformação, Quantização e codificação. Essas etapas são sequenciais e estão representadas na figura 2.9.

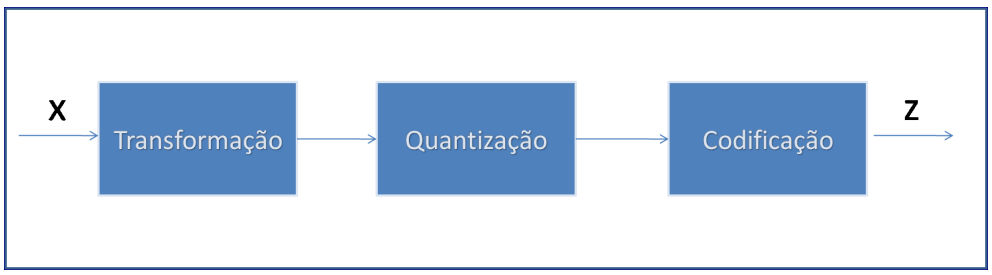


Figura 2.9: Esquemático - Compressão em três estágios

A primeira delas, a Transformação, é utilizada para facilitar a implementação da etapa seguinte. O propósito da mesma é tentar melhorar o desempenho da Quantização, bem como modelar características perceptuais na medida de distorção. Por exemplo, o

compressor de imagens JPEG utiliza a DCT (Discrete Cosine Transform) em blocos de 8x8 pixels na etapa de transformação. Para imagens naturais foi observado que os coeficientes gerados após a aplicação da DCT são aproximadamente descorrelatados, o que melhora o desempenho dos quantizadores escalares utilizados na etapa de quantização deste padrão. Além disso cada um dos 64 coeficientes de um bloco transformado é quantizado por um quantizador diferente. Deste modo é possível controlar separadamente a quantidade de distorção introduzida em cada componente espectral, o que permite um certo grau de controle perceptual (argumenta- se que a distorção seria mais visível em algumas componentes do que em outras).

A Quantização consiste em associar a informação da entrada a um universo finito de valores. É nessa etapa que é adicionada a distorção, pois a informação passa a ser uma aproximação do original. Essa quantização pode ser Vetorial ou Escalar. Na Quantização Escalar, cada símbolo é associado separadamente a um valor disponível no quantizador, já na Vetorial, associamos grupos de símbolos a um vetor disponível. Em outras palavras, considerando que tenhamos diversos vetores de dimensão k e um universo para associação Y, finito, com N vetores possíveis, o objetivo da quantização é mapear esses vetores (dimensão k) a um dos vetores possíveis do universo Y. Em geral a Quantização Vetorial possui um desempenho superior ao da Quantização escalar quanto à taxa-distorção, porém a complexidade computacional é muito maior. Contudo, se temos na entrada amostras descorrelatadas, essa diferença de desempenho diminui.

Por exemplo, no JPEG utiliza-se um conjunto de quantizadores escalares uniformes com passo diferente para cada componente do bloco transformado. Os passos são arrumados em matrizes 8x8 assim como os blocos transformados. Essas matrizes são referidas como matrizes de quantização. A norma JPEG sugere 2 matrizes diferentes, uma para codificar luminância e outra para crominância [12].

A terceira e última etapa é a codificação, que seria a compressão sem perdas aplicada à saída do Quantizador, onde são utilizados os algoritmos descritos anteriormente: Huffman, Codificador Aritmético e LZW. O codificador JPEG usa o código de Huffman nesta etapa.

Neste capítulo foram apresentados alguns conceitos importantes da compressão de dados. No capítulo que se segue, serão abordados conceitos do processamento paralelo que auxiliaram na implementação proposta nesse estudo.

Capítulo 3

O processamento paralelo

Observando-se a evolução dos sistemas de computação ao longo dos anos, pode-se perceber um grande esforço no sentido de aumentar o desempenho, tanto do ponto de vista de velocidade de processamento quanto de diminuição de consumo e custo.

Pode-se notar que durante uns 30 anos o aumento da velocidade de processamento dos dispositivos era alcançada a partir do aumento da frequência do clock da CPU (Central Processing Unit), porém os fabricantes foram obrigados a procurar alternativas para conseguirem um poder computacional ainda maior, pois se depararam com restrições de calor e limites físicos para que continuassem avançando nessa linha.

A partir de 2005, os principais fabricantes de microprocessadores começaram a oferecer processadores com dois núcleos e na sequência três, quatro, seis e oito.

Se pensarmos que hoje em dia o incomum é encontrar um computador, seja ele pessoal ou pra fins profissionais, com processador de apenas um núcleo e que o habitual são os processadores multicores, conseguimos começar a entender a importância do processamento paralelo.

Juntamente com a evolução da CPU, observamos também que as aplicações começaram a demandar cada vez mais dos gráficos 3D, o que impulsionou uma melhoria nos processadores gráficos em geral. Essa evolução, mais tarde, veio acompanhada da possibilidade de programação direta a essas unidades, o que possibilitou a utilização dos hardwares, não somente para as necessidades gráficas, mas também para computação. A GPU (Graphics Processing Unit), que é unidade de processamento presente nas placas de vídeo, se tornou instrumento útil a diversas áreas por possuir elevado desempenho devido ao seu uso maciço de paralelismo. Podemos dizer que a computação está evoluindo para

um processamento conjuto de CPU e GPU.

3.1 Amdalah's Law

Para a computação paralela, temos um modelo conhecido como *Amdalah's Law* que relaciona o aumento de velocidade esperada entre a versão paralelizada e a versão executada em série. Esse modelo é utilizado para prever o aumento de velocidade máxima teórica possível com a utilização de múltiplos processadores e a fórmula que define a máxima aceleração é dada por:

$$S(N) = \frac{1}{(1 - P) + \frac{P}{N}} \tag{3.1}$$

Onde $S(N)$ é a aceleração máxima utilizando N processadores, P é o percentual de tempo de execução do código que deveria ser paralelizado e $(1 - P)$ o percentual do tempo do código em série. Esse conceito será aplicado em um simples código na seção a seguir.

3.2 Arquitetura Cuda

O CUDA é uma arquitetura desenvolvida para as placas de vídeo da Nvidia que permite que a GPU possa ser usada tanto para computação em aplicações de uso geral quanto para as necessidades gráficas tradicionais. A linguagem é baseada em C e o que foi feito foi adicionar algumas palavras chaves para o CUDA e então chamamos de CUDA C. Quando escrevemos um código para que esse seja executado na placa de vídeo, a esse trecho chamamos de *kernel*.

3.2.1 Blocos e *Threads*

O primeiro conceito importante a ser entendido é o de bloco, pois é nele que está baseada a divisão do problema a ser tratado. Na arquitetura Cuda, para cada um deles, é feita uma cópia do *kernel* e dessa forma permite-se que as instruções associadas a cada bloco sejam executadas em paralelo. Se executarmos um *kernel* com apenas um bloco, significa que apenas uma instância será executada, porém se temos três blocos, podemos dizer que o mesmo código está sendo executado por três entidades diferentes e

ao mesmo tempo. Os blocos são agrupados em um grid. Na figura 3.1 podemos ver um grid bidimensional com 16 blocos, nesse caso, é um grid 4x4.

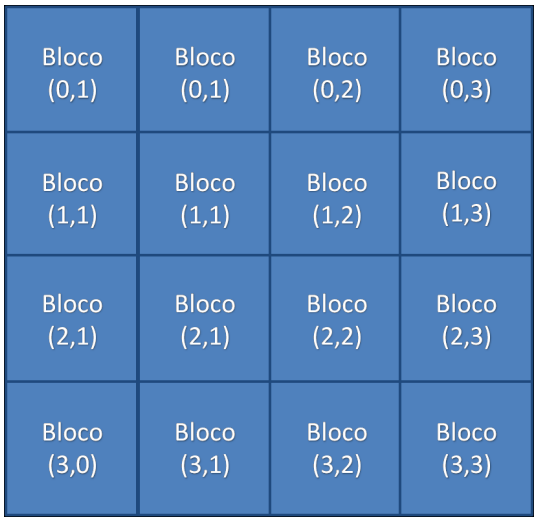


Figura 3.1: Exemplo de grid bidemensional 4x4.

Além dos blocos, temos também o conceito de *threads*. Um bloco pode conter um ou mais *threads* e os *threads* pertencentes ao mesmo bloco compartilham o *kernel* e também são executados paralelamente. A figura 3.2 ilustra uma organização de blocos e *threads*, onde o grid possui 16 locos e cada bloco 4 *threads*. Uma proposta como esta, permite que tenhamos 16 cópias do *kernel* sendo utilizadas e cada uma dessas cópias sendo executada por 4 *threads*.

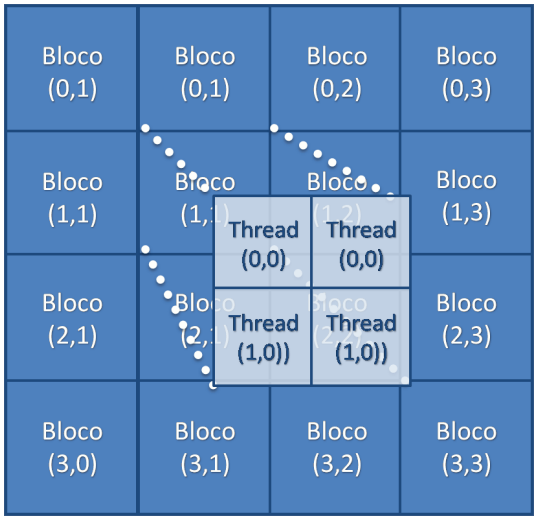


Figura 3.2: Exemplo de grid bidemensional 4x4 com cada bloco possuindo 2x2 threads

Os blocos e *threads* são identificados por índices que permitem a sua distinção

dentro do *kernel*, facilidade que auxilia no entendimento de que etapa do processo cada bloco e/ou thread estará executando.

Essas variáveis que identificam os blocos e *threads* são chamadas, respectivamente, `blockIdx` e `threadIdx`. Para o caso bidimensional, usa-se como forma de identificação as variações de x e y para ambos os casos. Na figura 3.1 estão representados esses valores no formato (`blockIdx.x, blockIdx.y`) para os blocos, assim como na figura 3.2, as referências de (`threadIdx.x, threadIdx.y`) para os *threads*. Para exemplificar, vejamos como um loop de um código de soma de vetores poderia ser escrito.

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

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

Se executarmos para apenas um bloco de dimensão 9x9, cada thread seria responsável pela inicialização de uma posição dos vetores e pela execução de um cáculo de soma. Teremos 81 *threads* executando ao mesmo tempo já com o loop anterior, seriam necessárias 81 iterações até o término do cálculo.

Com base na equação 3.1 e considerando que 80% do tempo de execução seja no loop e 20% nos demais trechos, ao paralelizarmos esse loop em 4 processadores, temos uma aceleração possível de:

$$S(4) = \frac{1}{(1-0,8) + \frac{0,8}{4}} \tag{3.2}$$

$$S(4) = 2,5 \tag{3.3}$$

Para 4 processadores, a aceleração máxima seria de 2,5 vezes. Agora vamos considerar que os tempos fossem invertidos, ou seja, o tempo do loop fosse apenas 20% do total. Assim teríamos:

$$S(4) = \frac{1}{(1-0,2) + \frac{0,2}{4}} \tag{3.4}$$

$$S(4) = 1,18 \tag{3.5}$$

O que nos faria concluir que a velocidade é limitada pelo tempo necessário para executar a parte serial do programa. Pode-se observar que para códigos não paralelizados onde a maior parte tempo de execução é consumida em loops e trechos que poderiam ser executados em paralelo, é possível obter-se um ganho relevante.

3.2.2 Tipos de memórias

Nessa arquitetura representada na figura 3.3 temos representadas quatro tipos de memórias: a memória global, a memória compartilhada, a memória constante e a memória de textura.

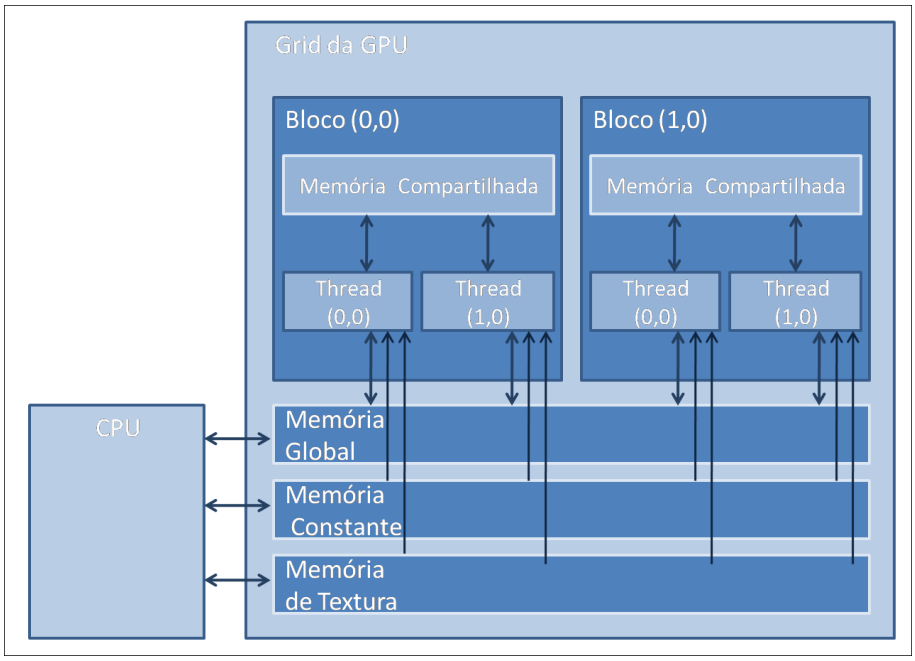


Figura 3.3: Tipos de memórias na arquitetura CUDA

A memória global é a memória da placa que é acessada por todos os blocos e pode ser escrita e lida por qualquer thread de qualquer bloco. É considerada uma memória de acesso lento, porém é a que possui maior dimensão dentre elas.

Já a memória compartilhada, distingue-se das demais porque o acesso é mais rápido do que a global e quando uma variável se encontra nessa área, o compilador trata essa variável de forma diferente. Ele faz uma cópia para cada bloco carregado na GPU e dessa forma todos os *threads* de um mesmo bloco tem acesso à informação, mas não há

visibilidade alguma entre os blocos. Essa é uma das vantagens da utilização dos *threads*, pois permitimos que os *threads* do mesmo bloco possam se comunicar entre si, além de o acesso ser mais veloz. Assim como na memória global, os *threads* podem ler e escrever na memória compratilhada.

Na memória constante os *threads* podem ler informação contida nela, mas não podem alterá-la. Além da leitura ser rápida, uma mesma leitura é aproveitada por mais de um thread. A memória constante é utilizada para dados que não sofrem alterações durante a execução do *kernel*.

Além dessas três, temos também a memória de textura que proporciona uma otimização para os casos em que o problema possa ser tratado em duas ou três dimensões e que as informações de vizinhança possuam alguma depêndencia. Por exemplo, aplicação de um filtro de média em que o valor de um pixel depende do valor dos pixel ao seu redor.

Neste capítulo foi possível entender alguns conceitos importantes utilizados na implementação de códigos a serem executados na GPU, assim como estruturas de sua arquitetura. No próximo capítulo será apresentado o algoritmo de compressão utilizado nessa dissertação: o MMP.

Capítulo 4

O MMP

O MMP [1] consiste em uma classe de algoritmos que tentam representar um fragmento de dados de entrada usando versões dilatadas ou contraídas de fragmentos previamente ocorridos. Nas seções seguintes encontraremos detalhes de seu funcionamento e funções implementadas no código do algoritmo.

4.1 Conceitos gerais

O MMP é um algoritmo que possui dicionários de diversas dimensões. Sua implementação é baseada no particionamento do dado de entrada e a criação a partir dele de uma árvore de segmentação que deverá ser codificada.

Vamos considerar um vetor $\bar{X}$ de entrada conforme a figura 4.1. Nela verificamos a representação de um vetor de dimensão 8x1.

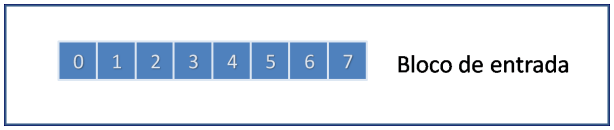


Figura 4.1: Exemplo de vetor de entrada de dimensão 8x1

Primeiramente, esse vetor de entrada será segmentado em dois vetores de dimensão 4x1 e assim sucessivamente até que se obtenham 8 vetores de dimensão 1x1. Essas etapas de segmentação estão representadas na figura 4.2.

Observamos também que cada etapa de segmentação gera um nível de profundidade na árvore e a cada um desses níveis chamaremos de escala. Para cada escala existe um

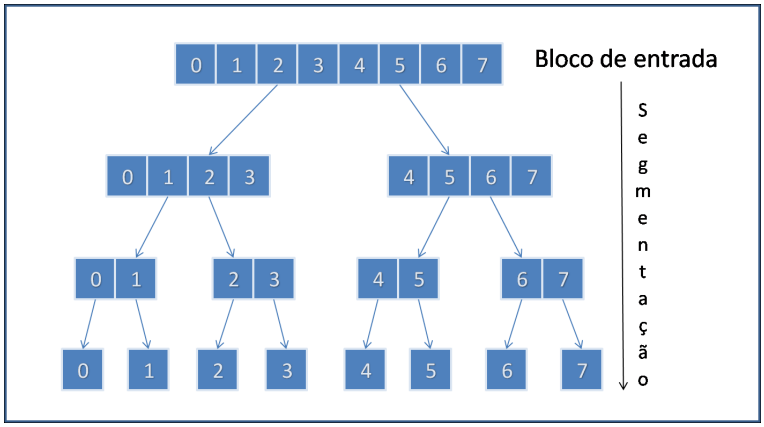


Figura 4.2: Segmentação do dado de entrada

dicionário correspondente e para esse caso, teríamos 4 dicionários nas dimensões de 8x1, 4x1, 2x1 e 1x1 que estão representados na figura 4.3.

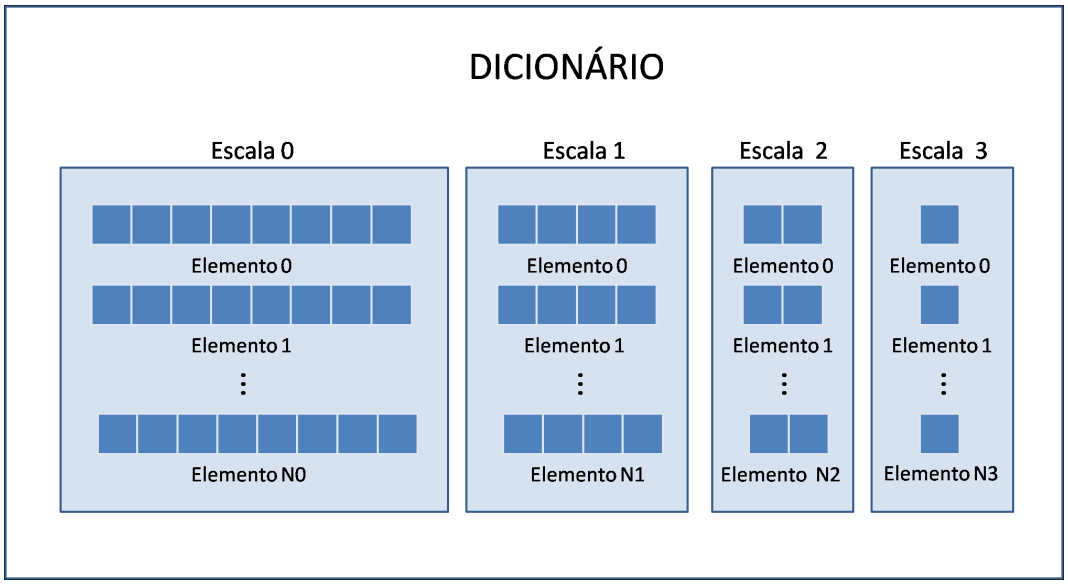


Figura 4.3: Exemplo de dicionário para dados de entrada 8x1.

Denominaremos fragmento o vetor de maior dimensão, ou seja o vetor na escala 0, e segmentos os vetores nas demais escalas. O termo elemento será utilizado para especificar os vetores que compõem os dicionários. Em trabalhos anteriores, usou-se o nome bloco no lugar do nome fragmento e subbloco ao invés de segmento. Neste trabalho a nomenclatura foi alterada ppara aumentar a clareza, uma vez que o grid de processamento utilizado na arquitetura Cuda é dividido em blocos e *threads* conforme visto no capítulo 3. Tanto para fragmento quanto para cada um dos segmentos resultantes da segmentação, é calculado o custo Lagrangeano:

$$J = D + \lambda R \tag{4.1}$$

O custo é calculado para cada elemento existente no dicionário da respectiva escala. O que significa dizer que para cada um dos dois segmentos existentes na figura 4.4 correspondentes a escala 1, serão calculados os custos Lagrangeanos para os elementos do dicionário de 0 a N1 dessa mesma escala representada na figura 4.3.

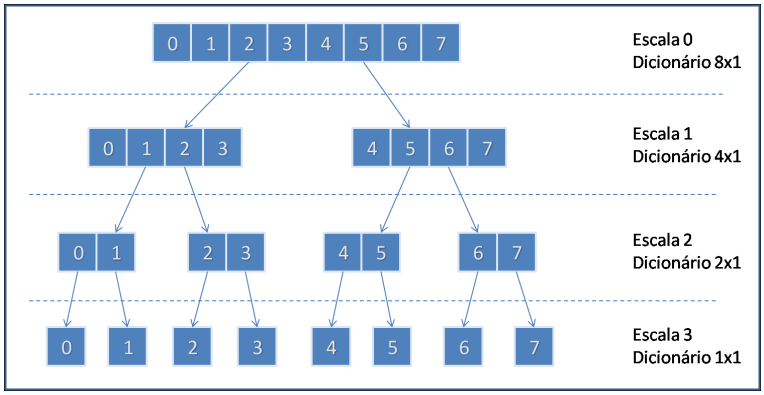


Figura 4.4: Segmentação com escalas determinadas.

Cada segmento da árvore será representado pelo elemento pertencente ao dicionário de escala correspondente que levar ao menor custo Lagrangeano. Na figura 4.5 temos um exemplo de árvore possível para $\bar{X}$. Nesse exemplo a codificação seria:

$$0, 1, i_0, 0, 0, 0, 1, i_1, 1, i_2, 1, i_3 \tag{4.2}$$

onde o flag 0 indica segmentação e o flag 1 indica que o segmento foi escolhido para ser transmitido. A leitura dos flags é feita da esquerda para a direita, conforme representado na figura 4.5, onde os segmentos que correspondem às folhas da árvore serão codificados. O índice i_k representa o índice do elemento do dicionário da escala correspondente, escolhido para representar o segmento original a ser codificado.

A árvore de segmentação é escolhida de forma a minimizar o custo Lagrangeano total que corresponde à soma dos custos de todos os nós folhas mais os custos de codificação dos flags.

A partir destes elementos escolhidos, os dicionários são atualizados com contrações e dilatações desses elementos. Por exemplo, para a figura 4.5 , após os índices i_1 e i_2 terem sido transmitidos, os elementos correspondentes podem ser concatenados e o resultado

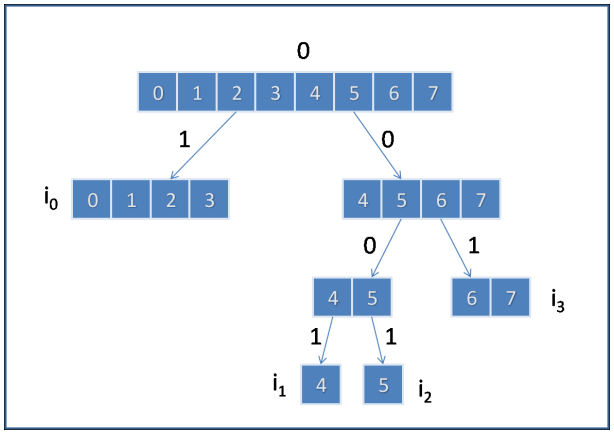


Figura 4.5: Exemplo de árvore de segmentação possível.

inserido no dicionário 2x1, ou seja, um novo elemento formado pela concatenação dos elementos de índices i_1 e i_2 do dicionário da escala 1 é criado. Em seguida este novo elemento é contraído e expandido para ser incluído nas demais escalas.

Esses conceitos apresentados para um vetor de entrada de uma dimensão podem ser expandidos para duas. E isso veremos na seção seguinte, quando analisamos as funções implementadas no MMP que servirão como base ao desenvolvimento proposto.

4.2 Análise das funções básicas

Nesta seção iremos estudar o funcionamento geral das funções implementadas pelo MMP. Abaixo estão listadas algumas funções utilizadas nesse algoritmo para a codificação.

```
1  for(n = 0; n < Ima.linhas; n = n + Nmax) {
2          for(m = 0; m < Ima.colunas; m = m + Mmax) {
3                  A = Ima.ExtraiBloco(n, m, Nmax, Mmax);
4                  CalculaMapa(A);
5                  ArvoreOtima(A, 0, 0, CodSeg, 0, 0);
6                  TamListaBlocos = 0;
7                  SegmentaBloco(A, 0, 0, &AC, 0, 0);
8                  AtualizaDic();
9                  A.Dimensiona(0, 0);
10         }
11 }
```

A primeira tarefa realizada é extrair da imagem original o fragmento a ser codificado. A função *ExtraiBloco* faz exatemente isso. Para ela passamos como parâmetro a

informação de qual trecho deve ser codificado, por exemplo, na primeira iteração, a variável A receberá os pixels de $(0,0)$ a $(16,16)$, considerando Nmax e Mmax de 16 pixels. Na figura 4.6 podemos observar o fragmento que estaria sendo analisado inicialmente.

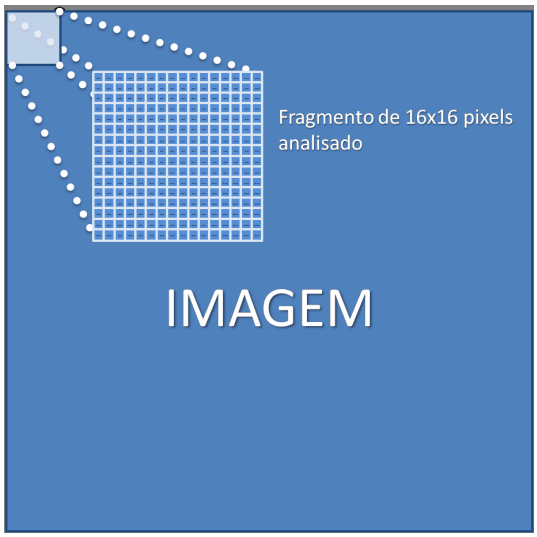


Figura 4.6: Exemplo de fragmento extraído pela função *ExtraiBloco* com dimensões de Nmax e Mmax de 16 pixels

A partir do fragmento selecionado, o próximo passo é escolher dentro do universo existente, qual elemento do dicionário possui a menor custo Lagrangeano. Como foi dito anteriormente, esse processo é feito para cada uma das escalas separadamente. Esse fragmento é dividido sucessivamente na horizontal e na vertical (processo de segmentação) e cada segmento obtido a partir desse processo é comparado aos elementos do dicionário da escala correspondente. Desta forma é possível escolher a melhor opção dentre os elementos já existentes e que possua o menor custo Lagrangeano. O processo é o mesmo explicado na seção anterior, porém estamos tratando de uma informação em duas dimensões.

Para um fragmento inicial de 16x16, temos um total de 25 escalas. A escala 0 é o próprio fragmento original, então o que fazemos é acessar o subdicionário que contém elementos de tamanho 16x16, calcular a distorção para todos os elementos do subdicionário e escolher o que possua o menor valor. Em seguida, dividimos o fragmento na direção horizontal gerando dois segmentos de tamanho 16x8, que também são comparados com o subdicionário da escala correspondente e para cada segmento um elemento é escolhido. Esse processo é repetido até que os segmentos em todas as possíveis escalas tenham sido analisados. Na figura 4.7 vemos uma representação de alguns desses segmentos em suas respectivas escalas e na figura 4.8 podemos ver as dimensões para todas elas.

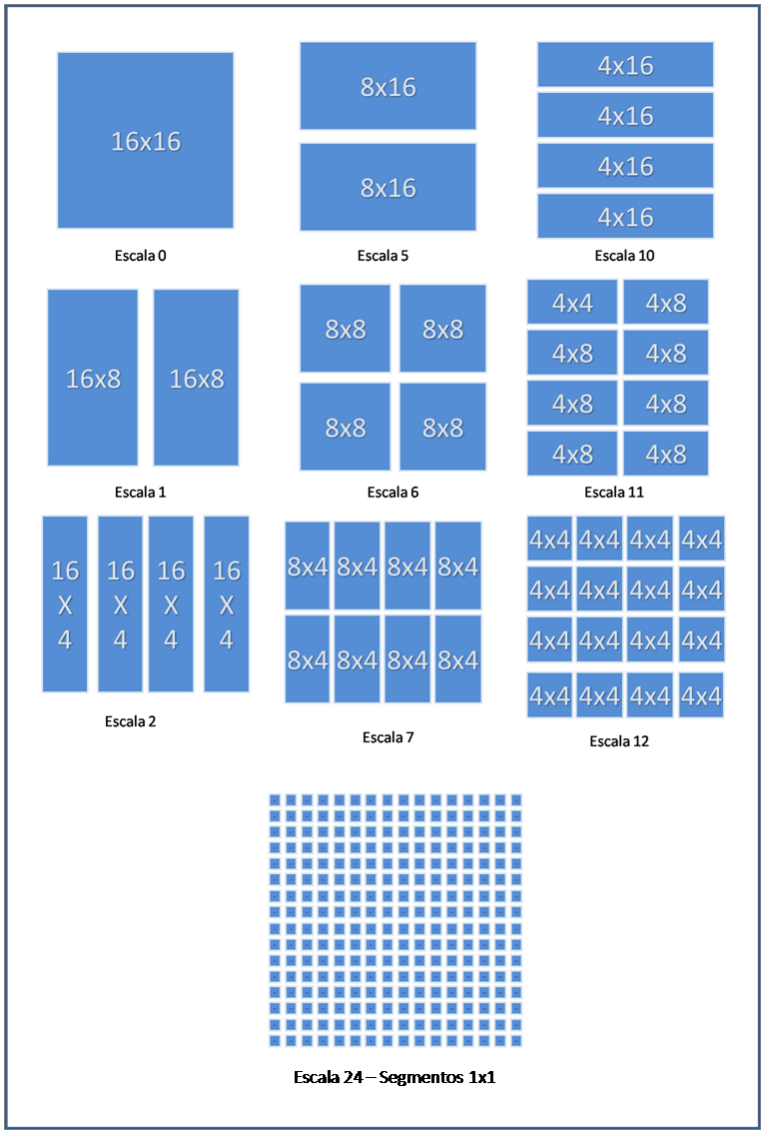


Figura 4.7: Exemplos de segmentos e suas escalas

No total, para um fragmento inicial de *16x16* temos 961 segmentos que precisam ser analisados em 25 escalas diferentes. O que a função *CalculaMapa* faz é retornar um vetor com 961 valores que representam o valor do menor custo encontrado para cada um dos segmentos. Além do valor do custo, também é necessário o índice do elemento do dicionário. O valor de J é calculado conforme especificado na equação 4.1.

Uma vez determinado qual elemento do dicionário é a melhor opção para cada segmento em cada uma das escalas, após a execução da função *CalculaMapa*, o MMP possui um árvore de valores de custo que vai desde o nó principal que é o fragmento 16x16 até as últimas folhas que são os segmentos 1x1. A partir desse momento, é preciso decidir qual a forma mais otimizada de transmitir essa informação. O que o algoritmo faz

Escala	Dimensão	Quantidade Segmentos	m	n
0	16x16	1	0	0
1	16x8	2	1	0
2	16x4	4	2	0
3	16x2	8	3	0
4	16x1	16	4	0
5	8x16	2	0	1
6	8x8	4	1	1
7	8x4	8	2	1
8	8x2	16	3	1
9	8x1	32	4	1
10	4x16	4	0	2
11	4x8	8	1	2
12	4x4	16	2	2
13	4x2	32	3	2
14	4x1	64	4	2
15	2x16	8	0	3
16	2x8	16	1	3
17	2x4	32	2	3
18	2x2	64	3	3
19	2x1	128	4	3
20	1x16	16	0	4
21	1x8	32	1	4
22	1x4	64	2	4
23	1x2	128	3	4
24	1x1	256	4	4

Figura 4.8: Informações em cada escala

é comparar se a soma dos custos dos segmentos filhos são inferiores ao valor do custo do segmento pai, para então tomar a decisão se é melhor transmitir o pai ou os dois filhos. Essa escolha da árvore final a ser transmitida é feita pela função *ArvoreOtima*.

O próximo passo é o aprendizado do algoritmo. A partir dos fragmentos codificados, são feitas concatenações para criação de novos elementos e esses são inseridos no dicionário de escala correspondente. O passo seguinte consiste em fazer as contrações e expansões em cada uma das escalas e verificar se os novos elementos gerados já existem no dicionário atual, já que não é permitido ter elementos repetidos. Esse teste é necessário porque dois vetores distintos ao serem transformados pela mudança de escala podem levar a um mesmo vetor na nova escala. Assim seria possível que, por exemplo, o resultado da contração para a escala 4 de um elemento recém criado na escala 2 já estivesse presente no dicionário da escala 4. Caso ele não exista, o mesmo é inserido no seu respectivo subdicionário e assim é realizado o aprendizado.

Terminado esse ciclo, um novo fragmento é selecionado e o processo recomeça, até que toda a informação seja codificada. Maiores informações e detalhes do MMP podem

ser encontradas em [1].

Capítulo 5

A paralelização do MMP usando CUDA

Considerando os conceitos apresentados nos capítulos anteriores, o propósito foi reescrever parte do algoritmo para que os trechos de execução mais demorada na CPU pudessem ser executados na GPU e assim tentar obter um ganho na velocidade de processamento considerando o paralelismo permitido pela mesma.

Deve-se ressaltar que a nomenclatura utilizada nas funções em C++ usadas para implementação do MMP segue a convenção de denominar bloco o fragmento a ser codificado e de denominar sub-blocos os segmentos derivados do fragmento. Isso corresponde à nomenclatura tradicional usada no MMP e foi tomada a decisão estratégica de não alterá-la para aumentar a confiabilidade do processo de desenvolvinento do software, uma vez que a versão que utiliza a GPU compartilha diversas funções usadas na versão que utiliza a CPU somente.

5.1 Implementação da função *CalculaMapa*

Inicialmente foi feita uma análise das funções do MMP utilizando-se o comando gprof do Linux, cujas primeiras linhas do resultado encontram-se na figura 5.1.

Considerando o resultado encontrado, primeiramente decidiu-se reescrever a função *CalculaMapa* para que esta fosse executada pela GPU, pois conforme dito no capítulo 4, ela é responsável por calcular o menor custo para cada um dos segmentos e as funções *DistanciaF*, *ArithmeticCoder.rate* e *MenorJ* fazem parte desse processo. Essas funções

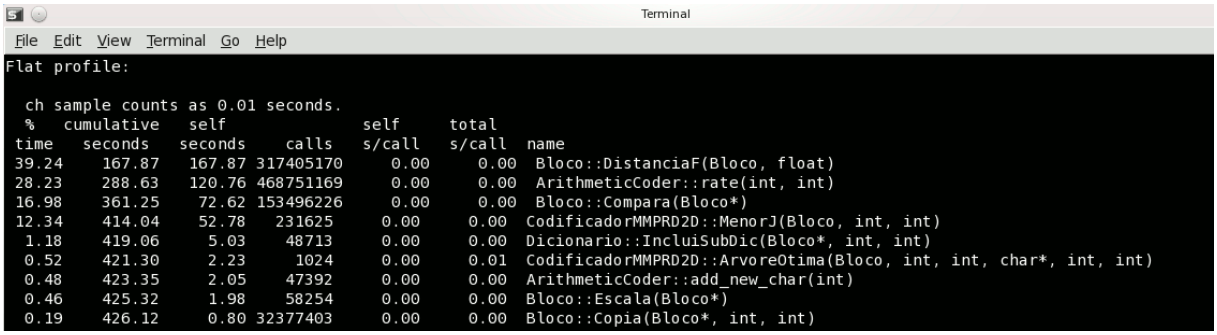


Figura 5.1: Saída do comando gprof

aparecem como consumidoras de 39,24%, 28,23% e 12,34% do tempo de execução respectivamente.

A seguir serão apresentadas as considerações realizadas para a implementação da função que chamamos de *CalculaMapaGPU*, ou seja, uma versão da função *CalculaMapa* que será executada na GPU.

5.1.1 Especificações da *CalculaMapaGPU*

5.1.1.1 *kernel* ***CalculaMapa***

Baseado na arquitetura Cuda, foi montado primeiramente um grid bidimensional de 128x128 blocos. Cada bloco do grid foi especificado para identificar um elemento do dicionário, ou seja, todos os *threads* do mesmo bloco estariam associados a um mesmo índice de elemento. Essa dimensão seria o suficiente para representar um dicionário de 16.384 elementos em cada escala.

Cada bloco foi definido como sendo bidimensional de 31x31 *threads* totalizando 961. A figura 5.2 representa essa distribuição de *threads* no bloco, onde os valores para as coordenadas variam de 0 a 30 em ambas as direções. Conforme dito no capítulo 4 esse valor de 961 é o total de segmentos gerados a partir dos particionamentos horizontais e verticais do fragmento a ser codificado originalmente de 16x16 pixels. Assim, cada *thread* ficou responsável por um segmento do fragmento a ser codificado e a dimensão do segmento determina a qual escala ele pertence e consequentemente a qual dicionário correspondente ele deve acessar.

Podemos observar na figura 5.3 a organização dos blocos do grid e os valores de identificação dos blocos que variam de 0 a 127 para as duas coordenadas. E para cada

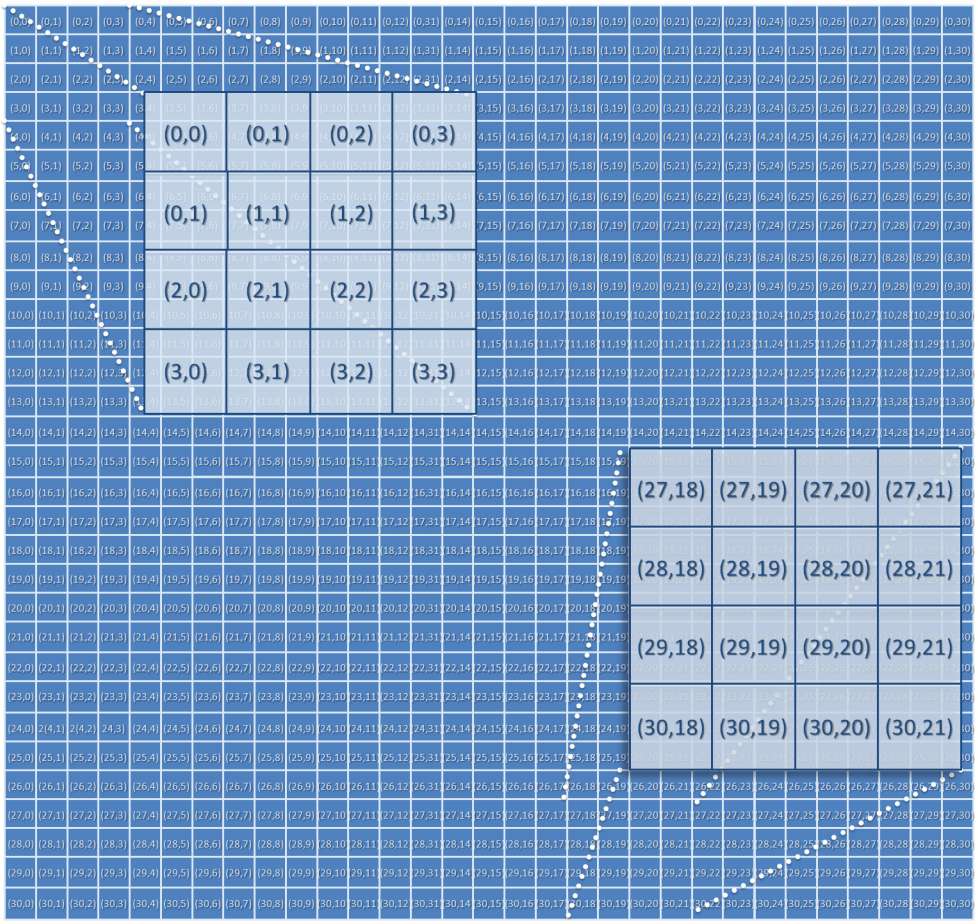


Figura 5.2: Distribuição dos *threads* no bloco

um desses blocos temos a configuração de *threads* já apresentada na figura 5.2.

No código da função *CalculaMapaGPU*, temos o trecho abaixo que especificamos como os *threads* e blocos estão configurados:

```
1 //Cada thread eh responsavel por um J
2 idThread= threadIdx.x + threadIdx.y*blockDim.x;
3
4 //Cada bloco eh responsavel por um elemento do dicionario
5 elementoDic= blockIdx.x + blockIdx.y*gridDim.x;
```

As associações dos blocos aos elementos dos dicionários é dada pela fórmula:

$$\mathsf{elementoDic} = \mathsf{blockIdx.x} + \mathsf{blockIdx.y} * \mathsf{gridDim.x} \tag{5.1}$$

Analisando a regra de associação, podemos verificar que o Bloco (0,0), $\mathsf{blockIdx.x} = 0$ e $\mathsf{blockIdx.y} = 0$, ficou responsável pelo elemento 0 do dicionário, independente da escala, assim como o Bloco (1,1) pelo elemento 129, pois a variável $\mathsf{gridDim.x}$, que faz parte da arquitetura, vale 128, que é a dimensão x do grid em questão.

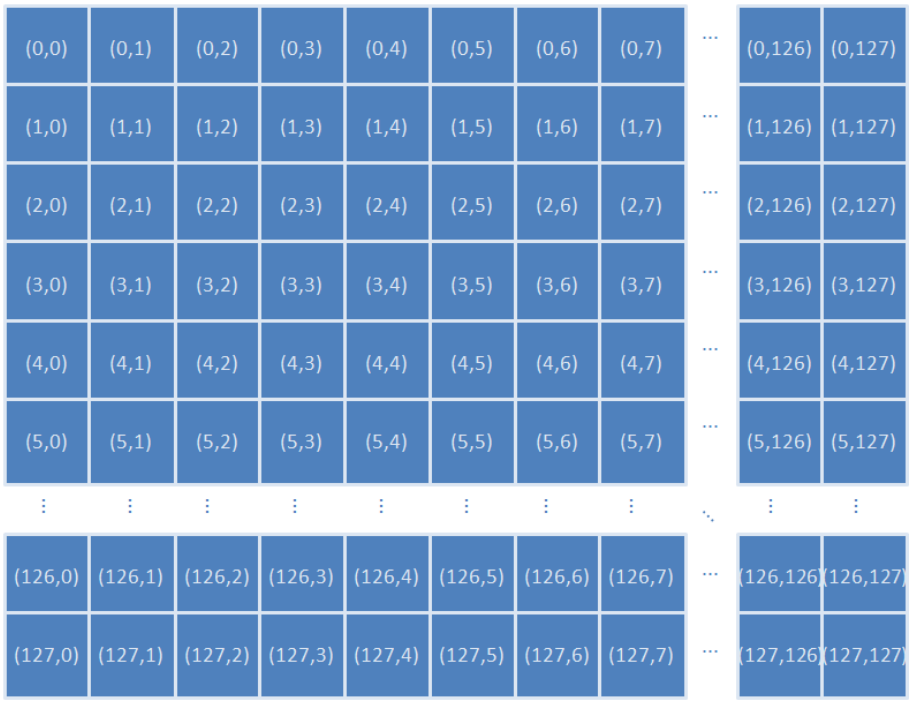


Figura 5.3: Blocos do Grid - CalculaMapaGPU

Para os *threads* temos:

$$\texttt{idThread} = \texttt{threadIdx.x} + \texttt{threadIdx.y} * \texttt{blockDim.x} \tag{5.2}$$

Ou seja, o *thread* (0,0) , $\texttt{threadIdx.x} = 0$ e $\texttt{threadIdx.y} = 0$, possui $\texttt{idThread} = 0$, assim como o *thread* (0,1), possui $\texttt{idThread} = 31$, pois a variável blockDim.x, que representa a dimensão do bloco e assim como a gridDim.x também é inerente da arquitetura, possui o valor de 31, visto que os blocos do grid são de tamanho 31x31.

O idThread está sendo incrementado percorrendo linha após linha, o que significa dizer que o último *thread* da primeira linha é o $\texttt{idThread} = 30$, o primeiro *thread* da segunda linha $\texttt{idThread} = 31$, assim como o último *thread* da quarta linha é o $\texttt{idThread} = 123$.

Além das identificações de blocos e *threads*, é preciso determinar para cada um dos *threads* qual segmento ele precisará acessar para realizar os cálculos dos custos. Abaixo podemos ver o trecho do código que associa os *threads* às escalas e também como a partir da escala a quantidade de segmentos que cada uma delas possui.

```
1 // escala horizontal
2 //(partir na direcao vertical, separando a direcao horizontal em duas)
3 m = floor(__log2f(float(threadIdx.x+1)));
4
```

```
5  // escala vertical
6  n = floor(__log2f(float(threadIdx.y+1)));
7
8  //tem que ser preenchida de acordo
9  int escala = m+n*5;
10
11 // determina a quantidade de segmentos o fragmento possui em cada escala
12 k = 0x1 << (m);
13 l = 0x1 << (n);
```

Nas tabelas 5.1 e 5.2 é possível identificar os valores de $\mathtt{m}$ e $\mathtt{n}$ para cada escala, as dimensões de cada segmento em cada uma delas e o total de segmentos de cada escala, que é calculado a partir dos valores de $\mathtt{k}$ e $\mathtt{l}$. Também estão listados os idThreads que atuam em cada escala.

Já na figura 5.4 é possível verificar visualmente na distribuição do bloco por qual escala cada um dos *threads* está responsável. O *thread* (0,0) pertence a escala 0, logo a análise que ele faz é a do fragmento 16x16. Para a escala 1 temos dois *threads*, o *thread* (1,0), $\mathtt{idThread} = 1$, e o *thread* (2,0), $\mathtt{idThread} = 2$, pois para a escala 1 temos dois segmentos de tamanho 16x8 conforme descrito na tabela 5.1.

Esse arranjo é utilizado para que cada *thread* calcule um valor do custo Lagrangeano para um elemento do dicionário. Qual elemento do dicionário que o *thread* irá verificar é o correspondente à identificação do seu bloco.

Assim, o *thread* (0,0) do bloco (0,0) do grid fará o cálculo do custo do segmento em relação ao elemento do dicionário (0,0), já o mesmo *thread* do bloco (0,15), calculará o custo do mesmo segmento a ser codificado, porém em relação ao elemento (0,15) do dicionário, que com base na equação 5.1, é o elemento de índice 1920 do dicionário da escala 0 e a escala é determinada exclusivamente pelos índices dos *threads*.

Dessa forma, temos 961 *threads* em cada um dos 128x128 blocos calculando simultaneamente o custo do segmento pelo qual é responsável em relação a um determinado elemento do dicionário. Ao final do cálculo, o que temos são os valores dos custos para cada um dos elementos dos dicionários. Esses valores são armazenados em um vetor que será utilizado posteriormente.

Essa etapa do cálculo é realizada conforme o código abaixo:

```
1 int qtd = tamanho[escala];
2 r=__log2f(float(tamanho[escala]));
```

Escala	Dimensão	m	n	k	l	Quantidade Segmentoss (k*l)	idThread
0	16x16	0	0	1	1	1	0
1	16x8	1	0	2	1	2	1, 2
2	16x4	2	0	4	1	4	3, 4, 5, 6
3	16x2	3	0	8	1	8	7, 8, 9, 10, 11, 12, 13, 14
4	16x1	4	0	16	1	16	15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30
5	8x16	0	1	1	2	2	31 62
6	8x8	1	1	2	2	4	32, 33 63, 64
7	8x4	2	1	4	2	8	34, 35, 36, 37 65, 66, 67, 68
8	8x2	3	1	8	2	16	38, 39, 40, 41, 42, 43, 44, 45 69, 70, 71, 72, 73, 74, 75, 76
9	8x1	4	1	16	2	32	46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61 77, 78, 79, 80, 81, 82, 83, 84 , 85, 86, 87, 88, 89, 90, 91, 92
10	4x16	0	2	1	4	4	93 124 155 186
11	4x8	1	2	2	4	8	94, 95 125, 126 156, 157 187, 188
12	4x4	2	2	4	4	16	96, 97, 98, 99 127, 128, 129, 130 158, 159, 160, 161 189, 190, 191, 192
13	4x2	3	2	8	4	32	100, 101, 102, 103, 104, 105, 106,107 131, 132, 133, 134, 135, 136, 137,138 162, 163, 164, 165, 166, 167, 168,169 193, 194, 195, 196, 197, 198, 199,200
14	4x1	4	2	16	4	64	108, 109, 110, 111, 112, 113,114, 115, 116, 117, 118, 119, 120, 121, 122, 123 139, 140, 141, 142, 143, 144,145, 146, 147, 148, 149, 150, 151, 152, 153, 154 170, 171, 172, 173, 174, 175,176, 177, 178, 179, 180, 181, 182, 183, 184, 185 201, 202, 203, 204, 205, 206,207, 208, 209, 210, 211, 212, 213, 214, 215, 216
15	2x16	0	3	1	8	8	217 248 279 310 341 372 403 434
16	2x8	1	3	2	8	16	218, 219 249, 250 280, 281 311, 312 342, 343 373, 374 404, 405 435, 436
17	2x4	2	3	4	8	32	220, 221, 222, 223 251, 252, 253, 254 282, 283, 284, 285 313, 314, 315, 316 344, 345, 346, 347 375, 376, 377, 378 406, 407, 408, 409 437, 438, 439, 440
18	2x2	3	3	8	8	64	224, 225, 226, 227, 228, 229, 230,231 286, 287, 288, 289, 290, 291, 292,293 255, 256, 257, 258, 259, 260, 261,262 317, 318, 319, 320, 321, 322, 323,324 348, 349, 350, 351, 352, 353, 354,355 379, 380, 381, 382, 383, 384, 385,386 410, 411, 412, 413, 414, 415, 416,417 441, 442, 443, 444, 445, 446, 447,448
19	2x1	4	3	16	8	128	232, 233, 234, 235, 236, 237,238, 239, 240, 241, 242, 243, 244, 245, 246, 247 294, 295, 296, 297, 298, 299,300, 301, 302, 303, 304, 305, 306, 307, 308, 309 263, 264, 265, 266, 267, 268,269, 270, 271, 272, 273, 274, 275, 276, 277, 278 325, 326, 327, 328, 329, 330,331, 332, 333, 334, 335, 336, 337, 338, 339, 340 356, 357, 358, 359, 360, 361,362, 363, 364, 365, 366, 367, 368, 369, 370, 371 387, 388, 389, 390, 391, 392,393, 394, 395, 396, 397, 398, 399, 400, 401, 402 418, 419, 420, 421, 422, 423,424, 425, 426, 427, 428, 429, 430, 431, 432, 433 449, 450, 451, 452, 453, 454,455, 456, 457, 458, 459, 460, 461, 462, 463, 464

Tabela 5.1: Escalas e variáveis associadas - Escala 0 a 19 - *CalculaMapaGPU*

```
3
4  if (qtd>= elementoDic){
5   D=0;
```

Escala	Dimensão	m	n	k	l	Quantidade Segmentoss (k*l)	idThread
20	1x16	0	4	1	16	16	465 496 527 558 589 620 651 682 713 744 775 806 837 868 899 930
21	1x8	1	4	2	16	32	466, 467 497, 498 528, 529 559, 560 590, 591 621, 622 652, 653 683, 684 714, 715 745, 746 776, 777 807, 808 838, 839 869, 870 900, 901 931, 932
22	1x4	2	4	4	16	64	468, 469, 470, 471 499, 500, 501, 502 530, 531, 532, 533 561, 562, 563, 564 592, 593, 594, 595 623, 624, 625, 626 654, 655, 656, 657 685, 686, 687, 688 716, 717, 718, 719 747, 748, 749, 750 778, 779, 780, 781 809, 810, 811, 812 840, 841, 842, 843 871, 872, 873, 874 902, 903, 904, 905 933, 934, 935, 936
23	1x2	3	4	8	16	128	472, 473, 474, 475, 476, 477, 478,479 503, 504, 505, 506, 507, 508, 509,510 534, 535, 536, 537, 538, 539, 540,541 565, 566, 567, 568, 569, 570, 571,572 596, 597, 598, 599, 600, 601, 602,603 627, 628, 629, 630, 631, 632, 633,634 658, 659, 660, 661, 662, 663, 664,665 689, 690, 691, 692, 693, 694, 695,696 720, 721, 722, 723, 724, 725, 726,727 751, 752, 753, 754, 755, 756, 757,758 782, 783, 784, 785, 786, 787, 788,789 813, 814, 815, 816, 817, 818, 819,820 844, 845, 846, 847, 848, 849, 850,851 875, 876, 877, 878, 879, 880, 881,882 906, 907, 908, 909, 910, 911, 912,913 937, 938, 939, 940, 941, 942, 943,944
24	1x1	4	4	16	16	256	480, 481, 482, 483, 484, 485,486, 487, 488, 489, 490, 491, 492, 493, 494, 495 511, 512, 513, 514, 515, 516,517, 518, 519, 520, 521, 522, 523, 524, 525, 526 542, 543, 544, 545, 546, 547,548, 549, 550, 551, 552, 553, 554, 555, 556, 557 573, 574, 575, 576, 577, 578,579, 580, 581, 582, 583, 584, 585, 586, 587, 588 604, 605, 606, 607, 608, 609,610, 611, 612, 613, 614, 615, 616, 617, 618, 619 635, 636, 637, 638, 639, 640,641, 642, 643, 644, 645, 646, 647, 648, 649, 650 666, 667, 668, 669, 670, 671,672, 673, 674, 675, 676, 677, 678, 679, 680, 681 697, 698, 699, 700, 701, 702,703, 704, 705, 706, 707, 708, 709, 710, 711, 712 728, 729, 730, 731, 732, 733,734, 735, 736, 737, 738, 739, 740, 741, 742, 743 759, 760, 761, 762, 763, 764,765, 766, 767, 768, 769, 770, 771, 772, 773, 774 790, 791, 792, 793, 794, 795,796, 797, 798, 799, 800, 801, 802, 803, 804, 805 821, 822, 823, 824, 825, 826,827, 828, 829, 830, 831, 832, 833, 834, 835, 836 852, 853, 854, 855, 856, 857,858, 859, 860, 861, 862, 863, 864, 865, 866, 867 883, 884, 885, 886, 887, 888,889, 890, 891, 892, 893, 894, 895, 896, 897, 898 914, 915, 916, 917, 918, 919,920, 921, 922, 923, 924, 925, 926, 927, 928, 929 945, 946, 947, 948, 949, 950,951, 952, 953, 954, 955, 956, 957, 958, 959, 960

Tabela 5.2: Escalas e variáveis associadas - Escala 20 a 24 - *CalculaMapaGPU*

```
6  int xDic = (blockIdx.x)*(blocoCol);
7  int yDic = (blockIdx.y)*(blocoLin);
```

0	1	1	2	2	2	2	3	3	3	3	3	3	3	3	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
5	6	6	7	7	7	7	8	8	8	8	8	8	8	8	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
5	6	6	7	7	7	7	8	8	8	8	8	8	8	8	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
10	11	11	12	12	12	12	13	13	13	13	13	13	13	13	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14
10	11	11	12	12	12	12	13	13	13	13	13	13	13	13	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14
10	11	11	12	12	12	12	13	13	13	13	13	13	13	13	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14
10	11	11	12	12	12	12	13	13	13	13	13	13	13	13	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14
15	16	16	17	17	17	17	18	18	18	18	18	18	18	18	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19
15	16	16	17	17	17	17	18	18	18	18	18	18	18	18	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19
15	16	16	17	17	17	17	18	18	18	18	18	18	18	18	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19
15	16	16	17	17	17	17	18	18	18	18	18	18	18	18	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19
15	16	16	17	17	17	17	18	18	18	18	18	18	18	18	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19
15	16	16	17	17	17	17	18	18	18	18	18	18	18	18	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19
15	16	16	17	17	17	17	18	18	18	18	18	18	18	18	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19
15	16	16	17	17	17	17	18	18	18	18	18	18	18	18	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19
20	21	21	22	22	22	22	23	23	23	23	23	23	23	23	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24
20	21	21	22	22	22	22	23	23	23	23	23	23	23	23	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24
20	21	21	22	22	22	22	23	23	23	23	23	23	23	23	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24
20	21	21	22	22	22	22	23	23	23	23	23	23	23	23	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24
20	21	21	22	22	22	22	23	23	23	23	23	23	23	23	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24
20	21	21	22	22	22	22	23	23	23	23	23	23	23	23	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24
20	21	21	22	22	22	22	23	23	23	23	23	23	23	23	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24
20	21	21	22	22	22	22	23	23	23	23	23	23	23	23	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24
20	21	21	22	22	22	22	23	23	23	23	23	23	23	23	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24
20	21	21	22	22	22	22	23	23	23	23	23	23	23	23	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24
20	21	21	22	22	22	22	23	23	23	23	23	23	23	23	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24
20	21	21	22	22	22	22	23	23	23	23	23	23	23	23	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24
20	21	21	22	22	22	22	23	23	23	23	23	23	23	23	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24
20	21	21	22	22	22	22	23	23	23	23	23	23	23	23	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24
20	21	21	22	22	22	22	23	23	23	23	23	23	23	23	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24
20	21	21	22	22	22	22	23	23	23	23	23	23	23	23	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24

Figura 5.4: Escalas associadas a cada *thread*

```
int xElemento = (threadIdx.x-(k-1))*(blocoCol/k);
int yElemento = (threadIdx.y-(l-1))*(blocoLin/l);
float blocoLido = 0;
float elementoDicLido = 0;
for(i = 0; i <blocoCol/k; i++) {
    for(j = 0; j <blocoLin/l; j++) {
        blocoLido=dev_elemento[(yElemento+j)*blocoCol+xElemento+i];
        elementoDicLido=dev_dicionario[escala*(128*blocoLin)*(128*blocoCol)
            + (yDic)*blocoCol*128 + (xDic)*blocoLin + (yElemento+j)*blocoCol
             + xElemento+i];
        D=D+(blocoLido-elementoDicLido)*(blocoLido-elementoDicLido);
    }
}
```

```
19    J=D+lambda*r;
20  }
21  mapa[elementoDic*961+idThread]=J;
```

A variável `qtd` identifica o tamanho do dicionário em cada escala, assim garantimos que não será realizado um cálculo de custo para um elemento que vá além dos limites do dicionário para aquela escala. Utilizamos também o logaritmo base dois desta variável para estimar a taxa `r` associada à transmissão desde elemento, uma vez que nesta versão do programa esta informação, que está em uma variável privada da classe CodificadorAritmético na CPU, não foi transferida para a GPU. Conforme os testes demonstram, não há diferença no desempenho da compressão obtida ao utilizarmos esta aproximação.

As variáveis `xDic` e `yDic` identificam o pixel inicial do elemento do dicionário que o *thread* deverá ler. Temos também `xElemento` e `yElemento` que posiciona o *thread* no pixel inicial no fragmento de entrada refente ao segmento que deve analisar. Por exemplo, para o *thread* (1,0), `idThread` = 1, com essa associação garantimos que ele fará a leitura a partir do pixel (0,0). Para o *thread* (2,0), `idThread` = 2 , que é responsável pelo segundo segmento da escala 1 realizará a leitura a partir do pixel (16,0). Na figura 5.5 podemos verificar como os *threads* (1,0) e (2,0) fazem a leitura do fragmento de entrada, assim como os *threads* das escalas 0, 2, 5, 6 e 7.

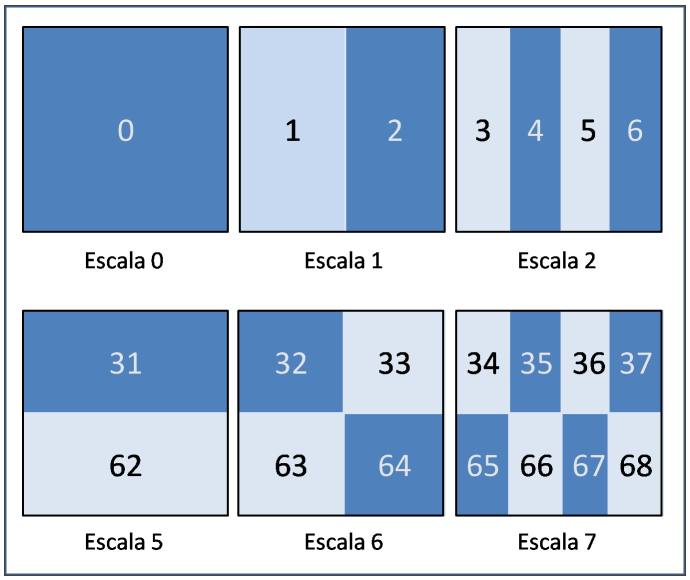


Figura 5.5: Exemplo de leitura do bloco de entrada para alguns iTdThreads

Para realizar o cálculo pixel a pixel, se faz necessária a utilização de `blocoLido` e `elementoDicLido`, ambos inicializados com 0, para armazenar o valor do pixel do frag-

mento de entrada e do elemento do dicionário, respectivamente. A distância quadrática é armazenada em D e ao fim do loop o valor de J é calculado. Por último em mapa é registrado o valor de todos os custos para todos os segmentos em relação a todos os elementos dos dicionários de todas as escalas.

Concluída a etapa de cálculo dos custos Lagrangenos J, o que precisamos descobrir é qual o menor valor do custo para cada um dos segmentos, para que a partir dessa informação a função ÁrvoreOtima possa decidir como o fragmento deverá ser transmitido.

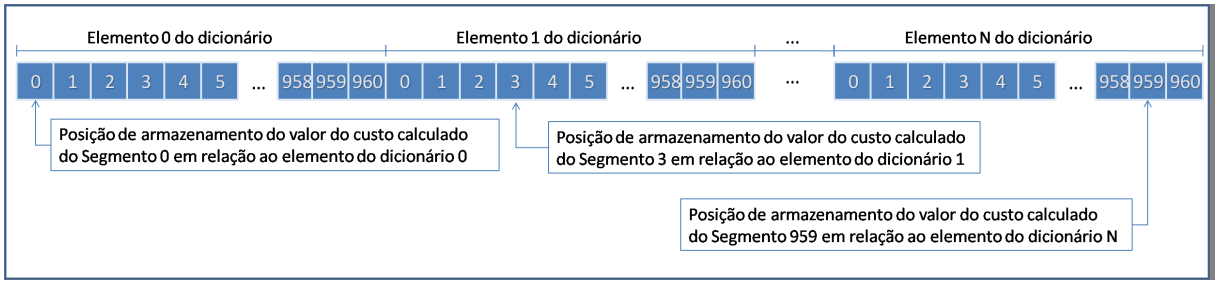


Figura 5.6: Organização do vetor que armazena o cálculo dos custos

O vetor mapa que possui o cálculo de todos os custos está organizado conforme a figura 5.6.

$$\mathtt{mapa}[\mathtt{elementoDic} * 961 + \mathtt{idThread}] = J \tag{5.3}$$

Para que fosse encontrado dentre esses valores o menor deles, fez-se necessária uma outra etapa. Essa etapa foi desenvolvida em dois novos *kernels*:*reduzMapa* e *reduzMapa2*.

5.1.1.2 *Kernel* ***reduzMapa***

O primeiro dos *kernels* necessário para o cálculo do menor J por segmento pode ser observado com o nome de *reduzMapa* no Apêndice A. Ele possui um grid com arranjo de (31x8)x31 blocos sendo cada um com dimensão de 32x32 *threads*. Os elementos dos dicionários foram separados em 8 faixas conforme observado na tabela 5.3.

Cada bloco foi associado a um segmento e os *threads* aos elementos do dicionário. É importante observar que apenas possuímos 32x32 *threads* por bloco e um dicionário máximo de 128x128 elementos, fazendo serem necessários 16 blocos de *threads* para que todos os valores fossem comparados em apenas um *kernel*, porém esse arranjo não foi possível por limitação de hardware e a etapa para encontrar o menor custo foi desenvolvida em dois *kernels* conforme dito anteriormente. Nessa arquitetura possuímos oito blocos sendo utilizados para comparar os valores para um mesmo segmento, ou seja, o índice

Faixa	Índices Elementos
0	0 a 2047
1	2048 a 4095
2	4096 a 6143
3	6144 a 8191
4	8192 a 10239
5	10240 a 12287
6	12288 a 14335
7	14336 a 16383

Tabela 5.3: Faixas dos dicionários para redução

do bloco vai determinar para qual a faixa e segmento o *thread* deverá consultar o valor calculado na etapa anterior.

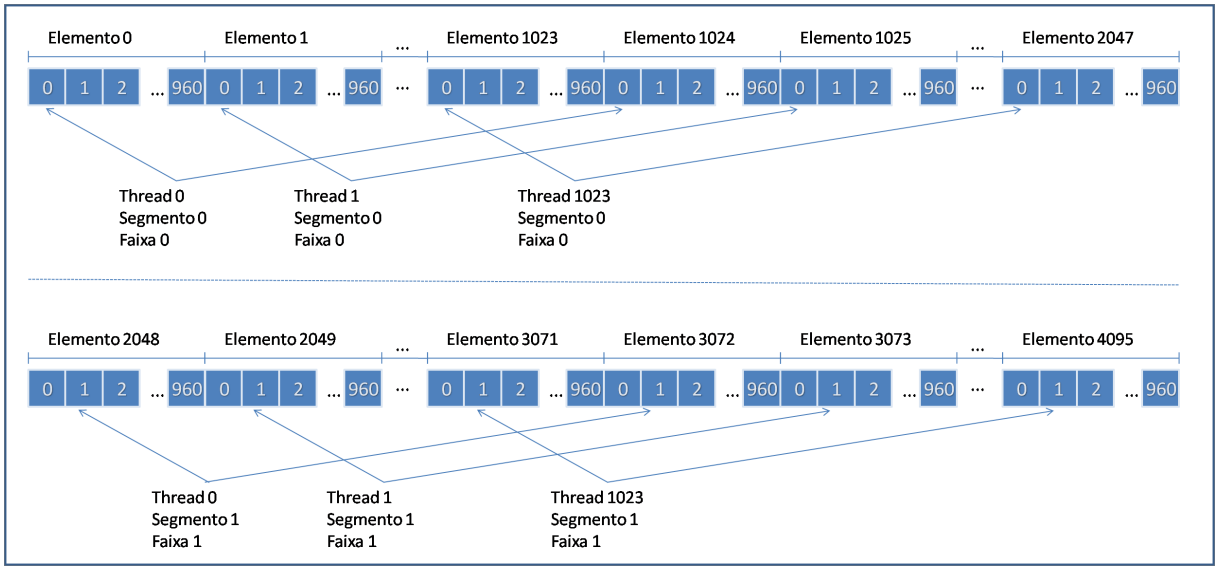


Figura 5.7: Primeira iteração *kernel ReduzMapa*

Na primeira iteração, cada um dos 1024 *threads* irá comparar dois valores abrangendo desta forma toda a faixa e armazenará o menor deles. Na figura 5.7 podemos verificar um exemplo para o *thread* 0 do segmento 0 da primeira faixa. Esse *thread* irá comparar o valor do custo calculado do segmento 0 em relação ao elemento do dicionário 0 e o elemento 1024. O menor desses dois valores será armazenado na posição mais a esquerda, que nesse caso é a posição do vetor que armazena o custo do elemento 0. Além dessa cópia do custo, um outro vetor organizado da mesma forma armazena as informações dos índices. Para o caso do *thread* 1023, suponhamos que o menor valor do custo Lagrangeano seja em relação ao elemento 2047 nessa iteração, então o vetor de índices na posição do segmento 0 para o elemento 1023 receberá o valor de 2047, assim como o vetor

de custo na mesma posição receberá o valor do custo do elemento 2047.

As definições no código abaixo garantem essa associação:

```
1    int idThread =  blockIdx.x + blockIdx.y*gridDim.x;
2    int subBloco=(idThread)%961;
3    int n=(idThread/961);
4    int indiceElemento = (threadIdx.x + threadIdx.y*blockDim.x);
5    int i = (blockDim.x*blockDim.y);
```

Onde `n` determina a faixa, `i` a quantidade de *threads* que irão realizar a comparação (inicialmente 1024), `indiceElemento` posiciona em relação ao elemento do dicionário e `subBloco` localiza os 961 segmentos.

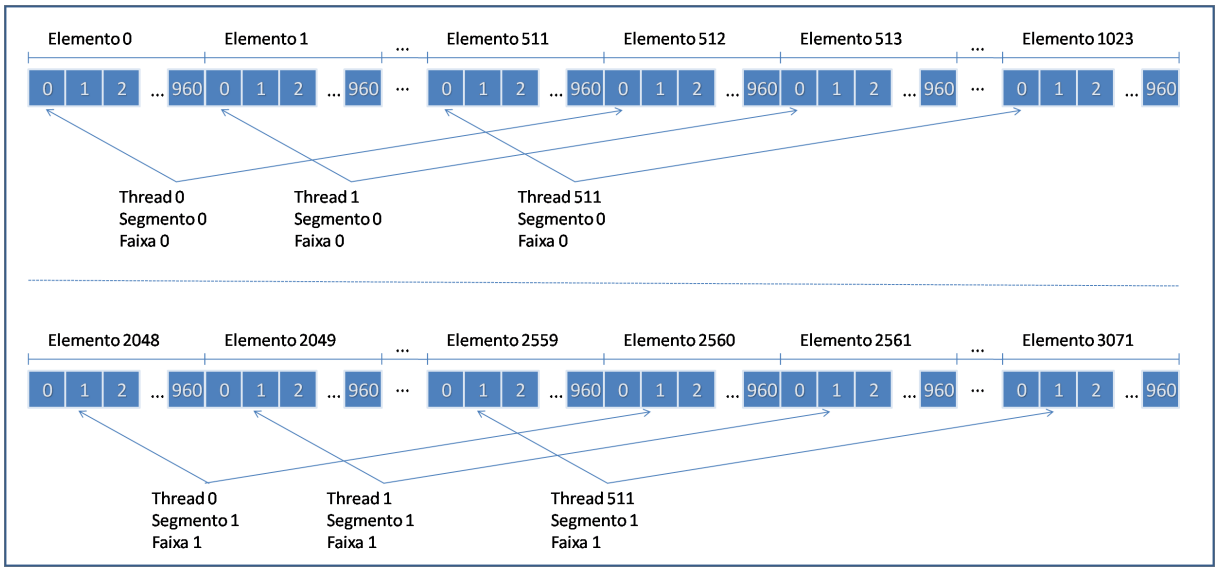


Figura 5.8: Segunda iteração *kernel ReduzMapa*

Na iteração seguinte, os valores a serem comparados é reduzido a metade, visto que os menores custos comparados na iteração anterior estão localizados mais a esquerda dos vetores. Na figura 5.8 podemos observar que agora apenas os primeiros 512 *threads* precisam reexecutar o processo. Assim é feito sucessivamente até que cada bloco possua o menor valor de custo da sua faixa na posição do primeiro elemento da faixa, assim como o índice do dicionário relacionado. Ao final desse *kernel* para cada faixa temos os menores 961 custos (um para cada escala) armazenados nas primeiras posições conforme figura 5.9.

O loop abaixo que garante a execução desse processo.

```
1   while (i != 0) {
```

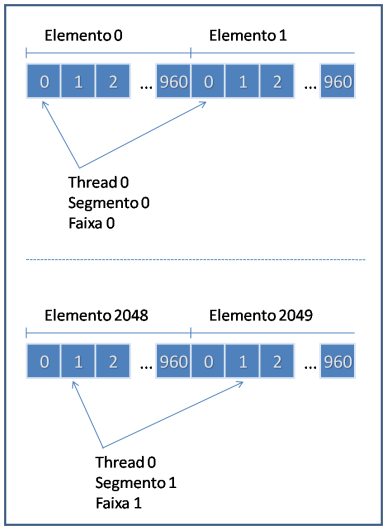


Figura 5.9: Última iteração *kernel ReduzMapa*

```
2    if ((indiceElemento)< i && subBloco <961 && mapa[(indiceElemento + n
         *1024*2 + i)*961+subBloco]>-1) {
3      if(mapa[(indiceElemento + n*1024*2)*961+subBloco] > mapa[(
           indiceElemento + n*1024*2 + i)*961+subBloco]){
4           mapa[(indiceElemento + n*1024*2)*961+subBloco] = mapa[(
                indiceElemento + n*1024*2 + i)*961+subBloco];
5           indice[(indiceElemento + n*1024*2)*961+subBloco] = indice[(
                indiceElemento + n*1024*2 + i)*961+subBloco];
6           }
7      }
8      __syncthreads();
9    i /= 2;
10 }
```

Importante observar que o valor de `i` é reduzido à metade a cada loop e se fez necessária a utilização da instrução `__syncthreads()` que garante que somente após todos os *threads* terem concluído o armazenamento do menor valor comparado e índice correspondente é dada continuidade à iteração seguinte.

5.1.1.3 *Kernel* ***reduzMapa2***

Conforme descrito no *kernel reduzMapa*, ao final dela temos selecionado o menor custo para cada uma das faixas totalizando oito valores para cada segmento. Porém a função *CalculaMapa* deve gerar apenas um valor de custo e índice. O segundo *kernel* dessa etapa *reduzMapa2*, que também se encontra detalhada no Apêndice A, realiza a

comparação dos valores restantes com um arranjo de 31x31 blocos cada um com 2x2 *threads*. O funcionamento da comparação é semelhante ao *kernel reduzMapa*, porém nesse segundo momento cada bloco do grid está associado a um segmento e cada *thread* verifica dois dos oito valores escolhidos da etapa anterior. O funcionamento desse segundo *kernel* está ilustrado na figura 5.10. Temos apenas 4 *threads* para cada segmento que em apenas 3 iterações determinam o menor o menor custo entre eles.

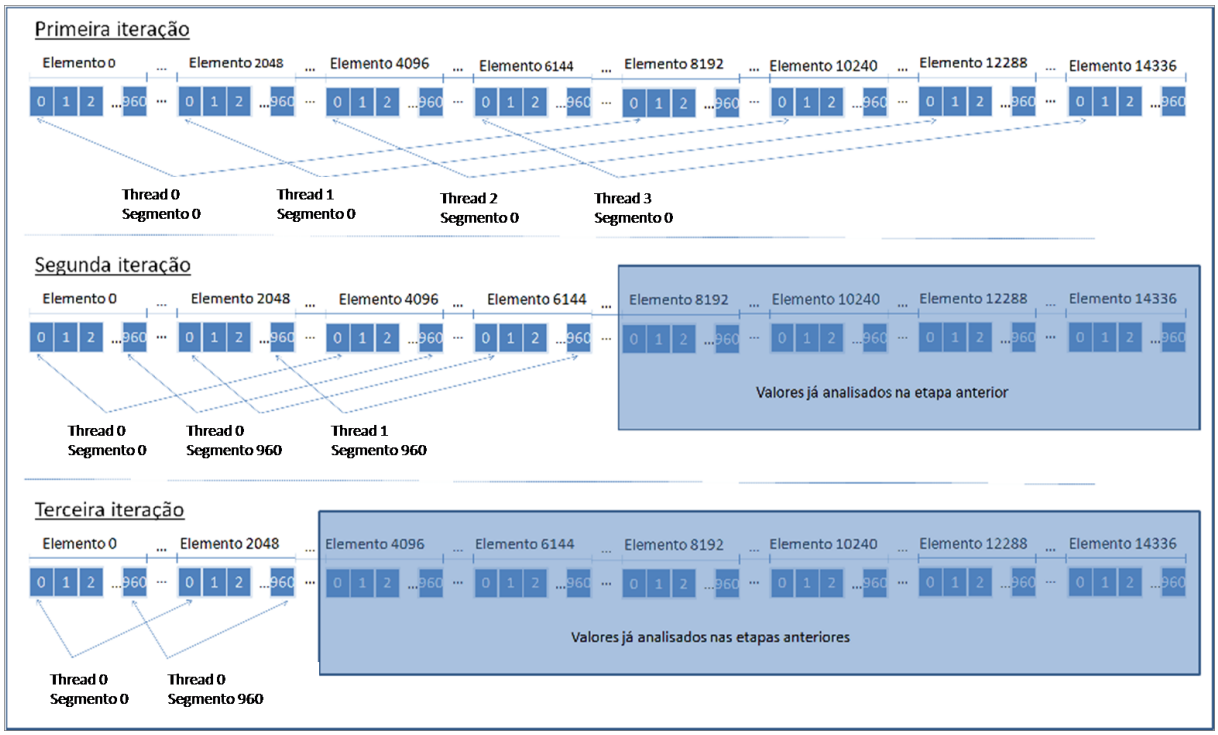


Figura 5.10: Funcionamento da Redução do *kernel ReduzMapa2*

Abaixo o código que implementa a função *reduzMapa2*.

```
1  int subBloco =  blockIdx.x + blockIdx.y*gridDim.x;
2  int indiceElemento = threadIdx.x + threadIdx.y*blockDim.x;
3  int i = (blockDim.x*blockDim.y);
4  while (i != 0) {
5   if (indiceElemento< i && subBloco <961 && mapa[(indiceElemento*2048 + i
       *2048)*961+subBloco]>-1) {
6       if(mapa[(indiceElemento*2048)*961+subBloco] > mapa[(indiceElemento*2048
            + i*2048)*961+subBloco]){
7         mapa[(indiceElemento*2048)*961+subBloco] = mapa[(indiceElemento*2048
              + i*2048)*961+subBloco];
8         indice[(indiceElemento*2048)*961+subBloco] = indice[(indiceElemento
              *2048 + i*2048)*961+subBloco];
```

```
9           }
10        }
11     __syncthreads();
12     i /= 2;
13  }
```

Os três *kernels* discutidos nessa seção (*calculaMapa* , *reduzMapa*, *reduzMapa2*) são executados nessa sequência, porém esses valores estão calculados e armazenados até o momento na GPU, o que faz com que seja necessária a cópia dos 961 valores de custos e índices para a CPU.

A organização dos dados gerados pela GPU não foi a mesma utilizada pela CPU da função original e o formato esperado pela função *ArvoreOtima* ainda é o mesmo. Assim, os vetores de mapa e índice resultantes da função *calculaMapaGPU* tiveram que ser mapeados para a variável *Cotimo* conforme código abaixo, onde m e n são os índices das escalas verticais e horizontais e k e l correspondem aos segmentos.

```
1  for(k = 0; k < Nmax/Dic.linhas[n][m]; k++) {
2     for(l = 0; l < Mmax/Dic.colunas[n][m]; l++) {
3        Cotimo[n][m][k][l].Jmin = mapa[l+(0x1<<m)-1+31*(k+(0x1<<n)-1)];
4        Cotimo[n][m][k][l].Imin = indice[l+(0x1<<m)-1+31*(k+(0x1<<n)-1)];
5     }
6  }
```

Dessa forma garantimos que os 961 valores de J e índices estão disponíveis para a CPU continuar a codificação do fragmento de entrada.

5.2 Implementação da função *Compara*

Após a função *CalculaMapa* ter sido substituída pela *CalculaMapaGPU*, novamente foi feita análise com o *gprof* que evidenciou a função *Compara* como sendo a ofensora no novo panorama. Essa evidência pode ser observada na figura 5.11.

Dessa forma, foi também implementada a função *Compara* em uma versão para ser executada pela GPU. Chamamos esse novo *kernel* de *comparaBloco*.

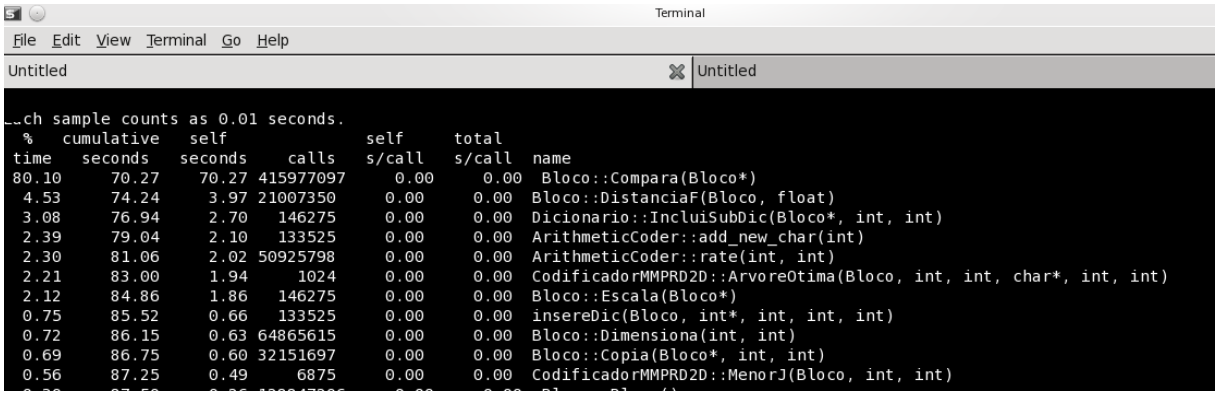


Figura 5.11: Saída do comando gprof com a função *CalculaMapaGPU* implementada

5.2.1 Especificações da *comparaBloco*

5.2.1.1 *Kernel* ***comparaBloco***

A função *Compara* é responsável por parte do aprendizado do MMP. Ela faz o teste para verificar se os novos elementos formados a partir das dilatações e contrações dos elementos criados já existem no dicionário. Ela precisa percorrer todos os elementos de todos os dicionários e decidir se o novo elemento deve ou não ser adicionado ao dicionário.

Para esse *kernel* o grid de blocos foi de tamanho (128/2)x(128/2) e os *threads* (5*2)x(5*2). Com esses valores temos 25 *threads* para cada um dos possíveis 128x128 elementos de cada dicionário.

Com as definições abaixo para `xElementoDic`, `yElementoDic` e `escalaBloco`, garantimos que a escala é sempre determinada a partir dos índices dos *threads* e temos cada bloco do grid responsável por quatro elementos do dicionário para cada escala.

```
1 int xElementoDic = 2*blockIdx.x + threadIdx.x/5;
2 int yElementoDic = 2*blockIdx.y + threadIdx.y/5;
3 escalaBloco = threadIdx.x%5 + 5 * (threadIdx.y%5);
```

O bloco (0,0), por exemplo, possuirá *threads* de todas as 25 escalas que acessarão os elementos dos dicionários de índices (0,0), (0,1), (1,0) e (1,1). Na tabela 5.4 estão representados os grupos de 25 *threads* que acessariam os elementos (0,0), (0,1), (1,0) e (1,1) dos dicionários para o caso do bloco (0,0) do grid.

Se temos três novos elementos criados, significa que serão gerados 75 novos elementos que deverão ser verificados se já existem nos dicionários, pois cada novo elemento gera versões para cada uma das 25 escalas. A função *comparaBloco* é chamada para cada novo elemento, ou seja, para o exemplo de três novos elementos criados ela seria executada três

	Lista de threads que acessam o elemento para o bloco (0,0)
Elemento Dicionário (0,0)	0 1 2 3 4 10 11 12 13 14 20 21 22 23 24 30 31 32 33 34 40 41 42 43 44
Elemento Dicionário (0,1)	50 51 52 53 54 60 61 62 63 64 70 71 72 73 74 80 81 82 83 84 90 91 92 93 94
Elemento Dicionário (1,0)	5 6 7 8 9 15 16 17 18 19 25 26 27 28 29 35 36 37 38 39 45 46 47 48 49
Elemento Dicionário (1,1)	55 56 57 58 59 65 66 67 68 69 75 76 77 78 79 85 86 87 88 89 95 97 97 98 99

Tabela 5.4: Lista de *threads* por elemento do dicionário - Bloco (0,0)

vezes. A cada execução ela recebe 25 novos elementos, um de cada escala, e o arranjo de blocos e *threads* conforme descrito anteriormente garante que esses 25 novos elementos sejam comparados com cada um dos elementos dos dicionários já existentes.

O resultado gerado pela execução do *kernel* é um vetor chamado `dev_resultado` de tamanho 25 que possui um flag indicando para cada uma das escalas se o novo elemento deve ou não ser adicionado ao dicionário. Abaixo o código do *kernel* que realiza as comparações e registra se o elemento procurado já existe.

```
1  if((yElementoDic*128+xElementoDic) <= tamanho[escalaBloco] && escalaBloco <
       25) {
2    float blocoLido = 0;
3    float elementoDicLido = 0;
4    for(i = 0; i <blocoCol/k; i++) {
5      for(j = 0; j <blocoLin/l; j++) {
6        blocoLido=dev_elemento[(escalaBloco)*16*16 + j*16+i];
7        elementoDicLido=dev_dicionario[escalaBloco*(128*16)*(128*16)+ (
             yElementoDic*128 + xElementoDic)*16*16 + j*16 + i];
8        if(blocoLido != elementoDicLido) D = 0;
9      }
10   }
11   if(D==1){
12     dev_resultado[escalaBloco]=0;
13   }
14 }
```

A partir da finalização do *kernel*, é retornado para CPU um array que indica para quais escalas de ser realizada a inclusão do novo elemento. O aprendizado é realizado tanto no dicionário da CPU e quanto no da GPU.

Capítulo 6

Resultados Experimentais

6.1 Especificação dos testes

Após implementação das funções descritas no capítulo 5 era necessário verificar se a utilização do paralelismo da GPU acarretaria uma redução no tempo de execução do MMP. Foram utilizadas para as simulações três placas de vídeo: a GTX-580, GTX-480 e GTS-450. Na tabela 6.1 é possível comparar as características técnicas dos três hardwares em questão.

Chip Gráfico	Clock	Clock da Memória	Memória	Taxa de Transf. Memória	Pixels por Clock
GeForce GTS 450	783 / 1.566 MHz	3,6 GHz	128 bits	57,7 GB/s	192
GeForce GTX 480	700 / 1.401 MHz	3.696 MHz	384 bits	177,4 GB/s	480
GeForce GTX 580	772 / 1.544 MHz	4.008 MHz	384 bits	192,4 GB/s	512

Tabela 6.1: Placas utilizadas nas simulações

Todos os testes foram realizados em ambiente Linux, onde foram feitas simulações para 32bits e 64bits. Os processadores utilizados são os que podem ser observados na tabela 6.2: i7 860 da Intel, i7 2600 da intel e o Phenom II X6 da AMD.

A placa GTS-450 operou em conjunto com o processador Phenom II X6 da AMD em um sistema Linux exclusivamente de 32 bits, a placa GTX-580 com o processador i7 860 da Intel em sistema Linux 32 bits e 64 bits e a GTX-480 com o processador i7 2600 também da Intel com Linux 64 bits somente.

Para cada uma dessas configurações foram realizados três testes. Um em que o algoritmo de compressão é executado exclusivamente na CPU com apenas 1 *thread*, outro

Processador	Número de Núcleos	Número de Threads	Cache	Velocidade do Clock	Canais de Memória
i7 2600	4	8	8 MB	3,4 GHz	2
i7 860	4	8	8 MB	2,8 GHz	2
Phenom II X6	6	6	6 MB	2,6 GHz	2

Tabela 6.2: Processadores utilizados nas simulações

em que ainda é executado exclusivamente na CPU, porém fazendo uso dos mulicores das mesmas e um terceiro em que o algoritmo é executado em conjunto na CPU com um único *thread* e na GPU. Os programas foram escritos em C++ e o compilador utilizado foi o gcc 4.4.4, com suporte a OpenMP para explorar o paralelismo na CPU. Para a GPU utilizou-se o mesmo gcc associado ao nvcc 4.0. Para cada um desses testes foram feitas simulações com λ de valores de 64, 32 e 16. Cada simulação foi realizada 10 vezes e o valor de tempo considerado foi a média das 10 iterações para cada um dos casos. Um resumo dos parâmetros utilizados em cada teste realizado pode ser encontrado na tabela 6.3

Hardware	Sistema Operacional	Valores de λ	Número de Iterações
i7 2600 + Placa GTX-480	64 bits	16, 32 e 64	10
i7 860 + Placa GTX-580	64 bits	16, 32 e 64	10
i7 860 + Placa GTX-580	32 bits	16, 32 e 64	10
Phenom II X6 + Placa GTS-450	32 bits	16, 32 e 64	10

Tabela 6.3: Resumo dos parâmetros utilizados nos testes

Para todos os testes foi utlizada a imagem Lena de 512 x 512 pixels. Uma cópia da Lena pode ser vista na figura 6.1.

6.2 Testes em Linux 64 bits

Para os testes em Linux 64 bits, os hardwares utilizados foram a placa GTX-580 com o processador i7 860 da Intel e a GTX-480 com o processador i7 2600 da Intel.

Os tempos em segundos para as 10 iterações realizadas nos testes em Linux 64 bits podem ser verificados nas tabelas B.1, B.2, B.3, B.4, B.5 e B.6, no Apêndice B.

Figura 6.1: Imagem Lena utilizada nos testes

A média dos tempos encontrados para as simulações com $\lambda = 16$ encontram-se na figura 6.2. Para as duas CPUs (i7 860 e i7 2600) executando o programa com apenas um *thread*, tivemos um tempo de quase 14 minutos (751.61 segundos) para uma e um pouco mais de 10 minutos (619.46 segundos) para outra. Cada uma das CPUs possui 8 núcleos e os valores encontrados com a execução multithread mostra uma redução do tempo de processamento para menos da metade do tempo. Para o i7 860 a mudança foi de 751.61 segundos para 279.57 segundos, e para o i7 2600 de 619.46 segundos para 266.91 segundos. Se observarmos os tempos de execução em conjunto CPU e GPU, notamos que estes foram reduzidos para valores inferiores a dois minutos: 118.46 segundos para a placa GTX-580 operando com o i7 860 e 118.49 segundos para a GTX-480 com o i7 2600.

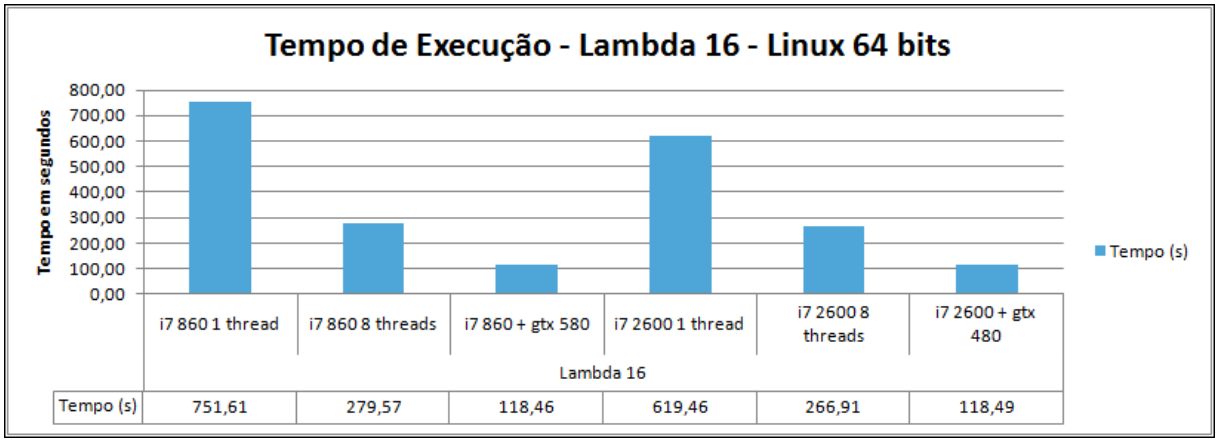


Figura 6.2: Lambda 16 - Linux 64 bits

Para os testes com $\lambda = 32$, verificamos que os tempos de execução são inferiores

ao dos testes com $\lambda = 16$, isto acontece porque ao tolerarmos mais distorção aumentamos a probabilidade de escolher árvores de segmentação menos profundas o que faz com que os dicionários cresçam menos, consequentemente tornando mais rápida a busca nos dicionários e a execução do algoritmo. Para o i7 860 a evolução dos tempos foi a partir de 486.38 segundos com apenas um *thread* sendo executado na CPU, houve uma redução para 177.42 segundos quando executando com 8 *threads* e no conjunto CPU e GPU 80.35 segundos. Para o i7 2600 os ganhos também foram observados: 383.45 segundos, 160.02 segundos e 81.90 segundos para 1 *thread*, 8 *threads* e CPU com GPU, respectivamente. Esses valores podem ser observados na figura 6.3.

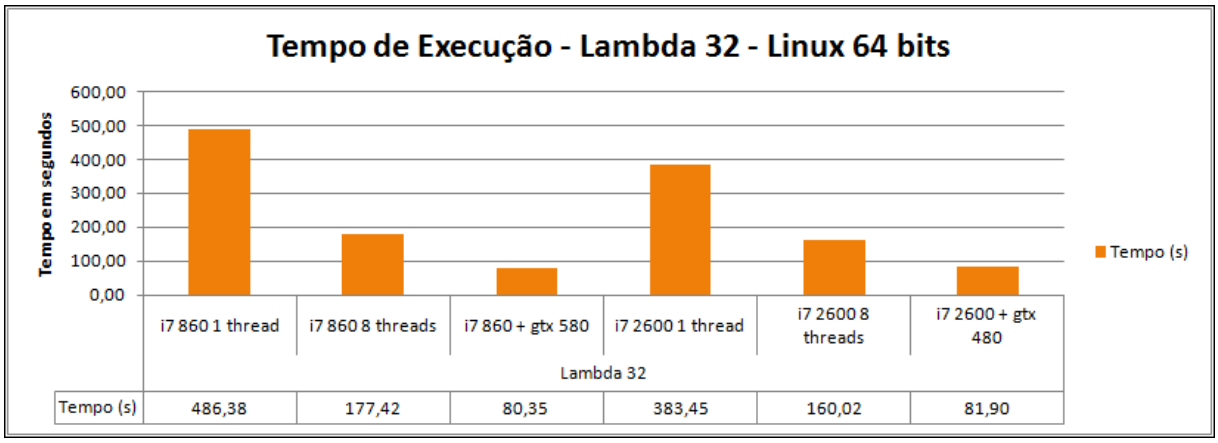


Figura 6.3: Lambda 32 - Linux 64 bits

O último teste, para $\lambda = 64$, está com seus valores representados na figura 6.4. Nela observamos um comportamento semelhante ao encontrado nos dois testes anteriores. Uma aceleração no tempo quando simulados com 8 *threads* e um ganho ainda maior quando executado na configuração CPU e GPU. Para esse, o programa executou em 329.97 segundos para um *thread* no processador i7 860, 117.88 segundos para oito *threads* e 58.82 segundos para a execução nesse processador em conjunto com a placa GTX-580. Para o i7 2600 os valores encontrados foram de 255.50 segundos para a execução com apenas um *thread*, 103.75 segundos com 8 *threads* e 61.77 segundos para execução com a placa GTX-480.

Se formos comparar somente as execuções com apenas um *thread*, para todos os três valores de λ, o processador i7 2600 obteve tempos menores que o i7 860, assim como para os testes exclusivos na CPU com os oito *threads* simultâneos. Esse comportamento é esperado devido ao seu clock de 3,4GHz ser mais rápido que o do i7 860.

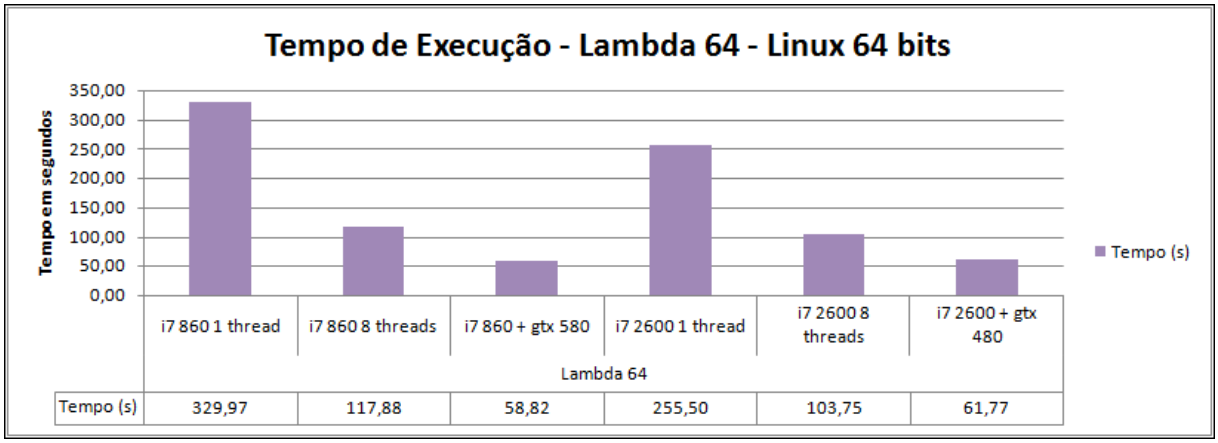


Figura 6.4: Lambda 64 - Linux 64 bits

Ao compararmos as especificações das placas GTX-580 e GTX-480 listadas na tabela 6.1, verificamos que a GTX-580 é uma placa mais robusta quando consideramos a taxa de transferência da memória. Essa característica faz com que mesmo associada à CPU menos veloz, sempre obtenha os menores tempos de execução.

Na figura 6.5 identificamos os ganhos das simulações executadas na CPU com oito *threads* e também os ganhos das execuções híbridas GPU e CPU ambas em relação à execução com apenas um *thread*. Apesar do tempo de execução do processador i7 860 ter sido sempre maior em comparação aos i7 2600, se verificarmos o ganho relativo, para os três valores de λ , ele possui um ganho ligeiramente superior.

O destaque está com as execuções do algoritmo que envolveram CPU e GPU em conjunto. Para o processador i7 860 operando em conjunto com a placa GTX-580 obtivemos um ganho de 6.34 vezes para $\lambda = 16$, 6.05 vezes para $\lambda = 32$ e 5.61 vezes para $\lambda = 64$. Não menos promissores foram os resultados para o processador i7 2600 com a placa GTX-480 que obtiveram ganhos de de 5.23 , 4.68 e 4.14 vezes para λ de 16, 32 e 64, respectivamente. Ganhos considerados bastante elevados, ainda mais se considerarmos os ganhos com o paralelismo utilizando os multicores da CPU que não foram superiores a 2.8 vezes.

6.3 Testes em Linux 32 bits

Nos testes em Linux 32 bits, utilizamos a placa GTS-450 que operou em conjunto com o processador Phenom II X6 da AMD e a placa GTX-580 com o processador i7 860

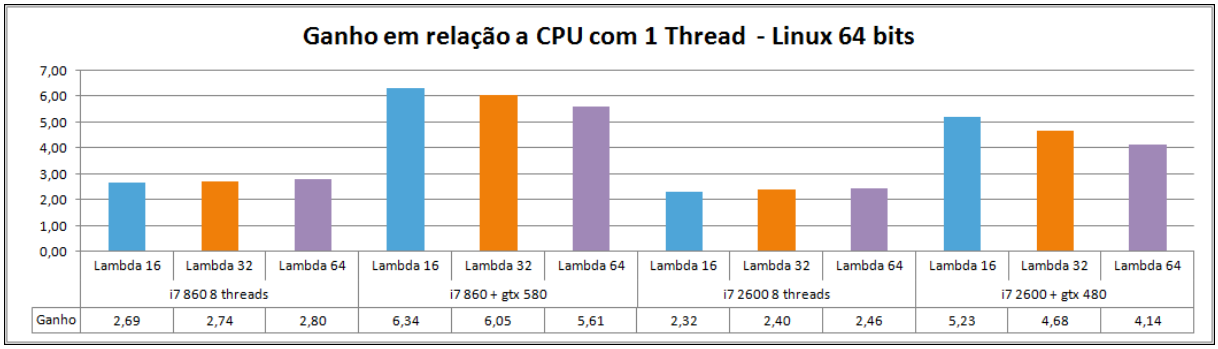


Figura 6.5: Ganho - Linux 64 bits

da Intel conforme especificado na tabela 6.3.

Conforme dito anteriormente, foram feitas 10 simulações para cada teste. Os tempos em segundos para as 10 iterações realizadas nos testes em Linux 32 bits podem ser verificados nas tabelas B.7, B.8, B.9, B.10, B.11 e B.12, que se encontram no Apêndice B.

A média dos tempos encontrados para as simulações com $\lambda = 16$ no ambiente Linux 32 bits estão representados na figura 6.6. Nesse gráfico observamos para a CPU i7 860 um tempo de 838.14 segundos para executação com apenas um *thread*, 240.28 segundos com oito *threads* e 115.91 segundos quando executado em conjunto com a GPU GTX-580. Além dessa configuração temos também o Phenom II X6 que consumiu 997.90 segundos com um *thread*, 296.38 segundos com seis *threads* e 350.83 segundos com a placa GTS-450. O processador Phenom II X6 da AMD possui menos núcleos que o i7 860 da Intel, o que limitou em seis o número de *threads* para essa CPU.

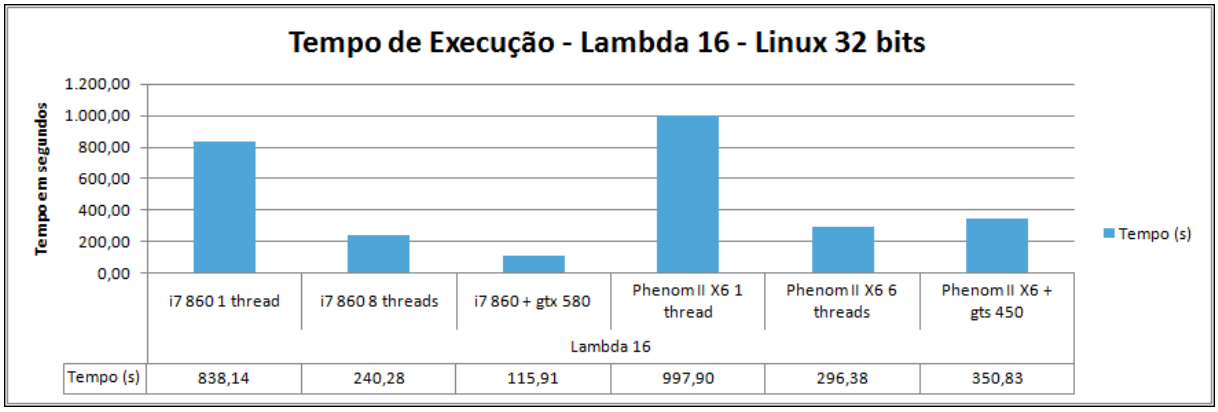


Figura 6.6: Lambda 16 - Linux 32 bits

Para $\lambda = 32$ o desempenho para o processador da Intel i7 860 foi de 533.55 segun-

dos com um *thread*, enquanto o Phenom II X6 da AMD apresentou um desempenho de 622.32 segundos. Assim como para o valor de $\lambda = 16$, o processador da Intel apresentou melhor desempenho. Porém com a execução mutithread, ainda exclusiva na CPU, proporcionalmente o Phenom II X6 obteve um ganho superior no seu tempo de 622.32 segundos para 178.44 em relação ao ganho de 533.55 segundos para 155.84 segundos do i7 860 que possui dois *threads* a mais. Apesar de um ganho relativo superior, o i7 860 executou mais rapidamente o algoritmo considerando o critério multithreads.

Para a simulação GPU e CPU o tempo do conjunto i7 860 e GTX-580 foi de 81.52 segundos, enquanto a placa GTS-450 com o Phenom II X6 apresentou desempenho de 248.23 segundos, tempo de execução superior aos 178.44 segundos para seis *threads* somente na CPU. Esses valores estão ilustrados na figura 6.7.

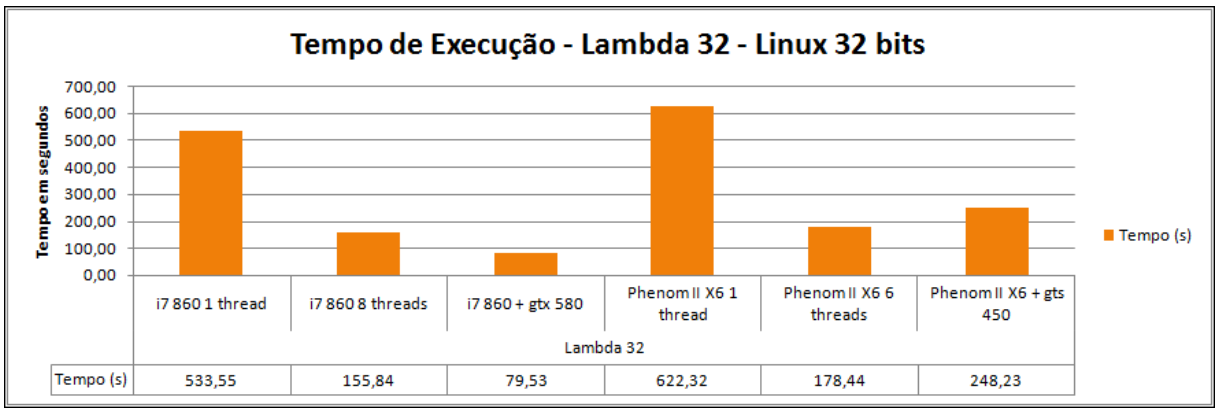


Figura 6.7: Lambda 32 - Linux 32 bits

Os resultados do terceiro teste para $\lambda = 64$ podem ser observados na figura 6.8. Nele é possível verificar o mesmo comportamento apresentado nos dois testes anteriores, um ganho considerável com a execução multithread exclusivamente na CPU para ambas as configurações, porém somente a placa GTX-580 apresentou ganho de execução em relação à proposta multithread.

Os ganhos apresentados pelas CPUs em relação à execução com apenas um *thread* não ultrapassou o valor de 3.5 vezes. Esses valores podem ser observados na figura 6.9. Interessante observar que o i7 860, que apresentou ganhos de 3.49 , 3.42 e 3.36 para λ de 16, 32 e 64, respectivamente, mesmo com dois *threads* a mais (oito), obteve um desempenho semelhante ao Phenom II X6 que possui apenas seis *threads* que obteve ganhos de 3.37, 3.49 e 3.41.

Apesar de um ganho de quase 3.5 vezes ser um valor considerável, o ganho para a

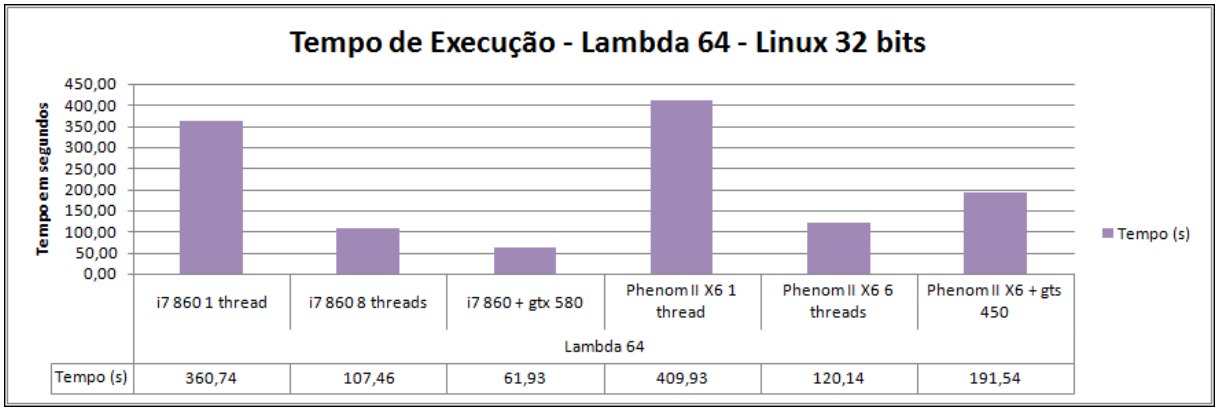


Figura 6.8: Lambda 64 - Linux 32 bits

placa GTX-580 operando em conjunto com o i7 860, chegou a 7.23 para $\lambda = 16$, 6.54 para $\lambda = 32$ e 5.82 para $\lambda = 64$. Valores bem superiores aos testes executados exclusivamente na CPU.

Porém para a placa GTS-450, os ganhos apresentados não superaram o valor 2.84. Na tabela 6.1 é possível observar que para essa placa a taxa de transferencia de dados para a memória é bem inferior às taxas dos outros dois hardwares. Essa característica influenciou diretamente no resultado, pois a implementação do algoritmo na GPU envolve cópias constantes entre a memória da placa e a memória principal do computador.

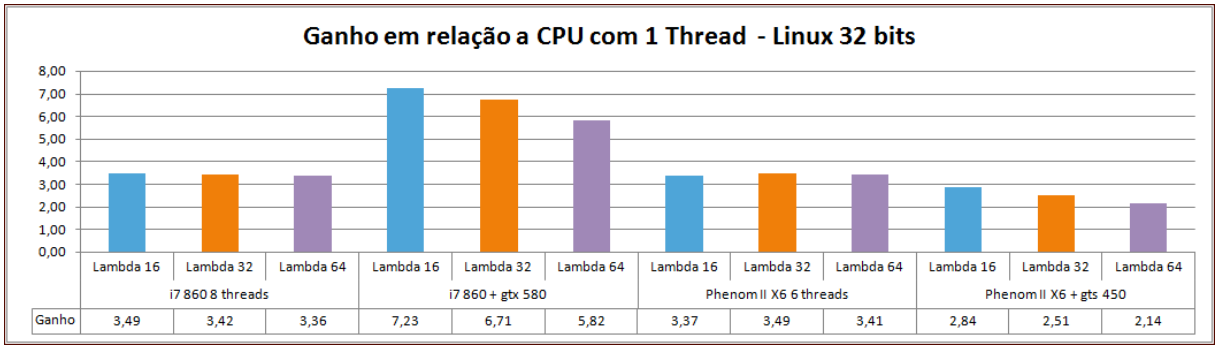


Figura 6.9: Ganho - Linux 32 bits

6.4 Simulação com duas instâncias simultâneas

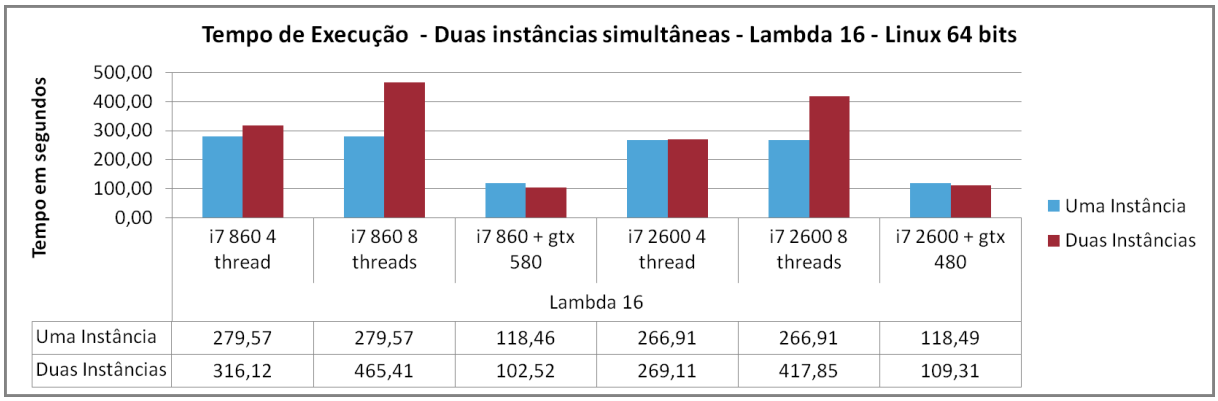
Considerando que às vezes pode surgir a necessidade de que o hardware processe mais de uma informação por vez, os testes da CPU com múltiplos *threads* e da GPU foram refeitos, porém com duas instâncias do MMP sendo executadas ao mesmo tempo.

Além da simulação com o máximo de *threads* suportado pela CPU, foram realizados testes com a CPU sendo executada com metade dessa capacidade.

No Apêndice C encontram-se todos os resultados dos testes e a partir dos valores das duas instâncias foi feita a média e o valor dividido por 2. Desta forma obtém-se o tempo de processamento de uma única imagem e o comparamos com os resultados encontrados anteriormente.

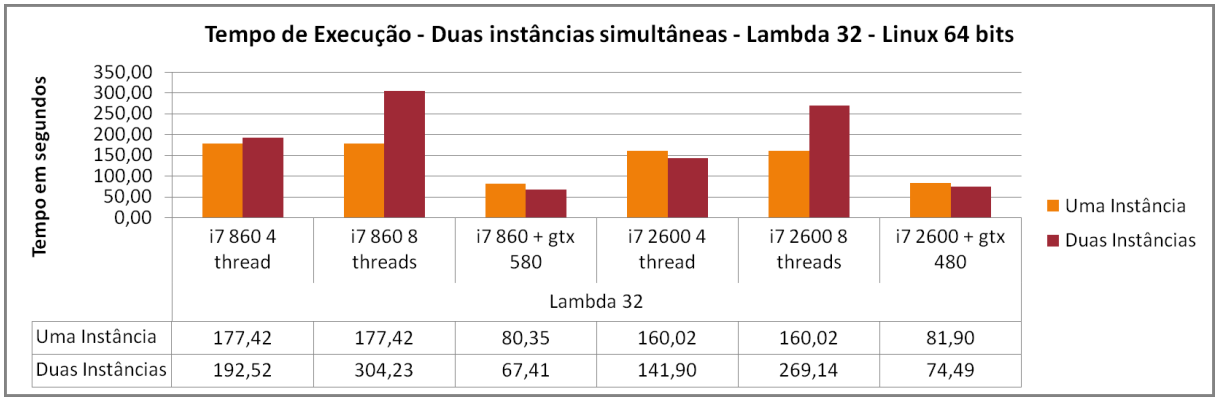
6.4.1 Resultados para 64 bits

Na figura 6.10 é possível observar o resultado de $\lambda = 16$ para uma e duas instâncias.



	i7 860 4 thread	i7 860 8 threads	i7 860 + gtx 580	i7 2600 4 thread	i7 2600 8 threads	i7 2600 + gtx 480
Uma Instância	279,57	279,57	118,46	266,91	266,91	118,49
Duas Instâncias	316,12	465,41	102,52	269,11	417,85	109,31

Figura 6.10: Lambda 16 - Duas instâncias simultâneas - 64 bits



	i7 860 4 thread	i7 860 8 threads	i7 860 + gtx 580	i7 2600 4 thread	i7 2600 8 threads	i7 2600 + gtx 480
Uma Instância	177,42	177,42	80,35	160,02	160,02	81,90
Duas Instâncias	192,52	304,23	67,41	141,90	269,14	74,49

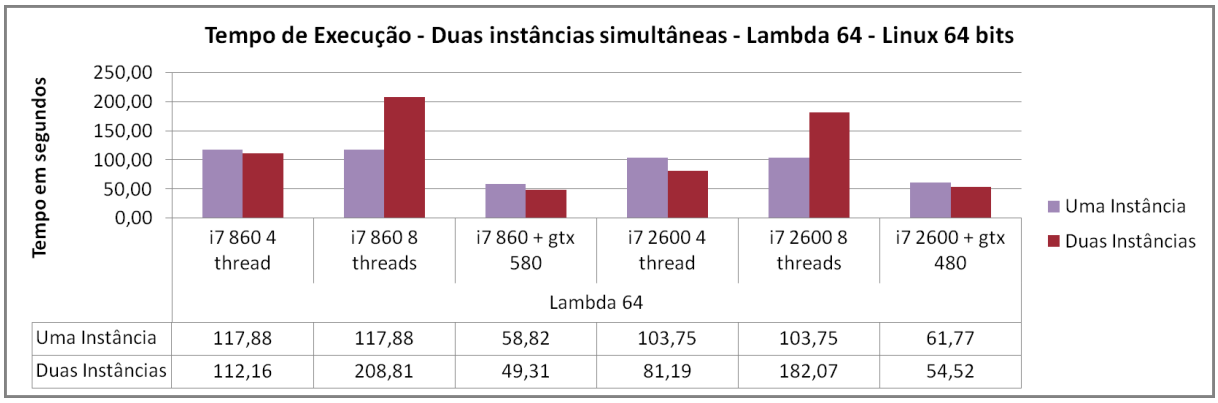
Figura 6.11: Lambda 32 - Duas instâncias simultâneas - 64 bits

A CPU i7 860 com 8 *threads* obteve uma degradação considerável no tempo de 465.41 segundos quando comparamos com os 279.57 segundos do mesmo teste, porém com uma instância. Com 4 *threads*, o tempo encontrado foi de 316.12 segundos, o que ainda é superior ao tempo de executar a codificação em multithread com apenas uma instância.

O mesmo ocorre para o i7 2600 que apresentou tempos de execução superiores com duas instâncias quando comparados aos valores encontrados na execução com apenas uma.

Se analisarmos as execuções em conjunto com a GPU, tanto o par i7 860 e GTX-580 quanto o i7 2600 e GTX-480, não apresentaram queda no desempenho, pelo contrário, os tempos foram melhores em aproximadamente 8%. Isto nos leva a crer que a GPU ainda está sendo subutilizada. A medida que mais funções do MMP sejam portadas para a GPU, os ganhos tendem a melhorar ainda mais.

Para $\lambda = 32$, os resultados estão resumidos na figura 6.11. O comportamento foi bastante semelhante ao resultado com $\lambda = 16$, com a diferença de que o i7 2600 sendo executado com 4 *threads* obteve um resultado melhor com duas instâncias, pois o tempo foi de 141.90 segundos.



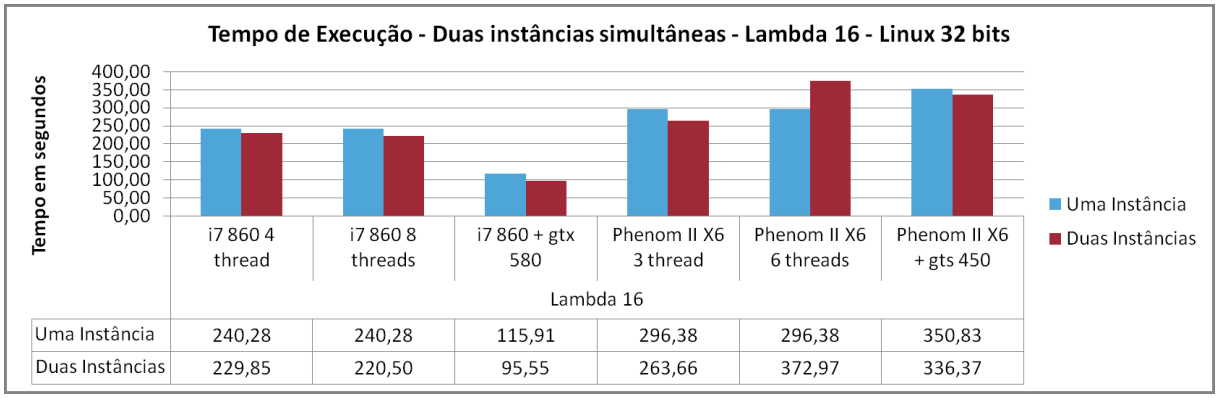
Uma Instância	117,88	117,88	58,82	103,75	103,75	61,77
Duas Instâncias	112,16	208,81	49,31	81,19	182,07	54,52

Figura 6.12: Lambda 64 - Duas instâncias simultâneas - 64 bits

Com $\lambda = 64$, ambas CPUs (i7 860 e i7 2600) obtiveram tempos com 8 *threads* muito superiores aos resultados com apenas uma instância. O aumento foi de 77.13% para o i7 860 e de 75.45% para o i7 2600. Esses valores podem ser visualizados na figura 6.12. Assim como a melhoria dos tempos para 4 *threads* e com a GPU para ambos os casos.

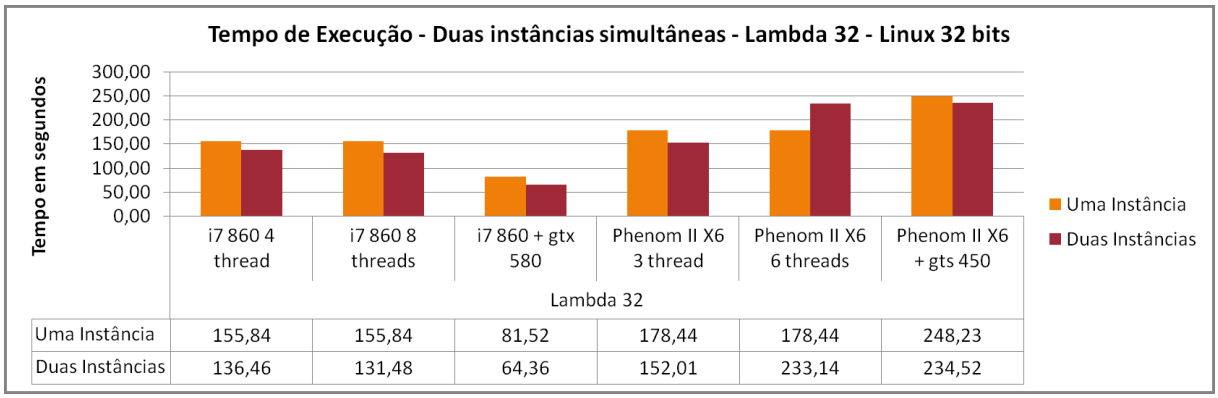
6.4.2 Resultados para 32bits

Nos testes com o sistema operacional de 32 bits, a CPU i7 860 apresentou tempos menores quando executada para duas instâncias, tanto para 4 *threads* quanto para 8. Esse comportamento pode ser observado para os três valores de λ (16, 32 e 64) nas figuras 6.13, 6.14 e 6.15.



	i7 860 4 thread	i7 860 8 threads	i7 860 + gtx 580	Phenom II X6 3 thread	Phenom II X6 6 threads	Phenom II X6 + gts 450
Uma Instância	240,28	240,28	115,91	296,38	296,38	350,83
Duas Instâncias	229,85	220,50	95,55	263,66	372,97	336,37

Figura 6.13: Lambda 16 - Duas instâncias simultâneas - 32 bits



	i7 860 4 thread	i7 860 8 threads	i7 860 + gtx 580	Phenom II X6 3 thread	Phenom II X6 6 threads	Phenom II X6 + gts 450
Uma Instância	155,84	155,84	81,52	178,44	178,44	248,23
Duas Instâncias	136,46	131,48	64,36	152,01	233,14	234,52

Figura 6.14: Lambda 32 - Duas instâncias simultâneas - 32 bits

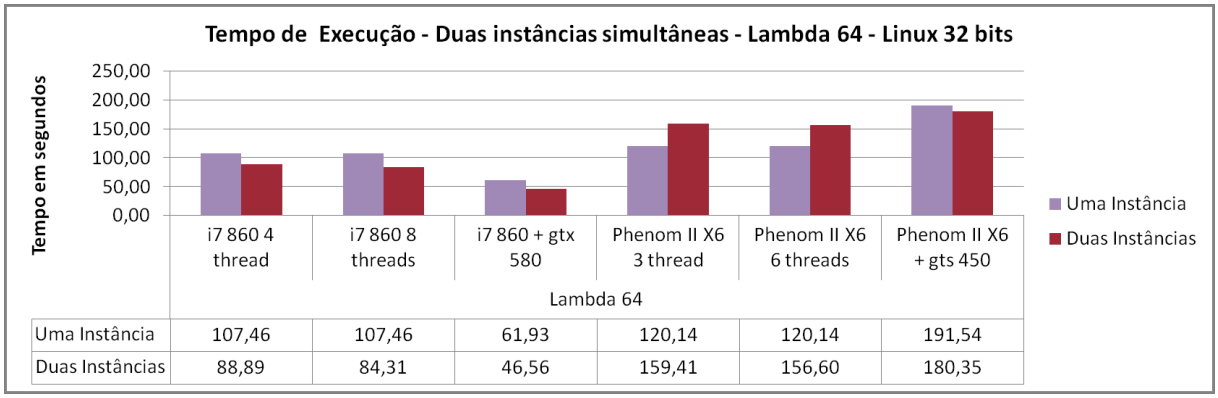
Já o Phenom II X6, apresentou um comportamento inverso, pois os valores encontrados são mais elevados para 6 *threads* com duas instâncias para todos os valores de λ. A execução com três *threads* mostrou-se com um desempenho pior somente para λ com valor de 64.

Assim como para 64 bits, as duas GPUs com o sistema operacional de 32 bits, apresentaram um desempenho melhor ao serem testadas com duas instâncias.

6.5 Opções de otimização dos compiladores

O gcc fornece uma ampla gama de opções que visam aumentar a velocidade de execução, ou reduzir o tamanho, dos arquivos executáveis que ele gera.

Otimização é um processo complexo e há várias combinações possíveis de instruções de máquina que podem ser usadas para conseguir o resultado final adequado. O compilador deve considerar essas possibilidades e escolher dentre elas. Para o gcc e o



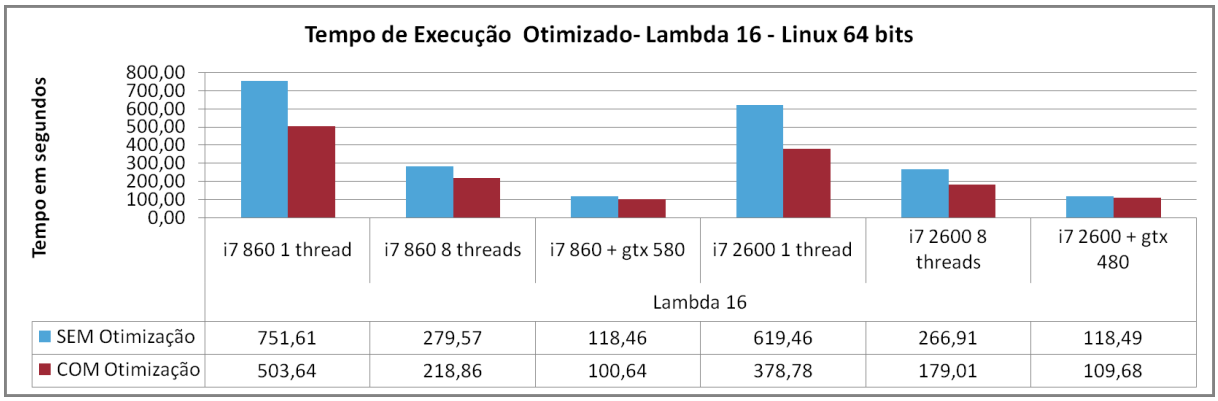
	i7 860 4 thread	i7 860 8 threads	i7 860 + gtx 580	Phenom II X6 3 thread	Phenom II X6 6 threads	Phenom II X6 + gts 450
Uma Instância	107,46	107,46	61,93	120,14	120,14	191,54
Duas Instâncias	88,89	84,31	46,56	159,41	156,60	180,35

Figura 6.15: Lambda 64 - Duas instâncias simultâneas - 32 bits

nvcc testamos uma opção de compilação que pode aumentar a velocidade do executável resultante: $-O3$. Além dessa foi utilizada também para o gcc a opção $-funroll-loops$ que otimiza a execução dos loops.

Com essas considerações, recompilamos os códigos e refizemos os testes. Os resultados estão representados no Apêndice D.

6.5.1 Resultados para 64 bits



	i7 860 1 thread	i7 860 8 threads	i7 860 + gtx 580	i7 2600 1 thread	i7 2600 8 threads	i7 2600 + gtx 480
SEM Otimização	751,61	279,57	118,46	619,46	266,91	118,49
COM Otimização	503,64	218,86	100,64	378,78	179,01	109,68

Figura 6.16: Tempos de execução COM e SEM Otimização para $\lambda = 16$

A otimização empregada pelos compiladores gerou tempos que podem ser observados nas figuras 6.16, 6.17 e 6.17. Para todos os hardwares é possível notar que houve uma melhoria nos tempos de execução.

Os ganhos em percentual com a utilização das opções de otimização podem ser observadas na tabela 6.4. Nela é possível verificar que para a GPU esse ganho é menos significativo do que para a CPU. Enquanto o i7 860 com 1 *thread* apresenta um ganho

de mais de 30% o par i7 860 e GTX-580 apresenta um ganho máximo de 15.04%.

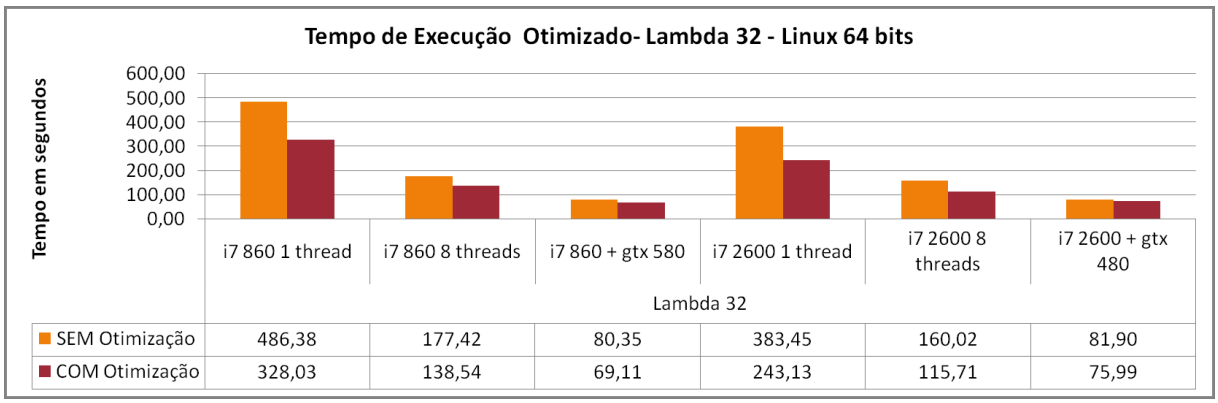


Figura 6.17: Tempos de execução COM e SEM Otimização para $\lambda = 32$

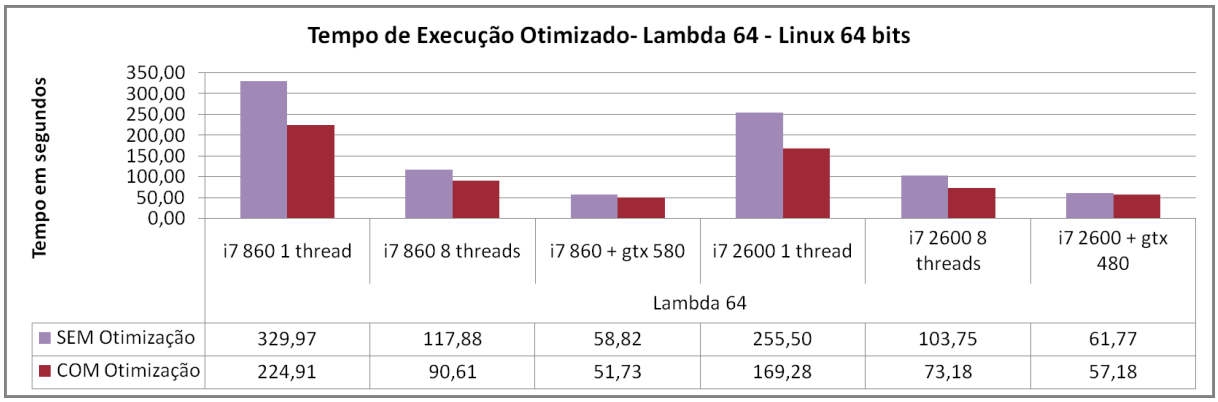


Figura 6.18: Tempos de execução COM e SEM Otimização para $\lambda = 64$

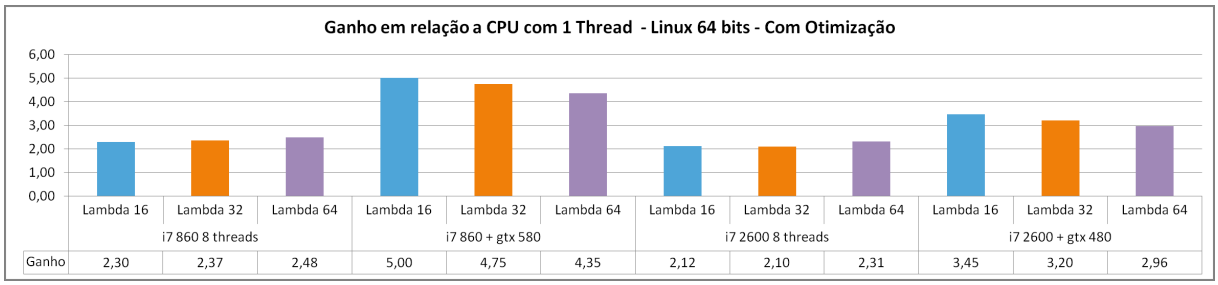


Figura 6.19: Ganho - Linux 64 bits - COM Otimização

6.5.2 Resultados para 32 bits

Para as configurações do i7 860 com GTX-580 e o Phenom II X6 com a GTS-450 em Linux 32 bits, obtivemos um resultado semelhante de ganho quando obsevamos os testes em 64 bits.

Hardware	Lambda	Ganho
i7 860 1 threads	Lambda 16	32,99%
	Lambda 32	32,56%
	Lambda 64	31,84%
i7 860 8 threads	Lambda 16	21,72%
	Lambda 32	21,92%
	Lambda 64	23,14%
i7 860 + gtx 580	Lambda 16	15,04%
	Lambda 32	13,99%
	Lambda 64	12,06%
i7 2600 1 threads	Lambda 16	38,85%
	Lambda 32	36,60%
	Lambda 64	33,75%
i7 2600 8 threads	Lambda 16	32,93%
	Lambda 32	27,69%
	Lambda 64	29,46%
i7 2600 + gtx 480	Lambda 16	7,43%
	Lambda 32	7,21%
	Lambda 64	7,43%

Tabela 6.4: Ganho em % dos tempos de execução COM Otimização

Na figura 6.20 verificamos os valores dos testes com e sem otimização para $\lambda = 16$, assim como na figura 6.21 e 6.22 para valores de λ de 32 e 64, respectivamente. Para todas as configurações observou-se um ganho quanto ao uso das opções de otimização, ou seja, os tempos de execução foram inferiores para todos os novos testes.

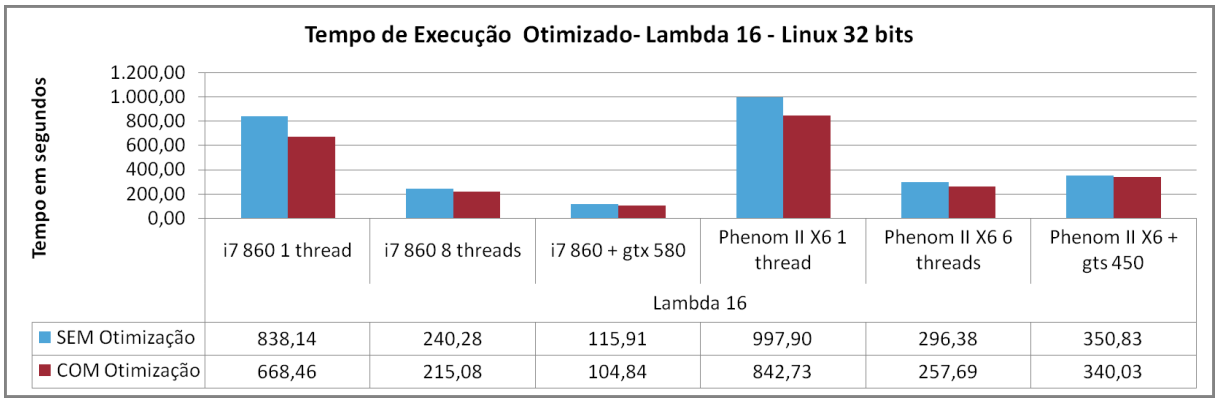


Figura 6.20: Tempos de execução COM e SEM Otimização para $\lambda = 16$

Na tabela 6.5 temos em percentual os ganhos apresentados para cada uma das configurações acima definidas. Nela observamos o maior ganho de 20.24% para o i7 860 sendo executado com 1 *thread* com λ valendo 16 e o menor ganho para o par Phenom II X6 com GTS 450 e λ valendo 64, que obteve um ganho de apenas 2.35%.

Na figura 6.23 observamos os ganhos para o i7 860 executando com 1 *thread*, 8 *threads* ou em conjunto com a GPU GTX-580. Analisando os dados, verificamos que

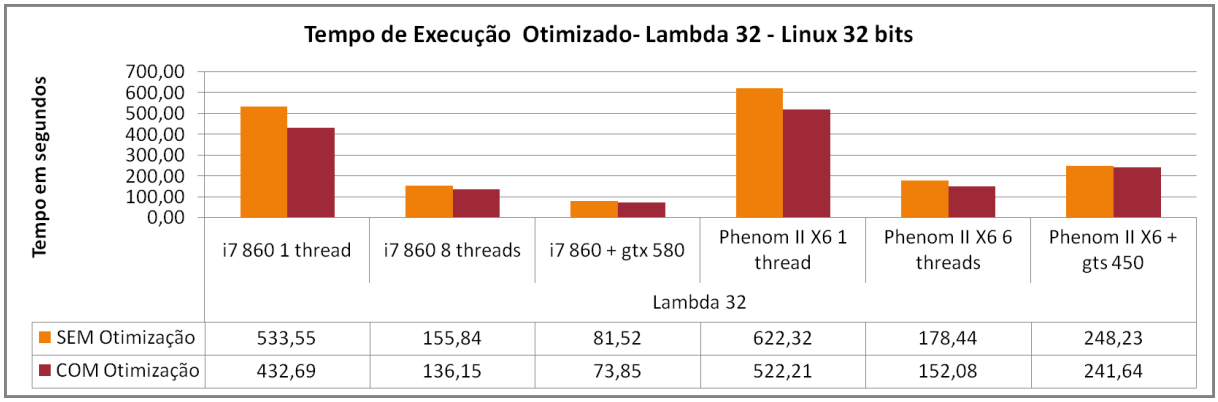


Figura 6.21: Tempos de execução COM e SEM Otimização para $\lambda = 32$

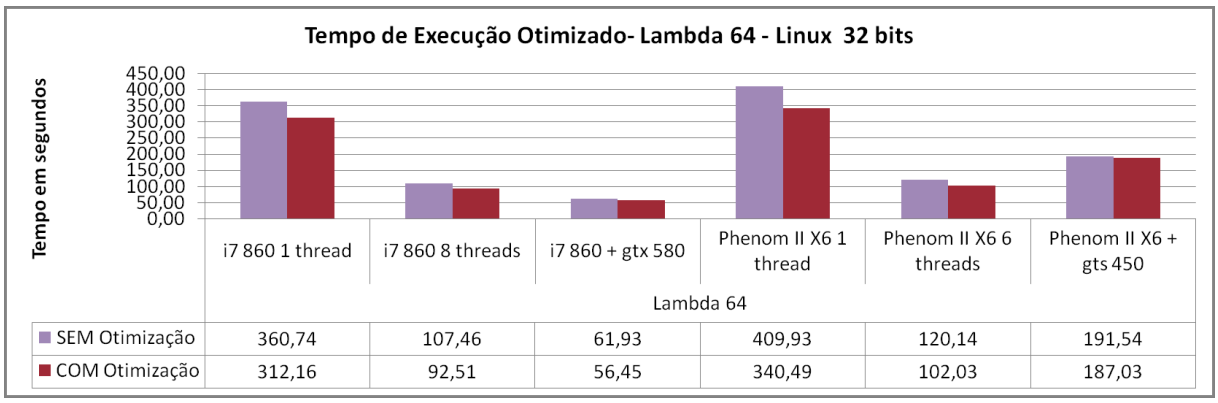


Figura 6.22: Tempos de execução COM e SEM Otimização para $\lambda = 64$

Hardware	Lambda	Ganho
i7 860 1 threads	Lambda 16	20,24%
	Lambda 32	18,90%
	Lambda 64	13,47%
i7 860 8 threads	Lambda 16	10,49%
	Lambda 32	12,64%
	Lambda 64	13,91%
i7 2600 + gtx 480	Lambda 16	9,55%
	Lambda 32	9,41%
	Lambda 64	8,85%
Phenom II X6 1 thread	Lambda 16	15,55%
	Lambda 32	16,09%
	Lambda 64	16,94%
Phenom II X6 6 threads	Lambda 16	13,06%
	Lambda 32	14,78%
	Lambda 64	15,08%
Phenom II X6 + gts 450	Lambda 16	3,08%
	Lambda 32	2,66%
	Lambda 64	2,35%

Tabela 6.5: Ganho em % dos tempos de execução COM Otimização

para as execuções exclusivamente na CPU, todos os ganhos foram superiores a 10% e as execuções CPU com GPU o maior ganho foi de 9.55% para λ de 16.

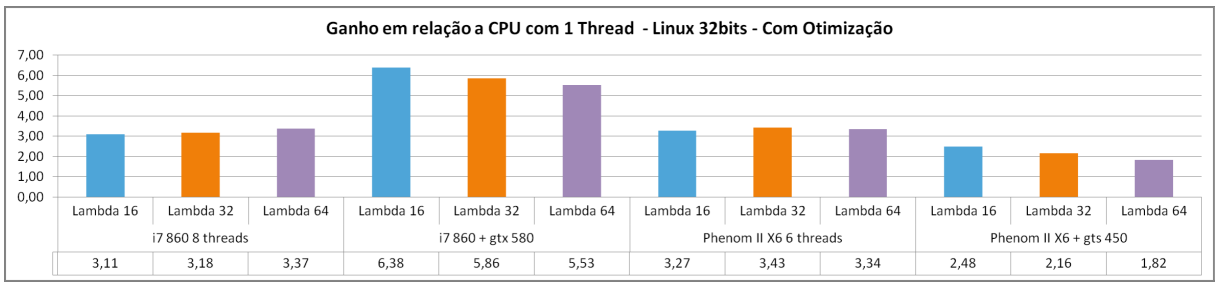


Figura 6.23: Ganho - Linux 32 bits - COM Otimização

Importante notarmos que apesar dos ganhos superiores para as execuções exclusivas na CPU, esses ganhos não foram suficientes para que a instância da CPU i7 860 com 8 *threads* superasse o desempenho da execução em conjunto com a GPU.

6.6 Análise da qualidade da compressão

Devido ao fato de a versão do código executada na GPU não mais fazer uso da frequência relativa de cada elemento do dicionário para o valor de r no cálculo do custo Lagrangeno é necessário verificar se a execução pela GPU de parte do algoritmo com a adaptação do cálculo da taxa gera imagens comprimidas com PSNR e taxa de bits coerentes. O novo valor de r é calculado a partir de uma aproximação dada por:

$$r = \log_2(\texttt{tamanhoDic}) \tag{6.1}$$

As curvas de desempenho para o MMP executando na CPU ou na GPU estão representadas na figura 6.24. Para a confecção deste gráfico foram utilizados pontos para valores de λ adicionais 128, 64, 32, 24, 16 e 10.

Com base na análise do gráfico podemos dizer que as compressões são equivalentes, pois as curvas estão quase sobrepostas, ou seja, os valores para a taxa de transmissão e PSNR estão bem próximos.

Desta forma, fica ainda mais claro que as alterações realizadas no código para que a execução fosse realizada em menos tempo foram válidas.

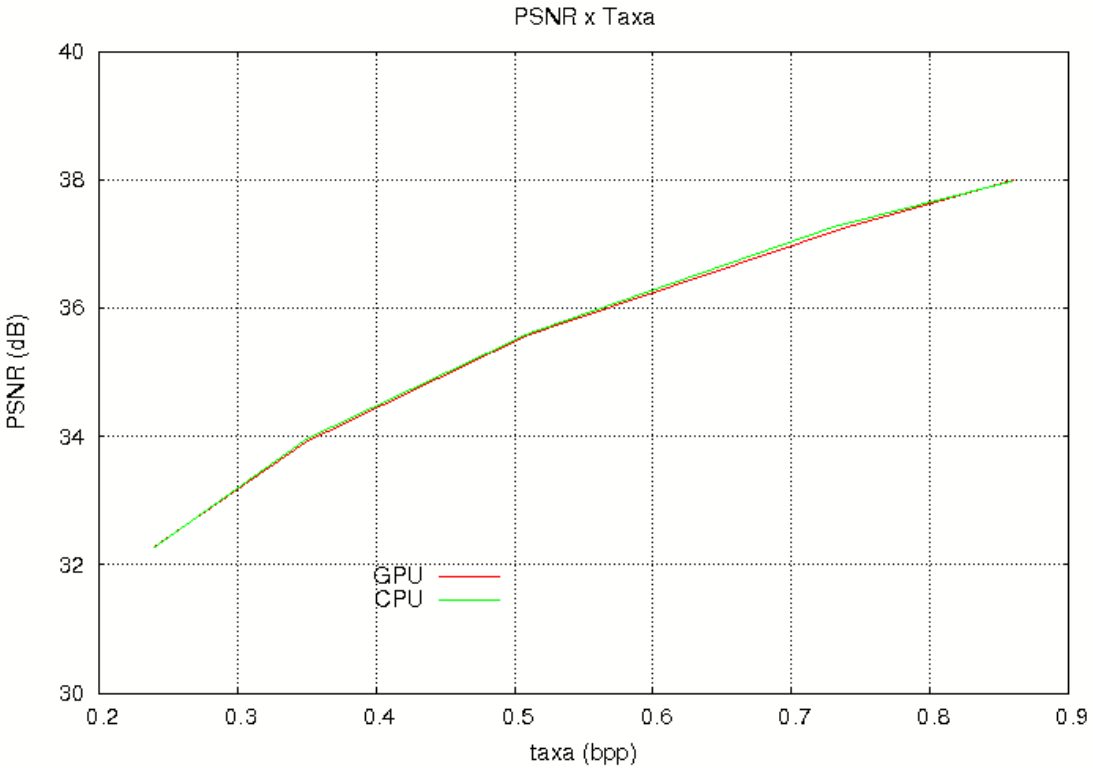


Figura 6.24: PSNR x Taxa - GPU e CPU

Capítulo 7

Conclusão

Implementar na GPU parte do código do MMP envolveu um entendimento do problema independente da forma de como o código já estava implementado na CPU. Isso significa que não basta reescrever cada loop em cuda C para que seja eficaz essa transformação. Para que se possa fazer uso da GPU utilizando seus benefícios do paralelismo é preciso entender o problema e realizar o mapeamento desse cenário para as entidades envolvidas (blocos e *threads*). Esse mapeamento envolve abstração e entendimento das limitações dos hardwares disponíveis. Por exemplo, uma das ideias iniciais era fazer com que cada *thread* ficasse responsável por um pixel. Porém em nenhum dos kernels foi possível executar devido ao fato dos dicionários serem grandes e envolverem muitos elementos, inviabilizando dessa forma a impementação, pois existe uma limitação para número de *threads* máximo permitido.

Uma outra dificuldade está no fato de o nvcc não apresentar mensagens de erros que ajudam o desenvolvedor na solução dos problemas. O *kernel* ser executado não significa que ele não possui erros, pois pode apresentar resultados incoerentes e a busca pelo problema não é um tarefa trivial.

Depois de vencidos os obstáculos de implementação, a partir dos testes realizados ficou bem aparente a eficiência das execuções nas GPUs. A única placa que não apresentou resultado satisfatório foi a GTS-540 e esse desempenho se deu devido ao fato de a taxa de transferência de memória dela ser muito inferior às demais bem com ao número de unidades de processamento ser menor.

Para as placas GTX-480 e GTX-580, os ganhos foram significativos. Considerando-se o gráfico da figura 7.1,que representa os tempos da CPU i7 860 e da placa GTX-580

para 32 bits, pode-se observar a têndencia de crescimento mais rápido das CPUs do tempo de execução quando se diminui o λ e consequentemente se aumentam os dicionários.

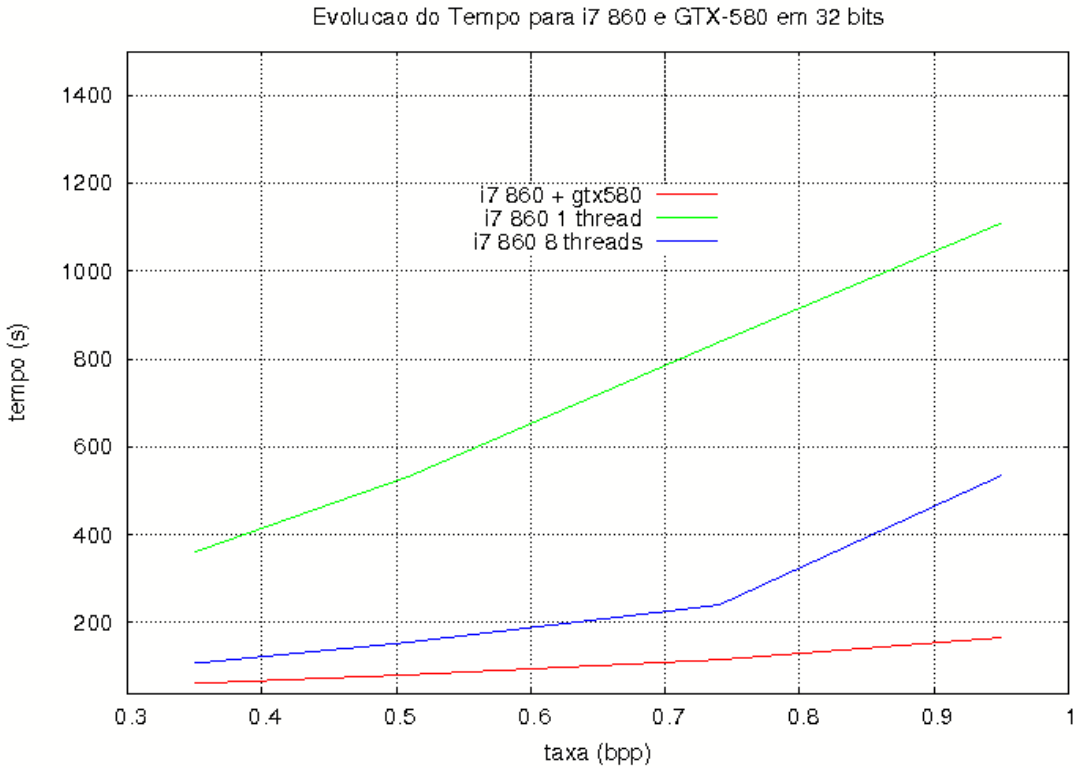


Figura 7.1: Evolução dos Tempos GPU x CPU - i7 860 32 bits e GTX-580

Essa tendência mostra que a utilização da GPU desempenha muito bem o papel de executar funções custosas como o MMP. A pesquisa nessa vertente é bastante promissora.

Se considerarmos a GPU GTS-450 que obteve o desempenho menos eficiente, podemos observar no gráfico da figura 7.2 que mesmo esta GPU poderia superar o tempo da CPU multithread para valores de lambda pequeno.

Além do desempenho excelente das GPUs, foi possível observar que as CPUs quando executadas em multitread conseguem também gerar uma melhoria nos tempos. Vale ressaltar que nenhum ganho apresentado por elas alcançou os valores de ganho das GPUs.

Os resultados das simulações com duas instâncias sugerem que a GPU ganhos maiores podem ser obtidos e que a GPU está sub-utilizada, pois com duas instâncias apresentou um tempo mais rápido de execução. Estes mesmos resultados sugerem que a implementação multithread na CPU por outro lado já se encontra próxima ao seu limite de desempenho, pois o desempenho com duas instâncias cai consideravelmente nos casos

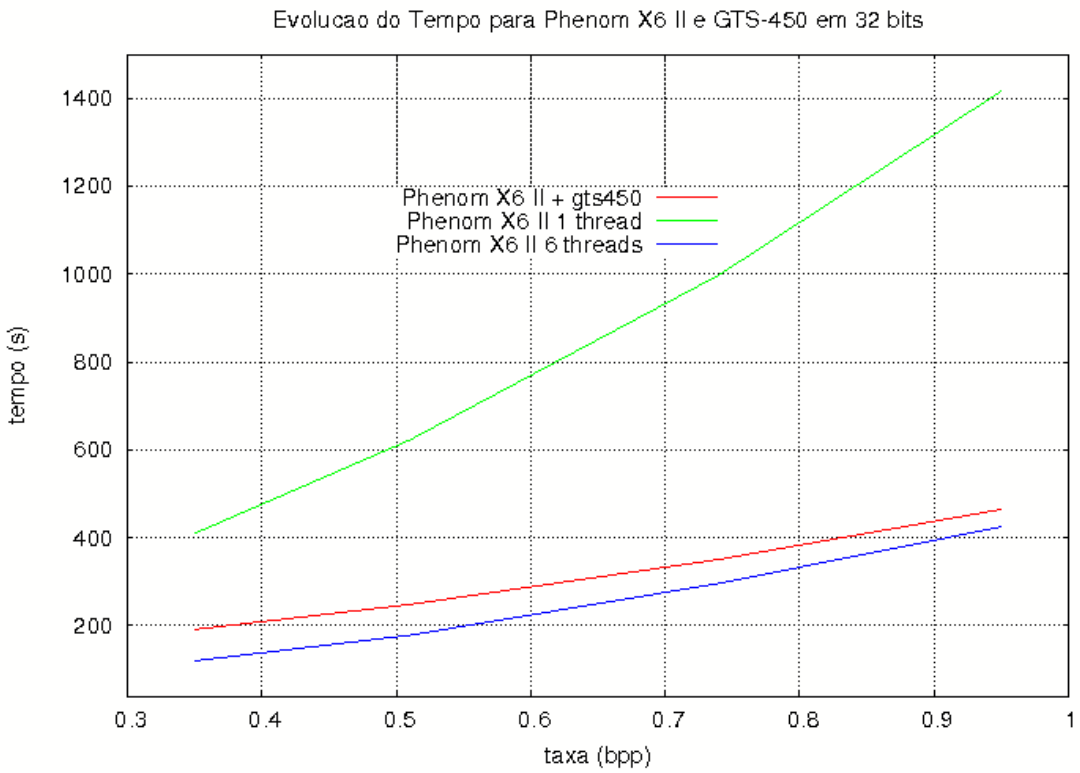


Figura 7.2: Evolução dos Tempos GPU x CPU - Phenom X6 II e GTS-450

de 8 e 6 *threads* simultâneos por instância.

64 bits	16	32	64
i7 860 + gtx 580	118,46 s	80,35 s	58,82 s
i7 2600 + gtx 480	118,49 s	81,90 s	61,77 s
Relação entre os tempos	1,00	1,02	1,05

Tabela 7.1: Relação dos tempos entre GPUs - 64 bits

32 bits	16	32	64
i7 860 + gtx 580	115,91 s	81,52 s	61,93 s
Phenom II X6 + gts 450	350,83 s	248,23 s	191,54 s
Relação entre os tempos	3,03	3,04	3,09

Tabela 7.2: Relação dos tempos entre GPUs - 32 bits

Verificou-se também que os tempos de cada uma das configurações com GPU apresentou um tempo de execução que está diretamente relacionado à quantidade de Cuda Cores de cada uma das placas. Nas tabelas 7.1 e 7.2 é possível observar uma relação entre os tempos das configurações para 64 bits e 32 bits. Podemos verificar que para os testes em 32 bits, a relação entre os tempos possui um valor na faixa de 3 vezes menor para a

placa GTX-580. Comparando a quantidade de Cuda Cores de ambas as placas (512 e 192 da coluna "Pixels por Clock"da tabela 6.1) essa relação é de 2.7 . Analogamente, para as placas utilizadas em 64 bits, a relação entre os tempos são valores próximos a 1, assim como a relação entre as quantidades de Cuda Cores das placas GTX-580 e GTX-480. Essa análise nos faz concluir que o tempo de execução está diretamente relacionado à quantidade de Cuda Cores existentes na placa.

i7 860 + gtx 540 64 bits		Kernel CalculaMapa, ReduzMapa e ReduzMapa2	Kernel Compara	Insere os novos elementos no Dicionário da GPU	Tempo de tranferência de dados entre CPU e GPU	Tempo de Execução na CPU	Tempo Total (s)
Lambda 16	tempo (s)	45,79	33,57	27,52	1,04	11,48	119,42
	tempo (%)	38,35%	28,11%	23,05%	0,88%	9,62%	
Lambda 32	tempo (s)	36,65	16,55	19,70	0,88	9,27	83,05
	tempo (%)	44,13%	19,93%	23,72%	1,05%	11,16%	
Lambda 64	tempo (s)	30,42	8,36	12,91	0,35	7,78	59,83
	tempo (%)	50,85%	13,97%	21,58%	0,59%	13,01%	

Tabela 7.3: Tempos com temporizador - i7 860 + gtx 580

Além disso, para analisar o tempo que foi gasto em cada etapa, fizemos uma simulação com as funções `cutStartTimer` e `cutStopTimer` do Cuda para a configuração i7 860 + gtx 580 em 64 bits. Os tempos dos testes aumentaram em mais ou menos 1 segundo fazendo uso dessas funções, mas com elas foi possível entender quanto tempo é consumido em cada uma das etapas do novo código proposto.

Na tabela 7.3 verificou-se que para λ=16 temos 66.46% do tempo de execução sendo utilizado pelos kernels CalculaMapa, ReduzMapa, ReduzMapa2 e Compara. Para λ de 32 e 64, a ordem de grandeza é semelhante, sendo esses valores de 64.06% e 64.82%, respectivamente. Como verificamos que o tempo gasto acompanha a quantidade de Cuda Cores existentes, analisando λ=16 e 66.46%, se supusermos uma GPU com infinitos núcleos, o tempo de execução seria apenas os tempos de cópia de dados e o tempo de execução do trecho na CPU, o que seria dado por

$$27.52 + 1.04 + 11.48 = 40.05s \tag{7.1}$$

Esse tempo de 40.05 segundos, em relação a execução com um *thread* em CPU (tempo de 751.61 segundos verificado na figura 6.2) faria com que o ganho que era anteriormente de 6.34 se tornasse igual a 18.56 vezes.

Por outro lado, caso migrássemos todo o código restante da CPU para a GPU, o tempo de 40.05 segundos gasto na CPU seria reduzido para 6.32 segundos na GPU, supondo-se que o ganho da GPU sobre a CPU permanecesse o mesmo. Nesse caso, o

tempo de execução total seria

$$45.79 + 33.57 + 6.32 = 85.68s \tag{7.2}$$

Com esse novo tempo, o ganho passaria a ser de 8.77 (também em relação ao tempo de 751.61 segundos para execução de um *thread* da figura 6.2).

Com essa análise, conseguimos concluir que migrar todo o código para a GPU elevaria um pouco o ganho que já foi obtido de 6.34 para 8.77, porém fazendo uso de uma GPU com o triplo ou quádruplo de Cuda Cores, o ganho pode ser bem maior. Para uma geForce gtx Titan que possui 2688 Cuda Cores, ou seja 5.25 mais núcleos que a gtx 580, podemos dizer que o tempo de execução seria aproximadamente

$$(45.79 + 33.57)/5.25 + 40.05 = 55.16s \tag{7.3}$$

O que geraria um ganho de 13.62. Ou seja, quanto maior a quantidade de Cuda Cores da placa, mais eficiente é sua a implementação.

Para trabalhos futuros, seria um tema proposto estudar o uso combinado do paralelismo na GPU e nos múltiplos núcleos da CPU. Além desse tema, a partir do fato de ter-se obtido um ganho na execução utilizando instâncias concorrentes, faz-se necessário um estudo mais detalhado de execução concorrente da CPU e GPU, bem como uma implementação com mais funções sendo executadas na GPU.

Apêndice A

Código funções CUDA

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <time.h>
5  #include <math.h>
6  #include "CalculaMapaGlobalCompara.h"
7
8  __global__ void calculaMapa(int blocoLin, int blocoCol, int *tamanho, float
       *dev_mapa, unsigned char *dev_elemento, unsigned char *dev_dicionario,
      int lambda);
9  __global__ void reduzMapa(float *mapa, unsigned short int *indice);
10 __global__ void reduzMapa2(float *mapa, unsigned short int *indice,
      unsigned char *dev_dicionario);
11 __global__ void comparaBloco(int *tamanho, unsigned char *dev_elemento,
      unsigned char *dev_dicionario, int TamListaBlocos, int *dev_resultado);
12
13 const int DimBloco = 31;
14 const int DimGrid = 128;
15
16 dim3 grid(DimGrid,DimGrid);
17 dim3 thread(DimBloco,DimBloco);
18 dim3 grid2 (DimBloco*8,DimBloco);
19 dim3 thread2(DimGrid/4,DimGrid/4);
20 dim3 grid3 (DimBloco,DimBloco);
21 dim3 thread3(2,2);
22
23 unsigned char *dev_elemento, *dev_dicionario, *dev_arrayElemento;
```

```
24 int *dev_status, *dev_resultado;
25 float *dev_mapa;
26 unsigned short int *dev_indice;
27 int *dev_tamanhoDic;
28
29 extern "C" void inicializaDic(unsigned char *h_biblioteca, int *tamanhoDic
       , int N , int M){
30       // Alocando a memoria na GPU
31       cudaMalloc((void**)&dev_elemento,M*N*sizeof(unsigned char)) ;
32       cudaMalloc((void**)&dev_dicionario,128*128*M*N*25*sizeof(unsigned char
             )) ;
33       cudaMalloc((void**)&dev_tamanhoDic, 25*sizeof(int)) ;
34       cudaMalloc((void**)&dev_mapa, 961*128*128*sizeof(float)) ;
35       cudaMalloc((void**)&dev_indice, 961*128*128*sizeof(unsigned short int)
             ) ;
36       cudaMalloc((void**)&dev_arrayElemento,16*16*25*sizeof(unsigned char));
37       cudaMalloc((void**)&dev_resultado, 25*sizeof(int));
38
39       // Copiando da CPU para a GPU
40        cudaMemcpy(dev_dicionario, h_biblioteca, 128*128*M*N*25*sizeof(
             unsigned char), cudaMemcpyHostToDevice );
41        cudaMemcpy(dev_tamanhoDic, tamanhoDic, 25*sizeof( int ),
             cudaMemcpyHostToDevice );
42 }
43
44 extern "C" void callCalculaMapa (unsigned char *elemento, float *mapa,
       unsigned short int *indice, int blocoLin, int blocoCol, int lambda) {
45
46       // Copiando da CPU para a GPU
47       cudaMemcpy(dev_elemento, elemento, blocoLin*blocoCol*sizeof(unsigned
             char), cudaMemcpyHostToDevice );
48
49       calculaMapa<<<grid,thread>>>(blocoLin, blocoCol, dev_tamanhoDic,
             dev_mapa, dev_elemento, dev_dicionario, lambda);
50
51       //observe a inversão de quantidade de threads e blocos
52       reduzMapa<<<grid2, thread2>>>(dev_mapa, dev_indice);
53
54       reduzMapa2<<<grid3, thread3>>>(dev_mapa, dev_indice, dev_dicionario )
```

```
        ;
55
56      // Copiando da GPU para a CPU
57      cudaMemcpy( indice, dev_indice, 961*sizeof(unsigned short int),
            cudaMemcpyDeviceToHost) ;
58      cudaMemcpy( mapa, dev_mapa, 961*sizeof(float), cudaMemcpyDeviceToHost
            ) ;
59  }
60
61  __global__ void calculaMapa( int blocoLin, int blocoCol, int *tamanho,
        float *mapa, unsigned char *dev_elemento, unsigned char *dev_dicionario,
         int lambda) {
62
63      //Para saber se o valor é válido depois
64      float J = -1, D =-1;
65
66      int idThread, elementoDic, m, n, k, l, i, j;
67      float r=0;
68
69      //Cada thread é responsavel por um J
70      idThread= threadIdx.x + threadIdx.y*blockDim.x;
71
72      //Cada bloco é responsável por um elemento do dicionário
73      elementoDic= blockIdx.x + blockIdx.y*gridDim.x;
74
75      // escala horizontal (partir na direção vertical, separando a direção
            horizontal em duas)
76      m = floor(__log2f(float(threadIdx.x+1)));
77
78      // escala vertical
79      n = floor(__log2f(float(threadIdx.y+1)));
80
81      //tem que ser preenchida de acordo
82      int escala = m+n*5;
83
84      // determina a quantidade de subblocos o bloco possui em cada escala
85      k = 0x1 << (m);
86      l = 0x1 << (n);
87
```

```
88          if (idThread <961 ){
89
90            int qtd = tamanho[escala];
91            r=__log2f(float(tamanho[escala]));
92
93          if (qtd>= elementoDic){
94              D=0;
95
96            // Mapeamento de cada bloco a um elemento no dicionário
97            // Mudança para evitar a concorrência de leitura do mesmo bloco do
                  dicionário
98            // Valores repetidos em números de subblocos
99              int xDic = (blockIdx.x)*(blocoCol);
100             int yDic = (blockIdx.y)*(blocoLin);
101
102             //Mapeamento de cada thread a um subbloco a ser comparado com o
                    elemento do dicionário determinado pelo blockidx
103             int xElemento = (threadIdx.x-(k-1))*(blocoCol/k);
104             int yElemento = (threadIdx.y-(l-1))*(blocoLin/l);
105
106             float blocoLido = 0;
107             float elementoDicLido = 0;
108             for(i = 0; i <blocoCol/k; i++) {
109                 for(j = 0; j <blocoLin/l; j++) {
110                   blocoLido=dev_elemento[(yElemento+j)*blocoCol+xElemento+i];
111                   elementoDicLido=dev_dicionario[escala*(128*blocoLin)*(128*
                          blocoCol)+ (yDic)*blocoCol*128 + (xDic)*blocoLin + (
                          yElemento+j)*blocoCol + xElemento+i];
112                  D=D+(blocoLido-elementoDicLido)*(blocoLido-elementoDicLido)
                          ;
113                 }
114               }
115               J=D+lambda*r;
116         }
117         mapa[elementoDic*961+idThread]=J;
118     }
119 }
120
121 __global__ void reduzMapa( float *mapa, unsigned short int *indice ) {
```

```
122
123         // Redução para pegar o J min da mesma escala e o índice do
                Dicionário escolhido
124
125         //cada bloco do grid vai representar um subbloco de uma escala do
                elemento(total de 961)
126        int idThread =  blockIdx.x + blockIdx.y*gridDim.x;
127        int subBloco=(idThread)%961;
128        int n=(idThread/961);
129
130         //cada thread compara 2 J para o mesmo subBloco e elementos do
                dicionário diferentes
131        int indiceElemento = (threadIdx.x + threadIdx.y*blockDim.x);
132
133        int i = (blockDim.x*blockDim.y); //eh a quantidade de threads
134        //mesmo assim só analisei 1/8 do dicionário
135
136        indice[(indiceElemento + n*i*2)*961+subBloco]=indiceElemento + n*i*2;
137        indice[(indiceElemento + n*i*2 + i)*961+subBloco]=indiceElemento + n*
               i*2 + i;
138
139        while (i != 0) {
140
141                if ((indiceElemento)< i && subBloco <961 && mapa[(
                       indiceElemento + n*1024*2 + i)*961+subBloco]>-1) {
142
143                  //Preenchimento do mapa foi feito com 961 valores para
                         cada elemento do dicionário para cada escala
144                  if(mapa[(indiceElemento + n*1024*2)*961+subBloco] > mapa
                         [(indiceElemento + n*1024*2 + i)*961+subBloco]){
145
146                     mapa[(indiceElemento + n*1024*2)*961+subBloco] = mapa
                            [(indiceElemento + n*1024*2 + i)*961+subBloco];
147                     indice[(indiceElemento + n*1024*2)*961+subBloco] =
                            indice[(indiceElemento + n*1024*2 + i)*961+subBloco
                            ];
148                  }
149                }
150                __syncthreads();
```

```
151                     i /= 2;
152             }
153 }
154
155 __global__ void reduzMapa2( float *mapa, unsigned short int *indice,
        unsigned char *dev_dicionario) {
156         // Redução para pegar o J min da mesma escala e o índice do
                Dicionário escolhido
157         //cada bloco do grid vai representar um subbloco de uma escala do
                elemento(total de 961)
158         int subBloco =  blockIdx.x + blockIdx.y*gridDim.x;
159
160         //cada thread compara 2 J para o mesmo subBloco e elementos do
                dicionário diferentes
161         int indiceElemento = threadIdx.x + threadIdx.y*blockDim.x;
162
163         int i = (blockDim.x*blockDim.y); //eh a quantidade de threads
164         //mesmo assim só analisei 1/8 do dicionário
165
166           while (i != 0) {
167
168                     if (indiceElemento< i && subBloco <961 && mapa[(
                            indiceElemento*2048 + i*2048)*961+subBloco]>-1) {
169
170                       //Preenchimento do mapa foi feito com 961 valores para
                            cada elemento do dicionário para cada escala
171                       if(mapa[(indiceElemento*2048)*961+subBloco] > mapa[(
                            indiceElemento*2048 + i*2048)*961+subBloco]){
172
173                          mapa[(indiceElemento*2048)*961+subBloco] = mapa[(
                               indiceElemento*2048 + i*2048)*961+subBloco];
174                          indice[(indiceElemento*2048)*961+subBloco] = indice[(
                               indiceElemento*2048 + i*2048)*961+subBloco];
175                       }
176                     }
177                     __syncthreads();
178                     i /= 2;
179           }
180 }
```

```
181
182
183 extern "C" void callInsereDic(unsigned char *arrayElemento,  int *
      tamanhoDic, int TamListaBlocos, int Lin , int Col, int escala, int *
      status){
184
185   if (TamListaBlocos >0 ) {
186
187       dim3 blocksPerGrid(64,64);
188       dim3 threadsPerBlock(10,10);
189
190      cudaMemcpy(dev_arrayElemento, arrayElemento, 16*16*25*sizeof(unsigned
           char), cudaMemcpyHostToDevice);
191      cudaMemcpy(dev_resultado, status, 25*sizeof(int),
           cudaMemcpyHostToDevice);
192
193     comparaBloco<<<blocksPerGrid,threadsPerBlock>>>(dev_tamanhoDic,
         dev_arrayElemento, dev_dicionario, TamListaBlocos, dev_resultado);
194
195      cudaMemcpy(status, dev_resultado, 25*sizeof(int),
           cudaMemcpyDeviceToHost);
196
197     for (int m = 0; m < 25; m ++) {
198       if(status[m]==1){
199           cudaMemcpy(&dev_dicionario[m*(128*16)*(128*16) + (16*16)*(
                tamanhoDic[m]+1)], &dev_arrayElemento[m*16*16], 16*16*sizeof(
                unsigned char), cudaMemcpyDeviceToDevice );
200           tamanhoDic[m]= tamanhoDic[m]+1;
201           cudaMemcpy(&dev_tamanhoDic[m], &tamanhoDic[m], sizeof(int),
                cudaMemcpyHostToDevice);
202       }
203      }
204 }
205 }
206
207 __global__ void comparaBloco(int *tamanho, unsigned char *dev_elemento,
      unsigned char *dev_dicionario, int TamListaBlocos, int *dev_resultado) {
208
209       int D=1, escalaBloco, i, j;
```

```
210         int xElementoDic = 2*blockIdx.x + threadIdx.x/5;
211         int yElementoDic = 2*blockIdx.y + threadIdx.y/5;
212
213         escalaBloco = threadIdx.x%5 + 5 * (threadIdx.y%5);
214
215         if((yElementoDic*128+xElementoDic) <= tamanho[escalaBloco] &&
                escalaBloco < 25) {
216
217           float blocoLido = 0;
218           float elementoDicLido = 0;
219
220           for(i = 0; i <blocoCol/k; i++) {
221             for(j = 0; j <blocoLin/l; j++) {
222
223                   blocoLido=dev_elemento[(escalaBloco)*16*16 + j*16+i];
224                   elementoDicLido=dev_dicionario[escalaBloco*(128*16)
                          *(128*16)+ (yElementoDic*128 + xElementoDic)*16*16 + j
                          *16 + i];
225
226                   if(blocoLido != elementoDicLido) D = 0;
227               }
228             }
229             if(D==1){
230                   dev_resultado[escalaBloco]=0;
231             }
232         }
233 }
```

Apêndice B

Resultados das Simulações

64 bits	i7 860 + gtx 580	i7 860 - 1 thread	i7 860 - 8 thread
Lamba 16	110,22	751,29	278,85
	118,71	750,80	278,67
	127,83	750,93	279,01
	118,29	750,97	279,63
	118,04	752,96	280,68
	118,47	751,43	279,52
	118,25	750,33	279,76
	118,26	752,54	280,50
	118,28	753,68	280,31
	118,30	751,13	278,81

Tabela B.1: Resultados para lambda 16 - Processador i7 860 + GTX-580 - 64 bits

64 bits	i7 860 + gtx 580	i7 860 - 1 thread	i7 860 - 8 thread
Lamba 32	80,20	482,96	176,45
	80,21	487,71	176,69
	80,18	486,42	177,90
	80,18	486,77	177,29
	80,16	486,34	177,41
	80,24	486,73	177,67
	80,22	486,13	177,30
	80,83	486,88	177,91
	80,27	486,72	178,10
	80,98	487,11	177,47

Tabela B.2: Resultados para lambda 32 - Processador i7 860 + GTX-580 - 64 bits

64 bits	i7 860 + gtx 580	i7 860 - 1 thread	i7 860 - 8 thread
Lamba 64	60,05	326,27	114,11
	58,22	327,20	115,78
	58,37	328,95	116,72
	57,99	331,01	117,50
	54,14	330,76	118,05
	59,79	330,80	118,77
	59,76	330,71	118,87
	59,78	331,64	119,78
	59,82	330,75	119,77
	60,32	331,55	119,49

Tabela B.3: Resultados para lambda 64 - Processador i7 860 + GTX-580 - 64 bits

64 bits	i7 2600 + gtx 480	i7 2600 - 1 thread	i7 2600 - 8 thread
Lamba 16	118,50	622,80	287,35
	118,20	618,93	264,03
	118,08	617,99	282,98
	119,13	620,16	257,15
	118,65	617,63	268,92
	118,08	617,31	239,04
	118,42	618,19	275,31
	118,48	619,98	252,81
	119,30	626,54	272,06
	118,06	615,03	269,49

Tabela B.4: Resultados para lambda 16 - Processador i7 2600 + GTX-480 - 64 bits

64 bits	i7 2600 + gtx 480	i7 2600 - 1 thread	i7 2600 - 8 thread
Lamba 32	82,01	381,56	161,26
	82,15	379,50	156,71
	82,51	383,64	158,20
	81,66	384,18	169,77
	81,56	385,66	155,33
	81,51	383,25	159,88
	81,81	384,97	155,62
	81,62	382,29	163,50
	82,66	384,33	155,00
	81,48	385,16	164,94

Tabela B.5: Resultados para lambda 32 - Processador i7 2600 + GTX-480 - 64 bits

64 bits	i7 2600 + gtx 480	i7 2600 - 1 thread	i7 2600 - 8 thread
Lamba 64	61,79	255,70	97,24
	61,71	256,95	96,51
	61,61	254,67	112,41
	61,73	256,57	117,51
	61,70	254,56	100,54
	61,74	254,88	101,53
	61,77	256,68	105,02
	62,25	254,61	101,19
	61,92	255,84	104,81
	61,49	254,57	100,73

Tabela B.6: Resultados para lambda 64 - Processador i7 2600 + GTX-480 - 64 bits

32 bits	i7 860 + gtx 580	i7 860 - 1 thread	i7 860 - 8 thread
Lamba 16	114,03	811,51	240,74
	115,46	863,69	239,86
	117,89	812,35	239,21
	113,97	862,54	240,66
	115,35	815,45	240,69
	118,15	861,73	240,79
	115,35	815,51	241,57
	115,73	860,86	239,33
	117,71	817,92	239,80
	115,43	859,82	240,16

Tabela B.7: Resultados para lambda 16 - Processador i7 860 + GTX-580 - 32 bits

32 bits	i7 860 + gtx 580	i7 860 - 1 thread	i7 860 - 8 thread
Lamba 32	80,62	530,46	156,05
	82,50	515,89	155,71
	82,60	530,16	156,19
	82,58	515,98	155,51
	83,13	531,62	156,40
	80,70	515,64	155,90
	79,50	534,20	155,71
	80,61	567,42	155,06
	80,56	530,70	155,82
	82,42	563,47	156,06

Tabela B.8: Resultados para lambda 32 - Processador i7 860 + GTX-580 - 32 bits

32 bits	i7 860 + gtx 580	i7 860 - 1 thread	i7 860 - 8 thread
Lamba 64	62,62	355,97	107,65
	60,96	365,94	106,88
	62,77	355,91	107,20
	61,27	365,76	107,48
	62,71	355,65	107,37
	61,13	365,46	107,54
	61,17	355,61	107,29
	62,83	365,31	107,85
	61,16	355,70	107,27
	62,73	366,13	108,08

Tabela B.9: Resultados para lambda 64 - Processador i7 860 + GTX-580 - 32 bits

32 bits	Phenom II X6 + gts 450	Phenom II X6 - 1 thread	Phenom II X6 - 6 threads
Lamba 16	350,63	999,72	296,48
	350,76	995,06	296,55
	350,86	996,82	296,82
	350,77	1.008,26	296,09
	351,00	995,39	295,62
	351,14	1.005,94	298,58
	350,84	995,04	295,27
	350,70	992,71	296,65
	350,77	995,11	294,80
	350,84	995,00	296,97

Tabela B.10: Resultados para lambda 16 - Processador Phenom II X6 + GTS-450 - 32 bits

32 bits	Phenom II X6 + gts 450	Phenom II X6 - 1 thread	Phenom II X6 - 6 threads
Lamba 32	248,42	620,21	177,70
	248,14	615,89	179,02
	247,98	616,81	178,45
	248,05	626,56	179,76
	248,20	630,41	177,86
	248,68	619,12	178,55
	248,09	628,53	178,03
	248,10	619,19	179,22
	248,20	620,65	177,82
	248,41	625,81	178,03

Tabela B.11: Resultados para lambda 32 - Processador Phenom II X6 + GTS-450 - 32 bits

32 bits	Phenom II X6 + gts 450	Phenom II X6 - 1 thread	Phenom II X6 - 6 threads
Lamba 64	191,49	413,44	120,81
	191,50	405,17	119,82
	191,68	416,20	120,04
	191,31	408,57	120,82
	191,36	406,03	119,60
	192,16	413,26	119,88
	191,50	408,27	120,18
	191,74	416,00	120,02
	191,27	406,36	120,04
	191,41	406,06	120,21

Tabela B.12: Resultados para lambda 64 - Processador Phenom II X6 + GTS-450 - 32 bits

Apêndice C

Resultados das Simulações - Duas instâncias simultâneas

64 bits - 2 Instâncias	i7 860 + gtx 580		i7 860 - 4 thread		i7 860 - 8 thread	
	Instância I	Instância II	Instância I	Instância II	Instância I	Instância II
Lamba 64	105,19	105,19	216,48	216,40	418,60	419,02
	95,31	95,99	216,30	216,36	424,59	420,52
	105,06	105,06	215,18	213,41	419,07	418,38
	104,77	104,77	230,71	233,50	424,37	412,23
	95,85	95,67	231,76	230,88	416,93	414,41
	95,93	96,10	224,94	226,18	417,15	420,29
	96,01	95,03	218,56	214,42	412,85	415,59
	96,32	96,14	223,30	225,48	415,95	417,75
	96,39	95,66	231,57	227,38	410,89	416,69
	96,15	96,00	238,38	235,07	418,55	418,53

Tabela C.1: Resultados para lambda 16 - Processador i7 860 + GTX-580 - 64 bits - Duas instâncias simultâneas

64 bits - 2 Instâncias	i7 860 + gtx 580		i7 860 - 4 thread		i7 860 - 8 thread	
	Instância I	Instância II	Instância I	Instância II	Instância I	Instância II
Lamba 32	132,73	133,30	400,82	399,13	604,59	602,70
	131,86	133,63	392,94	389,48	605,46	599,92
	143,81	143,81	399,81	391,18	603,47	605,01
	132,36	133,10	393,25	396,45	610,93	601,34
	133,17	133,02	378,78	379,31	605,85	608,15
	131,37	133,27	388,42	385,46	615,37	616,02
	131,31	132,98	371,92	374,60	605,51	605,44
	143,41	143,41	381,04	384,06	607,87	603,18
	132,38	133,13	377,38	374,27	616,92	616,96
	131,29	133,18	372,07	370,28	616,91	617,67

Tabela C.2: Resultados para lambda 32 - Processador i7 860 + GTX-580 - 64 bits - Duas instâncias simultâneas

64 bits - 2 Instâncias	i7 860 + gtx 580		i7 860 - 4 thread		i7 860 - 8 thread	
	Instância I	Instância II	Instância I	Instância II	Instância I	Instância II
Lamba 64	105,19	105,19	216,48	216,40	418,60	419,02
	95,31	95,99	216,30	216,36	424,59	420,52
	105,06	105,06	215,18	213,41	419,07	418,38
	104,77	104,77	230,71	233,50	424,37	412,23
	95,85	95,67	231,76	230,88	416,93	414,41
	95,93	96,10	224,94	226,18	417,15	420,29
	96,01	95,03	218,56	214,42	412,85	415,59
	96,32	96,14	223,30	225,48	415,95	417,75
	96,39	95,66	231,57	227,38	410,89	416,69
	96,15	96,00	238,38	235,07	418,55	418,53

Tabela C.3: Resultados para lambda 64 - Processador i7 860 + GTX-580 - 64 bits - Duas instâncias simultâneas

64 bits - 2 Instâncias	i7 2600 + gtx 480		i7 2600 - 4 thread		i7 2600 - 8 thread	
	Instância I	Instância II	Instância I	Instância II	Instância I	Instância II
Lamba 16	218,43	217,80	586,09	586,49	832,23	836,33
	218,40	215,26	562,97	567,86	826,83	834,57
	229,47	229,47	519,98	516,20	835,36	841,12
	215,22	217,11	492,77	489,80	839,49	834,47
	217,22	217,19	504,55	503,53	836,41	837,66
	217,32	218,07	497,95	495,74	829,74	833,77
	217,24	217,21	523,41	525,34	836,29	841,18
	218,08	218,03	601,64	609,32	842,91	836,46
	218,48	216,04	585,35	568,35	834,95	837,49
	218,20	218,17	515,61	511,46	831,77	834,85

Tabela C.4: Resultados para lambda 16 - Processador i7 2600 + GTX-480 - 64 bits - Duas instâncias simultâneas

64 bits - 2 Instâncias	i7 2600 + gtx 480		i7 2600 - 4 thread		i7 2600 - 8 thread	
	Instância I	Instância II	Instância I	Instância II	Instância I	Instância II
Lamba 32	146,51	147,20	276,06	271,65	536,39	534,33
	147,52	147,24	277,62	275,87	540,79	543,13
	147,97	148,43	274,85	274,32	546,00	521,13
	147,87	146,46	276,99	274,60	540,39	538,35
	156,44	156,44	277,02	277,86	541,60	542,66
	156,27	156,27	277,46	282,36	542,46	534,80
	146,87	147,82	271,82	272,59	536,20	537,87
	146,46	147,16	291,58	292,69	540,01	537,43
	146,13	147,04	313,54	309,18	538,80	539,98
	146,60	147,07	303,01	305,00	534,90	538,26

Tabela C.5: Resultados para lambda 32 - Processador i7 2600 + GTX-480 - 64 bits - Duas instâncias simultâneas

64 bits - 2 Instâncias	i7 2600 + gtx 480		i7 2600 - 4 thread		i7 2600 - 8 thread	
	Instância I	Instância II	Instância I	Instância II	Instância I	Instância II
Lamba 64	108,05	108,44	166,10	165,96	362,91	364,56
	108,04	108,20	163,41	162,16	358,46	363,32
	109,18	109,34	164,73	166,47	370,68	368,25
	116,04	116,04	159,01	160,04	365,13	369,28
	107,99	108,15	160,07	160,94	362,49	359,87
	107,63	107,96	163,01	163,00	368,96	368,97
	107,85	108,02	161,70	162,79	360,94	361,94
	108,94	109,09	161,63	159,96	363,88	365,59
	108,23	107,69	161,81	161,34	368,05	359,42
	108,21	107,84	161,82	161,80	358,45	361,53

Tabela C.6: Resultados para lambda 64 - Processador i7 2600 + GTX-480 - 64 bits - Duas instâncias simultâneas

32 bits - 2 Instâncias	i7 860 + gtx 580		i7 860 - 4 thread		i7 860 - 8 thread	
	Instância I	Instância II	Instância I	Instância II	Instância I	Instância II
Lamba 16	191,73	191,75	461,75	456,49	444,25	438,93
	188,95	193,19	462,07	460,11	443,09	443,15
	194,42	193,00	462,72	463,63	442,33	442,00
	190,63	190,64	461,00	459,17	439,47	439,24
	191,63	191,28	459,82	460,01	438,58	438,74
	191,49	191,26	456,74	455,70	439,03	438,00
	191,10	191,13	459,60	459,17	437,95	441,02
	191,55	188,59	459,15	458,78	439,63	438,67
	190,34	189,81	459,27	455,13	447,03	439,56
	188,07	191,58	461,81	462,03	442,72	446,63

Tabela C.7: Resultados para lambda 16 - Processador i7 860 + GTX-580 - 32 bits - Duas instâncias simultâneas

32 bits - 2 Instâncias	i7 860 + gtx 580		i7 860 - 4 thread		i7 860 - 8 thread	
	Instância I	Instância II	Instância I	Instância II	Instância I	Instância II
Lamba 32	129,31	129,29	271,32	270,71	262,94	261,26
	126,79	128,92	271,04	272,90	262,31	262,39
	127,48	129,35	273,50	273,18	262,18	262,26
	126,92	128,88	275,38	274,14	260,89	263,25
	129,21	127,33	275,21	275,37	263,80	265,08
	129,32	130,46	275,10	273,03	261,22	264,86
	127,65	129,79	273,05	274,57	264,14	264,21
	128,12	129,18	272,12	270,89	264,38	263,80
	129,18	128,09	271,09	272,06	263,31	263,17
	129,55	129,53	273,15	270,70	260,85	262,92

Tabela C.8: Resultados para lambda 32 - Processador i7 860 + GTX-580 - 32 bits - Duas instâncias simultâneas

32 bits - 2 Instâncias	i7 860 + gtx 580		i7 860 - 4 thread		i7 860 - 8 thread	
	Instância I	Instância II	Instância I	Instância II	Instância I	Instância II
Lamba 64	93,12	91,21	181,56	181,41	168,40	169,27
	93,99	94,01	180,17	180,78	168,28	169,47
	92,90	92,92	178,08	179,34	168,98	169,23
	94,09	91,59	178,13	178,70	169,02	169,87
	94,28	92,31	177,65	177,13	171,59	168,53
	92,12	94,00	177,06	176,82	170,11	169,49
	93,89	91,95	176,95	176,51	167,57	168,56
	94,05	92,08	175,24	177,65	167,36	167,62
	94,46	92,74	176,91	175,12	167,26	167,30
	92,41	94,36	175,21	175,17	166,23	168,21

Tabela C.9: Resultados para lambda 64 - Processador i7 860 + GTX-580 - 32 bits - Duas instâncias simultâneas

32 bits - 2 Instâncias	Phenom II X6 + gts 450		Phenom II X6 - 3 thread		Phenom II X6 - 6 threads	
	Instância I	Instância II	Instância I	Instância II	Instância I	Instância II
Lamba 16	681,92	681,92	524,34	524,35	752,19	752,11
	669,12	670,26	525,05	525,04	745,72	745,72
	668,79	669,47	524,18	524,17	745,88	745,83
	669,16	669,85	527,42	519,07	748,26	748,30
	669,23	669,77	534,49	529,81	746,61	744,90
	669,87	669,30	525,20	529,31	746,42	747,53
	679,87	679,87	528,59	522,48	739,99	742,53
	669,71	667,71	533,91	526,32	748,20	740,05
	681,47	681,47	533,29	529,43	746,63	747,84
	667,64	668,31	530,01	530,02	741,26	742,91

Tabela C.10: Resultados para lambda 16 - Processador Phenom II X6 + GTS-450 - 32 bits - Duas instâncias simultâneas

32 bits - 2 Instâncias	Phenom II X6 + gts 450		Phenom II X6 - 3 thread		Phenom II X6 - 6 threads	
	Instância I	Instância II	Instância I	Instância II	Instância I	Instância II
Lamba 32	471,00	464,79	302,92	305,67	465,74	466,83
	466,90	469,66	302,04	303,11	464,22	463,19
	464,81	467,61	301,05	300,76	465,39	469,72
	471,11	464,89	304,08	306,23	466,79	468,66
	472,83	467,04	304,03	304,96	465,44	465,13
	465,08	467,87	299,01	299,24	464,00	463,07
	466,48	468,86	306,50	302,05	467,91	467,93
	466,21	468,29	309,77	305,50	466,81	466,48
	481,43	481,43	306,69	308,70	464,07	467,52
	468,44	466,03	305,15	302,82	468,05	468,51

Tabela C.11: Resultados para lambda 32 - Processador Phenom II X6 + GTS-450 - 32 bits - Duas instâncias simultâneas

32 bits - 2 Instâncias	Phenom II X6 + gts 450		Phenom II X6 - 3 thread		Phenom II X6 - 6 threads	
	Instância I	Instância II	Instância I	Instância II	Instância I	Instância II
Lamba 64	358,42	358,40	341,27	260,68	311,85	312,79
	358,51	358,48	369,88	269,03	313,79	314,02
	358,57	357,10	376,60	271,89	315,37	311,07
	369,69	369,69	373,38	271,36	311,91	313,43
	358,46	358,44	372,96	271,81	312,59	312,76
	358,89	357,44	371,86	272,10	312,75	313,18
	370,34	370,34	374,36	271,07	312,10	312,32
	358,45	358,47	373,67	271,73	316,35	310,19
	358,69	358,72	374,21	271,73	317,02	315,13
	358,44	358,47	343,17	273,62	313,59	311,64

Tabela C.12: Resultados para lambda 64 - Processador Phenom II X6 + GTS-450 - 32 bits - Duas instâncias simultâneas

Apêndice D

Resultados das Simulações com uso da Otimização

64 bits - COM Otimização	i7 860 + gtx 580	i7 860 - 1 thread	i7 860 - 8 thread
Lamba 64	51,88	228,26	87,46
	51,75	224,03	88,74
	51,79	223,88	89,66
	51,65	223,88	90,89
	51,66	228,56	91,07
	51,67	224,17	91,33
	51,89	223,92	91,46
	51,69	224,51	91,43
	51,63	223,91	92,28
	51,67	223,97	91,80

Tabela D.1: Resultados para lambda 16 - Processador i7 860 + GTX-580 - 64 bits - Com Otimização

64 bits - COM Otimização	i7 860 + gtx 580	i7 860 - 1 thread	i7 860 - 8 thread
Lamba 32	69,30	327,89	138,29
	69,04	328,03	137,76
	69,03	328,21	138,18
	69,13	328,89	138,01
	69,01	326,55	139,84
	69,06	327,02	138,73
	69,10	327,33	138,23
	69,19	325,25	138,38
	69,13	325,20	138,73
	69,09	335,90	139,21

Tabela D.2: Resultados para lambda 32 - Processador i7 860 + GTX-580 - 64 bits - Com Otimização

64 bits - COM Otimização	i7 860 + gtx 580	i7 860 - 1 thread	i7 860 - 8 thread
Lamba 64	51,88	228,26	87,46
	51,75	224,03	88,74
	51,79	223,88	89,66
	51,65	223,88	90,89
	51,66	228,56	91,07
	51,67	224,17	91,33
	51,89	223,92	91,46
	51,69	224,51	91,43
	51,63	223,91	92,28
	51,67	223,97	91,80

Tabela D.3: Resultados para lambda 64 - Processador i7 860 + GTX-580 - 64 bits - Com Otimização

64 bits - COM Otimização	i7 2600 + gtx 480	i7 2600 - 1 thread	i7 2600 - 8 thread
Lamba 16	110,96	376,39	161,57
	109,38	382,82	190,31
	109,43	373,93	192,90
	109,44	379,65	160,11
	109,37	382,90	162,73
	110,62	374,07	219,98
	109,36	378,64	161,08
	109,45	373,63	161,27
	109,35	376,74	203,13
	109,46	389,06	177,00

Tabela D.4: Resultados para lambda 16 - Processador i7 2600 + GTX-480 - 64 bits - Com Otimização

64 bits - COM Otimização	i7 2600 + gtx 480	i7 2600 - 1 thread	i7 2600 - 8 thread
Lamba 32	75,85	246,43	119,33
	75,85	243,89	101,84
	77,01	244,18	127,33
	75,96	242,47	135,84
	75,78	241,50	103,11
	75,86	243,85	101,78
	75,85	241,94	103,40
	75,85	243,26	115,16
	75,75	242,21	147,50
	76,17	241,54	101,83

Tabela D.5: Resultados para lambda 32 - Processador i7 2600 + GTX-480 - 64 bits - Com Otimização

64 bits - COM Otimização	i7 2600 + gtx 480	i7 2600 - 1 thread	i7 2600 - 8 thread
Lamba 64	57,40	170,13	68,86
	57,06	168,61	68,61
	57,05	168,54	68,50
	58,00	171,27	68,35
	57,25	169,58	68,62
	56,99	166,91	71,73
	56,99	170,11	75,82
	57,06	170,49	76,64
	56,99	168,59	82,62
	57,03	168,59	82,05

Tabela D.6: Resultados para lambda 64 - Processador i7 2600 + GTX-480 - 64 bits - Com Otimização

32 bits - COM Otimização	i7 860 + gtx 580	i7 860 - 1 thread	i7 860 - 8 thread
Lamba 16	103,91	685,81	215,12
	105,65	651,39	215,28
	103,76	681,93	215,73
	104,72	651,13	215,11
	105,52	685,63	215,71
	103,82	657,85	214,12
	106,12	651,15	214,29
	103,80	681,95	214,46
	104,98	685,88	215,61
	106,06	651,90	215,38

Tabela D.7: Resultados para lambda 16 - Processador i7 860 + GTX-580 - 32 bits - Com Otimização

32 bits - COM Otimização	i7 860 + gtx 580	i7 860 - 1 thread	i7 860 - 8 thread
Lamba 32	74,17	451,88	136,41
	73,31	463,16	136,15
	74,32	433,77	135,45
	74,00	434,33	135,41
	73,21	426,48	137,16
	74,79	418,17	136,03
	74,34	431,64	136,19
	73,27	418,29	136,75
	72,44	431,10	136,23
	74,69	418,04	135,71

Tabela D.8: Resultados para lambda 32 - Processador i7 860 + GTX-580 - 32 bits - Com Otimização

32 bits - COM Otimização	i7 860 + gtx 580	i7 860 - 1 thread	i7 860 - 8 thread
Lamba 64	55,43	325,36	92,87
	57,29	320,21	92,67
	56,19	320,72	92,44
	57,39	302,29	92,03
	56,11	316,84	92,13
	57,41	301,61	92,30
	56,14	315,35	92,39
	56,34	302,19	92,89
	56,03	315,24	92,93
	56,19	301,81	92,49

Tabela D.9: Resultados para lambda 64 - Processador i7 860 + GTX-580 - 32 bits - Com Otimização

32 bits - COM Otimização	Phenom II X6 + gts 450	Phenom II X6 - 1 thread	Phenom II X6 - 6 threads
Lamba 64	186,95	341,20	101,77
	187,06	342,86	101,79
	187,18	341,33	101,93
	187,13	338,37	102,91
	186,99	339,64	101,85
	186,84	339,16	102,10
	186,97	340,57	102,16
	187,00	339,28	101,95
	186,95	340,67	101,87
	187,28	341,79	101,93

Tabela D.10: Resultados para lambda 16 - Processador Phenom II X6 + GTS-450 - 32 bits - Com Otimização

32 bits - COM Otimização	Phenom II X6 + gts 450	Phenom II X6 - 1 thread	Phenom II X6 - 6 threads
Lamba 64	186,95	341,20	101,77
	187,06	342,86	101,79
	187,18	341,33	101,93
	187,13	338,37	102,91
	186,99	339,64	101,85
	186,84	339,16	102,10
	186,97	340,57	102,16
	187,00	339,28	101,95
	186,95	340,67	101,87
	187,28	341,79	101,93

Tabela D.11: Resultados para lambda 32 - Processador Phenom II X6 + GTS-450 - 32 bits - Com Otimização

32 bits - COM Otimização	Phenom II X6 + gts 450	Phenom II X6 - 1 thread	Phenom II X6 - 6 threads
Lamba 64	186,95	341,20	101,77
	187,06	342,86	101,79
	187,18	341,33	101,93
	187,13	338,37	102,91
	186,99	339,64	101,85
	186,84	339,16	102,10
	186,97	340,57	102,16
	187,00	339,28	101,95
	186,95	340,67	101,87
	187,28	341,79	101,93

Tabela D.12: Resultados para lambda 64 - Processador Phenom II X6 + GTS-450 - 32 bits - Com Otimização

Referências Bibliográficas

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

[2] A. V. P. de Rezende,"Implementação de um algoritmo de compressão de imagens baseado em recorrência de padrões multiescalas em CPU multinúcleo e GPU massivamente paralelas," UFF, M.Sc, Engenharia de Telecomunicações, 2010

[3] J. Sanders, E. Kandrot,"CUDA by Example - An Introduction to General-Purpose GPU Programming , " Edwards Brothers, Michigan, First printing, July 2010

[4] A. M. Sanche, "Um Estudo de compactação de dados com a implementação do método de LZW, " UEMS, Ciência da Computação, 2001

[5] C.E. Shannon, "A mathematical theory of communication," Bell Syst. Tech. Journal, vol. 27, pp. 379-423, 1948.

[6] C.E. Shannon, "Coding theorems for a discrete source with a fidelity criterion," in IRE National Convention Record, Part 4, pp. 142-163, 1959

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

[8] The GNU Project. "GCC, the GNU Compiler Collection." http://gcc.gnu.org/, Acessado em 10 de fevereiro de 2013

[9] NVIDIA Corporation. "CUDA C BEST PRACTICES GUIDE - Design Guide", DG-05603-001_v4.1, January 2012

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

[11] Harris, Mark., “Optimizing Parallel Reduction in CUDA.”, NVIDIA Developer Technology.

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

[13] G. E. Blelloch, “Introduction to Data Compression*”, Carnegie Mellon University, Computer Science Department, blellochcs.cmu.edu, September 25, 2010

[14] A. K. Jain, “Fundamentals of Digital Image Processing,” Prentice-Hall, 1989.

[15] “Lempel-Ziv-Welch (LZW) Algorithm, “ http://www.cs.cf.ac.uk/Dave/Multimedia/node214.htm
Acessado em 06 de janeiro de 2013

[16] “Amdahl’s Law , ” http://en.wikipedia.org/wiki/Amdahl_s_law, Acessado em 22 de janeiro de 2013