Dissertação de Mestrado em

Engenharia Informática - Computação Móvel

CUDA-MMP - Codificação de imagens com sistemas com múltiplos núcleos

Tiago Martins Ribeiro

Leiria, 23 de Março de 2016

Dissertação de Mestrado em

Engenharia Informática - Computação Móvel

CUDA-MMP - Codificação de imagens com sistemas com múltiplos núcleos

Tiago Martins Ribeiro

Dissertação de Mestrado realizada sob a orientação do Doutor Patrício Rodrigues Domingues, Professor Adjunto, Doutor Nuno Miguel Morais Rodrigues, Professor Adjunto e Doutor Sérgio Manuel Maciel Faria, Professor Coordenador da Escola Superior de Tecnologia e Gestão do Instituto Politécnico de Leiria.

Leiria, 23 de Março de 2016

Agradecimentos

Reservo esta página para agradecer a todos os intervenientes que ao longo da escrita deste documento sempre me apoiaram.
Em primeiro lugar, reservo um especial agradecimento à minha família que em todos os momentos estiveram disponíveis e me apoiaram, dando-me força para continuar e por fim terminar esta etapa da minha carreira académica.
Um grande obrigado ao meu professor orientador, Doutor Patrício Rodrigues Domingues, pelas várias horas que disponibilizou com o intuito de me ajudar a desenvolver este documento e pela ajuda prestada durante o desenvolvimento do conteúdo apresentado no mesmo. Pelas ideias partilhadas e discussões que tivemos por forma a encontrar soluções onde nem sempre as mesmas estavam à vista de todos.
Um grande obrigado também aos meus professores coordenadores, Doutor Nuno Miguel Morais Rodrigues e Doutor Sérgio Manuel Maciel Faria pelo grande apoio fornecido ao longo desta atribulada jornada.
Um especial obrigado também ao meu grande colega João Filipe Crespo Silva, pelas horas partilhadas na busca de soluções. A sua ajuda foi imprescindível para o término e qualidade do trabalho desenvolvido neste documento.

Esta página foi intencionalmente deixada em branco.

Nota prévia

Do trabalho efetuado resultou a seguinte publicação:
"Optimizing Memory Usage and Accesses on CUDA-Based Recurrent Pattern Matching Image Compression", Patrício Domingues, João Silva, Tiago Ribeiro, Nuno Rodrigues, Murilo Carvalho, Sérgio Faria in ICCSA 2014 Conference on High Performance Computing in Engineering and Science (HPCES).

Esta página foi intencionalmente deixada em branco.

Resumo

Hoje em dia, a área de codificação de dados é transversal a diversos tipos de engenharias devido à sua grande importância. Com o aumento exponencial na criação de dados digitais, o campo da compressão de dados ganhou uma grande visibilidade nesta área. São constantemente desenvolvidos e melhorados algoritmos de compressão por forma a obter a maior compressão de dados possível seja com ou sem perda de dados, permitindo sustentar o rápido e constante crescimento dos mesmos. Um dos grandes problemas deste tipo de algoritmos deve-se ao grande poder computacional que por vezes é necessário para obter uma boa taxa de compressão mantendo a qualidade dos dados quando descompactados.

Este documento descreve uma estratégia para tentar reduzir o impacto do poder computacional necessário à codificação de imagens utilizando uma implementação heterogénea. O objetivo é tentar efetuar a paralelização das secções que requerem elevado poder computacional reduzindo assim o tempo necessário à compressão de dados. Este documento baseia-se na implementação desta estratégia para o algoritmo de codificação de imagens MMP-Intra. Utilizando inicialmente uma análise teórica, demonstramos que é viável efetuar a paralelização do algoritmo, sendo possível obter elevados ganhos de desempenho.

Por forma a provar que o algoritmo MMP-Intra era paralelizavel e identificar os ganhos reais foi desenvolvido um protótipo inicial, o qual obteve um desempenho muito inferiore ao do algoritmo original, necessitando de muito mais tempo para obter os mesmo resultados. Utilizando um processo de otimização iterativo o protótipo passou por várias etapas de refinação. O protótipo refinado final obteve resultados muito superiores ao algoritmo sequencial no qual o mesmo foi baseado chegando a obter desempenhos quatro vezes superior ao original.

Palavras-chave: CUDA, heterogéneo, codificação de imagem, MMP-Intra

Esta página foi intencionalmente deixada em branco.

Abstract

Nowadays, data encoding is an area that cross-cuts many different fields of engineering due to its increasing contemporary importance. With the exponential increase in the creation rate of digital data, data compression is a field that has attained a high profile in the area of data encoding. Various compression algorithms continue to be developed and improved in order to obtain the highest data compression rate possible, both with and without data loss, in order to allow the sustainability of the rapid and steady growth of digital data creation. A major problem with this type of algorithms is the high demand of computational power which is sometimes necessary to obtain good compression rates.

This document describes a strategy that tries to reduce the time demands of the needed computational power to encode an image by using an heterogeneous approach. Our aim is to parallelize the areas of the algorithm with computational bottlenecks, in order to reduce the time required for the process. This document describes the implementation of this strategy for the algorithm MMP-Intra, used in image compression. Initially, a theoretical analysis was undertaken to test the possible gains of the algorithm parallelization. The results of this test indicated that the parallelization of the algorithm may indeed be an effective approach.

In response to this test finding, an initial prototype was developed. This prototype was lower performing than the original algorithm, requiring much more time to achieve the same results. Using an iterative optimization process, the prototype underwent several refining stages. The final refined prototype achieved much better results than the sequential algorithm on which it was based reaching performances four times better than its predecessor.

Keywords: CUDA, heterogeneous, image coding, MMP-Intra

Esta página foi intencionalmente deixada em branco.

Lista de Figuras

Esta página foi intencionalmente deixada em branco.

Lista de Tabelas

Esta página foi intencionalmente deixada em branco.

Acrónimos

Acrónimo	Significado
API	Application Programming Interface
ALU	Arithmetic Logic Unit
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GPGPU	General Purpose Graphical Processing Unit
GPU	Graphical Processing Unit
GID	Global Identifier
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MMP	Multidimensional Multiscale Parser
NVCC	NVIDIA Cuda compiler driver
NVProf	NVIDIA Cuda command line profiler
NVVP	NVIDIA Visual profiler
PC	Personal Computer
PTX	Parallel Thread Execution
RISC	Reduced Instruction Set Computer
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SISD	Single Instruction Single Data
SM	Streaming Multiprocessor
TbID	Thread Block Identifier
UNIVAC	Universal Automatic Computer

Esta página foi intencionalmente deixada em branco.

Conteúdo

Capítulo 1

Introdução

Neste capítulo é feita uma breve contextualização sobre a origem do trabalho e o ambiente sobre o qual ele foi desenvolvido, após o qual é feita uma breve descrição sobre os motivos que levaram à realização do mesmo.

1.1 Contextualização

Este trabalho foi desenvolvido no âmbito de uma bolsa de investigação financiada pela Fundação para a Ciência e Tecnologia e pelo Instituto de Telecomunicações. A bolsa de investigação encontra-se inserida no projeto de acrónimo EPIC e referência PTDC/EIA-EIA/122774/2010 tendo sido iniciado o financiamento na data de 01/01/2012 e terminado na data de 01/07/2014. O projeto foi orientado pelos professores Nuno Miguel Morais Rodrigues, Sérgio Manuel Maciel Faria e Patrício Rodrigues Domingues.

1.2 Motivação

Estamos hoje em dia num tempo onde os sistemas e dados digitais são um bem essencial da sociedade tanto a nível de lazer, quando ouvimos músicas ou tiramos fotografias com máquinas fotográficas digitais, como a nível empresarial onde todas as informações de gestão e operações se encontram em sistemas digitais. Esta tendência para a digitalização implica que o ritmo de criação de dados digitais tem vindo a aumentar nos últimos anos. Para além do aumento de criação a qualidade dos dados também tem vindo a aumentar, temos o exemplo do vídeo em qualidade HD ou 4K. Com este aumento exponencial de dados digitais aumentou também a necessidade de conseguir representar os mesmos utilizando o menor espaço

possível pois como sabemos os periféricos como discos rígidos, dispositivos USB ou DVDs têm um espaço bastante limitado. O problema aumenta de magnitude quando falamos de transferência de dados, pois a largura de banda e as velocidades dos canais de transferência existentes hoje em dia são bastante reduzidos.

Por forma a colmatar este problema diversos algoritmos de codificação de dados foram desenvolvidos. Estes algoritmos têm como objetivo a modelação de um dado sinal tornando-o mais apropriado para um determinado ambiente ou fim. De entre os algoritmos de codificação existe um tipo designado por compressão. Este tipo de codificadores tem como objetivo a diminuição do espaço necessário para representar um determinado sinal. Dentre estes algoritmos existe uma classe de algoritmos cuja finalidade é a compressão de imagens. Este tipo de algoritmo tenta diminuir o espaço necessário para representar uma dada imagem, o *JPEG2000* [3] e o *H.265* [4] são dois exemplos deste tipo de algoritmos.

Outro algoritmo desenvolvido tendo em conta o âmbito descrito anteriormente designa-se por *Multidimensional Multiscale Parser* [5]. Este é um algoritmo de codificação/compressão de dados genérico que utiliza esquemas de correspondência de padrões de múltiplas escalas para representar os dados originais. Embora este algoritmo consiga rivalizar com os atuais estados da arte em termos de qualidade de compressão, o mesmo não acontece quando comparando o desempenho computacional, requerendo tempos de execução muito superiores.

Outro tema que tem ganho bastante atenção na comunidade científica é a computação paralela. Isto deve-se aos grandes ganhos de desempenho que se conseguem atingir. Embora este tipo de tecnologias já seja utilizado há bastante tempo em determinadas áreas especificas da computação, apenas recentemente se tem visto um crescimento da sua utilização em algoritmos de propósito geral. Este aumento deve-se à disponibilização de *frameworks* que permitem um fácil e eficiente uso das capacidades de paralelização existentes nos diversos sistemas existentes, tais como CPUs ou GPUs.

Este projeto foi iniciado com o propósito de implementar o algoritmo MMP-Intra, para codificação de imagens, em ambientes heterogéneos que consiga rivalizar com os atuais estados da arte da sua área tanto em qualidade de compressão como em desempenho computacional.

1.3 Objetivos

Este projeto foi iniciado com o objetivo de analisar e implementar uma nova versão do algoritmo sequencial MMP-Intra em sistemas heterogéneos CPU/GPU. Assim sendo diversos objetivos concretos foram definidos:

- Análise da versão sequencial do algoritmo MMP-Intra
- Identificação dos *bottlenecks*
- Análise da migração dos *bottlenecks* para uma arquitetura paralela
- Desenvolvimento de um protótipo do MMP-Intra com base na tecnologia CUDA (CUDA-MMP)
- Otimização do algoritmo CUDA-MMP

1.4 Organização do Documento

Este documento está organizado da seguinte forma.
O capítulo 2 efetua uma introdução à codificação de imagens, efetuando uma descrição mais específica sobre o modo de funcionamento do algoritmo de codificação de imagens MMP e do seu sucessor MMP-Intra.
O capítulo 3 começa por introduzir o conceito de computação de alto desempenho, descrevendo as principais motivações que levaram à criação do mesmo. Por fim, o mesmo foca-se na tecnologia CUDA, explicando os conceitos inerentes à tecnologia.
O capítulo 4 descreve o ambiente computacional utilizado ao longo de todo o desenvolvimento do trabalho descrito. Seguidamente é descrita a análise de *profiling* efetuada ao algoritmo MMP-Intra, terminando com a descrição do primeiro protótipo implementado, o CUDA-MMP.
O capítulo 5 descreve as análises e otimizações levadas a cabo no processo iterativo de refinamento do protótipo CUDA-MMP. Este capítulo divide-se em três secções distintas onde cada secção focou um tipo de arquitetura diferente. Em primeiro lugar é efetuada uma descrição das otimizações de memória, estas apenas relevantes à parte que implementa uma arquitetura paralela. Seguidamente são descritas as otimizações de fluxo de execução, onde o foco é em partes do algoritmo que implementem uma arquitetura heterogénea, ou seja, são otimizações com impacto na interoperabilidade entre ambas as arquiteturas sequencial e paralela. Este termina com uma descrição das otimizações de árvores de segmentação do resíduo, sendo estas direcionadas à parte do algoritmo que utiliza um fluxo sequencial.
A terminar o documento encontra-se um capítulo de conclusões e trabalhos futuros.

Capítulo 2

Compressão de Imagens

Este capítulo faz uma introdução aos algoritmos de compressão de imagens tendo o seu foco no algoritmo *Multidimensional Multiscale Parser* (MMP), sendo este o algoritmo base utilizado no desenvolvimento deste trabalho. Para isso são introduzidos alguns conceitos de codificação de imagens sendo posteriormente explicado o funcionamento base do algoritmo original e a sua evolução até à data atual.

2.1 Introdução

Os algoritmos de codificação de sinal têm como objetivo alterar a natureza de um dado sinal tornando-o mais apropriado para um determinado ambiente ou fim. Para isso é efetuada a modelação do mesmo utilizando vários tipos de técnicas distintas. Um dos tipos de algoritmos de codificação de sinal mais utilizados hoje em dia são os algoritmos de compressão de dados. Estes são muito utilizados por forma a facilitar o armazenamento e envio de dados em diversos meios, para tal o sinal é modelado com o objetivo de diminuir o seu custo de representação. Existem vários tipos de algoritmos de codificação, sendo as suas características dependentes dos dados que se pretendem codificar.

2.2 Teoria da Informação

A teoria da informação é uma área da matemática baseada na teoria das probabilidades e estatística que contempla o estudo dos limites da compressão, armazenamento e comunicação de dados. Desenvolvida com base no trabalho de Claude Shannon, a teoria da informação pretende estudar os limites do processamento de sinais tais como a compressão de dados ou

o envio de sinais através de canais com ruído. Para isso Shannon considera o problema como meramente matemático recorrendo a áreas da estatística [6].

A teoria da informação assenta sobre o conceito de entropia. A entropia é por definição uma medida de desordem num dado sistema e é geralmente utilizada como uma unidade de incerteza associada a um determinado acontecimento. Na teoria da informação, a entropia pretende quantificar a quantidade de informação existente num dado acontecimento. Esta é dependente do tamanho do domínio a partir do qual o evento é gerado tal como da sua distribuição probabilística. A teoria da informação define que quanto maior for o domínio de símbolos a partir dos quais um determinado evento é gerado maior será a quantidade de informação necessária para representar esse mesmo evento e assim maior será a sua entropia.

2.2.1 Taxa de Entropia

Utilizando esta definição de entropia como base no estudo da compressão de dados, Shannon formulou a teoria da compressão de dados. Esta define um limite máximo para a compressão de dados designado de taxa de entropia[7][8]. Tal como acontece com a definição de entropia, a taxa de entropia depende também do domínio a partir do qual o evento a codificar foi gerado, assim, dependendo das características do domínio, modelos distintos deverão ser utilizados:

- *Zero-Order Model*: Todos os símbolos contidos no domínio da fonte são estatisticamente independentes entre si e seguem uma distribuição probabilística uniforme contínua. Utilizado numa fonte de domínio de tamanho m a taxa de entropia pode ser calculado utilizando a seguinte equação:

$$H = \log_n(m) \tag{2.1}$$

- *First-Order Model*: Todos os símbolos contidos no domínio da fonte são estatisticamente independentes entre si e seguem uma distribuição probabilística arbitrária. Utilizado numa fonte de domínio de tamanho m e sendo P_i a probabilidade de ocorrência do símbolo i a taxa de entropia pode ser calculado utilizando a seguinte equação:

$$H = -\sum_{i=1}^{m} P_i \times \log_n(P_i) \tag{2.2}$$

Todos estes modelos são utilizados para calcular a taxa de entropia por forma a obter uma referência do limite máximo que se consegue comprimir um determinado evento. A base a ser utilizada no logaritmo dos vários modelos deverá ser o número máximo de estados que cada unidade de informação pode representar no sistema de representação escolhido. Para um sistema de representação de informação binário, a base do logaritmo a utilizar é 2 e H irá representar o número de *bits* mínimo necessário para representar cada símbolo presente no domínio m.

2.2.2 Codificação Entrópica

As técnicas de codificação de entropia são uma classe de técnicas de compressão de dados que utilizam como base um esquema de codificação tendo em conta a distribuição probabilística de cada símbolo.
A representação de cada símbolo de informação tende a seguir regras de representação estritas que englobam tamanhos de representação fixos. Isto significa que independentemente das características do domínio de símbolos, os mesmos são todos representados utilizando uma quantidade de informação fixa. Como exemplo consideremos um domínio de 5 símbolos, sabendo que cada *bit* pode guardar até 2 estados diferentes, para representar cada símbolo num domínio de 5 símbolos são precisos no mínimo 3 *bits*, sendo que $2^3 = 8$ estados diferentes.
As técnicas de codificação de entropia comprimem os símbolos mapeando os mesmos para esquemas de representação de tamanho variável, mapeando os símbolos com maior probabilidade de serem utilizados para representações de custo mais reduzido. Duas das técnicas de codificação de entropia mais utilizadas até aos dias de hoje são a Codificação de Huffman [9] e a Codificação Aritmética [10].

Codificação de Huffman

A Codificação de Huffman é um algoritmo que atribui códigos de representação binários únicos a cada símbolo diferente, garantindo que nenhum código é utilizado como prefixo de outro. Todos os símbolos e os seus mapeamentos para códigos binários são guardados num dicionário a que damos o nome de *codebook*.
Para efetuar a codificação da informação o algoritmo necessita de conhecer todos os símbolos a serem codificados e a sua distribuição probabilística. Tendo obtido essa informação é gerado um *codebook* que mapeia cada símbolo numa representação binária com o menor tamanho possível, garantindo mais uma vez que cada código binário não é prefixo de qualquer outro. Para esse fim o algoritmo gera o *codebook* sobre a forma de uma árvore binária. Uma

vez gerado o *codebook* o algoritmo percorre todos os símbolos a codificar e gera uma cadeia de bits mapeando os símbolos para as suas representações binárias.
Sabendo que nenhum código é prefixo de outro, durante a etapa de descodificação, sempre que é encontrado uma sequência binária pertencente ao *codebook* o algoritmo sabe que foi encontrado o símbolo e que o próximo *bit* irá ser o inicio da representação do próximo símbolo.
Como exemplo, utilizemos como domínio os seguintes símbolos e respetivas representações de tamanho fixo: $A : 000$, $B : 001$, $C : 010$, $D : 011$ e $E : 100$. Sabendo que cada símbolo é representado utilizando um esquema de representação binária de tamanho 3, são sempre necessários $3n$ *bits* para representar uma sequência de n símbolos.

Símbolo	**Probabilidade**	**Representação Binária**
A	0.5	0
B	0.25	10
C	0.15	110
D	0.05	1110
E	0.05	1111

Tabela 2.1: Exemplo de um *codebook* para o algoritmo de codificação Huffman com 5 símbolos

Assim para representar a sequência de símbolos $AACBD$ utilizando o esquema de representação de tamanho fixo seriam necessários $3 \times 5 = 15$ *bits*. Utilizando a representação binária definida no *codebook* representado na Tabela 2.1 os mesmos seriam representados por 00110101110, ou seja 11 *bits*. Quanto maior for a diferença de probabilidades entre os símbolos melhor será a compressão, pois o *codebook* será adaptado às características probabilísticas do domínio.
O algoritmo de codificação de Huffman permite a utilização de um *codebook* flexível onde as probabilidades dos vários símbolos podem ser alteradas durante a etapa de codificação.

Codificação Aritmética

Ao contrário da codificação de Huffman que codifica cada símbolo de uma sequência de cada vez, associando a cada símbolo uma representação binária de tamanho variável, a codificação aritmética codifica toda uma sequência de símbolos de uma única vez.
A codificação aritmética utiliza números reais compreendidos entre o intervalo $[0, 1)$ para representar qualquer sequência de símbolos. Quanto maior for o tamanho da sequência a codificar maior será a precisão necessária para representar o número real correspondente.
Para isso em primeiro lugar é necessário definir um modelo com todo o domínio de símbolos.

Consideremos como exemplo que o nosso domínio é composto pelos quatro símbolos *a*, *b*, *c* e *d*. Para ser definido o modelo é necessário saber tanto os símbolos que vão ser utilizados tal como a sua distribuição probabilística. O modelo define um intervalo de representação de cada símbolo definindo o limite inferior como o somatório de todas as probabilidades dos símbolos anteriores, e o limite superior igual ao limite inferior mais a probabilidade do símbolo atual.

Símbolo	**Probabilidade**	**Intervalo de Representação**
A	0.5	[0, 0.5)
B	0.25	[0.5, 0.75)
C	0.15	[0.75, 0.9)
D	0.10	[0.9, 1)

Tabela 2.2: Exemplo de um *codebook* para o algoritmo de codificação aritmética com 4 símbolos

Consideremos que o nosso modelo é definido pelos símbolos e probabilidades descritos na Tabela 2.2 e que queremos codificar a sequência de símbolos *aab*.

Antes de ler qualquer símbolo, o intervalo inicial é de $[0, 1)$. Ao ler o primeiro caracter *a*, o codificador irá reduzir o intervalo tendo em consideração tanto o intervalo atual como o intervalo do símbolo. Assim os novos limites do intervalo serão definidos pelas seguintes equações:

$$L = La + Lir * (Ha - La), \tag{2.3}$$

esta equação irá definir o novo limite inferior do intervalo, sendo *La* o limite inferior atual, *Lir* o limite inferior do intervalo de representação do símbolo definido no modelo e *Ha* o limite superior atual;

$$H = La + Hir * (Ha - La), \tag{2.4}$$

esta equação irá definir o novo limite superior do intervalo, sendo *La* o limite inferior atual, *Hir* o limite superior do intervalo de representação do símbolo definido no modelo e *Ha* o limite superior atual.

Assim utilizando as Equações 2.4 e 2.3, após ler o símbolo *a* o novo intervalo é definido por $[0, 0.5)$. Passando para o segundo símbolo, *a*, o intervalo é alterado para $[0, .25)$. Por fim o terceiro símbolo é um *b*, assim o intervalo é alterado para $[0.0625, 0.1875)$.

Uma vez lidos todos os símbolos da sequência, a mesma pode ser representada por qualquer valor pertencente ao intervalo final. No exemplo anterior qualquer valor pertencente a

$[0.0625, 0.1875)$.
A descodificação do valor real para o sinal original segue o mesmo processo de manipulação do intervalo inicial que a codificação, sendo apenas necessário para cada iteração identificar a que intervalo de representação pertence o valor real transmitido tendo em conta o intervalo atual.

2.2.3 Codificação de Bloco

Como podemos ver na secção anterior, as técnicas de codificação de dados podem ser divididas em dois tipos distintos: (1) codificação por símbolo ou (2) codificação por bloco [8].
As técnicas de codificação por símbolo utilizam esquemas de compressão que codificam um símbolo de cada vez. Por outro lado, as técnicas de codificação por bloco baseiam-se na codificação de agregados de símbolos. O facto de se codificarem agregados de símbolos permite explorar as probabilidades condicionais entre os símbolos de um mesmo agregado. A caso de exemplo consideremos que o domínio d é definido por dois símbolos $d = \{a, b\}$ e sabemos que o domínio segue uma distribuição probabilística uniforme. Utilizando uma árvore de Huffman para representar cada símbolo, podemos utilizar apenas um único *bit* para representar toda a árvore utilizando o *bit* 1 para representar o símbolo a e o *bit* 0 para representar o símbolo b. Utilizando esta árvore para representar a mensagem *aabbabaa* sabemos que serão precisos tantos *bits* quantos símbolos na mensagem ou seja 8 *bits*. Consideremos agora que embora a distribuição probabilística do domínio d seja uniforme, sabemos que a probabilidade condicional do símbolo a aparecer depois de um símbolo b é de 0.9 e a probabilidade condicional do símbolo b aparecer depois de um símbolo a é de 0.9. Todas as restantes probabilidades são de 0.1.

Símbolo	**Probabilidade**	**Representação Binária**
aa	0.45	1
bb	0.45	01
ab	0.05	000
ba	0.05	001

Tabela 2.3: Exemplo de um *codebook* para o algoritmo de codificação Huffman utilizando codificação por bloco

Se agregarmos os símbolos em blocos, podemos tirar partido das probabilidades condicionais por forma a representar mais símbolos com uma menor quantidade de *bits*. Agrupando os símbolos do domínio d em blocos de dois, podemos gerar um novo domínio $b = \{aa, bb, ab, ba\}$.

Utilizando o modelo definido na Tabela 2.3, podemos verificar que é possível representar a mensagem *aabbabaa* utilizando a sequência de *bits* 1010001, ou seja utilizando um *bit* a menos do que na codificação por símbolos.

Taxa-Distorção

A taxa de compressão é usualmente utilizada na compressão sem perdas, onde o sinal descodificado é uma réplica exata do sinal codificado. No entanto, é possível e por vezes aceitável, dependendo das características da informação a codificar, que o sinal possa ser codificado utilizando técnicas de codificação que percam parte da informação e que produzam um sinal não igual mas semelhante ao original, conseguindo obter maiores taxas de compressão do que os indicados pelo limite da taxa de entropia. Neste tipo de compressão a taxa de compressão só por si não pode ser considerado como uma medida de compressão, uma vez que dependendo da quantidade e magnitude das perdas de informação a taxa de entropia é variável. No limite podemos dizer que a taxa de compressão tende para 0 quando considerando grandes perdas de informação. Como é óbvio, uma tal magnitude de perdas de informação fará com que após a descodificação o sinal não possa ser utilizado para o fim pretendido.
Por forma a quantificar a distância entre o bloco original e o bloco codificado é utilizada uma função de distorção $d(x, y)$, esta devolve uma distância matemática entre o bloco original x e o bloco codificado y, a que damos o nome de distorção. Esta deverá ser sempre positiva e superior a 0.
Três das formas mais conhecidas de calcular distorção entre dois sinais são: (1) medida de Hamming, (2) medida de erros absolutos e (3) medida de erros quadráticos.

- A medida de Hamming permite o cálculo da distância entre dois tipos de dados arbitrários. Esta é uma medida de cálculo simples que devolve distorção 0 quando ambos os símbolos são iguais e distorção 1 quando os símbolos enviados para a função são diferentes.

- A medida de erros absolutos e a medida de erros quadráticos permite o cálculo da distância absoluta entre dois tipos de dados numéricos. As mesmas podem ser calculadas utilizando as seguintes equações:

$$d(x, y) = (x - y), \tag{2.5}$$

$$d(x, y) = (x - y)^2, \tag{2.6}$$

onde x é o símbolo antes de ser codificado e y é o símbolo depois de descodificado, sendo a Equação 2.5 utilizada no cálculo do erro absoluto e a Equação 2.6 utilizada no cálculo do erro quadrático.

Só por si a medida da distorção não nos permite saber qual a taxa de compressão que podemos obter uma vez que a mesma apenas nos indica a distância entre dois blocos. Para isso é utilizada uma função $R(D)$ a que damos o nome de Taxa-Distoção [11]. Esta recebe a distorção e relaciona a mesma com a taxa obtida na codificação do bloco. A função $R(D)$ é utilizada para medir o possível ganho na codificação de um sinal ao longo das várias distorções substituindo assim a taxa de entropia.

Quantificação

A quantificação é um processo da área da matemática e processamento de sinais que estuda a transformação de um dado conjunto de elementos em um conjunto de menor dimensão. Utilizando técnicas de quantificação em áreas de processamento de sinais, é possivel alterar o custo de representação de um conjunto de dados.
Para tal a informação a codificar é processada por uma função que irá efetuar o particionamento dos dados em vetores a que damos o nome de blocos. Cada bloco é mapeado para um elemento com um custo de representação menor recorrendo a um *codebook*. Utilizando uma função de mapeamento inversa e o mesmo *codebook*, é possível reverter o sinal para o seu estado inicial. Este tipo de codificação pode ser efetuado com ou sem perdas de informação. Ao componente que contém a lógica de particionamento e mapeamento damos o nome de *Quantizer* [12].

2.3 *Multidimensional Multiscale Parser*

O *Multidimensional Multiscale Parser* (MMP) [5] é um algoritmo de codificação de dados genérico que recorre a um modelo de quantificação vetorial. Dado um bloco de entrada X, este é projetado sobre um *codebook* D obtendo para cada projeção um valor qualitativo de fidelidade. Uma vez que X tenha sido projetado para todos os elementos de D, o padrão cuja projeção tenha obtido um melhor valor de fidelidade será escolhido para codificar X. Existe no entanto um valor mínimo de fidelidade, ou seja caso a projeção do bloco sobre o padrão escolhido seja superior ao valor mínimo de fidelidade este será escolhido para codificar o bloco e o algoritmo continuará o processo de codificação para os blocos seguintes. No entanto caso o valor de fidelidade do padrão escolhido seja inferior ao valor mínimo de

fidelidade o algoritmo irá segmentar X em dois blocos de escalas inferiores X_1 e X_2, tendo cada um metade do tamanho de X. Para cada bloco gerado todo o processo de codificação será repetido. Tendo sido encontradas projeções aceitáveis para todos os blocos gerados o algoritmo irá obter uma árvore a que damos o nome de árvore de segmentação do bloco. Cada nó final da árvore de segmentação irá apontar para um padrão do dicionário que irá ser utilizado para codificar o bloco associado a esse nó.

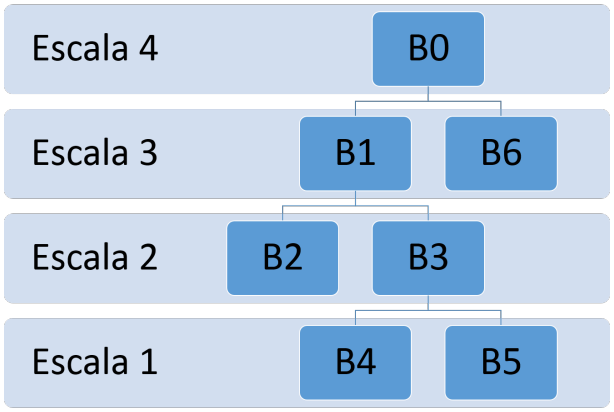

Figura 2.1: Árvore de segmentação de bloco

Cada nó não final será codificado utilizando uma concatenação de dois vetores representativos dos nós filhos. Estes poderão ser constituídos por elementos do dicionário ou concatenações dos mesmos. Para codificar um bloco através da sua árvore de segmentação o algoritmo percorre a árvore em profundidade e gera uma cadeia de *bits* que será o ficheiro codificado. Sempre que durante a fase de processamento da árvore, o algoritmo encontrar um nó de segmentação, uma *flag* com o valor 0 será inserida na cadeia de *bits*. Caso o nó seja um nó final então o algoritmo insere uma *flag* com o valor 1 seguida do índice do padrão escolhido.
Na Figura 2.1 temos um exemplo de uma árvore de segmentação. Se o algoritmo fosse codificar a árvore apresentada, este começaria por percorrer a árvore no nó $B0$, sendo este um nó segmentado, o algoritmo insere uma *flag* 0 e passa para o bloco $B1$. Mais uma vez encontramos um nó segmentado então uma nova *flag* 0 é inserida na cadeia de *bits* passando de seguida para o nó $B2$. Sendo o nó $B2$ um nó final o algoritmo insere uma *flag* 1 indicando que se encontra num nó final e insere o índice do padrão utilizado para codificar $B2$ passando de seguida para o nó $B3$. Sendo este um nó segmentado uma nova *flag* 0 é inserida e passamos para o nó $B4$. Mais uma vez encontramo-nos num nó final, uma *flag* 1 é inserida seguida do índice do padrão utilizado. Passamos assim para o nó $B5$, este é outro nó final então mais uma vez a *flag* 1 é inserida seguida do índice do padrão. Por fim chegamos ao nó $B6$ que é também um nó final. Uma vez percorrida a árvore demonstrada na Figura 2.1 a cadeia de

símbolos gerada é:

$$001I_{B2}01I_{B4}1I_{B5}1I_{B6}. \tag{2.7}$$

A cadeia de *bits* é codificada utilizando um codificador entrópico que implementa uma técnica de codificação aritmética permitindo assim a compressão do mesmo diminuindo o seu custo de representação.

O processo de segmentação permite reduzir a distorção durante o processo de quantificação de vetores. Isto acontece porque após uma segmentação, os vetores a aproximar com os padrões do *codebook* passam a ter um tamanho inferior, permitindo assim que o algoritmo consiga encontrar padrões com distorções inferiores. Embora este processo melhore os resultados obtidos na quantificação, os mesmos resultam numa diminuição da taxa de compressão. Esta diminuição deve-se à necessidade de inserir mais *flags* de sinalização na cadeia de *bits* por forma a indicar as várias segmentações.

Para diminuir o impacto das segmentações durante o processo de codificação o algoritmo utiliza um *codebook* adaptativo. Isto significa que ao longo da codificação o algoritmo vai adicionando novos padrões ao dicionário com transformações e concatenações de padrões escolhidos no processamento de blocos anteriores. Tendo em conta a redundância espacial da imagem, o processo de crescimento do dicionário utilizando blocos já utilizados, aumenta a probabilidade desses mesmos blocos serem escolhidos. Efetuando este processo para escalas superiores do dicionário, aumenta a probabilidade de blocos de escalas maiores serem escolhidos diminuindo assim o número de segmentações.

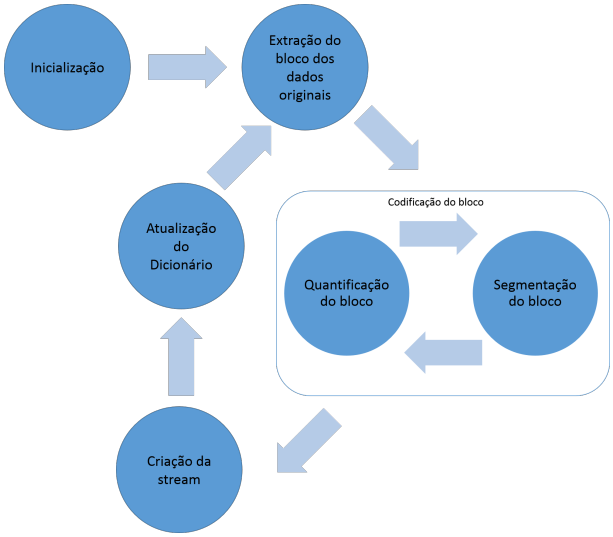

Figura 2.2: Modelo do algoritmo MMP

Como se pode ver na Figura 2.2 o algoritmo MMP pode ser dividido em cinco fases distintas: Inicialização (1), extração do bloco (2), codificação do bloco (3), criação da cadeia de *bits* (4) e atualização do dicionário (5).

2.4 O algoritmo MMP-Intra

O algoritmo MMP-Intra [13] é uma variante do algoritmo MMP que foi adaptado para a codificação de imagens em escalas de cinza, utilizando para tal um modelo modificado de correspondência de padrões utilizado pelo MMP aliando-o a técnicas de predição *intra-frame* do algoritmo H264. Neste capítulo será feita uma introdução às principais alterações deste algoritmo.

2.4.1 Esquema de segmentação

O MMP original foi desenvolvido para codificar dados genéricos. Tendo em conta essa natureza os blocos a codificar eram obtidos sobre a forma de *arrays* unidimensionais de tamanho fixo. Com a evolução do algoritmo focada na codificação de imagens revelou-se vantajoso que os dados originais fossem divididos em *arrays* bidimensionais. O MMP-Intra divide a imagem original em blocos de $16x16$ aos quais damos o nome de macro-blocos.

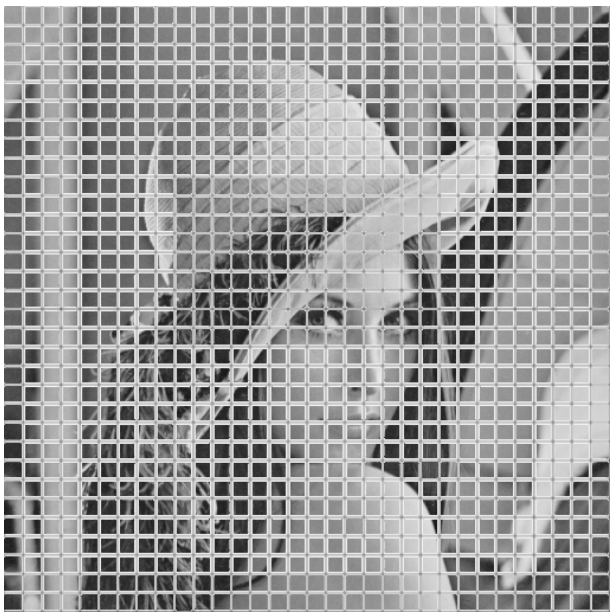

Figura 2.3: Segmentação da imagem Lena em macro-blocos

Na Figura 2.3 podemos ver a imagem Lena segmentada pelo algoritmo MMP-Intra em macro-blocos de dimensões de 16x16 píxeis. Caso seja necessário é utilizada uma técnica de preenchimento para inserir linhas e/ou colunas por forma a garantir que toda a imagem é segmentada em blocos de tamanho fixo.

Com a existência de blocos de duas dimensões o esquema de segmentação de blocos teve de ser alterado. O novo esquema de segmentação bidimensional permitia assim a segmentação dos blocos em ambas as direções vertical e horizontal. O novo processo de segmentação adotado pelo MMP para blocos bidimensionais utilizava um esquema simples de alternação de sentidos. Como se pode ver na Figura 2.4 um sentido inicial era definido na inicialização do algoritmo, este sentido era utilizado na primeira segmentação de cada bloco. Todas as segmentações posteriores utilizariam o sentido alternado do utilizado anteriormente.

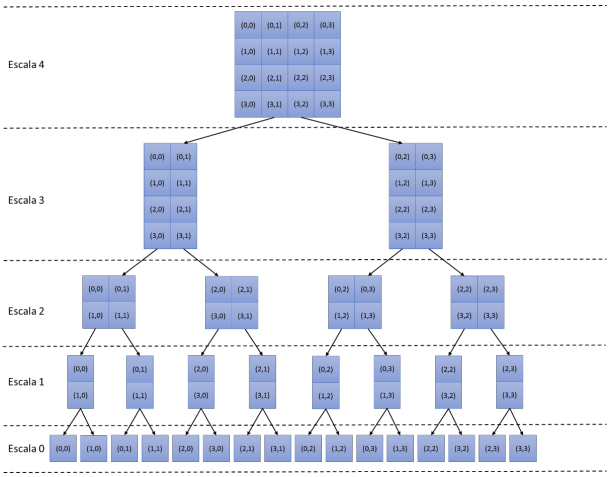

Figura 2.4: Segmentação bidimensional [14]

Embora o novo formato de divisão de dados aliado com o novo esquema de segmentação tenham obtido resultados superiores quando comparados com os originais, verificou-se que dependendo das características da imagem, o sentido escolhido durante a inicialização do algoritmo tinha um impacto significativo na qualidade de compressão final. Para corrigir a dependência existente entre a qualidade de compressão e a segmentação escolhida na inicialização do algoritmo um novo esquema de segmentação de blocos foi desenvolvido. Este esquema permite a segmentação do bloco em ambos os sentidos, ou seja, em vez do sentido a segmentar ser dependente do sentido anterior, o algoritmo segmenta os blocos em ambos os sentidos, vertical e horizontal, sempre que possível.

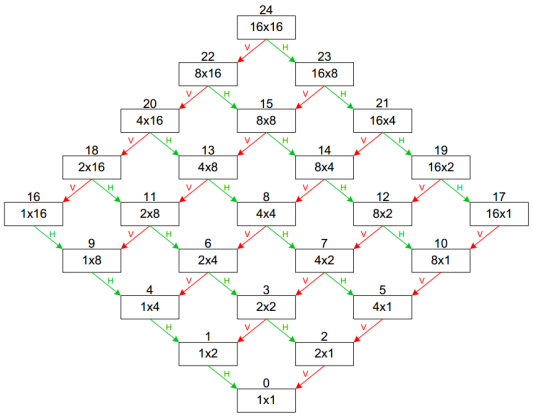

Figura 2.5: Segmentação Flexível [14]

O novo esquema de segmentação flexível [14] permite obter melhores resultados inserindo um maior domínio de escalas de bloco, passando de 9 escalas para 25 escalas diferentes como está ilustrado na Figura 2.5. Embora o novo esquema obtenha resultados de compressão superiores, existem algumas desvantagens ao seu uso:

- Ao existirem mais escalas disponíveis a complexidade computacional aumenta. No esquema inicial, cada nó das árvores de segmentação de bloco apenas tem no máximo 2 filhos, utilizando o novo esquema de segmentação cada nó das árvores de segmentação do bloco pode ter até um total de 4 filhos.

- No esquema de segmentação original existe a necessidade de utilizar apenas uma *flag* para sinalizar a segmentação do bloco. Tendo em conta que no novo esquema de segmentação flexível o bloco pode ser segmentado em qualquer direção a qualquer momento é necessário incluir mais *flags* de sinalização. Desta maneira a cadeia de *bits* passou a ter 3 *flags* distintas que sinalizam o tipo de segmentação: (1) não existe segmentação, (2) segmentação horizontal e (3) segmentação vertical.

2.4.2 Predição *Intra-frame*

Como foi explicado no capítulo 2.3 a natureza adaptativa do dicionário do MMP tem um impacto no processo de codificação. Esta baseia-se na distribuição probabilística do domínio do sinal de entrada aproveitando a redundância espacial do mesmo para criar padrões que se possam utilizar para codificar vários blocos ao longo de todo o sinal, diminuindo o número de segmentações. Tendo em conta essa característica foi desenvolvido um novo modelo de codificação que permite ao MMP-Intra maximizar os resultados através da manipulação do

sinal de entrada alterando a distribuição probabilística do bloco a codificar por forma a gerar picos de grande incidência. Para tal foram utilizadas técnicas de codificação preditiva que permitem a codificação não do sinal original mas sim de um sinal de resíduo. As técnicas de predição utilizadas para esse efeito foram baseadas nas técnicas de predição *intra-frame* utilizadas pelo algoritmo H.264/AVC.

As técnicas de predição *intra-frame* são técnicas de predição espaciais usualmente utilizadas em codificação de imagem e vídeo. O nome *intra-frame* é derivado do facto de este tipo de técnicas apenas utilizar como contexto a informação obtida existente na própria imagem (*frame*) a codificar, não tendo qualquer dependência temporal relativa a outras *frames*.

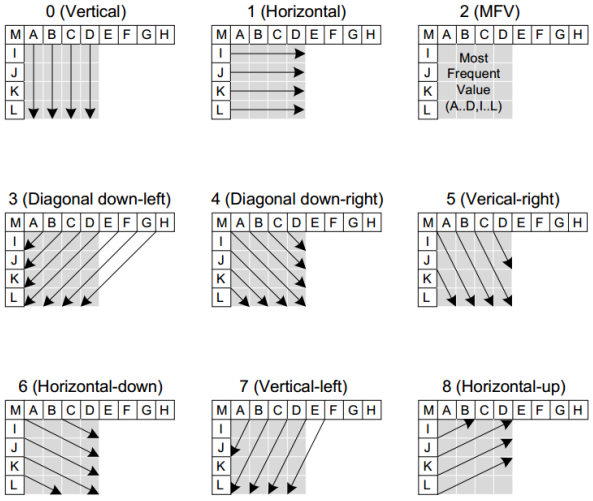

Figura 2.6: Modos de predição *intra-frame* [13]

Na Figura 2.6 são apresentados os modos de predição utilizados no MMP-Intra que foram implementados tendo como base as técnicas de predição utilizadas no algoritmo H.264 [15][16]. Estes podem ser divididos em dois tipos distintos de predições: predições direcionais (1) e predições uniformes (2).

As predições direcionais (1) são predições que efetuam uma extensão linear de uma determinada fronteira do bloco numa direção escolhida. A natureza linear deste tipo de modos de predição permite um bom desempenho para texturas de orientação uniforme e para texturas estruturadas. Uma das desvantagens de utilizar este tipo de predições é que a mesma introduz artefactos na predição gerada, estes tendem a ter mais incidência na predição de blocos de maior tamanho.

As predições uniformes (2) são predições que utilizam toda a vizinhança existente no bloco para gerar um bloco de predição de padrão uniforme. O algoritmo MMP-Intra apenas utiliza um modo de predição uniforme, o *Most-Frequent-Value* (MFV). Este modo de predição tal

como o nome indica efetua uma busca na fronteira do bloco pelo valor mais utilizado, ou seja o algoritmo percorre todos os pixéis da fronteira alvo e efetua uma contagem do número total de vezes que cada valor é utilizado, por fim a predição gerada será constituída por um padrão uniforme contendo apenas o valor encontrado anteriormente.

Tendo em conta que os modos de predição têm o seu desempenho dependente do tamanho do bloco a prever o MMP-Intra não utiliza todos os modos de predição sempre que possível. Como para blocos de maior dimensão as predições tendem a obter maiores distorções, para a maior escala existente (16x16) o algoritmo utiliza apenas quatro modos de predição distintos, sendo eles o modo Horizontal, o modo Vertical, o modo de Plano e o modo MFV. Para as restantes escalas o algoritmo utiliza todos os modos disponíveis, tendo apenas como critério eliminatório restrições de natureza espacial, onde não existe fronteira passível de gerar a predição.

Least Square Error Prediction

Um dos problemas das predições direcionais é que estas apenas conseguem prever com exatidão blocos uniformes ou blocos com fronteiras de direção fixa que seja contemplada por uma das direções. Na presença de blocos com fronteiras de direção arbitrária os métodos de predição anteriormente descritos tendem a obter resultados significativamente inferiores pois apenas exploram as propriedades das fronteiras quando a mesma se encontra numa das bordas utilizadas pelo algoritmo e quando a mesma se prolonga numa direção fixa.

O MMP-Intra implementa um método de predição recorrendo a técnicas de erros quadráticos, que permite ao algoritmo prever com maior exatidão blocos que contenham fronteiras de direção dinâmica. Este método de predição tem o nome de *Least-Square Prediction* (LSP) [17].

O LSP é um método de previsão estatístico que utiliza uma técnica de *point estimation* [18] para cada amostra da população a prever. A técnica utiliza um modelo de Markov com uma janela de treino de dados já anteriormente codificados para encontrar os melhores coeficientes que serão utilizados para prever cada píxel. O processo de encontrar os melhores coeficientes é efetuado para cada píxel mantendo o mesmo modelo mas sendo utilizada uma janela de treino diferente para cada um, desta forma o algoritmo obtém um contexto mais fidedigno ao contrário do que acontece com as predições direcionais.

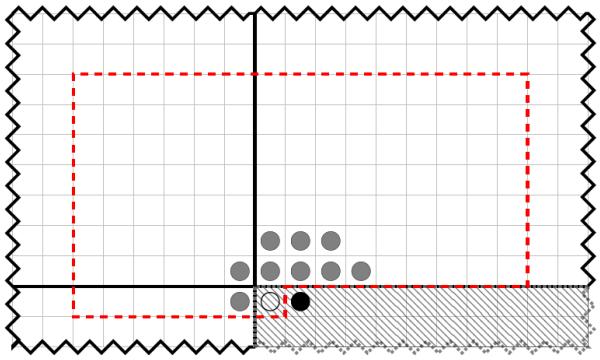

Figura 2.7: Janela de treino da predição LSP [17]

A Figura 2.7 demonstra um exemplo de uma janela de treino para prever um ponto da amostra. Como podemos ver a janela de treino é sempre formada por um conjunto de pixéis que faça fronteira com o píxel a codificar, sendo sempre constituída por pontos já codificados.

2.4.3 Codificação do Bloco

Como indicado anteriormente o MMP-Intra utiliza os modos de predição durante a fase de codificação. No início da fase de codificação do bloco, o algoritmo efetua a predição do bloco utilizando todos os modos de predição *intra-frame* disponíveis.

Figura 2.8: Predição gerada para a imagem Lena (esquerda) e resíduo gerado para a imagem Lena (direita)

Por forma a obter distribuições probabilísticas com grandes picos de incidência, o MMP-Intra codifica não o bloco de dados original, mas sim um bloco de resíduo obtido da diferença entre o bloco previsto e o bloco original obtendo assim distribuições probabilísticas que podem ser mapeadas utilizando modelos gaussianos generalizados. Os picos na distribuição probabilística serão tanto maiores quanto melhor a fidelidade do bloco previsto relativamente ao bloco original, pois quanto mais próximos forem os valores previstos dos valores originais menor será o resultado do resíduo, criando assim resultados picos perto de 0. A

Figura 2.8 demonstra a imagem prevista utilizando os métodos *intra-frame* (esquerda) e o resíduo entre a mesma e a imagem original (esquerda). Como se pode verificar o resíduo tem padrões muito mais homogéneos do que a imagem original. Em um caso ótimo em que a predição conseguisse prever com 100% de exatidão todos os blocos da imagem, o resíduo seria sempre um bloco com todos os valores a zero, permitindo dessa maneira representar toda a imagem apenas com um padrão do dicionário.

Uma vez geradas as predições para todos os modos disponíveis e obtidos os seus resíduos o algoritmo efetua o processo normal de quantificação e segmentação por forma a gerar a árvore de segmentação do bloco. O MMP-Intra testa todas as possibilidades, ou seja, o algoritmo irá segmentar todos os blocos até ao nível mais baixo. Após a árvore ter sido construída, é efetuada uma passagem em profundidade sobre a árvore para encontrar qual a subárvore que melhor representa o bloco em questão segundo um critério $R(D)$.

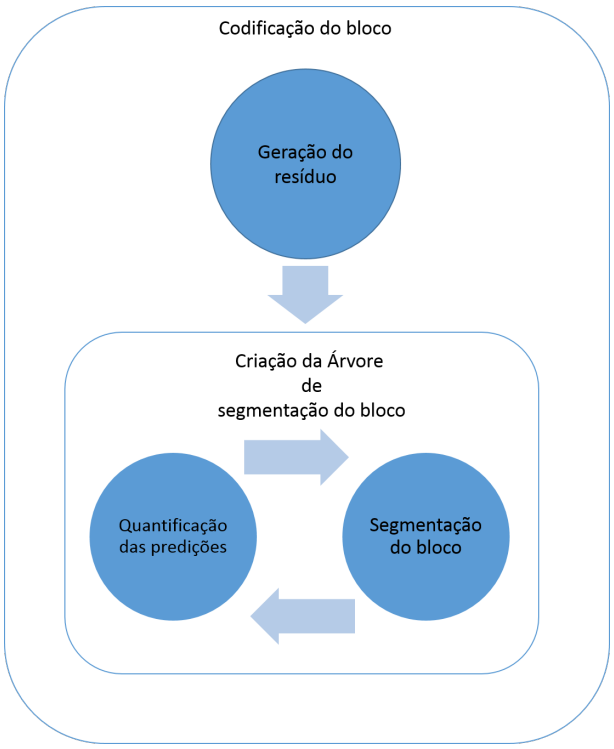

Figura 2.9: Modelo de codificação com predição

A Figura 2.9 demonstra o novo processo de codificação de um macro-bloco utilizando os métodos de predição *intra-frame*. Sabendo que o sinal de entrada (resíduo) segue uma distribuição probabilística gaussiana generalizada, com picos de incidência perto de 0, o MMP--Intra consegue efetuar uma construção dinâmica do dicionário aumentando o número de blocos com maior probabilidade de serem escolhidos e diminuindo o número de blocos com

uma probabilidade inferior de serem escolhidos. Para isso apenas é necessário que a maioria dos valores gerados se centrem perto dos picos encontrados na distribuição probabilística, neste caso valores próximos de 0.

2.4.4 Esquema de Predição Adaptativo

A versão original do MMP-Intra utilizava predição para blocos de tamanho fixo, ou seja o passo de predição era apenas efetuado para os blocos de maior nível (macro-blocos). Embora esta versão tenha obtido melhores resultados que o seu antecessor era possível obter melhores resultados utilizando um esquema de segmentação de predição adaptativo. Uma nova versão do MMP-Intra foi desenvolvida onde os métodos de predição são utilizados para blocos de tamanho variável.

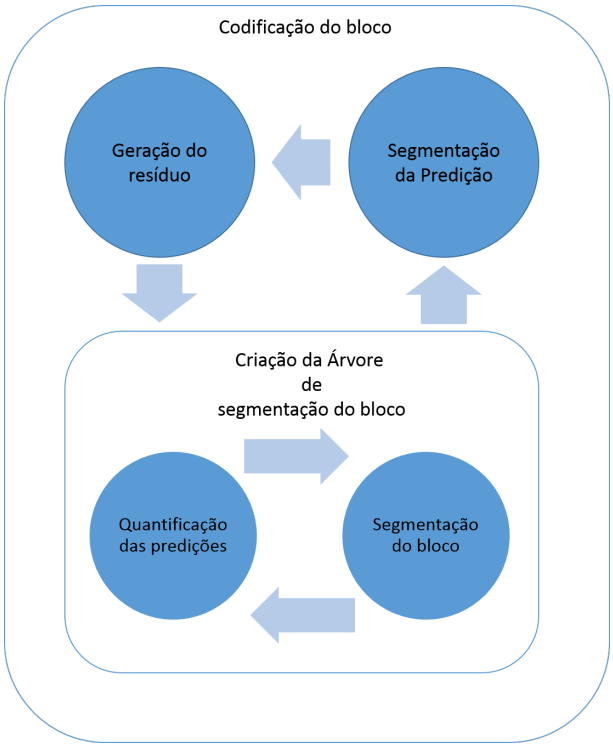

Figura 2.10: Esquema de predição adaptativo

Como se pode ver na Figura 2.10 um novo passo foi adicionado à codificação dos blocos. Após o processo de codificação do macro-bloco ter terminado o algoritmo efetua a segmentação do mesmo. Todo o processo descrito na secção 2.4.3 é então efetuado para cada um dos segmentos. Este processo é executado de forma recursiva, gerando assim uma árvore de segmentação da predição semelhante à árvore de segmentação do bloco. Para cada nó da

árvore de segmentação da predição está associado o modo de predição escolhido para representar esse bloco, tal como a melhor árvore de segmentação para codificar o bloco resíduo associado ao modo de predição escolhido.

A árvore de segmentação de predição é gerada em profundidade, ou seja todos os filhos de um dado nó são expandidos antes de se passar para a expansão de outro nó do mesmo nível. Quando todos os filhos de um dado nó são expandidos o algoritmo verifica qual a melhor forma de representar o nó em questão, ou utilizando a árvore de segmentação do resíduo do nó pai, ou concatenando as árvores de segmentação dos resíduos dos segmentos, filhos. Caso seja melhor utilizar o nó pai, o algoritmo efetua imediatamente a poda, removendo os nós filhos da árvore de segmentação da predição antes de expandir o próximo nó. Isto significa que no fim do processo de codificação do bloco o resultado será uma árvore podada de segmentação da predição.

2.4.5 Taxa-distorção

Como explicado anteriormente, ao contrário do algoritmo original, o MMP-Intra cria toda a árvore de segmentação de predição e de segmentação de resíduo. Ambas estas árvores têm de ser podadas por forma a encontrar a melhor forma de codificar cada bloco. Para tal, o MMP-Intra utiliza uma função de taxa-distorção. A função taxa-distorção utilizada recorre a multiplicadores de *Lagrange* por forma a penalizar os aumentos de taxa.

$$J = D + \lambda R \tag{2.8}$$

A Equação 2.8 define a função taxa-distorção utilizada, onde D é a distorção entre o elemento original e o elemento codificado, λ é um multiplicador de Lagranje, com um peso definido como parâmetro de entrada do algoritmo e R é o custo de representação de um dado bloco em específico, que inclui o custo de representação do índice do *codebook* escolhido e as *flags* de sinalização necessárias. O valor de λ permite o controlo da qualidade de compressão. Utilizando um λ mais baixo, as penalizações dos custos associados à representação dos blocos será menor, isto irá aumentar a qualidade da imagem diminuindo consequentemente a taxa de compressão. Por sua vez, o aumento de λ irá aumentar a penalização dos custos de representação dos blocos criando assim uma imagem com um maior número de perdas e consequentemente maior taxa de compressão. No limite podemos utilizar um valor nulo de λ para obter uma codificação sem perdas.

2.4.6 Cadeia de *bits*

O MMP original apenas necessitava de sinalizar a existência ou não de segmentações utilizando uma *flag* para o fazer. Com a implementação das predições intra e do novo esquema de predição adaptativo foi necessário utilizar três *flags* distintas: (1) não segmenta nem a predição nem o resíduo, (2) segmenta a predição mas não o resíduo e (3) segmenta a predição e o resíduo. Ao incluir o esquema de segmentação flexível duas novas *flags* tiveram de ser adicionadas às três já existentes ficando assim com cinco *flags* distintas: (1) sem segmentação do resíduo nem da predição, (2) segmentação do resíduo e da predição na vertical, (3) segmentação do resíduo da predição na horizontal, (4) segmentação apenas do resíduo na (5) horizontal e segmentação apenas do resíduo na vertical.

2.4.7 Conclusão

Com a implementação dos esquemas de predição e o particionamento flexível o MMP-Intra consegue obter uma qualidade de compressão superior aos estados da arte atuais.

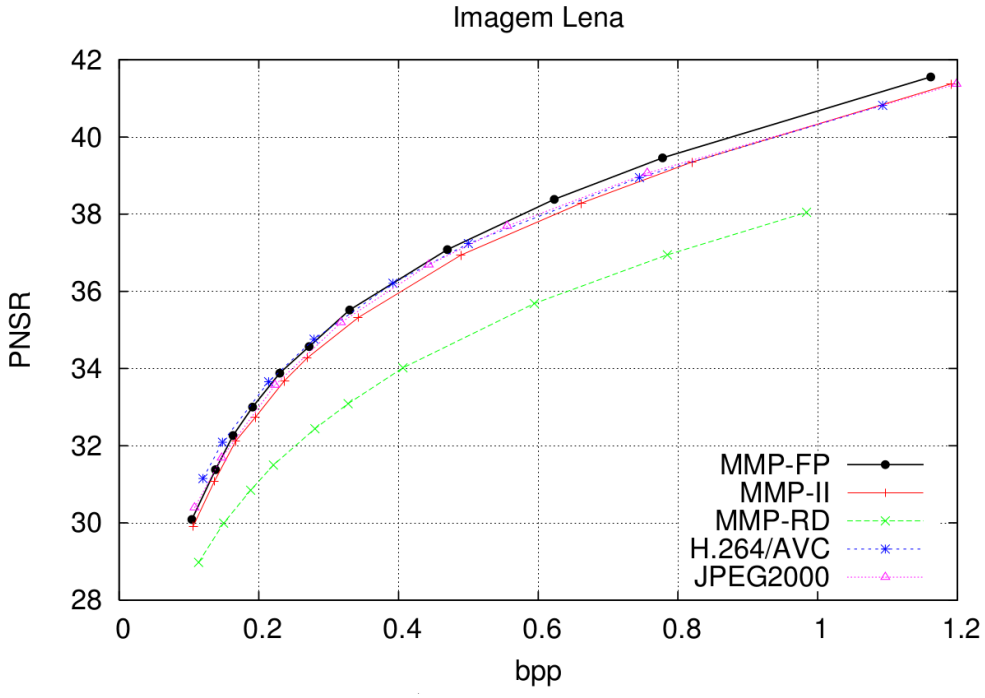

Figura 2.11: Comparação de qualidade/taxa de codificação entre o MMP e estados da arte [14]

Como se pode ver na Figura 2.11, utilizando técnicas de predição *intra-frame* aliadas a um particionamento de bloco flexível o MMP consegue atingir taxas de compressão superiores

ao H264 e JPEG200 na codificação de imagens suaves.

Figura 2.12: Comparação entre imagem original e predição para a imagem Lena para $\lambda = 10$ (superior) e $\lambda = 300$ (inferior)

A Figura 2.12 efetua uma comparação entre a imagem original e a imagem de predições gerada pelo algoritmo. A exatidão da predição e consequentemente a qualidade de compressão estão irreversivelmente condicionados pelo λ associado. Como se pode verificar não se consegue ver a olho nu nenhuma diferença entre a imagem original e a predição gerada para um $\lambda = 10$, no entanto para um $\lambda = 300$ já existem diferenças visíveis a olho nu.
A Figura 2.13 compara a segmentação da imagem Lena para um $\lambda = 10$ e um $\lambda = 300$. Tal como indicado anteriormente, para valores de λ pequenos o algoritmo efetua uma maior segmentação pois a penalização das *flags* de sinalização é inferior.
Embora os resultados de qualidade de compressão sejam superiores aos do codificador H.264, o poder computacional necessário para obter estes resultados inviabilizam o seu uso, traduzindo-se em tempos de execução demorados.

Lambda	Limite *Codebook* 1024	Limite *Codebook* 50000
$\lambda = 10$	2091,878 (s)	5643,951 (s)
$\lambda = 50$	1552,451 (s)	2444,698 (s)
$\lambda = 300$	1093,441 (s)	1176,888 (s)

Tabela 2.4: Tempos de execução do MMP-Intra para a imagem Lena

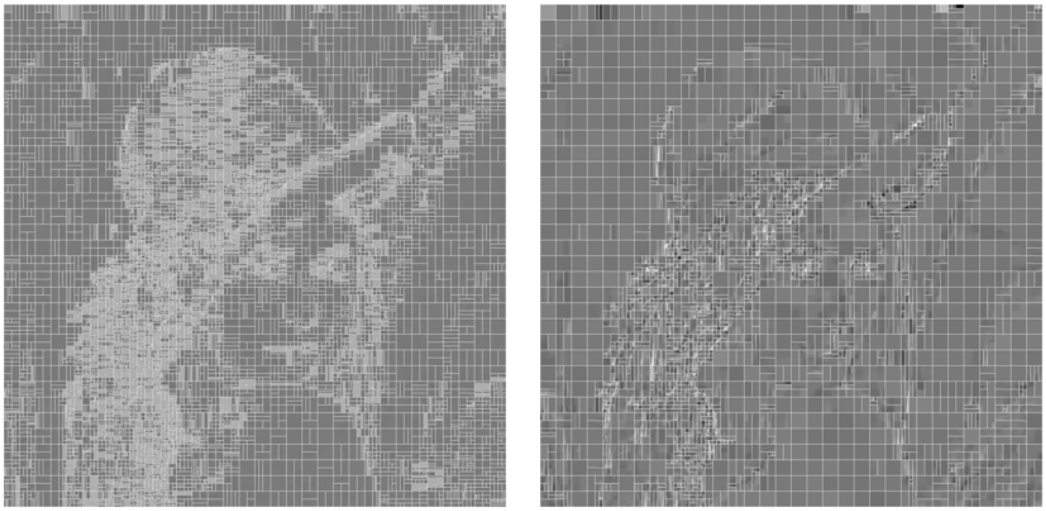

Figura 2.13: Comparação entre a segmentação da imagem Lena para $\lambda = 10$ (esquerda) e $\lambda = 300$ (direita)

A Tabela 2.4 demonstra os tempos de execução para valores de λ cuja qualidade da imagem descodificada não se degrade ao ponto de inviabilizar o uso do algoritmo. O algoritmo MMP-Intra permite limitar o crescimento do dicionário. Como é obvio quanto maior for o crescimento do dicionário maior será a probabilidade de existirem blocos de escalas superiores que poderão ser utilizados permitindo assim uma melhor taxa de compressão. No entanto o poder computacional necessário também será superior. Como podemos ver na Tabela 2.4, para um $\lambda = 10$ o tempo aumenta num fator de aproximadamente 2.7 quando aumentamos o limite do dicionário de $2091,878$ para $5643,951$.

Podemos assim concluir que embora o algoritmo MMP-Intra consiga competir com os estados da arte atuais na codificação de imagens em escalas de cinza, o mesmo é inviável devido ao poder computacional necessário para o fazer.

Capítulo 3

Computação de Alto Desempenho

Este capítulo tem como objetivo contextualizar o leitor no ambiente de sistemas paralelos de alto desempenho. Inicialmente, é descrita sumariamente a evolução das arquiteturas sequenciais para as arquiteturas paralelas. De seguida será efetuada uma introdução aos conceitos de *Graphical Processor Units* (GPU) e *General Purpose Graphical Processor Units* (GPGPU). Por fim são explicados os conceitos referentes à programação de algoritmos para sistemas paralelos utilizando a arquitetura CUDA.

3.1 Introdução

O conceito de computação de alto desempenho nasceu no início da década de 1960 com a criação dos primeiros supercomputadores [19]. Os supercomputadores são computadores que na generalidade dos casos implementam arquiteturas de computação paralela que conseguem atingir débitos computacionais superiores às distintas arquiteturas contemporâneas. Estes foram criados com o intuito de permitir resolver problemas de elevada complexidade computacional usualmente encontrados em diversas áreas de investigação tais como matemática e física, que embora fossem teoricamente exercíveis em computadores convencionais, o mesmo seria impossível devido ao tempo de processamento necessário. Para obter alto desempenho, os supercomputadores utilizam uma arquitetura de computação paralela visando técnicas de fatorização de problemas. Os primeiros supercomputadores continham apenas algumas dezenas de processadores. No entanto devido ao sucesso que os mesmos obtiveram e ao constante aumento da complexidade de problemas, rapidamente evoluíram para atingirem magnitudes nos milhares de processadores tais como o TIANHE-2 [20] com 3,120,000 núcleos de processamento ou o SEQUOIA [21] com 1,572,864 núcleos de processamento.

Embora a arquitetura paralela tenha tido uma maior incidência em áreas como a computação de alto desempenho, hoje em dia é uma tecnologia bastante comum no quotidiano de qualquer pessoa podendo ser visto nas microarquiteturas atuais dos processadores dos computadores pessoais como acontece com as microarquiteturas Core (Intel) [22] e Bulldozer (AMD) [23].

3.2 Evolução dos Sistemas Paralelos

No início da década de 1950, surgiu o que é conhecido como sendo o primeiro computador comercial, UNIVAC-I [24]. O UNIVAC-I e os seus sucessores utilizaram uma arquitetura de *harwade* à qual damos o nome de Arquitetura John Von Neumann [25]. Esta foi a primeira arquitetura que contemplava o armazenamento tanto de dados como de instruções em sistemas eletrónicos, ao contrário dos seus antecessores que tinham de ser programados fisicamente. Esta arquitetura implementava um esquema de execução sequencial, isto significa que os sistemas estavam preparados para efetuar uma única tarefa de cada vez. Este tipo de esquema de execução revelou-se muito ineficiente pois sempre que era necessário efetuar uma operação dependente de um periférico de entrada ou saída, o processador era obrigado a parar todo o processamento até que os dados estivessem disponíveis.

Por forma a colmatar este problema foram desenvolvidos sistemas sequenciais que permitiam a execução de várias tarefas em simultâneo. A estes sistemas foram dados o nome de sistemas multitarefas [26]. Os sistemas multitarefas permitiam a execução de vários processos em simultâneo. Embora estes sistemas permitissem que vários processos estivessem a ser processados em simultâneo, os mesmos não eram executados em paralelo, continuando cada processo a ser executado de forma sequencial. Um processo era substituído por outro na existência de esperas ou paragens de processamento.

Inicialmente os sistemas multitarefas implementavam um esquema de cooperação [26]. Este esquema permitia que um processo fosse programado para cooperar com outros processos, cedendo tempo de computação quando o mesmo não o estava a utilizar. Um grande problema deste esquema devia-se à grande dependência que o mesmo tinha com os processos. Uma vez que era delegado a cada processo a cedência do processador, um processo mal programado poderia monopolizar o tempo de processamento. Para eliminar esta dependência foi desenvolvido um esquema de multitarefa preemptivo [26]. Este esquema permitia que todos os processos obtivessem tempo de computação delegando o escalonamento do seu uso ao sistema operativo utilizando técnicas de multiplexagem. Com a evolução dos sistemas mul-

titarefas e dos respetivos escalonadores de processos, os sistemas informáticos começaram a obter aumentos de desempenho significativos, isto fez com que o *software* passa-se a ser desenhado não como um único processo, mas como um conjunto de processos que funcionavam de forma cooperativa para um único fim. Uma das desvantagens deste tipo de soluções passou a ser o custo associado à criação de um processo e a comunicação entre processos. Ambos os problemas estavam relacionados com a alocação de recursos por parte de cada processo, pois sendo executados em fluxos separados cada um necessitava que o sistema alocasse recursos específicos para cada.
Por forma a otimizar a criação de diferentes fluxos de execução e permitir uma fácil comunicação entre os mesmos foram desenhados sistemas de divisão de processos a que damos o nome de *threads* [27]. Uma *thread* é um componente de um processo que descreve um fluxo de execução isolado, podendo um processo conter várias *threads* a ser executadas em simultâneo, sendo delegado ao sistema o seu escalonamento utilizando esquemas de multiplexagem. A vantagem de utilizar várias *threads* e não processos deve-se principalmente ao facto das mesmas partilharem recursos tais como o espaço de endereçamento de memória e o contexto (pertencente ao seu processo) permitindo desta maneira uma fácil comunicação entre as várias *threads* e uma rápida mudança entre os vários fluxos de execução [27].
Todas essas técnicas permitiam dar ao utilizador a ilusão de que várias tarefas pudessem ser executadas em simultâneo devida à multiplexagem de tarefas, no entanto a arquitetura estava limitada a efetuar a execução de forma sequencial, pelo facto de apenas ter uma unidade de processamento.

3.2.1 Classificação de Arquiteturas

Tendo em conta a evolução das arquiteturas nas várias frentes da computação, uma grande quantidade de tipos de arquiteturas diferentes foram sendo desenhadas e implementadas. Por forma a conseguir classificar as mesmas em 1966, Michael J. Flynn propôs uma taxonomia por forma a classificar todas as arquiteturas que ainda hoje é aceite. A taxonomia de Flynn [28][29] propôs a divisão das arquiteturas em quatro tipos distintos tendo em conta as suas características de fluxo e dados:

- *Single Instruction Single Data* (SISD): As arquiteturas SISD são classificadas como tendo um único fluxo de execução operando sobre um único dado de cada vez.
- *Single Instruction Multiple Data* (SIMD): As arquiteturas SIMD são classificadas como tendo um único fluxo de execução, processando uma única instrução sobre um

conjunto de dados por forma a processar mais resultados na mesma unidade de tempo.

- *Multiple Instruction Single Data* (MISD): As arquiteturas MISD são classificadas como tendo diversos fluxos de execução, processando diversas instruções em simultâneo sobre um único dado.

- *Multiple Instruction Multiple Data* (MIMD): As arquiteturas MIMD são classificadas como tendo diversos fluxos de execução, processando diversas instruções sobre um conjunto de dados.

3.2.2 Arquitetura de Computação Paralela

A arquitetura de computação paralela é uma arquitetura que utiliza uma técnica de dividir e conquistar, isto é, tem como base a fatorização de problemas. Uma arquitetura paralela tem como objetivo a divisão de um problema de grande complexidade em vários problemas independentes entre si de complexidade mais reduzida que por sua vez possam ser executados em paralelo. Para isso esta arquitetura tem como pressuposto a utilização de várias unidades de processamento, tirando partido da possível paralelização da execução dos sub-problemas. Embora a computação paralela fosse usualmente utilizada na computação de alto desempenho e tenha tido uma grande adoção na fabricação de GPUs, a mesma apenas teve um maior foco recentemente. Isto deve-se tanto à preocupação do aumento exponencial de consumo energético inerente ao aumento da frequência de processamento pelas CPUs convencionais como ao limite do paralelismo que pode ser explorado num CPU superescalar [30]. Devido a esses fatores os fabricantes de sistemas informáticos migraram as suas implementações sequenciais para implementações paralelas passando a ser comum a fabricação de sistemas que integram CPUs com vários núcleos de processamento por forma a obter um aumento de desempenho tais como Computadores Pessoais (PC) [22][23], Tablets [31] ou Smartphones [31].

Tipologias de Paralelismo

As arquiteturas paralelas podem ser definidas consoante o tipo de paralelismo que implementam, sendo considerados os três principais tipos de paralelismo: (1) paralelismo binário, (2) paralelismo de instruções e (3) paralelismo de tarefas. Hoje em dia é bastante comum uma única arquitetura utilizar mais do que um tipo de paralelismo.
O paralelismo binário refere-se à capacidade de um sistema conseguir processar diversos bits por ciclo de relógio ou seja, utilizando um único ciclo de relógio ser possível a adição de dois

valores de 32 bits. Este tipo de paralelização é hoje em dia utilizado em todas as arquiteturas, quer paralelas quer sequenciais. Ao tamanho máximo utilizado pelo sistema para efetuar as operações damos o nome de *word*.

O paralelismo de instruções refere-se à capacidade de um sistema conseguir executar mais do que uma única instrução por ciclo de relógio. Isto é possível utilizando duas técnicas distintas, (1) sistema de linha de montagem para execução de instruções (*pipeline*) ou (2) emissão de mais do que uma instrução em simultâneo (superescalar) [32]. Ambas as técnicas podem e são implementadas em simultâneo para atingir um maior paralelismo. A execução de uma instrução necessita que o sistema efetue um conjunto de passos de forma sequencial. Sabendo que os diversos passos utilizam recursos do sistema diferentes para serem processados, é então possível criar um sistema de linha de montagem utilizando os vários tipos de recursos ao mesmo tempo. Assim o sistema poderá implementar um sistema de linha de montagem tendo em conta o número de passos necessário por instrução [32].

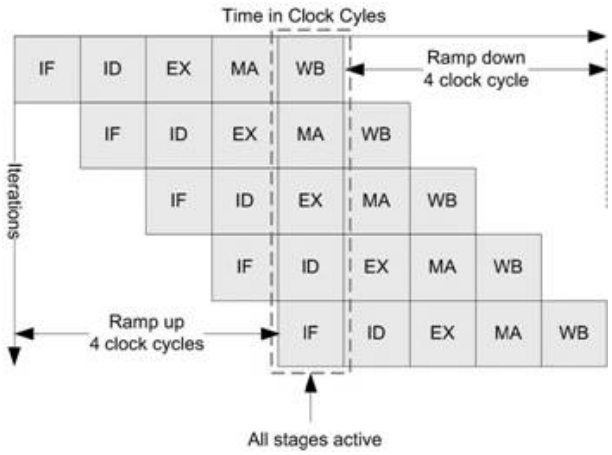

Figura 3.1: Pipeline de 5 estados de uma arquitetura RISC [33]

A Figura 3.1 demonstra o processo de execução de uma *pipeline* de 5 estados numa arquitetura *Reduced Instruction Set Computer* (RISC). A execução das várias instruções não é efetuada sempre na mesma unidade de processamento física, um sistema é na generalidade dos casos desenhado por forma a delegar a execução de diferentes tipos de instruções a diferentes unidades de processamento, as quais são otimizadas para esse efeito. É possível utilizar esta característica para benefício do sistema utilizando uma técnica de agrupamento de instruções que permite que as mesmas sejam emitidas e executadas em paralelo por diferentes unidades de processamento pertencentes ao mesmo sistema. A este tipo de sistemas damos o nome de sistemas superescalares [34].

O paralelismo de tarefas é execução de mais do que uma tarefa de forma paralela distribuída por várias unidades de processamento. Esta é uma técnica semelhante à introduzida no parágrafo anterior, utilizando um contexto de granularidade mais grossa paralelizando ao nível de blocos de código e não das instruções [35].

Condições de Paralelismo

Sabendo que a computação paralela tende a obter desempenhos computacionais significativamente superiores quando comparado com arquiteturas sequenciais, é apenas lógico que os programadores e utilizadores efetuem a passagem dos seus algoritmos baseados em arquiteturas sequências para arquiteturas paralelas. No entanto, mesmo com a massificação de sistemas paralelos, o nosso quotidiano está repleto de algoritmos e processos que são executados de forma sequencial. Embora para um grande conjunto de algoritmos o que foi indicado anteriormente seja verdade e os mesmos consigam obter desempenhos significativamente superiores efetuando a migração de arquitetura, a arquitetura de computação paralela tem o seu desempenho altamente dependente das características dos algoritmos a executar.
A característica mais importante de um algoritmo que permite a sua migração para uma arquitetura paralela é a sua fatorização. Na generalidade dos casos esta é a característica eliminatória para uma migração. Isto deve-se aos vários tipos de dependências que os sub-problemas podem ter entre si. As dependências podem ser divididas em três tipos distintos [36]: (1) dependências de dados, (2) dependências de controlo e (3) dependências de recursos. A dependência de dados (1) é definida por dois acessos de memória que referenciem o mesmo local, sendo um deles uma operação de escrita. Isto significa que entre esses dois acessos, um deles irá ser dependente do resultado da operação de escrita do outro. Caso os mesmos sejam executados num ambiente paralelo, poderá ocorrer o cenário em que o processo dependente efetue o acesso de memória antes que o outro processos efetue a sua operação de escrita, causando desta maneira uma incoerência de dados o que poderá revelar resultados não determinísticos, pois o resultado final ficará dependente do escalonamento de execução dos vários processos. As dependências de dados podem ainda ser divididas em quatro tipos distintos:

- Dependências de fluxo: Quando a saída de uma instrução I1 serve de entrada de outra instrução I2 e I1 precede I2 então I2 tem uma dependência de fluxo com I1

    1. $x = 2$
    2. $y = x + z$

- Antidependências: Quando uma instrução I1 escreve sobre um endereço de memória utilizado como entrada de outra instrução I2 e I2 precede I1 então I2 tem uma antidependencia com I1

  1. $x = y + z$
  2. $y = 2$

- Dependências de saída: Quando duas instruções I1 e I2 escrevem sobre o mesmo endereço de memória e I2 precede I1 então I2 tem uma dependência de saída com I1

  1. $x = 2$
  2. $x = 3$

- Dependências de entrada: Quando duas instruções I1 e I2 utilizam um mesmo endereço de memória como entrada e I1 precede I2 então I2 tem uma dependência de entrada com I1

  1. $x = y + 2$
  2. $z = y + 3$

Este tipo de dependências pode ser superado de duas maneiras distintas, ou mantendo a dependência e utilizando um escalonador para garantir que as instruções são executados pela ordem correta não utilizando a natureza paralela da arquitetura, ou eliminando a dependência alterando o código fonte quando possível.
As dependências de controlo são definidas por duas instruções diferentes onde a execução de uma instrução está condicionado pelo resultado da outra. Quando uma instrução I2 tem a sua execução condicionada pelo resultado de uma outra instrução I1, então I2 tem uma dependência de controlo com I1.

1. if($x > 2$) goto 2
2. $y = w + z$

Ambos os tipos de dependências descritos anteriormente são relacionados com o trabalho lógico a ser executado. As dependências de recursos estão relacionadas com utilização de recursos partilhados tais como registos ou unidades de processamento. Isto significa que os recursos existentes no sistema também podem condicionar o paralelismo que se consegue atingir. Um exemplo deste tipo de dependência é quando é necessário que duas instruções

executem uma instrução aritmética em simultâneo num sistema que apenas contenha uma *arithmetic logic unit* (ALU). Neste caso as instruções terão de ser executados de forma sequencial pois apenas existe um recurso disponível.

Quantificação de Ganho

Ao longo do tempo várias leis foram criadas para possibilitar uma quantificação do possível ganho que se pode obter na paralelização de sistemas sequenciais.
A lei de amdahl [37] é uma das formas mais aceites na quantização de ganhos em sistemas paralelos. A mesma define um modelo onde são contempladas ambas as partes, sequencial e paralela, do algoritmo onde demonstra que o possível ganho da paralelização de um algoritmo é dependente do tempo que a sua parte sequencial demora a ser processada.

$$G(N) = \frac{T(1)}{T(N)} = \frac{Ts + Tp}{Ts + \frac{Tp}{N}} \tag{3.1}$$

A Equação 3.1 [38] define o modelo desenhado por Amdahl, como se pode verificar o potencial ganho G com a paralelização de um algoritmo em N unidades de processamento pode ser encontrado utilizando a função $G(N)$ onde Ts é o tempo de execução sequencial e Tp é o tempo de execução paralelo. Como exemplo, quando paralelizando um algoritmo com uma sequencialidade de 20% podemos verificar que,

$$G(N) = \frac{0.2 + 0.8}{0.2 + \frac{0.8}{N}} \tag{3.2}$$

podendo atingir um *speedup* máximo teórico de 5 para quando N tende para mais infinito.

3.3 *Graphical Processor Units*

Com o sucesso inicial da arquitetura de computação paralela nos supercomputadores esta começou a ser utilizada em outras frentes da tecnologia. Uma delas foi em *Graphical Processor Units* (GPUs), estas são placas de circuitos integradas que foram criadas com o objetivo de acelerar a computação gráfica. Inicialmente as mesmas eram utilizadas apenas no processamento de imagem, permitindo o rápido processamento e desenho de linhas, quadrados e outras figuras que seriam posteriormente exibidas em um monitor.
Com a evolução da *computer-generated imagery* (CGI), a área de computação gráfica que engloba todos os métodos de geração de imagens através de sistemas digitais, estas placas tiveram a necessidade de permitir o processamento de cenários mais complexos tais como

cenários realistas em duas dimensões (2D) ou até cenários simples em três dimensões (3D). Com a evolução do poder computacional das GPUs, em meados da década de 1980 era já possível a renderização de cenários 3D simples em tempo real, sendo criado o primeiro jogo 3D comercial, 3D Monster Maze(1981). Esta tecnologia veio popularizar as CGI na criação de conteúdos multimédia e jogos digitais o que veio trazer uma grande popularização das GPUs nos computadores pessoais, traduzindo-se numa grande distribuição das mesmas por todo o mundo.

Tendo em conta o poder computacional das GPUs e a grande massificação das mesmas surgiu a questão de como utilizar as GPUs para outras tarefas que não as de processamento gráfico. Uma das diferenças entre as GPUs relativamente aos CPUs convencionais é que as GPUs tendem a ter um maior número de transístores dedicados ao processamento de dados enquanto as CPUs têm um maior numero de transístores dedicados a controlo de fluxo de manipulação de dados. Uma outra diferença é que as GPUs implementam uma arquitetura de execução altamente paralela quando comparadas com as CPUs. Esta característica permite às GPUs terem um maior desempenho em algoritmos que necessitem de um grande poder de computação e que sejam altamente paralelizáveis, tal como é o caso dos algoritmos de computação gráfica.

Assim foi criada a designação de *General Purpose Graphics Processing Unit* (GPGPU) [39], isto é, o nome dado a uma GPU que seja utilizada para tarefas que não sejam para processamento de imagem. Estas placas começaram a ser populares no inicio da década de 2000, quando foram primeiro introduzidos os *shaders* programáveis [40]. *Shaders* eram programas que corriam na GPU e eram utilizados para alterar a variação de cor por forma a atribuir um efeito de profundidade aos diversos objetos renderizados, ou seja os objetos era renderizados pela GPU e o *shader* era um processo de pós renderização que trabalhava a imagem, chegando mais tarde a ser utilizado para inserção de efeitos especiais nas imagens renderizadas. Uma vez que os *shaders* eram processos programáveis, a comunidade científica começou por testar diversas rotinas de manipulação de matrizes utilizando os mesmos. Um dos problemas dessa altura era que embora fosse possível utilizar as GPUs para processos que não fossem de natureza gráfica, as *API*'s disponibilizadas pelos fabricantes não estavam direcionadas para esse efeito o que tornava a implementação de algoritmos complexos um trabalho difícil e moroso, pois os mesmos tinham de ser reformulados para utilizar as funções de processamento de imagens que pudessem ser representados por triângulos e polígonos.

Tendo em conta os resultados positivos que a migração de algoritmos estava a ter para as GPGPUs um conjunto de linguagens de programação de alto nível foram introduzidas com o intuito de abstrair o programador da natureza de processamento de imagem, delegando

o processo de reformulação do algoritmo para as funções de processamento de imagem ao compilador. Uma destas linguagens era designada por Brook [41]. Esta foi desenvolvida por um grupo de investigadores da Universidade de Stanford liderada por Ian Buck. O objetivo do seu projeto foi o desenvolvimento de um compilador e sistema *runtime* que permitisse o desenvolvimento de algoritmos de propósito geral em sistemas GPGPUs. A linguagem Brook foi a pioneira na introdução de conceitos como *"stream"* e *"kernel"*, conceitos estes ainda utilizados pelas principais tecnologias da mesma área da atualidade tais como a *framework* CUDA [42] e a arquitetura OpenCL [43]. A evolução destas ferramentas, tornou hoje em dia as GPUs numa atraente alternativa para a execução de algoritmos de alto desempenho [44].

3.4 CUDA

A *Compute Unified Device Architecture* (CUDA) foi a plataforma de desenvolvimento para GPGPUs pioneira que contempla a combinação de *harwade* e *software* específicos para o desenvolvimento de soluções em sistemas heterogéneos. A tecnologia CUDA foi uma das sucessoras da linguagem Brook, desenvolvida por uma equipa de investigação da empresa NVIDIA liderada por Ian Buck. Primeiro apresentada em 2006, esta começou por ser exclusivamente uma extensão para a linguagem C (C99) [45], permitindo um ambiente de programação de baixo nível por forma a conseguir obter elevadas performances. Com a maturação da tecnologia, a mesma passou a suportar um maior leque de linguagens de programação tais como C++ [46], Fortran [47], Java [48] e Python [49].

A tecnologia CUDA é dividida em duas *API*s distintas: (1) *Driver API* e (2) *Runtime API*. A *Driver API* é uma *API* de baixo nível, a vantagem da utilização desta *API* é o facto dela permitir um maior controlo sobre a inicialização do sistema, carregamento de módulos cubin [50] e inicialização e controlo de execução através de chamadas de baixo nível. A desvantagem do uso da mesma é o aumento da complexidade e consequentemente do tempo de desenvolvimento das aplicações. A *Runtime API* é uma *API* de alto nível, permitindo uma abstração da maior parte do processo de inicialização. Todo o processo de carregamento e *linking* dos módulos é automatizado sendo delegado à própria *API*. O uso desta *API* permite um desenvolvimento menos complexo e mais rápido. No entanto não permite o carregamento dinâmico de módulos e permite acesso a uma quantidade inferior de informação das placas. Ambas as *API*s utilizam o mesmo modelo de programação paralelo.

Esta secção foca somente a *API* Runtime, pois foi a única empregue no desenvolvimento de

todo o trabalho.

3.4.1 *Hardware*

As GPUs são placas que contêm um grande número de núcleos de processamento, podendo chegar aos milhares de unidades, por exemplo a NVIDIA 980 disponibiliza 2048 cores [51]. Assim sendo os mesmos são utilizados para o que se designa de *massive parallel programming* [52], onde o objetivo é explorar o paralelismo do máximo número de núcleos de processamento possível. Efetuar a migração de um algoritmo de execução sequencial para um algoritmo de execução paralela com milhares de fluxos em simultâneo é uma tarefa de elevada complexidade onde nem sempre o custo de desenvolvimento e adaptação justifica ser suportado.

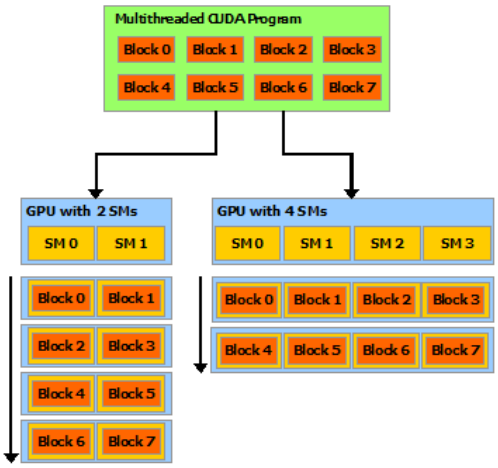

Figura 3.2: Escalabilidade da Arquitetura CUDA utilizando agrupamento de unidades de processamento [53]

Para diminuir a complexidade deste tipo de operações e por forma a permitir um ambiente mais acessível e recetivo, a arquitetura CUDA baseia o seu desenho no agrupamento lógico e físico de núcleos de processamento. O agrupamento físico dos núcleos de processamento é definido pelo nome de *Streaming Multiprocessor* (SM). Esta divisão dos núcleos em SMs facilita a migração dos algoritmos com um único fluxo para diversos fluxos. Como está ilustrado na Figura 3.2 a arquitetura alia a divisão lógica do *harwade* ao modelo de execução, criando dessa maneira um ambiente que permite efetuar uma divisão de trabalho em vários níveis de granularidade, permitindo uma migração faseada.
Outra vantagem de utilizar uma divisão por física de unidades de processamento é que per-

mite que a arquitetura tenha uma escalabilidade flexível. A mesma pode escalar na vertical, utilizando unidades de processamento com um maior poder de computação, aumentando a sua frequência de execução, ou escalar na horizontal, aumentando o número de unidades de processamento disponíveis no sistema.
Cada SM é responsável por efetuar vários fluxos de execução agrupados em paralelo. Para isso a arquitetura CUDA implementa um esquema de execução do tipo Single Instruction Multiple Thread (SIMT). Esta é uma classe de arquiteturas baseada na classe SIMD descrita na secção 3.2.1 onde uma única instrução é executada em paralelo por todo o conjunto de fluxos de execução sobre um conjunto de dados. Um único SM contém vários núcleos de processamento, isto permite que cada *thread* execute o seu fluxo num núcleo independente contendo recursos próprios. Isto permite que mesmo utilizando um esquema SIMD a arquitetura utilize técnicas de ramificação de código.

Compute Capability

Com o passar do tempo a arquitetura CUDA foi amadurecendo. Isto significa que foram sendo efetuadas alterações significativas à mesma tanto na camada lógica como na camada fisica. Por forma a classificar as caracteristicas de cada placa diferentes microarquiteturas foram sendo disoinibilizadas com o lançamento de novas placas, as microarquiteturas lançadas foram, por ordem cronológica: (1)Tesla, (2)Fermi, (3)Kepler e (4)Maxwel. Mesmo utilizando as microarquiteturas para classificar as caracteristicas de cada placa, dentro de cada microarquitetura existem implementações CUDA distintas entre si. Assim foi criada a designação de *Compute Capability*, esta é um valor que permite identificar tanto a a microarquitetura como a implementação dentroda mesma. Para isso a *Compute Capability* é composto por dois digitos separados por um ponto. O digito da esquerda identifica a *major version*, esta é a identificadora de uma grande mudança de implementação identificando a microarquitetura a que a placa pertence. O digito da direita identifica a *minor version*, esta é a identificadora de uma pequena mudança de implementação dentro da de uma microarquitetura. Utilizando ambos os valores é possivel identificar de forma inequivoca as caracteristicas fisicas e lógicas da placa.

3.4.2 Modelo de Programação

A plataforma CUDA baseia-se no fundamento de que um problema pode ser fatorizado em problemas de menor complexidade, os quais poderão ser executados em paralelo nas diversas GPUs. Para isso o modelo de programação utilizado em CUDA contempla três abstrações

chave que facilitam a divisão e sincronização do problema em diversos fluxos que serão por sua vez executados em paralelo. As abstrações são: (1) hierarquia de grupos de *threads*, (2) memórias partilhadas e (3) barreiras de sincronização [53]. Estas três abstrações permitem a fácil implementação de algoritmos paralelos.

Kernel e Stream

A plataforma utiliza como ponto de entrada à execução de trabalho ao dispositivo, funções designadas por *kernels* que se destinam a ser executados pelo dispositivo GPU, com um paralelismo determinado pela grelha de execução aquando do lançamento do *kernel*.
Para declarar e invocar funções, a plataforma disponibiliza uma sintaxe específica. Aquando da declaração de uma função, a mesma pode ser precedida por um de três qualificadores de funções distintos: (1) `__global__` , (2) `__device__` ou (3) `__host__`.

1. **`__global__`**: Este qualificador é utilizado para declarar uma função como sendo um *kernel*. A mesma pode ser invocada pelo *host*, ou por *device*s cuja *compute capability* seja superior 3.x [54] e será sempre executada no dispositivo. Funções do tipo `__global__` têm obrigatoriamente de retornar o tipo void.

2. **`__device__`**: Este qualificador é utilizado para declarar uma função do dispositivo (*device*), esta é apenas invocável do dispositivo e é sempre executada no mesmo. Funções do tipo `__device__` podem retornar qualquer tipo de valor.

3. **`__host__`**: Este qualificador é utilizado para declarar uma função dita de *host*, esta é apenas invocável do *host* e é sempre executada no mesmo. Este qualificador é facultativo, na ausência de qualquer qualificador o sistema classifica o mesmo como sendo de *host*. Funções do tipo `__host__` podem retornar valores de qualquer tipo.

Assim a criação de uma função do tipo *kernel* obriga à utilização do qualificador de função `__global__` durante a sua declaração.

Listing 3.1: Definição de um *kernel*

```
__global__ void kernelExample(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

O exemplo de Listagem 3.1 mostra o código de um *kernel* que recebe três vetores por parâmetro e tem como finalidade efetuar a soma vetorial entre A e B, guardando o resultado correspondente em C. Como se pode ver o mesmo é precedido do qualificador `__global__` e retorna void, tal como obriga as regras do qualificador.
A invocação de funções em CUDA é efetuada da forma convencional definida pela linguagem em questão exceto para funções qualificadas como `__global__`. Qualquer chamada a uma função do tipo `__device__` ou `__host__` é feita utilizando a sua assinatura. Quando se trata de uma função do tipo `__global__` a mesma tem de utilizar uma sintaxe específica para definir a geometria com a qual a mesma será executada. A geometria define a forma como o *kernel* será dividido logicamente entre os vários recursos, este tema será aprofundado no próximo capítulo. A sintaxe obriga a que seja introduzida uma expressão do tipo «<x, y, z, w» onde x é uma variável obrigatória do tipo `dim3` que representa as dimensões da rede, y é uma variável obrigatória do tipo `dim3` que representa as dimensões dos blocos, z é uma variável facultativa do tipo `size_t` e representa o numero de *bytes* de memória partilhada que será alocada dinamicamente por bloco. Finalmente w é uma variável facultativa do tipo `cudaStream_t` e especifica em qual *stream* o *kernel* irá ser executado. Se y não for definido o sistema irá utilizar 0 como valor por omissão, não sendo alocada nenhuma memória partilhada dinamicamente. Se w não for definido, o sistema irá utilizar a *stream* 0.

Listing 3.2: Invocação de um *kernel*

```
...
dim3 grid(512,1,1);
dim3 block(256,256,1);
kernelExample<<grid,block>>(A,B,C);
...
```

O exemplo acima lista a chamada do *kernel* definido anteriormente; este irá ser executado com um geometria de rede de 512x1x1 e uma geometria de bloco de 256x256x1. Como os parâmetros facultativos não foram definidos, o sistema ira utilizar os valores por omissão.
Embora o *kernel* seja o ponto de entrada para inserir fluxos de trabalho na GPU, este não pode ser executado sem que seja indicado à mesma algum tipo de contexto como o fluxo deve ser tratado. Para isso a *API* CUDA disponibiliza um conjunto de métodos que permite ao utilizador criar contextos sobre os quais os *kernels* serão executados. Uma das características associadas a estes contextos são *streams*. Uma *stream* identifica um fluxo de execução na

GPU e, é sobre o mesmo que os vários *kernels* ou cópias de/para a GPU irão ser executadas. Uma única *stream* pode conter vários *kernels* e cópias em simultâneo no entanto estas são sempre executadas de forma sequencial pela ordem com que foram chamadas para a *stream* para a GPU. A emissão de várias tarefas em simultâneo para a mesma *stream* garante uma lógica temporal sequencial, permitindo desta maneira que várias tarefas dependentes possam ser enviadas para a GPU para execução sem que haja a necessidade implementar uma lógica de sincronização entre as mesmas. Quando se pretende emitir vários *kernels* e cópias que seja independentes uns dos outros, estes deverão ser emitidos para *streams* diferentes, pois de outra maneira serão executados sequencialmente. O contexto CUDA permite a definição de diversas *streams* para que quando possível (ausência de dependências de recursos) o trabalho emitido para *streams* diferentes possam ser executados de forma paralela.
Como indicado acima quando é emitida uma operação onde a *stream* de execução não é identificada, o sistema utiliza uma *stream* identificada pelo valor 0. A *stream* 0 tem características diferentes das restantes, sendo implicitamente criada pelo próprio sistema e não permitindo a execução paralela de qualquer outra *stream*.

Hierarquia de Grupos de *Threads*

Uma das abstrações chaves da plataforma CUDA é a hierarquia de grupos de *threads*. A plataforma CUDA divide logicamente as *threads* em vários tipos de conjuntos hierárquicos. Um exemplo da utilização destas divisões lógicas pode ser vista na secção anterior na invocação do *kernel* onde é necessário especificar as dimensões da rede e dos blocos. A hierarquia de grupos de *threads* é composta por três tipos de distintos de conjuntos de *threads* que são definidos por granularidades diferentes. Os três tipos de grupos com granularidade mais fina para granularidade mais grossa respetivamente são: (1) *warp*, (2) bloco e (3) rede.
O *warp* é o grupo de granularidade fina que são sempre formados por conjuntos fixos de 32 *threads*. Os *warps* são utilizados pelo sistema para escalonar e executar trabalho de forma paralela, isto acontece porque todas as 32 *threads* que pertençam a um mesmo *warp* irão executar as mesmas instruções em paralelo utilizando uma arquitetura *Single Instruction Multiple Thread* (SIMT) [55]. Isto significa que um utilizador não tem controlo nem sobre a criação nem sobre o escalonamento dos *warps*.
Um bloco é um grupo de granularidade média e é formado por um ou mais *warps*. Os blocos são essenciais pois permitem que o utilizador efetue uma divisão lógica de trabalho pelas diversas *threads* que o constituem. O conjunto de *threads* dentro um dado bloco é controlado pelo utilizador no arranque de um *kernel* e pode ser definido num domínio de até três dimen-

sões. Por forma a permitir que cada *thread* se consiga contextualizar dentro do domínio de um bloco, cada uma tem acesso a um vetor de conveniência definido por três valores correspondendo às três dimensões do bloco. Este vetor pode ser acedido utilizando a referência *threadIdx*, onde as várias dimensões podem ser acedidas utilizando os componentes *x*, *y* e *z* do vetor. Um bloco pode atualmente ter uma geometria de no máximo (1024, 1024, 64) *threads*.

Cada *thread* tem acesso também a um vetor de conveniência com as dimensões do bloco, este pode ser acedido utilizando a referência *blockDim* e as três componentes referentes às várias dimensões podem ser acedidas utilizando as componentes *x*, *y* e *z* do vetor. Tendo em conta que o trabalho é dividido entre as várias *threads* de um bloco, é necessário que cada uma se consiga contextualizar por forma a conseguir identificar qual o trabalho a realizar. Utilizando ambos os vetores de conveniência é possível calcular um ID único dentro de um bloco (TbID) associado a cada *thread*, para isso são utilizadas as seguintes equações:

$$TbID = threadIdx.x \tag{3.3}$$

$$TbID = threadIdx.x + (threadIdx.y \times blockDim.x) \tag{3.4}$$

$$TbID = threadIdx.x + (threadIdx.y \times blockDim.x) + //(threadIdx.z \times blockDim.x \times blockDim.y) \tag{3.5}$$

Sendo utilizada a Equação 3.3 para blocos definidos em domínios unidimensionais, a Equação 3.4 para blocos definidos em domínios bidimensionais e a Equação 3.5 para blocos definidos em domínios tridimensionais.

A geometria com que são definidos os blocos é importante pois um bloco irá ser executado dentro de um único SM, isto significa que a quantidade de trabalho que será executado em paralelo dentro do bloco depende dos recursos que cada *thread* utiliza. A utilização de uma má geometria aliada a uma má codificação poderá obrigar o SM a efetuar várias mudanças de contextos entre as *threads* que constituem um único bloco o que poderá originar grandes repercussões ao nível do desempenho.

Por último temos a rede, este é o grupo de granularidade mais grossa. Uma rede é formada por um conjunto de blocos e permite uma divisão mais grosseira do trabalho que será mais tarde dividido finamente pelos blocos de *threads* que por sua vez será dividido mais uma vez em *warps*. Da mesma forma como os blocos têm uma divisão em *warps* e que pode ser re-

presentado utilizando geometrias utilizando até três dimensões, uma rede é formada por um conjunto de blocos que podem ser distribuídos num domínio de até três dimensões. Tendo em conta que o uso da rede adiciona mais três dimensões à hierarquia de grupos de *threads*, é necessário a inclusão de novos vetores que permitam que as *threads* se contextualizem globalmente. Estes são disponibilizados pela *API* através de duas referências distintas: (1) *blockIdx* e (2) *gridDim*. A referência *blockIdx* aponta para um vetor que permite a identificação do bloco no domínio tridimensional de uma dada rede, com as várias dimensões a serem acedidas através dos componentes *x*, *y* e *z* do vetor. A referência *gridDim* aponta para um vetor que contém as dimensões da rede em que as *threads* estão a ser executadas. Utilizando ambas as referências é possível uma *thread* contextualizar o bloco onde se encontra e dessa maneira obtenha um ID global único para toda a rede a que damos o nome de Global ID (GID). Para calcular esse ID único dentro da rede, a *thread* faz uso dos vetores de referência tanto do bloco como da rede nas seguintes equações:

$$GID = blockIdx.x \times Bs \times TbID \tag{3.6}$$

$$GID = (blockIdx.x \times Bs) + (gridDim.x \times blockIdx.y \times Bs) + TbID \tag{3.7}$$

$$\begin{aligned} GID = {} & (blockIdx.x \times Bs) + (gridDim.x \times blockIdx.y \times Bs) \\ & + (gridDim.y \times gridDim.x \times blockIdx.z \times Bs) + TbID \end{aligned} \tag{3.8}$$

$$Bs = blockDim.x \times blockDim.y \times blockDim.z \tag{3.9}$$

Onde *Bs* é o tamanho do bloco, ou seja é a multiplicação do total das várias componentes que constituem o seu domínio (*x*, *y* e *z*). Sendo utilizada a Equação 3.6 para redes definidas em domínios unidimensionais, a Equação 3.7 para redes definidos em domínios bidimensionais e a Equação 3.8 para redes definidos em domínios tridimensionais.

Tal como acontece com a geometria dos blocos, a geometria com que é definida a rede tende a ter grandes influências no desempenho computacional dos algoritmos. As duas geometrias, dos blocos e da rede, devem sempre ser definidas por forma a obter o maior proveito dos recursos do sistema no qual o algoritmo irá ser executado evitando ao máximo paragens e trocas de contextos [56]. Deve-se também ter especial atenção que um bloco que tenha sido escalonado para um SM, nunca irá migrar para outro. Isto é, mesmo na existência de SMs sem trabalho útil, um bloco irá sempre permanecer no SM para o qual foi escalonado esperando pela sua vez de execução. Assim caso o trabalho não tenha sido bem distribuído

pela rede e pelos blocos que a constituem, poderá dar-se o caso de blocos terem quantidades de trabalho superiores a outros. Isto poderá ser resultado de duas situações distintas, (1) dependências ou (2) má definição de geometria aliada a má codificação. Assim sempre que possível dever-se-á homogeneizar quantidade de trabalho distribuída pela rede [56].

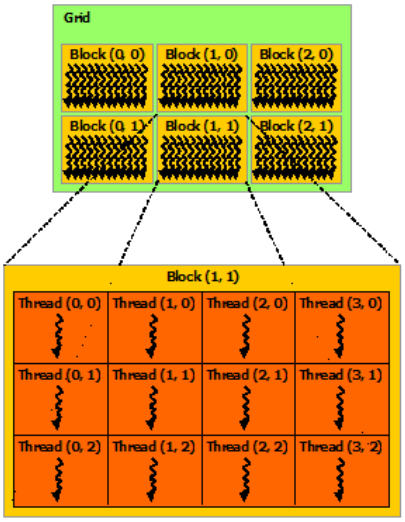

Figura 3.3: Hierarquia de grupos de *threads* [53]

A Figura 3.3 demonstra o exemplo de uma rede definida com uma geometria de $3x2x1$ que contém blocos com uma geometria definida por $4x3x1$, demonstrando a abstração de hierarquia de *threads* tal com foi descrita anteriormente.

Modelo de Execução

O modelo de programação CUDA segue uma arquitetura de execução heterogénea, isto significa que a mesma tira partido da sinergia de mais do que um único tipo de arquitetura. Tendencialmente a arquitetura CUDA é utilizada em sistemas que englobam ambas as arquiteturas sequencial e paralela.

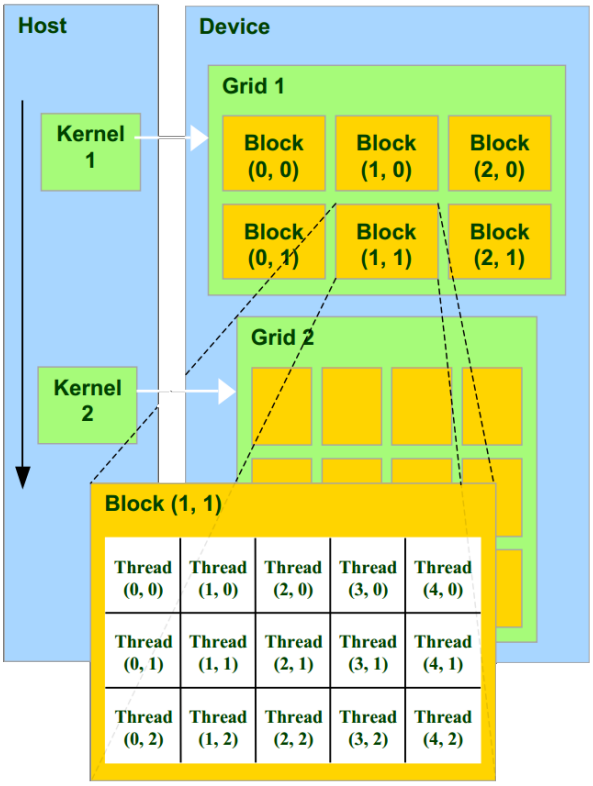

Figura 3.4: Modelo de execução heterogéneo [57]

A Figura 3.4 demonstra o uso comum da arquitetura CUDA. A secção *host* alude ao código corrido no sistema hospedeiro (CPU), enquanto que a secção *device* alude ao código corrido no dispositivo GPU.

Modelo de Memória

Um dos grandes problemas das arquiteturas de computação paralela é que os algoritmos tendem a ser *memory bound*, isto significa que o tempo necessário para executar o algoritmo está maioritariamente dependente do tempo de acesso aos dados. Isto deve-se ao facto deste tipo de arquiteturas efetuar o processamento de uma grande quantidade de dados em simultâneo utilizando os milhares de núcleos de processamento. Em determinadas situações as unidades de memória não conseguem ter um débito de dados suficientemente alto para manter os núcleos ocupados com trabalho gerando assim *botlenecks* de acessos à memória.
Por forma a minimizar o impacto que os acessos à memória causam, a plataforma CUDA disponibiliza um conjunto de memórias de características distintas, delegando ao utilizador a responsabilidade de discernir qual o tipo de memória que melhor se adequa a cada caso

tendo em conta as características das mesmas e do algoritmo em questão.

Memória	E.E.	Tipo	Tamanho	Latência	Declaração	Scope	Lifeycle
Global	*Device*	R/W	GiB	Elevada	Automático	Rede	Aplicação
Local	*Device*	R/W	GiB	Elevada	Automático	*Thread*	*Kernel*
Partilhada	*OnChip*	R/W	KiB	Média	`__shared__`	Bloco	*Kernel*
Cache L2	*Device*	R/W	KiB	Média	Automático	Bloco	*Kernel*
Cache L1	*OnChip*	R/W	KiB	Média	Automático	Bloco	*Kernel*
Constant	*Device*	R	KiB	Dependente	`__constant__`	Rede	Aplicação
Texture	*Device*	R	GiB	Dependente	Automático	Rede	Aplicação
Registos	*OnChip*	R/W	*Bytes*	Reduzida	Automático	*Thread*	*Kernel*

Tabela 3.1: Tipos de Memórias da arquitetura CUDA

A Tabela 3.1 lista os tipos de memória disponíveis na plataforma CUDA tal como as suas características principais. Os tipos de memória disponibilizados podem ser classificados em dois subtipos relativamente ao espaço de endereçamento onde residem: (1) fora do processador do GPU (*Device*) ou (2) integrados no processador do GPU (*OnChip*). As memórias que residem no espaço de endereçamento do GPU tendem a ter uma maior capacidade e um maior tempo de latência. Por sua vez, as memórias que residem integradas no SM tendem a ter uma resposta mais rápida mas capacidades bastante reduzidas. A memória de *device* é onde está o maior espaço de endereçamento existente na GPU chegando às unidades de GiB no entanto é uma memória com uma latência de acesso elevada variando entre os 200 e os 800 ciclos de relógio dependendo do *compute capability* [53]. Na memoria de *device* estão englobados os tipos de memória Global, Local, Textura, e Constante.

- Global: Este é o tipo de memória com maior capacidade podendo chegar aos vários *gigabytes* (GiB), que reside no espaço de endereçamento do *device*, é no entanto a memória de acesso mais lento tendo a maior latência entre um pedido e a respetiva resposta. Os acessos a este tipo de memória utilizam a *cache* L2 para acelerar os pedidos de memória. O código que executa no CPU pode efetuar operações de alocação e copiar valores de e/ou para este tipo de memória utilizando funções especificas. Contudo, não pode efetuar acessos diretos utilizando o endereçamento retornado. Por sua vez, o *device* pode apenas efetuar operações simples de escrita e leitura de endereços de memória já alocados utilizando o endereçamento de memória global.

- Local: Este tipo de memória é implementado no espaço de endereçamento da memória global e é utilizado pela plataforma de forma automática. O controlo sobre estes acessos é feito pelo compilador. Algumas das variáveis que podem ser inseridas na

memória local incluem vetores de tamanho indeterminado durante a etapa de compilação, grandes estruturas ou *arrays* que puderam ocupar um grande número de registos ou qualquer variável definida no *kernel* sempre que exista um elevado consumo de registos. Os acessos a este tipo de memória utilizam a *cache* L2 para acelerar os pedidos de memória. Apenas o *device* pode efetuar operações de escrita e de leitura sobre este espaço de endereçamento. O utilizador não pode efetuar nenhum tipo de operação de forma direta sobre este tipo de memória.

- Texturas: A memória das texturas utiliza o mesmo espaço de endereçamento que a memória global, no entanto mecanismos de endereçamento especiais são utilizados permitindo um mais rápido acesso utilizando. Para isso é utilizada uma *cache* intermédia a que damos o nome de *cache* de texturas. Este tipo de memória utiliza *harwade* específico para efetuar os cálculos de endereçamento necessários e estão otimizados para atingir um melhor desempenho para acessos a dados com localidade espacial 2D. Para utilizar este tipo de memória o *host* tem de efetuar as operações de alocação e cópia para a memória global descritas anteriormente e posteriormente sinalizar os endereços de memória pretendidos como sendo de texturas. Para aceder aos mesmos dentro do *kernel* existem funções específicas para acessos em uma, duas e três dimensões. Este tipo de memória é apenas de leitura não podendo ser modificada em *runtime*.

- Constantes: Ao contrário do que acontece com as memórias anteriormente descritas, embora a memória das constantes utilize o espaço de endereçamento do *device* esta está limitada a 64KiB de memória. Este tipo de memória está otimizado para padrões de acesso em que todas as *threads* de um dado *warp* acedem ao mesmo endereço de memória em simultâneo, permitindo para isso efetuar *broadcasts* dos dados requisitados para todas as *threads*. O *host* pode alocar e copiar de e/ou para este tipo de memória em *runtime*. O *device* apenas pode aceder ao mesmo utilizando a referência recebida pelas funções de alocação do *host*. Este tipo de memória é apenas de leitura não pudendo ser alterada dentro de um *kernel*.

- *Cache* L2: A *cache* L2 é uma memória que reside no espaço de endereçamento do *device* sendo assim comum a todos os SMs. Esta é utilizada para acelerar pedidos de memória a ambos os tipos de memória Global e Local (os outros dois tipos têm *caches* para uso próprio). Apenas o sistema tem controlo sobre operações de escrita e leitura sobre este tipo de memória.

As memórias de chip tendem a ter tamanhos bastante reduzidos não ultrapassando a unidade

dos KiB por cada SM, no entanto são memórias com tempos de latência bastante reduzidos que permitem um desempenho superior quando bem utilizadas. Os vários tipos de memória de chip são a memória partilhada, *cache* L1 e Registos.

- Partilhada: A memória partilhada é um dos tipos de memória que se encontra no chip e é uma memória de latência média. Este tipo de memória é dividido em segmentos a que damos o nome de bancos, podendo ter entre 16 bancos para *compute capability* 1.x ou 32 bancos para *compute capability* 2.x, 3.x e 5.x onde em ambos os casos cada palavra de 32 bits consecutiva é mapeada para um banco de memória consecutivo. Este tipo de memória está otimizado para efetuar acessos a endereços de memória que não partilhem bancos entre si, ou seja para acessos uniformes onde várias *s* efetuem um acesso ao mesmo endereço de memória, sendo efetuado nesse caso um *broadcast* dos dados para todas as *threads* que os requisitaram. A memória partilhada é ainda apropriada para acessos a endereços de memória diferentes mas que não se encontrem no mesmo banco. Quando são efetuados dois acessos de memória a endereços de memória diferentes mas pertencentes a um mesmo banco temos um conflito de bancos, o desempenho dos mesmos irá ser degradado por um fator igual ao número de conflitos existentes [53]. A memória partilhada pode ser declarada tanto em tempo de compilação, utilizando o modificador `__shared__` a partir de uma função de *device*, ou em *runtime* utilizando os parâmetros de iniciação do *kernel*. O *host* não pode efetuar qualquer tipo de operação sobre este tipo de memória. Até a *compute capability* 5.x (exclusive) esta memória tem o seu espaço de endereçamento compartilhado com a *cache* L1, podendo ser utilizado um de dois cenários: 48 KiB para a memória partilhada e 16KiB para a *cache* de L1 ou vice-versa. Para *compute capability* 5.0 cada dispositivo tem cerca de 64KiB de memória partilhada por SM. Finalmente, para dispositivos com *compute capability* de 5.2, cada SM tem 96KiB de memória partilhada tendo em ambos estes casos a memória partilhada um espaço de endereçamento privado.

- *Cache* L1: A *cache* L1 é uma memória que reside no espaço de endereçamento do chip podendo apenas ser acedida pelas *threads* pertencentes aos blocos alocados nesse mesmo SM. Esta é utilizada para acelerar pedidos de memória a ambos os tipos de memória Global e Local. Apenas o sistema tem controlo sobre operações de escrita e leitura sobre este tipo de memória. Como descrito na memória partilhada, até a compute 5.0 este tipo de memória tem o seu espaço de endereçamento compartilhado com a memória partilhada podendo ter ou 48KiB ou 16KiB de capacidade. Para dispositivos com *compute capability* 5.x a *cache* L1 tem o seu espaço de endereçamento

partilhado com a *cache* das texturas tendo uma capacidade conjunta de 24KiB.

- Registos: Os registos são a unidade de memória mais rápida do sistema sendo também a de menor capacidade. Os registos são divididos por SMs e existem sempre entre 8000 a 16000 registos disponíveis [58] que serão partilhados pelas várias *threads* associadas a um dado SM. Sendo o tipo de memória mais rápido é aconselhável a que o utilizador maximize o seu uso, minimizando a quantidade de *registry spilling* [53]. Os registos são acessíveis apenas dentro de uma dada *thread* e têm um tempo de vida igual ao da *thread* que os está a utilizar.

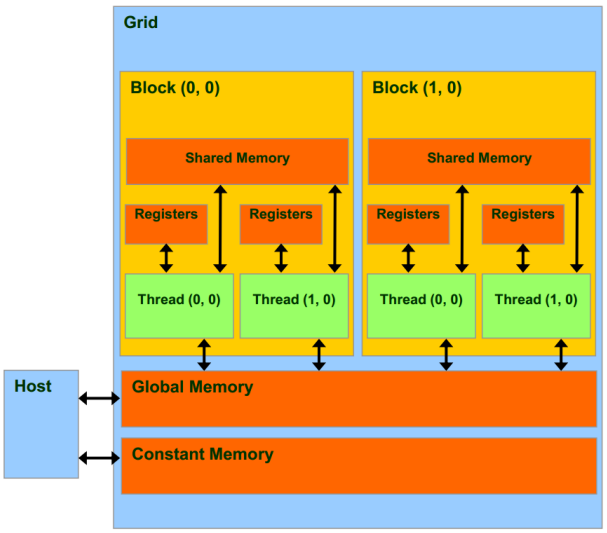

Figura 3.5: Modelo de memórias da arquitetura CUDA [59]

A Figura 3.5 ilustra o modelo de memória descrito anteriormente. Como é evidente, é aconselhável que o programador utilize sempre a memória de chip pois a mesma tem tempos de latência mais reduzidos que se irão traduzir em desempenhos superiores. No entanto nem sempre é uma tarefa fácil otimizar o uso das memórias devido a vários tipos de constrangimentos, sobretudo devido a dependências de recursos.

Sincronização

Por forma a permitir que os vários fluxos de execução possam ser sincronizados, a plataforma CUDA disponibiliza dois métodos de sincronização para serem utilizados em dois contextos distintos.

A primeira barreira de sincronização serve para sincronizar todos os fluxos de execução que pertençam a um mesmo bloco. Para o efeito, a *Application Programming Interface* (API) CUDA disponibiliza o método `__syncthreads` que bloqueia o fluxo das *threads* até que todas as *threads* pertencentes ao mesmo bloco se encontrem no mesmo ponto de execução.
A segunda barreira de sincronização disponibilizada pela *API* permite a sincronização de um contexto de granularidade mais grosso, sendo utilizado pelo *host* por forma a sincronizar o mesmo com a finalização do trabalho já anteriormente emitido. Para tal a *API* disponibiliza dois métodos distintos: (1) `cudaDeviceSynchronize` e (2) `cudaStreamSynchronize`. O método `cudaDeviceSynchronize` é empregue quando se pretende que o *host* pare o seu fluxo de execução até que todos os processos anteriormente emitidos independentemente da *stream* para onde tenham sido emitidos tenham sido finalizados, isto inclui tanto execução de *kernels* como cópias de dados. O método `cudaStreamSynchronize` deverá ser utilizado quando se pretende que o *host* pare o seu fluxo de execução até que uma *stream* especificada tenha terminado todo o trabalho que tenha sido emitido para a mesma.

Capítulo 4

CUDA-MMP

4.1 Introdução

Todo o trabalho descrito neste documento teve como base um protótipo do algoritmo MMP--Intra já anteriormente implementado a que damos o nome de CUDA-MMP, sendo o trabalho desenvolvido neste documento uma continuação do projeto desenvolvido no âmbito do mesmo trabalho de investigação. Esta secção efetua uma breve introdução às alterações implementadas no protótipo.

Este capítulo começa por descrever o ambiente computacional onde todo o trabalho foi realizado. De seguida é efetuado uma pequena análise de *profiling* sobre a versão sequencial. O capítulo termina com uma breve descrição do trabalho efetuado sobre o protótipo inicial com uma demonstração dos resultados obtidos.

4.2 Ambiente Computacional

Este trabalho foi desenvolvido utilizando dois computadores de alto desempenho e duas GPUs NVIDIA. Em cada experiência/execução, cada computador usou apenas uma GPU em simultâneo, embora ambas as GPUs tenham sido empregues em ambos os computadores. A especificação dos computadores utilizados foi a seguinte:

Processador:	2x Intel Xeon CPU E5-2630 @ 2.3 GHz (24 núcleos)
Memória RAM:	64GiB
Sistema Operativo:	Ubuntu 14.04 LTS (Trusty Tahr), *kernel* 3.13.0-36-generic

Tabela 4.1: Especificações do Servidor 1

Processador:	2x Intel Xeon CPU E5-2620 @ 2.0 GHz (24 núcleos)
Memória RAM:	32GiB
Sistema Operativo:	Ubuntu 14.04 LTS (Trusty Tahr), *kernel* 3.13.0-39-generic

Tabela 4.2: Especificações do Servidor 2

Ao longo de todo este documento ambos os servidores serão identificados por *Servidor 1* e *Servidor 2*.
As placas gráficas utilizadas foram as seguintes:

Arquitetura:	Kepler
CUDA Capability:	3.5
Multiprocessadores:	15 multiprocessadores com 192 núcleos cada
Clock rate:	0.889 GHz
Boost clock rate:	0.980 GHz
Memória:	6 GiB

Tabela 4.3: Especificações da GPU GeForce GTX TITAN Black [1]

Arquitetura:	Kepler
CUDA Capability:	3.0
Multiprocessadores:	8 multiprocessadores com 192 núcleos cada
Clock rate:	1.006 GHz
Boost clock rate:	1.058 GHz
Memória:	2 GiB

Tabela 4.4: Especificações da GPU NVIDIA GeForce GTX 680 [2]

Embora o trabalho tenha sido desenvolvido ao longo de diversas versões CUDA, os resultados apresentados neste documento resultam da execução do algoritmo sobre a versão CUDA 6.5 sendo esta executada sobre os *drivers* de versão 340.58 disponibilizados pela NVIDIA.

4.2.1 Recolha de Resultados

Os resultados apresentados ao longo deste documento são obtidos efetuando uma média aritmética de 5 execuções por forma a minimizar o impacto de desvios que possam existir pontualmente. Todos os testes foram executados sobre um conjunto de parâmetros referência. O valor de referência do lambda foi de 10, sendo este um valor onde a perda de qualidade não é detetável a olho nu, como se pode depreender da comparação entre as imagens da Figura 4.1 (original) e da Figura 4.2 (descompactada após compressão MMP). O número máximo de

blocos do dicionário foi fixado nos 1024 elementos por nível. Nos testes, todos os modos de predição estão ativos e a segmentação da predição é efetuada até ao nível 8.

Figura 4.1: Imagem Lena (original)

Figura 4.2: Imagem Lena (codificada com lambda 10)

A imagem de referência na recolha dos tempos foi a imagem Lena apresentada na Figura 4.1.

4.3 *Profiling*

Para verificar se existiria alguma vantagem na migração do algoritmo MMP sequencial para uma implementação paralela, o processo de migração começou por uma fase de estudo. Este começou pelo *profiling* do algoritmo MMP-Intra sequencial com o objetivo de detetar zonas

de grandes consumos computacionais. Para isso foi utilizada uma ferramenta de *profiling* que recorrem a métodos de *sampling*, o GProf [60].

Função	Taxa de poder computacional utilizado (%)
`quad_err`	76.3310
`optimize_block`	11.6854
`least_sq`	10.8378
`optimize_block_and_pred_mode`	3.4878
`getClosestPixels`	2.0842
`rate`	1.8398
`rate_dicseg`	1.0820

Tabela 4.5: Distribuição do poder computacional das funções com maior consumo

A Tabela 4.5 mostra a distribuição de poder computacional utilizado de entre as funções com um maior consumo. Através desta análise foram identificadas três funções críticas que se distinguiam das restantes com um consumo de poder computacional bastante superior: (1) a função `quad_err`, (2) a função `optimize_block` e (3) a função `least_sq`.

- **`quad_err`**: A função `quad_err` é responsável pelo cálculo da distorção entre dois blocos. A sua complexidade origina apenas do cálculo dos erros quadráticos entre os píxeis de ambos os blocos.

- **`optimize_block`**: A função `optimize_block` é responsável pela construção da árvore de segmentação de resíduo. Para cada bloco de resíduo gerado, o algoritmo efetua uma chamada à função `optimize_block`. Esta por sua vez efetua um número de chamadas à função `quad_err` igual ao número de blocos existentes no dicionário que sejam do mesmo nível do bloco de resíduo recebido como parâmetro por forma a calcular a distorção entre ambos. Entre cada chamada à função `quad_err`, a função `optimize_block` calcula o custo de representar o bloco de resíduo utilizando o bloco do dicionário enviado para a função `quad_err`, guardando sempre o bloco com menor custo.

  Com a pesquisa terminada para todos os padrões do dicionário, a função `optimize_block` verifica se é possível segmentar o bloco de resíduo em uma das direções (vertical ou horizontal). Caso o bloco possa ser segmentado, a função é invocada recursivamente para cada um dos blocos originados da sua segmentação. Uma vez finalizado o processo de segmentação, a função compara o custo de representar o bloco de resíduo

utilizando um único bloco do dicionário ou utilizando a concatenação de blocos, retornando uma árvore de segmentação do resíduo podada com a representação do resíduo cujo custo foi inferior.

A sua complexidade computacional está relacionada com os constantes cálculos dos custos para cada nó das árvores de segmentação do resíduo.

- **`least_sq`**: a função `least_sq` é responsável pelo cálculo dos coeficientes utilizados no modo de predição LSP para gerar o melhor valor em cada píxel. Para o efeito, o modo de predição LSP utiliza um método de fatorização *LU* em conjunto com uma técnica de *forward factorization*, sendo essas operações a razão para a sua elevada complexidade computacional.

Utilizando a lei de Amdahl definida na Equação 3.1 podemos estimar o ganho obtido paralelizando estas três funções utilizando a seguinte equação:

$$G(N) = \frac{1}{0.011522 + \frac{0.988478}{N}} \tag{4.1}$$

onde N é o número de núcleos de execução paralelos. Assim podemos concluir que paralelizando as três funções referidas anteriormente o limite teórico para o *speedup* atingível será de aproximadamente 86.79, isto para quando o valor de N é tão elevado que se pode desprezar o tempo de execução do fluxo paralelo.

4.4 Implementação

Após a análise dos dados encontrados durante o *profiling* da versão sequencial, foi definido como objetivo inicial a migração das funções `quad_err` e `optimize_block` para a GPU. Esta decisão teve como fundamento três fatores importantes: (1) em primeiro lugar o impacto computacional que ambas estas funções têm sobre o desempenho do algoritmo, pois são as duas funções com maior consumo de poder computacional do algoritmo, (2) em segundo lugar o facto de ambas as funções estarem relacionadas entre si, ou seja, a função `optimize_block` está dependente dos resultados da função `quad_err` para que consiga calcular os custos associados a cada bloco permitindo assim uma mais fácil sincronização de dados entre a CPU e a GPU e (3) o facto de ambas as funções terem a sua complexidade computacional relacionada com cálculos matemáticos, pois as operações aritméticas são o tipo de operação de execução mais rápida em GPUs.

Com a migração destas funções para a GPU, o objetivo é delegar para a GPU o cálculo tanto dos erros quadráticos como dos custos de cada bloco, utilizando posteriormente uma função de redução que permita encontrar os melhores blocos para todos os níveis da árvore. Isto permite que a função `quad_err` seja eliminada por completo e que a função `optimize_block` seja utilizada apenas para efetuar a construção da árvore de segmentação do resíduo podada, utilizando para isso os valores previamente calculados pelos três passos descritos anteriormente.

4.4.1 Dicionário

Sendo tanto o cálculo dos erros quadráticos entre o bloco de resíduo e os padrões do dicionário tal como o cálculo do custo de cada bloco efetuados na GPU, é necessário alterar o fluxo de execução do algoritmo para permitir que os dados estejam disponíveis na GPU quando são necessários. Assim sendo durante a etapa de inicialização do algoritmo, é necessário reservar memória para o dicionário na GPU, por forma a prevenir custos de desempenho associados a constantes realocações de memória. Para o efeito, o algoritmo aloca a memória necessária para guardar todo o dicionário. Isto significa que este valor está dependente do número máximo de elementos que cada nível do dicionário pode conter uma vez que este valor é definido pelo utilizador como parâmetro. Após a alocação com sucesso, o dicionário gerado na CPU é então copiado para a GPU.

Embora a GPU seja a responsável pelo cálculo das distorções, uma cópia do dicionário tem de existir tanto na GPU como na CPU. Isto deve-se ao processo de atualização do dicionário que existe sempre que um macro bloco é codificado. O MMP utiliza um dicionário adaptativo, guardando transformações de blocos previamente utilizados na codificação. O processo de atualização do dicionário utiliza técnicas que impedem a inserção de novos blocos com um grau de semelhança muito elevado a bloco já existentes, para isso o algoritmo calcula a distorção entre o novo bloco e todos os blocos do dicionário pertencentes ao mesmo nível. No caso em que uma das distorções calculadas seja inferior a um limite definido por parâmetro, o algoritmo considera que o grau de semelhança entre os dois blocos é elevado e o mesmo será descartado.

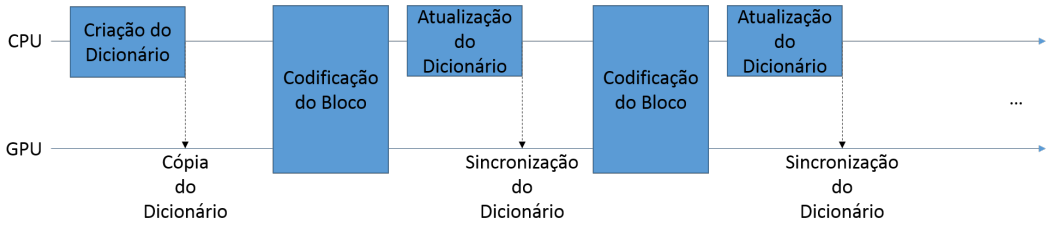

Figura 4.3: Sincronização do dicionário

Todo o processo de atualização do dicionário é efetuado na CPU, obrigando a que exista sempre uma cópia atualizada do dicionário tanto na CPU, para efetuar o processo de atualização, como na GPU, para efetuar o cálculo das distorções e correspondentes custos. Uma vez que o processo de atualização tenha sido concluído na CPU, é necessário que os resultados se reflitam na cópia do dicionário existente na GPU como ilustrado na Figura 4.3.

Foram identificadas duas possíveis implementações distintas para resolver o problema de sincronismo: (1) utilização dos métodos de cópia disponibilizados pela plataforma CUDA para copiar os diversos blocos ou (2) criação de um *kernel* de cópia.

A vantagem de utilizar os métodos de cópia disponibilizados pela plataforma CUDA é a diminuição da complexidade de desenvolvimento. Para utilizar estas basta apenas que indiquemos a posição dos dados originais, a posição destino para onde se pretenda que a cópia seja feita e a quantidade de dados a copiar. No entanto existe uma desvantagem inerente a este tipo de implementação. Quando o algoritmo MMP efetua a atualização do dicionário, esta acontece para todos os níveis do dicionário, podendo ser adicionados novos blocos em qualquer um dos níveis. Assim sendo iremos ter possivelmente um grande conjunto de blocos novos mas distribuídos pelos vários níveis. Como já foi descrito anteriormente o desempenho de efetuar cópias na plataforma CUDA são grandemente condicionadas por dois fatores: (1) latência e (2) débito. Este tipo de placas têm uma elevada latência e também um grande débito ou seja o seu desempenho prende-se em reduzir o numero de cópias e aumentando a quantidade de dados em cada cópia. Sabendo que o cenário apresentado na atualização do dicionário contrasta com a premissa apresentada anteriormente, a solução assenta na implementação de um *kernel* que recebe os blocos e os níveis por parâmetro e insere cada bloco na posição correta. Deste modo são efetuadas menos operações de cópia para efeito de sincronização do dicionário.

4.4.2 Codificação do Bloco

A primeira versão do CUDA-MMP teve como objetivo a migração dos blocos de código utilizados para gerar a árvore de segmentação do resíduo. Para isso foram criados dois *kernels*: (1) *kernel* do cálculo das distorções e custos (K1) e (2) *kernel* de redução (K2).

O *kernel* K1 calcula todas as distorções e correspondentes custos associados a todos os blocos existentes no dicionário para todas as partições possíveis do bloco de resíduo, guardando-os numa matriz unidimensional. Isto significa que caso o bloco de resíduo a ser codificado seja de nível 0 (1 x 1), o *kernel* K1 irá efetuar a comparação entre a única partição existente no bloco de resíduo e todas os blocos do dicionário, caso o bloco de resíduo a ser codificado fosse de nível 24 (16x16) o *kernel* iria efetuar a comparação de todas as 961 possíveis partições do *kernel* de resíduo e os blocos do dicionário dos respetivos níveis.

O *kernel* K2 foi concebido com o objetivo de percorrer a matriz de resultados criado pelo *kernel* K1 e encontrar, para cada nível, qual o bloco do dicionário com um custo menor, ou seja efetuar uma operação de redução sobre os resultados de K1, guardando os melhores blocos numa nova matriz. Uma vez terminados ambos os *kernels*, é necessário copiar a matriz de resultados de K2 para a CPU por forma a possibilitar à função `optimize_block` acesso aos resultados, para que a mesma consiga gerar a árvore de segmentação da predição já podada.

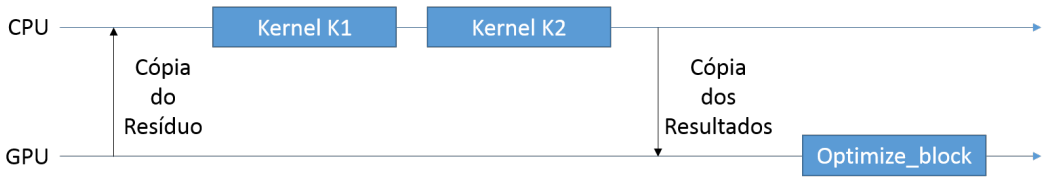

Figura 4.4: Fluxo de execução heterogéneo

A Figura 4.4 demonstra o novo fluxo de execução heterogéneo.

4.4.3 Resultados

Embora a nova versão tenha sido implementada com sucesso, esta revelou uma grande perda de desempenho.

Sistema	Tempo Sequencial (s)	Tempo (s)	Speedup
Servidor 1 GTX Titan	2091.878	11248.366	0.186
Servidor 1 GTX 680	2091.878	16189.448	0.129
Servidor 2 GTX Titan	2091.878	11306.530	0.185
Servidor 2 GTX 680	2091.878	16279.100	0.129

Tabela 4.6: Resultados após a implementação do primeiro protótipo CUDA-MMP

Como se pode verificar na Tabela 4.6, a primeira versão paralela do algoritmo CUDA-MMP revelou-se bastante mais lenta que a versão sequencial atingindo tempos de execução de cinco vezes superiores. Uma breve análise a estes tempos revelou que cerca de 65,5% do tempo de execução deve-se ao *kernel* de redução K2 e cerca de 28,1% do tempo deve-se ao *kernel* de quantificação K1.

4.5 Conclusão

Este capítulo apresentou o ambiente computacional utilizado ao longo do desenvolvimento de todo o projecto. Analisando os resultados de *profiling* da versão sequencial, verificamos quais os pontos de grande consumo de poder computacional que se encontravam no algoritmo, chegando à conclusão que seria benéfico a migração dos mesmo para uma versão sequencial. Por fim foi apresentada uma primeira abordagem à paralelização do algoritmo, a qual obteve um desempenho muito inferior à versão sequencial.

Capítulo 5

Otimizações

5.1 Otimizações de Memória

Tal como foi explicado no Capítulo 3, a plataforma CUDA disponibiliza várias memórias com diferentes características para serem utilizadas pelas aplicações. Dependendo do tipo de dados e do padrão de acesso a aplicar, o tipo de memória a utilizar irá ser distinto. Esta secção aborda as otimizações de memória levadas a cabo ao longo do estudo da otimização do algoritmo CUDA-MMP.

5.1.1 Memória Local e Constantes

Análise e *Profiling*

O compilador NVCC permite especificar diversas opções por forma a alterar o seu comportamento durante as varias fases de compilação. Uma destas fases é a fase de criação e otimização do ficheiro PTX (*Parallel* thread *Execution*). Este é o ficheiro criado pelo compilador para os métodos que são executados na GPU. O ficheiro contém as instruções pertencentes ao *instruction set architecture* (ISA) das placas CUDA. Utilizando a opção de compilação `-ptxas-options=-v` (verbose) obtém-se para cada *kernel* compilado métricas referentes à quantidade de registos utilizados, quantidade de memória local utilizada e quantidade de memória para constantes utilizada.

```
nvcc -O3 -c -Iinc -I../mmpcommon/inc -odir obj src/coder.cu src/main_mmpenc.cu  ../mmpcommon/src/adaptive_model.cu ../mmpcommon/src/cuda_includes.cu ../mmpcommon/src/dic.cu
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z19cuda_optimize_blockPiS_S_PliiiiiifiiS_PmS1_S1_S1_S1_S1_PfS2_S2_S2_f' for 'sm_30'
ptxas info    : Function properties for _Z19cuda_optimize_blockPiS_S_PliiiiiifiiS_PmS1_S1_S1_S1_S1_PfS2_S2_S2_f
    2800 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 33 registers, 484 bytes cmem[0], 20 bytes cmem[2]
ptxas info    : Compiling entry function '_Z23cuda_reduct_jCostBufferPiPfS0_S0_S0_PliS0_' for 'sm_30'
ptxas info    : Function properties for _Z23cuda_reduct_jCostBufferPiPfS0_S0_S0_PliS0_
    5104 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 27 registers, 384 bytes cmem[0]
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z15Cuda_update_dicPimS_i' for 'sm_30'
ptxas info    : Function properties for _Z15Cuda_update_dicPimS_i
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 18 registers, 348 bytes cmem[0]
ptxas info    : 0 bytes gmem
ptxas info    : 0 bytes gmem
ptxas info    : 0 bytes gmem
ptxas info    : 0 bytes gmem
```

Figura 5.1: Análise do código PTX dos valores locais

Ao analisarmos os resultados do compilador utilizando a opção indicada (Figura 5.1) verificamos que tanto o *kernel* de quantificação como o *kernel* de redução estavam a utilizar uma elevada quantidade de memória local, respetivamente 2800KiB e 5104KiB.

Sabendo que os acessos à memória local e global têm grande impacto de desempenho, pois ambos residem na mesma memória física no dispositivo, a plataforma CUDA implementa um sistema de *cache* para acelerar os pedidos de memória efetuados a estes tipos de memória. Dependendo da *compute capability* do dispositivo, diferentes técnicas de *caching* poderão ser utilizadas para aumentar a velocidade de acesso à memória local e à memória global.

Para dispositivos com *compute capability* 1.x nenhum sistema de *cache* é implementado, pelo que todos os acessos à memória local e global são sempre grandemente penalizados.

Para dispositivos com *compute capability* superior ou igual a 2.0, os pedidos efetuados à memória global são otimizados utilizando dois níveis de *caches* de memória auxiliares: L1 e L2.

Para dispositivos com *compute capability* 2.x os acessos à memória local utilizam sempre ambas as *cache*s L1 e L2. Os pedidos à memória global podem ser configurados através de determinadas opções de compilação por forma a permitir o uso de ambas as *cache*s L1 e L2 ou apenas da *cache* L2. A mesma memória de 64KiB física é utilizada pela *cache* L1 e pela memória partilhada. Estas podem ser configuradas para utilizar uma de duas possíveis distribuições: i) 48KiB para a *cache* L1 e 16KiB de memória para a memória partilhada ou ii) 16KiB para a *cache* L1 e 48KiB para a memória partilhada. Esta distribuição pode ser definida tanto por contexto, sendo assim geral a todos os *kernels* que executem dentro do mesmo contexto, ou pode ser configurada por *kernel*, sendo assim possível a definição de diferentes distribuições de memória L1 e partilhada para diferentes *kernels*. Isto permite uma melhor distribuição dos tipos de memória tendo em conta as características dos *kernels*.

Para dispositivos com *compute capability* 3.x os acessos à memória local utilizam sempre ambas as *cache*s L1 e L2. Os pedidos à memória global podem ser configurados dependendo

da *minor version*. Para dispositivos com *compute capability* 3.0, os pedidos à memória global utilizam apenas a *cache* L2. Para alguns dispositivos com *compute capability* 3.5 ou 3.7 os pedidos à memória global podem ser configurados utilizando *flags* durante a compilação dos *kernels* permitindo o uso de ambas as *cache*s L1 e L2 ou apenas da *cache* L2, no entanto nem todos os dispositivos permitem este tipo de configuração [61], como por exemplo a placa Tesla K20S. Mais uma vez a *cache* L1 existe na mesma memória física do que a memória partilha, podendo novamente esta ser configurada. Para dispositivos com *compute capability* 3.x existem três possíveis distribuições de memória L1/memória partilhada: (1) 48KiB para *cache* L1 e 16KiB para memória partilhada, (2) 16KiB para a *cache* L1 e 48KiB para a memória partilhada ou (3) 32KiB para a *cache* L1 e 32KiB para a memória partilhada. Para dispositivos com *compute capability* 3.7 cada distribuição contém mais 64KiB de memória partilhada.

Para dispositivos com *compute capability* 5.0 os acessos à memória global utilizam apenas *cache* L2. Neste caso a *cache* L1 encontra-se unificada com a *cache* para a memória das texturas e é apenas utilizada como *buffer* para acesso a dados apenas de leitura. Para dispositivos com *compute capability* 5.2, operações de leitura à memória podem utilizar ambos os níveis de *cache* L1 e L2.

A *cache* do tipo L1 encontra-se sempre no *chip* (circuito integrado principal da GPU) sendo assim uma memória com tempos de latência bastantes reduzidos. Este tipo de *cache* é assim partilhada apenas por um multiprocessador, ou seja, as vantagens deste tipo de *cache* apenas são visíveis após um dos cores do multiprocessador que se encontra no mesmo *chip* ter acedido anteriormente ao endereço de memória pretendido. Este tipo de memória utiliza transações de 128 *bytes*.

A *cache* do tipo L2 encontra-se no dispositivo (*offchip*). Isto significa que tem uma latência superior à *cache* do tipo L1, no entanto a *cache* L2 é partilhada por todos os multiprocessadores, isto significa que basta que um multiprocessador aceda a um endereço de memória para que todos os acessos posteriores consigam ter benefícios na utilização da *cache* de tipo L2, mesmo que estes venham de multiprocessadores distintos. Este tipo de memória efetua transações de 32 *bytes*.

Embora o uso de técnicas de *caching* permitam um grande aumento de desempenho quando se utilizam acessos às memórias local e global, este aumento está diretamente relacionado com as características dos *kernels* executados. Estes mecanismos de *cache* apenas são úteis quando é possível manter em *cache* os dados necessários, ou seja quando é possível obter uma elevada taxa de *cache hits*. Quando isto não acontece a utilização de ambas as *cache*s pode ser prejudicial para o desempenho do algoritmo. Isto acontece por duas razões distintas:

(1) aumento do número de instruções e (2) congestionamento dos canais de memória.
Quando ocorre um *cache miss* na *cache* L1, um de dois cenários distintos pode ter ocorrido: (1) os dados necessários nunca foram lidos e dessa maneira não se encontram na *cache* L1 ou (2) não existia espaço suficiente para guardar novos pedidos e os dados tiveram de ser despejados para dar espaço para novos dados.
Para ambos os cenários é necessário efetuar um pedido de 128 *bytes* à *cache* L2, pois a *cache* L1 funciona apena com transações de 128 *bytes*, mesmo que o sistema apenas necessite de 1 valor inteiro (quatro *bytes*). Como já foi descrito anteriormente, a *cache* L2 apenas utiliza transações de 32 *bytes* assim, para satisfazer o pedido de 128 *bytes* da *cache* L1 são necessárias quatro transações com a *cache* L2. Caso ocorra um *cache miss* para a *cache* L2 outras quatro transações terão de ser efetuadas para a memória global.
Assim sempre que os dados necessários não se encontram na *cache* L1, cinco instruções distintas têm de ser emitidas: uma para efetuar o pedido de memória de 128 *bytes* na *cache* L1 e mais quatro para efetuar quatro pedidos de memória de 32 *bytes* na *cache* L2. Caso exista um *cache miss* na *cache* L2 outras quatro instruções irão ser emitidas para efetuar quatro pedidos de memória de 32 *bytes* à memória global perfazendo assim um total de nove instruções. Para o segundo cenário, onde os dados tiveram de ser despejados, instruções adicionais são necessárias pois os dados que se encontram na memória L1 caso tenham sido alterados terão de ser escritos para a memória L2 sendo que posteriormente poderão ter de ser escritos para a memória global, implicando assim mais quatro ou oito operações de escrita, aumentando assim o possível total de instruções emitidas para as dezassete. Caso a *cache* L1 não seja utilizada no pior dos casos serão necessárias oito instruções de leitura, quatro para a *cache* L2 e quatro para a memória global, sendo necessários mais quatro instruções de escrita para o cenário onde os dados são despejados da memória L2 perfazendo um total de doze instruções.
Uma elevada taxa de *cache miss* poderá também levar a um aumento de congestionamento de memória provocando uma *sobre procura de dados*, obrigando desta maneira a que as *threads* fiquem sem trabalho útil para processar devido a dependências de memória. Isto acontece devido à *cache* L1 apenas utilizar transferências de 128 *bytes*, independentemente do tamanho dos dados necessários. Assim caso todas as *threads* num *warp* acedam a um mesmo endereço de memória para efetuar o carregamento de um valor inteiro (quatro *bytes*) pertencente à memória local, é efetuado um pedido 128 *bytes*. Caso vários pedidos deste tipo sejam efetuados para endereços que estejam espalhados pela memória local, várias transações de 128 *bytes* irão ser efetuadas apenas para utilizar quatro *bytes* de cada vez, provocando uma *sobre procura de dados*. Desativando a *cache* L1, apenas transações de 32 *bytes* são efetuadas,

diminuindo assim a quantidade de dados copiados desnecessariamente.
Podemos então concluir que mesmo em dispositivos que implementem técnicas de *caching*, deverá ser efetuada uma análise à distribuição dos dados e ao padrão de acesso efetuado por forma a verificar qual a melhor solução a implementar, pois as técnicas de *caching* poderão revelar-se prejudiciais ao desempenho do algoritmo quando aplicadas em situações impróprias.
Por forma a verificar o impacto que as técnicas de *caching* têm sobre o sistema foi utilizada a ferramenta de *profiling* nvprof para recolher contadores e métricas [62]. Em primeiro lugar procedeu-se a análise da taxa de *cache hits* em acessos a *cache* L1, verificando que apenas aproximadamente 4% de todos os pedidos efetuados a *cache* L1 eram servidos imediatamente. Como foi explicado, a existência de uma baixa taxa de *cache hits*, pode levar a um grande congestionamento. O *profiler nvprof* contém uma métrica para medir a taxa entre a quantidade de memória transferida entre a *cache* L1 e a *cache* L2 devido a pedidos de memória local e devido a pedidos de memória global. Esta métrica revelou uma taxa de aproximadamente 70%, ou seja a a maioria de todos os dados que são transferidos entre a *cache* L1 e a *cache* L2 devem-se a *cache misses* devido a pedidos efetuados a memória local.
Para solucionar este problema várias soluções podem ser aplicadas. Uma possível solução é o aumento da *cache* L1 por forma a permitir que uma maior quantidade de dados esteja presente minimizando o despejo da *cache*. Esta solução apenas e viável quando a razão dos *cache misses* se deve a despejos de memória. Sabendo que a análise de *profiling* foi efetuada utilizando já o máximo de memória para a *cache* L1 esta solução não se mostrou viável.
Outra possível solução é o aumento do número de registos utilizado pelo *kernel*, diminuindo assim o *register spilling* e dessa maneira a quantidade de dados que se encontra na memória local, pois sempre que e necessário libertar um registo, o seu valor e guardado na memória local. Esta solução também se mostrou inviável pois utilizando a opção de compilação `-ptxas-options=-v`, verificamos que nenhum dos *kernels* continha qualquer tipo de *register spilling*.
Uma outra possível solução é a passagem dos valores que se encontram na memória local para um outro tipo de memória, sendo a memória escolhida dependente dos padrões de acesso. Assim foi efetuada uma análise ao padrão de acesso utilizado para verificar a viabilidade deste cenário.
A análise ao padrão de acesso revelou que na maioria dos casos todas as *threads* acedem ao mesmo endereço de memória em simultâneo. Existindo no máximo pedidos a dois endereços de memória em simultâneo por *warp*, isto significa que utilizando a memória local com *cache*s L1 e L2 iremos obter um elevado custo devido a sobre procura de valores. Verifica-

mos também que os valores que estavam a ser guardados na memória local eram apenas de leitura, sendo os mesmos *arrays* estáticos declarados diretamente no código fonte. Ambas estas características revelaram que estes valores eram bons candidatos a serem guardados na memória das constantes.

A memória das constantes é um tipo de memória que reside na memória física do dispositivo tendo uma *cache* de uso exclusivo no *chip*[63]. Isto significa que o primeiro acesso a um endereço deste tipo de memória tem uma latência considerável, pois tal como acontece com a memória local é necessário efetuar uma cópia da memória do dispositivo para a memória *cache*. Contudo, todos os acessos posteriores ao mesmo endereço de memória têm uma latência de acesso bastante reduzida. Adicionalmente, a memória das constantes implementa um sistema de acesso distinto dos outros tipos de memória, o mesmo está otimizado para efetuar *broadcasts* para todas as *threads* de um *warp*, sendo o seu desempenho degradado em um fator igual ao número de endereços distintos a serem acedidos num mesmo instante. Uma outra característica desta memória é que é apenas de leitura durante a execução de *kernels*. A mesma apenas pode ser escrita utilizando métodos próprios da *API*, tendo os mesmos de serem executados fora do contexto de um *kernel*. Este tipo de memória é também empregue pelo compilador para guardar alguns valores gerados durante a fase de compilação. Do lado do *kernel*, os valores estão disponíveis apenas para leitura, por exemplo, é assim que se efetua a passagem de parâmetros para o *kernel*. Este tipo de memória foi introduzido na arquitetura CUDA na *compute capability* 2.0 e tem um tamanho de 64 KiB para todas as *compute capabilities* até à corrente (5.2) tendo a *cache* um tamanho de 8 KiB para as *compute capabilies* 2.x e 3.x, e um tamanho de 10 KiB para as *compute capabilities* 5.x.

Utilizando o *kernel* de quantificação cerca de 2800 *bytes* de memória global e o *kernel* de redução 5104 *bytes* de memória global, sabemos que no máximo serão necessários cerca de 7904 *bytes* de memória para manter todos os dados que anteriormente eram guardados na memória local. Podemos também verificar que já existe previamente alguma memória das constantes a ser utilizada, proveniente de valores estáticos criados pelo compilador e de parâmetros de entrada dos *kernels*. O *kernel* de quantificação utiliza 504 *bytes* de memória, o *kernel* de redução utiliza 384 *bytes* de memória e o *kernel* de atualização do dicionário utiliza 348 *bytes* de memória. Assim podemos verificar que nem sempre será possível guardar todos os dados na *cache* das constantes em simultâneo pois para dispositivos com *compute capability* inferior a 5.x a *cache* das constantes é de 8 KiB enquanto a soma de todos os valores a guardar nas constantes é de aproximadamente 9 KiB. Ao analisarmos o conteúdo dos dados verificámos no entanto que a maioria dos mesmos era comum aos *kernels* de quantificação e de redução, ou seja a quantidade de memória constante que é necessária para

ambos os *kernels* será substancialmente inferior à quantidade de memória máxima calculada anteriormente. Utilizando a informação proveniente da compilação do código fonte, verificámos que a memória total necessária seria de 7668 *bytes*. Desta maneira sabemos que após o primeiro acesso, a percentagem de *cache hits* à memória *cache* das constantes será de aproximadamente 100%, uma vez que todos os dados podem estar na *cache* em simultâneo.

Resultados Experimentais

Como foi descrito anteriormente todos os valores que são guardados na memória local são apenas de leitura, isto significa que os mesmos apenas têm de ser escritos uma vez ao longo de toda a codificação da imagem. Assim, a cópia dos valores para a memória das constantes é efetuada uma única vez durante a fase de inicialização do algoritmo. Isto permite que o primeiro *kernel* efetue a cópia dos valores para a memória *cache*, acelerando substancialmente todos os *kernels* posteriores durante toda a codificação da imagem.
Para efetuar a cópia dos valores para a memória das constantes foi utilizado a função `cudaMemcpyToSymbol`.

```
ptxas info    : 0 bytes gmem, 6424 bytes cmem[3]
ptxas info    : Compiling entry function '_Z19cuda_optimize_blockPiS_PliiiiiifiiS_PmS1_S1_S1_S1_S1_PfS2_S2_S2_f' for 'sm_30'
ptxas info    : Function properties for _Z19cuda_optimize_blockPiS_PliiiiiifiiS_PmS1_S1_S1_S1_S1_PfS2_S2_S2_f
    104 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 30 registers, 476 bytes cmem[0], 20 bytes cmem[2]
ptxas info    : Compiling entry function '_Z23cuda_reduct_jCostBufferPiPfS0_S0_S0_PliS0_' for 'sm_30'
ptxas info    : Function properties for _Z23cuda_reduct_jCostBufferPiPfS0_S0_S0_PliS0_
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 20 registers, 384 bytes cmem[0]
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z15Cuda_update_dicPimS_i' for 'sm_30'
ptxas info    : Function properties for _Z15Cuda_update_dicPimS_i
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 18 registers, 348 bytes cmem[0]
ptxas info    : 0 bytes gmem
ptxas info    : 0 bytes gmem
ptxas info    : 0 bytes gmem
ptxas info    : 0 bytes gmem
```

Figura 5.2: Saída do NVCC

Uma vez efetuada a passagem dos valores para a memória das constantes foi utilizado o compilador para verificar que todos os valores tinham sido passados para a memória das constantes. Como se pode verificar na Figura 5.2, a compilação da nova implementação apresenta 0 *bytes* de memória local para todos os três *kernels*. É também possível verificar que para além da memória utilizada para os parâmetros de entrada dos *kernels* e das variáveis criadas pelo compilador estão agora a ser utilizados cerca de 6424 *bytes* adicionais de memória das constantes, perfazendo um total de 7652 *bytes*. Assim podemos afirmar que após o primeiro acesso a cada endereço de memória das constantes, os dados copiados agora para

esta memória deverá ter uma taxa de *cache hits* de 100% uma vez que nunca será necessário remover as mesmas da *cache*.

Sistema	Tempo de Referência (s)	Tempo (s)	speedup
Servidor 1 GTX Titan	11248.365	5935.002	1.895
Servidor 1 GTX 680	16189.447	7367.226	2.197
Servidor 2 GTX Titan	11306.533	6054.388	1.867
Servidor 2 GTX 680	16279.099	7452.201	2.184

Tabela 5.1: Resultados após a remoção da utilização da memória local

Otimizando os acessos às memórias por ambos os *kernels*, a Tabela 5.1 demonstra o *speedup* obtido. Este confirma a premissa anterior, demonstrando que o aumento de hits à memória *cache* das constantes tem um impacto substancial no desempenho do algoritmo, alcançando um *speedup* compreendido entre 1.867 e 2.197.

5.1.2 Alinhamento de Memória

Análise e *Profiling*

O dicionário de padrões utilizado pelo algoritmo MMP pode e tendencialmente atinge tamanhos na ordem dos MiB. Devido ao elevado tamanho, o dicionário tem necessariamente de ser alocado na memória global. Sendo esta a memória com menor desempenho devido aos grandes tempos de latência associados, é importante que os acessos à mesma sejam feitos de forma agrupada e alinhada por forma a permitir o menor número de transações para satisfazer cada pedido de memória.
A memória global é uma memória com uma grande largura de banda, permite pois débitos de dados bastante elevados. No entanto é também uma memória cujo tempo de latência é bastante elevado, compreenda-se como tempo de latência o tempo decorrido desde que um pedido de memória e efetuado até que exista uma resposta por parte do sistema. Uma outra característica importante da tecnologia CUDA é a sua arquitetura de execução SIMT, onde um *warp* de *threads* executa a mesma instrução em simultâneo excetuando em casos explícitos de difusão de fluxo através de instruções condicionais. Isto significa que na maioria dos casos, 32 *threads* irão efetuar acessos de memória em simultâneo. Tendo em conta ambas as características descritas, a arquitetura CUDA implementa uma técnica de acesso a dados que permite a fusão das instruções de memória de todas as *threads* [63]. Isto significa que quando num *warp* várias *threads* efetuam um pedido de memória em simultâneo em vez de se efetuar uma transação de memória para escrever ou obter um valor para cada *thread*, a

tecnologia CUDA agrupa todas as instruções e tenta efetuar todos os pedidos utilizando o menor número de transações possíveis. O objetivo é diminuir a latência de acesso, uma vez que o número de acessos diminui. Para que seja possível tirar partido da técnica de acessos agrupados várias condições devem ser satisfeitas:

- O *warp* de *threads* que se pretenda que efetue um acesso de memória agrupado deve aceder a um conjunto de dados consecutivo em memória
- Os acessos de memória deverão ser efetuados de forma alinhada, isto significa que para *compute capability* 1.x todos os dados a serem manipulados deverão estar contidos num mesmo segmento de 64 *bytes* ou 128 *bytes* e para *compute capability* superior a 2.x os dados a serem manipulados deverão estar contidos num mesmo segmento de 128 *bytes*.

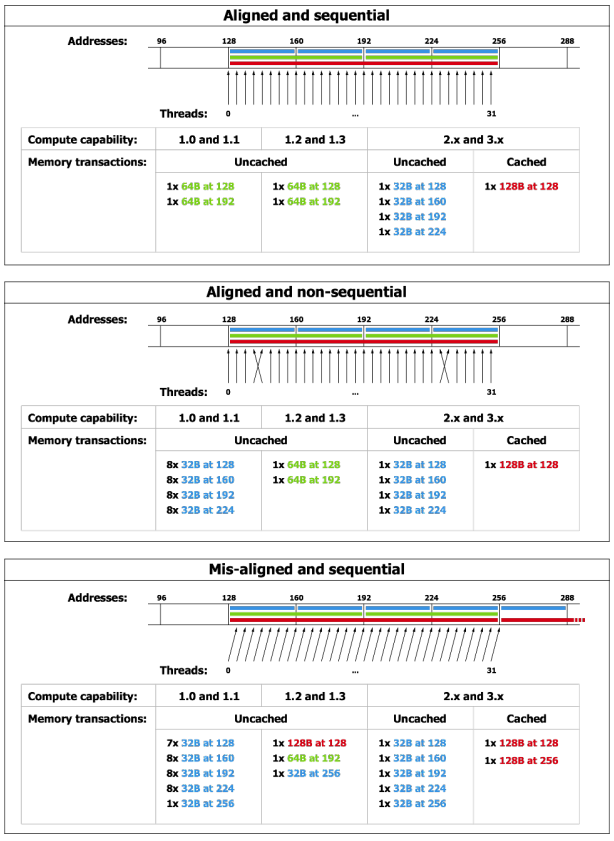

Figura 5.3: Análise do número de transações por pedido de memória [63]

A Figura 5.3 ilustra a diferença entre o número de transações de memória necessárias para um pedido de memória de 128 *bytes* comparando quando as condições anteriormente descritas são ou não satisfeitas. Para *compute capability* 1.x, sempre que as condições são

satisfeitas, um pedido de 128 *bytes* pode ser satisfeito em apenas duas transações de 64 *bytes* e um pedido de 256 *bytes* pode ser satisfeito em dois pedidos de 128 *bytes*. Isto acontece pois para sistemas com *compute capability* 1.x, o agrupamento de pedidos de memória é efetuado para cada meio *warp*. Quando as condições não são satisfeitas, qualquer tipo de pedido de memória ira efetuar tantas transações quantas *threads* efetuarem o pedido. Para sistemas que implementem *compute capability* superior a 2.x, o agrupamento de pedidos e efetuado por *warp*, ou seja para cada 32 *threads*. Aqui quando os pedidos de memória seguem as condições definidas, a plataforma consegue satisfazer um pedido de 128 *bytes* utilizando apenas uma transação e pedidos de 256 *bytes* utilizando duas transações de 128 *bytes* cada uma. Quando as condições não são satisfeitas, a plataforma agrupa pedidos cujo endereço de memória pertença ao mesmo segmento.

Por forma a garantir o melhor desempenho possível, a tecnologia de acesso agrupado deverá ser sempre utilizado por forma a executar o máximo de instruções de acesso a dados na memória utilizando o menor número de transações possível, garantindo assim um elevado débito através da diminuição do número de operações.

O algoritmo MMP-Intra guarda o dicionário em memória dividido por níveis. Dentro de cada nível, os padrões são guardados utilizando matrizes bidimensionais com tamanhos dependentes do nível em questão.

Por exemplo, para o nível 24 (16 x 16 píxeis) cada padrão é guardado utilizando uma matriz bidimensional de 16 x 16. Sendo o tamanho máximo do dicionário um parâmetro de entrada, durante a fase de inicialização, o MMP-Intra efetua o cálculo da quantidade de memória necessária para guardar o dicionário completo. Tal como acontece na versão sequencial, o CUDA-MMP reserva durante a fase de inicialização tanta memória quanto a necessária para guardar um dicionário completo. Na plataforma CUDA, não é possível definir uma matriz bidimensional no espaço de endereçamento da GPU. Deste modo, a matriz bidimensional do MMP empregue para o dicionário de padrões é mapeada para uma matriz unidimensional, inserindo-se as várias linhas de um bloco de forma consecutiva em memória.

Nível	Dimensões	Nº Pixeis
0	1 x 1	1
1	1 x 2	2
2	2 x 1	2
3	2 x 2	4
4	1 x 4	4
5	4 x 1	4
6	2 x 4	8
7	4 x 2	8
8	4 x 4	16
9	1 x 8	8
10	8 x 1	8
11	2 x 8	16
12	8 x 2	16
13	4 x 8	32
14	8 x 4	32
15	8 x 8	64
16	1 x 16	16
17	16 x 1	16
18	2 x 16	32
19	16 x 2	32
20	4 x 16	64
21	16 x 4	64
22	8 x 16	128
23	16 x 8	128
24	16 x 16	256

Tabela 5.2: Níveis existentes utilizando um esquema de segmentação flexível

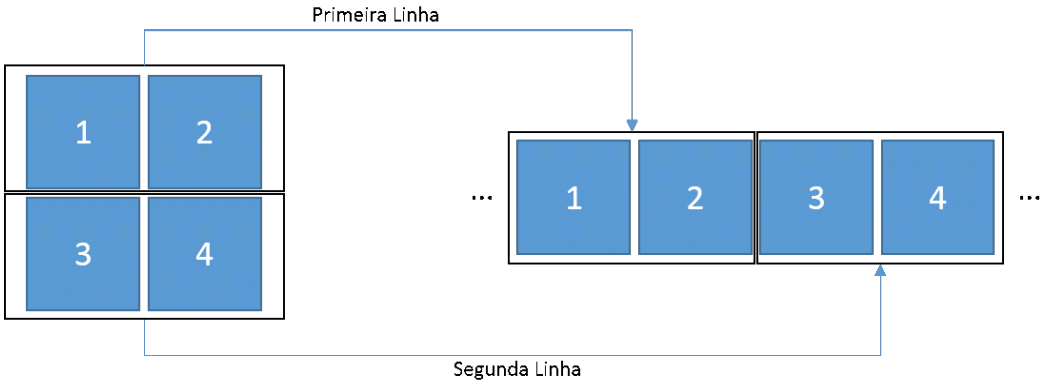

Figura 5.4: Mapeamento do dicionário na memória global

A Figura 5.4 ilustra um mapeamento de um bloco de nível 3 (2x2) de uma matriz bidimensi-

onal (CPU) para uma matriz unidimensional (GPU). Para efetuar o mapeamento do endereço de memória é utilizada a seguinte expressão:

$$M(x, y, z) = (z * NC * NL) * y * NC + x, \tag{5.1}$$

sendo z o índice do padrão no nível em questão, x a coordenada horizontal do píxel a aceder, y a coordenada vertical do píxel a aceder, NL o número de linhas do nível do bloco a codificar e NC o número de colunas do nível do bloco a codificar. Utilizando este mapeamento, o CUDA-MMP insere todos os níveis e correspondentes matrizes numa matriz unidimensional. Os diversos níveis estão organizados por ordem numérica do nível 0 ao 24. Para cada nível é reservado o espaço de memória necessário tendo em conta o número máximo de elementos que o mesmo poderá ter utilizando o valor indicado pelo parâmetro de entrada do MMP.

Durante a execução do *kernel* k1, cada *thread* consecutiva irá calcular a distorção entre uma partição de resíduo e um padrão consecutivo do dicionário que pertença ao mesmo nível. Isto apenas não se verifica no caso em que já não existam padrões disponíveis. Nesse caso a *thread* para a qual já não existe um padrão disponível irá comparar a partição de resíduo seguinte com o primeiro padrão do dicionário do mesmo nível e mais uma vez as *threads* consecutivas irão calcular a distorção entre a nova partição de resíduo selecionada e os padrões consecutivos existentes. Ao analisar o padrão de acesso considerando o caso em que todas as *threads* pertencentes a um mesmo *warp* efetuam o cálculo da distorção utilizando a mesma partição de resíduo, verifica-se que cada *warp* de *threads* não irá aceder a um conjunto consecutivo de dados. Isto deve-se à disposição dos padrões do dicionário na memória global.

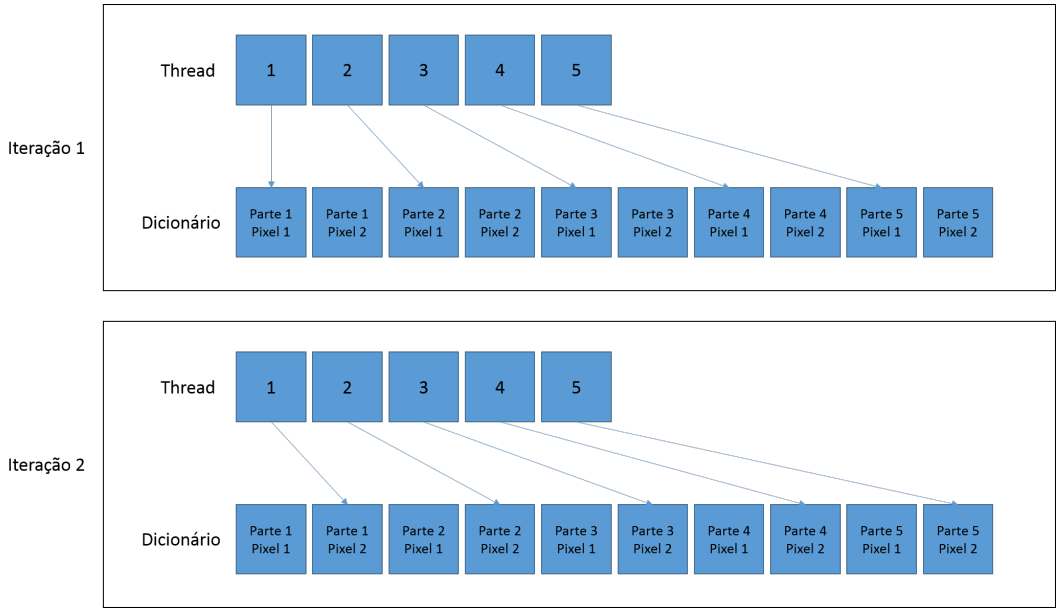

Figura 5.5: Análise do padrão de acesso desalinhado

Como se pode verificar, o facto do dicionário ser guardado de forma sequencial na GPU inviabiliza a utilização de acessos agrupados. Durante a primeira iteração do cálculo da distorção, todas as *threads* acedem ao primeiro valor do padrão do dicionário. No nível 0 este padrão de acesso não se revela problemático uma vez que cada padrão do dicionário é composto apenas por um píxel, o que significa que todas as *threads* efetuam um acesso sequencial. Para todos os outros níveis, *threads* sequenciais não efetuam acessos de memória sequenciais pois entre cada acesso existe um deslocamento dependente do número de píxeis do nível. Na Figura 5.5 podemos verificar que efetuando o cálculo das distorções para um bloco de nível 1 (1x2), o deslocamento entre os vários acessos de memória de *threads* sequenciais será de 2, ou seja, tão grande quanto o tamanho dos blocos para os quais o cálculo é efetuado.

Kernel	**Média do Número de Transações / Pedido**	**Média de Transações / execução Kernel**
K1	1.656312	1628
K2	1.237073	1472

Tabela 5.3: Número médio de transações de memória por pedido, com padrão de acesso original

A Tabela 5.3 apresenta a média do número de transações de memória efetuadas por pedido e a média do número de transações total por execução de *kernel*. Estes valores foram obtidos

utilizando a ferramenta de *profiling* NVVP [64].

O novo padrão de acesso tem em conta o tamanho de cada *warp*, e insere de forma sequencial o valor com o mesmo índice para partições sequenciais. Isto significa que o primeiro valor das primeiras 32 partições são inseridos de forma sequencial, de seguida o segundo valor das mesmas partições é inserido de forma sequencial e assim por diante até que todos os valores tenham sido inseridos. Uma vez terminado este processo para as primeiras 32 partições, o processo é repetido para as próximas 32 partições até que todas as partições do mesmo nível tenham sido mapeadas. Sendo posteriormente mapeados os restantes níveis utilizando o mesmo processo.

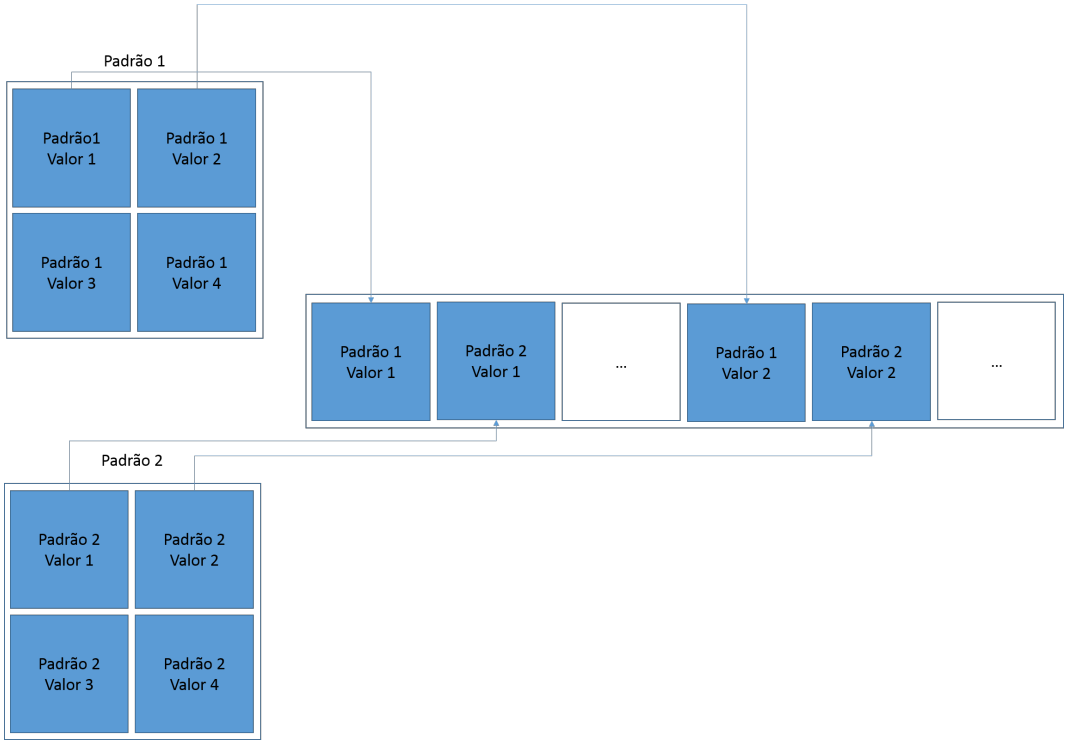

Figura 5.6: Mapeamento alinhado do dicionário na memória global

Como se pode verificar na Figura 5.6, com o novo mapeamento cada valor com o mesmo índice será mapeado de forma sequencial. Isto até um máximo de 32 valores consecutivos. Desta maneira, quando as *threads* de um *warp* estão a processar o mesmo índice para blocos consecutivos, os acessos à memória decorrem de forma alinhada e agrupada (Figura 5.7). Para efetuar o mapeamento do endereço de memória com o novo processo de mapeamento é utilizada a seguinte expressão:

$$M(x, y, z) = floor(2/32) * 32 * NL * NC + 2\%32 + (y * NL + x) * 32, \quad (5.2)$$

sendo z o índice do padrão no nível em questão, x a coordenada horizontal do píxel a aceder, y a coordenada vertical do píxel a aceder, NL o número de linhas do nível do bloco a codificar e NC o número de colunas do nível do bloco a codificar.

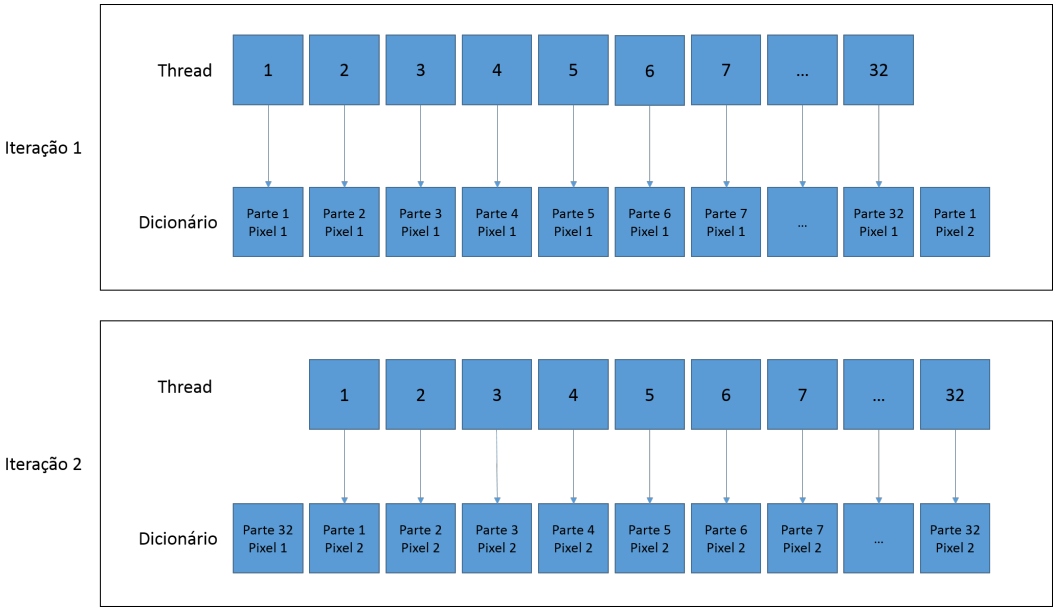

Figura 5.7: Análise do padrão de acesso alinhado

Para que o novo padrão de acesso seja completamente alinhado é necessário que todos os níveis sejam obrigatoriamente múltiplos de 32. Quando um dos níveis não é múltiplo de 32 acontece que as ultimas *threads* de um determinado *warp* irão começar a efetuar o trabalho do nível seguinte. No entanto, os níveis são alocados durante a fase de inicialização do algoritmo e cada nível é alocado de forma sequencial, ou seja, todo um nível é alocado e apenas posteriormente é alocado um outro nível. Aqui podem ser identificados dois cenários problemáticos: (1) o número máximo de padrões por nível não é múltiplo de 32, isto significa que quando o nível estiver completamente cheio, existirão *warps* que efetuarão trabalho de dois níveis distintos e (2) enquanto o dicionário está a ser adaptado, poderão existir cenários em que alguns níveis terão um número de padrões não múltiplo de 32 e mais uma vez existirão *warps* que efetuarão trabalho de dois níveis distintos.

Um dos problemas com o processamento de dois níveis em simultâneo por um único *warp* deve-se ao facto de que dois níveis distintos podem ter um número de elementos diferentes entre si. Isto significa que em determinadas situações poderia existir num mesmo *warp*

threads a processarem um determinado padrão e outras já no padrão seguinte, originando acessos não alinhados.

Outro problema ocorre quando um nível ainda não se encontra completamente preenchido. De facto, existe um preenchimento entre o último elemento desse nível e o primeiro elemento do próximo nível com o objetivo de ser preenchido com os novos padrões que possam surgir para o primeiro nível. Isto significa que se um *warp* efectuar o processamento de ambos os níveis em simultâneo, os valores a serem utilizados não se encontram de forma sequencial na memória e mais uma vez o processo efetua acessos não alinhados. Para piorar a situação, uma vez que o padrão de acesso depende do tamanho do *warp* para efetuar um mapeamento de todos os blocos, efetuando agrupamentos de 32 em 32 padrões no dicionário, existindo um *warp* que irá efetuar o processamento de dois níveis em simultâneo este irá quantificar os padrões finais de um nível e os padrões iniciais do nível seguinte. Assim os *warps* que irão processar o próximo nível não irão começar nos padrões iniciais, e assim já seria necessário que o mapeamento utilizando agrupamentos de 32 padrões fosse efetuado a partir do valor cujo *warp* anterior processou. Caso isso suceda, todos os acessos posteriores irão estar desalinhados pois cada *thread* não irá começar a processar blocos com índices múltiplos de 32.

Ambos os cenários descritos anteriormente foram solucionados utilizando um mecanismo de preenchimento. Isto significa que sempre que um nível tem um número de padrões que não seja múltiplo de 32, são inseridos padrões de preenchimento obrigando assim a que o número de padrões seja sempre múltiplo de 32. Os valores de preenchimento podem ser identificados pelas *thread*, evitando assim o processamento destes valores.

Resultados Experimentais

Para permitir que os acessos ao dicionário sejam feitos de forma alinhada foi analisada e implementada um novo mapeamento do dicionário para a memória global. Sabendo que cada *warp* tem um tamanho fixo de 32 *threads* e que cada *thread* efetua o acesso a um valor inteiro (4 *bytes*), a melhor solução ocorre quando todas as 32 *threads* efetuam um acesso a 32 elementos do dicionário de forma sequencial, pois tal resulta em apenas um acesso agrupado de 128 *bytes*.

Cada *thread* pertencente a um *warp* compara uma mesma partição com um padrão diferente do dicionário. Isto acontece para todos os padrões pertencentes ao mesmo nível da partição, ou seja apenas quando não existem mais padrões para comparar poderá acontecer que *threads* pertencentes a um mesmo *warp*, efetuem comparações utilizando partições distin-

tas. Sabendo que o dicionário é iniciado com 511 padrões de nível 0 e 69 partições para cada um dos restantes níveis, podemos concluir que nunca irá acontecer *threads* pertencentes ao mesmo *warp* efetuarem comparações utilizando três partições distintas, pois um *warp* é constituído sempre por 32 *threads* e para cada partição um mínimo de 69 comparações têm de ser feitas, pois este é o número mínimo de padrões existente por nível em qualquer dado momento da codificação.

Kernel	Média do Número de Transações/Pedido	Média de Transações/Kernel
K1	1.0	1329
K2	1.0	1203

Tabela 5.4: Número médio de transações de memória por pedido, com novo padrão de acesso

A Tabela 5.4 apresenta a média do número de transações de memória efetuadas por pedido e a média do número de transações total por execução de *kernel*, após a implementação do novo padrão de acesso ao dicionário. Como se pode verificar, o novo padrão de acesso ao dicionário permite um acesso 100% alinhado, sendo cada pedido de memória resolvido apenas com uma única transação de 128 bits, ou seja um inteiro para cada *thread*.

Sistema	Tempo de Referência (s)	Tempo (s)	speedup
Servidor 1 GTX Titan	857.233	845.6666	1.013
Servidor 1 GTX 680	957.6004	937.4036	1.021
Servidor 2 GTX Titan	949.5384	909.0632	1.044
Servidor 2 GTX 680	1017.3994	1003.463	1.013

Tabela 5.5: Resultados após a implementação de um padrão de acesso alinhado

Como se pode verificar na Tabela 5.8, tal como esperado, a redução do número de transações teve um impacto nos tempos de execução do algoritmo obtendo um *speedup* entre 1.013 e 1.044.

5.1.3 Optimização de Registos

Análise e *Profiling*

Um dos problemas que limita o desempenho de algoritmos que utilizem a tecnologia CUDA é a sobre utilização de recursos. A arquitetura CUDA divide as unidades de processamento em grupos físicos aos quais damos o nome de multiprocessadores onde são executados os *kernels*. Para esse efeito, cada *kernel* é dividido numa rede lógica de blocos constituídos por várias *threads*. Estes blocos são distribuídos pelos vários multiprocessadores, sendo as *threads* que os constituem executadas nas unidades de processamento que se encontram no

multiprocessador para o qual o bloco a que as mesmas pertencem foi escalonado. Sempre que um bloco de *threads* é escalonado para um multiprocessador, as *threads* do mesmo passam a ser designadas como *threads* residentes e reservam recursos do multiprocessador, tais como memória e tempo de execução, que serão necessários à sua execução, obtendo assim exclusividade sobre os mesmos até que o blocode *threads* tenha sido completamente executado e removido do multiprocessador. Sendo os recursos reservados quando um bloco de *threads* é escalonado para um multiprocessador, este apenas pode ser escalonado para um multiprocessador se o mesmo ainda tiver recursos disponíveis suficientes.

Embora um bloco de *threads* seja na maioria dos casos constituído por dezenas de *warps* e um único multiprocessador possa ter centenas de núcleos de processamento, cada *warp* é escalonado e executado de forma cooperativa e não de forma concorrente. Assim apenas 32 *threads* são executadas de forma concorrente. A arquitetura efetua uma pesquisa entre os vários *warps* e insere os *warps* prontos a executar numa fila de espera. Cada *warp* irá ser removido da fila e executado de forma individual. Este sistema de execução cooperativa é utilizando pela arquitetura para esconder elevados tempos de latência que possam surgir durante a execução dos *warps* devido a instruções de sincronização ou de memória, pois as operações de memória têm tempos de latência bastante elevados. Quando uma destas situações ocorre, a arquitetura efetua uma comutação de *warps*, desativando temporariamente o *warp* que se encontra à espera da resolução de uma instrução e ativando a execução de um outro *warp* que esteja pronto a ser executado tentando assim maximizar o tempo de processamento útil.

O número de *warps* residentes tem um grande impacto no desempenho desta técnica uma vez que a mesma depende da existência de *warps* que possam ser comutados, isto significa que um multiprocessador com um reduzido número de *threads* residentes irá consequentemente ter um menor número de *warps*, podendo em determinados momentos não ser possível comutar um *warp* que esteja parado devido à inexistência de *warps* prontos a serem executados. Este problema pode ser resolvido de duas formas distintas: (1) quando num cenário onde o problema pode ser modelado utilizando uma rede lógica de *threads* dinâmica, é possível aumentar o número de *threads* residentes aumentando o número de *threads* existentes em cada bloco, ou (2a) quando num cenário onde o problema não pode ser modelado utilizando uma rede lógica de *threads* dinâmica, é possível aumentar o número de *threads* residentes aumentando o número de *threads* existentes em cada bloco, ou (2b) quando num cenário onde o problema não pode ser modelado utilizando uma rede lógica de *threads* dinâmica, é possível aumentar o número de *threads* residentes diminuindo o número de recursos utilizado por cada *thread* até que seja possível inserir um maior número de blocos por cada

multiprocessador.

O equilíbrio de trabalho entre *warps* é outro fator com bastante importância no desempenho da técnica de comutação. De facto, a existência de uma grande disparidade de trabalho entre *warps* pertencentes a um mesmo bloco leva a que os *warps* com uma menor quantidade de trabalho terminem muito antes dos *warps* com uma grande quantidade de trabalho, levando a uma redução do número de *warps* em execução. Uma vez que seja necessário comutar os *warps* que ainda se encontram a executar, poderá não estar disponível nenhum *warp* pronto a ser executado, pois grande parte dos *warps* já terminou o seu trabalho. Assim é importante que uma boa distribuição de trabalho seja feita permitindo que a quantidade de *warps* existentes prontos a ser executados seja elevada e constante ao longo de toda a sua execução.

Outro fator que influencia o impacto das comutações de *warps* é a quantidade e magnitude das dependências existentes. A magnitude das dependências é bastante importante e está relacionada com a ocupância atingida. A existência de dependências com um grande tempo de latência tem como consequência um maior tempo de espera por parte dos *warps* até que as mesmas sejam resolvidas. Isto significa que cada *warp* irá demorar mais tempo até que se encontre pronto a ser executado e seja inserido na fila de espera para ser escalonado. A existência de dependências com elevados tempos de latência alidadas as uma baixa ocupância poderá levar mais uma vez à inexistência de *warps* prontos a serem executados, o que se irá traduzir em intervalos de tempo onde os multiprocessadores irão estar parados diminuindo consequentemente o seu desempenho.

Uma elevada quantidade de dependências poderá também levar à constante comutação de *warp*s. Embora esta técnica permita diminuir a quantidade de tempo que os multiprocessadores estão parados, a comutação de *warps* tem um custo associado, pois o sistema tem de salvaguardar o estado do *warp* que está em execução, procurar por um *warp* que esteja pronto a ser executado e efetuar a comutação. Mesmo tendo em conta que o custo de comutação é bastante reduzido, a constante comutação de *warps* devido a um elevado número de dependências irá traduzir-se numa diminuição de desempenho.

Kernel	**Número de registos**
K1	26
K2	18

Tabela 5.6: Número de registos utilizado por cada *thread*

Utilizando a ferramenta de compilação NVCC foi possível determinar o número de registos utilizado por cada *kernel*, conforme apresentado na Tabela 5.6.

Resultados Experimentais

Efetuando a reutilização de alguns registos foi implementada uma versão onde o foram reduzidos o número de registos do *kernel* K1 como se pode ver na Tabela 5.7.

Kernel	Número de registos
K1	24
K2	18

Tabela 5.7: Número de registos utilizado por cada *thread*

Sistema	Tempo de Referência (s)	Tempo (s)	speedup
Servidor 1 GTX Titan	474.2414	474.1636	1.000
Servidor 1 GTX 680	478.2124	477.1018	1.000
Servidor 2 GTX Titan	515.501	516.1314	0.999
Servidor 2 GTX 680	520.459	516.3268	1.008

Tabela 5.8: Resultados após a implementação da otimização de registos

Embora o número de registos tenha efetivamente diminuído, os tempos de execução mantiveram-se constantes como se pode verificar na Tabela 5.8. Utilizando a ferramenta NVVP verificou-se que embora o limite de blocos por multiprocessador tenha aumentado de 6 para 8 tendo em conta a dependência de registos, o número de blocos ativos por multiprocessador manteve-se em 4 devido à quantidade de memória partilhada em uso como demonstrado na Figura 5.8.

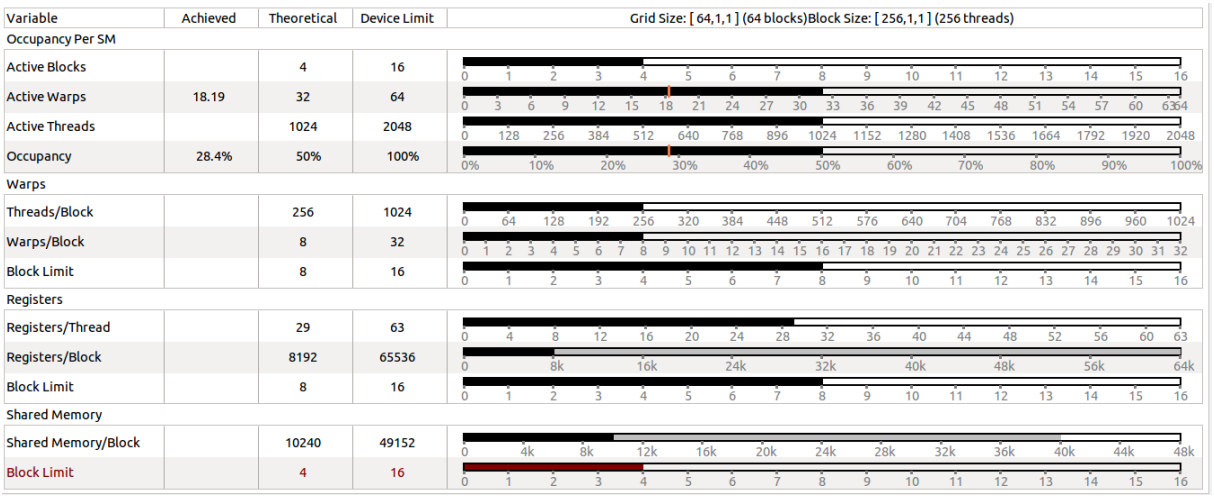

Figura 5.8: Análise do número máximo de blocos por multiprocessador

5.2 Optimizações de Fluxo

A implementação de algoritmos utilizando a arquitetura CUDA consegue por vezes atingir grandes *speedup*s devido a paralelização de tarefas [65]. No entanto dependendo das características do algoritmo, nem sempre é possível paralelizar todo o trabalho existente o que poderá levar a grandes perdas de desempenho. Esta paralelização de trabalho poderá dar-se em diversos níveis tais como a paralelização do trabalho nos diversos núcleos de processamento ou a paralelização de instruções resultante do sistema de linha de montagem dos núcleos. Esta secção aborda otimizações relativas ao fluxo de execução do algoritmo.

5.2.1 *Streams*

Análise e *Profiling*

Todas as operações da arquitetura CUDA são executadas dentro de uma *stream* [66][67]. Uma *stream* define um fluxo de execução independente, ou seja, é uma sequência de operações que são executadas pela ordem que são emitidas. É possível criar e executar várias *streams*, podendo as mesmas ser processadas em paralelo na presença de recursos suficientes. A vantagem do uso de *streams* na plataforma CUDA é que as mesmas permitem a introdução de um novo nível lógico de paralelismo e injeção de dependências de fluxo.
Anteriormente todo o paralelismo dava-se quando várias *threads* executavam a mesma lógica em paralelo, distribuídas pelos núcleos de processamento, que por sua vez estão distribuídos pelos multiprocessadores. A introdução de *streams* permite agrupar logicamente as *threads*, atribuindo a cada grupo diferentes fluxos de execução.
Uma característica das *streams* é que todas as operações emitidas para uma mesma *stream* são executadas de forma sequencial pela ordem que foram emitidas, permitindo a introdução de dependências de fluxo, obrigando cada fluxo a seguir uma sequência de operações. Outra característica importante das *streams* é a possibilidade de escalonar diversas *streams* em simultâneo.
Utilizando ambas as características descritas anteriormente é possível criar e escalonar diversos fluxos de execução com cadeias de dependências isoladas, podendo a sua execução ser feita de forma intercalada ou paralela dependendo da *compute capabilty* e dos recursos do sistema.
A plataforma CUDA disponibiliza através das suas *API*'s, funções que permitem a criação de *streams* que serão utilizadas para escalonar trabalho. Quando nenhuma *stream* é criada, ou quando o trabalho não é associado a nenhuma *streams*, o sistema contém uma *stream*

com características especiais: *stream* de sistema. Quando a *stream* de sistema está a efetuar trabalho, nenhuma outra *stream* é ativada seja para efetuar cópias ou para execução de *kernels*.
A plataforma CUDA permite vários tipos de escalonamento de fluxos de trabalho utilizando *streams*: (1) intercalação de trabalho, tanto cópias como execução de *kernels*, (2) paralelização de *kernels*, (3) paralelização de cópias e (4) paralelização de *kernels* e cópias. Dependendo do tipo de paralelização que se pretende atingir diferentes regras deverão ser seguidas:

- Para efetuar execução de *kernels* de forma intercalada entre *streams*, apenas é necessário que os *kernels* sejam emitidos para *streams* distintas que não a *stream* de sistema.

- Para efetuar execução de *kernels* de forma paralela é necessário que os *kernels* sejam emitidos para *streams* distintas que não a de sistema e é necessário que a *compute capability* do sistema seja igual ou superior a 2.0.

- Para efetuar execução de *kernels* em simultâneo com cópias de dados é necessário que o trabalho seja emitido para *streams* distintas que não a de sistema. As cópias deverão ser efetuadas de forma assíncrona utilizando acesso direto a dados.

- Para efetuar paralelização de cópias é necessário que as operações de cópia sejam emitidas para *streams* distintas e que não a de sistema. É ainda necessário que sejam efetuadas de forma assíncrona utilizando acesso direto de dados. As cópias a paralelizar deverão ter sentidos distintos, sendo uma do *host* para o *device* e a outra do *device* para o *host*. Não é possível paralelizar cópias no mesmo sentido.

Ao efetuar o *profiling* do fluxo de execução do algoritmo CUDA-MMP, verificamos que todo o processo de codificação se efetuava de forma sequencial na GPU recorrendo unicamente à *stream* de sistema. Esta incluía as cópias do *host* para o *device* dos blocos de resíduo, a execução dos *kernels* K1 e K2 e as cópias do *device* para o *host* dos resultados como ilustrado na Figura 5.9.

Figura 5.9: Análise do fluxo de execução utilizando a *stream* de sistema

Sabendo que o processo de codificação é efetuado para todos os modos de predição, verificámos que seria possível inserir algum tipo de paralelismo. Neste caso a utilização de *streams* é uma ótima solução uma vez que é necessário inserir dependências sequencias durante o processo de codificação. Para cada modo de predição é necessário que a cópia do resíduo, a execução do *kernel* K1 e do *kerel* K2 e a cópia dos resultados sejam efetuados por essa ordem como ilustrado na Figura 5.10.

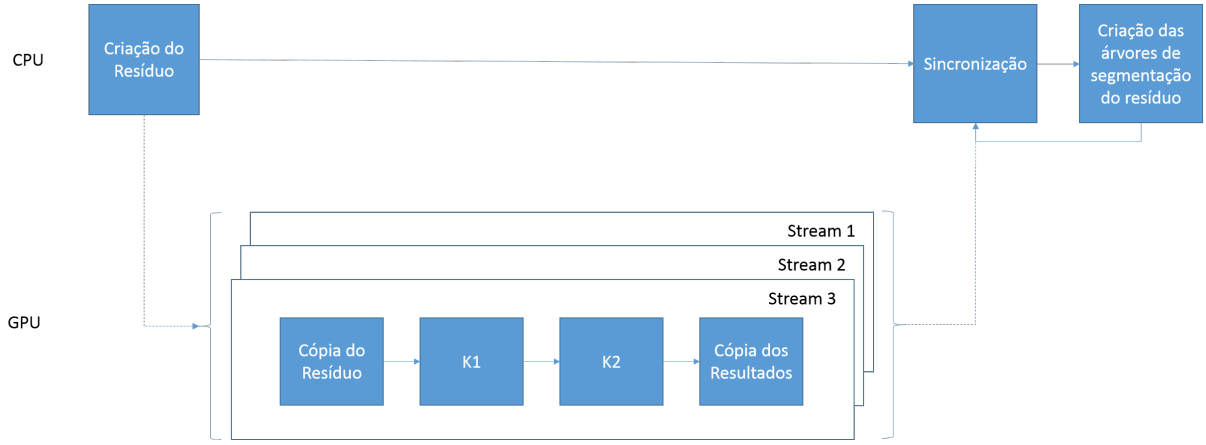

Figura 5.10: Novo fluxo de execução utilizando múltiplas *streams*

Utilizando diversas *streams* é assim possível inserir as dependências de fluxo exigidas pelo algoritmo para cada modo de predição individual paralelizando as execuções dos *kernels*. Para permitir também a paralelização de ambos os tipos de cópias, dos resíduos do *host* para o *device* e dos resultados do *kernel* K2 do *device* para o *host*, é também necessário substituir as utilizações de cópias síncronas por cópias de dados assíncronas em conjunto com a tecnologia de acesso direto a dados. Esta que permite ao *device* aceder a determinados endereços que não estão paginados de forma direta sem a necessidade da intervenção do *host*. Na procura de um melhor desempenho, durante a etapa de inicialização do algoritmo é criada uma *stream* para cada modo de predição existente. Esta é reutilizada para todos os processos de codificação do modo de predição ao qual a mesma está associada, evitando assim perdas de tempo com a constante criação e destruição de *streams*.

Resultados Experimentais

Recorrendo ao uso de *streams* foi possível delinear um fluxo de execução onde cada *stream* distinta efetua o fluxo de execução para um modo de predição distinto, possibilitando também a paralelização entre execução de *kernels* entre si em conjunto com a cópia de valores

tanto de *device* para *host* como de *host* para *device*. Obviamente, foram mantidas todas as dependências sequenciais necessárias.

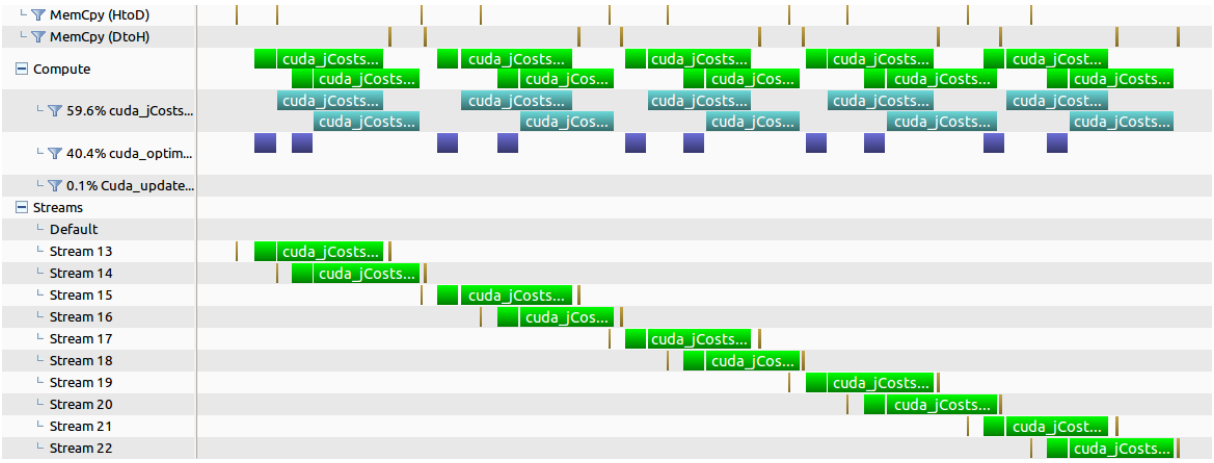

Figura 5.11: Análise do fluxo de execução utilizando múltiplas *streams*

A Figura 5.11, obtida pela ferramenta NVVP, mostra o fluxo de execução assente em múltiplas *streams*. Como se pode verificar, os processos de cópia e *kernels* K1 e K2 são agora emitidos para o *device* para todos os resíduos gerados em simultâneo e de forma assíncrona, possibilitando a paralelização dos vários fluxos de execução. Como foi explicado anteriormente, a paralelização de fluxos apenas ser possível quando o *device* tem recursos suficientes para tal, pois caso contrário mesmo utilizando a implementação de invocações assíncronas, os diversos fluxos de execução são escalonados de forma intercalar.

Sistema	**Tempo de Referência (s)**	**Tempo (s)**	**speedup**
Servidor 1 GTX Titan	1048.7986	903.4948	1.160
Servidor 1 GTX 680	1123.8206	1051.5998	1.068
Servidor 2 GTX Titan	1121.018	975.3612	1.149
Servidor 2 GTX 680	1177.833	1125.6702	1.046

Tabela 5.9: Resultados após a implementação de *streams*

Como se pode verificar na Tabela 5.9 ambas as GPUs mostraram melhorias significativas, obtendo um *speedup* compreendido entre 1.046 e 1.160. No entanto em ambos os ambientes, o melhor *speedup* obtido verificou-se para a placa GTX Titan. Este comportamento era esperado uma vez que quanto maior for a quantidade de recursos disponíveis na placa maior será a melhoria de desempenho atingido utilizando a paralelização de trabalho lógico obtido com a implementação de *streams*.

5.2.2 Diminuição de invocações à *API* CUDA

Análise e *Profiling*

Como foi discutido no Capítulo 3, o desempenho da arquitetura CUDA está intrinsecamente ligado ao nível de paralelismo passível de ser atingido. Esta utiliza uma estratégia de dividir e conquistar, assente em milhares de fluxos de execução a decorrer em paralelo sobre uma plataforma composta por centenas ou milhares de núcleos de processamento, cada núcleo com uma capacidade computacional modesta.

Quando é possível utilizar os milhares de núcleos para efetuar o trabalho de forma paralela, estes conseguem obter desempenhos superiores quando comparados com desempenhos de processadores convencionais. No entanto o mesmo não acontece quando o algoritmo não é paralelizavél. Nessas situações, como apenas um pequeno número de núcleos pode efetuar trabalho útil de forma concorrente, os processadores convencionais conseguem obter resultados superiores devido aos seus grandes desempenhos individuais [65].

Para além do nível de paralelização é também importante que o algoritmo tenha uma grande necessidade de poder computacional. Mesmo que um algoritmo seja completamente paralelizavél, caso o mesmo não necessite de um grande poder computacional, poderá acontecer que apenas sejam necessários algumas dezenas de núcleos ou até menos para paralelizar todo o trabalho, não sendo possível atingir grandes desempenhos devido à mesma razão explicada anteriormente.

Assim sabemos que o desempenho atingível utilizando a arquitetura CUDA está condicionado por dois fatores: (1) o nível de paralelização passível de ser atingido, ou seja, é todo o algoritmo paralelizável ou apenas parte do mesmo e (2) a complexidade computacional necessária à sua paralelização, pois caso a parte paralelizável do algoritmo não necessite de um grande poder computacional, podem ser atingidos melhores resultados utilizando um uma situação sequencial numa plataforma de maior poder computacional.

Numa situação ideal onde todo o algoritmo é paralelizável apenas uma invocação a um *kernel* poderá ser o suficiente para efetuar todo o trabalho, no entanto na maioria dos casos este não é o cenário existente. Não sendo todo o algoritmo paralelizável ou sendo necessário múltiplas invocações a *kernels* por forma a permitir uma paralelização faseada. Isto significa que o algoritmo deverá ser quantificado por forma a permitir a sua implementação utilizando uma arquitetura heterogénea, sendo utilizada apenas uma implementação paralela quando necessário e possível.

O algoritmo CUDA-MMP encontra-se no segundo cenário descrito anteriormente, tendo dois pontos distintos de grande dependência que impossibilitam a sua total paralelização:

1. O algoritmo utiliza um dicionário adaptativo que é atualizado após a codificação de cada macro bloco por forma a utilizar o contexto dos macro blocos já codificados para melhorar a qualidade de compressão. Assim sendo, sempre que se codifica um macro bloco é necessário que o bloco anterior tenha sido codificado e o dicionário atualizado, existindo assim uma dependência sequencial entre macro blocos contíguos. Esta característica impossibilita a paralelização da codificação de vários macro blocos de forma concorrente, pois as características adaptativas do dicionário seriam perdidas. Um outro problema é a utilização de uma taxa distorção que é adaptada entre codificações de macro blocos, tendo em conta a codificação anterior.

2. O algoritmo utiliza também um esquema de particionamento flexível para o bloco de predição gerado [13]. Este esquema foi introduzido na versão MMP-FP [14] e permite que um bloco de predição seja gerado utilizando uma concatenação de diversos blocos de predição de níveis inferiores utilizando um mesmo ou diferentes modos de predição. Por forma a escolher qual a melhor forma de representar um macro bloco, este é codificado utilizando todas as combinações possíveis de segmentações de predição para todos os possíveis modos de predição existentes. Só por si, o processo de quantificação pode ser executado de forma concorrente pois não existe nenhuma dependência entre a quantificação de diferentes blocos. No entanto a segmentação da predição introduz dependências sequenciais uma vez que cada resíduo gerado tem como base a reconstrução do bloco anterior, sendo empregue a fronteira dos blocos do dicionário utilizados para codificar o resíduo anterior. Assim sendo, não é possível efetuar a quantificação de todos os blocos de resíduo gerados em simultâneo.

Devido a estas duas as dependências, o algoritmo MMP obriga a uma implementação paralela parcial e faseada articulada com a parte sequencial por forma a efetuar a ligação entre as diversas invocações paralelas.

O CUDA-MMP implementa uma execução paralela na etapa de quantificação de todas as partições existentes para um único bloco de resíduo, ou seja para todas as partições existentes em cada árvore de segmentação da predição, sendo efetuada uma invocação ao *kernel* de quantificação para cada árvore de segmentação do resíduo criada. Utilizando o esquema de segmentação flexível, um macro bloco pode ser dividido até um máximo de 961 partições dependendo da profundidade máxima do particionamento da predição. Sabendo que para cada partição poderão existir até 10 modos de predição disponíveis, o *kernel* de quantificação é invocado um elevado número de vezes.

A Tabela 5.10 apresenta o número médio de invocações ao *kernel* de quantificação para

Nível máximo de segmentação	Número de invocações / macro bloco
0	218727
3	26819
8	2995
15	167

Tabela 5.10: Número de invocações do *kernel* K1 por macro bloco

diversas combinações entre a profundidade máxima do particionamento da predição. Como se pode verificar em determinadas situações o número de invocações poderá chegar em média aos vários milhares de invocações por macro bloco.

Sempre que é efetuada uma invocação a um *kernel* utilizando a *API Runtime*, a mesma efetua um conjunto de operações tal como a cópia dos parâmetros para a memória das constantes e a configuração do contexto onde o *kernel* deverá ser executado. Para além do tempo de execução consumido por este tipo de operações, a invocação de um *kernel* tem também associado um tempo de latência associado à comunicação entre o *host* e o *device*. Isto significa que a constante invocação de *kernels* deteriora o desempenho em comparação com uma única invocação que efetue todo o mesmo trabalho.

Uma outra vantagem de utilizar uma única invocação está relacionada com a possível reutilização do contexto de execução atual, ou seja, a reutilização de recursos já disponíveis como, por exemplo, reutilização de valores que se encontrem em registos ou em memória *cache*. Desta forma aumenta-se o desempenho dos pedidos de memória, evitando mesmo por completo a emissão de algumas instruções de memória, uma vez que as mesmas poderão já se encontrar nos registos.

Quando não é possível diminuir o número de chamadas aos *kernels*, uma solução alternativa é a paralelização de trabalho ao nível da CPU. Esta abordagem é viável pois as invocações aos *kernels* são feitos de forma assíncrona, assim sendo a mesma não é bloqueante e é possível atribuir trabalho à CPU enquanto a GPU não efetua trabalho. No entanto no algoritmo CUDA-MMP o mesmo não é possível, pois o trabalho realizado pela CPU depende dos resultados do trabalho efetuado na GPU. Assim sendo a única possibilidade é a diminuição do número de invocações.

Após uma análise ao *kernel* de quantificação verificou-se que o mesmo executa as mesmas instruções para os distintos resíduos gerados a partir de uma dada partição, seguindo precisamente o mesmo fluxo de execução. A única diferença entre as várias iterações de quantificação são os dados sobre os quais as instruções irão ser executadas. Assim foi implementado

um *kernel* que em vez de efetuar a quantificação uma única vez para um único bloco de resíduo, efetua o trabalho tantas vezes quantas as necessárias para os diversos blocos de resíduo numa única interação, sendo o número de blocos de resíduo parametrizável.
O *kernel* K1 tem como objetivo efetuar a comparação de todas as partições existentes no bloco de resíduo recebido como parâmetro com todos os blocos do mesmo tamanho existentes no dicionário. Para tal, o mesmo recebe como parâmetros dados sobre o contexto atual do processo de codificação tais como o bloco a quantificar, o nível do mesmo, o dicionário atualizado e o número de blocos em cada nível do dicionário. Utilizando estes dados, cada *thread* sequencial irá efetuar o cálculo do custo entre uma partição do bloco de resíduo com um bloco sequencial do dicionário guardando o seu resultado.
A quantificação é efetuada por ordem crescente de índice do dicionário e de tamanho das partições, sendo efetuada em primeiro lugar a quantificação das partições mais pequenas. Cada *thread* consecutiva irá efetuar a comparação de uma partição consecutiva, isto significa que a *thread* com o ID 0 irá cálcular o custo entre a partição número 0 de tamanho 1x1 (nível 1) e o bloco do dicionário de nível 1 com o índice 0, a *thread* número 1 irá efetuar o cálculo do custo entre a partição número 0 de tamanho 1x1 e o bloco do dicionário de nível 1 com o índice 1, e assim sucessivamente até que não existam mais blocos no dicionário de nível 1 com os quais se possa efetuar o cálculo do custo. De seguida o mesmo processo é repetido para a partição número 1 de tamanho 1x1. Uma vez efetuado o cálculo dos custos entre todas as partições existentes no bloco de resíduo de nível 0 com todos os blocos de nível 0 do dicionário, todo o processo é repetido para as partições do nível seguinte, até que não existam mais partições a quantificar. Sabendo que o dicionário pode crescer até conter milhares de blocos, o número de quantificações a efetuar deverá ser bastante elevado sendo normal que não exista um número de *threads* suficiente para que cada *thread* apenas efetue trabalho uma única vez. Uma técnica de *stride* foi implementada para aquando esta situação acontece.

Listing 5.1: Pseudo código do método de quantificação

```
INICIO
PARA CADA PARTIÇÃO
    PARA CADA BLOCO DO DICIONÁRIO
        CÁLCULAR CUSTO
        GUARDAR CUSTO
    FIM
FIM
```

Todo este processo é efetuado em invocações diferentes para blocos de resíduo distintos, mesmo quando gerados a partir da mesma partição. Por forma a diminuir o número de invocações todo o processo foi fundido para a quantificação dos vários resíduos gerados a partir de uma mesma partição do bloco original. Para tal um novo ciclo foi introduzido durante a comparação entre a partição de resíduo e um único bloco do dicionário.

Listing 5.2: Pseudo código do novo método de quantificação implementado

```
INICIO
PARA CADA PARTIÇÃO
    PARA CADA BLOCO DO DICIONÁRIO
        PARA CADA BLOCO DE RESIDUO
            CÁLCULAR CUSTO
            GUARDAR CUSTO
        FIM
    FIM
FIM
```

Como se pode ver nas Listagens 5.1 e 5.2 a única diferença entre ambas as implementações foi a inserção de um ciclo que permite que cada *thread* efetue a comparação entre uma mesma partição e um mesmo bloco do dicionário mas para blocos de resíduo diferentes. Utilizando esta implementação é possível reduzir em média o número de chamadas em um fator igual à média do número de predições disponíveis.

Esta nova implementação permite também a reutilização de contexto. Durante o ciclo descrito anteriormente, cada *thread* necessita de efetuar uma etapa de contextualização para cada iteração dos ciclos a executar. Esta contextualização contém dados tais como o cálculo do tamanho da partição e bloco do dicionário a comparar, o cálculo de quais os índices dos píxeis da partição a serem utilizados na comparação ou o índice do dicionário onde o bloco do dicionário se encontra. Sendo efetuada a quantificação dos vários blocos de resíduo em invocações distintas, todo o processo de contextualização irá ser efetuado tantas vezes quantos modos de predição serão utilizados. Efetuando a quantificação de todos os blocos de resíduo gerados para uma única partição de um macro bloco em uma única invocação, permite a reutilização total do contexto. Esta reutilização permite reduzir o número de operações aritméticas necessárias pois o cálculo dos valores é efetuado um menor número de vezes. A reutilização permite ainda a redução de tempos de acesso de memória, pois uma vez recolhidos os valores do bloco do dicionário da memória global para os registos, estes

são mantidos nos registos durante a quantificação das partições de todos os blocos de resíduo.

Resultados Experimentais

Após a implementação do novo *kernel* K1, verificámos uma redução elevada do número de invocações à *API*.

Nível máximo de segmentação	Número de invocações / macro bloco
0	31381
3	4501
8	661
15	69

Tabela 5.11: Número de invocações do *kernel* K1 por bloco com nova implementação

A Tabela 5.11 mostra uma diminuição compreendida entre 50% e 85% dependendo da profundidade utilizada, sendo a otimização tão eficaz quanto maior for a segmentação desejada.

Sistema	Tempo de Referência (s)	Tempo (s)	speedup
Servidor 1 GTX Titan	863.577	784.218	1.1012
Servidor 1 GTX 680	937.403	786.494	1.1918
Servidor 2 GTX Titan	909.063	853.085	1.0656
Servidor 2 GTX 680	1003.463	860.448	1.1662

Tabela 5.12: Resultados após a redução de chamadas ao *kernel* K1

A Tabela 5.12 demonstra o ganho de desempenho obtido com as alterações introduzidas. Como se pode verificar a redução de invocações teve um efeito positivo obtendo um *speedup* compreendido entre 1,0656 e 1,1918.

5.2.3 Branching

Análise e *Profiling*

A plataforma CUDA utiliza o modelo de execução *Single Instruction Multiple Thread* (SIMT), isto significa que cada instrução é emitida para diversas *threads* em simultâneo, sendo estas executadas de forma paralela. A arquitetura CUDA utiliza este modelo de execução no contexto de *warps*, emitindo a mesma instrução para todas as *threads* pertencentes a um mesmo *warp*. No entanto é possível que o fluxo de execução dentro de um *warp* seja divergente ao longo da sua execução, contendo *threads* que seguem fluxos de execução distintos devido à

combinação de instruções lógicas e de controlo de fluxo. Neste cenário é impossível utilizar um modelo SIMT, uma vez que cada *thread* irá executar zonas de código distintas.
O modelo de execução implementado permite a execução de *threads* com fluxos divergentes mesmo quando as mesmas se encontram dentro do *warp*. Para tal, duas situações distintas são contempladas pela arquitetura CUDA: (1) o compilador consegue utilizar mecanismos de predicados por forma a manter apenas um único fluxo de execução evitando um aumento de instruções emitidas e executadas e (2) o compilador separa o fluxo de execução das *threads* pertencentes a um mesmo *warp* dividindo as mesmas em grupos onde todas as *threads* seguem o mesmo fluxo, serializando posteriormente a execução dos vários fluxos existentes consequentemente aumentando o número de instruções emitidas e executadas [68].
A primeira situação é possível e comum quando a execução de um determinado bloco de código de pouca complexidade é condicional, ou quando o compilador consegue efetuar técnicas de *loop unrolling* [69][70], onde os valores utilizados para as condições dos ciclos são conhecido em tempo de compilação.

Listing 5.3: Pseudo código do cálculo do erro quadrático sequencial

```
VARIAVEIS
threadIdx
erroQuadratico

INICIO
erroQuadratico = 0

SE threadIdx > 10
    erroQuadratico += 10
erroQuadratico += 10
FIM
```

A Listagem 5.3 demonstra uma situação onde um bloco de código de baixa complexidade é executado de forma condicional. No cenário apresentado é bastante provável que o compilador efetue uma otimização de predicado, permitindo assim que não exista divergência do fluxo de execução.
Quando o compilador não consegue efetuar este tipo de otimizações, sempre que existe algum tipo de divergência no fluxo de execução em alguma das *threads* pertencentes a um *warp*, a plataforma agrupa todas as *threads* cujo fluxo de execução é igual. De seguida cada

grupo de *threads* é executado de forma distinta, sendo apenas emitidas as mesmas instruções para todas as *threads* pertencentes ao mesmo grupo. Assim o desempenho é degradado em um fator tanto maior quanto o número de grupos com fluxos de execução distintos.

Por forma a melhorar o seu desempenho em implementações sequenciais, o algoritmo MMP efetua uma otimização aquando do cálculo do erro quadrático de um determinado bloco. Assim, caso o erro quadrático calculado seja superior ao melhor encontrado anteriormente, o processamento é interrompido mesmo que o cálculo ainda não tenha terminado, passando para o processamento do próximo bloco, tal como demonstrado na Listagem 5.4. A otimização assenta no facto que uma vez que o custo do bloco é já superior a um encontrado anteriormente, o bloco em processamento nunca será escolhido.

Listing 5.4: Pseudo código do cálculo do erro quadrático sequencial

```
VARIAVEIS
erroQuadratico
menorCustoEncontrado

INICIO
PARA CADA pixelX SE erroQuadratico < menorCustoEncontrado
    PARA CADA pixelY SE erroQuadratico < menorCustoEncontradoZ
        erroQuadratico += bloco[pixelX][pixelY]
        GUARDAR CUSTO
    FIM
FIM
```

A implementação do *kernel* K1 contempla também esta otimização. Para cada *thread* e para cada interação do ciclo que efetua o cálculo do erro quadrático é verificado se o valor do erro quadrático é superior ao encontrado anteriormente. Caso a condição anterior seja verdadeira a *thread* em questão irá alterar o seu fluxo de execução, saltando o ciclo.

Esta otimização apenas é vantajosa no contexto atual quando todas as *threads* de um único *warp* seguem o mesmo fluxo, ou seja, quando todas as *threads* saltam o ciclo. Quando isto não acontece, a otimização poderá revelar resultados inferiores pois o mesmo número de instruções será emitido, uma vez que irá existir pelo menos uma *thread* que necessita de processar o ciclo mais as instruções necessárias para a verificação da condição anterior.

Resultados Experimentais

Após a remoção da otimização encontrada na versão sequencial verificou-se um aumento de desempenho. Uma análise das várias métricas de instruções revelou que o número de instruções executadas e emitidas tem um impacto direto na eficiência do algoritmo.

Métrica	**Impl. com otimização**	**Impl. sem otimização**	**Taxa**
Instruções Executadas/*kernel*	30324	29087	1.0425
Instruções Emitidas/*kernel*	33395	32154	1.0385
Instruções de *Branch*/*warp*	2850	2746	1.0862
Número de *Branches* Divergentes/*warp*	311	286	1.0379

Tabela 5.13: Médias das métricas de instruções para o *kernel* K1

Como se pode verificar na Tabela 5.13, a remoção da otimização para saída precoce resultou numa redução significativa nas métricas obtidas. Podemos ver que o facto de existir menos uma condição no algoritmo diminuiu tanto o número de*branches* divergentes como o número de instruções de*branch*. Ambos estes fatores influenciam tanto o número de instruções executadas, como o número de instruções emitidas pois as *threads* que antes eram executadas em*branches* divergentes e obrigavam o escalonador de instruções a emitir instruções que só seriam executadas no seu fluxo são agora executadas no fluxo de execução principal, utilizando as mesmas instruções que o*branch* não divergente.

Sistema	**Tempo de Referência (s)**	**Tempo (s)**	**speedup**
Servidor 1 GTX Titan	784.218	774.705	1.0122
Servidor 1 GTX 680	786.494	777.388	1.0117
Servidor 2 GTX Titan	853.085	843.106	1.0118
Servidor 2 GTX 680	860.448	850.536	1.0116

Tabela 5.14: Resultados após a implementação diminuição de *branching*

Como se pode verificar na Tabela 5.14 a diminuição do número de *branches* divergentes e consequentemente do número de instruções tem um impacto direto, embora reduzido, no tempo de execução do algoritmo. A nova implementação onde a otimização no cálculo dos erros quadráticos foi removida obteve um *speedup* compreendido entre 1.0116 e 1.0122. O baixo aumento de desempenho deve-se a dois fatores: (1) à baixa taxa de diminuição de *branches* divergentes e de instruções executadas e emitidas; e (2) ao facto da execução de instruções ser efetuada para esconder os tempos de latência dos acessos à memória. O primeiro fator tem um impcato direto, pois embora a otimização tenha sido implementada com

sucesso, devido a restrições de fluxo existentes no algoritmo apenas foi possível remover um reduzido número de divergências de fluxo. O segundo fator influencia também o desempenho pois diminuindo o tempo de execução dedicado à execução de instruções e diminuindo o número de instruções entre cada pedido de memória traduz-se em (1) mais tempo a escalonar *warps* e (2) uma menor janela temporal para executar instruções de acesso à memória em paralelo com instruções de tipos distintos.

5.3 Optimizações de Árvores de Segmentação do Resíduo

Para além das otimizações orientadas par a implementação paralela, foram estudadas e implementadas um conjunto de otimizações para a parte sequencial do algoritmo. Estas otimizações focaram a geração das árvores de segmentação do resíduo.

5.3.1 Optmização e Reutilização de Estruturas

Análise e *Profiling*

Durante a etapa de codificação de um dado bloco, para cada modo de predição disponível, e uma vez efetuada a redução do mesmo na GPU, a CPU constrói a árvore de segmentação do resíduo gerado. Este processo gera sempre uma árvore completa com uma partição raiz de nível igual ao que está a ser codificado sendo a mesma podada apenas posteriormente. Isto significa que para cada macro bloco, esta árvore é gerada tantas vezes quantas o número de nós existentes na árvore de segmentação da predição multiplicado pelo número de predições disponíveis para cada partição de predição. Sendo este processo repetido para cada macro bloco, a mesma árvore de segmentação do resíduo é gerada um elevado número de vezes. Sabemos também que de cada vez que a árvore de segmentação do resíduo é gerada, a mesma é alocada em memória e libertada no fim do processo de poda. A análise da codificação da imagem Lena (512 x 512 píxeis) demonstrou que a árvore de segmentação do resíduo é gerada cerca de 5864816 vezes totalizando 257143274 operações de alocação de memória e o mesmo número de operações de libertação de memória. Este número será tão maior quanto o número de macro blocos a codificar.
O processo de codificação de partições de predição é feitos de forma sequencial. Isto significa que cada árvore de segmentação do resíduo é gerada também de forma sequencial, não sendo nunca necessário manter em memória duas árvores de segmentação do resíduo em simultâneo. Isto permite que em vez de efetuar a manutenção da memória para cada

árvore de segmentação do resíduo é possível alocar a mesmo uma única vez e reaproveitá-la para a criação de árvores posteriores, diminuindo significativamente a quantidade de tempo necessária para efetuar a manutenção de memória.

Resultados Experimentais

A nova implementação efetua a alocação de uma árvore completa, ou seja com uma partição de nível 24 como raiz, durante a fase de inicialização do algoritmo. Sendo a mesma reutilizada na geração de todas as árvores de resíduo como ilustrado na Tabela 5.15.

Versão	# de partições alocadas	# de Árvores alocadas
Versão de Referência	257143274	5864816
Versão Alocação Única	961	1

Tabela 5.15: Comparação do número de alocações para a codificação da imagem Lena

Sistema	Tempo de Referência (s)	Tempo (s)	speedup
Servidor 1 GTX Titan	774.705	661.370	1.1713
Servidor 1 GTX 680	777.388	662.850	1.1727
Servidor 2 GTX Titan	843.106	726.242	1.1609
Servidor 2 GTX 680	850.536	738.744	1.1513

Tabela 5.16: Resultados após a implementação da reutilização de estruturas de memória

Como se pode verificar na Tabela 5.16, a diminuição de chamadas ao sistema para manutenção de memória teve um impacto significativo no desempenho do algoritmo obtendo um *speedup* entre 1.1513 e 1.1727.

5.3.2 Otimizações da Geração das Árvores de Segmentação do Resíduo

O esquema de particionamento flexível da predição introduziu um ganho substancial na qualidade de compressão. No entanto este ganho faz-se à custa do acréscimo do poder computacional necessário, pois o mesmo obriga a um maior número de quantificações. Anteriormente na versão pré-predição, apenas era efetuado um único processo de codificação para cada bloco de 16 x 16 e para cada modo de predição disponível isto é, para todo o processo de criação do resíduo, quantificação e redução do mesmo com os blocos do dicionário e criação da árvore de segmentação do resíduo.

O novo esquema de particionamento flexível da predição requer mais computação em dois pontos distintos do algoritmo: (1) introdução de uma árvore de segmentação da predição e

(2) geração de múltiplas árvores de segmentação do resíduo por macro bloco.

Sem a utilização de qualquer esquema de segmentação da predição, apenas um resíduo é gerado para cada macro bloco e para cada predição, sendo gerados tantos resíduos distintos quantos modos de predição disponíveis existem. Cada resíduo é então codificado utilizando uma única árvore de segmentação do resíduo, tendo como nó raiz a partição de nível 24 (16x16) e a árvore com um menor custo de codificação é escolhida como um todo para codificar o bloco. A introdução de um esquema de segmentação da predição introduziu a geração de uma árvore de segmentação de resíduo para cada predição e para cada partição distinta. Cada nó da árvore de segmentação da predição representa todo um processo de geração de resíduo, quantificação do mesmo, redução dos resultados e construção do bloco codificado. Isto significa que cada nó irá ter associada uma árvore de segmentação do resíduo, tendo como base um bloco de nível igual ao nível da partição do resíduo gerado.

A vizinhança utilizada para a geração da predição é a reconstrução dos blocos vizinhos, ou seja, os mesmos foram já codificados e reconstruídos utilizando a melhor árvore de codificação encontrada, simulando assim o processo de descodificação. Este processo garante que o resultado obtido no descodificador é igual ao obtido no codificador.

Uma vez que todos os modos de predição utilizam a fronteira da esquerda, a fronteira da direita ou ambas, sabemos que existe uma dependência sequencial obrigando a que pelo menos um destes blocos tenham de ser codificados em primeiro lugar para que os blocos posteriores tenham acesso a uma fronteira já codificada na qual irão basear a sua predição. Devido a esta dependência sequencial é necessário que as partições que se encontrem à esquerda tenham precedência sobre as partições que se encontrem à direita. A mesma precedência é necessária para as partições que se encontrem no topo do bloco relativamente às partições que se encontrem numa posição mais abaixo. Por forma a seguir esta regra, o algoritmo foi implementado utilizando uma expansão da árvore de segmentação do resíduo em profundidade dos níveis mais elevados para os níveis mais baixos, seguindo a seguinte ordem de precedência: esquerda, direita, cima e baixo.

Análise e *Profiling*

Durante toda a expansão de um determinado ramo das árvores de segmentação do resíduo, e dependendo da partição a ser expandida, a fronteira utilizada para a geração da predição poderá manter-se constante para alguns modos de predição. Caso a fronteira seja a mesma, sabemos que o bloco de resíduo a ser gerado será igual a um resíduo gerado anteriormente durante a codificação de uma partição de nível superior. Sendo o resíduo gerado igual a

um resíduo já gerado anteriormente, sabemos que já foi, anteriormente, expandida e podada uma árvore de segmentação de resíduo igual. Isto significa que se for guardado o estado de todas as diferentes árvores de resíduo geradas ao longo da codificação de um macro bloco, as mesmas podem ser reaproveitadas para árvores de níveis inferiores dependendo apenas na fronteira do bloco. Consequentemente, caso a fronteira do bloco de nível inferior seja a mesma que a fronteira do bloco de nível superior este processo pode ser aplicado. Isto significa que todo o processo de codificação da predição de nível inferior pode ser evitado, sendo reaproveitada a árvore anterior.

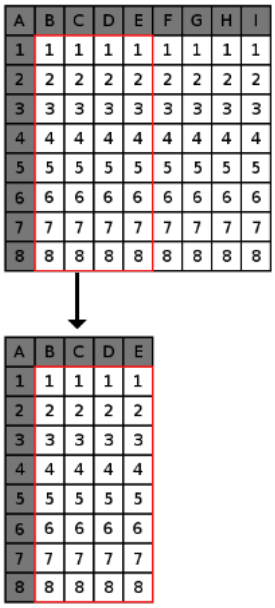

Figura 5.12: Comparação de predições entre níveis distintos para modo de predição horizontal

A Figura 5.12 apresenta uma situação onde a árvore de segmentação do resíduo gerado para o bloco de nível superior, irá conter a árvore de segmentação do resíduo do bloco de nível inferior. Sendo neste caso desnecessária a geração da árvore de nível inferior uma vez que a mesma será igual a um ramo da árvore de nível superior.
Analisando os vários modos de predição, sabemos que existem dois tipos distintos: (1) predições lineares e (2) predições não lineares. Para as predições lineares, dependendo da posição da partição a ser codificada, podemos inferir quais os modos de predição que podem ser reaproveitados de árvores anteriores pois, os mesmos apenas dependem da posição da partição atual relativamente à partição de nível superior de onde se pretende reutilizar a folha da árvore de segmentação de resíduo. Dentre os modos de predição apresentados na Secção 2.4.2,

são descritos de seguida quais e em que circunstâncias podem ser reaproveitados:

- Esquerda: para partições que se encontrem à esquerda relativamente à partição de nível superior das quais foram geradas podem ser reaproveitados os modos de predição vertical, horizontal, diagonal baixo direita, vertical direita, horizontal baixo e horizontal cima.

- Direita: para partições que se encontrem à direita relativamente à partição de nível superior das quais foram geradas podem ser reaproveitados os modos de predição vertical, diagonal baixo esquerda e vertical esquerda.

- Cima: para partições que se encontrem acima relativamente à partição de nível superior das quais foram geradas podem ser reaproveitados os modos de predição vertical, horizontal, diagonal baixo esquerda, diagonal baixo direita, vertical direita, horizontal baixo e vertical esquerda.

Para as predições não lineares, o seu possível reaproveitamento depende das características do modo de predição em questão. Para o modo de predição MFV, caso o valor gerado seja o mesmo que para a árvore de nível anterior, então o mesmo pode ser sempre reaproveitado para todas as direções, uma vez que toda a predição será constituída pelo mesmo valor. Para o modo de predição LSP, o processo de reaproveitamento não pode ser aplicado uma vez que a fronteira do mesmo não se limita apenas aos píxeis fronteira entre os dois blocos.

Resultados Experimentais

Uma característica importante para que o processo de reaproveitamento possa ser efetuado é a necessidade de manter a árvore de segmentação do resíduo constante até que todas as partições filhas tenham sido expandidas na árvore. Ou seja, é necessário que uma vez gerada a árvore para um bloco de nível 24 (16x16) a mesma se mantenha inalterada em memória até que ambos os blocos de nível 23 (16x8) e 22 (8x16) tenham sido expandidos e possivelmente reaproveitados. O mesmo irá suceder para cada um desses blocos. Isto significa que uma vez expandida a árvore de segmentação de nível 23 (16x8) a árvore tem de se manter inalterada em memória para que seja possível reaproveitar os blocos de nível 20 (4x16) e 15 (8x8).
Uma vez que cada processo de geração da árvore de segmentação do resíduo é efetuado de forma sequencial, o algoritmo reutiliza sempre uma única estrutura de memória para todos os processos de codificação, fazendo uso da otimização referida na Secção 5.3.1. Devido à geração da árvore em profundidade, sabemos que o requisito descrito anteriormente não se

verifica, pois existindo apenas uma única estrutura de dados, esta será rescrita pelas partições de níveis inferiores antes que todos os filhos de uma partição sejam expandidos. Um outro problema deve-se à existência de vários modos de predição. De facto, existindo vários modos de predição, mas apenas uma estrutura de dados, a mesma será rescrita tantas vezes quantos modos de predição distintos existem para uma única partição. Isto impossibilita o reaproveitamento de todos os modos de predição exceto do último que ainda se encontra guardado em memória.

Para resolver estes dois problemas foram adicionadas duas novas dimensões à estrutura de dados que representa a árvore de segmentação do resíduo. Anteriormente a árvore era constituída apenas por duas dimensões que representavam o nível da partição e o índice de mesma. A nova estrutura de dados acrescenta duas novas dimensões. Uma das novas dimensões representa os modos de predição e é utilizada para impedir que diversos modos de predição sejam escritos sobre uma mesma estrutura de dados, preservando desta maneira todos os modos de predição gerados, uma vez que é criada uma estrutura de dados distinta para cada modo de predição distinto. A segunda dimensão tem como objetivo impedir que partições que pertençam a níveis inferiores reescrevam a mesma estrutura de dados utilizada por blocos de níveis superiores, possibilitando assim que uma partição da direita possa utilizar a técnica de reaproveitamento mesmo que partições esquerdas de níveis inferiores tenham sido codificadas, uma vez que as mesmas irão utilizar diferentes estruturas de dados.

Modo de Predição	**# de Utilizações**	**# de Reutilizações**	**% de Reutilização**
0	670112	500160	74.64
1	670112	500160	74.64
2	676864	309086	45.66
3	277057	150464	54.31
4	662440	327270	49.40
5	662440	327270	49.40
6	662440	327270	49.40
7	276096	150464	54.50
8	669120	331552	49.55
9	638135	0	00.00
total	5864816	2923696	49.85

Tabela 5.17: Taxa de reutilização de ramos da segmentação do resíduo para a imagem Lena

Sistema	Tempo de Referência (s)	Tempo (s)	speedup
Servidor 1 GTX Titan	536.526	506.649	1.0589
Servidor 1 GTX 680	525.614	502.784	1.0454
Servidor 2 GTX Titan	565.674	545.080	1.0377
Servidor 2 GTX 680	581.717	553.132	1.0516

Tabela 5.18: Resultados após a implementação da reutilização de ramos de resíduo

Após a execução do novo algoritmo de codificação com reutilização de ramos verificámos uma elevada taxa de reutilização para a maioria dos modos. Como é visível na Tabela 5.17, a maioria dos modos de predição obteve uma reutilização de ramos perto dos 50%. No entanto, estes resultados apenas se traduzem num *speedup* compreendido entre 1.0377 e 1.0589 (Tabela 5.18). Isto acontece pois nunca é possível reaproveitar o modo de predição LSP (modo 9). Assim é sempre necessário que o processo de codificação seja executado para este modo, sendo o mesmo o modo de predição com um maior consumo de poder computacional. Esta otimização reduz também o impacto da otimização descrita na Secção 5.2.3, uma vez que o impacto dessa otimização é dependente no número de trabalho que é efetuado pelo *kernel* de otimização, e a reutilização de ramos reduz a quantidade de trabalho a ser efetuada tanto pelo *kernel* de otimização como pelo *kernel* de redução.

5.4 Conclusão

Este capítulo descreve a análise e implementação de todas as otimizações efetuadas, tanto focadas na parte paralela como na parte sequêncial do algoritmo CUDA-MMP. As otimizações efetuadas na parte paralela do algoritmo focaram-se em otimizações relacionadas com o modo de funcionamento da tecnologia CUDA. Estas tiveram na generalidade um maior impacto positivo no desempenho e foram maioritariamente relacionadas com os acessos aos diferentes tipos de memória e o fluxo de execução dos *kernels*, tentando minimizar os tempos de latência das instruções e reaproveitar os dados que já se encontravam nas memórias com menores tempos de acesso permitindo dessa forma um menor tempo de resposta e uma grande diminuição de comutações entre *threads* ativas. Foi também focada a comunicação entre a CPU e a GPU por forma a reduzir os tempos de sincronização e comunicação entre ambas. As otimizações de memória foram as que obtiveram um maior impacto de desempenho atingindo *speedups* de 2.197. Embora não tenham sido tão bem sucedidas, as otimizações de fluxo e comunicação foram também benéficas para o desempenho do algoritmo atingindo *speedups* de 1.192.

As otimizações efetuadas na parte sequencial do algoritmo tiveram um maior foco no funcionamento do mesmo, tentado eliminar pontos de grande consumo de poder computacional. Estes focaram-se em minimizar a manutenção de memória e o aumentar reaproveitamento das estruturas criadas durante os processos de segmentação das árvores de resíduo. Este tipo de otimizações foi também bem sucedido atingindo *speedups* de 1.1727.

Embora quase todas as otimizações tenham tido um impacto benéfico no desempenho do algoritmo, as mesmas obtiveram tendencialmente benefícios abaixo dos esperados. Isto verificou-se maioritáriamente nas otimizações da parte paralela do algoritmo, onde as otimizações do algoritmo foram baseadas na resolução de limites impostos pela tecnologia, no entanto, após a resolução de um limite, logo outro era detetado, nem sempre sendo possível a resolução dos mesmos. Outra razão para os ganhos serem inferiores ao esperado é que em diversas situações, determinadas otimizações eram influenciadas por otimizações já antes introduzidas, nem sempre sendo possível criar uma sinergia perfeita entre todas as otimizações introduzidas.

Capítulo 6

Conclusão e trabalhos futuros

Este estudo teve como finalidade melhorar o desempenho do algoritmo MMP-Intra através da criação de um protótipo que explorasse a sinergia entre uma arquitetura de computação paralela e uma arquitetura de computação sequencial. Para tal foram averiguadas as tecnologias de codificação de dados, de computação sequencial e de computação paralela tal como o impacto que cada uma poderia ter. Ainda dentro dos objetivos do trabalho foi efetuado um processo iterativo de otimização na busca do melhoramento do protótipo anteriormente concebido, sendo que para tal foi efetuada uma extensa pesquisa no modo de funcionamento da tecnologia CUDA, tanto ao nível do funcionamento do *hardware* como da *framework* por forma a tirar o melhor partido da mesma.

A maioria dos estudos efetuados na área da computação paralela, demonstra que esta consegue obter um desempenho muito superior quando comparado com computação sequencial, isto, quando as características do algoritmo a implementar são favoráveis. Através de experiencias empíricas, o nosso estudo verificou a premissa anteriormente descrita através da implementação de um primeiro protótipo cujos resultados obtidos foram muito inferiores à implementação sequencial base, chegando a obter um desempenho cinco vezes inferior.

A análise do protótipo inicial demonstrou que o desempenho da tecnologia CUDA assenta grandemente em quatro fatores importantes: (1) possivel paralelização de trabalho, (2) existência de um fluxo de excecão que permita uma fácil sincronização entre dispositivos, (3) boa gestão de memória e (4) fluxo de execução uniforme, onde todo o trabalho siga o mesmo fluxo de execução minimizando ramificações. A otimização do protótipo CUDA-MMP, por forma a respeitar todos os quatro fatores descritos anteriormente, obteve um aumento de desempenho elevado chegando a ser quatro vezes mais rápido que o algoritmo sequencial MMP-Intra e 30 vezes mais rápido que a primeira implementação do protótipo.

Embora a implementação do protótipo tenha sido efetuada com sucesso e o mesmo tenha obtido ganhos visíveis, os mesmos continuam a traduzir-se num limite para o uso generalizado do algoritmo MMP, uma vez que o tempo final atingido continua a ser muito superior ao aceitável.

Como trabalho futuro deve ser considerado o estudo de uma implementação que recorra ao uso de diversas placas pertencentes à mesma arquitetura a trabalhar de forma cooperativa. Esta abordagem passa pelo uso de várias CPUs e/ou GPUs. Outro ponto a ser investigado é o funcionamento do modo de predição LSP. De facto tendo este um custo computacional bastante elevado, é importante efetuar a análise do mesmo por forma a compreender se é possível a execução do mesmo de forma paralela e qual o impacto em termos de tempo de execução final.

Bibliografia

[1] NVIDIA. *GeForce GTX TITAN Black Specifications GeForce*, accessed March 2016. `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-black/specifications`.

[2] NVIDIA. *GeForce GTX 680 Specifications GeForce*, accessed March 2016. `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-680/specifications`.

[3] David Taubman and Michael Marcellin. *JPEG2000 Image Compression Fundamentals, Standards and Practice: Image Compression Fundamentals, Standards and Practice*, volume 642. Springer Science & Business Media, 2012.

[4] Gary J Sullivan, J-R Ohm, Woo-Jin Han, and Thomas Wiegand. Overview of the high efficiency video coding (hevc) standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 22(12):1649–1668, 2012.

[5] Murilo Bresciani de Carvalho. *Compressão de sinais multi-dimensionais usando recorrência de padrões multiescalas*. PhD thesis, Universidade Federal do Rio de Janeiro, 2001.

[6] Claude Elwood Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001.

[7] Claude Shannon. Collected papers. 1993.

[8] Joy A Thomas and Joy A Thomas. *Elements of information theory*. Wiley New York, 2006.

[9] David A Huffman et al. A method for the construction of minimum redundancy codes. *proc. IRE*, 40(9):1098–1101, 1952.

[10] Ianh Willen, Radford M Neal, and John G Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6), 1987.

[11] Toby Berger. Rate distortion theory: A mathematical basis for data compression. 1971.

[12] Allen Gersho and Robert M Gray. *Vector quantization and signal compression*. Springer, 1992.

[13] Nuno Miguel Morais Rodrigues. Multiscale recurrent pattern matching algorithms for image and video coding. 2009.

[14] Nelson Carreira Francisco. *Estudo da Utilização de Recorrência de Padrões Multiescala na Codificação de Documentos Compostos*. PhD thesis, Universidade de Trás-os-Montes e Alto Douro, 2007.

[15] Iain EG Richardson. H. 264/mpeg-4 part 10 white paper. *White Paper/www. vcodex. com*, 2003.

[16] Iain E Richardson. *H. 264 and MPEG-4 video compression: video coding for next-generation multimedia*. John Wiley & Sons, 2004.

[17] Danillo Bracco Graziosi. *Contribuições à compressão de imagens com e sem perdas utilizando recorrência de padrões multiescala*. PhD thesis, Universidade Federal do Rio de Janeiro, 2011.

[18] Erich Leo Lehmann, George Casella, and George Casella. *Theory of point estimation*. Wadsworth & Brooks/Cole Advanced Books & Software, 1991.

[19] Steven Koiti Tsukamoto. *Arquitetura dos Supercomputadores*, accessed March 2016. `http://grenoble.ime.usp.br/~paulo/MAC0412/Monografias/monografia-steven.pdf`.

[20] TOP500.org. *Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P TOP500 Supercomputer Sites*, accessed March 2016. `http://www.top500.org/system/177999`.

[21] TOP500.org. *Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom TOP500 Supercomputer Sites*, accessed March 2016. `http://www.top500.org/system/177556`.

[22] Intel. *Intel Core Processor Family*, accessed March 2016. http://www.intel.com/content/www/us/en/processors/core/core-processor-family.html.

[23] Colin Donahue Jason Lowden. *AMD Bulldozer*, accessed March 2016. http://meseec.ce.rit.edu/551-projects/winter2011/2-2.pdf.

[24] Nancy B Stern. *From ENIAC to UNIVAC: An appraisal of the Eckert-Mauchly computers*. Digital Press Bedford, MA, 1981.

[25] William Aspray. *John von Neumann and the origins of modern computing*, volume 191. MIT Press Cambridge, MA, 1990.

[26] Daniel Page. *A Practical Introduction to Computer Architecture*. Springer, 2009.

[27] Uresh Vahalia. *UNIX Internals: The New Frontiers*. People's Posts & Telecommunications Publishing House, 2003.

[28] Michael Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972.

[29] Andrew S Tanenbaum and Wagner Luiz Zucchi. *Organização estruturada de computadores*. Pearson Prentice Hall, 2009.

[30] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

[31] Qualcomm. *Snapdragon Mobile Processors and Chipsets Qualcomm*, accessed March 2016. https://www.qualcomm.com/products/snapdragon.

[32] Peter M Kogge. *The Architecture of Pipelined Computers*. CRC Press, 1981.

[33] Off-Chip Communications LLC. *Low-Power Design*, accessed March 2016. http://www.low-powerdesign.com/article_HLS-book_082310.html.

[34] John Paul Shen and Mikko H Lipasti. *Modern processor design: fundamentals of superscalar processors*. Waveland Press, 2013.

[35] M.J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Higher Education, 2004.

[36] Hwang. Program and network properties. Technical report, University of Nebraska Omaha, 2012.

[37] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[38] Usenix. *Amdahl's Law & Parallel Speedup*, accessed March 2016. `https://www.usenix.org/legacy/publications/library/proceedings/als00/2000papers/papers/full_papers/brownrobert/brownrobert_html/node3.html`.

[39] David Luebke, Mark Harris, Naga Govindaraju, Aaron Lefohn, Mike Houston, John Owens, Mark Segal, Matthew Papakipos, and Ian Buck. GPGPU: General-purpose computation on graphics hardware. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[40] Dave Shreiner, Graham Sellers, John M Kessenich, and Bill M Licea-Kane. *OpenGL programming guide: The Official guide to learning OpenGL, version 4.3*. Addison--Wesley Professional, 2013.

[41] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, August 2004.

[42] NVIDIA. *Parallel Programming and Computing Platform*, accessed March 2016. `http://www.nvidia.com/object/cuda_home_new.html`.

[43] Khronos Group. *OpenCL - The open standard for parallel programming of heterogeneous systems*, accessed March 2016. `https://www.khronos.org/opencl/`.

[44] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[45] Samuel P Harbison and Guy L Steele. *C: a Reference Manual*. Prentice Hall PTR, 2002.

[46] NVIDIA. *Programming Guide :: CUDA Toolkit Documentation*, accessed March 2016. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#c-cplusplus-language-support`.

[47] PGI Compilers and Tools. *CUDA Fortran Programming Guide and Reference*, accessed March 2016. http://www.pgroup.com/doc/pgicudaforug.pdf.

[48] jcuda.org. *jcuda.org - Java bindings for CUDA*, accessed March 2016. http://www.jcuda.org.

[49] NVIDIA. *PyCUDA*, accessed March 2016. https://developer.nvidia.com/pycuda.

[50] NVIDIA. *CUDA Binary Utilities :: CUDA Toolkit Documentation*, accessed March 2016. http://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#axzz3MGYxS8vh.

[51] NVIDIA. Nvidia geforce gtx 980. 2014.

[52] David B Kirk and W Hwu Wen-mei. *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.

[53] NVIDIA. *Programming Guide :: CUDA Toolkit Documentation*, accessed March 2016. http://docs.nvidia.com/cuda/cuda-c-programming-guide.

[54] NVIDIA. *Programming Guide :: CUDA Toolkit Documentation*, accessed March 2016. http://docs.nvidia.com/cuda/cuda-c-programming-guide/#compute-capability-3-0.

[55] NVIDIA. *Programming Guide :: CUDA Toolkit Documentation*, accessed March 2016. http://docs.nvidia.com/cuda/cuda-c-programming-guide/#simt-architecture.

[56] NVIDIA. *Best Practices Guide :: CUDA Toolkit Documentation*, accessed March 2016. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#optimizing-cuda-applications.

[57] NVIDIA. *CUDA Programming Model Overview*, accessed March 2016. http://www.learningace.com/doc/4327789/f2aac47182d5518486f8888d745f1673/nvidia-02-basicsofcuda.

[58] Paulius Micikevicius. Local memory and register spilling. Technical report, NVIDIA, 2011.

[59] Mohamed Zahran. CUDA memories. Technical report, New York University, 2012.

[60] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, volume 17, pages 120–126. ACM, 1982.

[61] Yingying Tian, Sooraj Puthoor, Joseph L Greathouse, Bradford M Beckmann, and Daniel A Jiménez. Adaptive gpu cache bypassing. In *Proceedings of the 8th Workshop on General Purpose Processing using GPUs*, pages 25–35. ACM, 2015.

[62] NVIDIA. *Profiler :: CUDA Toolkit Documentation*, accessed March 2016. `http://docs.nvidia.com/cuda/profiler-users-guide/#axzz42aWSucl0`.

[63] NVIDIA. *Programming Guide :: CUDA Toolkit Documentation*, accessed March 2016. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/#device-memory-accesses`.

[64] NVIDIA. *NVIDIA Visual profiler*, accessed March 2016. `https://developer.nvidia.com/nvidia-visual-profiler`.

[65] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *SIGARCH Comput. Archit. News*, 38:451–460, June 2010.

[66] NVIDIA. *Programming Guide :: CUDA Toolkit Documentation*, accessed March 2016. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/#streams`.

[67] NVIDIA. *CUDA C/C++ Streams and Concurrency*, accessed March 2016. `http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf`.

[68] NVIDIA. *Programming Guide :: CUDA Toolkit Documentation*, accessed March 2016. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/#control-flow-instructions`.

[69] Vasily Volkov. *Unrolling parallel loops*, accessed March 2016. `http://www.cs.berkeley.edu/~volkov/volkov11-unrolling.pdf`.

[70] Christopher Cooper. *GPU Computing with CUDA Lecture 5 - More Optimizations*, accessed March 2016. http://www.bu.edu/pasi/files/2011/07/Lecture5.pdf.