Dissertação

Mestrado em Engenharia Informática – Computação Móvel

OpenCL-MMP – Codificação de Imagens com Sistemas com Múltiplos Núcleos

João Filipe Crespo Silva

Leiria, *Março de 2015*

Dissertação

Mestrado em Engenharia Informática – Computação Móvel

OpenCL-MMP – Codificação de Imagens com Sistemas com Múltiplos Núcleos

João Filipe Crespo Silva

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

Leiria, *Março de 2015*

À Minha Família

"O único lugar onde o sucesso vem antes do trabalho é no dicionário."

Albert Einstein

Agradecimentos

Em primeiro lugar, gostaria de agradecer ao meu orientador, Doutor Patrício Rodrigues Domingues, pela grande disponibilidade e sacrifício quase até ao último dia da entrega do presente documento. Sem esta ajuda este trabalho não teria um fim tão grandioso. Acrescento ainda, os coorientadores Doutor Nuno Miguel Morais Rodrigues e Doutor Sérgio Manuel Maciel Faria que em conjunto com o professor orientador forneceram um grande apoio, orientações e sugestões sempre construtivas ao longo do decorrer deste trabalho. Ainda um agradecimento ao Doutor Murilo de Carvalho que com o seu conhecimento do algoritmo permitiu trocas de ideias cruciais para o desenrolar deste trabalho.

Quero agradecer à Escola Superior de Tecnologia e Gestão do Instituto Politécnico de Leiria e ao Instituto de Telecomunicações – Pólo de Leiria pelas condições fornecidas para o desenvolvimento deste trabalho e a possibilidade de participar nos dois projetos no qual o âmbito deste trabalho se insere: projetos "EPIC - Efficient Pattern-matching Image Compression on many-core systems" - PTDC/EIA-EIA/122774/2010 e " OPAC - Optimization of pattern-matching compression algorithms for GPU's" - PEst-OE/EEI/LA008/2013, financiados pela Fundação para a Ciência / Tecnologia e Ministério da Educação e Ciência (FCT/MEC).

Um especial agradecimento ao colega Tiago Martins Ribeiro cuja parceria e troca de ideias ao longo da nossa participação no projeto EPIC permitiram enriquecer o presente documento com muita informação útil. Também, permitiu aumentar o meu próprio conhecimento como investigador e crescer como pessoa. Ainda ao colega Pedro Miguel Marques Pereira cuja disponibilidade foi muito importante para a fase de recolha de resultados.

Um agradecimento aos demais colegas do Mestrado de Engenharia Informática – Computação e amigos de longa data com a sua palavra amiga e incentivo.

Em último lugar mas mais importante, gostaria de agradecer aos meus pais e ao meu irmão pelo imenso apoio e incentivo dados ao longo do tempo durante esta etapa da minha vida. Sem eles não teria as condições para me dedicar com tanto tempo e força ao presente trabalho.

Resumo

Esta tese, no âmbito dos projetos "EPIC - Efficient Pattern-matching Image Compression on many-core systems" - PTDC/EIA-EIA/122774/2010 e " OPAC - Optimization of pattern-matching compression algorithms for GPU's" - PEst-OE/EEI/LA008/2013, investiga a adaptação e otimização de um algoritmo de compressão de imagens assente na metodologia de correspondência de padrões para sistemas de processamento gráfico de alto desempenho, isto é, as placas gráficas (GPU). O foco principal foi a migração do algoritmo Multiscale Multidimensional Parser (MMP) para um sistema de múlti-núcleos utilizando CUDA e OpenCL.

Esta dissertação foca particularmente a adaptação do algoritmo MMP ao paradigma OpenCL, onde foi necessário proceder a um estudo do algoritmo existente e identificar as funções que consumiam a maior percentagem de tempo de processamento e aferir a viabilidade de o transcrever para um paradigma de processamento paralelo. A principal ação detetada para efeitos de paralelização prende-se com as pesquisas que o algoritmo efetua sob um dicionário adaptativo na busca do melhor elemento que identifica determinado bloco. Esta pesquisa é efetuada durante a codificação dos blocos e na atualização do dicionário, no final da codificação de cada bloco. Para tal, foram criados quatro funções a executar pela GPU, denominados de *kernels.* Os dois primeiros *kernels* são executados durante a codificação dos blocos têm como objetivo devolver o melhor elemento do dicionário para cada partição do bloco. O terceiro e quarto *kernels* são responsáveis pela atualização do dicionário: um efetua o controlo de redundância dos elementos do dicionário e o outro fica responsável por atualizar o dicionário presente em memória da GPU.

Os dois protótipos implementados, CUDA-MMP e OpenCL-MMP, permitiram ganhos de tempo na ordem de 2 a 10 vezes, mantendo a qualidade de compressão original. Estes resultados permitem concluir que os paradigmas de multi-núcleo permitem acelerar o desempenho das aplicações. Contudo, este desempenho apenas é conseguido com um estudo iterativo dos *kernels* e otimizando-os de forma a garantir o máximo de desempenho por parte da GPU.

Palavras-chave: OpenCL, CUDA, computação de alto desempenho, GPU, múlti-núcleos, speedup

Abstract

This thesis, developed under the scope of the projects "EPIC - Efficient Pattern-matching Image Compression on many-core systems" - PTDC/EIA-EIA/122774/2010 and " OPAC - Optimization of pattern-matching compression algorithms for GPU's" - PEst-OE/EEI/LA008/2013, investigates the adaptation and optimization of a high performance recurrent pattern matching-based compression algorithms in many-core platforms, namely in Graphic Processing Units (GPUs). The main target was the migration of a Multiscale Multidimensional Parser (MMP) to a many-core system in CUDA and OpenCL, with a particular focus on the adaptation of the MMP algorithm to an OpenCL paradigm.

A detailed analysis of the algorithm was made in order to identify the functions with the highest percentage of processing time and ensure the viability of the migration to a parallel processing paradigm. The exhaustive search performed by the algorithm, in an adaptive dictionary to get the best element that identifies a particular block, was the key part identified to take advantage of the parallel processing. This search is performed during the encoding of the blocks and the update of the dictionary at the end of encoding each block. In order to reach this goal, four functions named kernels, were created to be performed by the GPU. The first two kernels execute the encoding of the blocks and returns the best dictionary element for each block partition. The third and fourth kernels are responsible for the dictionary update: one performs the redundancy control of the dictionary elements and the other one is responsible for the update of the GPU's dictionary.

The two prototypes implemented, CUDA-MMP and OpenCL-MMP, obtained a speedup in the range of 4 to 10 times, while maintaining the original compression quality. These results show that the many-core paradigms can increase the performance of some applications. However, this performance is only achieved with an iterative study of the kernels and optimizing them in order to ensure the maximum performance from the GPU.

Key-Words: OpenCL, CUDA, high computing, GPU, many-core, speedup

Índice de Figuras

Índice de Quadros

Lista de Acrónimos

Acrónimo	Descrição
AMD	Advanced Micro Devices, Inc.
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GP-GPU	General Purpose of Graphics Processing Unit
GPU	Graphics Processing Unit
LSP	Least-Square Prediction
MMP	Multidimension Multiscale Parser
NVVP	Nvidia Visual Profiler
OpenCL	Open Computing Language
PSNR	Peak Signal-to-Noise Ratio
SIMD	Single Instruction, Multiple Data
SM	Streaming Multiprocessor
SGPR	Scalar General-Purpose Register
VGPR	Vector General-Purpose Register

Índice

1. Introdução

1.1 Motivação

Atualmente vivemos uma era digital onde os computadores e internet conquistaram uma grande parte da população mundial. Aliado a esta evolução, existe cada menos a utilização do papel como forma de apresentação e transporte da informação. Deste modo, nasceu o conceito de informação digital. A informação digital tem a vantagem de poder ser representada em diversos modos, seja por texto, imagem, som ou qualquer outro suporte multimédia. Outras vantagens prendem-se com a acessibilidade e armazenamento. Ou seja, pode ser acedida em qualquer lugar quer esteja armazenado no dispositivo em utilização ou alojada na *cloud.*

Com a evolução exponencial dos dispositivos de captura/gravação de som, imagem ou vídeo, a informação multimédia encontra-se limitada pela capacidade de armazenamento e de transporte dos dispositivos. Atualmente, estes dispositivos permitem a captura com uma qualidade de detalhe muito grande, isto é, qualidade Full-HD (1080p) ou, mais recentemente, 4K que em termos de armazenamento ocupam muito espaço. Além disso, tornou-se hábito comum a partilha desta informação seja em redes sociais, seja de forma direta com o remetente, ficando a velocidade de transmissão dos dados condicionada à largura de banda disponível.

Para colmatar estes problemas, surgiu a necessidade de comprimir a informação de modo a que na mesma capacidade de armazenamento seja possível guardar mais informação ou para que a transmissão de dados seja efetuada com maior rapidez. Deste modo, foram implementados vários compressores de dados. Estes algoritmos de compressão de dados podem fornecer uma compressão com ou sem perda de dados. A perda de dados assenta no paradigma de perda de algum detalhe da informação sem que esta perca o seu significado. Esta renúncia de detalhe permite que se obtenha melhores rácios de compressão, principal objetivo dos algoritmos de compressão. Exemplo de algoritmos de compressão existe o JPEG2000 [1], Portable Network Graphics (PNG) [2], H.264 [3] e H.265 [4].

O Multidimensional Multiscale Parser (MMP) [5, 6] é um codificador com ou sem perdas (suporta ambos os modos) que utiliza um esquema de aproximação de padrões na codificação dos dados originais. O MMP, aplicado à codificação de imagens, apresenta resultados cuja qualidade é superior aos algoritmos estandardizados, à exceção do algoritmo H.265. A maior desvantagem do MMP deve-se à sua elevada complexidade computacional que provoca/requer elevados tempos de computação. Por exemplo, para a codificação de uma imagem com dimensões de 512x512 pixéis, a versão sequencial do MMP demora várias horas na codificação da imagem enquanto os algoritmos opositores desempenha o seu processo em alguns segundos.

Uma forma de conseguir diminuir o tempo de computação de uma aplicação assenta no uso de computação paralela. Este modelo de computação visa a realização de várias ações em simultâneo. Inicialmente, esta forma de computação era aplicada em computação de alto desempenho com o uso de supercomputadores ou *clusters* de vários computadores. Com o abandono da corrida para se obter a maior frequência de relógio e a construção dos dispositivos com múltiplos núcleos, devido a questões de eficiência energética e arrefecimento [7], a computação paralela ganhou ainda maior importância.

Cada núcleo pode executar uma diferente operação obtendo-se assim maior uma maior capacidade de processamento.

As GPUs (*Graphic Processing Unit*) revolucionaram a computação de alto desempenho com a disponibilização de placas que alojam milhares de núcleos. A Unidade de Processamento Gráfico de Propósito Geral (GP-GPU) permitiu a implementação de aplicações, anteriormente executadas pela CPU, fossem executadas pela GPU. Com o uso das milhares de unidades de processamento das GPU torna-se possível executar várias ações em simultâneo e desta forma acelerar significativamente o processamento de aplicações complexas.

Para ajudar os programadores na criação de aplicações orientadas para a exploração do paralelismo por parte da GPU, foram criadas várias plataformas de computação (*frameworks*), com destaque para as plataformas CUDA [8] e OpenCL [9]. A plataforma CUDA é proprietária da marca NVIDIA, estando somente disponível em placas da marca NVIDIA. Por sua vez, a plataforma OpenCL visa suportar qualquer dispositivo com múltiplos núcleos (CPU, GPU, etc).

O trabalho apresentado no presente documento insere-se no âmbito dos projetos: "EPIC - Efficient Pattern-matching Image Compression on many-core systems" - PTDC/EIA-EIA/122774/2010 , financiado pela FCT/PTDC e realizado no Instituto de Telecomunicações – Polo de Leiria na Escola Superior de Tecnologia e Gestão do Instituto Politécnico de Leiria; e "OPAC - Optimization of pattern-matching compression algorithms for GPU's" financiado por fundos nacionais (PIDDAC) através da FCT/MEC e realizado no Instituto de Telecomunicações – Polo de Leiria na Escola Superior de Tecnologia e Gestão do Instituto Politécnico de Leiria.

1.2 Objetivos

O objetivo global dos projetos EPIC e OPAC centra-se na implementação de duas aplicações protótipo do algoritmo MMP com recurso à computação de alto desempenho. Especificamente, um protótipo é orientado para a plataforma CUDA (CUDA-MMP), ao passo que o outro foca a plataforma OpenCL (OpenCL-MMP).

Como mencionado anteriormente, o MMP demonstra resultados no rácio de compressão e qualidade acima dos algoritmos estandardizados. Contudo, o MMP demora várias horas na codificação de uma imagem o que torna difícil a sua aplicação a nível comercial. Com o estudo da computação de alto desempenho e o uso de placas gráficas para o processamento do algoritmo pretende-se acelerar o desempenho do MMP. No final, pretende-se que este consiga apresentar tempos de execução próximos de outros algoritmos de compressão de dados.

Os objetivos do presente trabalho resumem-se nos seguintes tópicos:

- Estudo do algoritmo MMP de modo a identificar os principais *bottlenecks* da execução do algoritmo e analisar a viabilidade da sua implementação com recurso à computação paralela;
- Implementação das aplicações protótipo CUDA-MMP e OpenCL-MMP sem perda do rácio de compressão e qualidade;
- Otimização das duas aplicações multi-núcleo de modo a tirar o maior proveito do *hardware*, ou seja, das características da GPU;
- Comparação do desempenho entre a aplicação CUDA e OpenCL;
- Comparação do desempenho da aplicação OpenCL em GPUs das marcas NVIDIA e AMD.

1.3 Contributos

Como resultado deste trabalho foi conseguida uma redução bastante significativa do tempo de execução do algoritmo MMP usando um paradigma múlti-núcleos. Apesar do objetivo principal deste trabalho se focar sobre o OpenCL e as otimizações a efetuar para melhor o desempenho de uma aplicação sob esta plataforma, existiu também um estudo da plataforma CUDA. Pois a versão sequencial foi primeiramente adaptada ao CUDA e depois portada para OpenCL. Deste modo, neste trabalho encontram-se descritas otimizações a efetuar em aplicações implementadas em CUDA e OpenCL.

Do trabalho resultante do desenvolvimento da aplicação CUDA, resultou o seguinte artigo científico, que focou as otimizações de memória em CUDA:

"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).

1.4 Organização do documento

O Capítulo 2 apresenta as duas plataformas mais usadas em computação de alto desempenho com GPUs, isto é, o CUDA e o OpenCL. São ainda descritas duas APIs de programação paralela com o uso de diretrizes.

No Capítulo 3 é feita a explicação do algoritmo em estudo, o MMP, começando na versão original do algoritmo até às versões mais avançadas, nomeadamente as otimizações "predição *intra" e "segmentação flexível".* Estas otimizações são descritas pois são relevantes para o decorrer deste trabalho. É ainda mostrado os resultados obtidos por este algoritmo ao nível da qualidade de compressão e desempenho do mesmo.

No Capítulo 4 é descrito o ambiente de testes utilizado para recolha de resultados e trabalho. A descrição engloba o *hardware* dos servidores, versões do *software* e *drivers* em uso, bem como as imagens de teste.

No Capítulo 5 inicia-se o estudo da aplicação sequencial onde foram identificados os principais pontos que consumiam mais tempo e potenciais pontos de otimização. É ainda descrita o protótipo CUDA-MMP que existia aquando do início desta dissertação. Seguidamente são mostradas as otimizações efetuadas sob o protótipo CUDA-MMP de modo a melhorar o seu desempenho. O capítulo termina com a apresentação de algumas versões sub-ótimas que conseguem maiores velocidades de execução sacrificando a qualidade de compressão.

No Capítulo 6 são apresentadas ferramentas de conversão de código em CUDA para OpenCL. De seguida, descreve-se a migração da versão CUDA para uma versão OpenCL e os seus resultados. São ainda descritas algumas otimizações efetuadas no âmbito da versão OpenCL de modo a aumentar o desempenho da mesma.

No Capítulo 7 é feita a apresentação dos resultados obtidos e a sua comparação entre plataformas de programação e *hardware.*

Por último, no Capítulo 8, são apresentadas algumas conclusões relativamente ao trabalho desenvolvido e sendo ainda sugerido o trabalho futuro.

2. Estado da Arte

Neste capítulo é feito um estudo sobre os modelos de programação existentes para a programação paralela para dispositivos múlti-núcleos. Primeiro são explicados os modelos OpenCL, foco desta dissertação, e CUDA que são os mais utilizados. São ainda abordados outros modelos que permitem a computação paralela.

2.1 OpenCL

A empresa Apple Inc. foi a grande impulsionadora da criação do padrão aberto Open Computing Language (OpenCL). A motivação da Apple era procurar uniformizar a programação dos vários tipos de dispositivos de processamento, tais como processadores múlti-núcleos, placas gráficas e outros tipos de dispositivos de aceleração de processamento. De facto, muitos desses dispositivos têm uma interface de programação própria, o que reduz ou elimina mesmo a portabilidade do código. Assim, com a iniciativa OpenCL, a Apple procurava que uma mesma interface de programação, definida como uma extensão à linguagem C, pudesse ser empregue para a extração do desempenho nos mais variados tipos de dispositivos. A versão OpenCL 1.0 foi divulgada com o lançamento do Mac OS X Snow Leopard em 9 de junho de 2008 [10].

Pouco depois, no dia 16 de junho de 2008, The Khronos™ Group, grupo responsável pela criação de vários padrões abertos importantes na área da informática como por exemplo OpenGL [11], anunciou a formação de um novo grupo de trabalho para a criação de uma padrão aberto para a programação heterogénea e computação paralela utilizando GPUs e CPUs. Este grupo de trabalho agregou um vasto número de empresas de diversas áreas como a 3Dlabs, AMD, Apple, ARM, Codeplay, Ericsson, Freescale, Graphic Remedy, IBM, Imagination Technologies, Intel, Nokia, NVIDIA, Motorola, QNX, Qualcomm, Samsung, Seaweed, TI, e Umeå University [12]. Posteriormente, as versões OpenCL 1.1 em 14 de junho de 2010 [13], 1.2 em 15 de novembro de 2011 [14], 2.0 em 18 de novembro de 2013 [15], e, mais recentemente, a versão 2.1 em 29 de janeiro de 2015 [16] foram já especificadas pelo The Khronos™ Group.

2.1.1 Objetivos

OpenCL é um padrão aberto para a programação paralela que tem como principal objetivo e premissa de “write once, run anywhere”, ou seja, um programa em OpenCL fica preparado para correr em qualquer dispositivo que suporte OpenCL, seja este uma GPU ou um CPU (Figura 2.1). Nas GPU este modelo faz uso da metodologia GPGPU (*general purpose of computing on graphics processing units*) para a execução das tarefas nestas unidades.

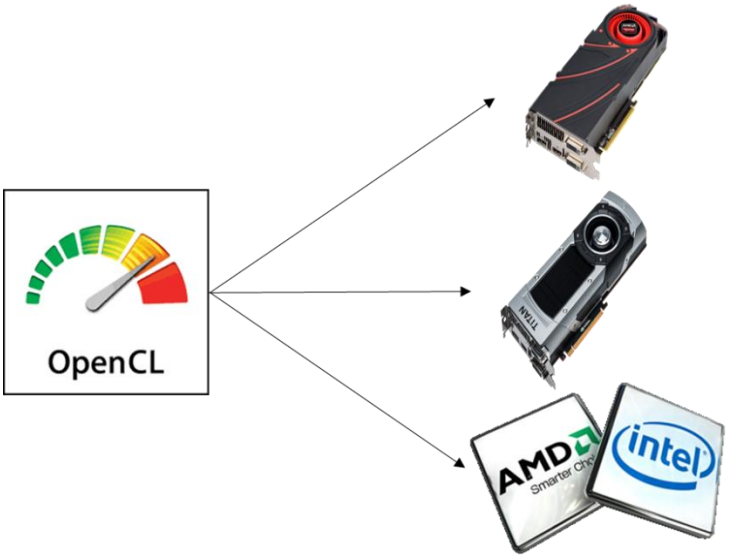

Figura 2.1 - Dispositivos suportados por OpenCL

2.1.2 Arquitetura

A arquitetura OpenCL é composta por um sistema hospedeiro (*host)* e um ou mais dispositivos (*devices*) que estão instalados no sistema hospedeiro *(*Figura 2.2*)*. O *host* é responsável por identificar os vários dispositivos e escolher quais os que pretende utilizar durante a execução da aplicação. Um *device* pode ser uma placa gráfica (GPU) ou um processador (CPU) desde que suporte OpenCL, usualmente sob a forma de software específico, designado de *drivers*. Dentro de cada *device* existem vários *compute unit* que são responsáveis pela realização do trabalho enviado pelo *host*, ou seja, o *kernel*. Por exemplo, num programa que faça a multiplicação de dois vetores e guarde os resultados num terceiro vetor, o *host* (computador) é responsável por selecionar um ou mais *devices* (CPUs ou GPUs) que irão executar o *kernel* que efetua a multiplicação dos vetores (Figura 2.2).

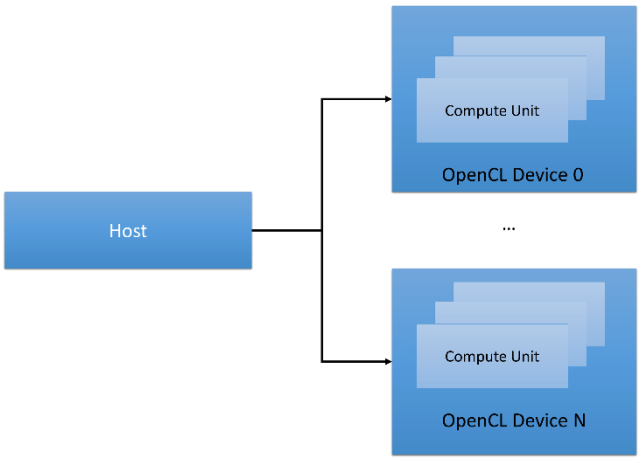

Figura 2.2 - Arquitetura OpenCL (Adaptado de [17])

2.1.3 Kernel

O *kernel*, que corresponde à função a executar pelo *device*, é escrito na linguagem OpenCL C que é baseada na linguagem C 99 com algumas modificações e acréscimos para comportar as arquiteturas heterogêneas. De entre as alterações existem:

- Identificadores para os diferentes tipos de memória (secção 2.1.4).
- Bibliotecas nativas de funções:
    - Controlo de fluxo, estas funções permitem identificar a unidade que está a executar e de sincronismo (secção 2.1.5);
    - Aritmética e trigonométricas, etc. De destacar que algumas dessas funções matemáticas têm uma dupla implementação: uma que se comporta de acordo com as normas próprias (usualmente a norma IEEE 754 [18]) e uma outra implementação orientada para o desempenho. Essa segunda implementação troca velocidade de execução por precisão, executando pois mais rapidamente mas com menor precisão aritmética.
    - Lógica binária
    - Atómicas, usadas para evitar *race conditions* a um local de memória.
- Inclusão dos tipos vetoriais. Notação do tipo Tn, em que T corresponde ao tipo de variável (char, int, float, etc) e n identifica o número de elementos (2, 3, 4, 8, 12).

Um *kernel* é chamado/ativado pelo sistema hospedeiro para ser executado num dispositivo com suporte a OpenCL, através de uma determinada configuração/geometria de execução. Esta geometria de execução define o número de blocos de execução e o número de entidades ativas por bloco de execução. Os blocos de execução são denominados por *work-group* e as entidades por *work-item* ou mais genericamente por *threads* no seio da comunidade *many-cores*. O modelo de execução em OpenCL é detalhado na secção 2.1.5.

Usando como exemplo uma função sequencial (`vectorMul`) que calcula o vetor C como sendo o produto do vetor A por B:

```
1.  void vectorMul (float* a, float* b, float* c, int n)
2.  {
3.       int i;
4.       for (i=0; i<n; i++)
5.            c[i] = a[i] * b[i];
6.  }
```

A mesma função implementada em OpenCL C, origina o *kernel* com o código seguinte:

```
1.  __kernel vectorMul (global float* a, global float* b, global float*c, int n)
2.  {
3.       int id = get_global_id(0);
4.       if (id < n)
5.            c[id] = a[id] * b[id];
6.  }
```

A versão sequencial recorre a uma estrutura de repetição, iterando por cada elemento i dos vetores A, B e C. Na versão OpenCL, a multiplicação é distribuída pelas entidades ativas, ficando cada entidade (identificada pelo valor de retorno da função OpenCL *get_global_id(0))* encarregue de calcular um elemento do vetor C. Desta forma, o cálculo de todos os elementos do vetor C é feito em paralelo.

2.1.4 Modelo de Memória

O modelo de memória consiste em quatro tipos de memória: memória global, memória das constantes, memória local e memória privada (Figura 2.3).

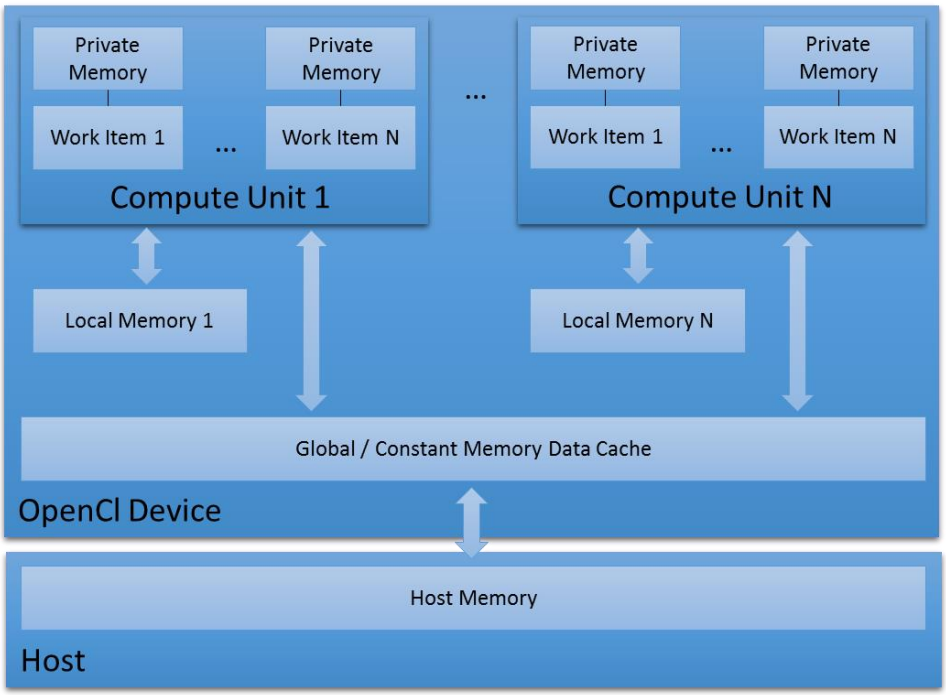

Figura 2.3 - Modelo de Memória (Adaptado de [17])

Cada tipo de memória tem um identificador próprio em OpenCL C (Tabela 2.1).

Memória	Identificador
Global	__global
Constante	__constant
Local	__local
Privada	__private (opcional)

Tabela 2.1 - Correspondência entre tipo de memória e o seu identificador OpenCL C

2.1.4.1 Hierarquia de memória

A memória global é a que disponibiliza maior espaço mas que apresenta menor velocidade de leitura e escrita. Esta memória é utilizada para guardar os dados que serão depois processados. Por exemplo, a especificação da GPU AMD R9 290X indica que tem 3 GiB de memória na placa. Para efeitos do OpenCL, é essa memória que é designada por memória Global, sendo identificada no código OpenCL através da palavra-chave "*__global*".

A memória das constantes é uma memória apenas de leitura relativamente ao dispositivo. O *host* escreve os dados e o *kernel* apenas consegue efetuar leituras sob esta memória. Tipicamente, este espaço de memória é de pequena dimensão (64 KiB).

A memória dita local é uma memória acessível a todos os elementos ativos (*work-items*) que pertençam a um mesmo grupo (*work-group*) (secção 2.1.5). A memória local a ser empregue por um *kernel* pode ser alocada de duas formas: estaticamente ou dinamicamente. A alocação estática é definida no código fonte do *kernel*.

Exemplo:

```
1.  __kernel void my_kernel (…)
2.  {
3.      __local int my_local_memory[1024];
4.  }
```

Por sua vez, a alocação dinâmica requer o acrescentar de um parâmetro de entrada ao kernel, parâmetro marcado com o identificador "*__constant*".

```
1.  __kernel void my_kernel ( __local int* my_local_memory, …)
2.  {
3.      …
4.  }
```

O tamanho de memória deste parâmetro é definido no *host* através do código que define os argumentos do *kernel*.

```
1.  clSetKernelArg(my_kernel, 0, 1024*sizeof(int), NULL);
```

Finalmente, a memória privada apenas está acessível ao work-item que o alocou. O uso do seu identificador (*__private*) é opcional pois o compilador trata a omissão de identificador como sendo de memória privada.

A velocidade de acesso a cada tipo de memória por parte de cada *work-item* depende da distância a que esta memória se encontra em relação ao elemento ativo e dos elementos que lhe possam aceder. Desta forma, a memória mais rápida será a memória privada do *work-item* em que apenas ele tem acesso. De seguida a memória local que é acessível por todos os *work-items* dentro do mesmo *work-group.* A próxima memória é a memória das constantes que por ser uma memória apenas de leitura existem mecanismo de *cache* próprios que aceleram o acesso a esta memória. Por fim, a memória global é a mais lenta mas em contrapartida é a de maior dimensão permitindo guardar mais dados.

2.1.5 Modelo de Execução

O modelo de execução (Figura 2.4) consiste numa grelha de grupos de trabalho denominada *NDRange*. Os grupos de trabalho em OpenCL são denominados *work-group* e estão dispostos numa grelha multidimensional que pode ter até 3 dimensões (x,y,z). Dentro de cada *work-group* existem os itens de trabalho que são denominados por *work-items*. Os work-items, também, estão dispostos numa grelha multidimensional que comporta até 3 dimensões (x,y,z).

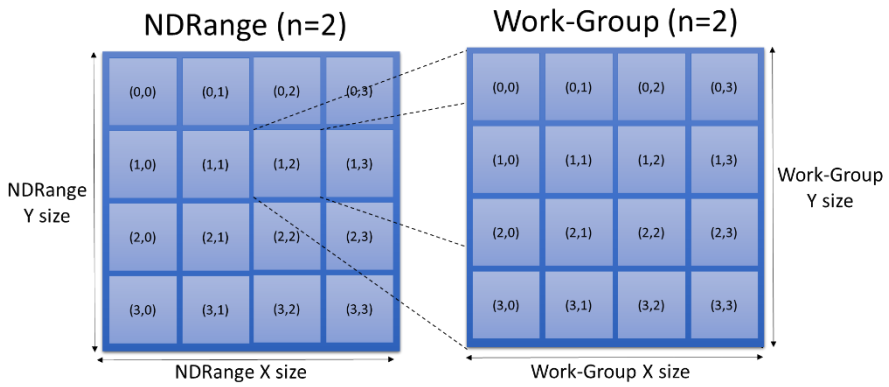

Figura 2.4– Exemplo de modelo de execução com NDRange de dimensão 2 e work-group de dimensão 2 (Adaptado de [17])

Um *work-item* tem acesso a funções da API que lhe permite determinar a sua localização (Figura 2.5) dentro do seu *work-group* (*local* ID), identificar o *work-group* em que está inserido (*work-group* ID) e qual a sua localização global (*global* ID), isto é, qual o seu identificador (ID) de entre todos os *work-items* existentes.

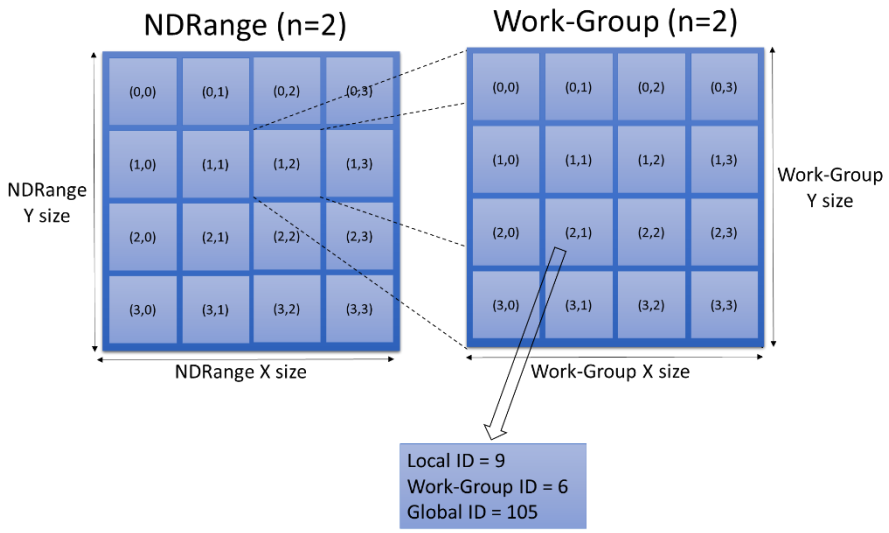

Figura 2.5 - Localização de um work-item (Adaptado de [17])

Em termos de sincronismo, apenas, é possível sincronizar os *work-items* que estejam contidos no mesmo *work-group*. Este sincronismo é efetuado com o uso da função *barrier* que recebe como parâmetro uma flag a indicar qual o tipo de memória que o sincronismo está associado. Se o sincronismo estiver associado à memória partilhada deve-se utilizar a flag CLK_LOCAL_MEM_FENCE. Caso o sincronismo esteja associado à memória global é empregue a flag CLK_GLOBAL_MEM_FENCE.

2.1.6 Aplicação em OpenCL

Uma aplicação OpenCL divide-se em quatro principais etapas: (1) a inicialização, (2) a preparação da memória, (3) a execução do *kernel* e, por fim, (4) a libertação de recursos (Tabela 2.2).

Estado	*Host*	*Device*
Inicialização do OpenCL	Descobrir e escolher os *devices*	
	Criar contexto	
	Criar lista de tarefas (uma por *device)*	
	Carregar e compilar os *kernels*	
	Instanciar os *kernels*	
Preparação da memória	Alocação dos *buffers* de memória	
Execução do *kernel*	Cópia dos dados a ser tratados pelo *kernel*	
	Carregamento dos argumentos do *kernel*	
	Lançamento do *kernel*	
	Execução do *kernel*	
	Sincronismo	
	Cópia dos resultados do *kernel*	
	Validar/utilizar os resultados	
Libertação de memória	Libertação da memória dos *buffers*	
	Libertação dos objetos OpenCL	

Tabela 2.2 Sequência de uma aplicação em OpenCL

Inicialização

A inicialização consiste em descobrir, escolher e configurar os dispositivos OpenCL que se pretende utilizar para a realização de trabalho.

Após a escolha dos dispositivos, é criado um contexto que é usado para identificar os objetos OpenCL. Um contexto pode conter um ou mais *devices*. O passo seguinte consiste em criar uma lista de tarefas para cada *device*, i.e. uma lista do trabalho que o *host* pretende que o *device* realize. As tarefas incluem operações de cópias de memória e/ou execuções de *kernels*.

O último passo da inicialização é o carregamento do programa propriamente dito. O programa pode ser um ficheiro binário ou código fonte OpenCL. Quando se trata de código fonte, i.e. no formato de texto, este pode estar definido de forma estática no código através de uma *string* ou lido de um ficheiro de texto externo, durante a execução.

Formato de texto estático:

```
1.  constant char* kernel_vectorMul_str =
2.  "__kernel vectorMul(global float* a, global float* b, global float*c, int n)"\
3.  "{"\
4.  "       int id = get_global_id(0);"\
5.  "       if (id < n)"\
6.  "               c[id] = a[id] * b[id];"\
7.  "}";
```

Formato de texto em ficheiro:

```
1.  __kernel vectorMul (global float* a, global float* b, global float*c, int n)
2.  {
3.       int id = get_global_id(0);
4.       if (id < n)
5.            c[id] = a[id] * b[id];
6.  }
```

A forma estática, usualmente, é usada para pequenos exemplos ou pequenos *kernels* visto que esteticamente é de difícil leitura e manipulação.

Quando se trata de um ficheiro binário, este já foi compilado anteriormente pelo que é efetuado apenas o carregamento dos *kernels.* Quando o código se apresenta em formato de código fonte, a compilação é efetuada em *runtime*. A compilação em *runtime* tem a vantagem de ser feita uma compilação de acordo com o compilador OpenCL instalado no dispositivo hospedeiro que estará de acordo e atualizado para o dispositivo em utilização. Outra vantagem prende-se com a portabilidade do código que não obriga a nova compilação de toda a aplicação para diferentes dispositivos ou diferentes versões de OpenCL. A desvantagem prende-se com o tempo consumido para a compilação do programa que é necessário proceder-se em cada execução da aplicação.

Caso existam erros de compilação a aplicação deve terminar. Após uma compilação sem erros, procede-se à instanciação de cada *kernel* existente no programa.

Preparação da memória

A preparação da memória consiste em reservar blocos de memória (*buffers* na terminologia OpenCL) na memória do dispositivo. Os blocos de memória podem ser de leitura e/ou escrita por parte dos *work-items* do *device.* Um bloco de memória definido para leitura apenas pode ser lido pelos elementos ativos do dispositivo (*work-items*), sendo a memória escrita pelo sistema hospedeiro. Os blocos de memória para leitura estão usualmente associados à memória das constantes.

Execução do kernel

A fase de execução do *kernel* abrange as operações de cópia dos dados a ser tratados pelo *kernel,* o carregar dos argumentos do *kernel*, a execução do *kernel*, o sincronismo entre o sistema hospedeiro e o dispositivo para saber se o dispositivo já terminou a execução do *kernel*, e por fim a cópia dos resultados do *kernel* para o *host*. A cópia dos resultados também pode ser utilizada como mecanismo de sincronismo caso o parâmetro booleano de bloqueio esteja ativo.

Libertação de recursos

O último passo consiste em libertar a memória alocada durante a execução da aplicação, ou seja, a memória dos *buffers* e dos objetos OpenCL.

2.2 CUDA

O CUDA é uma plataforma de computação paralela somente disponível para GPUs da marca NVIDIA [19]. Em termos de programação, e à semelhança do OpenCL, o CUDA assenta numa extensão à linguagem C. Especificamente, a interface de programação da plataforma CUDA acrescenta vários identificadores, tipos de dados e funções à versão 99 da linguagem C (C99). O modelo de execução é

similar ao seguido pelo OpenCL, sendo conveniente relembrar que o CUDA, cuja 1ª versão foi lançada em 2006, foi a plataforma pioneira para a programação de forma genérica de GPUs, precedendo a própria norma OpenCL.

2.2.1 Principais diferenças entre CUDA e OpenCL

Os modelos de arquitetura (Tabela 2.3), memória (Tabela 2.4) e de execução (Tabela 2.5) em CUDA e OpenCL são muito semelhantes, divergindo somente na nomenclatura de algumas funcionalidades.

A arquitetura em CUDA é composta pelo sistema hospedeiro (*host*), isto é, a CPU, e por uma ou mais dispositivos *(devices)*, todos eles da marca NVIDIA que suportem CUDA. A diferença em relação ao OpenCL é que este último pode ser utilizado com GPUs de várias marcas e outros dispositivos tais como CPUs múlti-núcleos, desde que os dispositivos disponibilizem os apropriados drivers para suportar a norma OpenCL. O CUDA está limitado a dispositivos GPU da marca NVIDIA.

CUDA	OpenCL
GPU	Device
Streaming Multiprocessor (SM)	Compute Unit (CU)

Tabela 2.3 - Diferenças de CUDA e OpenCL na arquitetura

No modelo de memória, tanto em CUDA como em OpenCL existem quatro tipos de memória. A memória global de grande dimensão e de leitura e escrita, memória das constantes que é apenas de leitura, memória partilhada (local em OpenCL) que pode ser partilhada entre as threads de um mesmo bloco e, por fim, os registos (correspondem à memória privada em OpenCL). Importa notar que os registos constituem a memória de acesso mais rápido.

CUDA	OpenCL
Global Memory	Global Memory
Constant Memory	Constant Memory
Shared Memory	Local Memory
Registers	Private Memory
Textures	Image

Tabela 2.4 – Designação dos vários componentes do modelo de memória em CUDA e OpenCL

Na nomenclatura definida pelo CUDA, a entidade ativa é designada de *thread*. Uma *thread* em CUDA corresponde a um *work-item* e executa o código indicado pelo kernel. O CUDA, à semelhança do OpenCL, executa as threads em bloco, agregando 32 threads no que é designado como um *wrap*. Assim, o escalonador do CUDA procura executar 32 threads simultaneamente, usando para o efeito um multiprocessador. As threads de um wrap são executadas de forma síncrona, cada thread executa a mesma instrução de código sobre um conjunto de dados próprios da thread. Obviamente, tal deixa de suceder quando o código a executar apresente uma estrutura de controlo que leva à execução divergente. Por exemplo, se existir uma estrutura de controlo *if* que indica que as threads com identificador (*threadID*) par executam a condição, ao passo que as restantes (identificadores impares)

executam o código definido no bloco do respetivo *else*, a execução passa a ser feita em dois tempos. Assim, primeiro é executado um bloco de código (por exemplo, o bloco associado ao *if*), e só após o bloco seguinte (neste caso, o bloco ligado ao *else*). Este problema é denominado por *branch divergence*.

O CUDA disponibiliza dois modos de programação: *runtime* e *driver*. Como é mostrado na secção seguinte, o modelo *runtime* é um modelo de alto nível que simplifica significativamente o uso da plataforma CUDA, tratando de forma automática os pormenores de baixo nível. Por exemplo, a chamada a um kernel processa-se através da indicação do nome do kernel seguido do operador <<<...>>>. Através desse operador, o programador indica a grelha de execução, ou seja, dimensionamento dos blocos de execução e dimensionamento das threads dentro de cada bloco de execução e ainda os eventuais parâmetros a serem passados para o kernel. Opcionalmente, o programador pode ainda indicar como 3º elemento do operador <<<...>>>, o tamanho de memória partilhada que será alocada dinamicamente para uso no *kernel* e como 4º elemento qual a *stream* da GPU que o *kernel* irá executar.

Exemplo:

```
1.  my_cuda_kernel<<64, 256, 0, my_stream>>>(arg1, arg2,...);
```

Assim, para o programador que faz uso do modo *runtime*, o lançamento de um *kernel* CUDA assemelha-se muito a uma chamada a uma função, contrastando com a configuração laboriosa necessária tanto no OpenCL como no modo *driver* do CUDA [20].

CUDA	OpenCL
Thread-Block	Work-Group
Thread	Work-Item

Tabela 2.5 - Diferenças de CUDA e OpenCL no modelo de execução

2.2.2 CUDA Runtime API, CUDA Driver API e OpenCL API

Conforme referido anteriormente, a programação em CUDA do código a executar pelo *host* oferece duas possibilidades: *CUDA Runtime API* de alto nível que é a mais utilizada e *CUDA Driver API* que é de baixo nível. *CUDA Runtime API* permite ao programador abstrair a inicialização e gestão do *device*, no entanto existem funções para a gestão do mesmo [21].

A interface de programação *CUDA Driver API* requere uma maior quantidade de código e necessita que o código do *kernel* seja carregado a partir de um ficheiro binário. Torna-se mais difícil para o programador mas contudo oferece um maior nível de controlo. As interfaces de programação *OpenCL API* e CUDA Driver API são de baixo nível mas têm algumas diferenças sendo que CUDA Driver API oferece ainda mais abstração do que OpenCL API (Tabela 2.6) [22].

API OpenCL	*API CUDA Driver*	*API CUDA Runtime*
Inicialização		
Procura dos *devices* Escolha dos *devices* Criação do contexto Criação de listas de tarefas	Inicialização do *driver* Procura dos *devices* (Escolha do *device*) Criação do contexto	Escolha do *device* (opcional, por omissão é o device 0)
Memória		
Alocação memória *Host* Alocação memória *Device* Copia dos dados	Alocação memória *Host* Alocação memória *Device* Copia dos dados	Alocação memória *Host* Alocação memória *Device* Copia dos dados
	Inicialização dos *Kernels*	
Carregamento da *source* Criação do Programa Compilação do Programa Instância dos *kernels*	Carregamento do módulo dos *kernels* (Compilação do Programa) Obter os *kernels*	-
Execução do *Kernel*		
Set dos argumentos Configuração do *NDRange* Invocação	*Set* dos argumentos Configuração da *grid* Invocação	Invocação do Kernel (configuração da *grid e set dos argumentos*)
	Libertação de Memória	
Libertação dos objetos de memória, *kernels*, programa, lista de tarefas, contexto	Libertação dos objetos de memória e contexto	Libertação dos objetos de memória

Tabela 2.6 - Passos de execução usando OpenCL API, CUDA Driver API e CUDA Runtime API

Uma das vantagens da utilização de uma API de baixo nível encontra-se na inicialização dos *kernels*. Visto estes serem compilados em *runtime*, permite que o programador faça alterações à *source* dos *kernels* sem necessidade da compilação de toda a aplicação. A desvantagem encontra-se no facto de ser necessário o carregamento da *source* e a sua compilação, tornando a execução da aplicação mais lenta.

2.3 Outras APIs de programação paralela

2.3.1 OpenMP

O OpenMP [23] é uma API de programação de alto-nível que permite um paralelismo de execução em processadores (multi-cores). O programador identifica as regiões de código que pretende paralelizar, normalmente ciclos, e adiciona uma diretriz do compilador *"pragma"* que para o caso específico do OpenMP será "#pragma omp". Esta diretriz permitirá ao compilador gerar o código necessário para o paralelismo entre os processadores. Assim, a criação, gestão, sincronização e término de *threads* é efetuada através de código gerado pelo compilador. Para utilizar o OpenMP para um ciclo de repetição *for* usa-se a diretriz "*#pragma omp parallel for*". A API permite identificar cláusulas à execução, por exemplo, quais as variáveis que são privadas ou partilhadas entre as *threads*, número de *threads* a utilizar, entre outras [24]. A API OpenMP encontra-se implementada em vários compiladores

comerciais como o *gcc*, com a *flag –openmp*, no compilador da IBM XL C/C++/Fortran, no compilador da Oracle C/C++/Fortran com a *flag –xopenmp*, no compilador da Intel C/C++/Fortran com a *flag –Qopenmp* em Windows e a *flag –openmp* em Linux ou Mac OSX, entre outros.

Usando a função sequencial anterior de multiplicação de vetores, adicionamos a diretriz corresponde à paralelização de um ciclo de repetição *for* (linha 4 do trecho de código seguinte) e o compilador irá gerar o código de paralelismo.

```
1.  void vectorMul (float* a, float* b, float* c, int n)
2.  {
3.      int i;
4.      #pragma omp parallel for
5.      for (i=0; i<n; i++)
6.          c[i] = a[i] * b[i];
7.  }
```

2.3.2 OpenACC

O OpenACC [25] é uma API que como o OpenMP faz uso de diretrizes do compilador. A diferença para o OpenMP é que esta API permite gerar código para executar em GPUs. A API trata da alocação da memória, cópia dos dados, geração do *kernel* e a sua chamada e por fim a cópia dos resultados. O programador deve inserir a diretriz *"#pragma acc parallel*". Apesar de a API incluir o prefixo Open, esta API não é gratuita mas disponibiliza uma licença *trial* de trinta dias de modo a testar as funcionalidades da API [26].

3. Algoritmo MMP

Este capítulo foca a implementação do algoritmo MMP desde a sua criação até à versão usada como referência para este trabalho. Apenas as versões mais relevantes são expostas pois tiveram o maior impacto na complexidade do algoritmo e servem de contexto para as otimizações que serão explicadas nos capítulos seguintes.

3.1 Introdução

O Multidimensional Multiscale Parser (MMP) é um algoritmo de compressão genérico, ou seja, pode ser aplicado a sinais de diversas categorias desde voz [27], imagem [28, 29] ou vídeo [30, 31]. Trata-se de um algoritmo de compressão apto a ser utilizado em compressão com ou sem perdas. No contexto desta dissertação, o MMP é focado na perspetiva da codificação de imagens com perdas.

O MMP baseia-se num algoritmo de correspondência aproximada de blocos da imagem a codificar com padrões de diferentes tamanhos previamente processados. Para tal, o MMP faz uso de um dicionário adaptativo que guarda padrões previamente codificados da imagem e que podem sofrer transformações de escala, permitindo obter blocos de diferentes tamanhos, disponíveis para a codificação da imagem.

3.2 MMP

A primeira versão do MMP [28] particiona a imagem a codificar em blocos de tamanho fixo (macro-blocos) e tenta aproximar cada um deles aos padrões guardados no dicionário. Para cada bloco é realizado, ainda, um particionamento e a sua análise recursiva com os padrões do dicionário. Cada partição do bloco original está associada a uma escala. Cada escala representa uma diferente combinação de altura e largura de um bloco.

Devido ao particionamento do bloco inicial da imagem, o dicionário está também divido em escalas para se efetuar a correspondência dos blocos com padrões do mesmo tamanho. As escalas têm a particularidade de ter metade do tamanho de bloco igual ao tamanho da escala acima, isto porque a segmentação do bloco é feita alternadamente nas direções vertical ou horizontal, a meio do bloco, até se alcançar a escala zero (blocos 1x1, isto é, um píxel). A Figura 3.1 representa o esquema de segmentação de um bloco inicial de 4x4 onde a primeira segmentação é realizada verticalmente.

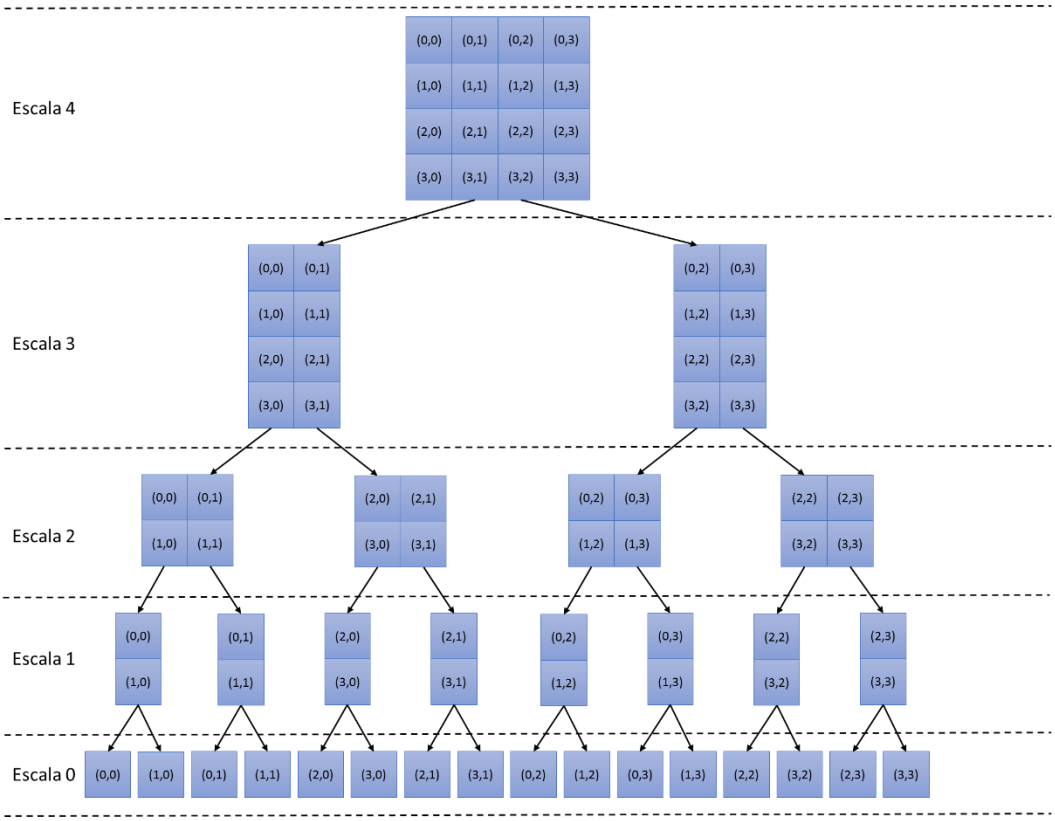

Figura 3.1 - Esquema de segmentação de um bloco 4x4 [29]

Em cada um dos blocos representados é realizada uma pesquisa no dicionário para determinar qual o melhor elemento que representa o bloco em codificação através de um critério de custo. A codificação do bloco inicial gera uma árvore de segmentação binária, que representa a codificação do bloco e será codificada na cadeia de bits (*bitstream)* final.

No final da codificação de cada macro-bloco da imagem é efetuada uma atualização do dicionário. Esta característica permite que na codificação dos blocos seguinte exista informação sobre os blocos previamente codificados.

Todas as etapas do algoritmo MMP serão explicadas com maior detalhe na secção seguinte.

3.3 Algoritmo

O algoritmo MMP pode ser divido em quatro etapas: (1) inicialização, (2) cálculo da distorção e construção da árvore de segmentação, (3) codificação das *flags* de segmentação e dos índices e, por fim, (4) atualização do dicionário.

Inicialização

A primeira fase consiste na inicialização do dicionário e carregamento da imagem a codificar.

O dicionário é uma estrutura de dados dividida em escalas e elementos. Os valores dos pixéis dos elementos do dicionário pertencem ao intervalo de valores [0;255] (correspondente a um *char*). A escala zero, cujo bloco tem tamanho 1x1 (um píxel), contém todos os valores possíveis. Desta forma permite ao algoritmo ter uma base de total conhecimento e assim construir os blocos das escalas acima com concatenação de elementos desta escala. As restantes escalas, nove na primeira versão do MMP

(Tabela 3.1), são inicializadas com blocos uniformes, isto é, todos os pixéis com o mesmo valor. Para não sobrecarregar o dicionário apenas com blocos uniformes e porque também se impõe um limite de elementos por escala, existe um passo entre os elementos uniformes do dicionário. Este passo corresponde a um parâmetro de entrada da aplicação e serve de limitador do número de elementos que cada escala contém na inicialização. A criação do dicionário inicia com o valor mínimo (0) e em cada iteração é incrementado o valor do passo até se atingir o valor máximo (255).

Escala	Dimensões	N.º de pixéis
8	16 x 16	256
7	16 x 8	128
6	8 x 8	64
5	8 x 4	32
4	4 x 4	16
3	4 x 2	8
2	2 x 2	4
1	2 x 1	2
0	1 x 1	1

Tabela 3.1 - Escalas e dimensões dos blocos na primeira versão do MMP

O carregamento da imagem a codificar consiste na divisão da imagem nos blocos de tamanho máximo definido (normalmente 16x16) onde cada bloco é sujeito às etapas 2, 3 e 4.

Cálculo do custo e construção da árvore de segmentação

Para cada macro-bloco (16x16), o algoritmo começa por pesquisar no dicionário, na escala correspondente, qual o melhor elemento que minimiza o custo Lagrangeano [32] da representação do bloco. O custo Lagrangeano (J) é calculado através da fórmula:

$$J = D + \lambda R,$$

onde D corresponde à distorção entre o bloco a codificar e o elemento do dicionário e R é uma taxa (número de bits necessários para a transmissão) associada ao elemento do dicionário. O fator λ é uma constante que permite definir a importância relativa de cada um dos parâmetros, isto é, taxa ou distorção. Um valor baixo de λ oferece maior importância à distorção, o que significa que se pretende diminuir o erro entre a imagem e o elemento do dicionário. Por outro lado, um valor alto de λ aumenta a importância relativa da taxa, isto significa que o algoritmo irá ter tendência para criar aproximações que utilizem um número de bits mais reduzido.

De seguida, o bloco é segmentado verticalmente, obtendo-se dois elementos (X_{esq} e X_{drt}) e procede-se à nova pesquisa no dicionário, desta vez, na escala abaixo. Este processo é aplicado recursivamente até se atingir a escala definida como critério de paragem ou a última escala (zero). No final, é obtida uma árvore de segmentação totalmente expandida que é percorrida de baixo para cima para se realizar o seu desbaste/poda. Se a soma do custo dos dois nós filhos for menor que o custo do nó pai então a melhor opção é utilizar a segmentação. Caso contrário, o bloco não é segmentado e os nós filhos são removidos (desbaste).

Na Figura 3.2 é representado um exemplo da segmentação de um bloco 4x4, que corresponde à escala 4, bem como a árvore de segmentação. Os nós com a cor laranja correspondem a nós segmentados e os nós com a cor azul correspondem aos nós finais, estes nós contém o índice do elemento do dicionário usado para representar aquela partição.

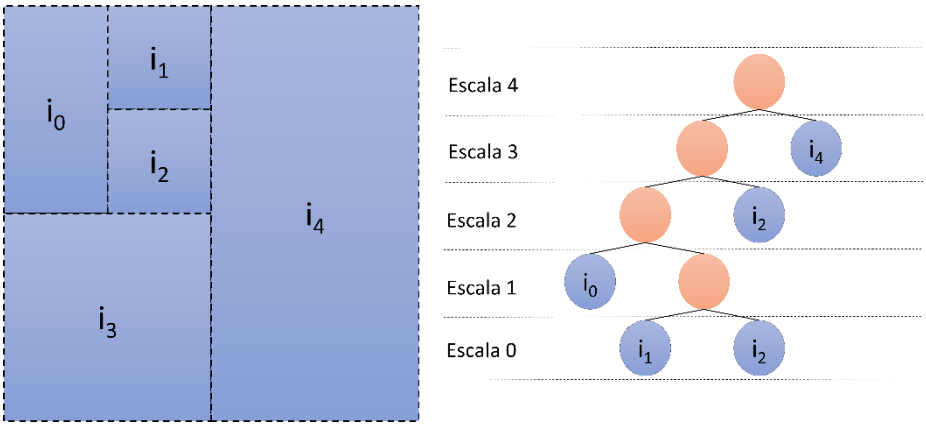

Figura 3.2 - Segmentação de um bloco 4x4, i.e. escala 4 (esquerda), e correspondente árvore binária (direita)

Codificação das *flags* de segmentação e dos índices

Após a construção da árvore de segmentação do bloco, o algoritmo realiza a codificação da árvore para uma cadeia de bits que corresponde ao resultado do algoritmo (*output*). Esta codificação é realizada percorrendo a árvore de segmentação em profundidade iniciando-se na escala mais alta. De acordo com o tipo de nó, isto é, nó segmentado ou nó terminal, o codificador envia um símbolo (*flag)* com um valor diferente. Para um nó segmentado a *flag* é representada pelo valor 0 e para um nó terminal a *flag* contém o valor 1 sendo que de seguida é necessário indicar o índice do elemento do dicionário utilizado. Para o exemplo em estudo (Figura 3.2), a cadeia de bits seria a seguinte:

$$0\ 0\ 0\ 1\ i_0\ 0\ i_1\ i_2\ 1\ i_3\ 1\ i_4\ .$$

Esta codificação é realizada com recurso a um codificador aritmético adaptativo [33] que utiliza um contexto diferente baseado na probabilidade relativa dos símbolos para cada escala do dicionário.

Atualização do dicionário

A atualização do dicionário é um ação efetuada tanto pelo codificador como pelo descodificador. O codificador constrói o dicionário com os padrões codificados da imagem, enquanto o descodificador apenas com a informação da descodificação dos blocos consegue replicar o dicionário construído pelo codificador. Desta forma, não existe a necessidade de transmitir o dicionário construído, aquando da codificação, em conjunto com a imagem codificada.

Como mencionado anteriormente, o MMP utiliza um dicionário adaptativo sendo uma das maiores vantagens do algoritmo em relação aos algoritmos tradicionais baseados em correspondência de padrões. O dicionário é inicializado com um número reduzido de elementos (Inicialização do MMP) e ao longo da codificação este vai crescendo com os padrões da imagem em codificação.

Os novos elementos do dicionário resultam de concatenações de blocos, sempre que existe uma segmentação. Ou seja, sempre que um bloco é particionado, a concatenação dos seus nós filhos é utilizada para gerar um novo bloco que é introduzido no dicionário na escala correspondente. Usando

o exemplo da Figura 3.2, na escala 1 (blocos 1x2) houve uma segmentação em dois blocos da escala 0 (i_1 e i_2). Com a concatenação dos dois blocos da escala 0 (blocos 1x1) é originado um novo bloco para a escala 1. No exemplo estudado seriam gerados quatro novos blocos a adicionar no dicionário em diferentes escalas (blocos com cor laranja na Figura 3.3).

Os blocos gerados pelas concatenações podem sofrer outro tipo de transformações de modo a aumentar a informação disponível para a codificação da restante imagem. Estas transformações podem ser simetrias em relação a zero, simetrias geométricas, rotações ou translações. Deste modo é possível aumentar a probabilidade de existir uma correspondência de padrões em escalas superiores o que corresponde a uma menor transmissão de *flags* e índices.

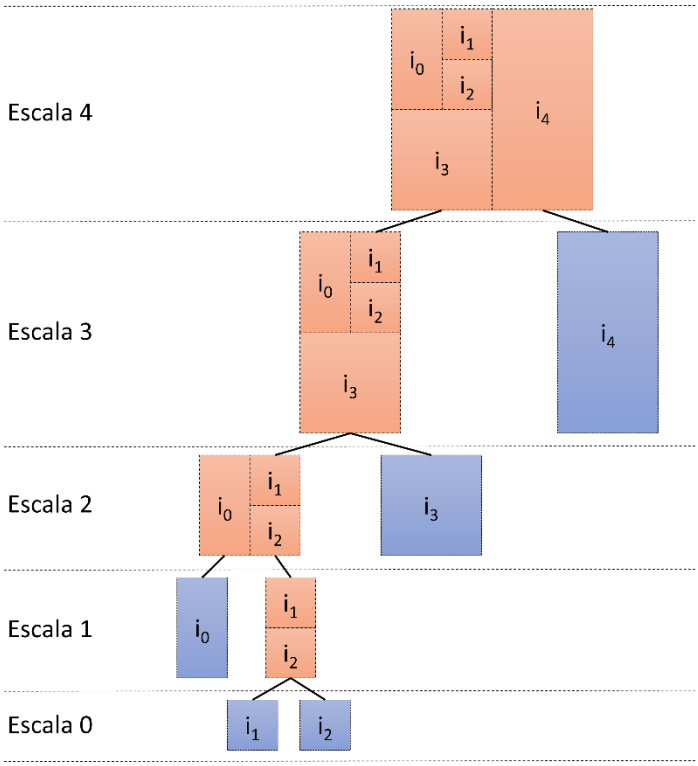

Figura 3.3 - Concatenação dos nós finais (azul) para originar os novos blocos (laranja)

Para cada novo bloco construído, é feito um controlo de redundância antes de este ser inserido no dicionário. Este controlo corresponde à verificação se o dicionário na mesma escala já contém algum elemento igual ou parecido. O critério de parecença corresponde a um parâmetro de entrada denominado *Radius Distortion*. Caso a distorção entre um elemento do dicionário e o novo elemento esteja dentro da esfera originada pelo *Radius Distortion* então concluímos que existe um elemento suficientemente parecido com o novo elemento e este é descartado.

3.4 Predição Intra

A versão do MMP denominada de MMP-Intra [6] resulta da combinação do algoritmo MMP, explicado anteriormente, com os esquemas de predição *intra-frame* utilizados por outros algoritmos de compressão como o H.264/AVC [34]. A predição *intra-frame* é uma técnica que contempla a utilização de blocos previamente codificados na vizinhança do bloco em codificação para gerar uma predição.

Para cada bloco da imagem, são determinados os diversos modos de predição (Figura 3.4) e de seguida o valor gerado é subtraído ao valor real da imagem obtendo-se o resíduo. Este resíduo é usado pelo

MMP para se realizar a codificação. Para cada modo de predição é gerada uma árvore de segmentação e no final é escolhida a que apresentar menor custo. Para a escala 8 (blocos 16x16) são apenas utilizados quatro modos de predição, isto porque com blocos grandes a predição tende a perder precisão e o gasto computacional não compensa as melhorias conseguidas.

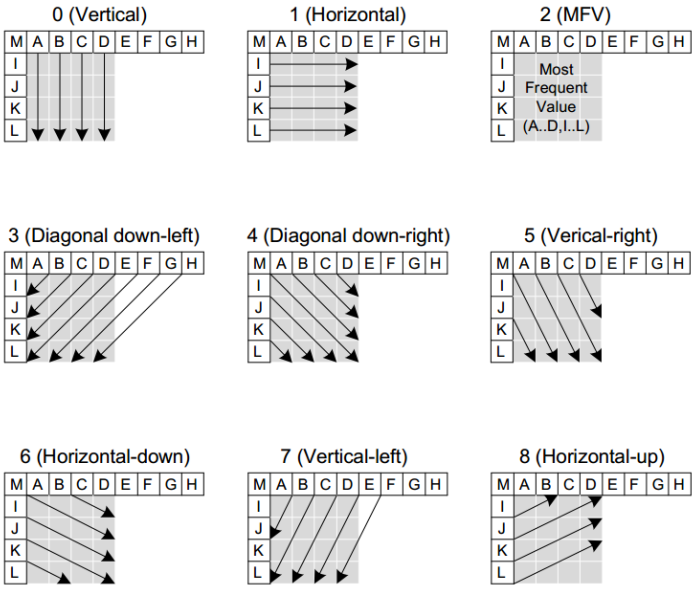

Figura 3.4 - Modos de predição do MMP [34]

Devido ao uso do resíduo para a codificação, no dicionário os valores dos pixéis dos elementos do dicionário pertencem ao intervalo de valores [-255;255]. Nos piores cenários, em que a predição calculada tem um valor 255 (ou 0) e o valor real da imagem é 0 (ou 255), então o resíduo será -255 (ou 255).

A escolha da codificação do resíduo permite aumentar a probabilidade da escolha dos índices do dicionário próximo do valor zero. Isto porque uma boa predição consegue aproximar o valor real da imagem e quando se subtrai o valor da predição ao valor real da imagem obtém-se um valor próximo de zero. Deste modo, a escolha dos índices do dicionário apresenta uma distribuição muito concentrada em torno do valor nulo [6]. Devido a esta particularidade, o passo usado na inicialização do dicionário é variável, ou seja, quando se adiciona blocos afastados do valor zero o passo toma um valor grande. Próximo do valor zero, o passo toma um valor pequeno de modo a criar mais elementos.

A segmentação dos blocos de predição foi anexada à partição do resíduo pelo que obriga à utilização de três *flags*, em vez das duas utilizadas pelo MMP convencional. Estas três *flags* servem para indicar a não segmentação, para indicar a segmentação da predição e do resíduo e a terceira para indicar a segmentação do resíduo mas não da predição.

3.5 Segmentação Flexível

Como visto anteriormente, o MMP inicia a segmentação verticalmente e nas segmentações seguinte utiliza a direção de segmentação inversa à anterior, sucessivamente até atingir a última escala. Testes experimentais demonstraram que para algumas imagens obtêm-se melhores resultados caso se inicie com a segmentação vertical e para outras imagens obtém-se melhores resultados caso se inicie com a segmentação horizontal.

Para colmatar este problema, surgiu o MMP com segmentação flexível [29] em que ambas as direções são testadas e é escolhida a que apresentar um menor custo Lagrangeano. Mais uma vez, caso o custo do nó pai seja menor que a soma dos custos dos nós filhos então não existirá particionamento do nó. Esta nova implementação aumentou o número de escalas disponíveis de 9 para 25 (Figura 3.5) pois cada bloco pode ser particionado vertical ou horizontalmente.

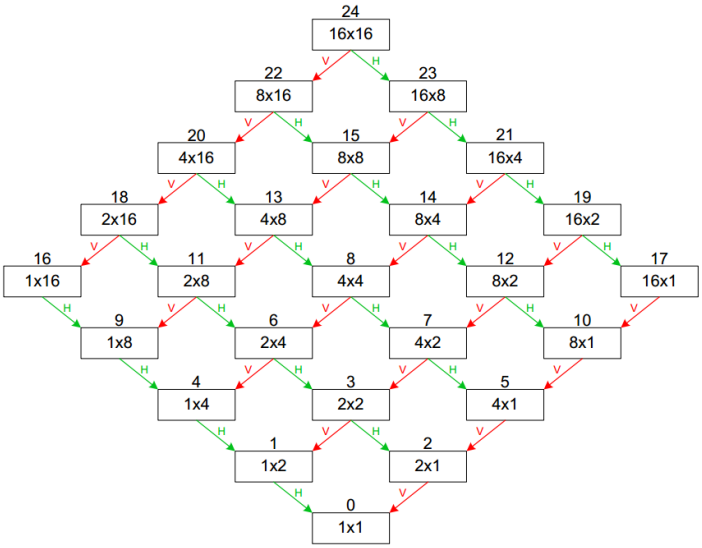

Figura 3.5 - Diagrama de escalas com segmentação flexível

Esta versão necessita de incluir mais duas *flags* em relação à versão anterior para indicar qual a direção da segmentação. Portanto, existem cinco *flags*: (1) não segmentação da predição e do resíduo, (2) segmentação da predição e do resíduo na vertical, (3) segmentação da predição e do resíduo na horizontal, (4) segmentação do resíduo na vertical mas não da predição e, por último, (5) segmentação do resíduo na horizontal e não da predição.

3.6 Resultados

De seguida são apresentados alguns resultados da codificação da imagem Lena com três valores de lambda diferentes: um valor baixo (λ=10) na Figura 3.6, um valor intermédio (λ=50) na Figura 3.7 e um valor alto (λ=300) na Figura 3.8.

Figura 3.6 - Codificação da imagem Lena com λ=10 (original, codificada, diferença)

Figura 3.7 - Codificação da imagem Lena com λ=50 (original, codificada, diferença)

Figura 3.8 - Codificação da imagem Lena com λ=300 (original, codificada, diferença)

Quanto menor o valor de lambda, melhor é a qualidade da codificação, o que resulta também num maior tempo de computação para a codificação da imagem. Como apresentado na Tabela 3.2, o tempo de computação depende do valor de lambda escolhido mas também do número de elementos máximo do dicionário por nível.

Lambda	10	50	300
Limite Dic. 1024	2091,878	1552,451	1093,441
Limite Dic. 50000	5643,951	2444,698	1176,888

Tabela 3.2 - Tempos (segundos) de codificação da imagem Lena com diferentes limites do dicionário e lambdas

Com valores baixo de lambda permite-se que se diminua o erro entre a partição em codificação e os elementos do dicionário. Desta forma, existe um maior particionamento dos blocos o que resulta em mais blocos no dicionário que têm por consequência um maior tempo de pesquisa na busca dos elementos com menor custo. Contudo, permite que se obtenha uma melhor qualidade de codificação da imagem. Com valores altos de lambda é favorecido a taxa pelo que existe menor partição dos blocos e o uso de elementos do dicionário com maior utilização (Figura 3.9).

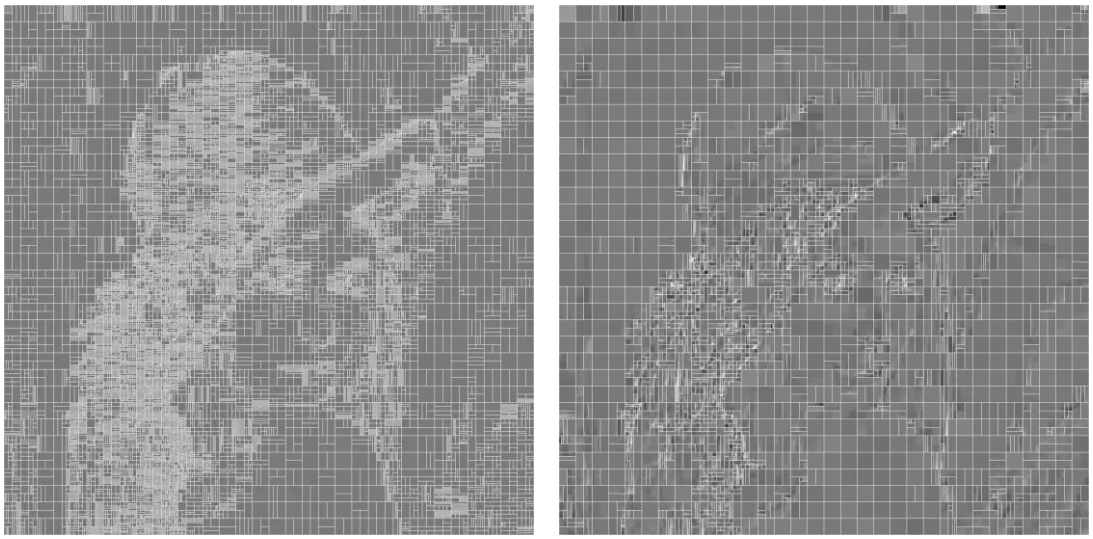

Figura 3.9 - Partição da imagem Lena na codificação com lambda 10 e 300

O limite do tamanho do número de elementos do dicionário tem a sua influência na pesquisa ao dicionário. Quanto maior for o dicionário, mais tempo é gasto na pesquisa do elemento com menor custo.

O algoritmo MMP é computacionalmente muito exigente, o que perfaz que seja necessário muito tempo para a codificação das imagens. Devido a esta exigência, o MMP torna-se um bom candidato para a sua portabilidade para um paradigma de computação paralela.

4. Ambiente Computacional

Neste capítulo são descritos o *hardware*, *software* e *drivers* utilizados na recolha de resultados. Sendo o principal objetivo o estudo dos paradigmas *many-core*, um fator muito relevante para os resultados é o ambiente computacional nomeadamente as GPUs que são empregues. Na parte final do capítulo são descritas as duas imagens usadas nos testes efetuados no decorrer deste trabalho.

4.1 Hardware e Sistemas Operativos

No desenvolvimento deste trabalho foram usados dois servidores diferentes e quatro placas gráficas diferentes, sendo duas delas da marca NVIDIA e as outras duas da marca AMD.

No texto desta dissertação, os servidores são identificados pela nomenclatura de Servidor 1 e Servidor 2 enquanto as GPU são identificadas pelos seus nomes de origem. Assim, para as placas NVIDIA são usados os nomes GTX Titan e GTX 680 e para as placas AMD são usados os nomes R9-290x e HD 7970.

Servidor 1

O servidor 1 é composto por dois processadores Intel® Xeon® CPU E5-2630 @ 2.30 GHz com um total de 24 *cores* e um total de 64 GiB de RAM. O sistema operativo instalado é o Ubuntu 14.04 LTS com a versão do *kernel* 3.13.0-36-generic.

Em relação aos *drivers* instalados neste sistema, para as placas NVIDIA é empregue a versão 340.58 e para as placas AMD a versão 14.30.4.

Servidor 2

O servidor 2 é composto por dois processadores Intel® Xeon® CPU E5-2620 @ 2.00 GHz com um total de 24 *cores* e um total de 32 GiB de RAM. Importa notar que em termos de CPU, o servidor 2 é muito similar ao Servidor 1, diferindo ligeiramente na frequência de relógio (2.30 GHz vs 2.00 GHz). O sistema operativo instalado é o Ubuntu 14.04 LTS com a versão do *kernel* 3.13.0-39-generic.

Em relação aos *drivers* instalados neste sistema, para as placas NVIDIA é a versão 340.58 e para as placas AMD a versão 14.30.4, isto é, foram empregues as mesmas versões do Servidor 1.

NVIDIA GTX Titan Black Edition

A placa NVIDIA GTX Titan Black Edition pertence à arquitetura Kepler [35] com CUDA Capability 3.5. Com 15 multiprocessadores e 192 *cores* por multiprocessador atinge o valor total de 2880 CUDA *Cores* com um *clock rate* de 0,98 GHz.

NVIDIA GTX 680

A placa NVIDIA GTX 680 pertence à arquitetura Kepler [36] com CUDA Capability 3.0. Com 8 multiprocessadores e 192 *cores* por multiprocessador, totaliza 1536 CUDA *Cores* com um *clock rate* de 1,06 GHz.

AMD R9-290x

A placa AMD R9-290x pertence à família Hawaii com suporte até à versão OpenCL 2.0 [37]. Dispõe de 44 *compute units* com um *clock rate* de 925 MHz.

AMD HD7970

A placa AMD R9-290x pertence à família Tahiti com suporte até à versão OpenCL 1.2 [38]. Dispõe de 32 *compute units* com um *clock rate* de 800 MHz.

4.2 CUDA e OpenCL

A versão CUDA final utilizada neste trabalho foi o CUDA 6.5. No decorrer do trabalho iniciou-se com a versão 5.5 mas com a saída de novas versões foi-se atualizando os sistemas pelo que a recolha final de resultados incidiu sob a versão 6.5.

Para o OpenCL foi usada a versão 1.1. A escolha desta versão em detrimento de versões mais recentes deve-se ao facto desta ser a última versão OpenCL disponível para sistemas NVIDIA. Assim, para obtenção de resultados comparáveis fez-se uso da versão 1.1. O SDK da AMD utilizado para *profiling* das aplicações foi a versão 2.9.1.

4.2.1 Resultados de Bandwidth Measurement Test

Para cada placa foi executado o teste de largura de banda (*bandwidth*) disponível na instalação quer do CUDA, para as placas NVIDIA, quer do SDK AMD, para as placas AMD. Este teste mede a largura de banda existente entre a memória do CPU *(host)* e a memória global da GPU *(device).* Os resultados são apresentados nas Tabelas 4.1, 4.2, 4.3 e 4.4.

NVIDIA GTX Titan Black Edition

	Tipo de Cópia	Memória Paginável	Memória Não Paginável (*pinned*)[1]
Servidor 1	Host to Device	2358,9 (MB/s)	6119,2 (MB/s)
	Device to Host	3047,8 (MB/s)	6529,1 (MB/s)
Servidor 2	Host to Device	1819,2 (MB/s)	11548,7 (MB/s)
	Device to Host	2908,6 (MB/s)	12276,5 (MB/s)

Tabela 4.1 - Resultados do bandwidth com memória paginável e não paginável (pinned) para a placa NVIDIA GTX Titan Black Edition

[1] Os valores obtidos para esta métrica são muito diferentes entre os dois servidores. Após várias repetições do teste, inclusive com o reiniciar de ambos os servidores, estes valores mantiveram-se.

NVIDIA GTX 680

	Tipo de Cópia	Memória Paginável	Memória Não Paginável (*pinned*)
Servidor 1	Host to Device	2707,7 (MB/s)	6003,9 (MB/s)
	Device to Host	2465,5 (MB/s)	6530,3 (MB/s)
Servidor 2	Host to Device	3018,4 (MB/s)	6122,1 (MB/s)
	Device to Host	2862,9 (MB/s)	6485,4 (MB/s)

Tabela 4.2 - Resultados do bandwidth com memória paginável e não paginável (pinned) para a placa NVIDIA GTX 680

AMD R9-290x

	Tipo de Cópia	Memória Paginável	Memória Não Paginável (*pinned*)
Servidor 1	Host to Device	3877 (MB/s)	7718 (MB/s)
	Device to Host	4187 (MB/s)	9787 (MB/s)
Servidor 2	Host to Device	4006 (MB/s)	6616 (MB/s)
	Device to Host	3415 (MB/s)	5656 (MB/s)

Tabela 4.3 - Resultados do bandwidth com memória paginável e não paginável (pinned) para a placa AMD R9-290x

AMD HD7970

	Tipo de Cópia	Memória Paginável	Memória Não Paginável (*pinned*)
Servidor 1	Host to Device	3613 (MB/s)	10250 (MB/s)
	Device to Host	3848 (MB/s)	10166 (MB/s)
Servidor 2	Host to Device	4309 (MB/s)	9466 (MB/s)
	Device to Host	3409 (MB/s)	5739 (MB/s)

Tabela 4.4 - Resultados do bandwidth com memória paginável e não paginável (pinned) para a placa AMD HD7970

Em todas as placas o valor de largura de banda é superior quando se utiliza memória não paginável (*pinned memory).*

4.3 Recolha de resultados e imagens de teste

Os tempos recolhidos e mostrados para cada otimização e resultados finais resultam da média da execução de cinco execuções.

Os parâmetros de referência utilizado incidem sob o lambda com valor 10, tamanho do dicionário até 1024 elementos por escala, modo de predição LSP ativo, a transformação ativa de deslocamentos de ½ e particionamento da predição até à escala 8.

Na recolha de tempos em cada versão a imagem de referência foi a imagem Lena.pgm de 512x512 pixéis e 8 bits por píxel (Figura 4.1).

Figura 4.1 - Imagem de testes Lena.pgm

No estudo efetuado sobre o modo de predição LSP (secção 5.9) foi utilizada a imagem Barbara.pgm (Figura 4.2), também com 512x512 pixéis e 8 bits por píxel. A escolha desta imagem deve-se às texturas presentes na imagem que têm direções que variam muito e neste tipo de texturas o LSP é capaz de fazer uma predição mais eficiente do que os modos direcionais tradicionais.

Figura 4.2 - Imagem de teste Barbara.pgm

5. Estudo da Versão CUDA-MMP

Este capítulo apresenta os resultados do *profiling* da versão sequencial onde foram identificados as secções do algoritmo passíveis de serem implementadas usando um paradigma *many-core*. De seguida, é apresentada a primeira implementação do MMP com recurso a CUDA e os resultados. No final do capítulo, são descritas várias otimizações efetuadas na aplicação CUDA-MMP com o intuito de melhorar o tempo de execução desta aplicação. De notar que as otimizações descritas não estão por ordem cronológica, pelo que nos resultados de cada otimização é feita a comparação com a versão de referência, isto é, a versão anterior à otimização implementada.

5.1 Speedup

A comparação dos resultados das novas otimizações assenta no conceito de *speedup*. O *speedup* é uma métrica usada para calcular a melhoria de desempenho de uma determinada tarefa. Neste trabalho, o *speedup* será calculado usando a equação:

$$SpeedUP = \frac{Tempo\ da\ versão\ de\ referência}{Tempo\ da\ nova\ versão}.$$

Caso o resultado seja superior a 1, significa que houve melhoria do desempenho com a nova implementação, caso o valor esteja no intervalo de]0;1[então existiu uma degradação do desempenho pelo que a alteração deve ser descartada. Finalmente, caso o valor seja igual a 1, então não existiu variação do desempenho.

5.2 Profiling da Versão Sequencial

Antes de se iniciar a implementação *many-core* do MMP procedeu-se a um estudo sobre a versão sequencial. O estudo teve como objetivo encontrar as principais fases/operações do algoritmo que consomem mais tempo de execução e que são os potenciais candidatos a implementar numa arquitetura paralela.

De modo a identificar as funções da versão sequencial do algoritmo MMP que demoram mais tempo, foi empregue a ferramenta de *profiling GNU gprof* [39]. Para ativar esta ferramenta é necessário adicionar a opção de compilação "*-pg*" ao compilador GCC [40]. A Figura 5.1 apresenta os resultados da codificação da imagem Lena (lambda 500) com a versão sequencial do MMP. A Figura 5.1 foi gerada pela ferramenta *KProf* [41], uma ferramenta de *profiling* gráfica que processa o resultado proveniente da ferramenta *gprof*.

Da análise dos resultados depreende-se que a função *quad_err* é a que consome mais tempo de execução no codificador MMP. Esta função é responsável por determinar qual o elemento do dicionário que melhor representa a partição dada. Importa notar que o elevado consumo de tempo desta função também se deve ao elevado número de vezes que é chamada.

A segunda e terceira funções que consomem mais tempo*, optimize_block_and_pred_mode* e *optimize_block*, respetivamente, são responsáveis pelo particionamento dos blocos e escolherem a melhor segmentação para cada partição/bloco.

A quarta função com maior consumo de tempo é *add_block2dic_md*. Esta função também efetua uma pesquisa no dicionário mas com o intuito de verificar se o dicionário já contém um elemento igual ou parecido (considerando o *Radius Distortion*) ao novo elemento candidato a ser adicionado ao dicionário, controlo de redundância.

Flat Profile | Hierarchical Profile | Object Profile | Graph View | Method View

Function/Method	Count	Total (s)	%	Self (s) ▲	Total ms/call	Self ms/call
Hierarchy						
quad_err	2029629461	1411.350	59.900	1411.350	0.000	0.000
optimize_block_and_pred_mode	1024	2233.520	94.800	355.670	2.180	0.350
optimize_block	5226681	1819.180	77.300	290.100	0.000	0.000
add_block2dic_md	22614	120.210	5.100	113.840	0.010	0.010
rate	42926522	60.160	2.600	60.160	0.000	0.000
rate_dicseg	42926522	42.400	1.800	42.400	0.000	0.000
free_residue_tree	3385987328	41.730	1.800	41.730	0.000	0.000
ReconstructOriginalBlock	676864	7.930	0.300	7.930	0.000	0.000
intmatrix	456838811	5.320	0.200	5.320	0.000	0.000
Escala	126679	4.560	0.200	4.560	0.000	0.000
rate_flag	294361466	4.450	0.200	4.450	0.000	0.000
free_intmatrix	456838812	3.410	0.100	3.410	0.000	0.000
block_prediction	676864	6.750	0.300	3.320	0.000	0.000
block_prediction_dec	676864	3.430	0.100	3.180	0.000	0.000
rate_lossless	165750624	2.140	0.100	2.140	0.000	0.000
new_arithmetic_symbol	114996	1.780	0.100	1.780	0.000	0.000

Figura 5.1 - Resultado da ferramenta de profiling GNU gprof

A Figura 5.2 mostra o diagrama de execução desenhado pela ferramenta *KProf*, relativo à codificação da imagem Lena (λ=500). Este diagrama apresenta o fluxo do algoritmo MMP identificando onde são chamadas as funções com maior consumo de tempo.

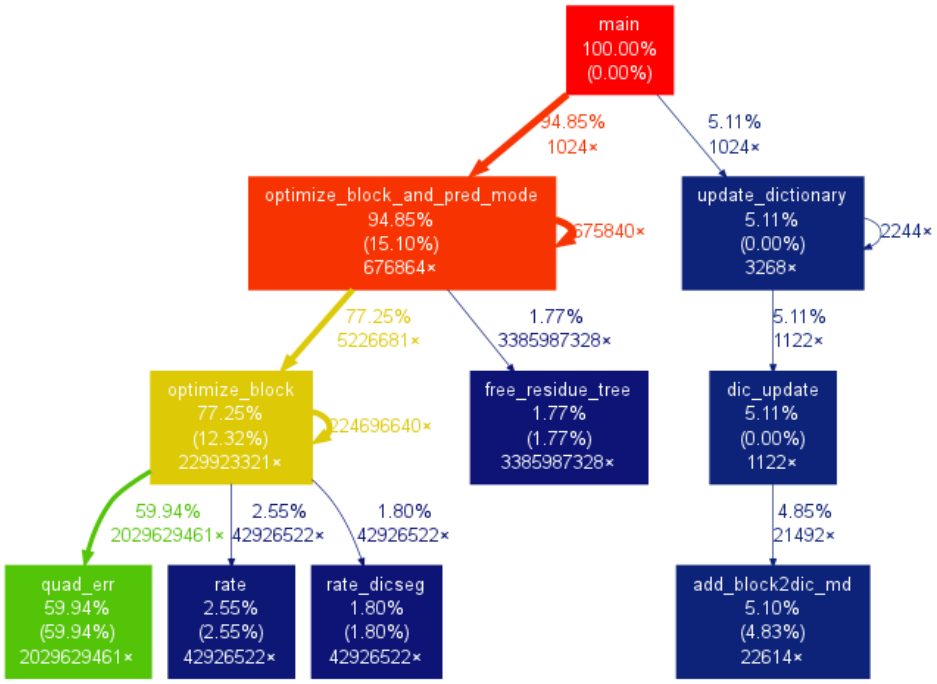

Figura 5.2 - Diagrama de execução da codificação da imagem Lena com λ=500

Podemos concluir que as ações do algoritmo MMP que mais tempo consome são as pesquisas no dicionário. Estas pesquisas são efetuadas na codificação do bloco e na atualização do dicionário. A

primeira pesquisa mencionada apenas é executada pelo codificador pelo que apenas influencia o tempo de execução desta aplicação. A pesquisa efetuada na atualização do dicionário é realizada pelo codificador e descodificador pelo que permite influenciar o tempo de execução de ambas as aplicações.

5.3 Introdução à versão CUDA-MMP

Aquando do início deste trabalho, já existia implementado um protótipo do MMP em CUDA. Este protótipo não atingiu os objetivos de acelerar a codificação das imagens. A primeira fase deste trabalho consistiu em estudar esta versão de modo a identificar os pontos prejudiciais à velocidade de execução e melhorá-los.

5.3.1 Implementação do protótipo CUDA-MMP

Como mostrado anteriormente, o algoritmo MMP é composto por quatro etapas (secção 3.3) e o *profiling* da aplicação sequencial indica que o maior consumo de tempo está associado às operações de pesquisa do dicionário. As pesquisas no dicionário ocorrem na etapa dois para o cálculo da distorção e construção da árvore de segmentação e na etapa quatro para validar se um novo elemento deve ser inserido no dicionário. A versão CUDA-MMP visou essencialmente otimizar a operação de pesquisa do dicionário na fase dois do algoritmo, dado que representa a maior fatia do tempo de execução.

Esta implementação contém duas cópias do dicionário sincronizadas, uma na memória da CPU e outra na memória da GPU. Esta decisão prendeu-se com o facto de o cálculo das distorções ser realizado pela GPU e a atualização do dicionário ser realizada pela CPU. Desta forma, após a inicialização do dicionário na CPU, é alocada memória na GPU correspondente ao tamanho máximo possível para todos os elementos do dicionário (parâmetro de entrada) e posteriormente é efetuada a cópia do dicionário para a GPU. Após o processo da atualização do dicionário, existe um *kernel* ("*cuda_update_dic*") responsável por escrever os novos elementos no local certo. A implementação deste *kernel* permite que seja efetuada apenas uma cópia com todos os novos elementos ao invés de uma cópia para cada novo elemento. Deste modo, minimizam-se as operações de cópia entre CPU e GPU, e aumenta-se a utilização da largura de banda disponível pelo dispositivo.

A segunda etapa do algoritmo é composta por duas fases. A primeira fase consiste na pesquisa do melhor elemento do dicionário, ou seja, o melhor padrão que identifica a partição. A segunda fase corresponde à construção da árvore de segmentação. Apenas a primeira parte foi implementada com recurso à plataforma CUDA pois é onde se efetua a pesquisa. A complexidade de partilha de memória, devido à dependência de dados entre escalas, da construção da árvore de segmentação torna difícil chegar a uma implementação em CUDA. Para o efeito, foram implementados dois *kernels:* o primeiro *kernel* denominado "*cuda_optimize_block*" é responsável pelo cálculo do custo Lagrangeano de cada elemento do dicionário para cada sub-partição da partição a codificar. O segundo *kernel* com o nome "*cuda_j_reducter*" é responsável por efetuar uma redução dos valores do *kernel* anterior e devolver o elemento do dicionário com o menor custo para cada partição. Para a "comunicação" entre os dois *kernels* foram criados quatro *buffers*: um para os custos, dois para as taxas (taxa do elemento e taxa do segmento onde o elemento está inserido) e outro para os índices. A Figura 5.3 apresenta o diagrama de sequência para a codificação de uma partição do macro-bloco. Esta sequência é repetida para cada modo de predição disponível na partição em codificação.

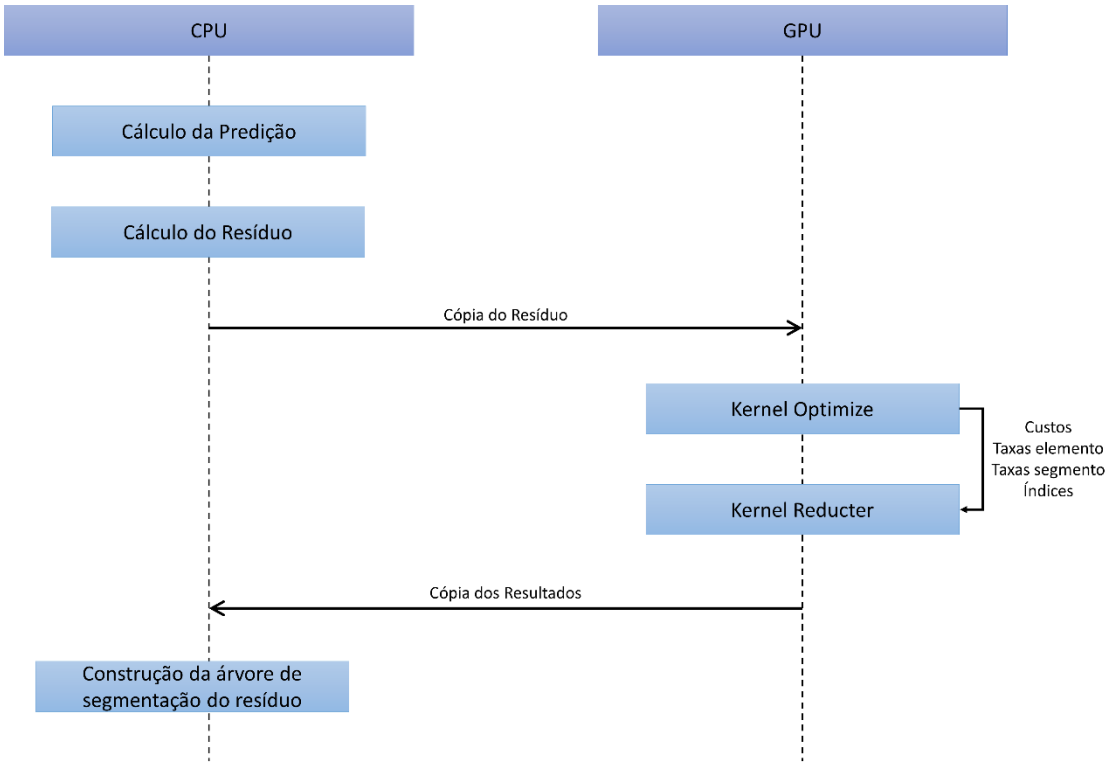

Figura 5.3 - Esquematização da codificação de uma partição

Portanto para cada modo de predição são realizadas duas chamadas de *kernel* e duas cópias entre membros, uma da CPU para a GPU com os resíduos e outra da GPU para a CPU com os resultados do processamento dos dois *kernels.*

5.3.2 Resultados

Apesar dos esforços em tornar a versão CUDA-MMP mais rápida que a versão sequencial, ou seja, a versão de referência, esta versão multi-núcleo relevou-se muito mais lenta que a própria versão sequencial (Tabela 5.1).

	Servidor1			**Servidor2**		
Versão	**Sequencial**	**CUDA-MMP**		**Sequencial**	**CUDA-MMP**	
		GTX Titan	**GTX 680**		**GTX Titan**	**GTX 680**
Tempo	2091,878	11248,366	16189,448	2091,878	11306,530	16279,100
Speedup	-	0,186	0,129	-	0,185	0,129

Tabela 5.1 Resultados da versão sequencial e CUDA-MMP

Um estudo mais aprofundado mostra que o *kernel* de redução, é o maior consumidor de tempo, correspondente a 65,5% do tempo total de codificação da imagem. Por sua vez, o *kernel* de otimização corresponde a 28,1% do tempo total.

5.4 Reengenharia do Kernel de Redução

O *kernel* de redução (*cuda_j_reducter*) é responsável pela redução dos dados provenientes do *kernel* de otimização (*cuda_optimize_block*). Após a execução do *kernel* de otimização, são gerados os custos Lagrangeano de cada elemento do dicionário para cada sub-partição do bloco em codificação, pelo que no final pretende-se apenas o elemento que apresente o menor custo. O *kernel* de redução para cada sub-partição do bloco tem de efetuar a redução dos dados para que seja devolvido para a CPU o melhor elemento do dicionário de cada sub-partição que será utilizado na construção da árvore de segmentação.

5.4.1 Introdução e Profiling

Na implementação existente do *kernel* de redução, cada *thread* é responsável pela redução dos custos para uma determinada partição do bloco em codificação. Por exemplo, na codificação do primeiro bloco da imagem (escala 24), a primeira *thread* (id=0) irá ser responsável pela redução dos dados para a primeira partição do bloco (o próprio bloco). Inicia com o primeiro valor, que corresponde ao primeiro elemento do dicionário da escala 24, e guarda esse valor num registo. Caso algum elemento do dicionário seguinte tenha um valor de custo menor, esse valor será guardado no registo criado para esse efeito. No final, a *thread* escreve o resultado numa memória global que será copiada para a CPU. As *threads* seguintes são responsáveis pelas partições seguintes até se atingir a escala 0. Caso o número de total de *threads* em uso seja menor que o número total de partições do bloco em codificação, cada *thread* poderá efetuar mais que uma redução.

A implementação do *kernel* de redução encontrava-se mal concretizada pois não existe balanceamento de trabalho. Visto o número de elementos do dicionário por escala não ser constante, isto é, cada escala contém um número diferente de elementos, então existem *threads* que efetuam mais trabalho do que outras. Isto provoca divergência de *branchs*, ou seja, diferentes *threads* seguem diferentes caminhos de execução. Neste caso, algumas *threads* já tinham terminado a sua pesquisa mas as restantes continuavam a percorrer os elementos do dicionário. A ferramenta de *profiling* NVIDIA Visual Profiler (NVVP) mostra que cerca de 95% do tempo as *threads* ficam inativas (Figura 5.4), o que ajuda a explicar o fraco desempenho do *kernel*.

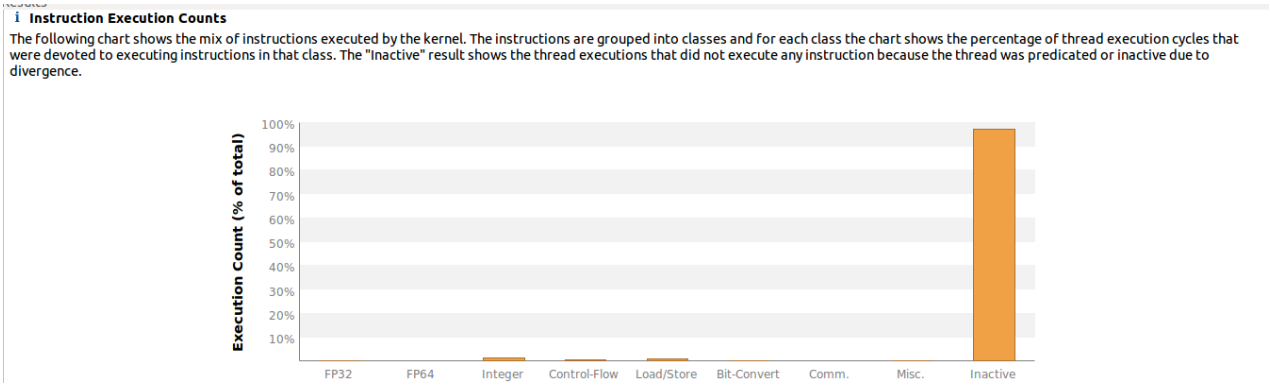

Figura 5.4 - Trabalho das threads na execução do kernel de redução

Outro problema encontrado, corresponde ao baixo valor de ocupância [42] do *kernel*. A ocupância é uma métrica que descreve a percentagem de capacidade de processamento do *kernel* por parte do *hardware.* Em suma, mede a percentagem de tempo que um multiprocessador (SM) se encontra a executar processamento. Como podemos verificar na Figura 5.5, o valor atingido é de 4,7% o que

corresponde a um valor muito baixo. Desta forma não se consegue esconder a latência das operações executadas pelas *threads*.

Variable	Achieved	Theoretical	Device Limit
Occupancy Per SM			
Active Blocks		8	8
Active Warps	2.27	8	48
Active Threads		256	1536
Occupancy	4.7%	16.7%	100%

Figura 5.5 - Ocupância do kernel de redução

Como mencionado na secção 5.3.2, o *kernel* de redução é a ação com maior representação de tempo do codificador. Na versão de referência para esta otimização, este *kernel* representa 73,47% do tempo total de execução.

5.4.2 Implementação do novo kernel de redução

De modo a melhorar o balanceamento de trabalho entre as *threads,* o comportamento da nova implementação foi alterado para que cada bloco de *threads* seja responsável pela redução dos custos de um sub-bloco do bloco em codificação. Devido a esta particularidade, o *kernel* é lançado com uma configuração da grelha de execução cujo número de blocos é igual ao número total de partições existente no bloco em codificação.

Esta nova implementação divide o *kernel* em três passos: (1) contextualização, (2) redução do espaço de valores até um valor igual ao número de unidades ativas e, por fim, (3) redução dos valores do segundo passo, usando uma redução clássica de computação paralela, e escrita dos resultados.

1. **Contextualização**

A contextualização tem como objetivo que a *thread* identifique qual a partição do bloco que irá realizar a redução, através do identificador (ID) do bloco em que está inserido. Aliado a isso é calculado o nível a que pertence essa partição de modo a obter quantos elementos do dicionário existem para se realizar o passo 2 do *kernel*.

2. **Redução do espaço de valores**

Este passo consiste na redução de todos os custos existentes de uma determinada partição para um número de custos igual ao número de *threads* que serão reduzidos no último passo.

Esta redução é efetuada percorrendo o espaço de valores usando o ID das *threads* dentro do bloco (*threadIdx.x*). Cada *thread* faz a comparação com o elemento seguinte usando como deslocamento múltiplos do número de *threads* até se atingir um valor superior ao número de elementos do dicionário naquela escala. Esta escolha de deslocamento permite que existam acessos alinhados à memória, e consequentemente um maior aproveitamento dos recursos pois os acessos à memória global são os acessos mais lentos. A Figura 5.6 esquematiza o acesso ao vetor de custos realizado pela *thread* com ID=0, num exemplo em que sejam utilizadas oito *threads*.

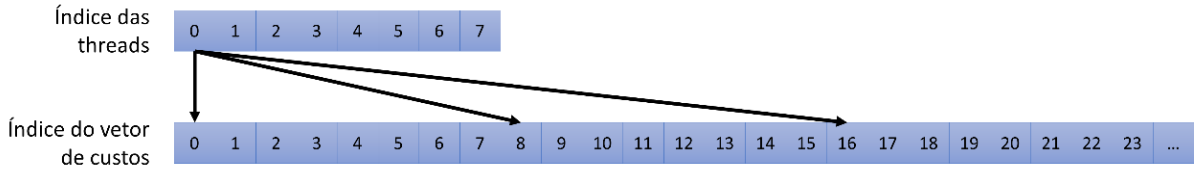

Figura 5.6 - Esquema de acessos ao vetor de custos pela thread com ID=0

No final desta fase, cada *thread* guardou o índice do dicionário com menor custo encontrado nos seus acessos e escreve esse valor numa memória partilhada. Após a escrita é feito um sincronismo entre as *threads* para que passe ao próximo passo apenas quando todas as *threads* terminarem a escrita dos valores na memória. Desta forma, o *buffer* de comunicação entre o *kernel* de otimização e o *kernel* de redução que guardava os índices deixou de ser necessário pelo que se procedeu à sua remoção.

3. Redução do último conjunto de valores e escrita dos resultados

Este passo reduz o último conjunto de valores que corresponde ao número de *threads* em utilização. Este conjunto de valores encontra-se na memória partilhada, aproveitando o facto de esta memória estar acessível a todas as *threads* que se encontram no mesmo *thread-block.*

Neste passo é utilizada a técnica de *Sequential Addressing* [43], usualmente utilizada em algoritmos de redução em computação paralela. Esta técnica apenas funciona com um número de *threads* cujo valor é uma potência de 2. Inicia-se com um valor de *threads* ativas igual a metade ao número de total de *threads* e esse valor é usado como deslocamento para a comparação de dados. Em cada iteração o número de *threads* ativas é dividido por 2 até se atingir uma única unidade ativa que irá comparar os dois últimos valores e obter o resultado desejado. Cada unidade ativa faz a comparação entre o seu valor, ou seja, valor que se encontra na memória partilhada no índice com valor igual ao seu ID, e o valor cujo índice é igual ao seu ID mais o deslocamento. Caso o segundo valor seja menor, então este é escrito na posição com ID igual ao da *thread* e este torna-se o seu novo menor valor. De realçar que no final de cada iteração é necessário proceder-se a um sincronismo entre as *threads* para que caso existam escritas em memória, se espere pelo término desta operação.

A Figura 5.7 esquematiza a técnica de *Sequential Addressing* usando oito *threads*. Na primeira iteração, temos quatro *threads* ativas e na *thread com ID*=2, existiu uma alteração do menor valor pelo que este é escrito na posição 2 da memória partilhada. Este processo é aplicado até se obter apenas uma *thread* ativa (deslocamento=1) e, no final, o índice zero da memória partilhada contém o menor valor.

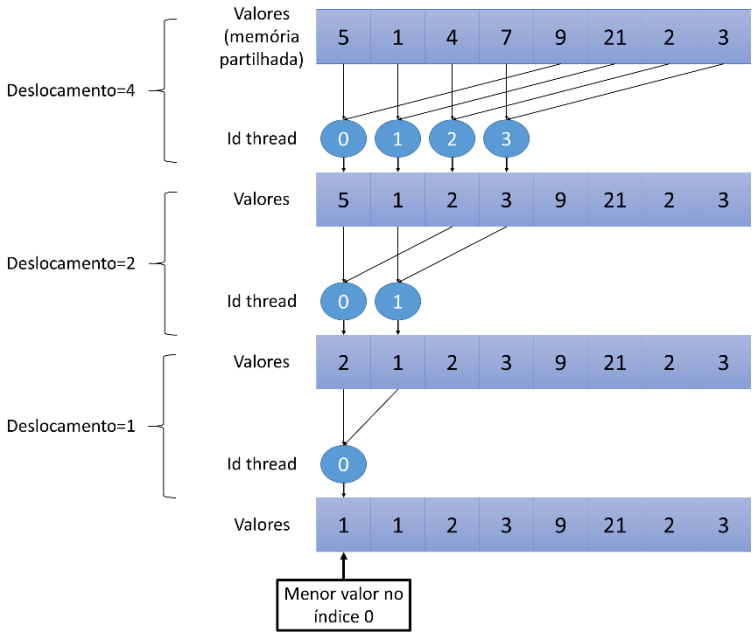

Figura 5.7 - Esquema de redução com Sequential Addressing (adaptado de [43])

No final deste passo, a última *thread* ativa (ID=0) escreve o valor encontrado num vetor de resultados na memória global. Além dos índices, é também escrito o custo, a distorção, as taxas e o nível como resultado deste *kernel* para cada partição. No final da execução do *kernel*, este vetor com os resultados é copiado para a CPU para se continuar a execução do algoritmo MMP, ou seja, a construção da árvore de segmentação.

5.4.3 Resultados

Comparando o *profiling* desta nova implementação do *kernel* de redução com a versão de referência podemos concluir que a percentagem de tempo em que as *threads* estão inativas diminui de 95% (Figura 5.4) para 35% (Figura 5.8).

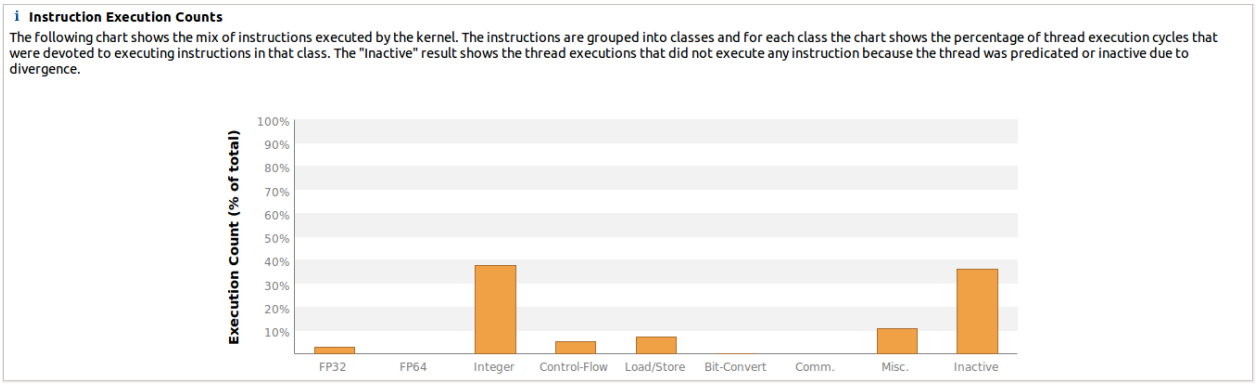

Figura 5.8 - Trabalho das threads na execução do novo kernel de redução

O valor de ocupância, capacidade de esconder a latência das instruções, subiu de 4,5% (Figura 5.5) para 30% (Figura 5.9).

Variable	Achieved	Theoretical	Device Limit
Occupancy Per SM			
Active Blocks		8	8
Active Warps	14.6	16	48
Active Threads		512	1536
Occupancy	30.4%	33.3%	100%

Figura 5.9 - Ocupância do novo kernel de redução

A Tabela 5.2 mostra os resultados obtidos com a nova implementação do *kernel* de redução. Foi conseguido um speedup médio de 3,7, o que prova as melhorias já encontradas usando a ferramenta de *profiling*.

	Servidor 1 GTX Titan	Servidor 1 GTX 680	Servidor 2 GTX Titan	Servidor 2 GTX 680
Versão Referência	5935,002	7367,227	6054,389	7452,202
Novo *Kernel* de Redução	1773,222	1793,745	1827,911	1858,432
Speedup	3,347	4,107	3,312	4,010

Tabela 5.2 - Resultados de desempenho do novo kernel de redução

Com esta nova implementação, o impacto do *kernel* de redução sob o tempo total da codificação diminui para 11,70%. Desta forma, deixou de ser a ação com maior impacto.

5.5 Otimizações aos Kernels

Como já foi explicado anteriormente, a aplicação CUDA-MMP contém três *kernels*. Dois foram implementados para realizar a pesquisa de qual o melhor elemento do dicionário que representa cada bloco e sub-bloco em codificação (*cuda_optimize_block* e *cuda_j_reducter*) e o outro para atualizar o dicionário contido na memória do dispositivo (*cuda_update_dic*).

5.5.1 Operações Aritméticas

As operações aritméticas são o tipo de operações para as quais as GPUs se encontram fortemente otimizadas. Isto porque o processamento gráfico para o qual as GPUs foram desenhadas necessita de muitos cálculos. Contudo certas operações necessitam de vários ciclos de relógios o que implica que a *thread* esteja parada nessas operações mais demoradas. Os *kernels* não são exceção e o *profiling* mostra que esta é uma operação que consome um significativo tempo de execução (Figura 5.10).

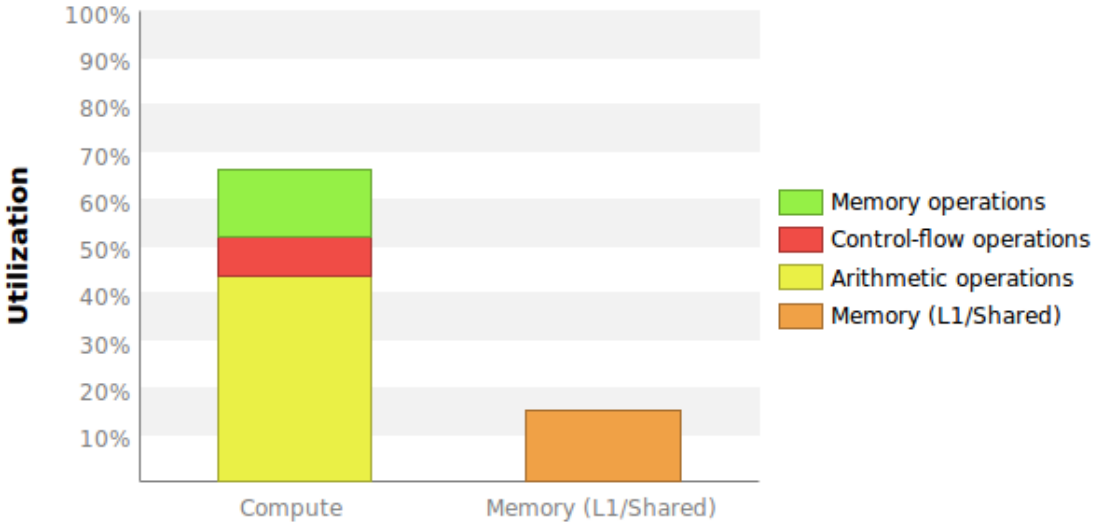

Figura 5.10 - Latência de operações da versão de referência

As operações de divisão e módulo são operações que são utilizadas com bastante regularidade nos *kernels* implementados. Estes dois tipos de operações têm particularidades que se podem aproveitar quando o divisor é uma potência de 2. Quando este detalhe acontece, podemos substituir a operação de divisão e módulo por operações de deslocamento e lógica:

$$\frac{i}{n} = i \gg \log_2 n \quad \text{e} \quad i \,\%\, n = i \,\&\&\, (n-1), \text{ onde n é potência de 2}$$

Visto o logaritmo também ser uma operação aritmética pesada foi criada uma tabela de logaritmos, isto é, estes são calculados na CPU (já existiam visto o algoritmo MMP usar estes valores) e copiados para uma memória das constantes da GPU. Assim, em lugar de calcular o logaritmo, procede-se à pesquisa do valor na tabela. Esta tabela contém os valores dos logaritmos até ao elemento 32, pois é o índice de maior valor que os *kernels* necessitavam. Cada posição da tabela contém o valor do logaritmo correspondente ao índice. Portanto, para calcular o logaritmo do elemento 32, acede-se ao índice da tabela com valor 32.

Como se pode verificar na Figura 5.11, em comparação ao gráfico anterior (Figura 5.10), existiu um ligeiro decréscimo da latência das operações aritméticas. A Tabela 5.3 mostra os resultados obtidos por esta otimização pelo que podemos registar um *speedup* médio de 1,009 vezes.

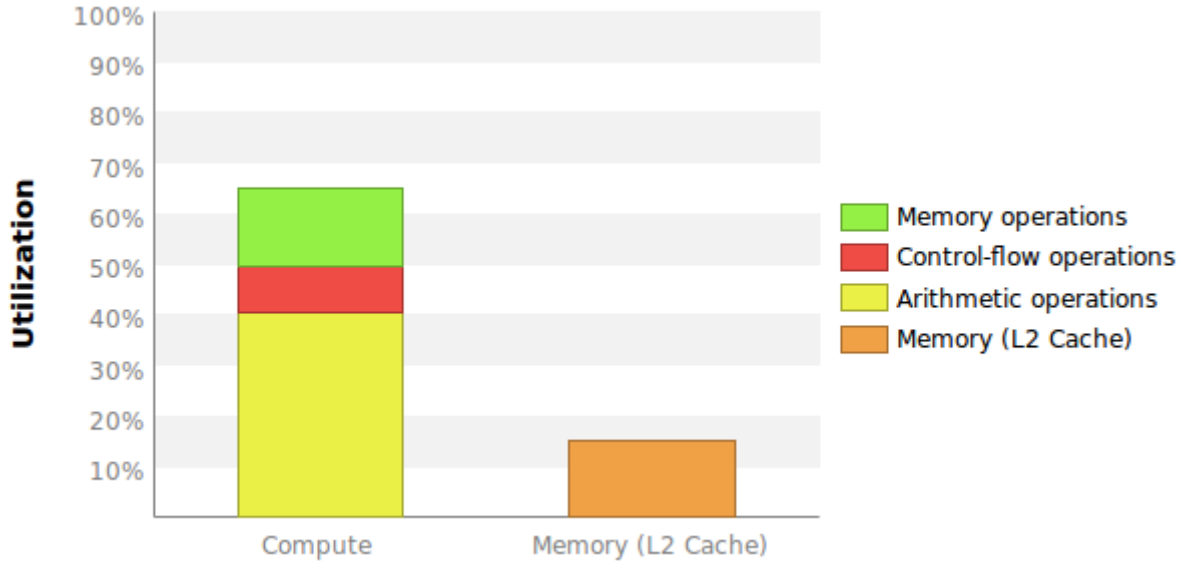

Figura 5.11 - Latência de operações da versão com otimização das operações aritméticas

	Servidor 1 GTX Titan	Servidor 1 GTX 680	Servidor 2 GTX Titan	Servidor 2 GTX 680
Versão Referência	496,193	490,298	541,497	533,013
Otimização operações aritméticas	493,757	488,193	533,566	527,430
Speedup	1,005	1,004	1,015	1,011

Tabela 5.3 - Resultados da otimização das operações aritméticas

5.5.2 Loop Unrolling

O *Loop Unrolling* [44] é uma técnica de transformação de ciclos de repetição para otimizar a velocidade de execução das aplicações. Esta técnica permite diminuir o número de iterações em ciclos de repetição e consequente verificação de saltos, com a desvantagem do aumento do tamanho do código. Esta transformação pode ser efetuada manualmente ou usando compiladores otimizados para o mesmo.

No passo 2 do *kernel* de redução, esta técnica pode ser aplicada para diminuir o número de saltos do ciclo de repetição usado para percorrer todos os valores do custo. Foram implementadas duas versões, uma que realiza o *loop unrolling* de duas vezes e outra versão com quatro vezes, ou seja, em cada iteração em vez de se verificar apenas um valor, são verificados dois ou quatro valores. Os resultados obtidos (Tabela 5.4) mostram que a versão onde foi introduzido o *loop unrolling* de duas vezes foi o que obteve melhores resultados, embora o *speedup* seja pouco significativo.

	Servidor 1 GTX Titan	Servidor 1 GTX 680	Servidor 2 GTX Titan	Servidor 2 GTX 680
Versão Referência	492,335	488,914	532,691	527,834
Loop Unrolling 2x	491,878	488,323	531,652	528,378
Speedup	1,001	1,001	1,002	0,999
Loop Unrolling 4x	493,429	489,869	536,229	530,057
Speedup	0,998	0,998	0,993	0,996

Tabela 5.4 - Resultados da aplicação de loop unrolling no kernel de redução

5.5.3 Otimização do Sincronismo

Como mostrado anteriormente, entre o segundo e o terceiro passo do *kernel* de redução existe um sincronismo entre as *threads* para se assegurar que a escrita dos resultados para a memória partilhada foi completada. Ainda dentro do terceiro passo, também, é necessário sincronismo no final de cada iteração da redução realizada neste passo. Este sincronismo é realizado com recurso à função CUDA "*__syncthreads()*" que espera que todas as *threads* dentro do mesmo bloco terminem esta instrução. Devido a esta quantidade de sincronismo necessário, a execução do *kernel* é obrigada a parar e esperar que todas as *threads* terminem as suas ações até ao ponto de sincronismo. O profiling ao *kernel* de redução mostra que o sincronismo das *threads (synchronization)* é um dos principais critérios de paragem (Figura 5.12).

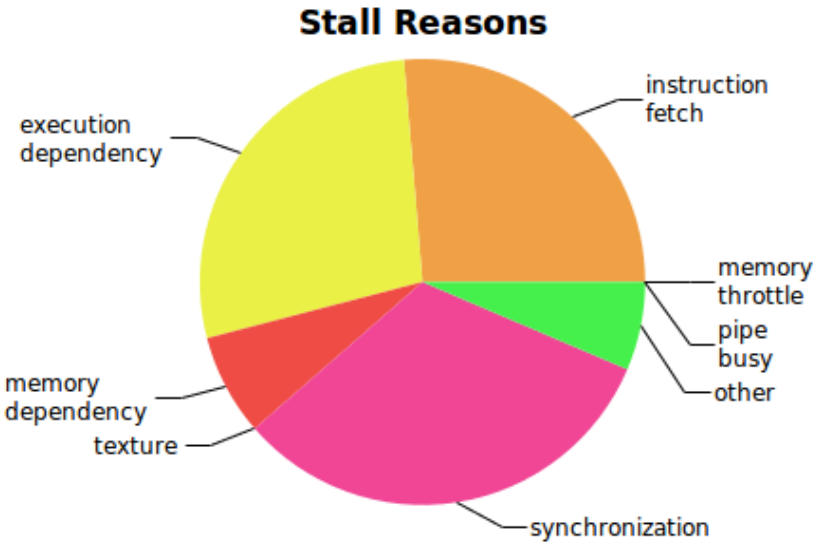

Figura 5.12 - Critérios de paragem do kernel de redução

Visto as instruções serem SIMD (*Single Instruction, Multiple Data*) e síncronas dentro do mesmo *warp*, ou seja, dentro do mesmo *warp* todas as *threads* se encontram a executar a mesma instrução. Isto significa que no terceiro passo do *kernel* de redução podemos descartar algum sincronismo no final das iterações.

```
1.  while(i != 0)
2.  {
3.      if(threadIdx.x < i)
4.      {
5.          //reduction threadIdx.x and threadIdx.x+i
6.          …
7.      }
8.      __syncthreads();
9.
10.     i >>= 1;
11. }
```

Quando o número de *threads* ativas atinge o valor igual ao número de *threads* de um *warp* então não é necessário sincronismo nestas *threads*. Portanto, esta otimização consiste na paragem do ciclo de repetição quando se atinge o número de *threads* ativas igual a 32 (*warp size*). Desta forma passa a ser empregue repetição de código, similar/idêntico ao *loop unrolling*, e os deslocamentos são explícitos com constantes aritméticas em vez de uma variável, conforme a listagem seguinte:

```
1.  while(i > 32)
2.  {
3.      if(threadIdx.x < i)
4.      {
5.          //reduction threadIdx.x and threadIdx.x+i
6.      }
7.      __syncthreads();
8.
9.      i >>= 1;
10. }
11.
12. if (threadIdx.x < 32)
13. {
14.     //reduction threadIdx.x and threadIdx.x+32
15.     ...
16.     //reduction threadIdx.x and threadIdx.x+16
17.     ...
18.     //reduction threadIdx.x and threadIdx.x+8
19.     ...
20.     //reduction threadIdx.x and threadIdx.x+4
21.     ...
22.     //reduction threadIdx.x and threadIdx.x+2
23.     ...
24.     //reduction threadIdx.x and threadIdx.x+1
25.     ...
26. }
```

Apesar de um novo *profiling* ao *kernel* de redução mostrar que existe uma diminuição do impacto do sincronismo como critério de paragem (Figura 5.13), os tempos de execução mantém-se muitos próximos da versão de referência. Acresce-se que para a placa GTX 680 os resultados apresentam um desempenho ligeiramente negativo (Tabela 5.5).

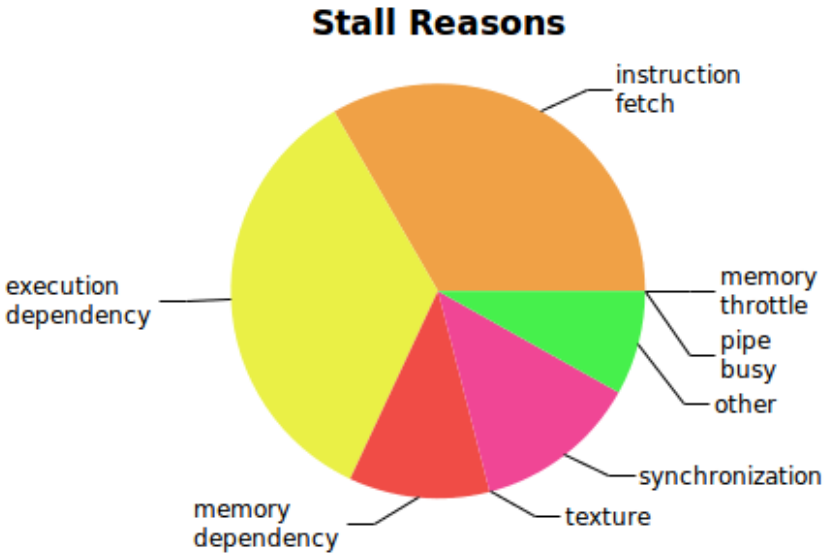

Figura 5.13 - Critérios de paragem após otimização do sincronismo no último warp no kernel de redução

	Servidor 1 GTX Titan	Servidor 1 GTX 680	Servidor 2 GTX Titan	Servidor 2 GTX 680
Versão Referência	493,757	488,193	533,566	527,430
Otimização do sincronismo	492,335	488,914	532,691	527,834
Speedup	1,003	0,999	1,002	0,999

Tabela 5.5 - Resultados da otimização do sincronismo no último warp no kernel de redução

5.5.4 Registos e ocupância do Kernel de Redução

Como mencionado, o *kernel* de redução é chamado com uma configuração de número de blocos igual ao número de partições do bloco em codificação. Acontece que em blocos de escalas mais baixas existem poucas partições pelo que este *kernel* é chamado com um baixo valor de blocos. Este baixo valor poderá levar a que não sejam utilizados todos os recursos disponíveis pelo *hardware*. Isto é, poderá acontecer não existir blocos suficientes para preencher todos os multiprocessadores da GPU e/ou esconder a latência dos acessos à memória e *fetch* das instruções.

Foi implementada uma versão onde existe apenas uma chamada do *kernel* de redução para todos os modos de predição. Deste modo, o número de blocos é multiplicado pelo número de modos de predição disponíveis. Por uma questão de localização por parte das *threads*, o *kernel* é chamada com uma configuração bidimensional (x,y). Em x mantém-se o número de partições do bloco em codificação e y regista o número de modos de predição disponíveis para o bloco atual. Utilizando o identificador *blockIdx.y*, as *threads* sabem qual o modo de predição que têm de efetuar o processamento.

Os resultados não são conclusivos (Tabela 5.6) pois existiu melhorias no desempenho nalguns cenários de testes (GTX Titan) enquanto noutros existiu uma ligeira perda de desempenho (GTX 680).

	Servidor 1 GTX Titan	Servidor 1 GTX 680	Servidor 2 GTX Titan	Servidor 2 GTX 680
Versão Referência	491,878	488,323	531,652	528,378
Otimização do nr. de blocos	483,525	489,758	523,102	528,920
Speedup	1,017	0,997	1,016	0,999

Tabela 5.6 - Resultados da otimização do número de blocos

Dentro de cada multiprocessador da GPU podem estar vários blocos de *threads* a executar pelo que os recursos são alocados para cada bloco. Devido a estas características, os limitadores da ocupância são o número de registos, tamanho da memória partilhada em uso e o número de *threads* em cada *thread-block*.

Usando a *flag* de compilação *"-Xptxas –v"* do compilador nvvc, é indicado na saída padrão a quantidade de registos e memória utilizada por cada *kernel.* O *kernel* de redução utiliza um total de 18 registos (Figura 5.14)

```
ptxas info    : Compiling entry function '_Z26cuda_jCostsBuffer_reducterPfS_PliifS_S_iPiS1_'
ptxas info    : Function properties for _Z26cuda_jCostsBuffer_reducterPfS_PliifS_S_iPiS1_
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 18 registers, 1024 bytes smem, 112 bytes cmem[0]
ptxas info    : 0 bytes gmem
ptxas info    : 0 bytes gmem
ptxas info    : 0 bytes gmem, 65064 bytes cmem[2]
```

Figura 5.14 - Resultado da compilação usando a flag "-Xptxas -v" para o kernel de redução

Foi realizado uma otimização para reduzir o número de registos requeridos pelo *kernel.* Assim, procedeu-se a uma análise cuidada do código com o objetivo de detetar registos que possam ser reutilizados entre cada passo do *kernel*. Após a alteração ao código fonte do *kernel,* os resultados mostraram que o próprio compilador *CUDA* consegue identificar estes registos pelo que se manteve o valor de registos após as alterações efetuadas.

Foi realizado ainda um teste para determinar qual o melhor valor de unidades ativas por bloco que apresente melhores resultados. O teste iniciou com o valor de 64 *threads* até ao valor de 1024 *threads* sendo que o deslocamento utilizado entre iterações corresponde ao dobro do número de unidades da iteração anterior. Esta escolha prende-se com a particularidade do terceiro passo do *kernel* de redução necessitar de um número de unidades que seja uma potência de 2 devido às consequentes divisões para metade do número de valores a reduzir.

A Tabela 5.7 mostra os resultados obtidos pelos diferentes números de *threads* no *kernel* de redução. De destacar o valor de 128 *threads* que obteve a melhor média no conjunto das duas placas e dois servidores.

	Servidor 1 GTX Titan	Servidor 1 GTX 680	Servidor 2 GTX Titan	Servidor 2 GTX 680	Média
64 *threads*	474,525	477,358	510,058	515,924	494,466
128 *threads*	470,631	476,009	508,905	514,357	492,476
256 *threads*	469,816	475,010	511,193	518,579	493,650
512 *threads*	475,780	485,772	513,112	524,074	499,685
1024 *threads*	475,780	504,451	524,497	547,260	512,997

Tabela 5.7 - Resultados do desempenho com diferentes números de threads para o kernel de redução

5.6 Otimizações da Componente híbrida entre CPU e GPU

A componente híbrida entre CPU e GPU consiste em que uma das unidades não fique parada enquanto a outra se encontra a executar trabalho. Por exemplo, após uma chamada a um *kernel* e enquanto a GPU executa o seu processamento, a CPU pode efetuar outro trabalho que não necessite do resultado do processamento do *kernel.* Analisam-se de seguida duas dessas situações.

5.6.1 Pipeline com construção da árvore de segmentação do resíduo

A segunda etapa do algoritmo CUDA-MMP consiste, para cada modo de predição, na chamada a dois *kernels* (otimização e redução) e na construção da árvore de segmentação do resíduo. Destas duas operações: uma é executada pela GPU (os dois *kernels*) e outra pela CPU (construção da árvore). Deste modo, podemos criar uma sobreposição de trabalho entre as duas componentes.

Enquanto a GPU efetua o processamento para um modo de predição, a CPU pode construir a árvore de segmentação do modo de predição anteriormente processado pela GPU. Ou seja, na primeira chamada aos *kernels* não é possível sobrepor trabalho, mas nas chamadas seguintes já é possível existir esta sobreposição. No final, a última construção de árvore de segmentação, também, não contém sobreposição de trabalho com a GPU (Figura 5.15).

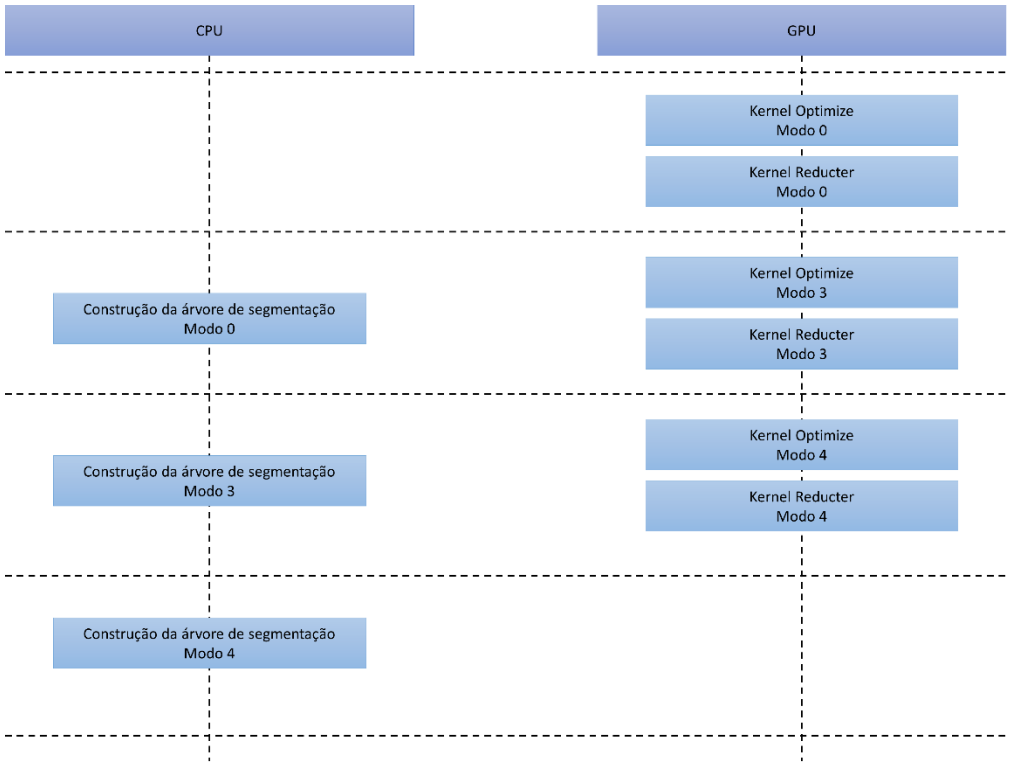

Figura 5.15 - Sobreposição de trabalho entre CPU e GPU durante a codificação dos blocos

Os resultados (Tabela 5.8) mostram que não existe a necessidade de a CPU ficar à espera que a GPU termine a sua execução e depois efetuar o seu processamento. Esta sobreposição de trabalho entre a GPU e a CPU permitiu um *speedup* médio de 1,131.

	Servidor 1 GTX Titan	Servidor 1 GTX 680	Servidor 2 GTX Titan	Servidor 2 GTX 680
Versão Referência	1773,222	1793,745	1827,911	1858,432
Pipeline Construção da Árvore Segmentação	1559,149	1613,572	1586,468	1652,056
Speedup	1,137	1,111	1,152	1,125

Tabela 5.8 - Resultados da sobreposição das construções da árvore de segmentação

5.6.2 Pipeline com cálculo do modo de predição LSP

O modo de predição *Least Square Prediction* (LSP) [45, 46] é o modo que demora mais tempo a calcular de entre os dez modos existentes no MMP. Devido aos cálculos complexos deste modo, a fatia de tempo necessária para os cálculos do mesmo corresponde a cerca de 57% do tempo total da aplicação.

Uma forma de diminuir o impacto destes cálculos passa pela sobreposição de trabalho dos cálculos deste modo de predição enquanto a GPU executa o processamento dos restantes modos de predição

(Figura 5.16). Esta implementação obriga a uma cópia extra do resíduo do modo de predição LSP e à chamada dos dois *kernels* para o processamento deste resíduo.

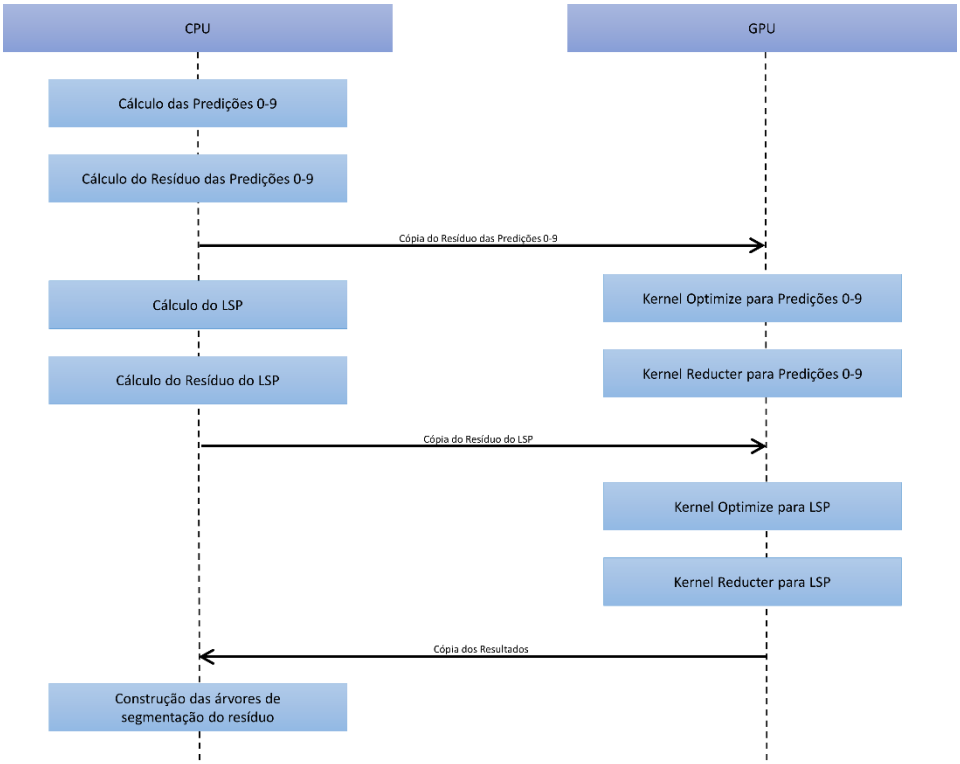

Figura 5.16 - Sobreposição do cálculo do LSP com codificação dos restantes modos de predição

Mais uma vez, é mostrado que efetuar sobreposição de trabalho entre a CPU e GPU permite aumentar o desempenho da aplicação (Tabela 5.9). Apesar das cópias e chamadas aos *kernels* extra para o modo de predição LSP, o ganho na sobreposição permite esconder o tempo perdido nas ações extra.

	Servidor 1 GTX Titan	Servidor 1 GTX 680	Servidor 2 GTX Titan	Servidor 2 GTX 680
Versão Referência	543,108	535,036	586,600	570,432
Sobreposição LSP com restantes modos	536,527	525,615	581,717	565,674
Speedup	1,012	1,018	1,008	1,008

Tabela 5.9 - Resultados da sobreposição do LSP com os restantes modos de predição

5.7 Redução de trabalho

5.7.1 Cálculo das Taxas

O *kernel* de otimização é responsável pelo cálculo dos custos Lagrangeano para cada sub-partição da partição em codificação. A equação do custo (secção 3.3) obriga ao cálculo da distorção D e da taxa R. O cálculo da distorção corresponde à maior fatia do trabalho deste *kernel*. A taxa R é calculada, também, dentro do *kernel* sendo composta pela soma de duas taxas, isto é, a taxa do segmento onde o elemento do dicionário está inserido (*rateDicSeg*) e a taxa do elemento dentro do segmento (*rateIndex*). A escolha das duas taxas deve-se a que o codificador usa menos *bits* para transmitir dois símbolos de um espaço de treino pequeno do que um símbolo de um espaço de treino grande.

Uma taxa corresponde ao valor do logaritmo da probabilidade de ocorrência do elemento ou segmento dentro de uma escala, ou seja,

$$R = -\log_2\big(P(elemento)\big).$$

O sinal negativo antes da função de logaritmo deve-se a que o logaritmo de um valor entre 0 e 1 (exclusive) devolve um valor negativo. Portanto para a taxa do elemento dentro do segmento (*rateIndex*), temos

$$rateIndex = -\log_2\left(\frac{Nr.\ vezes\ elemento\ foi\ utilizado}{Nr.total\ de\ utilizações\ de\ elementos}\right).$$

Para a taxa do segmento onde o elemento do dicionário está inserido (*rateDicSeg*), temos

$$rateDicSeg = -\log_2\left(\frac{Nr.\ vezes\ segmento\ foi\ utilizado}{Nr.total\ de\ utilizações\ de\ segmentos}\right).$$

O cálculo destas duas taxas pressupõe o acesso a quatro valores que estão na memória global, ou seja, a memória mais lenta, e o cálculo dos logaritmos.

O *kernel* de redução é responsável por determinar o elemento do dicionário com menor custo para cada sub-partição. Contudo, se dois elementos do dicionário têm o mesmo custo, o fator de desigualdade será a taxa. Como mostrado anterior, existem quatro *buffers* para a comunicação entre os dois *kernels.* Dois desses buffers contêm as duas taxas necessárias para se utilizar na condição de comparação.

De modo a otimizar os acessos à memória por parte dos dois *kernels,* foi implementada uma versão em que o *kernel* de otimização não fica responsável pelo cálculo do custo. Desta forma, este *kernel* apenas calcula a distorção entre as sub-partições e os elementos do dicionário. O *kernel* de redução fica então responsável pelo cálculo das taxas, cálculo dos custos e a redução dos mesmos e o *kernel* de otimização apenas calcula as distorções. Esta decisão prendeu-se com a necessidade dos valores da taxa no *kernel* de redução na condição de comparação pelo que obrigatoriamente este *kernel* necessitava destes valores. Desta forma, foram removidos os dois *buffers* de comunicação criados para guardar os valores das taxas (Figura 5.17). Nesta versão foram criados ainda três novos vetores de memória partilhada que guardam os menores custos e as taxas que são escritos no final da segunda etapa do *kernel.* O objetivo é que no terceiro passo do *kernel* de redução não exista repetição do cálculo dos mesmos.

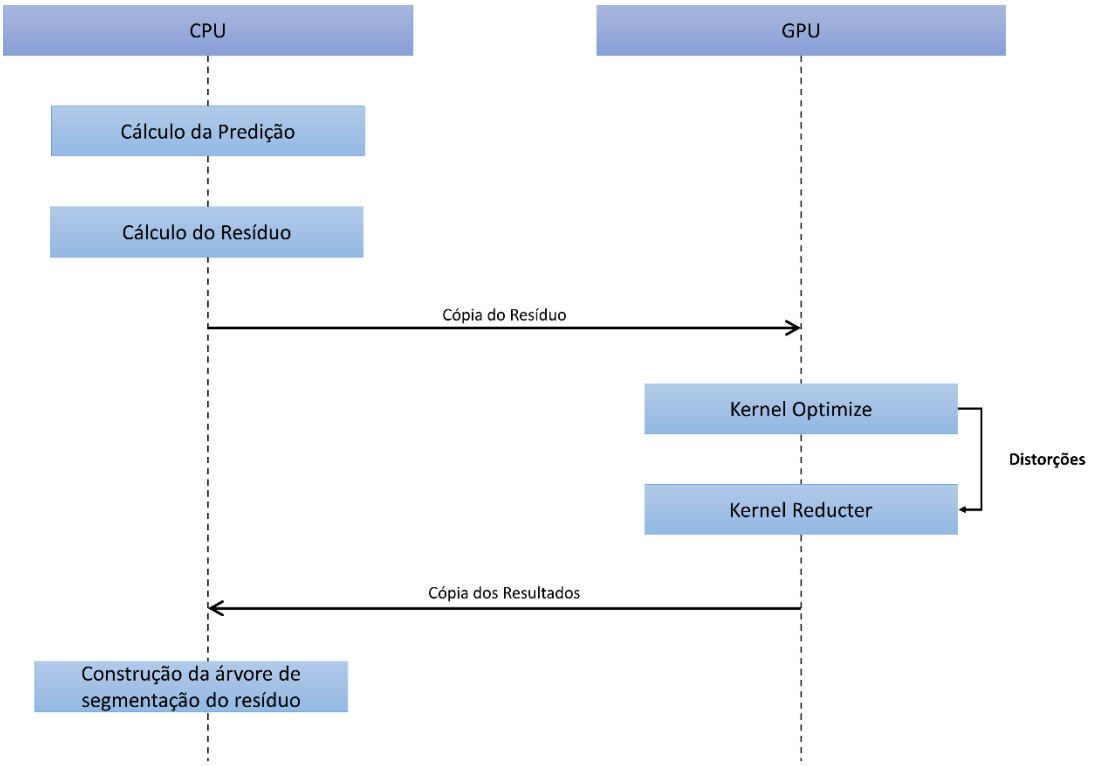

Figura 5.17 - Novo esquema do algoritmo CUDA-MMP

Esta alteração da distribuição de trabalho pelos *kernels* provocou uma perda de desempenho (Tabela 5.10).

	Servidor 1 GTX Titan	Servidor 1 GTX 680	Servidor 2 GTX Titan	Servidor 2 GTX 680
Versão Referência	903,495	1051,600	975,361	1125,670
Alteração do cálculo das taxas para o *kernel* de redução	980,431	1243,448	1063,556	1340,589
Speedup	0,922	0,846	0,917	0,840

Tabela 5.10 - Resultados da alteração do cálculo das taxas do kernel de otimização para o kernel de redução

Um *profiling* sobre o desempenho do *kernel de redução* demonstra que a nova organização aumentou a dependência de instruções (de Figura 5.18 para Figura 5.19). Como mencionado anteriormente, para calcular os dois tipos de taxas, são necessários vários acessos à memória global (a memória mais lenta), e ainda o cálculo dos logaritmos. Estas operações demoram vários ciclos de relógio, o que não permite a execução passar às próximas instruções. Deste modo, esta espera por parte da execução tem um impacto significativo no desempenho. Devido ao efeito negativo no desempenho, esta otimização foi descartada.

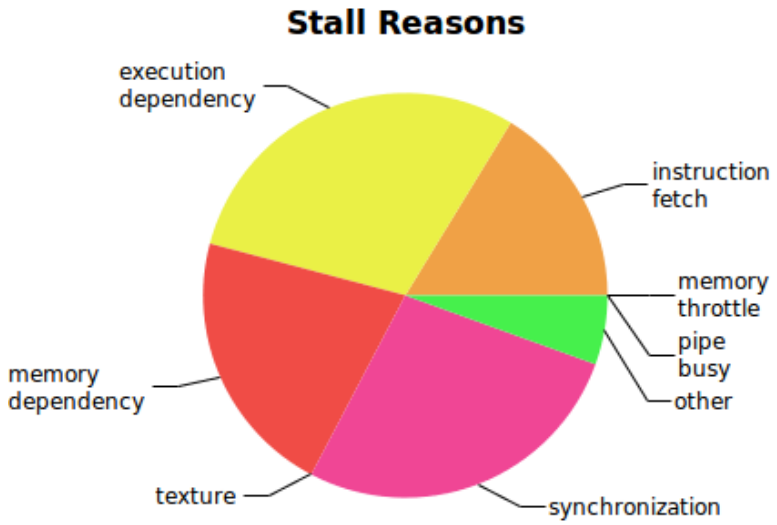

Figura 5.18 - Condições de paragem do kernel de redução da versão de referência

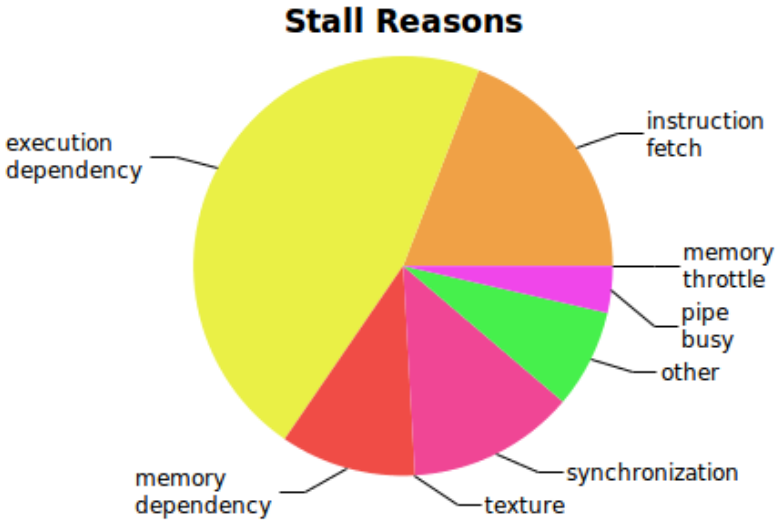

Figura 5.19 Condições de paragem do kernel de redução com cálculo das taxas

Durante a codificação de um macro-bloco da imagem, o valor das taxas mantém-se inalterado. Isto porque estes valores apenas sofrem alterações após a codificação do macro-bloco pois os blocos que forem escolhidos na codificação terão o valor de taxa diminuído. Desta forma, não existe a necessidade para cada chamada ao *kernel* de redução de serem calculadas as taxas para cada elemento do dicionário. A solução para este problema consiste no cálculo de todas as taxas antes do início da codificação de cada macro-bloco por parte da CPU e na cópia destes valores para a GPU. Com esta implementação, o *kernel* de redução agora acede a dois valores na memória global: um valor para a taxa do segmento do elemento no dicionário e outro valor para a taxa do elemento dentro do segmento. Este comportamento já ocorria na versão de referência mas nesta nova versão as taxas são calculadas somente uma vez pela CPU, não existindo este cálculo na GPU.

Na Tabela 5.11 podemos verificar que existiu uma ligeira melhoria no desempenho da codificação.

	Servidor 1 GTX Titan	Servidor 1 GTX 680	Servidor 2 GTX Titan	Servidor 2 GTX 680
Versão Referência	903,495	1051,600	975,361	1125,670
Alteração do cálculo das taxas para a CPU	855,887	969,300	970,994	1050,632
Speedup	1,056	1.085	1,004	1,071

Tabela 5.11 - Resultados da alteração do cálculo das taxas para a CPU

Visto o valor das taxas já se encontrarem em memória na CPU, deixou de existir a necessidade de se proceder à cópia das taxas nos resultados do *kernel* de redução para se realizar a construção da árvore de segmentação do resíduo.

Esta diminuição do tamanho de memória a copiar faz com que se use menos *throughput* que passou de 2,66GB/s para 1,74GB/s, mas com benefícios para o desempenho da aplicação que melhorou o seu desempenho (Tabela 5.12).

	Servidor 1 GTX Titan	Servidor 1 GTX 680	Servidor 2 GTX Titan	Servidor 2 GTX 680
Alteração do cálculo das taxas para a CPU	855,887	969,300	970,994	1050,632
Remoção das taxas como resultado	857,233	957,600	949,538	1017,399
Speedup	0,998	1,012	1,023	1,033

Tabela 5.12 - Resultados da remoção dos taxas como resultado

5.7.2 Escala zero

A escala 0 é um caso especial no algoritmo MMP. Por apenas corresponder a blocos com um píxel, o número total de elementos necessários do dicionário para classificar todas as partições desta escala é reduzido e conhecido. Devido a esta particularidade, o dicionário nesta escala encontra-se totalmente preenchido, existindo um elemento no dicionário para cada valor do intervalo de valores do resíduo, isto é, [-255;255] (secção 3.4).

Na codificação de uma partição que pertence à escala 0, o elemento do dicionário que melhor representa essa partição é o elemento cujo valor é igual ao valor do resíduo, pois este apresenta uma distorção igual a zero. Desta forma, não existe a necessidade do *kernel* de otimização percorrer todos

os elementos do dicionário da escala 0 para calcular a respetiva distorção. Além disso, posteriormente o *kernel* de redução teria de calcular os custos para cada elemento e realizar a respetiva redução.

A título de exemplo, na codificação de um bloco de escala 24 (16x16), existem 256 partições da escala 0. O que equivale a 131072 (256*512) ciclos de *threads* no *kernel* de otimização para calcular as distorções apenas das partições da escala 0. No *kernel* de redução seriam necessário 256 *thread-blocks* para realizar as reduções de todas as partições desta escala. No final, seria necessária a cópia de 768 (256*3) valores para se utilizar na construção da árvore de segmentação. Nas escalas seguintes estes valores são menores, mas o seu impacto continua a ser relevante.

A codificação da escala 0 passou a ser realizada pela CPU que apenas necessita de fazer um mapeamento direto entre o valor do resíduo e o índice do elemento do dicionário. Os resultados decorrentes de libertar a GPU do cálculo da escala zero mostram uma ligeira melhoria do desempenho (Tabela 5.13).

	Servidor 1 GTX Titan	Servidor 1 GTX 680	Servidor 2 GTX Titan	Servidor 2 GTX 680
Versão Referência	548,923	538,917	589,996	578,347
Remoção da escala zero	543,108	535,036	586,600	570,432
Speedup	1,011	1,007	1,006	1,014

Tabela 5.13 - Resultados com a remoção da escala zero

5.8 Otimização da atualização do dicionário

Como já foi mencionado, o MMP utiliza um dicionário adaptativo que vai crescendo com a codificação da imagem. Esta ação permite aprender novos padrões que estão presentes na imagem e assim alcançar uma melhor codificação com informação útil.

A atualização do dicionário é realizada tanto pelo codificador como pelo descodificador. No codificador ocorre após a codificação de cada macro-bloco onde são realizadas transformações sob os diversos nós finais da árvore de segmentação da predição. No descodificador também é replicado o dicionário, ou seja, através da informação dos índices dos nós finais consegue-se realizar as mesmas transformações que o codificador. Deste modo, não é necessário transmitir o dicionário para o descodificador.

5.8.1 Objetivos

Como mostrado na secção 5.2, a atualização do dicionário corresponde a uma das operações com maior impacto no desempenho da aplicação MMP. Isto deve-se à pesquisa que é realizada sob o dicionário no controlo de redundância. Ou seja, antes de se adicionar um novo elemento no dicionário é verificado se existe algum elemento que seja igual ou parecido. O critério de parecença depende de um parâmetro de entrada (*radius distortion*) do algoritmo. Este parâmetro define uma esfera e caso a diferença entre o novo elemento e um elemento existente no dicionário (*distortion)* esteja contida dentro da esfera, então este novo elemento não é adicionado ao dicionário.

Com esta implementação pretende-se a criação de um *kernel* que efetue a pesquisa no dicionário e no final indique se o elemento pode ser adicionado ao dicionário. O ganho desta implementação ganha maior foco quando é ativada a denominada super-atualização do dicionário, isto é, quando todas as transformações estão ativas. De facto, como existem mais transformações, então existem mais elementos para se realizar o controlo de redundância.

De notar, que como esta ação é realizada pelo codificador e descodificador, as possíveis melhorias no desempenho obtido pelo *kernel* beneficiam ambas as aplicações.

5.8.2 Implementação

Visto a árvore de segmentação da predição ser percorrida de baixo para cima, então existe uma dependência dos blocos que vão sendo concatenados dos níveis inferiores (secção 3.3 na Figura 3.3). Devido a esta característica, o controlo de redundância têm de ser realizado em cada nó da árvore de segmentação levando a que o novo *kernel* tivesse perca de computação por chamada. Para evitar esse ineficiente cenário, optou-se por uma implementação híbrida entre CPU e GPU.

A CPU realiza o controlo dos blocos que provém das concatenações de blocos e a GPU realiza o controlo dos blocos que provém das transformações realizadas sob os blocos concatenados. Partindo da Figura 3.3 que explica a concatenação dos blocos, a CPU realiza o controlo da redundância dos blocos a laranja e a GPU realiza o controlo dos blocos originados pelas transformações aos blocos laranja.

O trabalho realizado pela CPU manteve-se da versão original. O comportamento dos blocos que são processados pela GPU foi alterado, isto é, são guardados em memória e, no final, realiza-se uma cópia de todos os blocos a verificar para a GPU aproveitando-se de melhor forma a largura de banda entre CPU e GPU. O processamento destes blocos é efetuado pelo *kernel* denominado *cuda_compare_blocks,* sendo no final copiados os resultados de volta para a CPU. A CPU fica encarregue de percorrer os resultados e verificar os blocos que são para adicionar ao dicionário. Estes novos blocos do dicionário são guardados em memória em conjunto com o seu índice que serão depois tratados pelo *kernel cuda_update_dic,* responsável pela escrita dos novos blocos no dicionário da GPU.

5.8.2.1 Kernel de comparação de blocos

O *kernel* implementado consiste em dois passos. No primeiro passo é realizado o cálculo da distorção entre os elementos do dicionário e o bloco em verificação. Este passo verifica ainda se a distorção se encontra dentro da esfera de verificação, isto é, a esfera construída pelo parâmetro *Radius Distortion.* Para se manter o comportamento da versão sequencial que termina a busca quando encontra um elemento parecido, é necessário proceder a uma redução dos elementos do dicionário com o qual o bloco em codificação é parecido. Desta forma, obtém-se o elemento do dicionário com menor índice que corresponde ao elemento no qual a versão sequencial termina a sua busca.

Cada bloco de *threads* fica responsável pela verificação de um bloco. No primeiro passo, cada *thread* calcula a distorção entre um elemento do dicionário e o bloco em verificação. De seguida, verifica se a distorção se encontra dentro da esfera referente ao parâmetro *Radius Distortion*. Se a distorção se encontra dentro da esfera, é guardado o índice do elemento do dicionário e esta *thread* pode terminar a execução do primeiro passo. Caso contrário, a *thread* passa para o elemento do dicionário seguinte de acordo com o deslocamento (número de *threads* por bloco). No final deste passo, cada *thread* escreve o índice que encontrou numa memória partilhada. Se não encontrou nenhum elemento então é escrito o valor de omissão (INT_MAX). Por fim, é feito um sincronismo no final da escrita para garantir

que todos os valores foram efetivamente escritos. No segundo passo é realizada uma redução dos valores escritos no primeiro passo. É usada a técnica de *Sequential Addressing* [43] (igual à realizada pelo *kernel* de redução na secção 5.4.2) para obter o índice com menor valor. Caso nenhum elemento do dicionário seja igual/parecido ao bloco em verificação então no final da redução obtém-se o valor de omissão, o que significa que este bloco pode ser adicionado ao dicionário. Este valor é escrito numa memória global que no final da execução do *kernel* é copiada para a CPU.

5.8.3 Resultados

Foram implementadas três versões deste *kernel* que diferem no tipo de memória empregue para guardar os blocos a comparar. A primeira versão guarda os blocos na memória global, a segunda mantém os blocos na memória das constantes e na terceira os blocos estão na memória global mas são carregados para a memória partilhada.

5.8.3.1 Novos blocos na *memória global*

A memória global é, conforme anteriormente referido, a memória com maior espaço pelo que seria a única onde caberiam todos os novos blocos a comparar. A desvantagem desta memória prende-se com a sua relativamente lenta velocidade de acesso. Para os parâmetros de referência e para a imagem Lena, este *kernel* é invocado 964 vezes e totaliza um total de 1,75s de tempo de execução.

Com esta versão foi conseguido um speedup médio de 1,230 (Tabela 5.14). De notar que quantas mais transformações estejam ativas, mais blocos são necessários para realizar o controlo de redundância pelo que os ganhos em termos de *speedup* serão ainda maiores.

	Servidor 1 GTX Titan	Servidor 1 GTX 680	Servidor 2 GTX Titan	Servidor 2 GTX 680
Versão Referência	663,948	651,333	726,243	738,744
Novos blocos na memória global	549,144	539,754	589,983	581,100
Speedup	1,209	1,207	1,231	1,271

Tabela 5.14 - Resultados com os novos blocos na memória global

5.8.3.2 Novos blocos na *memória das constantes*

Como mencionado na versão anterior, apenas a memória global contém espaço suficiente para guardar todos os blocos a realizar o controlo de redundância. Portanto uma versão assente na memória das constantes obriga a múltiplas cópias e múltiplas chamadas ao *kernel* de comparação dos blocos. A memória das constantes está limitada a 64 KB [47] e visto esta memória guardar outros dados necessários, então apenas cabem 63 blocos em memória para cada execução do *kernel*. Importa notar que segundo a documentação CUDA [47], se todas as *threads* dentro do mesmo *warp* acederem ao mesmo local de memória das constantes, então este acesso consegue ser tão rápido como se tratasse de um acesso a um registo. Visto que dentro do mesmo *warp* todas as unidades ativas se encontram a executar a mesma instrução, então todas estarão a incidir sobre a mesma posição do bloco em comparação que se encontra na memória das constantes. O que difere entre cada *thread* é o índice do dicionário com o qual é feita a comparação.

Na versão anterior, para a imagem Lena com λ=10 e os demais parâmetros com os valores de referência, o *kernel* é chamado 962 vezes. Enquanto, com os mesmos parâmetros, para esta versão o kernel é chamado 6345 vezes. Em relação ao tempo total de computação, para a versão anterior (memória global), a GPU demora cerca de 1,75s para as 962 chamadas enquanto na versão com a memória das constantes, a GPU apenas demora 0,53s para as 6345 chamadas. Desta forma, concluímos que usando a memória das constantes, o *kernel* demora muito menos tempo, mas a necessidade de realizar mais cópias e o *overhead* associado às chamadas das cópias e lançamento dos *kernels* impede a obtenção de melhores resultados.

O uso deste tipo de memória permitiu um *speedup* médio de 1,233 vezes (Tabela 5.15), ligeiramente superior que o valor obtido usando a memória global.

	Servidor 1 GTX Titan	Servidor 1 GTX 680	Servidor 2 GTX Titan	Servidor 2 GTX 680
Versão Referência	663,948	651,333	726,243	738,744
Novos blocos na memória das constantes	548,077	539,896	588,336	578,107
Speedup	1,211	1,206	1,234	1,278

Tabela 5.15 - Resultados com os novos blocos na memória das constantes

5.8.3.3 Novos blocos na memória global com carregamento para memória partilhada

Esta versão, carrega, no início do *kernel*, o novo bloco para a memória partilhada. Deste modo todas as *threads* têm acesso aos valores do bloco numa memória mais rápida que a memória global. Outra razão para a escolha do uso da memória partilhada deve-se ao facto de esta memória permitir *broadcast* de valores [48].

Por uma questão de otimização, nesta versão são utilizadas 256 *threads* para que o carregamento inicial do bloco para a memória partilhada seja realizado com o uso do ID da *thread* dentro do bloco e para não se necessitar de ciclos para percorrer todos os valores, pois o número máximo de valores é 256, correspondendo a 16x16.

Com o uso da memória partilhada, o tempo de execução deste *kernel* representa 1,43s para um número de invocações igual a 968. Em termos de tempo total da aplicação de codificação foi conseguido um *speedup* médio de 1,231 vezes. Este valor é superior ao obtido pelo uso da memória global mas menor que o conseguido com o uso da memória das constantes.

	Servidor 1 GTX Titan	Servidor 1 GTX 680	Servidor 2 GTX Titan	Servidor 2 GTX 680
Versão Referência	663,948	651,333	726,243	738,744
Novos blocos na memória das constantes	548,175	538,917	589,996	580,116
Speedup	1,211	1,209	1,231	1,273

Tabela 5.16 - Resultados com os novos blocos na memória partilhada

5.8.3.4 Conclusões

Com os resultados obtidos podemos concluir que a alteração do controlo de redundância para uma solução híbrida permitiu ganhos na ordem dos 20% para a aplicação de codificação. Desta forma, a CPU apenas efetua o trabalho necessário, enquanto a GPU fica com o restante trabalho que consegue efetuar de uma forma paralela e com maior velocidade.

Como o *kernel* de comparação corresponde a uma pequena fração do tempo total da codificação, cerca de 0,30%, então a sua relevância sob o tempo total da codificação é praticamente nula. Portanto, o tipo de memória com o qual é utilizada para guardar os blocos relevou-se pouco significativo. Contudo, a memória das constantes relevou-se ser muito mais rápida que os restantes tipos de memória (Tabela 5.17).

Tipo de Memória	Tempo (s)	Nr. Chamadas	Média (ms)	Speedup Médio
Global	1,75	964	0,182	1,230
Constantes	0,53	6345	0,008	1,233
Partilhada	1,43	968	0,146	1,231

Tabela 5.17 - Tempo do kernel de comparação por tipo de memória

Visto, nos parâmetros de referência não se utilizar o máximo de transformações possíveis, caso todas as transformações forem ativadas obrigaria que o *kernel* com o uso de memória das constantes aumentasse ainda mais o número de invocações e cópias dos blocos. Enquanto os restantes se mantinham. Desta forma, é expectável que o uso das memórias das constantes possa ter pior desempenho quando se utilizam mais transformações.

Como referido anteriormente, esta alteração da atualização do dicionário não só afetaria o tempo de execução do codificador bem como do descodificador. A Tabela 5.18 mostra os resultados obtidos por esta alteração no tempo de execução do descodificador. De notar que o *speedup* obtido é superior ao obtido pelo codificador, isto porque esta era a principal função consumidora de tempo no descodificador.

Descodificador	Servidor 1 GTX Titan	Servidor 2 GTX 680
Versão Referência	36,126	37,880
Nova atualização do dicionário	20,876	24,136
Speedup	1,731	1,570

Tabela 5.18 - Resultados da nova atualização do dicionário no descodificador

5.9 Versões Sub-ótimas

São consideradas versões sub-ótimas as implementações que não sigam o normal comportamento do algoritmo, resultando numa degradação da qualidade de compressão. Contudo, esta degradação se for ligeira e permitir obter *speedups* significativos, poderá ser uma versão a ter em conta.

5.9.1 Estudo do modo de predição LSP

Como já referenciado na secção 5.6.2, o cálculo do modo de predição LSP é uma das operações que mais tempo consome no decorrer da codificação das imagens. Durante este capítulo, os resultados apresentados são referentes à codificação da imagem Barbara com o valor de λ=10. A escolha desta imagem deve-se ao facto de no conjunto de imagens de testes, esta ser a que apresenta maior dependência do modo de predição LSP. A razão para isso assenta sob as texturas presentes na imagem que têm direções que variam muito e neste tipo de texturas o LSP efetua uma predição mais eficiente do que os modos direcionais tradicionais.

Como podemos verificar na Tabela 5.19, o tempo necessário para o cálculo do modo de predição LSP corresponde a mais de 80% do tempo total da codificação da imagem. Deste forma esta operação corresponde à maior fatia de tempo da aplicação.

	Tempo (s)	Tempo LSP (s)	% LSP
Servidor 1 GTX Titan	476,136	383,724	80,59 %
Servidor 1 GTX 680	478,473	383,598	80,17 %

Tabela 5.19 - Tempo de cálculos do modo LSP para imagem Barbara (λ=10)

Os complexos cálculos associados a este modo [45] e dependência dos resultados entre pixéis [46] tornam a portabilidade deste problema para um paradigma multi-núcleo muito difícil. Ou seja, para o cálculo deste modo num bloco de escala 15 (bloco 8x8) é necessário efetuar o treino dos 64 pixéis, o que obrigaria à cópia da área de treino, à chamada do *kernel* que fosse implementado e ainda a cópia dos resultados para cada píxel. Devido à dependência de resultados dos pixéis anteriores, não é possível transformar esta operação para apenas uma chamada a um *kernel.*

A Figura 5.20 e Tabela 5.20 mostram a utilização do modo de predição durante a codificação da imagem Barbara com λ=10 por escala. Da escala 0 à 7, este modo não foi aplicado devido ao parâmetro

de entrada que indica até qual escala existe segmentação da predição. Na escala 24, este modo também não foi aplicado pois nesta escala são utilizados diferentes modos de predição.

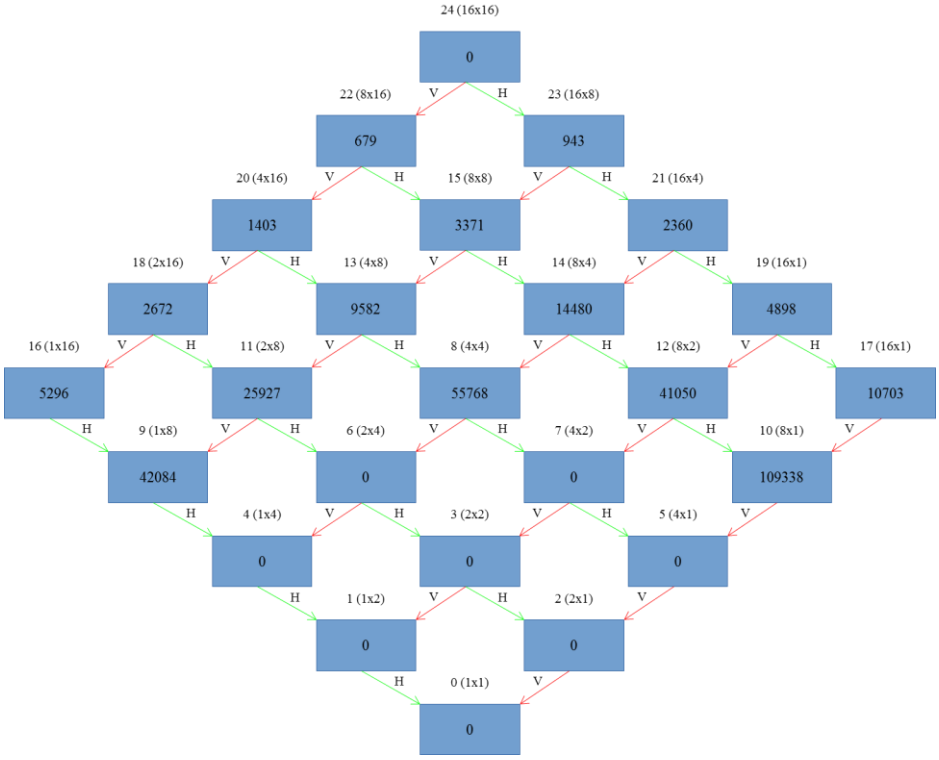

Figura 5.20 - Utilização do modo de predição LSP por escala (diagrama)

Escala	# Calculado	# Escolhido	% Escolha	% Escolha Total
0	0	0	0,00%	0,00%
1	0	0	0,00%	0,00%
2	0	0	0,00%	0,00%
3	0	0	0,00%	0,00%
4	0	0	0,00%	0,00%
5	0	0	0,00%	0,00%
6	0	0	0,00%	0,00%
7	0	0	0,00%	0,00%
8	93000	55768	59,97%	16,87%
9	153760	42084	27,37%	12,73%
10	155651	109338	70,25%	33,08%
11	61504	25927	42,15%	7,84%
12	62248	41050	65,95%	12,42%
13	23064	9582	41,55%	2,90%
14	23250	14480	62,28%	4,38%
15	7688	3371	43,85%	1,02%
16	15376	5296	34,44%	1,60%
17	15562	10703	68,78%	3,24%
18	7688	2672	34,76%	0,81%
19	7781	4898	62,95%	1,48%
20	3844	1403	36,50%	0,42%
21	3875	2360	60,90%	0,71%
22	1922	679	35,33%	0,21%
23	1922	943	49,06%	0,29%
24	0	0	0,00%	0,00%
Total:	**638135**	**330554**	**51,80%**	**100,00%**

Tabela 5.20 - Utilização do modo de predição LSP por escala

5.9.1.1 Aplicação *do LSP às escalas "finas"*

Como mencionado anteriormente, o cálculo do LSP é realizado por píxel e este depende do resultado do cálculo dos pixéis vizinhos. Sendo o LSP um modo de predição, então este não se pode considerar que o resultado deste seja de acordo com a realidade. Portanto à medida que se prolonga esta dependência de valores, também se prolonga o erro que a predição vai gerando. Esta propagação do erro é mais suscetível em escala de maiores dimensões, portanto uma forma de diminuir este erro seria aplicar o modo de predição apenas às escalas de menores dimensões.

Foram realizados quatros testes onde em cada teste se aumentava o número de escalas onde o modo de predição LSP estava ativo: (1) escalas cuja uma das dimensões tenha o tamanho de 1 píxel, (2) escalas cuja uma das dimensões seja menor ou igual que 2 pixéis, (3) escalas cuja uma das dimensões seja menor ou igual que 4 pixéis e, por fim, (4) escalas cuja uma das dimensões seja menor ou igual que 8 pixéis. A próxima iteração seria com 16 pixéis o que corresponde a ativar todas as escalas.

Usando os valores da Tabela 5.20, podemos ter uma noção do número de blocos que serão afetados em cada teste (Tabela 5.21), pelo que no quarto teste se atinge o total dos blocos. A Tabela 5.22 mostra os resultados dos tempos para cada teste. Os resultados revelaram ser o esperado, ou seja, quanto mais escalas o LSP estiver ativo, maior é o tempo de computação.

Pixéis	Percentagem
1	50.65%
<= 2	73.20%
<= 4	98.49%
<= 8	100.00%

Tabela 5.21 - Utilização do modo de predição LSP por dimensão das escalas

		Tempo Referência	Tempo	Speedup	Tempo LSP (%)
1 píxel	**Servidor 1 GTX Titan**	476,136	227,313	2,095	142,550 (62,71%)
	Servidor 1 GTX 680	478,473	230,081	2,080	142,624 (61,99%)
2 pixéis	**Servidor 1 GTX Titan**	476,136	325,299	1,464	237,534 (73,02%)
	Servidor 1 GTX 680	478,473	326,607	1,465	237,252 (72,64%)
4 pixéis	**Servidor 1 GTX Titan**	476,136	424,196	1,122	334,803 (78,93%)
	Servidor 1 GTX 680	478,473	426,837	1,121	334,303 (78,32%)
8 pixéis	**Servidor 1 GTX Titan**	476,136	466,289	1,021	374,543 (80,32%)
	Servidor 1 GTX 680	478,473	468,500	1,021	374,393 (79,91%)

Tabela 5.22 - Resultados dos testes do LSP por dimensão de escalas

Apesar de o teste com as escalas de 1 píxel ter sido mais rápido que as restantes, por se tratar de uma versão sub-ótima é necessário validar as curvas RD para se verificar se houve perda de qualidade na

compressão dos dados. As Figura 5.21, Figura 5.22, Figura 5.23 e Figura 5.24 mostram as curvas RD para os quatro testes efetuados, respetivamente. Após o teste onde são usadas as escalas com uma dimensão menor ou igual a 4 pixéis é que as curvas coincidem. Esta versão obteve um *speedup* médio de 1,122 e foi validado pelas curvas RD pelo que se poderá ter em consideração como uma versão para produção.

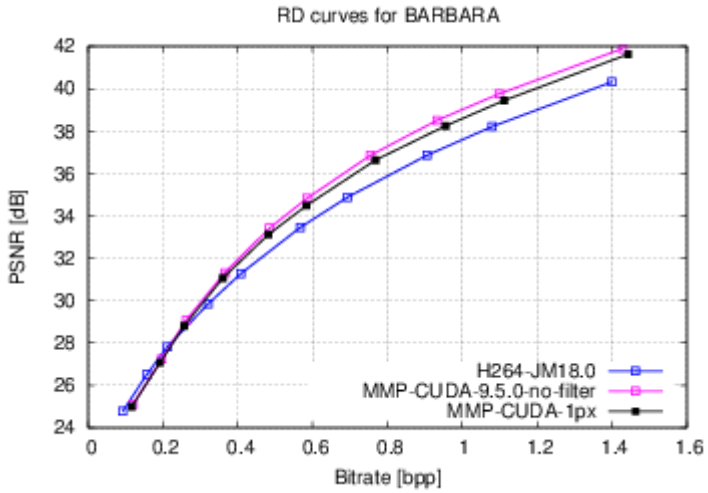

Figura 5.21 - Curva RD para escalas com 1 píxel

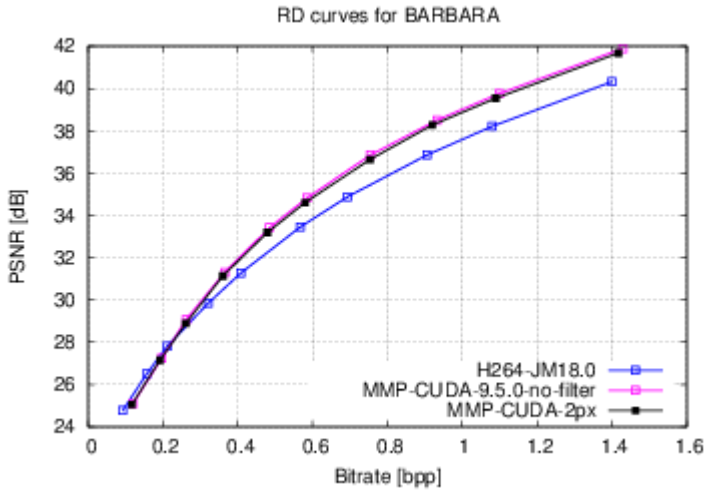

Figura 5.22 - Curva RD para escalas com 2 pixéis

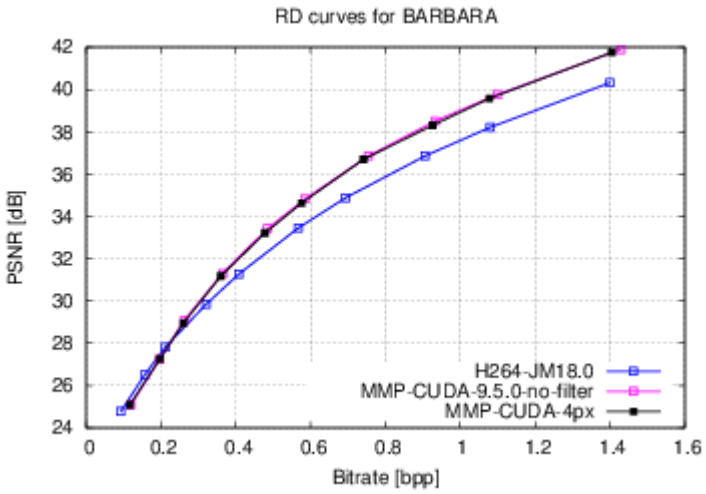

Figura 5.23 - Curva RD para escalas com 4 pixéis

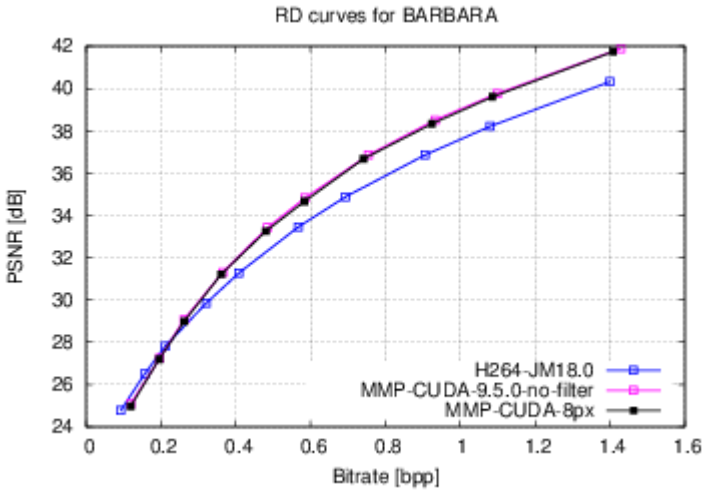

Figura 5.24 - Curva RD para escalas com 8 pixéis

5.9.1.2 Aplicação do LSP às escalas mais utilizadas

Uma análise aos resultados da Tabela 5.20 permitem identificar que acumulando as escalas 8 a 12, conseguimos englobar cerca de 83% dos blocos onde este modo de predição foi escolhido. As escalas 13, 14, 16 e 17 perfazem cerca de 12% dos blocos escolhidos e as restantes escalas (15 e 18 a 23), os restantes 5% (Tabela 5.23).

Escala	# Calculado	# Escolhido	% Escolha	% Escolha Total	Acum
0	0	0	0,00%	0,00%	0,00%
1	0	0	0,00%	0,00%	
2	0	0	0,00%	0,00%	
3	0	0	0,00%	0,00%	
4	0	0	0,00%	0,00%	
5	0	0	0,00%	0,00%	
6	0	0	0,00%	0,00%	
7	0	0	0,00%	0,00%	
8	93000	55768	59,97%	16,87%	82,94%
9	153760	42084	27,37%	12,73%	
10	155651	109338	70,25%	33,08%	
11	61504	25927	42,15%	7,84%	
12	62248	41050	65,95%	12,42%	
13	23064	9582	41,55%	2,90%	12,12%
14	23250	14480	62,28%	4,38%	
15	7688	3371	43,85%	1,02%	
16	15376	5296	34,44%	1,60%	
17	15562	10703	68,78%	3,24%	
18	7688	2672	34,76%	0,81%	4,94%
19	7781	4898	62,95%	1,48%	
20	3844	1403	36,50%	0,42%	
21	3875	2360	60,90%	0,71%	
22	1922	679	35,33%	0,21%	
23	1922	943	49,06%	0,29%	
24	0	0	0,00%	0,00%	0,00%
Total:	**638135**	**330554**	**51,80%**	**100,00%**	100,00%

Tabela 5.23 - Acumulação dos blocos escolhidos com o modo LSP por escalas mais utilizadas

Desta forma foram implementadas duas versões, onde em cada uma se aumenta o número de escalas onde o LSP está ativo. A primeira versão terá as escalas ativas com a cor laranja e a segunda com as escalas a laranja e cinza da Figura 5.25.

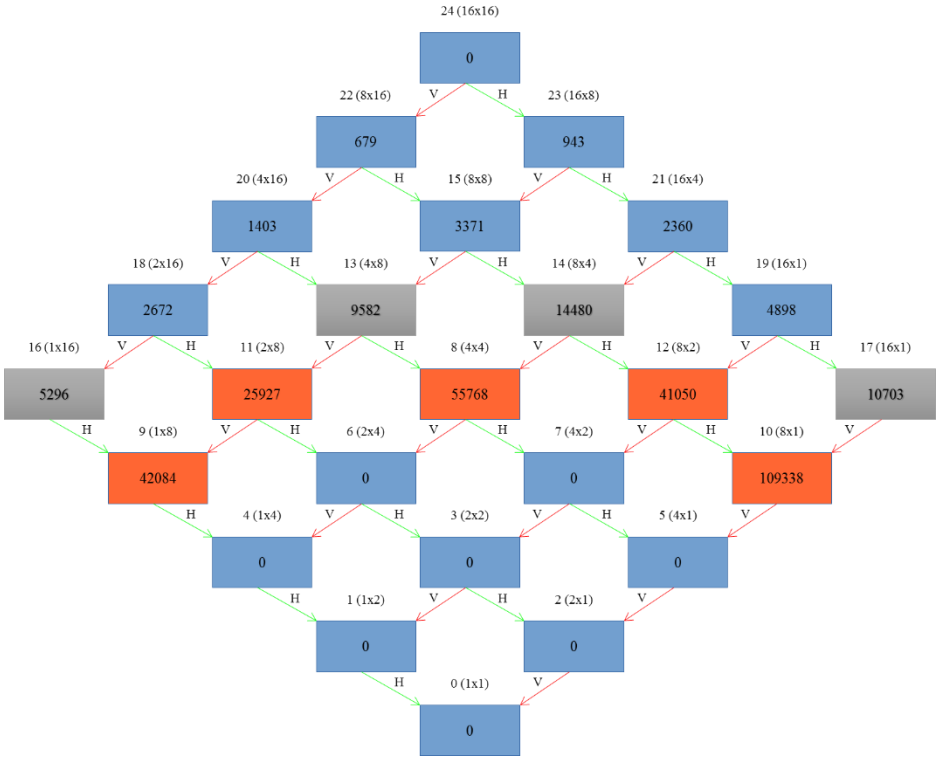

Figura 5.25 - Aplicação do LSP por escalas mais utilizados

Mais uma vez os resultados são como esperado, quanto menos escalas estiverem ativas para realizar os cálculos do LSP, mais rápida é a execução da aplicação (Tabela 5.24).

		Tempo Referência	Tempo	Speedup	Tempo LSP (%)
Versão com escalas 8, 9, 10, 11 e 12	**Servidor 1 GTX Titan**	476,136	209,228	2,276	123,974 (59,25%)
	Servidor 1 GTX 680	478,473	212,508	2,252	123,873 (58,29%)
Versão com escalas 8, 9, 10, 11, 12, 13, 14, 16 e 17	**Servidor 1 GTX Titan**	476,136	359,035	1,326	269,306 (75,01%)
	Servidor 1 GTX 680	478,473	362,010	1,322	269,191 (74,36%)

Tabela 5.24 - Resultados da aplicação do LSP por escalas mais utilizados

Mais uma vez, visto se tratar de versões sub-ótimas é necessário proceder à sua validação com recurso às curvas RD. As curvas RD (Figura 5.26 e Figura 5.27) mostram que existiu degradação significativa da qualidade de compressão pelo que apesar dos *speedups* conseguidos, estas duas versões têm de ser descartadas.

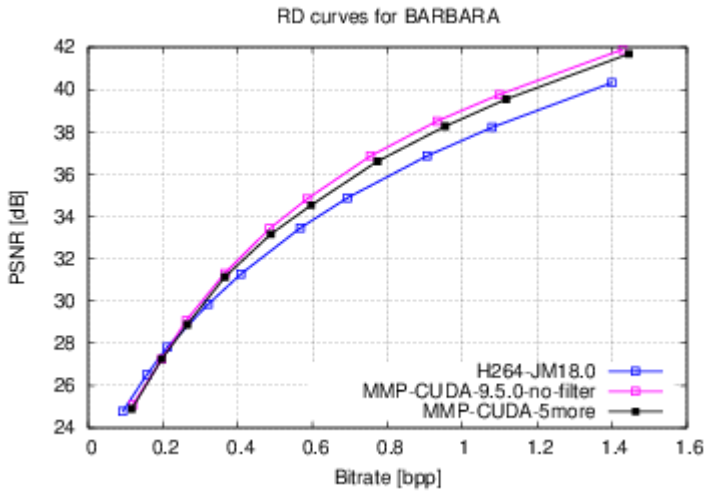

Figura 5.26 – Curva RD da aplicação do LSP às cinco escalas mais utilizadas

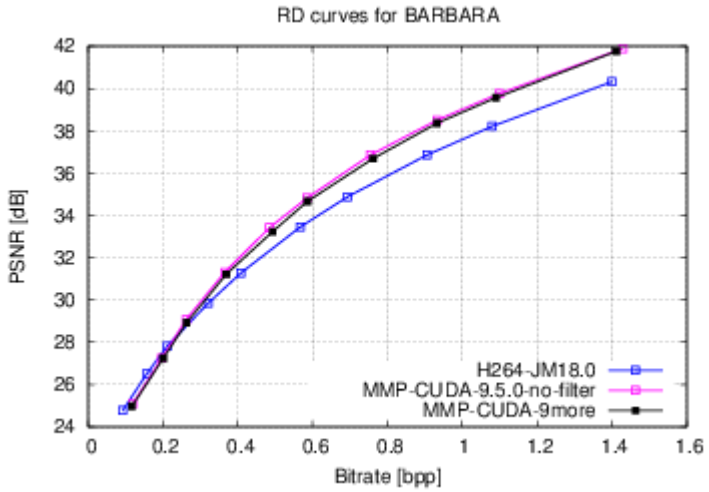

Figura 5.27 – Curva RD da aplicação do LSP às nove escalas mais utilizadas

5.9.1.3 Aplicação do LSP em escalas alternadas

Esta versão não tem como fundamento o estudo da utilização do modo de predição LSP da Tabela 5.19, mas tem como objetivo partir em cerca de metade a utilização do LSP mas sem a rigidez por escalas como nas versões anteriores. Nesta implementação, após a primeira utilização do LSP num determinado bloco, a utilização deste modo será feita em modo alternado entre gerações. Isto é, num determinado bloco A o LSP é aplicado mas aos blocos filhos do bloco A não será aplicado, contudo nos blocos "netos" do bloco A o modo LSP será aplicado.

De uma forma teórica, usando o estudo de referência (Tabela 5.19), esta versão conseguiria uma percentagem de blocos escolhidos de 55,87% (escalas a laranja na Tabela 5.25).

Escala	# Calculado	# Escolhido	% Escolha	% Escolha Total	Acum
0	0	0	0,00%	0,00%	
1	0	0	0,00%	0,00%	
2	0	0	0,00%	0,00%	
3	0	0	0,00%	0,00%	
4	0	0	0,00%	0,00%	
5	0	0	0,00%	0,00%	
6	0	0	0,00%	0,00%	
7	0	0	0,00%	0,00%	
8	93000	55768	59,97%	16,87%	55,87%
9	153760	42084	27,37%	12,73%	
10	155651	109338	70,25%	33,08%	
11	61504	25927	42,15%	7,84%	
12	62248	41050	65,95%	12,42%	
13	23064	9582	41,55%	2,90%	
14	23250	14480	62,28%	4,38%	
15	7688	3371	43,85%	1,02%	
16	15376	5296	34,44%	1,60%	
17	15562	10703	68,78%	3,24%	
18	7688	2672	34,76%	0,81%	
19	7781	4898	62,95%	1,48%	44,13%
20	3844	1403	36,50%	0,42%	
21	3875	2360	60,90%	0,71%	
22	1922	679	35,33%	0,21%	
23	1922	943	49,06%	0,29%	
24	0	0	0,00%	0,00%	
Total:	**638135**	**330554**	51,80%	**100,00%**	100,00%

Tabela 5.25 - Cálculo teórico da percentagem de blocos escolhidos com aplicação do LSP em modo alternado

Os resultados desta versão revelaram ser mais rápido que a versão de referência (Tabela 5.26). Contudo a validação desta versão por parte da curva RD mostra que esta se encontra ligeiramente afastada da curva da versão de referência (Figura 5.28) pelo que não se deve considerar esta versão. Porém, analisando esta versão com as restantes podemos considerar que esta obtém o melhor rácio entre valor de speedup e degradação de qualidade.

		Tempo Referência	Tempo	Speedup	Tempo LSP (%)
Versão LSP alternado	**Servidor 1 GTX Titan**	476,136	280,325	1,699	193,160 (68,91%)
	Servidor 1 GTX 680	478,473	282,168	1,696	192,919 (68,37%)

Tabela 5.26 - Resultado da aplicação do LSP em modo alternado

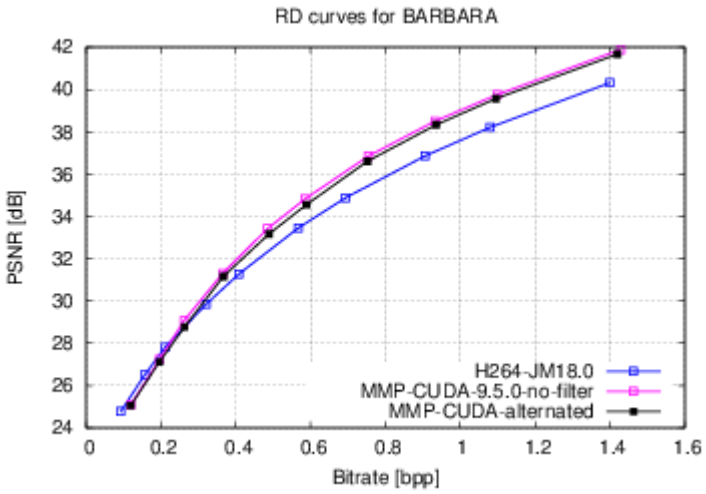

Figura 5.28 – Curva RD da aplicação do LSP em modo alternado

6. Migração de CUDA-MMP para OpenCL-MMP

Neste capítulo é focado a migração da versão CUDA para a versão OpenCL. O capítulo inicia com um estudo sobre os conversores de CUDA para OpenCL analisando a viabilidade da conversão automática de código CUDA para OpenCL. Depois é explicado o trabalho efetuado para se obter a primeira versão OpenCL-MMP. Por fim, são descritas as otimizações realizadas sob a versão OpenCL com o intuito de melhorar o desempenho da mesma.

6.1 Conversores de CUDA para OpenCL

Dadas as muitas semelhanças entre CUDA e OpenCL, a conversão manual de uma aplicação em CUDA para OpenCL é geralmente uma tarefa relativamente simples para um programador. Contudo é uma tarefa tediosa e propensa a erros.

Com o intuito de automatizar o processo de conversão, já foram realizadas várias aplicações que permitam a fácil conversão de uma aplicação CUDA para OpenCL. De seguida é feito um resumo das principais aplicações existentes.

6.1.1 SWAN

O SWAN [49] é uma aplicação para que corra em sistema operativo Linux que visa a facilitar a transição entre CUDA e OpenCL. Esta aplicação não efetua uma tradução direta do código CUDA para OpenCL. Inicialmente é necessário uma pré-conversão por parte do programador das funções da API CUDA (alocação de memória, cópias e chamadas aos *kernels*) para funções da API SWAN (Tabela 6.1) e a aplicação realiza o mapeamento para as funções CUDA (*libswan_cuda*) ou OpenCL (*libswan_ocl*). De notar que o SWAN cria uma função de chamada para cada *kernel*. Essa função tem o nome igual ao *kernel* com o acréscimo do prefixo "k_". Recebe como argumentos o tamanho da *grid*, o tamanho dos blocos, o tamanho da memória partilhada (no caso de alocação dinâmica) e, por fim, os argumentos do kernel. O código dos kernels devem estar de acordo com as normas de CUDA e são traduzidos através de um script em PERL que usa expressões regulares para identificar o código a modificar.

API CUDA	API SWAN
cudaMalloc	swanMalloc
cudaMemcpy (cudaMemcpyHostToDevice)	swanMemcpyHtoD
cudaMemcpy (cudaMemcpyDeviceToHost)	swanMemcpyDtoH
my_kernel<<<grid,block>>>(args…)	k_my_kernel(grid,block,args…)
cudaFree	swanFree

Tabela 6.1 - Mapeamento das funções da API CUDA para a API SWAN

Resumidamente, uma conversão usando a aplicação SWAN é realizada em três passos:

1. Separar o código *host* e o código do *kernel* para diferentes ficheiros.
2. Conversão das funções da API CUDA para as funções da API SWAN
3. Alteração da *makefile* para incluir as bibliotecas SWAN e utilizar o compilador nativo de C para o código host.

Após os três passos anteriores, basta alterar o modo de conversão do SWAN para CUDA ou OpenCL, compilar a aplicação e correr.

A aplicação SWAN foi aplicada com sucesso num algoritmo ACEMD (*accelerated molecular dynamics*) em CUDA para OpenCL [50]. O ACEMD é um algoritmo clássico de dinâmica molecular em que é possível obter um grande ganho de desempenho em CUDA [51]. Em termos de desempenho, a aplicação CUDA é mais rápida que a aplicação OpenCL num fator de 1,4 vezes usando uma placa NVIDIA Tesla C1060. A aplicação OpenCL foi ainda testada usando uma placa AMD e esta correu devidamente sem necessidade de recompilação. Em termos de resultados, os autores apenas indicam que usando a placa AMD o desempenho ficou ligeiramente mais lento concluindo que são necessárias otimizações específicas para se obter o mesmo desempenho do que o obtido com uma placa NVIDIA.

No âmbito deste trabalho, este conversor não foi testado devido a problemas de configuração encontrados. Importa notar que a aplicação SWAN não é atualizada desde dezembro de 2010.

6.1.2 CU2CL

O CU2CL [52] é uma aplicação que tem como objetivo a tradução automática de um código em CUDA para OpenCL. O código gerado deve conter toda a estrutura antiga (incluindo comentários) para que os programadores possam continuar a sua evolução e/ou manutenção deste código gerado.

O CU2CL foi desenvolvido usando o compilador *front-end* Clang [53] devido a este ter sido desenvolvido sob um *back-end* LLVM [54] que oferece um vasto conjunto de bibliotecas de léxico, análise gramática, análise semântica, entre outras. Outra vantagem do Clang é que este tem suporte para análise de extensões CUDA C.

Através das bibliotecas fornecidas pelo Clang, este conversor identifica as alterações que devem ser realizadas e altera o código usando as funções de *rewrite* existentes na biblioteca (Figura 6.1).

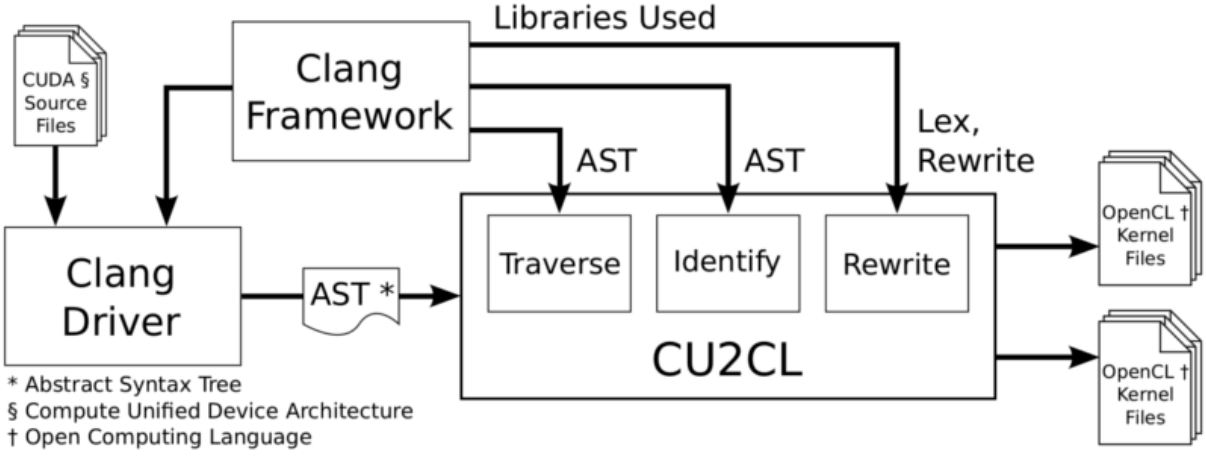

Figura 6.1 - Visão geral do processo de tradução da aplicação CU2CL [55]

De notar que caso exista alguma função que ainda não seja suportada ou que a aplicação não consiga traduzir, esta emite uma mensagem de erro para que o programador posso efetuar a alteração necessária.

Este conversor foi testado [56] em algumas aplicações do CUDA SDK e em aplicações de testes do *benchmark* Rodinia [57]. No geral, o conversor foi capaz de traduzir mais de 88,6% das linhas necessárias em cada aplicação sendo que em apenas numa aplicação o conversor conseguiu uma percentagem de 100%. O tempo de tradução foi sempre inferior a meio segundo para cada aplicação. A maior aplicação com cerca de 17768 linhas em CUDA foi traduzida em 0,35 segundos, em que 3491 linhas foram traduzidas e 1786 necessitaram de alteração manual para OpenCL.

Usando este conversor foi conseguida a conversão de uma aplicação em CUDA que efetua a soma dos elementos de dois vetores e guarda o resultado num terceiro. Visto ser uma aplicação muito simples, foi conseguida uma conversão com sucesso.

6.1.3 CUDACL

O CUDACL [58] não é um conversor tradicional. Esta aplicação funciona como gerador de código para chamada de *kernels* (Figura 6.2). Foi desenvolvida uma extensão para o IDE Eclipse que recebe um ficheiro com um *kernel* em CUDA ou OpenCL e deteta os parâmetros para posterior alocação de memória e estabelecer os parâmetros na chamada do *kernel*.

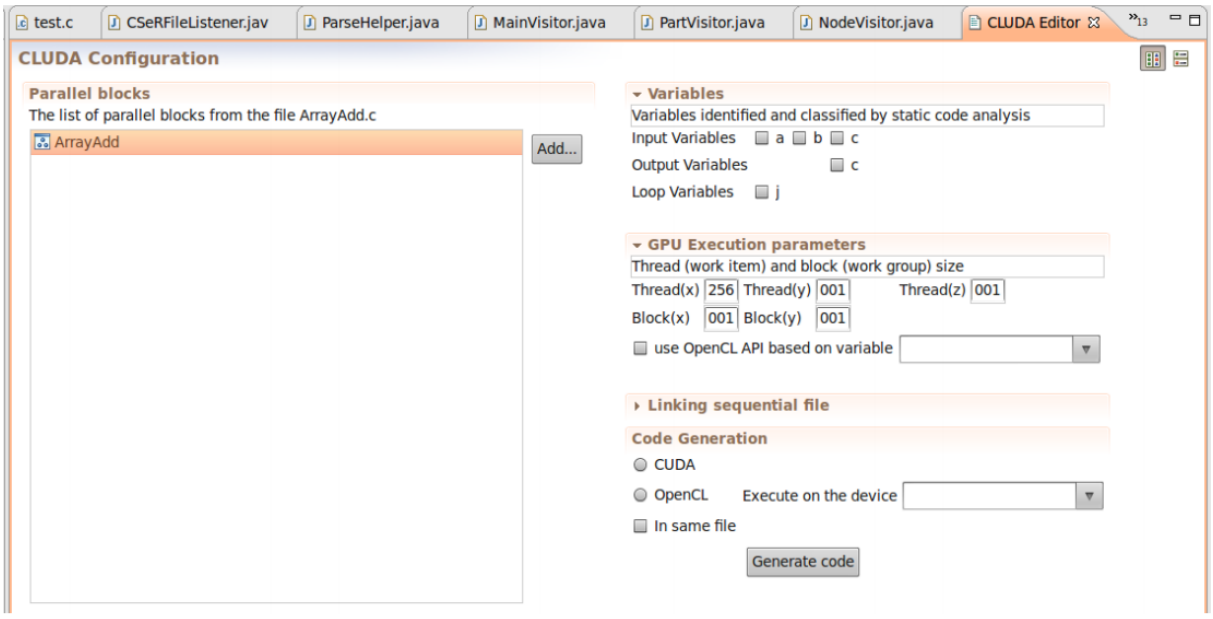

Figura 6.2 - Configuração de execução do kernel "ArrayAdd" usando a extensão CUDACL [58]

Esta aplicação pode gerar código para aplicações em C ou em Java. Caso a chamada a gerar seja em OpenCL, também, é gerado o código da configuração inicial necessária numa aplicação em OpenCL (secção 2.1.6).

O CUDACL foi utilizado com sucesso no desenvolvimento de dois casos de estudo tendo-se verificado que não causa impacto no desempenho das aplicações. Um problema identificado pelos autores, é que como os parâmetros de entrada são lidos através do ficheiro que contém o *kernel,* não existe informação que permita identificar o tamanho necessário para cada parâmetro quando se trata de ponteiros (*arrays)*. Esta solução não foi testada no decorrer do trabalho.

6.1.4 Conclusão

Os conversores de CUDA para OpenCL necessitam ainda de uma significativa melhoria de modo a abrangerem todas as necessidades para um projeto de média a grande dimensões. O conversor SWAN obrigaria, também, a uma pré-conversão das funções CUDA API para SWAN API mas desta forma o programador perde o controlo sob as interações entre *host* e *device*. Deste mesmo modo pode-se realizar logo a conversão para as funções OpenCL API mantendo a organização e simplicidade do código ao gosto de cada programador. De entre as funcionalidades implementadas pelos conversores, estes não se encontram preparados para o uso de memória das constantes que foram implementadas na versão CUDA-MMP devido às suas caraterísticas benéficas para o desempenho dos *kernels* implementados.

6.2 OpenCL-MMP

Devido aos conversores estarem ainda pouco preparados para a conversão de um projeto com as dimensões do CUDA-MMP, a migração da versão CUDA-MMP para OpenCL foi realizada por uma conversão direta manual.

6.2.1 Implementação

A versão OpenCL-MMP, ao contrário da versão CUDA *Runtime*, necessita de código extra inicial que inicializa o OpenCL (Capítulos 2.1.6). Este código inicial precisa de ser executado pelo codificador e descodificador pelo que foi criado um novo ficheiro na pasta comum das duas aplicações.

O código-fonte referente aos *kernels* OpenCL foi colocado num ficheiro de texto em vez da utilização de texto estático nos ficheiros do código fonte onde existe as chamadas aos *kernels* como em CUDA. Esta decisão deveu-se a que num ficheiro de texto qualquer IDE consegue fazer a leitura das palavras-chaves da linguagem facilitando a análise e alterações ao código. Desta forma, também, é possível fazer alterações ao código de qualquer *kernel* sem ser necessário compilar toda a aplicação, pois o ficheiro com o código dos *kernels* é lido e compilado em *runtime*. Esta compilação em *runtime* permite ainda que o tamanho da memória partilhada seja alocado de acordo com os parâmetros de entrada da aplicação, tornando o uso da memória partilhada mais eficiente.

Em CUDA, a memória das constantes é declarada com o identificador *__constant__* e com indicação do tamanho que esta ocupa. Após a declaração estes ficam acessíveis a todos os *kernels.* Exemplo:

```
1.  __constant__ int my_constant_variable [1024];
```

A memória das constantes em OpenCL funciona como um *buffer* com a propriedade apenas de leitura, ou seja, é necessária a sua alocação com a *flag* CL_MEM_READ_ONLY. Exemplo:

```
1.  my_constant_variable = clCreateBuffer(…, CL_MEM_READ_ONLY, sizeof(int) * 1024, …);
```

Estes *buffers* são passados para o *kernel* como argumento acompanhados pela palavra-chave *__constante*. Exemplo:

```
1.  __kernel void my_kernel (…, __constant int* my_constant_variable, …) {…}
```

6.2.2 Profiling

De modo a estudar-se quanto tempo a aplicação OpenCL-MMP demora a realizar cada operação das chamadas à API OpenCL, foram implementadas dois formatos em cada chamada das funções da API. O primeiro formato consiste na chamada das funções na forma dita normal, com os parâmetros necessários ao correto funcionamento. O segundo formato, empregue para medir os tempos é ativado pelo *define* da constante DEBUG_TIMES, ficando o compilador responsável pela compilação deste formato. Neste formato, as funções da API OpenCL recebem um parâmetro de *cl_event* que são utilizados ou para sincronismo ou para depuração da aplicação. Para que os eventos possam ser utilizados para depuração, a lista de tarefa do *device* necessita de ser criada com a *flag* CL_QUEUE_PROFILING_ENABLE. Um evento contém quatro estados:

1. CL_PROFILING_COMMAND_QUEUED: momento em que a tarefa é inserida na lista de tarefas;
2. CL_PROFILING_COMMAND_SUBMIT: momento em que a tarefa sai da lista de tarefas e é submetida para o *device*;
3. CL_PROFILING_COMMAND_START: momento em que o *device* inicia a operação referente à tarefa;
4. CL_PROFILING_COMMAND_END: momento em que o *device* termina a tarefa.

A variável do tipo *cl_event* guarda o momento em que a tarefa mudou de estado e desta forma podemos calcular três métricas: (1) tempo que a tarefa esteve na lista de tarefas, (2) tempo que o *device* demorou a iniciar tarefa e (3) tempo que o *device* demorou a realizar a tarefa. Para se obter os valores guardados pelo evento é utilizada a função "*clGetEventProfilingInfo*".

No final da codificação são mostrados os tempos de execução de cada *kernel,* tempo que se demorou nas cópias (*host* para *device* e *device* para *host*), soma do tempo que cada *kernel* esteve na lista de tarefas (1) e soma do tempo que o *device* demorou a iniciar cada *kernel* (2).

De notar que com a aplicação *sprofile* disponível no pacote de desenvolvimento fornecido pela empresa AMD [59] também é possível realizar um profiling às aplicações OpenCL usando placas da marca AMD. Com a aplicação Code XL [60], também da AMD, é possível obter uma representação gráfica dos dados provenientes da aplicação *sprofile*

6.2.3 Resultados

A primeira versão OpenCL-MMP revelou ser mais lenta que a versão CUDA (Tabela 6.2) em comparação com as referentes placas NVIDIA. Observa-se ainda que a execução em placas AMD demora mais do dobro do tempo que a execução em placas NVIDIA.

	Servidor 1 GTX Titan	Servidor 1 GTX 680	Servidor 1 R9-290X	Servidor 1 HD 7970
Tempo OpenCL	494,044	478,006	1078,669	985,079
Tempo CUDA	469,461	469,454	-	-
***Speedup* CUDA**	1,052	1,018	-	-

	Servidor 2 GTX Titan	Servidor 2 GTX 680	Servidor 2 R9-290X	Servidor 2 HD 7970
Tempo OpenCL	519,010	532,605	1081,763	994,994
Tempo CUDA	512,163	531,966	-	-
***Speedup* CUDA**	1,013	1,001	-	-

Tabela 6.2 - Resultados da primeira versão OpenCL-MMP

6.3 Otimizações

Após a migração da aplicação CUDA para OpenCL, foi necessário otimizar esta versão para melhor se adaptar à *framework* OpenCL.

6.3.1 Transferências de memória

O uso de memória não paginável (*pinned memory*) é uma técnica que permite acelerar o desempenho das cópias entre o *host* e *device*, e vice-versa [61]. Isto acontece porque caso os valores a copiar se encontrem numa memória paginável, a plataforma realiza primeiro a cópia destes valores para um memória não paginável e posteriormente a cópia para o *device* (Figura 6.3).

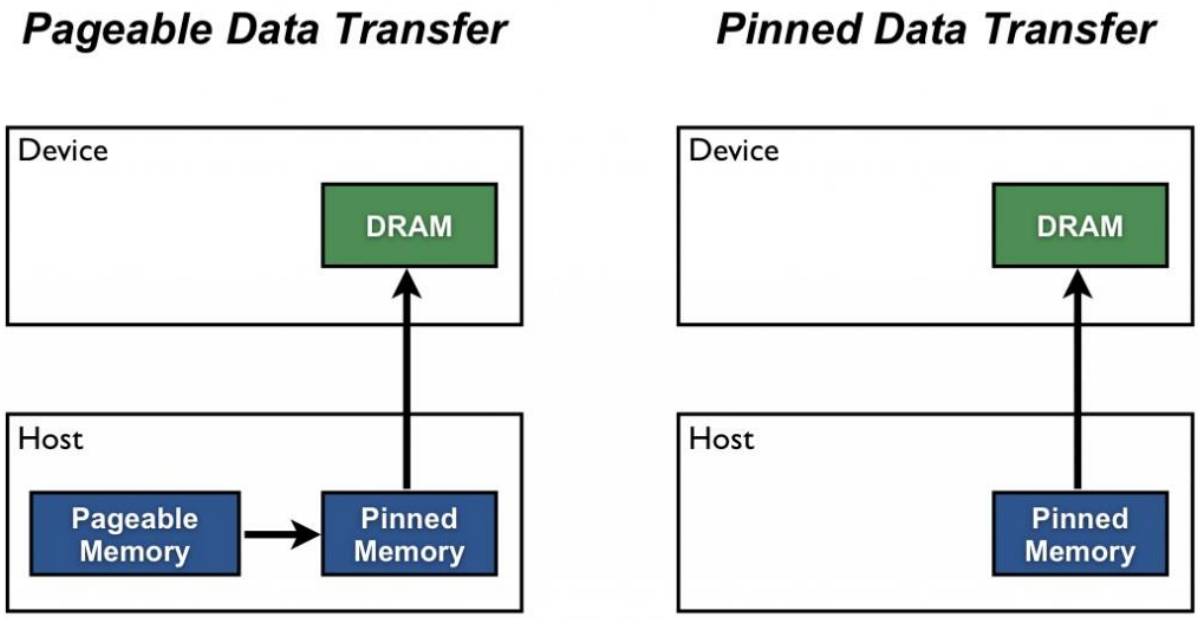

Figura 6.3 - Esquema das cópias com memória paginável (esquerda) e memória não paginável (direita) [62]

Na plataforma OpenCL, quando se cria um *buffer* (memória do *device*) existem três *flags* que permitem utilizar a memória não paginável: (1) CL_MEM_ALLOC_HOST_PTR que indica além da alocação do *buffer* é criado um espaço de memória acessível pelo *host*; (2) CL_MEM_USE_HOST_PTR em que indicamos um ponteiro de memória alocada (*malloc*) para que seja criado um sistema de cache para esta memória e, por último, (3) CL_MEM_COPY_HOST_PTR que aloca a memória no dispositivo e copia os dados existentes no ponteiro de memória indicado.

Caso seja utilizada a *flag* CL_MEM_ALLOC_HOST_PTR para aceder ao ponteira criado pela plataforma é necessário utilizar a função da API OpenCL de *map*, ou seja, *clEnqueueMapBuffer*. Esta função recebe ainda como parâmetro o tipo de acesso que se pretende, isto é, de leitura e/ou de escrita (CL_MAP_READ, CL_MAP_WRITE ou CL_MAP_READ_WRITE). Após o mapeamento, a informação que se encontrava no *buffer* fica acessível para o sistema hospedeiro, caso se tenha utilizado a *flag* de leitura. Caso se tenha utilizado a *flag* de escrita, quando se efetuar o *unmap* do ponteiro (*clEnqueueUnmapBuffer*) é feito a escrita dos valores que se encontram no ponteiro retornado pela função de *map* para o *buffer* que se encontra no dispositivo.

Desta forma, foram implementadas três versões diferentes usando diferentes combinações de *flags* e funções de leitura e escrita dos dados. De notar que apenas os *buffers* com maior utilização foram convertidos para memória paginável, pois são estes que apresentam maior impacto no tempo das cópias entre sistema hospedeiro e dispositivo.

A primeira versão (1) faz uso da *flag* CL_MEM_ALLOC_HOST_PTR pelo que para efetuar as leituras aos valores é necessário utilizar a funções de *map* e para realizar as escritas a função de *unmap.*

	Servidor 1 GTX Titan	Servidor 1 GTX 680	Servidor 1 R9-290X	Servidor 1 HD 7970
Versão Referência	494,044	478,006	1078,669	985,079
Versão (1)	595,030	589,319	1008,352	901,992
Speedup	0,830	0,811	1,070	1,092

	Servidor 2 GTX Titan	Servidor 2 GTX 680	Servidor 2 R9-290X	Servidor 2 HD 7970
Versão Referência	519,010	532,605	1081,763	994,994
Versão (1)	642,554	635,150	930,135	832,325
Speedup	0,808	0,839	1,163	1,195

Tabela 6.3 - Resultados da versão com memória paginável (1)

A segunda versão (2) faz uso da *flag* CL_MEM_ALLOC_HOST_PTR e para efetuar as leituras aos valores é utilizada a funções de *map* e para realizar as escritas a função de *unmap.*

	Servidor 1 GTX Titan	Servidor 1 GTX 680	Servidor 1 R9-290X	Servidor 1 HD 7970
Versão Referência	494,044	478,006	1078,669	985,079
Versão (2)	490,990	485,139	1007,596	904,547
Speedup	1,006	0,985	1,071	1,089

	Servidor 2 GTX Titan	Servidor 2 GTX 680	Servidor 2 R9-290X	Servidor 2 HD 7970
Versão Referência	519,010	532,605	1081,763	994,994
Versão (2)	525,921	523,282	929,241	835,981
Speedup	0,987	1,018	1,164	1,190

Tabela 6.4 - Resultados da versão com memória paginável (2)

Similarmente à segunda versão, a terceira versão (3) faz uso da *flag* CL_MEM_ALLOC_HOST_PTR, mas em vez das funções de *map* e *unmap* são utilizadas as funções de leitura e escrita ditas normais. Isto é, são empregues as funções *clEnqueueReadBuffer* e *clEnqueueWriteBuffer* para leitura e escrita, respetivamente.

	Servidor 1 GTX Titan	Servidor 1 GTX 680	Servidor 1 R9-290X	Servidor 1 HD 7970
Versão Referência	494,044	478,006	1078,669	985,079
Versão (3)	467,778	469,314	966,353	893,928
Speedup	1,056	1,019	1,116	1,102

	Servidor 2 GTX Titan	Servidor 2 GTX 680	Servidor 2 R9-290X	Servidor 2 HD 7970
Versão Referência	519,010	532,605	1081,763	994,994
Versão (3)	510,868	507,544	993,074	908,920
Speedup	1,016	1,049	1,089	1,095

Tabela 6.5 - Resultados da versão com memória paginável (3)

A primeira versão obteve um *speedup* médio de 0,976 (Tabela 6.3), isto é, obteve pior desempenho no geral. A segunda versão obteve uma média de 1,006 (Tabela 6.4) e a terceira de 1,007 (Tabela 6.5). Apesar de serem muito próximos, a terceira versão, onde utiliza a *flag* de CL_MEM_ALLOC_HOST_PTR e faz uso das funções ditas normais de leitura e escrita, obteve um valor ligeiramente superior e será a versão escolhida.

6.3.2 Ocupância

Da mesma forma que em CUDA, em OpenCL o valor da ocupância está ligado à forma como os recursos do *hardware* são utilizados, isto é, o número de *wave-fronts* ativos em relação ao máximo que o dispositivo permite. Em OpenCL existem três fatores que limitam o valor da ocupância [63]: (1) o número de registos (*general purpose registers)* utilizado por cada *work-item*, (2) a quantidade de memória local em utilização pelo *kernel* e, por fim, (3) a configuração do *work-group*, isto é, o número de blocos em execução.

6.3.2.1 Memória local

A memória local, equivalente à memória partilhada em CUDA, é utilizada no *kernel* de otimização para guardar valores intermédios das distorções sendo que no final estes valores são escritos para a memória global que será copiada para o sistema hospedeiro.

```
1.  __kernel void kernel_optimize_block(...){
2.      __local int dists[NUMBER_OF_THREADS_OPTIMIZE * 10];
3.  }
```

Dado que no MMP existem dez modos de predição, o tamanho de memória necessário para guardar os valores intermédios é o número de unidades ativas multiplicado pelo número de modos de predição. No entanto, durante a codificação dos blocos, nem sempre estão disponíveis todos os modos de predição pelo que parte da memória local alocada não é utilizada. Um *profiling* ao *kernel* de otimização por parte da aplicação *sprofile*, mostra que o *kernel* necessita de 10240 bytes de tamanho de memória local num total de 32768 bytes. Ainda mostra que dos três fatores, mencionados anteriormente, a memória local é a que limita mais o valor da ocupância (Tabela 6.6).

Kernel	Máx. Wave Fronts	Limitação por Registos	Limitação por Memória Local	Limitação por *Work-Group*	Ocupância
kernel_optimize_block	40	24	12	40	30%

Tabela 6.6 - Resultados do profiling ao kernel de otimização

Uma solução para este problema passa por implementar vários *kernels* de otimização onde a diferença entre cada um se encontra no tamanho de memória local alocada. Desta forma, iremos ter dez *kernels* e é invocado aquele que contém a memória necessária para executar o processamento dos dados de acordo com o bloco em codificação.

```
1.  __kernel void kernel_optimize_block_1(...){
2.      __local int dists[NUMBER_OF_THREADS_OPTIMIZE];
3.  }
4.
5.  //...
6.
7.  __kernel void kernel_optimize_block_5(...){
8.      __local int dists[NUMBER_OF_THREADS_OPTIMIZE * 5];
9.  }
10.
11. //...
12.
13. __kernel void kernel_optimize_block_10(...){
14.     __local int dists[NUMBER_OF_THREADS_OPTIMIZE * 10];
15. }
```

Um *profiling* aos novos *kernels* (Tabela 6.7) mostra que o valor de ocupância atingiu um máximo de 60%, nos *kernels* que utilizam até 5120 de tamanho de memória local, sendo que a memória local deixou de ser o limitador passando a ser os registos. De notar, que devido a uma versão implementada no âmbito do projeto EPIC onde existe reutilização de modos de predição, então em momento algum da execução da aplicação, para a imagem Lena, os dez modos de predição estão ativos. Isto impõe que o *kernel* que utiliza o máximo de memória, ou seja, para os dez modos de predição, nunca seja lançado.

	Versão de Referência		Otimização da memória local	
	Ocupância	Memória Local	Ocupância	Memória Local
kernel_optimize_block_1	-	-	60%	1024
kernel_optimize_block_2	-	-	60%	2048
kernel_optimize_block_3	-	-	60%	3072
kernel_optimize_block_4	-	-	60%	4096
kernel_optimize_block_5	-	-	60%	5120
kernel_optimize_block_6	-	-	50%	6144
kernel_optimize_block_7	-	-	40%	7168
kernel_optimize_block_8	-	-	40%	8192
kernel_optimize_block_9	-	-	30%	9216
kernel_optimize_block_10	30%	10240	-	-

Tabela 6.7 - Ocupância e tamanho de memória local dos novos kernels de otimização

Em termos de tempo de desempenho dos *kernels* existiu uma melhoria no tempo de execução. Assim, enquanto na versão de referência o *kernel* de otimização demorou cerca de 1112.814 ms (Figura 6.4), no total da soma das execuções de todos os novos *kernels* de otimização perfaz um valor de 943,130 ms (Figura 6.5).

Kernel Name	Device Name	# of Calls	Total Time(ms)	Avg Time(ms)	Max Time(ms)	Min Time(ms)
kernel_optimize_block_10	Tahiti	19958	1112.81378	0.05576	0.70133	0.01511
kernel_j_reducer	Tahiti	19958	624.90875	0.03131	0.62889	0.01393
kernel_compare_blocks	Tahiti	34	20.51763	0.60346	1.22474	0.27333
kernel_update_dic	Tahiti	34	10.25630	0.30166	0.46563	0.11733

Figura 6.4 - Tempo de execução dos kernels na versão de referência

Kernel Name	Device Name	# of Calls	Total Time(ms)	Avg Time(ms)	Max Time(ms)	Min Time(ms)
kernel_j_reducer	Tahiti	19958	503.44407	0.02522	0.62978	0.01407
kernel_optimize_block_6	Tahiti	3647	201.60163	0.05528	0.70518	0.02400
kernel_optimize_block_5	Tahiti	3748	170.65229	0.04553	0.68222	0.02222
kernel_optimize_block_3	Tahiti	3292	169.76724	0.05157	0.44133	0.01881
kernel_optimize_block_2	Tahiti	3170	133.63070	0.04215	0.63467	0.01719
kernel_optimize_block_1	Tahiti	3908	125.70803	0.03217	0.41185	0.01600
kernel_optimize_block_4	Tahiti	1604	73.66460	0.04593	0.82815	0.02104
kernel_optimize_block_7	Tahiti	584	65.12771	0.11152	0.54252	0.02622
kernel_compare_blocks	Tahiti	34	16.28933	0.47910	2.00548	0.27793
kernel_update_dic	Tahiti	34	8.02844	0.23613	0.46044	0.10489
kernel_optimize_block_9	Tahiti	3	1.88296	0.62765	0.63733	0.61585
kernel_optimize_block_8	Tahiti	2	1.17185	0.58592	0.58755	0.58430

Figura 6.5 - Tempo de execução dos kernels na versão de otimização da memória local

Apesar de no *profiling* se verificarem melhorias significativas no valor da ocupância, no geral o tempo total da aplicação piorou (Tabela 6.8).

	Servidor 1 GTX Titan	Servidor 1 GTX 680	Servidor 1 R9-290X	Servidor 1 HD 7970
Versão Referência	467,778	469,314	966,353	893,928
Otimização memória local	470,346	465,464	967,307	898,521
Speedup	0,995	1,008	0,999	0,995

	Servidor 2 GTX Titan	Servidor 2 GTX 680	Servidor 2 R9-290X	Servidor 2 HD 7970
Versão Referência	510,868	507,544	993,074	908,920
Otimização memória local	510,511	504,957	996,341	911,357
Speedup	1,001	1,005	0,997	0,997

Tabela 6.8 - Resultados da otimização da memória local

6.3.2.2 Memória privada

Na plataforma OpenCL, a memória privada está dividida em dois tipos de registos: (1) registos escalares e (2) registos vetoriais. Os registos vetoriais (VGPR) são utilizados pelo vetor ALU e sistema de memória de vetores. Os registos escalares (SGPR) são utilizados para as necessidades dos *work-items*. Dentro de um *wavefront* estes registos estão acessíveis para todos os *work-items* [64].

O uso excessivo dos registos implica que o valor de ocupância baixe, isto porque o número de registos se encontra limitado pelo *hardware.* Com a otimização anterior, o número de registos em utilização passou a ser o principal fator de limitação da ocupância para os *kernels* com menor valor de memória partilhada (Tabela 6.9).

Kernel	Máx. Wave Fronts	Limitação por Registos	Limitação por Memória Local	Limitação por *Work-Group*	Ocupância
kernel_optimize_block_1	40	24	40	40	60%
kernel_optimize_block_5	40	24	24	40	60%
kernel_optimize_block_9	40	24	12	40	30%
kernel_j_reducter	40	40	32	40	80%

Tabela 6.9 - Limitações ao valor da ocupância

Os *kernel* de redução encontra-se num estado otimizado pelo que não necessita de alterações. No caso do *kernel* de otimização foi feito uma revisão do número de registos e do código de modo a que se possam reutilizar registos depois de estes não serem mais necessários para a execução. Após a revisão do código, o *profiling* indica que não existiu alteração da limitação por parte do número de registo (Tabela 6.10), o que comprova que o compilador consegue detetar estes registos e trata da sua reutilização. Esta otimização também foi realizada em CUDA onde se chegou à mesma conclusão.

Kernel	Máx. Wave Fronts	Limitação por Registos	Limitação por Memória Local	Limitação por *Work-Group*	Ocupância
kernel_optimize_block_1	40	24	40	40	60%
kernel_optimize_block_5	40	24	24	40	60%
kernel_optimize_block_9	40	24	12	40	30%
kernel_j_reducter	40	40	32	40	80%

Tabela 6.10 - Limitações ao valor da ocupância após revisão do código

Apesar de o *profiling* indicar que as limitações e o valor de ocupância se mantém inalterados, os resultados finais mostram uma ligeira melhoria no tempo de execução (Tabela 6.11).

	Servidor 1 GTX Titan	Servidor 1 GTX 680	Servidor 1 R9-290X	Servidor 1 HD 7970
Versão Referência	470,346	465,464	967,307	898,521
Otimização memória privada	468,722	465,221	962,728	901,332
Speedup	1,003	1,001	1,005	0,997

	Servidor 2 GTX Titan	Servidor 2 GTX 680	Servidor 2 R9-290X	Servidor 2 HD 7970
Versão Referência	510,511	504,957	996,341	911,357
Otimização memória privada	508,415	505,909	991,304	909,775
Speedup	1,004	0,998	1,005	1,002

Tabela 6.11 - Resultados da otimização da memória privada

7. Desempenho da aplicação OpenCL-MMP

Nesta secção é feito um estudo mais aprofundado do desempenho da aplicação OpenCL-MMP. Este estudo mostra os resultados da aplicação usando placas da marca NVIDIA e AMD e a respetiva comparação. No final é feita uma comparação entre a versão OpenCL, CUDA e Sequencial do MMP.

7.1 Comparação entre AMD e NVIDIA

A Tabela 7.1 apresenta os resultados da versão final da aplicação OpenCL-MMP executado em placas da marca AMD.

Limite Dicionário	Lambda (λ)	Servidor 1 R9-290X	Servidor 1 HD 7970	Servidor 2 R9-290X	Servidor 2 HD 7970
1024	10	962,728	901,332	991,304	918,561
	50	931,890	915,183	964,650	916,921
	300	915,540	905,032	944,690	909,775
50000	10	1102,934	957,430	1132,230	971,985
	50	987,901	890,602	1024,140	896,961
	300	916,654	882,293	950,119	887,413

Tabela 7.1 - Resultados da aplicação OpenCL-MMP em placas AMD

Por outro lado, a Tabela 7.2 apresenta os resultados da mesma versão mas para placas da marca NVIDIA.

Limite Dicionário	Lambda (λ)	Servidor 1 GTX Titan	Servidor 1 GTX 680	Servidor 2 Titan	Servidor 2 680
1024	10	470,346	465,464	510,511	504,957
	50	460,945	460,235	497,728	495,315
	300	458,276	455,070	496,792	492,251
50000	10	519,374	523,502	555,219	562,906
	50	475,751	476,509	514,374	515,279
	300	462,927	458,472	499,081	495,231

Tabela 7.2 - Resultados da aplicação OpenCL-MMP em placas NVIDIA

Visto a AMD ser a principal impulsionadora da plataforma OpenCL, pois a NVIDIA alterou os seus esforços para a sua plataforma proprietária CUDA, seria de esperar que as placas AMD apresentassem

melhores resultados. Os resultados obtidos mostram que existe uma grande discrepância de tempos, cerca do dobro do tempo, entre as placas de diferentes marcas sendo que as placas NVIDIA obtém melhores resultados. Importa notar que a discrepância entre tempos de execução se verifica mesmo quando se compara a GTX 680 com a AMD R9-290x que são placas de gerações distintas, com a GTX 680 mais antiga que a R9-290x. Uma possível explicação para a elevada diferença de desempenho pode dever-se à qualidade dos compiladores e dos *drivers.*

7.1.1 Estados de um kernel em OpenCL

Um *profiling* sob os *kernels* demonstram que a execução em placas AMD, em comparação às placas NVIDIA, é mais lenta em todos os intervalos entre estados que um *kernel* pode apresentar na plataforma OpenCL. Um *kernel* em OpenCL desde a sua invocação até ao seu término apresentam quatro estados (secção 6.2.2). O intervalo de tempo entre estes quatro estados permite calcular três métricas: (1) tempo que o *kernel* esteve na lista de tarefas, (2) tempo que o *device* demorou a iniciar o *kernel* e (3) tempo que o *device* demorou no processamento do *kernel*.

A Tabela 7.3 mostra o resultado das métricas obtidas entre as diferentes placas NVIDIA e AMD. A primeira métrica (tempo que os *kernels* estiveram na lista de tarefas), as placas NVIDIA no total demoram menos de 4s enquanto as placas AMD atingem um tempo superior a 80s. A segunda métrica (tempo que o dispositivo demora a iniciar o *kernel* após este ser submetido) é o que apresenta a maior diferença onde as placas AMD apresentam um valor na ordem das centenas enquanto as placas NVIDIA na ordem das unidades. A terceira métrica (tempo de execução dos *kernels)* é a que apresenta menor discrepância de tempo entre as duas marcas de GPUs.

	Servidor 1 GTX Titan	Servidor 1 GTX 680	Servidor 1 R9–290X	Servidor 1 HD 7970
K. Otimização	21,987	23,524	47,9078	62,071
K. Redução	11,577	12,167	25,5804	27,541
K. Comparação	0,657	0,670	4,5504	9,3596
K. Atualização	0,096	0,097	0,1174	0,1894
Total Métrica 1	3,979	1,927	82,8062	81,469
Total Métrica 2	16,831	4,806	321,9976	228,743
Tempo Total	468,722	465,221	962,728	901,332

Tabela 7.3 - Métricas de tempo por placa da versão OpenCL-MMP (imagem Lena e λ=10)

A discrepância de valores entre estas três métricas permitem justificar a diferença de tempos obtidos do protótipo OpenCL em placas NVIDIA e AMD.

7.2 Comparação entre protótipo em OpenCL e em CUDA

A primeira comparação efetuada entre OpenCL e CUDA aconteceu após a primeira implementação OpenCL (secção 6.2.3), onde existia superioridade da versão CUDA executado em placas NVIDIA. Após as otimizações efetuadas sob a versão OpenCL, este consegue se aproximar bastante dos tempos

obtidos pela versão CUDA em placas NVIDIA. Nalguns testes, a versão OpenCL consegue mesmo superar os tempos da versão CUDA (Tabelas 7.4, 7.5, 7.6 e 7.7).

Limite do Dic.	Lambda (λ)	Protótipo	Servidor 1 GTX Titan	Servidor 1 GTX 680	Servidor 1 R9-290X	Servidor 1 HD 7970
1024	10	Tempo OpenCL	470,346	465,464	962,728	901,332
		Tempo CUDA	469,461	469,454	-	-
		Speedup CUDA	1,002	0,992	-	-
	50	Tempo OpenCL	460,945	460,235	931,890	915,183
		Tempo CUDA	460,931	459,852	-	-
		Speedup CUDA	1,000	1,001	-	-
	300	Tempo OpenCL	458,276	455,070	915,540	905,032
		Tempo CUDA	453,796	454,212	-	-
		Speedup CUDA	1,010	1,002	-	-

Tabela 7.4 - Comparação da versão final OpenCL-MMP e CUDA-MMP (servidor 1 e limite do dicionário a 1024)

Limite do Dic.	Lambda (λ)	Protótipo	Servidor 1 GTX Titan	Servidor 1 GTX 680	Servidor 1 R9-290X	Servidor 1 HD 7970
50000	10	Tempo OpenCL	519,374	523,502	1102,934	957,430
		Tempo CUDA	515,928	519,381	-	-
		Speedup CUDA	1,007	1,008	-	-
	50	Tempo OpenCL	475,751	476,509	987,901	890,602
		Tempo CUDA	473,725	474,297	-	-
		Speedup CUDA	1,004	1,005	-	-
	300	Tempo OpenCL	462,927	458,472	916,654	882,293
		Tempo CUDA	457,986	458,179	-	-
		Speedup CUDA	1,011	1,001	-	-

Tabela 7.5 - Comparação da versão final OpenCL-MMP e CUDA-MMP (servidor 1 e limite do dicionário a 50000)

Limite do Dic.	Lambda (λ)	Protótipo	Servidor 2 GTX Titan	Servidor 2 GTX 680	Servidor 2 R9-290X	Servidor 2 HD 7970
1024	10	Tempo OpenCL	510,511	504,957	991,304	918,561
		Tempo CUDA	512,163	531,957	-	-
		Speedup CUDA	0,997	0,949	-	-
	50	Tempo OpenCL	497,728	495,315	964,650	916,921
		Tempo CUDA	512,163	517,585	-	-
		Speedup CUDA	0,972	0,957	-	-
	300	Tempo OpenCL	496,792	492,251	944,690	909,775
		Tempo CUDA	497,378	512,456	-	-
		Speedup CUDA	0,999	0,961	-	-

Tabela 7.6 - Comparação da versão final OpenCL-MMP e CUDA-MMP (servidor 2 e limite do dicionário a 1024)

Limite do Dic.	Lambda (λ)	Protótipo	Servidor 2 GTX Titan	Servidor 2 GTX 680	Servidor 2 R9-290X	Servidor 2 HD 7970
50000	10	Tempo OpenCL	555,219	562,906	1132,230	562,906
		Tempo CUDA	559,059	580,810	-	-
		Speedup CUDA	0,993	0,969	-	-
	50	Tempo OpenCL	514,374	515,279	1024,140	515,279
		Tempo CUDA	516,841	536,102	-	-
		Speedup CUDA	0,995	0,961	-	-
	300	Tempo OpenCL	499,081	495,231	950,119	495,231
		Tempo CUDA	498,970	517,874	-	-
		Speedup CUDA	1,000	0,956	-	-

Tabela 7.7 - Comparação da versão final OpenCL-MMP e CUDA-MMP (servidor 2 e limite do dicionário a 50000)

7.3 Comparação entre protótipo em OpenCL e versão sequencial

As Tabelas 7.8, 7.9, 7.10 e 7.11 efetuam a comparação entre a versão final de OpenCL e a versão Sequencial de referência para o projeto. Como já referido anteriormente, o OpenCL em placas NVIDA corre mais rápido do que em placas AMD pelo que o valor de *speedup* será diferente de acordo com as marcas. Com as placas da marca NVIDIA o valor de *speedup* encontra-se entre 2,1 a 10,8 em comparação com as placas AMD que se encontra entre 1,1 a 5,8.

Limite do Dic.	Lambda (λ)	Protótipo	Servidor 1 GTX Titan	Servidor 1 GTX 680	Servidor 1 R9-290X	Servidor 1 HD 7970
1024	10	**Versão Sequencial**	2109,020			
		Tempo OpenCL	470,346	465,464	962,728	901,332
		Speedup	4,484	4,531	2,191	2,340
	50	**Versão Sequencial**	1557,910			
		Tempo OpenCL	460,945	460,235	931,890	915,183
		Speedup	3,380	3,385	1,672	1,702
	300	**Versão Sequencial**	1057,282			
		Tempo OpenCL	458,276	455,070	915,540	905,032
		Speedup	2,307	2,323	1,155	1,168

Tabela 7.8 - Comparação da versão final OpenCL-MMP e Sequencial (servidor 1 e limite do dicionário a 1024)

Limite do Dic.	Lambda (λ)	Protótipo	Servidor 1 GTX Titan	Servidor 1 GTX 680	Servidor 1 R9-290X	Servidor 1 HD 7970
50000	10	Versão Sequencial	5646,805			
		Tempo OpenCL	519,374	523,502	1102,934	957,430
		Speedup	10,872	10,787	5,120	5,898
	50	Versão Sequencial	2388,571			
		Tempo OpenCL	475,751	476,509	987,901	890,602
		Speedup	5,021	5,013	2,418	2,682
	300	Versão Sequencial	1173,213			
		Tempo OpenCL	462,927	458,472	916,654	882,293
		Speedup	2,534	2,559	1,280	1,330

Tabela 7.9 - Comparação da versão final OpenCL-MMP e Sequencial (servidor 1 e limite do dicionário a 50000)

Limite do Dic.	Lambda (λ)	Protótipo	Servidor 2 GTX Titan	Servidor 2 GTX 680	Servidor 2 R9-290X	Servidor 2 HD 7970
1024	10	Versão Sequencial	2109,020			
		Tempo OpenCL	510,511	504,957	991,304	918,561
		Speedup	4,131	4,177	2,128	2,296
	50	Versão Sequencial	1557,910			
		Tempo OpenCL	497,728	495,315	964,650	916,921
		Speedup	3,130	3,145	1,615	1,699
	300	Versão Sequencial	1057,282			
		Tempo OpenCL	496,792	492,251	944,690	909,775
		Speedup	2,128	2,148	1,119	1,162

Tabela 7.10 - Comparação da versão final OpenCL-MMP e Sequencial (servidor 2 e limite do dicionário a 1024)

Limite do Dic.	Lambda (λ)	Protótipo	Servidor 2 GTX Titan	Servidor 2 GTX 680	Servidor 2 R9-290X	Servidor 2 HD 7970
50000	10	Versão Sequencial	5646,805			
		Tempo OpenCL	555,219	562,906	1132,230	971,985
		Speedup	10,170	10,032	4,987	5,810
	50	Versão Sequencial	2388,571			
		Tempo OpenCL	514,374	515,279	1024,140	896,961
		Speedup	4,644	4,635	2,332	2,663
	300	Versão Sequencial	1173,213			
		Tempo OpenCL	499,081	495,231	950,119	887,413
		Speedup	2,351	2,369	1,235	1,322

Tabela 7.11 - Comparação da versão final OpenCL-MMP e Sequencial (servidor 2 e limite do dicionário a 50000)

7.4 Validação do protótipo OpenCL

A Figura 7.1 mostra a curva RD que compara os resultados obtidos entre as diversas versões do MMP. De destacar que a versão OpenCL manteve a qualidade de compressão obtida pela versão sequencial pelo que os resultados obtidos são válidos.

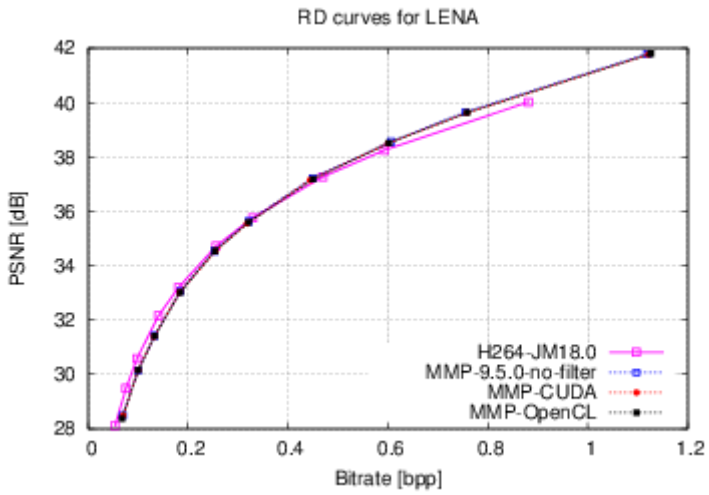

Figura 7.1 - Curva RD imagem Lena com λ=10

8. Conclusão e Trabalho Futuro

8.1 Conclusão

Os objetivos globais deste trabalho incluía a implementação de dois protótipos, um em CUDA e outro em OpenCL, seguindo um paradigma de multi-núcleo em ambiente GPU sob o algoritmo de compressão de sinais MMP. Este algoritmo é altamente recursivo pelo que se trata de um bom candidato para a implementação num ambiente *many-core.* Contudo, esta paralelização apenas é possível durante a codificação de um macro-bloco devido à dependência que existe entre macro-blocos, isto é, a atualização do dicionário no final da codificação de cada bloco.

Das duas implementações surgiram vários tipos de otimizações, não só do estudo do comportamento dos *kernels* bem como da interação entre o sistema hospedeiro e os dispositivos. Este estudo apenas é possível com recurso a ferramentas de *profiling* que auxiliam bastante o trabalho do programador para detetar os pontos que devem ser otimizados.

Como mostrado pela primeira versão implementada em CUDA, caso de estudo no início desta dissertação, não basta portar as principais funções críticas de desempenho da aplicação para serem executados pela GPU e esperar resultados positivos. O uso irregular dos recursos do dispositivo apenas atrasam mais o desempenho do mesmo pelo que no final ficará mais lento que a versão de referência. É necessário realizar um constante estudo do comportamento dos *kernels* bem como da iteração entre o sistema hospedeiro e os dispositivos de modo à aplicação adaptar-se da melhor forma às especificações dos paradigmas multi-núcleo. As otimizações efetuadas tinham sempre como objetivo permitir aumentar o desempenho da aplicação sendo que nem todas foram conseguidas com resultados positivos. No entanto, a grande maioria das otimizações permitiram aumentar o desempenho da aplicação e acelerar o tempo de execução do algoritmo.

Usando os parâmetros de referência, a aplicação sequencial que executa na ordem dos 35 minutos passou para cerca 8 minutos usando os paradigmas CUDA e OpenCL em placas NVIDA e 15 minutos na versão OpenCL usando placas AMD. Estes resultados permitem obter um *speedup* de 4 vezes e 2 vezes, respectivamente. Contudo, com o uso de parâmetros que solicitem o maior crescimento do dicionário, maior se reflete o valor de *speedup* para as versões múlti-núcleos. A título de exemplo, permitindo o crescimento do dicionário até 50 mil elementos, a versão sequencial demora cerca de 1 hora e 35 minutos, enquanto a versão CUDA e OpenCL demoram cerca de 9 minutos usando a placa NVIDIA GTX Titan, o que perfaz um *speedup* superior a 10 vezes.

Dos resultados obtidos, os dispositivos NVIDIA empregues nos testes desenvolvidos permitiram um melhor desempenho, cerca de duas vezes mais rápido, do que os dispositivos AMD quando se faz uso da aplicação OpenCL-MMP. Destaca-se ainda que no *hardware* NVIDIA, esta aplicação conseguiu ter um desempenho muito semelhante à versão CUDA-MMP.

De notar que numa comparação de tempo mais justa, a versão executada apenas pela CPU também deveria ser implementada com recurso a um paradigma de *multi-thread.* Esta versão também poderia obter um desempenho superior à versão sequencial e desta forma concluir qual o melhor paradigma a utilizar.

As versões ditas sub-ótimas permitem ainda aumentar o desempenho da aplicação, à custa de qualidade de compressão. Caso se consiga utilizar um rácio entre desempenho e perda de qualidade que satisfaça as condições de validação do algoritmo poderá passar-se para um outro nível de otimizações que permitam aumentar o desempenho do algoritmo. Desta forma mais rapidamente se consegue atingir o objetivo de implementar uma versão do MMP que possa considerar uma versão comercial e de uso no dia-a-dia.

8.2 Trabalho Futuro

As evoluções constantes da tecnologia faz com que apareçam novas técnicas para a computação paralela. No decorrer desta dissertação foram lançadas novas funcionalidades que poderão ser aplicadas à versão multi-núcleo de forma a obter ainda um melhor desempenho.

A NVIDIA lançou uma nova funcionalidade de memória unificada que não se encontra implementada sob o protótipo CUDA [65]. Esta funcionalidade abstrai o programador da necessidade de efetuar as cópias entre o sistema hospedeiro e o dispositivo, e vice-versa.

Outra forma de conseguir um melhor desempenho para os protótipos em CUDA e OpenCL seria o uso de mais do que um dispositivo GPU. Desta forma poder-se-ia balancear melhor o trabalho entre as GPUs e efetuar mais trabalho em paralelo de modo a obter ainda melhor desempenho.

Como mostrado nesta dissertação, no final, o processo que consome mais tempo prende-se com os cálculos para o modo de predição LSP. Pelo que uma implementação em GPU ou outro paradigma será um caso necessário de modo a conseguir acelerar ainda mais o desempenho do MMP.

A versão sequencial pode ser explorada para uma implementação com recurso ao paradigma OpenMP, referenciado no estado da arte deste documento, de modo a obter uma versão *many-core* ao nível da CPU.

Bibliografia

[1] D. S. Taubman e M. Marcelin, "JPEG2000: Image Compression Fundamentals, Standards and Practice," *Kluwer Academic Publishers,* 2001.

[2] "Information technology -- Computer graphics and image processing -- Portable Network Graphics (PNG): Functional specification," *ISO/IEC 15948:2004,* 2004.

[3] J. V. T. (. o. I. M. &. I.-T. V. (. J. a. I.-T. S. Q.6), "Draft of Version 4 of H.264/AVC," *ITU-T Recommendation H.264 and ISO/IEC 14496-10 (MPEG-4 part 10) Advanced Video Coding,* March 2005.

[4] G. J. Sullivan,, J.-R. Ohm, W.-J. Han e T. Wiegand, "Overview of the High Efficiency Video Coding (HEVC) Standard," *IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY, VOL. 22, NO. 12,* dezembro 2012.

[5] E. d. S. e. W. F. M. de Carvalho, "Multidimensional Signal Compression using Multiscale Recurrent Patterns," *Elsevier Signal Processing, Vol. 82, Pp. 1559–1580,* novembro 2002.

[6] N. M. M. Rodrigues, "Multiscale Recurrent Pattern Matching Algorithms for Image and Video Coding, Phd Thesis," Faculdade de Ciências e Tecnologia - Universidade de Coimbra, Outubro 2008.

[7] W. I. Fidelity, "The 'free lunch' is over," agosto 2013. [Online]. Available: https://www.fidelity.de/static/pdf/institutional/investment-management/no-free-lunch-perspective.pdf. [Acedido em março 2015].

[8] "NVIDIA CUDA Zone," NVIDIA, [Online]. Available: http://developer.nvidia.com/cuda. [Acedido em março 2015].

[9] "The open standard for parallel programming of heterogeneous systems," The Khronos Group, [Online]. Available: https://www.khronos.org/opencl/. [Acedido em março 2015].

[10] "Apple Previews Mac OS X Snow Leopard to Developers," Apple, 9 Junho 2008. [Online]. Available: http://www.apple.com/pr/library/2008/06/09Apple-Previews-Mac-OS-X-Snow-Leopard-to-Developers.html. [Acedido em Julho 2014].

[11] “OpenGL,” The Khronos Group, [Online]. Available: https://www.khronos.org/opengl/. [Acedido em Setembro 2014].

[12] “Khronos Launches Heterogeneous Computing Initiative,” The Khronos™ Group, 2008 Junho 16. [Online]. Available: https://www.khronos.org/news/press/2008/06. [Acedido em Julho 2014].

[13] “Khronos Drives Momentum of Parallel Computing Standard with Release of OpenCL 1.1 Specification,” The Khronos Group, 14 Junho 2010. [Online]. Available: https://www.khronos.org/news/press/khronos-group-releases-opencl-1-1-parallel-computing-standard. [Acedido em Setembro 2014].

[14] “Khronos Releases OpenCL 1.2 Specification,” The Khronos Group, 15 Novembro 2011. [Online]. Available: https://www.khronos.org/news/press/khronos-releases-opencl-1.2-specification. [Acedido em Setembro 2014].

[15] “Khronos Finalizes OpenCL 2.0 Specification for Heterogeneous Computing,” The Khronos Group, 18 Novembro 2013. [Online]. Available: https://www.khronos.org/news/press/khronos-finalizes-opencl-2.0-specification-for-heterogeneous-computing. [Acedido em Setembro 2014].

[16] T. K. Group, “The OpenCL Specification Version 2.1,” janeiro 2015. [Online]. Available: https://www.khronos.org/registry/cl/specs/opencl-2.1.pdf. [Acedido em março 2015].

[17] A. M. D. (AMD), Introduction to OpenCL Programming - Training Guide, 2010.

[18] IEEE, “754-2008 - IEEE Standard for Floating-Point Arithmetic,” pp. 1-70, 2008.

[19] “CUDA Parallel Computing Platform,” NVIDIA, [Online]. Available: http://www.nvidia.com/cuda. [Acedido em Julho 2014].

[20] D. B. Kirk e W.-m. W. Hwu, Programming Massively Parallel Processors - A Hands-on Approach, Elsevier Inc., 2010.

[21] NVIDIA, “NVIDIA CUDA Runtime API,” NVIDIA, [Online]. Available: http://docs.nvidia.com/cuda/cuda-runtime-api/#axzz3LXc2kp4d. [Acedido em junho 2014].

[22] D. Nandakumar, “Automatic translation of CUDA to OpenCL and comparison of performance optimizations on GPUS,” University of Illinois, 2011. [Online]. Available: http://hdl.handle.net/2142/24279. [Acedido em 2014 setembro].

[23] “OpenMP,” [Online]. Available: http://openmp.org/wp/. [Acedido em Julho 2014].

[24] T. Mattson, “A “Hands-on” Introduction to OpenMP,” [Online]. Available: http://openmp.org/mp-documents/Intro_To_OpenMP_Mattson.pdf. [Acedido em Julho 2014].

[25] “OpenACC Home,” OpenACC, [Online]. Available: http://www.openacc-standard.org/. [Acedido em Julho 2014].

[26] “OpenACC Directives,” NVIDIA, [Online]. Available: http://www.nvidia.com/object/openacc-gpu-directives.html. [Acedido em Julho 2014].

[27] G. C. B. Abrahão, “Codificação de Voz Utilizando Recorrência de Padrões Multiescala, Phd thesis,” COPPE - Universidade Federal do Rio de Janeiro, Novembro 2005.

[28] M. B. Carvalho, “Compression of Multidimensional Signals based on Recurrent Multiscale Patterns,” COPPE - Universidade Federal do Rio de Janeiro, Março 2001.

[29] N. C. Francisco, “Estudo da Utilização de Recorrência de Padrões Multiescala na Codificação de Documentos Compostos, Msc Thesis,” Universidade de Trás-os-Montes e Alto Douro, 2007.

[30] N. M. M. Rodrigues, E. A. B. da Silva, M. B. Carvalho, S. M. M. de Faria e V. M. M. Silva, “An efficient h.264-based video encoder using multiscale recurrent patterns,” *SPIE - Applications of Digital Image Processing,* Agosto 2006.

[31] N. M. M. Rodrigues, . E. A. B. da Silva, M. B. de Carvalho, S. M. M. de Faria e V. M. M. Silva, “Improving h.264/avc inter compression with multiscale recurrent patterns,” *ICIP 2006 - IEEE International Conference on Image,* Outubro 2006.

[32] A. Ortega e K. Ramchandran, “Rate-distortion methods for image and video compression,” *IEEE Signal Processing Magazine,* 1998.

[33] K. Sayood, “Introduction to Data Compression,” *Morgan Kaufman Publishers,* 2000.

[34] T. S. S. o. ITU, “SERIES H: AUDIOVISUAL AND MULTIMEDIA SYSTEMS - Infrastructure of audiovisual services - Coding of moving video,” *Advanced video coding for generic audiovisual,* fevereiro 2014.

[35] “Features GeForce GTX Titan Black,” NVIDIA, [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-black/features. [Acedido em março 2015].

[36] “Features GeForce GTX 680,” NVIDIA, [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-680/features. [Acedido em 2015 março].

[37] “AMD Radeon R9 290X,” Tech Power Up, [Online]. Available: http://www.techpowerup.com/gpudb/2460/radeon-r9-290x.html. [Acedido em março 2015].

[38] “AMD Radeon HD 7970,” Tech Power Up, [Online]. Available: http://www.techpowerup.com/gpudb/296/radeon-hd-7970.html. [Acedido em março 2015].

[39] “GNU gprof,” The University of Utah - Department of Computer Science, janeiro 1993. [Online]. Available: https://www.cs.utah.edu/dept/old/texinfo/as/gprof.html. [Acedido em setembro 2014].

[40] “GNU gprof - Compiling a Program for Profiling,” The University of Utah - Department of Computer Science, janeiro 1993. [Online]. Available: https://www.cs.utah.edu/dept/old/texinfo/as/gprof.html#SEC2. [Acedido em setembro 2014].

[41] F. Pillet, “What is KProf ?,” SourceForge, 2000-2001. [Online]. Available: http://kprof.sourceforge.net/. [Acedido em Junho 2014].

[42] NVIDIA Developer Zone, “CUDA Toolkit Documentation,” NVIDIA, [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#occupancy. [Acedido em Fevereiro 2015].

[43] M. Harris, “Optimizing Parallel Reduction in CUDA,” [Online]. Available: http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf. [Acedido em Novembro 2014].

[44] “CMSC 411 Computer Systems Architecture,” [Online]. Available: https://www.cs.umd.edu/class/spring2012/cmsc411/lectures/lec08.pdf. [Acedido em março 2015].

[45] X. Li, “Least-square prediction for backward adaptive video coding,” *EURASIP Journal on Applied Signal Processing,* pp. 126-129, 2006.

[46] D. B. Graziosi, N. M. M. Rodrigues, E. A. B. Silva, S. M. M. Faria e M. B. Carvalho, “Improving Multiscale Recurrent Pattern Image Coding with Least-Squares Prediciton Mode,” *ICIP 2009,* 2009.

[47] NVIDIA CUDA, “CUDA Toolkit Documentation - Constant Memory,” NVIDIA, [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#constant-memory.

[48] NVIDIA, “Shared Memory and Memory Banks,” NVIDIA, [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#shared-memory-and-memory-banks. [Acedido em Novembro 2014].

[49] “Swan: A simple tool for porting CUDA to OpenCL,” multiscalelab, [Online]. Available: http://www.multiscalelab.org/swan. [Acedido em Julho 2014].

[50] M. Harvey e G. De Fabritiis, “Swan: A tool for porting CUDA programs to OpenCL,” *Computer Physics Communications,* pp. 10930-1099, 11 Janeiro 2011.

[51] M. J. Harvey, G. Giupponi e G. De Fabritiis, “ACEMD: Accelerated molecular dynamics simulations in the microseconds timescale,” *J. Chem. Theory Comput.,* 2009.

[52] “CU-2-CL Automating CUDA-to-OpenCL Translation,” Synergy Lab, Virginia Tech, 2014. [Online]. Available: http://chrec.cs.vt.edu/cu2cl/index.php. [Acedido em Julho 2014].

[53] “clang: a C language family frontend for LLVM,” [Online]. Available: http://clang.llvm.org/. [Acedido em Julho 2014].

[54] T. L. C. Infrastructure, “University of Illinois at Urbana-Champaign - Computer Science Department,” [Online]. Available: http://www.llvm.org/. [Acedido em setembro 2014].

[55] “CU2CL - Automating CUDA-to-OpenCL Translation,” Synergy Lab, Virginia Tech, [Online]. Available: http://chrec.cs.vt.edu/cu2cl/overview.php. [Acedido em março 2015].

[56] M. Gardner, P. Sathre, W.-c. Feng e G. Martinez, “Characterizing the challenges and evaluating the efficacy,” *Parallel Computing,* pp. 769-786, 1 Outubro 2013.

[57] “Rodinia:Accelerating Compute-Intensive Applications with Accelerators,” University of Virginia - Department of Computer Science, [Online]. Available: http://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Rodinia:Accelerating_Compute-Intensive_Applications_with_Accelerators. [Acedido em setembro 2014].

[58] F. Jacob, D. Whittaker, S. Thapaliya, P. Bangalore, M. Mernik e J. Gray, “CUDACL: A Tool for CUDA and OpenCL Programmers,” *2010 International Conference on High Performance Computing (HiPC),* 2010.

[59] “APP SDK – A Complete Development Platform,” AMD, [Online]. Available: http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/. [Acedido em março 2015].

[60] “CodeXL – Powerful Debugging, Profiling & Analysis,” Advanced Micro Devices, Inc., [Online]. Available: http://developer.amd.com/tools-and-sdks/opencl-zone/codexl/. [Acedido em março 2015].

[61] A. Developers, “OpenCL Optimization Guide,” Advanced Micro Devices, Inc., [Online]. Available: http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/opencl-optimization-guide/#50401315_pgfId-502946. [Acedido em março 2015].

[62] N. C. Zone, “How to Optimize Data Transfers in CUDA C/C++,” NVIDIA, [Online]. Available: http://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/. [Acedido em março 2015].

[63] “APP Profiler Kernel Occupancy,” Advanced Micro Devices, Inc., [Online]. Available: http://developer.amd.com/tools-and-sdks/archive/amd-app-profiler/user-guide/app-profiler-kernel-occupancy/. [Acedido em março 2015].

[64] A. A. P. Processing, “Southern Islands Series Instruction Set Architecture,” dezembro 2012. [Online]. Available:

http://developer.amd.com/wordpress/media/2012/12/AMD_Southern_Islands_Instruction_Set_Architecture.pdf. [Acedido em fevereiro 2015].

[65] “Unified Memory in CUDA 6,” NVIDIA, [Online]. Available: http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/. [Acedido em março 2015].