FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Utilização de Computação Heterogénea na Codificação de Vídeo

João Paulo Abreu Nogueiro Estevinho

Mestrado Integrado em Engenharia Informática e Computação

Orientador: Jorge Manuel Gomes Barbosa

Julho de 2013

Utilização de Computação Heterogénea na Codificação de Vídeo

João Paulo Abreu Nogueiro Estevinho

Mestrado Integrado em Engenharia Informática e Computação

Aprovado em provas públicas pelo Júri:

Presidente: Pedro Alexandre Guimarães Lobo Ferreira Souto (Professor Auxiliar)

Vogal Externo: Luís Paulo Peixoto dos Santos (Professor Auxiliar)

Orientador: Jorge Manuel Gomes Barbosa (Professor Auxiliar)

_____

17 de Julho de 2013

Resumo

O processamento de Vídeo é uma área em que a performance tem uma elevada importância devido à necessidade de garantir que o sistema é capaz de processar vídeo em tempo real com a melhor qualidade possível. Atualmente, com o desenvolvimento das tecnologias de computação paralela e de computação heterogénea com o GPU, surgem novas possibilidades para o desenvolvimento de *software* de codificação de vídeo de alta performance. O MPEG-2 IMX é um formato de codificação de vídeo popular na indústria de vídeo utilizado principalmente nas máquinas de filmar XDCAM.

O objetivo deste trabalho consistiu em estudar as aplicações da computação heterogénea na área da codificação de vídeo, mais concretamente no algoritmo de codificação do MPEG-2 IMX.

Neste projeto analisou-se o algoritmo de codificação do FFmpeg e desenvolveram-se duas novas soluções baseadas neste algoritmo utilizando computação paralela com o CPU e computação heterogénea com o CPU e GPU recorrendo ao OpenMP e CUDA. Desenvolveu-se ainda uma ferramenta para analisar o processo e calcular a distribuição ótima do trabalho entre o CPU e o GPU.

Os resultados obtidos demonstram que a performance da implementação heterogénea permitiu ganhos de performance entre os 9 e os 30% em relação à implementação mais rápida utilizando apenas o CPU.

Atendendo aos resultados obtidos pode concluir-se que os GPUs são uma ferramenta poderosa que pode ser explorada na codificação de vídeo para baixar a fasquia da codificação em tempo real.

Abstract

Video processing is an area in which performance is extremely important, as it is essential to have solutions that are capable of processing video in real time with the best quality possible. Nowadays with the intensive development of parallel computing and heterogeneous computing with the GPU, there are new possibilities to develop high performance video encoding software.

The objective of this work was to study the possible applications of heterogeneous computing in video encoding, most specifically in the MPEG-2 IMX encoding algorithm. The MPEG-2 IMX format is a very popular video format in the broadcast industry, it is used in XDCAM camcorders that are widespread in this area.

In this project we analysed the FFmpeg implementation of the MPEG-2 IMX encoding algorithm and developed two new solutions based on this implementation using OpenMP and CUDA. The first solution uses parallel computing with the CPU while the second one uses heterogeneous computing with the CPU and GPU. We also developed a profiling tool to automatically calculate the optimum amount of work that should be offloaded to the GPU.

The registered results show a performance gain of 9% to 30% when comparing the hybrid solution to the fastest solution that uses only the CPU.

From this work we concluded that GPUs are powerful computational resources in the video encoding area. They have great potential for further development in speeding up encoding tasks.

Agradecimentos

Quero agradecer a todos aqueles que me ajudaram a realizar esta dissertação.

Ao meu orientador Professor Jorge Barbosa, à MOG, aos meus supervisores o Eng. Pedro Ferreira e ao Eng. João Coelho, o meu agradecimento pela disponibilidade e apoio.

A todos aqueles que influenciaram o meu percurso, à minha família, aos meus amigos e aos meus colegas os meus agradecimentos.

João Estevinho

“Software is like entropy. It is difficult to grasp, weighs nothing, and obeys the second law of thermodynamics; i.e. it always increases.”

Norman Ralph Augustine

Conteúdo

1. Introdução 1
1.1 Contexto 2
1.2 Motivação 2
1.3 Objetivos 3
1.4 Estrutura da Dissertação 4

2. Computação Heterogénea 5
2.1 Arquitetura de Processadores 5
2.1.1 Arquitetura de von Neuman 5
2.1.2 Central Processing Unit (CPU) 6
2.1.3 Graphics Processing Unit (GPU) 9
2.1.4 Coprocessadores 12
2.2 Modelos de Computação Paralela 13
2.2.1 Memória Partilhada 14
2.2.2 Memória Distribuída 15
2.2.3 Paralelismo de Dados 15
2.2.4 Híbrido 16
2.3 Computação utilizando o GPU 16
2.4 APIs para Computação Paralela 18
2.4.1 *Threads* Nativos 18
2.4.2 OpenMP 18
2.4.3 Intel Threading Building Blocks 19
2.4.4 Comparação das APIs de Computação Paralela 19
2.5 APIs para Computação Heterogénea com o GPU 20
2.5.1 OpenCL 20
2.5.2 CUDA 21
2.5.3 C++ AMP 22
2.5.4 OpenACC 23
2.5.5 Comparação das APIs de Computação Heterogénea 23
2.6 Conclusões 24

3. Computação Heterogénea na Codificação de Vídeo 25
3.1 Codificação de Vídeo com o GPU 25
3.2 Implementações do Algoritmo de Codificação do MPEG-2 27
3.2.1 Implementação em Paralelo do MPEG-2 27
3.2.2 Implementação do *codec* MPEG no coprocessador Tilera 64 27
3.3 Bibliotecas de codificação 28
3.3.1 Aceleração com o GPU do *codec* MPEG-2 28
3.3.2 Aceleração com o GPU de outros *codecs* de vídeo 29
3.4 Conclusões 30

4. Algoritmo de Codificação: MPEG-2 IMX 31
4.1 Análise do MPEG-2 IMX 31
4.1.1 Estrutura do *Stream* 32
4.1.2 Etapas de Codificação 33
4.2 FFmpeg 37
4.2.1 Algoritmo de Codificação 37
4.2.2 Profiling 41

5. Paralelização do Algoritmo 45
5.1 Solução utilizando Computação Paralela no CPU 45
5.1.1 Estrutura de dados 46
5.1.2 Paralelismo e Alocação das tarefas 47
5.1.3 Bit *Buffer* 49
5.2 Solução utilizando Computação Heterogénea com o CPU e GPU 50
5.2.1 Algoritmo 50
5.2.2 Paralelismo e Alocação de Tarefas 51
5.2.3 Cópia da memória para o GPU 52
5.2.4 Sobreposição de tarefas / *Pipelining* 54
5.2.5 *Memory coallescing* e Memória Partilhada 55
5.2.6 Ocupação do GPU e Alocação dos Threads 57
5.2.7 Otimização da Performance 58

6. Resultados e Discussão 61
6.1 Metodologia 61
6.2 Processamento Sequencial 63
6.3 Processamento em Paralelo com o CPU 64
6.4 Processamento em Paralelo com o CPU e GPU 67
6.5 Ferramenta de Otimização da Performance 70
6.6 Escalabilidade 71
6.6.1 Evolução da performance com o CPU 71
6.6.2 Evolução da performance com o GPU 72

7. Conclusões e Trabalho Futuro 75
7.1 Trabalho Realizado e Satisfação dos Objetivos 76
7.2 Trabalho Futuro 76
Referências 77
Anexos 81

Lista de Figuras

Figura 1: Logotipo e missão da MOG. 2
Figura 2: (a) Fotograma codificado utilizando codificação sem perdas; (b) Fotograma codificado utilizando codificação com perdas de alta qualidade; (c) Fotograma codificado utilizando codificação com perdas de baixa qualidade. 3
Figura 3: Diagrama de componentes da arquitetura de von Neuman.[1] 6
Figura 4: Gráfico comparativo do número de transístores nos CPUs desenvolvidos entre 1971 e 2004 e a Lei de Moore. [2] 7
Figura 5: Registos de 64 bits MMX contendo os 4 tipos de dados introduzidos pelo MMX. 7
Figura 6: Diagrama exemplificativo do funcionamento de um CPU com Intel Hyper-Threading.[5] 8
Figura 7: (a) Modelo de Memoria Partilhada, (b) Modelo de Memoria Distribuída. [2] 9
Figura 8: Diagrama da arquitetura de um computador com GPU dedicado. [7] 9
Figura 9: Diagrama das trocas de contexto entre threads no GPU para esconder a latência da memória. [9] 10
Figura 10: Performance relativa do GPU NVIDIA GTX280 em relação ao CPU Intel Core i7-960 em diferentes algoritmos de processamento em paralelo. [13] 11
Figura 11: Modelos de execução num ambiente de computação heterogénea com dois CPU Intel Xeon e um Coprocessador Intel Xeon Phi. [14] 12
Figura 12: Diagrama dos modos de ligação de um coprocessador: (a) à esquerda no modo de rede como um nó Linux (b) à direita no modo coprocessador. [15] 13
Figura 13: Tarefa sequencial complexa. 13
Figura 14: Tarefa sequencial complexa dividida em tarefas paralelizáveis. 13
Figura 15: Diagrama representativo do modelo de memória partilhada. 14
Figura 16: Exemplo de ocorrência de *false sharing* quando *threads* em diferentes processadores acedem à mesma linha de *cache*. [16] 15
Figura 17: Diagrama representativo do modelo de memória distribuída. 15
Figura 18: Diagrama do processamento de uma matriz em blocos com paralelismo de dados. 16
Figura 19: Diagrama representativo do modelo hibrido. 16

Figura 20: Diagrama exemplificativo da interação CPU/GPU para Computação no GPU. [19] 17
Figura 21: Modelo de processamento em *streams* em que um *stream* de saída é o resultado do processamento de um *kernel* num *stream* de entrada.[21] 17
Figura 22: Utilização de um pragma do OpenMP para paralelizar um ciclo for em C. 18
Figura 23: Tabela com uma lista de funções do Intel TBB. [25] 19
Figura 24: Conversão de uma função desenvolvida em C para um Kernel do OpenCL. O ciclo for de 0 a n é transformado em n *threads* do *kernel.* [28]. 21
Figura 25: Fluxo de processamento da plataforma CUDA. [30] 21
Figura 26: Conversão de uma função desenvolvida em C para um *kernel* CUDA. O ciclo for de 0 a n é transformado em *threads* do *kernel* a serem executadas nos núcleos CUDA. [29] 22
Figura 27: Exemplo da soma de dois *arrays* utilizando C++ AMP. [31] 22
Figura 28: Paralelização de uma função sequencial utilizando OpenACC. [32] 23
Figura 29: Divisão do algoritmo de codificação do MPEG-2. [36] 27
Figura 30: Diagrama da rede de núcleos do processador Tilera e alocação dos *slices*. [37] 28
Figura 31: Arquitetura do Intel Media SDK 2013. [38] 29
Figura 32: Diagrama do funcionamento do *wrapper* do *codec* H.264 da MainConcept.[42] 30
Figura 33: Hierarquia de um *stream* MPEG [44]. 32
Figura 34: Diagrama de dependências das *frames* do tipo I, P e B. [47] 33
Figura 35: Diagrama de blocos do algoritmo de codificação do MPEG-2 IMX. [47] 33
Figura 36: Equações de conversão entre RGB e YCbCr. 34
Figura 37: Componentes RGB de uma *frame*. [47] 34
Figura 38: Componentes YCbCr de uma *frame*. [47] 34
Figura 39: Diferença de codificação entre um espaço de cor RGB e um espaço de cor YCbCr 4:2:0. [48] 34
Figura 40: Função do cálculo de uma DCT de duas dimensões. [47] 35
Figura 41: Conversão de uma matriz YCbCr de um bloco para uma matriz de coeficientes DCT. [50] 35
Figura 42: Função de quantização. [47] 36
Figura 43: Quantização de uma matriz de coeficientes DCT de um bloco. [50] 36
Figura 44: Zig-Zag de uma matriz de quantização de um bloco. [50] 37
Figura 45: *Run-Length Encoding* de uma matriz de quantização reordenada em Zig-Zag. [50] 37
Figura 46: Diagrama da implementação do *codec* MPEG-2 IMX do FFmpeg. 38
Figura 47: Diagrama do algoritmo de DCT bidimensional baseado numa passagem do algoritmo unidimensional em cada dimensão. 40

Figura 48: Diagrama do algoritmo de quantização com pesquisa do último elemento quantizado num valor diferente de zero. 40
Figura 49: *Clipping* de uma matriz de quantização de um bloco. 41
Figura 50: Gráfico representativo da distribuição de tempo despendido no programa de codificação de vídeo. 42
Figura 51:Gráfico do *speedup* de Amdahl do programa de codificação em função do número de unidades de processamento. 44
Figura 52: Gráfico do *speedup* de Amdahl do *codec* MPEG-2 em função do número de unidades de processamento. 44
Figura 53: Diagrama do algoritmo de codificação MPEG-2 IMX implementado utilizando paralelismo de dados para N *threads* do CPU. Neste diagrama apenas estão exemplificados 2 *threads*. 46
Figura 54: Diagrama UML de classes das estruturas implementadas. 47
Figura 55: Diagrama da utilização e tempo de vida dos *threads* no *codec* implementado. Na figura está representado o funcionamento do *codec* num processador com quatro núcleos. 48
Figura 56: Excerto de código da implementação de um bloco de código em paralelo com OpenMP utilizando um *pragma* para distribuir as iterações de um ciclo *for* entre os *threads*. 48
Figura 57: *Frame* dividida em 4 *chunks* constituídos por *slices* horizontais. Na imagem, o *chunk* superior está dividido numa grelha correspondente aos *slices* e macroblocos que o constituem. 49
Figura 58: Diagrama exemplificativo do processo de escrita de 8 bits para o *buffer* de bits. 49
Figura 59: Diagrama do algoritmo de codificação MPEG-2 IMX implementado utilizando paralelismo de dados para N *threads* de processamento do CPU e o GPU utilizando a plataforma CUDA. A caixa verde contém operações processadas na plataforma CUDA, todas as outras são processadas por defeito pelo CPU. 50
Figura 60: Diagrama da utilização e tempo de vida dos *threads* da solução implementada. A figura representa o funcionamento do *codec* num sistema com quatro processadores lógicos e um GPU CUDA controlado pelo *thread* 4. 51
Figura 61:Excerto de código da implementação de tarefas com etapas distribuídas pelo CPU e CPU + GPU. 52
Figura 62: Diagrama e excerto de código utilizado para a cópia síncrona de memória paginável. 53
Figura 63: Diagrama e excerto de código utilizado para a cópia assíncrona através de DMA de um endereço de memória não paginável para a memória do GPU. 53

Figura 64: Diagrama e excerto de código utilizado para o acesso direto do GPU à memória RAM através de DMA. 54
Figura 65: Diagrama de uma linha temporal de um *pipeline* de três *streams* CUDA. As operações de um *stream* são processadas sequencialmente pela ordem que são registadas. Os *streams* são processados concorrentemente à medida que os recursos são disponibilizados. 54
Figura 66: Excerto de código utilizado para a implementação de tarefas sobrepostas na plataforma CUDA. 55
Figura 67: Diagrama do modelo de acesso não coalescido à memória pelos *threads* que processam os blocos de vídeo. 56
Figura 68: Diagrama do modelo de acesso coalescido à memória da solução implementada utilizando memória partilhada. 57
Figura 69: Código utilizado para a cópia coalescida de um *array* de objetos para a *cache* e o processamento de cada um dos objetos num *thread* independente. 57
Figura 70: Divisão de um *slice* de 16 macroblocos por dois SMP cada um com capacidade para armazenar 8 macroblocos na memória partilhada. Cada bloco é processado por um *thread* independente. 58
Figura 71: Algoritmo para explorar e encontrar a distribuição ótima de *slices*. 59
Figura 72: Exemplo de uma execução do programa de optimização de performance. 60
Figura 73:Diagrama do esquema utilizado para a medição do tempo total de execução e tempo de codificação. 62
Figura 74: Primeira *frame* do vídeo de teste utilizado 62
Figura 75: Comparação da performance do *codec* do FFmpeg e da solução de computação paralela implementada quando executadas sequencialmente em apenas um processador. 64
Figura 76: Tempo de codificação com múltiplos *threads* no Computador 1 64
Figura 77: Tempo de codificação com múltiplos *threads* no Computador 2 65
Figura 78: *Speedup* da solução implementada utilizando *n threads* no Computador 1. 65
Figura 79: *Speedup* da solução implementada utilizando *n thread* no Computador 2. 66
Figura 80: Tempo de codificação no Computador 1 em função do número de *slices* processados no GPU. 67
Figura 81: Tempo de codificação no Computador 2 em função do número de slices processados no GPU. 67
Figura 82:*Speedup* em função do número de *slices* alocados ao GPU na solução de dois *threads* de processamento no Computador 1. 69
Figura 83: *Speedup* em função do número de *slices* alocados ao GPU na solução de três *threads* de processamento mais no Computador 2. 69
Figura 84: Resultados da ferramenta de otimização no Computador 1 70
Figura 85: Resultados da ferramenta de otimização no Computador 2 70

Lista de Tabelas

Tabela 1: Comparação das funcionalidades de *threads* nativos, OpenMP e Intel TBB. [26] 20
Tabela 2: Comparação das funcionalidades das plataformas CUDA, OpenCL, C++ AMP e OpenACC. 24
Tabela 3: Valores de *speedup* obtidos para a solução implementada em função do número de *tiles*. 28
Tabela 4: Percentagem do tempo de codificação por etapa. 42
Tabela 5: Tabela representativa das dependências de dados do algoritmo do FFmpeg. 42
Tabela 6: Especificações dos sistemas de teste utilizados. 63

Abreviaturas e Símbolos

ALU	Arithmetic Logic Unit
API	Application Programming Interface
CC	Compute Capability
CPU	Central Processing Unit
CU	Control Unit
DMA	Direct Memory Access
DCT	Discrete Cosine Transform
FPU	Floating-Point Unit
GOP	Group of Pictures
GPGPU	General-Purpose computing on Graphics Processing Units
GPU	Graphics Processing Unit
HDTV	High-Definition Television
MIMD	Multiple Instruction, Multiple Data
MISD	Multiple Instruction, Single Data
MPEG	Moving Picture Experts Group
MXF	Material Exchange Format
RGB	Red Green and Blue
RLE	Run-Length Encoding
SDK	Software Development Kit
SIMD	Single Instruction, Multiple Data
SISD	Single Instruction, Single Data
SMP	Streaming Multiprocessor
SP	Stream Processor
SSE	Streaming SIMD Extensions
VLC	Variable-Length Coding

Capítulo 1

Introdução

A televisão surgiu nos finais dos anos vinte como um meio de comunicação que permitia transmitir imagem em movimento para dispositivos recetores e subsequentemente reproduzi-la num ecrã de raios catódicos. Com o aparecimento da televisão surgiu a necessidade de armazenar a imagem em movimento num meio físico que permitisse não só conservar a imagem para momentos posteriores como também desassociar os momentos de produção e transmissão dos conteúdos, eliminando a necessidade do "direto".

O primeiro dispositivo capaz de armazenar sequências de imagens surgiu no início dos anos 30, o *kinescope*, que foi rapidamente adotado pelo mundo da televisão. Anos mais tarde, em 1951 surgiram as cassetes de vídeo que permitiam gravar, copiar, transmitir e reproduzir sinal de vídeo analógico composto de uma forma barata e eficaz utilizando uma fita magnética.

Com o evoluir da tecnologia surgiram novas possibilidades para armazenar sinais de vídeo utilizando equipamentos digitais através da conversão do sinal analógico para uma representação digital discreta, do mesmo, compatível com diferentes equipamentos. A necessidade de compatibilidade entre equipamentos deu origem à especificação de diferentes formatos de vídeo, cada qual com diferentes características e posicionamento, tais como qualidade mais elevada ou menor tamanho do vídeo codificado.

Hoje em dia, no mundo da televisão, é necessário garantir a mais alta qualidade com dimensões tão reduzidas quanto possível numa solução em tempo real de gravação e reprodução de vídeo. A computação paralela e heterogénea são duas áreas em forte expansão que permitem a execução num computador de diferentes linhas de processamento de forma concorrente utilizando não só o CPU como também o GPU que podem ser aplicadas de forma a acelerar as tarefas de processamento de vídeo permitindo melhores níveis de qualidade em tempo real.

1.1 Contexto

O tema desta dissertação foi proposto pela MOG, empresa que desenvolve soluções para ambientes de pós-produção como ferramentas centralizadas de *Ingest* e de manipulação de *containers* MXF. O MXF consiste num *wrapper* normalizado de áudio e vídeo que facilita a transmissão de ficheiros e adição de *metadata*. A MOG coopera há mais de uma década com varias cadeias de televisão internacionais, facilitando os seus *workflows* com soluções altamente flexíveis com suporte para um variado leque de formatos, sistemas e plataformas. A missão da MOG é superar os desafios do mundo do vídeo e oferecer soluções totalmente interoperáveis para a indústria da televisão.

Figura 1: Logotipo e missão da MOG.

Com a evolução da computação paralela e heterogénea torna-se essencial estudar formas de aplicar estas tecnologias na área das soluções de vídeo por forma a obter ganhos de desempenho e encontrar novas maneiras de superar os obstáculos existentes.

1.2 Motivação

Codificação de vídeo é o processo de converter um sinal de vídeo para um formato digital *standard* que possa ser armazenado, reproduzido ou editado. As *bit-rates* de vídeo não comprimido, em formato *raw*, são muito altas podendo no caso da HDTV atingir valores da ordem de 1 Gbps e no caso da nova tecnologia de 4K na ordem dos 8 Gbps. A codificação de vídeo pode ser realizada em dois tipos diferentes de qualidade, a codificação sem perdas *(lossless)* e a codificação com perdas *(lossy)* exemplificados na Figura 2. A codificação sem perdas oferece a melhor qualidade utilizando no entanto *bit-rates* muito elevados, próximos dos sinais não comprimidos. Por outro lado a codificação com perdas oferece menor qualidade utilizando no entanto *bit-rates* muito inferiores.

Figura 2: (a) Fotograma codificado utilizando codificação sem perdas; (b) Fotograma codificado utilizando codificação com perdas de alta qualidade; (c) Fotograma codificado utilizando codificação com perdas de baixa qualidade.

Relativamente à codificação com perdas, este tipo de codificação consiste em comprimir os dados, ignorando dados de menor relevância que não são codificados, oferecendo diferentes níveis de qualidade dependendo do grau de sensibilidade com que o algoritmo vai descartar detalhes tal como evidenciado nos diferentes níveis de qualidade apresentados na Figura 2. De uma forma geral o processo, de codificação com perdas, vai realizar uma maior quantidade de processamento quanto melhor for a qualidade de codificação definida.

O MPEG-2 é um formato de codificação com perdas muito utilizado no mundo da televisão que requer o melhor compromisso entre a qualidade da imagem e o tempo de processamento necessário, para que o processo de codificação possa ser realizado em tempo real. Uma das principais implementações do algoritmo MPEG-2 é a do projeto *open-source* FFmpeg que tem no entanto limitações em termos de performance pela implementação sequencial do *codec* que não tira partido de todas as capacidades de computação paralela e heterogénea dos computadores atuais.

1.3 Objetivos

Os objetivos desta dissertação foram a realização de um estudo do algoritmo MPEG-2 e da sua implementação no FFmpeg para implementar uma solução otimizada deste algoritmo utilizando as capacidades de computação paralela existentes nos computadores atuais.

Em suma, os objetivos desta dissertação são:

- Estudar o algoritmo de codificação do MPEG-2 IMX;
- Estudar o modelo computacional do *codec* MPEG-2 IMX do FFmpeg;
- Implementar o *codec* MPEG-2 IMX utilizando computação paralela com o CPU;

- Implementar o *codec* MPEG-2 IMX utilizando computação heterogénea com o CPU e GPU.

1.4 Estrutura da Dissertação

Para além deste capítulo introdutório, este documento contém mais 5 capítulos. No capítulo 2, Computação Heterogénea, é descrito o estado da arte no âmbito das arquiteturas de computadores e computação paralela e no capítulo 3 são apresentados trabalhos relacionados sobre a aplicação de computação heterogénea na codificação de vídeo. O capítulo 4, Algoritmo de Codificação, apresenta uma análise à especificação do algoritmo de codificação em estudo e à implementação deste algoritmo no FFmpeg. O capítulo 5 descreve e especifica as soluções desenvolvidas e o capítulo 6 apresenta uma discussão em torno dos resultados obtidos para cada uma das soluções implementadas. Para concluir, o capítulo 7, aborda as principais conclusões alcançadas no desenvolvimento deste estudo e uma análise ao trabalho futuro que poderá ser realizado nesta área.

Capítulo 2

Computação Heterogénea

Neste capítulo realiza-se uma reflexão sobre o estado da arte no âmbito da computação paralela e heterogénea. Numa primeira parte abordam-se os principais detalhes de arquitetura de computadores que propiciam o desenvolvimento de *software* utilizando computação paralela. De seguida, é feita uma análise sobre os modelos de computação paralela utilizados para o desenvolvimento de *software* e a sua aplicação nos diferentes tipos de arquiteturas. Concluída esta análise abordam-se os principais conceitos de computação no GPU e as APIs de computação paralela utilizadas tanto a nível do CPU como do GPU para o desenvolvimento de *software* paralelo.

2.1 Arquitetura de Processadores

2.1.1 Arquitetura de von Neuman

Os computadores atuais utilizam uma arquitetura baseada no *design* da máquina de von Neuman, especificada em 1945 por John von Neuman. A partir dai, os computadores começaram a ser desenvolvidos utilizando este princípio de *design* que descreve um computador subdividido em três componentes essenciais como exemplificado na Figura 3. A primeira componente é a unidade de processamento central que inclui a unidade de lógica aritmética, os registos do processador, e uma unidade de controlo que contém um registo de instruções e um contador do programa e é responsável por descodificar as instruções dos programas e de coordenar a sua execução de forma sequencial. As restantes componentes são a memória de acesso aleatório (RAM) que armazena os dados utilizados e as instruções dos programas que são executados na máquina e a interface de entrada/saída que possibilita a ligação a *hardware* especializado e periféricos [1].

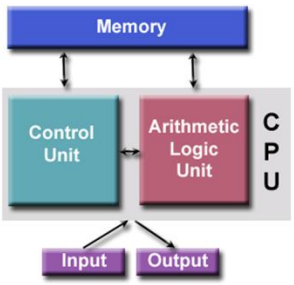


Figura 3: Diagrama de componentes da arquitetura de von Neuman.[1]

Os sistemas de computação paralela seguem o mesmo princípio possuindo no entanto várias unidades com este funcionamento ao invés de uma única, podendo executar ao mesmo tempo diferentes dados e instruções.

2.1.2 Central Processing Unit (CPU)

A Unidade Central de Processamento (CPU) é um componente que executa as instruções de um programa executando as tarefas aritméticas, lógicas e de E/S do sistema. Como já foi referido anteriormente o CPU está dividido em duas componentes a Unidade Lógica Aritmética (ALU) que executa as tarefas aritméticas e lógicas das operações e a Unidade de Controlo (CU). O primeiro microprocessador foi o modelo 4004 de 4 *bits* produzido pela Intel no início dos anos setenta. A cada geração os processadores foram-se tornando mais pequenos, mais rápidos e mais eficientes energeticamente. Os CPUs mais utilizados em computadores utilizam a arquitetura x86 de 32 bits introduzida pela Intel ou x64 de 64bits desenvolvida pela AMD [2].

2.1.2.1 Lei de Moore

Em 1965, Gordon Moore fundador da Intel, publicou aquilo que viria a ser conhecido como a Lei de Moore (Moore's Law) na qual referiu que o número de transístores num *chip* vai aproximadamente duplicar a cada ano. Em 1975 depois do desenvolvimento do 4004 Gordon Moore redefiniu o conceito original para a cada dois anos. Atualmente a lei de Moore é muitas vezes enunciada, segundo a revisão por Dave House, que a performance de um computador duplica a cada 18 meses. Na Figura 4 podemos observar que o número de transístores acompanhou a Lei de Moore no período de 1971 a 2004 [2].

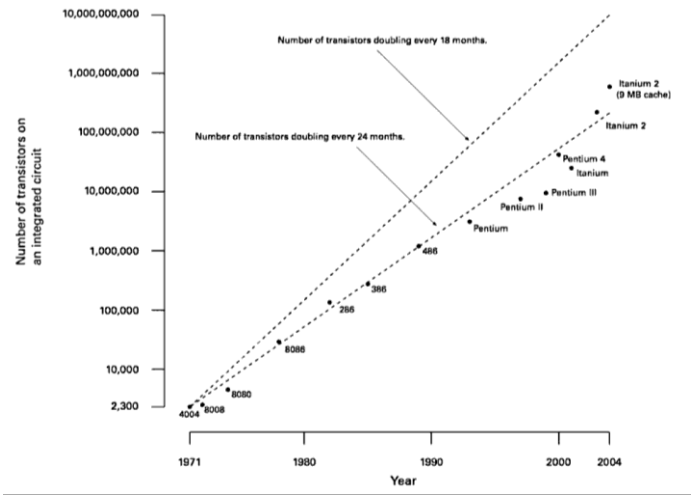


Figura 4: Gráfico comparativo do número de transístores nos CPUs desenvolvidos entre 1971 e 2004 e a Lei de Moore. [2]

Um estudo realizado por W. Knight publicado na revista IEE Review de Setembro de 2005 notou que o consumo energético aumenta 60% com cada aumento de 400MHz na frequência de relógio mas com uma abordagem de dois núcleos é possível atingir um aumento de performance significativo sem a necessidade de aumentar a frequência de relógio [3]. O resultado deste estudo provou que era incomportável continuar a aumentar a performance dos CPUs pelo aumento da frequência de relógio, abrindo assim as portas às arquiteturas *Multi-Core*.

2.1.2.2 Instruções SIMD

A tecnologia *Single Instruction Multiple Data* (SIMD) foi introduzida em 1996 pela Intel sob o nome de MMX, mais tarde revista na extensão SSE e consiste na adição de 8 registos de 64bits ao CPU com suporte para 4 novos tipos de dados utilizados para empacotar variáveis de 32, 16 ou 8 bits e realizar uma operação logica sobre todas as variáveis empacotadas numa só instrução [4]. A Figura 5 apresenta os tipos de dados utilizados nos registos MMX e o respetivo conteúdo.

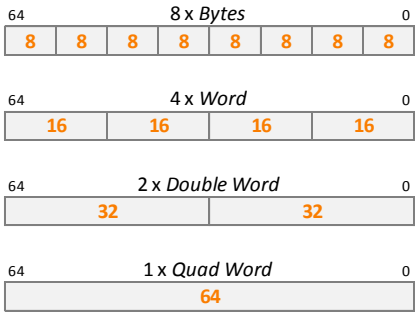


Figura 5: Registos de 64 bits MMX contendo os 4 tipos de dados introduzidos pelo MMX.

2.1.2.3 Multi-Threading

A tecnologia de *Multi-Threading* consiste num tipo de arquitetura em que o CPU tem suporte de hardware para a execução de múltiplos threads no mesmo núcleo. Esta tecnologia procura melhorar a utilização das capacidades de processamento do CPU reduzindo tempos mortos de acesso à memória ou ao disco que têm latências elevadas, ou seja, quando ocorre um *cache miss* o CPU muda de contexto e executa outro *thread* até que os recursos necessários estejam disponíveis ou ocorra outro *cache miss* [5].

Esta tecnologia pode ainda ser utilizada em sistemas *multi-core*. O *Hyper-Threading*, implementação da Intel do *Simultaneous Multi-Threading*, simula a existência de dois processadores virtualmente duplicando o número de núcleos disponíveis num CPU. A Figura 6 exemplifica o funcionamento da tecnologia *Hyper-Threading* [5].

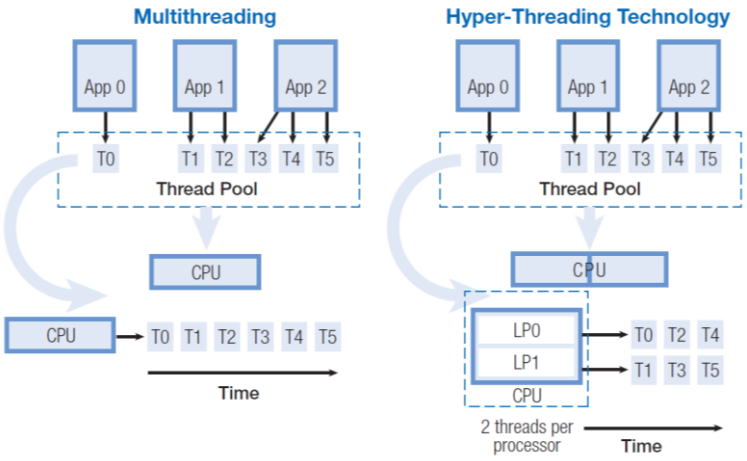


Figura 6: Diagrama exemplificativo do funcionamento de um CPU com Intel Hyper-Threading.[5]

2.1.2.4 Multi-Core

Um processador *multi-core* consiste num processador do tipo MIMD constituído por um circuito integrado constituído por várias unidades de processamento com ALU e UC independentes e capazes de executar um programa em paralelo. Desta forma, um processador *multi-core* é o equivalente a ter um computador com vários CPUs independentes apesar de a performance ser ligeiramente inferior a uma arquitetura com múltiplos CPUs, contudo os consumos energéticos são também inferiores [2].

Atualmente são utilizados dois tipos de comunicação entre núcleos num processador *multi-core*, o modelo de memória partilhada em que todos os núcleos têm acesso ao mesmo espaço de memoria, utilizado nos CPU mais comuns e o modelo de memória distribuída em que cada núcleo tem memória independente e a transferência de dados entre núcleos é realizada através

da troca de mensagens utilizando a rede interna ao processador. A Figura 7 apresenta um diagrama dos dois modelos anteriormente referidos [2].

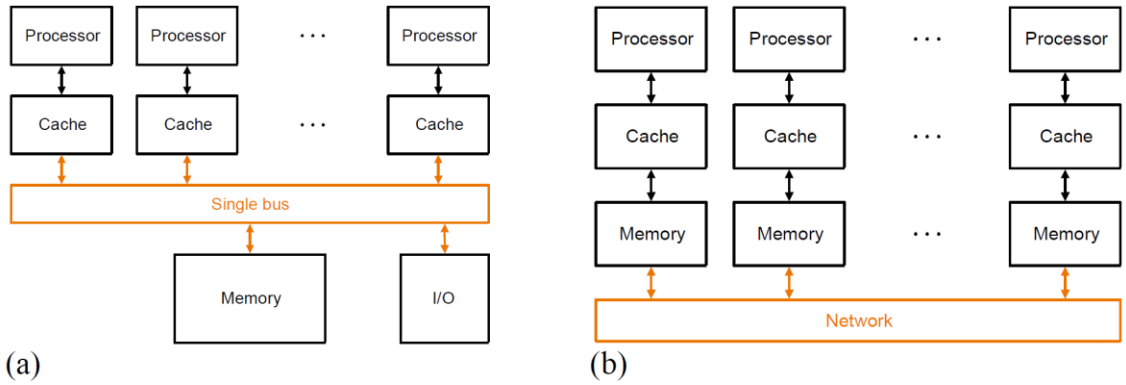


Figura 7: (a) Modelo de Memoria Partilhada, (b) Modelo de Memoria Distribuída. [2]

2.1.3 Graphics Processing Unit (GPU)

A Unidade de Processamento Gráfico (GPU) é um componente especializado desenhado para alterar e manipular memória para acelerar a criação de imagens para o processamento das *frames*. O diagrama da Figura 8 apresenta a arquitetura de um sistema com uma placa gráfica discreta. Os GPU são constituídos por múltiplos *Streaming Multiprocessors* (SMP) que partilham os mais altos níveis de *cache*. Cada SMP é constituído por vários níveis de cache e memória internos ao SMP e vários Stream Processors (SP) que executam em paralelo e de forma síncrona um *kernel* sobre múltiplos dados [6].

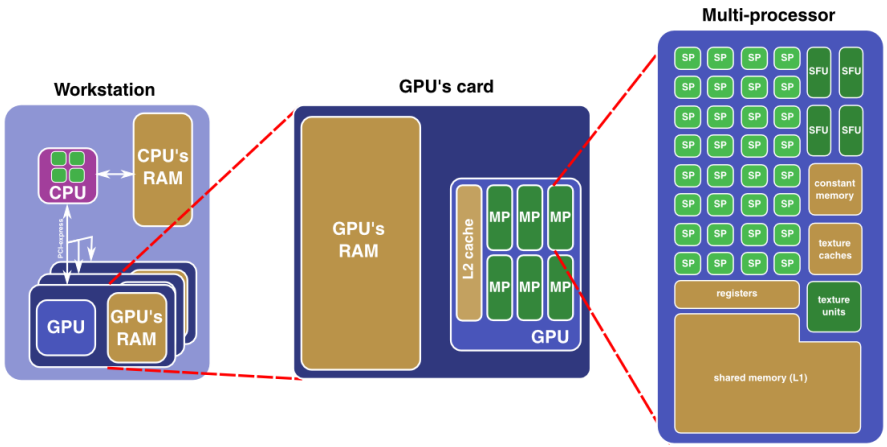


Figura 8: Diagrama da arquitetura de um computador com GPU dedicado. [7]

Esta estrutura de forma altamente paralela torna os GPUs mais eficazes que os CPUs para algoritmos em que seja necessário processar blocos de informação em paralelo.

2.1.3.1 Unidades de processamento

Um GPU é constituído por dois níveis de unidades de processamento:

- *Streaming Multiprocessors* – são as unidades principais de processamento de um GPU e consistem em processadores de 32 bits com uma arquitetura SIMD. Cada SMP é constituído por um conjunto de SPs e uma unidade de controlo de instruções que executa sincronamente a mesma instrução em todos os SPs. Quando os *threads* tomam diferentes saltos condicionais as instruções são serializadas incorrendo numa ligeira perda de desempenho dependendo do número de instruções serializadas [8].Os SMP implementam *multi-threading* de *hardware* escondendo a latência da memória alternando entre *threads* como exemplificado na Figura 9 [9];

- *Stream Processor* – são as unidades de processamento escalar de *threads* do GPU com uma arquitetura *Single Instruction Single Data* (SISD). Os SPs não possuem uma unidade de controlo de instruções, recebendo as instruções do SMP e executando sincronamente a mesma instrução para o *stream* de dados correspondente ao *thread* a ser processado [8].

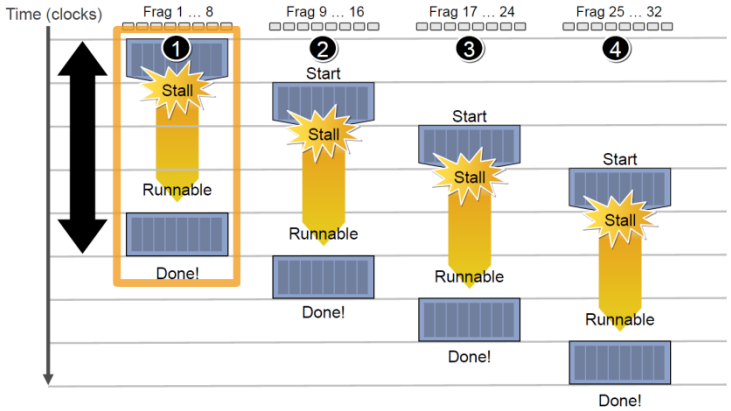


Figura 9: Diagrama das trocas de contexto entre threads no GPU para esconder a latência da memória. [9]

2.1.3.2 Níveis de memória

Um GPU é normalmente constituído pelos seguintes níveis de memória:

- Registos – memória de alta velocidade e reduzida latência integrada no SMP que armazena o contexto dos *threads*. Os registos podem ser explicitamente copiados de e para a memória global caso seja previsto durante a compilação que as necessidades de memória são superiores ao tamanho da memória de registos, reduzindo no entanto a performance [10];

- Memória Partilhada – memória de alta velocidade e reduzida latência integrada no SMP que é partilhada pelos *threads* de um bloco [11]. Esta memória é utilizada explicitamente pelo programador podendo ser utilizada para processar blocos de memória que são utilizados múltiplas vezes [12];

- Memória Global – memória DRAM externa ao GPU, de menor velocidade e alta latência atingindo valores entre os 400 e os 800 ciclos [11]. Para otimizar o acesso à memória global, a maioria dos GPUs requer padrões de acesso à memória em que todos os *threads* de um conjunto acedem a blocos de memória contíguos, "*memory coalescing*", de modo a condensar os vários acessos num único acesso de maiores dimensões [12]. É comum existir um mecanismo adicional de acesso a blocos de memória global declarados como constantes que permite aos SMP armazenar os valores em *cache* e propaga-los pelos SPs [11].

2.1.3.3 Performance CPU vs GPU

Segundo o estudo realizado por Lee comparando a capacidade de processamento do CPU e GPU mais avançados do mercado à data de realização do estudo, o CPU Intel Core i7-960 e o GPU NVIDIA GTX280, o GPU tem em média velocidades de processamento 2,5 vezes superiores às do CPU num conjunto de algoritmos de cálculo intensivo em paralelo. Os valores de performance relativa obtidos neste estudo estão apresentados na Figura 10. Na realização deste estudo Lee conclui ainda que, apesar de terem explorado otimizações do código para cada plataforma, a performance de ambas é limitada pela largura de banda da memória. Por último as otimizações com maior impacto na performance apontadas para cada plataforma foram a reestruturação do código para otimizar o acesso à memória, as funcionalidades de SIMD e *multi-threading* do CPU e a nível do GPU a redução das sincronizações globais, que são muito custosas em termos de processamento [13].

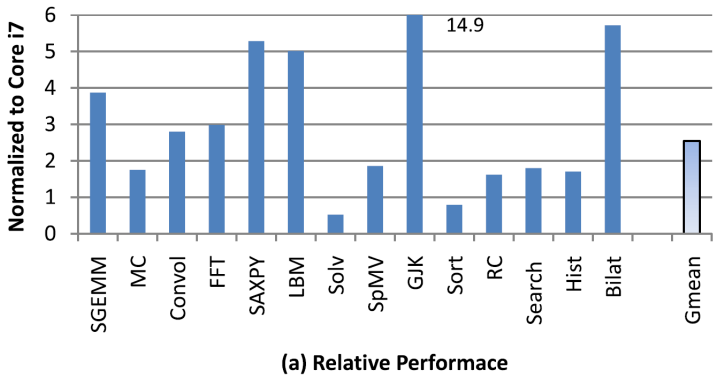


Figura 10: Performance relativa do GPU NVIDIA GTX280 em relação ao CPU Intel Core i7-960 em diferentes algoritmos de processamento em paralelo. [13]

2.1.4 Coprocessadores

O coprocessador é um componente que auxilia o CPU constituído por uma ou mais unidades de processamento *multi-core* de arquitetura x86 ou x64 capazes de processar simultaneamente centenas de *threads*. Uma vez que utilizam a mesma arquitetura dos CPUs convencionais estes sistemas requerem alterações mínimas no código para executarem uma aplicação estando a cargo do programador atribuir tarefas ao coprocessador. A Figura 11 representa um diagrama dos modelos de distribuição de tarefas possível num ambiente de computação heterogénea com coprocessador. Os coprocessadores podem ser configurados para funcionarem como um periférico do sistema anfitrião ou simularem um servidor Linux independente que comunica com o sistema anfitrião por uma interface de troca de mensagens em rede como exemplificado na Figura 12 [14].

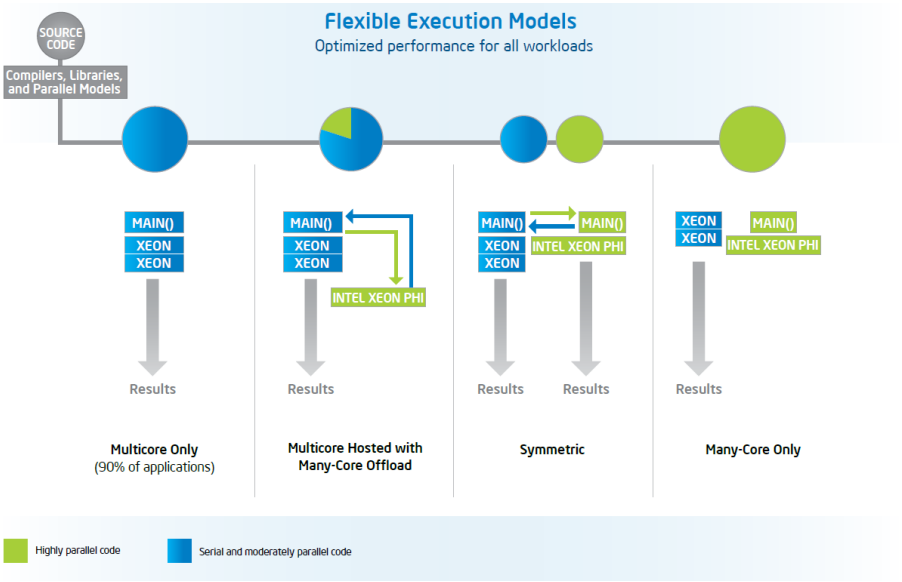


Figura 11: Modelos de execução num ambiente de computação heterogénea com dois CPU Intel Xeon e um Coprocessador Intel Xeon Phi. [14]

Segundo a Intel as aplicações que obtêm maiores vantagens deste tipo de sistemas são aplicações com um elevado número de *threads*, que requeiram uma elevada largura de banda de acesso à memória ou que utilizem cálculo vetorial extensivo [14].

Atualmente existem dois sistemas de coprocessadores de maior destaque no mercado, o Xeon Phi da Intel e o TileEncore da Tilera que podem ser instalados na *slot PCI Express x16* com uma largura de banda de até 16GB/s [15].

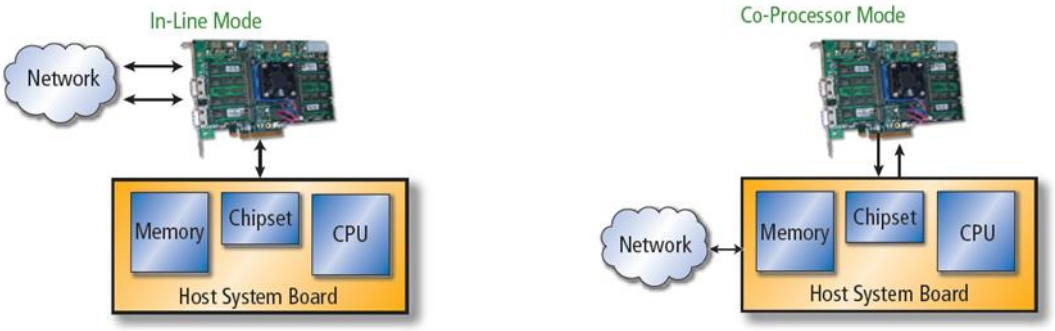


Figura 12: Diagrama dos modos de ligação de um coprocessador: (a) à esquerda no modo de rede como um nó Linux (b) à direita no modo coprocessador. [15]

2.2 Modelos de Computação Paralela

Para tirar proveito das capacidades de processamento em paralelo que possuímos atualmente é necessário decompor tarefas sequenciais complexas em tarefas mais simples que possam ser executadas em paralelo tal como representado na Figura 13 e Figura 14.

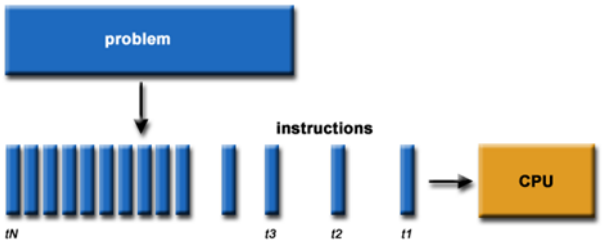


Figura 13: Tarefa sequencial complexa.

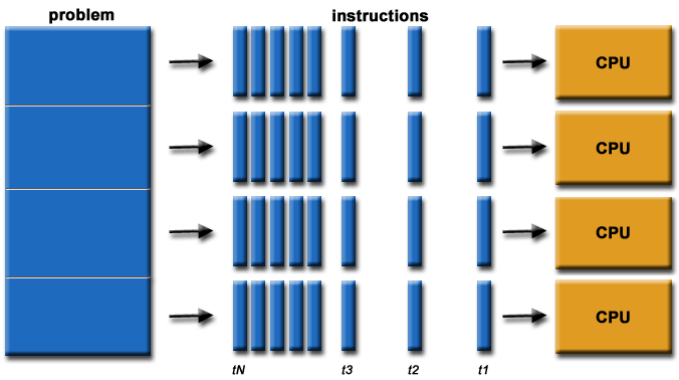


Figura 14: Tarefa sequencial complexa dividida em tarefas paralelizáveis.

Para implementar o processamento das tarefas resultantes da decomposição de uma tarefa complexa deve utilizar-se um modelo de computação paralela que permite ao programador abstrair-se do *hardware* e arquitetura de memória da plataforma alvo.

2.2.1 Memória Partilhada

No modelo de programação paralela de memória partilhada as tarefas partilham o mesmo espaço de memória podendo realizar operações assíncronas sobre estes dados. Neste modelo, o programador não necessita de sincronizar os dados entre as tarefas uma vez que estes são os mesmos, necessitando apenas de mecanismos de controlo como semáforos caso uma tarefa necessite de acesso exclusivo à memória.

Este modelo é geralmente implementado utilizando *threads* assíncronos do mesmo processo como representado na Figura 15.

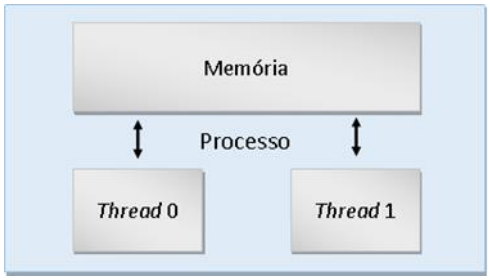


Figura 15: Diagrama representativo do modelo de memória partilhada.

False sharing é um problema muito comum em sistemas que utilizam um modelo de memória partilhada que tem um forte impacto na performance e que ocorre quando *threads* em diferentes processadores modificam variáveis na mesma linha da *cache*, obrigando os processadores a acederem constantemente à memória principal para sincronizarem a linha que está a ser alterada, como exemplificado na Figura 16. Tal facto degrada a performance do sistema devido à menor largura de banda e maiores tempos de acesso à memória em comparação com a *cache* [16].

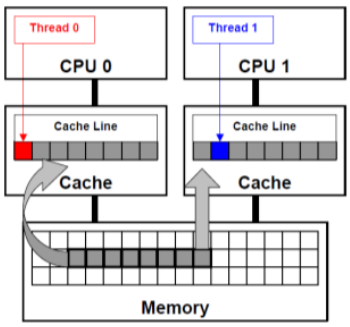


Figura 16: Exemplo de ocorrência de *false sharing* quando *threads* em diferentes processadores acedem à mesma linha de *cache*. [16]

2.2.2 Memória Distribuída

No modelo de programação paralela de memória distribuída as tarefas não partilham o mesmo espaço de memória existindo em cada tarefa uma cópia dos dados. Uma vez que as tarefas têm o seu espaço de memória independente surge a necessidade de realizar de forma síncrona operações de sincronização de dados através do envio e receção de mensagens.

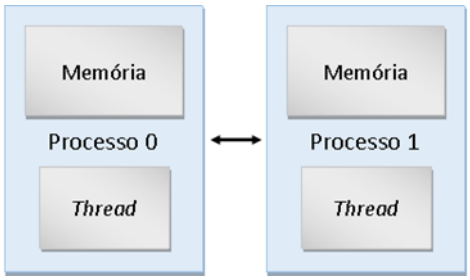


Figura 17: Diagrama representativo do modelo de memória distribuída.

2.2.3 Paralelismo de Dados

Este modelo consiste em aplicar-se em paralelo a mesma operação em diferentes partições de um conjunto de dados [1]. Este modelo, pode ser implementado em sistemas de memória partilhada acedendo a diferentes elementos de uma estrutura da memória global ou em sistemas de memória distribuída através de uma abordagem do tipo *MapReduce* em que o conjunto de dados é dividido em *chunks,* pedaços, que são processados de forma distribuída e por fim o resultado é reduzido para apenas uma estrutura com todos os dados processados [17]. A Figura 18 apresenta a divisão de uma matriz em blocos processados paralelamente utilizando paralelismo de dados.

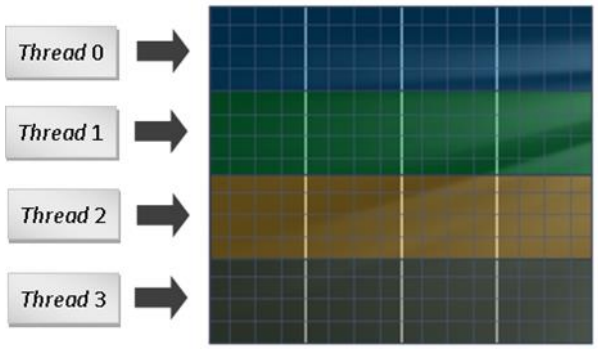


Figura 18: Diagrama do processamento de uma matriz em blocos com paralelismo de dados.

2.2.4 Híbrido

O modelo de programação paralela híbrido combina os dois modelos referidos anteriormente. Este modelo pode ser utilizado para realizar tarefas paralelamente em diferentes nós de um sistema distribuído utilizando o modelo de memória distribuída para a sincronização entre nós e o modelo de memória partilhada para paralelizar as tarefas entre *threads* num nó.

Atualmente este modelo é utilizado frequentemente em sistemas de computação heterogénea em que, apesar de ambos os sistemas utilizarem um modelo de memória partilhada, a interligação entre a memória do sistema e a memória da placa gráfica utiliza um modelo de memória distribuída e a computação ao nível do GPU é realizada utilizando um modelo de paralelismo de dados.

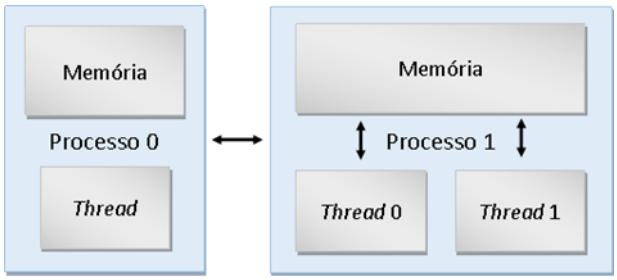


Figura 19: Diagrama representativo do modelo hibrido.

2.3 Computação utilizando o GPU

A computação utilizando o GPU ou GPGPU consiste no uso do GPU com o CPU para acelerar tarefas computacionais. O CPU é constituído por menos núcleos otimizados para processamento sequencial enquanto o GPU é constituído por um maior numero de pequenos núcleos, os *shaders*, desenhados para realizar tarefas em paralelo [18]. Este paradigma consiste

em alocar tarefas de computação paralela intensiva para serem executadas no GPU enquanto o resto do programa é executado pelo CPU tal como exemplificado no diagrama da Figura 20 [19].

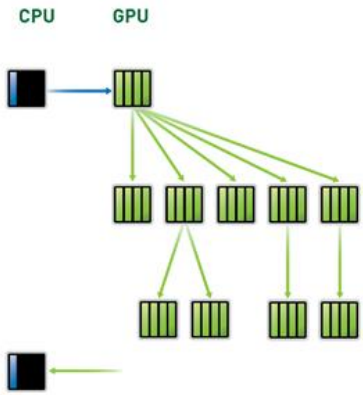


Figura 20: Diagrama exemplificativo da interação CPU/GPU para Computação no GPU. [19]

Em GPGPU, devido às características de processamento do GPU, utiliza-se um modelo de paralelismo de dados com processamento em *streams* em que se aplica um *kernel* aos vários elementos de um *stream*. Um *kernel* corresponde a uma função independente que pode ser aplicada a dados independentes a nível dos núcleos do GPU sem partilhar variáveis entre diferentes núcleos. Um *stream* é um conjunto de valores da memória ao qual é aplicado um mesmo *kernel* em paralelo. Em termos funcionais os GPUs implementam *streams* como vértices ou texturas e *kernels* como conjuntos de funções aritméticas[20]. A Figura 21 representa um modelo de processamento em *streams* em que o GPU recebe conjuntos de *streams* de dados e *kernels* e devolve *streams* com os dados aos quais foram aplicados os kernels [21].

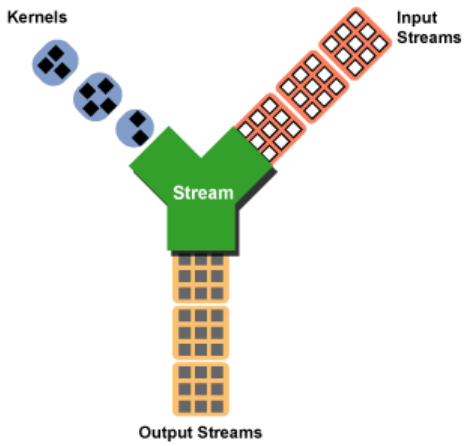


Figura 21: Modelo de processamento em *streams* em que um *stream* de saída é o resultado do processamento de um *kernel* num *stream* de entrada.[21]

2.4 APIs para Computação Paralela

2.4.1 *Threads* Nativos

Um *thread* corresponde a uma sequência de instruções que pode ser executada e agendada independentemente pelo sistema operativo. De uma forma geral um *thread* está contido num processo e utiliza o espaço de memória deste último.

No padrão ANSI C não existe uma especificação para a implementação de *threads* utilizando funções da linguagem estando assim a implementação de *threads* dependente da API do sistema operativo de destino ou do uso de bibliotecas específicas.

Para sistemas operativos compatíveis com a especificação POSIX de *threads*, como Linux, é utilizada a API da biblioteca PThreads que inclui funções para a gestão (criação e junção) de *threads*, implementação de semáforos, variáveis condicionais e sincronização de *threads* utilizando bloqueios de leitura/escrita e barreiras [22].

No Windows a API nativa do sistema operativo permite a criação de *threads* e *fibers* que consistem em unidades de execução que utilizam o contexto do *thread* que as criou e cuja execução é manualmente agendada [23].

2.4.2 OpenMP

Open Multi-Processing (OpenMP) é uma API constituída por diretivas para o compilador e variáveis de ambiente utilizada para desenvolver software utilizando um modelo de memória partilhada. Uma vez que o OpenMP é uma especificação de uma API, a sua implementação está a cargo do compilador podendo ser implementada de diferentes maneiras para diferentes compiladores ou sistemas operativos [24].

De uma forma geral o OpenMP permite aos programadores paralelizarem facilmente blocos de código, sem terem de reescrever completamente o programa adicionando *pragmas,* ou seja diretivas para o compilador, como representado na Figura 22.

```
int main(){

#pragma omp parallel for
     for (int i = 0; i < 10; i++){
         cpu_task(...);
     }

     return 0;
}
```

Figura 22: Utilização de um pragma do OpenMP para paralelizar um ciclo for em C.

2.4.3 Intel Threading Building Blocks

O Intel Threading Building Blocks (Intel TBB) é uma biblioteca de alta performance que permite o desenvolvimento de *software* com processamento em paralelo nas linguagens de programação C e C++. Esta biblioteca permite ao programador abstrair-se de detalhes de implementação de mais baixo nível, como a criação e sincronização de *threads*, tratando as operações como tarefas que podem ser alocadas dinamicamente em "*runtime*" a diferentes unidades de processamento. O *software* desenvolvido mantem um elevado grau de abstração e o suporte para diferentes plataformas e compiladores é assegurado pelo TBB [25].

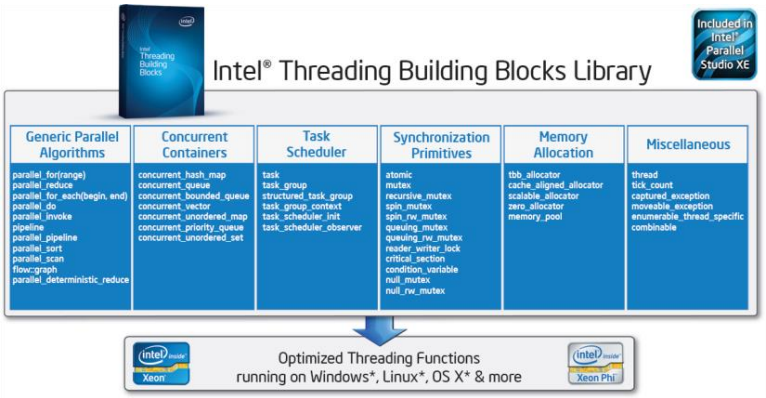


Figura 23: Tabela com uma lista de funções do Intel TBB. [25]

2.4.4 Comparação das APIs de Computação Paralela

A escolha de uma API de computação paralela depende das necessidades do projeto e das funcionalidades da API. Analisando comparativamente as três APIs previamente abordadas neste relatório, cada uma apresenta pontos fortes e pontos fracos que as diferenciam umas das outras. A implementação de *threads* nativos é a mais trabalhosa sendo que todas as funcionalidades têm de ser desenvolvidas sobre a API do sistema operativo e explicitamente invocadas pelo programador. De uma forma geral a API que permite uma implementação mais fácil é o OpenMP que apenas requer a inclusão de *pragmas* a indicar os blocos de código paralelizáveis, no entanto, muitas vezes é necessário otimizar o acesso à memória o que obriga a alterações similares às necessárias para implementar outra abordagem. O Intel TBB permite o desenvolvimento da aplicação de uma forma versátil tendo como principal vantagem uma alta escalabilidade principalmente na definição de tarefas paralelas dentro de tarefas já paralelizadas que é um dos pontos fracos do OpenMP. A Tabela 1 apresenta uma comparação mais detalhada das APIs referidas [26].

Tabela 1: Comparação das funcionalidades de *threads* nativos, OpenMP e Intel TBB. [26]

Funcionalidade	*Threads*	*OpenMP*	*IntelTBB*
Task level parallelism	-	+	+
Scalable nested parallelism	-	-	+
Load Balancing	-	+	+
Compiler independente	+	-	+
OS independente	-	+	+
Minor changes	-	+	-

2.5 APIs para Computação Heterogénea com o GPU

2.5.1 OpenCL

Open Computing Language (OpenCL), atualmente na versão 1.2, é uma *framework* aberta desenvolvida pelo Khronos Group para o desenvolvimento de aplicações para ambientes de computação heterogénea. Esta *framework* consiste numa linguagem baseada em C (ISO C99) e um conjunto de APIs que permitem aos programadores escreverem *kernels* que serão executados em dispositivos compatíveis com o OpenCL como GPUs, coprocessadores e até mesmo CPUs. Uma vez que o OpenCL é compatível com múltiplas plataformas os *kernels* são compilados durante a execução, o compilador apenas compila o código para uma representação intermédia que é futuramente compilada para linguagem máquina de acordo com o dispositivo alvo (compilação *online*) [27].

O OpenCL utiliza um sistema baseado em tarefas, sendo cada *kernel* uma tarefa. Quando o sistema cria uma nova tarefa, ela é colocada numa *queue* e processada logo que os dispositivos estejam disponíveis. Na Figura 24 está representado a conversão de uma função escrita em C para uma *kernel* do OpenCL utilizando paralelismo de dados, isto é, em vez se serem executadas varias iterações do ciclo *for* são criados *threads* de um *kernel* para cada iteração do ciclo que serão executados em paralelo [28].

Traditional loops

```
void
trad_mul(int n,
         const float *a,
         const float *b,
         float *c)
{
  int i;
  for (i=0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

Data Parallel OpenCL

```
kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *c)
{
  int id = get_global_id(0);

  c[id] = a[id] * b[id];

} // execute over "n" work-items
```

Figura 24: Conversão de uma função desenvolvida em C para um Kernel do OpenCL. O ciclo for de 0 a n é transformado em n *threads* do *kernel*. [28].

2.5.2 CUDA

CUDA, atualmente na versão 5.0, é uma plataforma de computação paralela proprietária da NVIDIA destinada ao desenvolvimento de aplicações para ambientes de computação heterogénea utilizando GPUs NVIDIA. O CUDA funciona como uma extensão que pode ser utilizada com as linguagens de programação mais comuns como C/C++ e que permite aos programadores mapearem tarefas que serão executadas na plataforma CUDA, ou seja, no GPU [29]. O CUDA possui ainda uma poderosa ferramenta de *debug*, o NVIDIA Nsight, que permite a visualização gráfica da carga de cada núcleo e alocação dos *threads* em tempo real.

Uma vez que o CPU utiliza a memória principal do sistema e o GPU utiliza a memória independente da placa gráfica a plataforma CUDA utiliza um modelo de memória hibrido. A Figura 25 representa o fluxo de processamento na plataforma CUDA, utilizando o modelo anteriormente referido. Numa primeira fase os dados e o *kernel* são enviados para o GPU. Depois de concluído o processamento os resultados são copiados de volta para a memória principal.

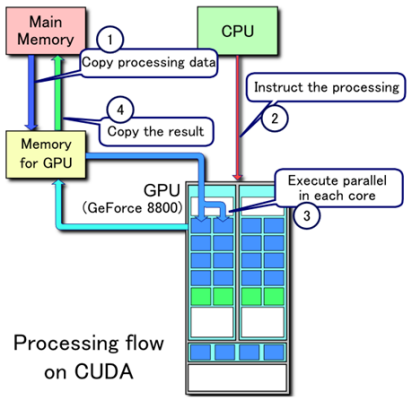


Figura 25: Fluxo de processamento da plataforma CUDA. [30]

Na Figura 26 está representado a conversão de uma função escrita em C para uma *kernel* CUDA com sincronização de dados explícita utilizando paralelismo de dados, criando 4096 blocos de 256 *threads* na plataforma CUDA, cada *thread* equivalente a uma iteração do ciclo *for*.

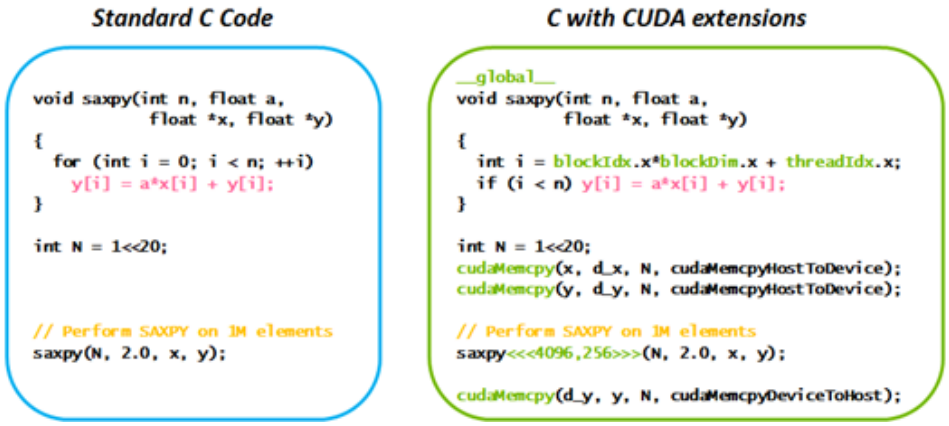


Figura 26: Conversão de uma função desenvolvida em C para um *kernel* CUDA. O ciclo for de 0 a n é transformado em *threads* do *kernel* a serem executadas nos núcleos CUDA. [29]

2.5.3 C++ AMP

O C++ AMP é uma especificação livre para uma biblioteca de C++, semelhante a uma biblioteca do STL, desenhada para o desenvolvimento de algoritmos que utilizem paralelismo de dados diretamente em C++, sem a necessidade de recorrer a uma plataforma intermédia como o CUDA ou OpenCL. A API desta biblioteca especifica o modo de invocação de funções e inclui um conjunto de classes que embrulham as estruturas de dados do CPU copiando-as de forma transparente ao programador entre o *Host* e o GPU. As funções de C++ que podem ser executadas pelo GPU utilizam apenas um subconjunto das funcionalidades da linguagem que é suportada pela maioria dos GPUs. A diretiva *restrict(amp)* instrui o compilador que uma função utiliza apenas este subconjunto e é compatível com o GPU. O C++ AMP é ainda compatível com *fallback* das operações para um ou mais núcleos do CPU utilizando funções de SSE. A Figura 27 inclui o código de uma função de soma de *arrays* utilizando C++ AMP [31].

```
// C++ AMP Add Arrays
void AddArrays(int n, int m, int * pA, int * pB, int * pSum) {
    concurrency::array_view<int,2> a(n, m, pA), b(n, m, pB), sum(n, m, pSum);
    concurrency::parallel_for_each(sum.extent,
          [=](concurrency::index<2> i) restrict(amp) {
             sum[i] = a[i] + b[i];
    });
}
```

Figura 27: Exemplo da soma de dois *arrays* utilizando C++ AMP. [31]

A implementação de maior destaque do C++ AMP é a implementação da Microsoft incluída no Visual Studio 2012 utilizando o DirectX 11 para fazer a ponte entre o CPU e o GPU. O *software* implementado em AMP é compatível com o Windows 7 ou superior e qualquer GPU que tenha *drivers* com suporte para o DirectX 11. O Visual Studio inclui um extenso leque de ferramentas de *debug* e *profiling* compatíveis com C++ AMP.

2.5.4 OpenACC

OpenACC é uma norma desenvolvida pela PGI, Cray, CAPS e NVIDIA desenhada para permitir o desenvolvimento fácil e rápido de aplicações que utilizem as capacidades de computação paralela do CPU e GPU através de diretivas para o compilador que indiquem os blocos de código a paralelizar de uma forma semelhante ao OpenMP. A Figura 28 apresenta um diagrama representativo da paralelização de uma função sequencial utilizando anotações do OpenACC [32].

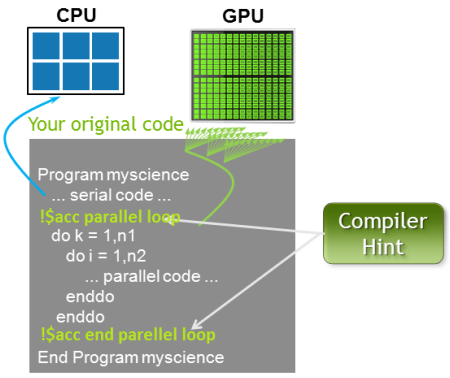


Figura 28: Paralelização de uma função sequencial utilizando OpenACC. [32]

Apesar do OpenACC geralmente automatizar a organização dos dados e das *threads* no dipositivo o programador pode especificar manualmente estas definições através do uso da API do OpenACC [33].

2.5.5 Comparação das APIs de Computação Heterogénea

Comparando as plataformas de computação heterogénea com o GPU, o OpenACC é a plataforma que requere um menor número de alterações ao código para a implementação de blocos de código utilizando paralelismo no GPU. No entanto, o OpenACC ainda é uma norma recente, sendo suportada por um reduzido número de compiladores. O CUDA é a plataforma na qual o *debug* é mais fácil devido às ferramentas de *debug* e *profiling* desenvolvidas pela NVIDIA. Relativamente ao *fallback* para o CPU, ou seja a capacidade de executar código no CPU caso o GPU não seja compatível, esta funcionalidade está disponível no C++ AMP,

OpenCL e OpenACC não existindo no CUDA. O C++ AMP é a API mais versátil em termos de *hardware* suportando todo os GPUs compatíveis com DirectX 11. A Tabela 2 apresenta uma comparação mais detalhada das funcionalidades existentes nas respetivas plataformas.

Tabela 2: Comparação das funcionalidades das plataformas CUDA, OpenCL, C++ AMP e OpenACC.

Funcionalidade	*CUDA*	*OpenCL*	*C++ AMP*	*OpenACC*
Fall-Back to CPU	-	+	+	+
Open Platform	-	+	+	+
Debugging & profiling tools	+	-	+	-
Minor code changes	-	-	-	+

2.6 Conclusões

O estudo realizado permite-nos concluir que os CPUs e os GPUs têm diferentes características que os tornam mais propícios para diferentes tipos de tarefas de processamento.

Os CPUs são processadores extremamente versáteis que minimizam os tempos de acesso à memória RAM do sistema utilizando diferentes níveis de *cache* de alta velocidade. Esta estrutura garante que os CPUs tenham uma latência mínima de acesso à memória a não ser quando ocorrem *cache misses*. O *simultaneous multi-threading* é uma tecnologia que permite a um CPU trocar rapidamente de contexto entre dois *threads* quando ocorre um *cache miss* rentabilizando ao máximo todos os ciclos de processamento.

Por outro lado, os GPUs têm uma estrutura altamente paralela em que múltiplos núcleos processam sincronamente a mesma instrução e escondendo os *overhead* de acesso à memória através da troca de contexto para outros threads até que os valores de memória estejam disponíveis. Esta estrutura com um menor número de unidades de controlo e sem os múltiplos níveis de *cache* permite aos GPUs trocarem a flexibilidade de processamento do CPU por um número muito superior de unidades de processamento.

Existem diversas APIs que nos permitem tirar proveito das capacidades de processamento em paralelo disponíveis tanto nos CPUs como nos GPUs. O OpenMP é uma API de computação paralela com o CPU extremamente flexível que permite o desenvolvimento de paralelismo de dados e paralelismo funcional com o menor impacto no código sequencial já desenvolvido e desempenhos competitivos. A nível dos GPUs o CUDA sobressai por oferecer o melhor desempenho nos GPUs NVIDIA e por incluir as melhores ferramentas de desenvolvimento e a documentação mais completa e detalhada. As APIs utilizadas neste trabalho são o OpenMP e o CUDA por oferecem o melhor compromisso entre flexibilidade e desempenho.

Capitulo 3

Computação Heterogénea na Codificação de Vídeo

Com o aumento exponencial da capacidade de processamento dos GPUs e o desenvolvimento de soluções cada vez mais flexíveis de GPGPU, têm sido estudadas e desenvolvidas soluções de vídeo utilizando computação heterogénea. Atualmente existem vários trabalhos sobre as aplicações da computação heterogénea no processamento e codificação de vídeo e inclusivamente algumas das bibliotecas e SDKs utilizam as capacidades dos GPUs em algumas das suas tarefas.

Neste capítulo são abordados os seguintes pontos:

- Analisam-se as principais considerações na divisão e paralelização de um algoritmo de vídeo;
- Analisam-se dois trabalhos científicos sobre a paralelização do algoritmo de codificação do MPEG-2;
- Analisam-se algumas das bibliotecas de codificação com aceleração do GPU disponíveis com um especial foco nas bibliotecas com aceleração de MPEG-2 e de H.264/AVC que é um dos formatos com maior desenvolvimento na área da computação heterogénea.

3.1 Codificação de Vídeo com o GPU

Os algoritmos de codificação de vídeo podem ser paralelizados utilizando diferentes esquemas dividindo o trabalho entre as diferentes unidades de processamento em função das

dependências existentes no processo de codificação do *stream* [34]. As divisões mais comuns seguem quatro esquemas:

- *Group of Pictures* (GOP) – cada GOP é processado independentemente por uma unidade de processamento, simples mas de elevada latência [34];
- *Frame* – cada *frame* é codificada independentemente sendo pouco comum devido às dependências entre *frames* de um mesmo GOP [34];
- *Slices* – a arquitetura mais comum, a *frame* é dividida em *slices* que são processados paralelamente [34];
- Macroblocos/Blocos – cada macrobloco/bloco é processado paralelamente, eficaz caso o processo de codificação das estruturas seja independente [34].

Atendendo às propriedades computacionais de um GPU apenas algumas tarefas se adequam ao modelo de processamento deste dispositivo. Para um melhor aproveitamento deste modelo os algoritmos devem ser desenhados utilizando um número massivo de *threads* aproveitando ao máximo o elevado número de núcleos de processamento [35]. Por outro lado tarefas com elevado número de instruções de controlo como *if*, *switch*, *do*, *while* e *for* podem ter performances consideravelmente inferiores no GPU uma vez que diferentes *threads* podem seguir diferentes caminhos de processamento obrigando o GPU a serializar as instruções [35]. As tarefas devem ser implementadas minimizando o número de acessos à memória do GPU realizando o maior número de operações aritméticas possíveis por cada acesso à memória.

Para atingir os melhores desempenhos o processamento deve ser combinado entre o CPU e o GPU procurando dividir o trabalho de forma ótima entre as duas unidades de processamento atendendo a três condições fundamentais:

- Sobrepor as tarefas de modo a possibilitar o processamento de tarefas pelo CPU e GPU simultaneamente;
- Minimizar as transferências de dados entre a memória RAM e a memória do GPU;
- Identificar quais os módulos mais suscetíveis de serem implementados no GPU utilizando uma abordagem de paralelismo de dados e quais os módulos que devido às dependências de dados ou elevada quantidade de instruções de controlo devem ser processados sequencialmente no CPU.

Posto isto, podemos concluir que as tarefas mais propicias a serem processadas no GPU são tarefas de aritmética com o menor número de saltos condicionais possíveis e que sejam

aplicadas independentemente a diferentes partes do vídeo podendo cada parte ser processada num *thread* independente.

3.2 Implementações do Algoritmo de Codificação do MPEG-2

3.2.1 Implementação em Paralelo do MPEG-2

No artigo [36] é descrita uma abordagem à paralelização do algoritmo de codificação do MPEG-2. Neste artigo, o algoritmo é dividido em duas partes, como exemplificado na Figura 29, uma parte paralelizável que engloba as etapas de DCT, Quantização e predição de movimento e uma parte não paralelizável correspondente à etapa de VLC. A solução implementada foi desenvolvida utilizando um sistema de memória distribuída dividindo o trabalho entre 4 processadores iguais utilizando MPI.

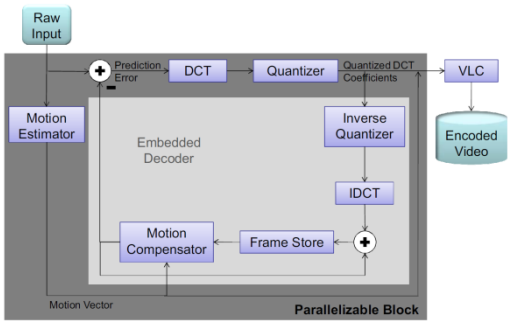


Figura 29: Divisão do algoritmo de codificação do MPEG-2. [36]

3.2.2 Implementação do *codec* MPEG no coprocessador Tilera 64

Neste artigo foi analisada uma implementação utilizando computação paralela do *codec* MPEG num processador de memória distribuída Tilera de 64 núcleos. O *codec* foi implementado utilizando processamento em rede em que cada núcleo processa em paralelo um conjunto de macroblocos de uma *frame*, ou seja, um *slice* [37]. O algoritmo faz a distribuição homogénea dos *slices* sendo que uma possível melhoria futura, apontada no artigo, é a distribuição dos *slices* tendo em conta a distância dos núcleos ao centro da rede uma vez que o algoritmo de alocação implementado considera um lado da rede como ponto de entrada como exemplificado no diagrama da Figura 30.

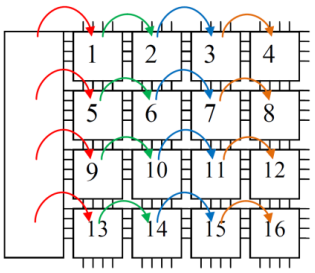


Figura 30: Diagrama da rede de núcleos do processador Tilera e alocação dos *slices*. [37]

Na Tabela 3 estão registados os valores do *speedup* para 4, 6 e 16 núcleos calculados em relação ao tempo de execução teórico desta implementação utilizando apenas um núcleo. Os resultados obtidos demonstraram que o algoritmo apresenta valores de *speedup* teóricos lineares para um maior número de núcleos.

Tabela 3: Valores de *speedup* obtidos para a solução implementada em função do número de *tiles*.

Tiles	*Speedup (t1/tb)*
4	3,986
16	15,989
64	63,992

3.3 Bibliotecas de codificação

3.3.1 Aceleração com o GPU do *codec* MPEG-2

O Intel Media SDK 2013 consiste num conjunto de bibliotecas produzidas pela Intel para o desenvolvimento de aplicações que realizem a codificação e descodificação de vídeo [38]. O *codec* MPEG desta biblioteca inclui aceleração de *hardware* para a codificação de MPEG-2 e H.264/AVC, no entanto esta aceleração, apelidada de Intel Quick Sync Video, é implementada ao nível dos controladores dos GPU Intel não sendo compatível com GPUs de outros fabricantes [38]. O Intel Quick Sync Video utiliza cerca de 10% da capacidade de processamento do GPU para aliviar a carga de um núcleo do CPU sendo por isso possível explorar a restante capacidade de processamento do GPU utilizando OpenCL [39]. A Figura 31 representa um diagrama da interação do Intel Media SDK com as *drivers* dos GPUs Intel previamente descrita.

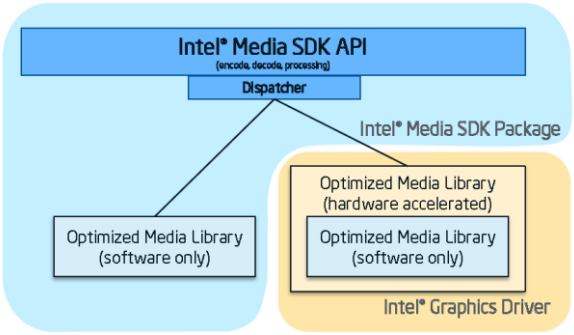


Figura 31: Arquitetura do Intel Media SDK 2013. [38]

3.3.2 Aceleração com o GPU de outros *codecs* de vídeo

Existem várias soluções de aceleração de GPGPU para a codificação de outros formatos de vídeo, em especial o AVC/H.264, tirando partido da possibilidade de distribuir o trabalho entre o CPU e o GPU.

A biblioteca NVCUVENC da NVIDIA é uma biblioteca hibrida CPU/GPU que utiliza uma plataforma CUDA para acelerar a codificação de *streams* de vídeo em AVC/H.264. Esta biblioteca utiliza o GPU para processar parte das tarefas das etapas mais paralelizáveis do processo de codificação limitando as etapas sequenciais à execução no CPU. A NVIDIA disponibiliza ainda a biblioteca NVENC que utiliza os módulos de *hardware* dedicados à codificação de H.264 incluídos nos GPUs de arquitetura *Kepler* [40].

A AMD disponibiliza a biblioteca Open Video Encode, desenvolvida utilizando OpenCL e o AMD Accelerated Parallel Processing (APP) SDK, que contém uma ferramenta de codificação para plataformas AMD com suporte para o processamento heterogéneo de H.264 fazendo o *offloading* de parte do processo para o GPU [41].

A MainConcept, líder mundial da produção de *codecs* de áudio e vídeo para a indústria de vídeo, oferece *codecs* de H.264/AVC utilizando CUDA, OpenCL e o Intel Quick Sync, disponibilizando um *wrapper* que permite a utilização do *codec* mais adequado para cada máquina em *runtime* como exemplificado na Figura 32 [42].

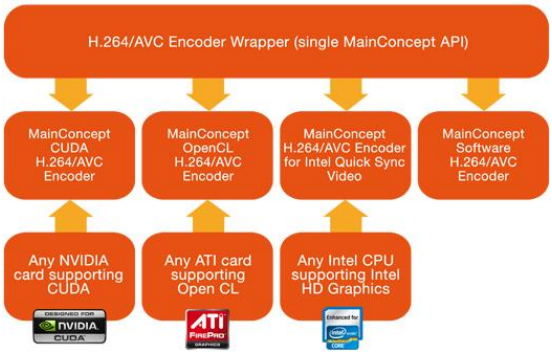


Figura 32: Diagrama do funcionamento do *wrapper* do *codec* H.264 da MainConcept.[42]

3.4 Conclusões

Neste capítulo descreveram-se os principais métodos de divisão de trabalho na paralelização de um algoritmo de codificação de vídeo, analisaram-se dois trabalhos científicos sobre a paralelização do MPEG-2 e apresentaram-se algumas bibliotecas de codificação de vídeo com o GPU. Apesar do H.264/AVC ser um formato mais recente e ter um maior potencial, o formato em estudo é o MPEG-2 IMX para o qual não existem soluções adequadas de computação heterogénea embora exista a sua necessidade. Nos próximos capítulos vamos analisar o algoritmo de codificação do MPEG-2 IMX e explorar a aplicação de modelos de paralelismo neste algoritmo.

Capítulo 4

Algoritmo de Codificação: MPEG-2 IMX

O estudo e compreensão do formato e algoritmo de codificação MPEG-2 IMX é essencial para identificar etapas propícias à aplicação de paralelismo e definir a abordagem e modelo de paralelismo a implementar. Esta análise foi dividida nas seguintes componentes:

- Análise ao formato e especificação da codificação;
- Análise à implementação do FFmpeg;
- Estudo das dependências de dados nas diferentes etapas de codificação;
- *Profiling* das etapas de codificação.

4.1 Análise do MPEG-2 IMX

O MPEG-2 é um formato *standard* de codificação de vídeo com perdas, desenvolvido pela *Moving Picture Experts Group* (MPEG), desenhado para codificar vídeo progressivo ou interlaçado de alta resolução sendo compatível com os formatos de televisão mais comuns como PAL, NTSC e HDTV [43].

No MPEG-2 a cor dos pixéis, enviados para os monitores no formato RGB (*Red*, *Green* and *Blue*), é codificada sob a forma de componentes YUV com luminância (Y) e crominância (UV) também representado por YCbCr. A largura de banda da crominância pode ser reduzida comparativamente à luminância, sem originar uma perda significativa de qualidade. São

utilizados os fatores 4:2:2 e 4:2:0 para representar os fatores de crominância em relação à luminância [44].

O MPEG-2 IMX ou SMPTE D10 é uma especificação de um formato de vídeo profissional composto por vídeo MPEG-2 4:2:2 com todas as *frames* independentes e áudio AES3 [45].

4.1.1 Estrutura do *Stream*

O MPEG-2 codifica as *frames* em grupos chamados de GOP (*Group of Pictures*), tal como representado na Figura 33, que esquematiza a hierarquia de codificação de um *stream* de vídeo codificado em MPEG-2. O *stream* é organizado em sequências de GOPs, sendo que cada GOP contém várias imagens. As imagens são consequentemente subdivididas em *slices*, macroblocos e blocos [44].

O MPEG especifica que pode existir predição de movimentos *inter-frame* ou seja, uma *frame* pode ser codificada pelas semelhanças em relação a uma outra frame reduzindo consideravelmente o seu tamanho [46]. As *frames* de um GOP podem ser subdivididas em três tipos:

- I – *frame* independente, a sua codificação não requer informação das restantes *frames*;
- P – *frame* dependente, a sua codificação requer informação da *frame* I ou P anterior;
- B – *frame* bidireccionalmente dependente, necessita informação das *frames* I ou P anterior e posterior.

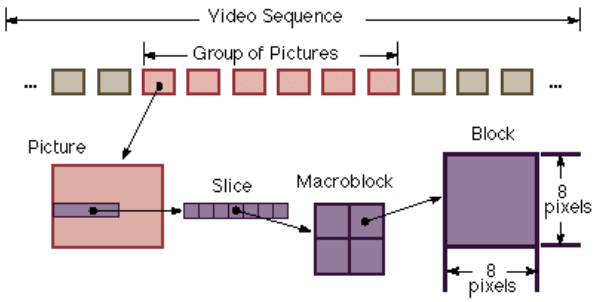


Figura 33: Hierarquia de um *stream* MPEG [44].

A Figura 34 representa as dependências de dados das *frames* do tipo I, P e B anteriormente referidas. Como exemplificado na Figura 34 as *frames* do tipo I apenas requerem informação *stream* de entrada referentes à própria *frame*, as *frames* do tipo P requerem informação da última *frame* codificada do tipo I ou P e as *frames* do tipo B requerem informações da última e da próxima frame do tipo I ou P.

O MPEG IMX especifica a codificação de vídeo com uma *bitrate* fixa e com *GOPs* com apenas uma *frame* do tipo I [45].

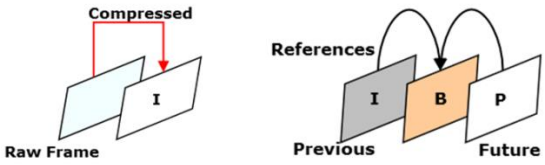


Figura 34: Diagrama de dependências das *frames* do tipo I, P e B. [47]

4.1.2 Etapas de Codificação

O algoritmo de codificação do MPEG-2 IMX representado no diagrama da Figura 35 é constituído por 5 etapas de processamento independentes realizadas de forma sequencial. Este algoritmo baseia-se na redundância espacial dos blocos de uma *frame* para comprimir a *frame* utilizando um conjunto de técnicas de codificação com e sem perdas seguidamente descritas com maior detalhe.

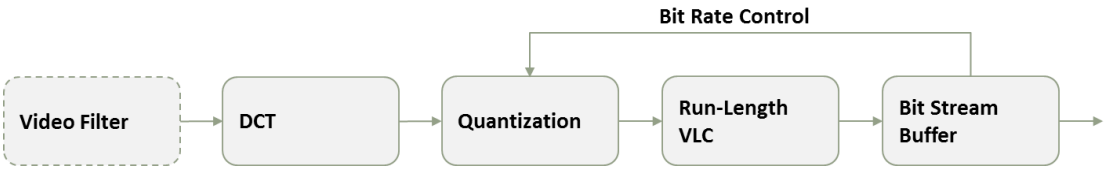


Figura 35: Diagrama de blocos do algoritmo de codificação do MPEG-2 IMX. [47]

4.1.2.1 Filtro de Vídeo

O filtro de vídeo é uma técnica de compressão com perdas utilizada para comprimir redundâncias espaciais a nível dos macroblocos de uma *frame* que consiste em realizar operações de conversão de cor explorando o facto da visão humana ser menos sensível a variações de cor (crominância) que a variações de luminosidade (luminância). Os dois processos geralmente utilizados são a conversão para YCbCR e a subsequente subamostragem das crominâncias [47].

4.1.2.1.1 Conversão para YCbCR

Este processo consiste em converter as componentes RGB de uma imagem para componentes YCbCR utilizando as funções da Figura 36, reduzindo a redundância entre componentes que existe no espaço de cor RGB e consequentemente reduzindo o tamanho da *frame*. A Figura 37 apresenta a decomposição de uma *frame* nas componentes de cor RGB e a Figura 38 representa a decomposição nas componentes de cor YCbCR. Como se pode observar nas figuras as componentes YCbCR têm menor redundância de cores, principalmente para cores escuras em tons de preto e cinzento [47].

$$\begin{aligned} Y &= 0.299R + 0.587G + 0.114B \\ Cb &= (B - Y) * 0.565 \\ Cr &= (R - Y) * 0.713 \end{aligned} \quad \longrightarrow \quad \begin{aligned} R &= Y + 1.403Cr \\ G &= Y - 0.344Cb - 0.714Cr \\ B &= Y + 1.770Cb \end{aligned}$$

Figura 36: Equações de conversão entre RGB e YCbCr.

Figura 37: Componentes RGB de uma *frame*. [47]

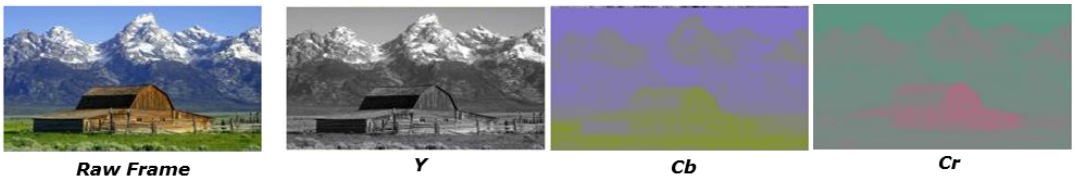


Figura 38: Componentes YCbCr de uma *frame*. [47]

4.1.2.1.2 Subamostragem das crominâncias

A subamostragem da crominância consiste em reduzir o número de amostras da crominância, utilizando uma amostra de cada crominância para um bloco de pixéis ao invés de uma amostra de crominância por *pixel* como exemplificado na Figura 39[48]. Na codificação em MPEG-2, na qual se utilizam 8 bits para a cor, este processo permite reduzir o número de bits utilizados para codificar um bloco de 4 *pixels* dos 12 bytes (3 por *pixel*) de uma amostragem 4:4:4 para 6 bytes em 4:2:0.

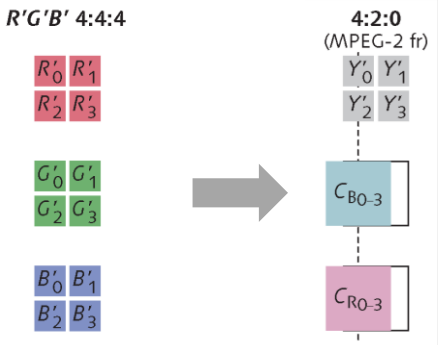


Figura 39: Diferença de codificação entre um espaço de cor RGB e um espaço de cor YCbCr 4:2:0. [48]

4.1.2.2 Transformada Discreta de Cosseno (DCT)

Uma transformada discreta de cosseno (DCT) consiste numa função que representa um conjunto finito de pontos sob a forma de um somatório de funções cosseno com diferentes frequências. A DCT permite converter variações espaciais dentro do macrobloco em variações de frequências sem alterar a informação representada. Uma DCT de duas dimensões pode ser calculada com o algoritmo da Figura 40 [47].

$$\boldsymbol{F(u,v)} = \frac{2}{N} C(u)C(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y) cos \frac{(2x+1)u\pi}{2N} cos \frac{(2y+1)v\pi}{2N}$$

$u \quad Frequencia\ espacial\ horizontal\ para\ os\ inteiros\ 0 \leq u < N$

$v \quad Frequencia\ espacial\ horizontal\ para\ os\ inteiros\ 0 \leq v < N$

$\boldsymbol{C(u)} = \boldsymbol{C(v)} = \begin{cases} \frac{1}{\sqrt{2}}, se\ u,v = 0 \\ 1, se\ u,v \neq 0 \end{cases}$ função de normalização

$\boldsymbol{f(x,y)}\ Valor\ do\ pixel\ nas\ coordenadas\ x,y$

$\boldsymbol{F(u,v)}\ Coeficiente\ de\ DCT\ nas\ coordenadas\ u,v$

Figura 40: Função do cálculo de uma DCT de duas dimensões. [47]

Em termos práticos, a DCT é utilizada para converter a representação de componentes YCbCr de um bloco numa representação baseada em frequências. De notar que a DCT não reduz o número de bits utilizados para representar um bloco, pelo contrário os coeficientes são representados utilizando 11bits ao invés dos 8bits utilizados na representação anterior [49]. A Figura 41 apresenta um exemplo do processo de DCT.

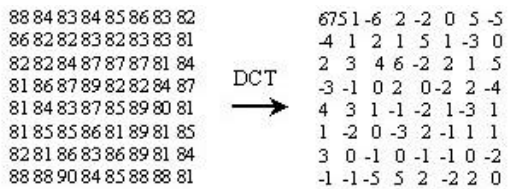


Figura 41: Conversão de uma matriz YCbCr de um bloco para uma matriz de coeficientes DCT. [50]

4.1.2.3 Quantização

Quantização é um mecanismo utilizado para eliminar as baixas frequências da matriz reduzindo os valores da matriz o que permite codificar a utilizando um menor número de bits.

Este processo consiste em dividir a matriz gerada na DCT por uma matriz padrão com valores constantes utilizando a fórmula da Figura 42. A qualidade do processo de quantização vária de acordo com a matriz de quantização utilizada e do tamanho dos blocos [47].

$$B_{j,k} = round\left(\frac{G_{j,k}}{Q_{j,k}}\right) \; em \; que \; j = 0, 1, 2, \ldots, N_1 - 1 \; \wedge \; K = 0, 1, 2, \ldots, N_2 - 1$$

Figura 42: Função de quantização. [47]

Uma vez que nesta operação ocorre a perda de informação relativa às luminâncias e crominâncias, a matriz resultante da quantização sofre uma degradação da qualidade de imagem [47]. Esta é a única etapa da codificação em MPEG-2 em que ocorrem perdas. A Figura 43 apresenta um exemplo da aplicação da quantização à matriz de DCT da Figura 41.

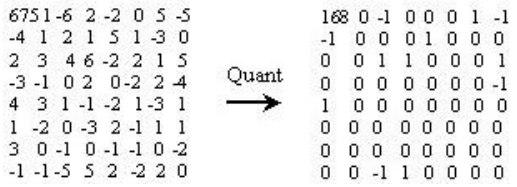


Figura 43: Quantização de uma matriz de coeficientes DCT de um bloco. [50]

4.1.2.4 *Run-Length Variable Length Coding* (VLC)

Depois de realizada a DCT e quantização realiza-se a compressão sem perdas da matriz quantizada segundo três etapas:

1. Zig-Zag – a matriz de quantização de cada bloco é reordenada em Zig-Zag para juntar o maior número possível de zeros consecutivos como demonstrado na Figura 44 [47];

2. *Run Length Encoding (RLE)* – a matriz reordenada de cada bloco é processada utilizando um algoritmo de RLE. É gerada uma representação intermédia com a matriz codificada condensando varias ocorrências consecutivas de um elemento sob a forma da posição do primeiro elemento com um determinado valor e esse valor como exemplificado na Figura 45 [47]. O coeficiente da posição (0,0) da matriz de quantização é chamado de coeficiente DC e é ignorado nesta etapa [50];

3. *Huffman VLC* – o algoritmo recebe as tabelas de RLE codificando a matriz utilizando o algoritmo de Huffman.

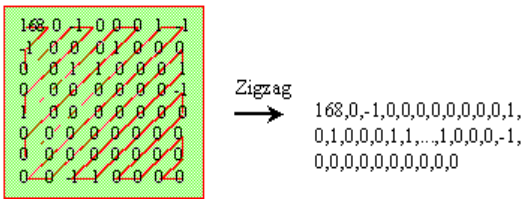


Figura 44: Zig-Zag de uma matriz de quantização de um bloco. [50]

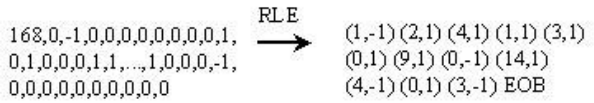


Figura 45: *Run-Length Encoding* de uma matriz de quantização reordenada em Zig-Zag. [50]

4.1.2.5 Escrita para o *Buffer* e Controlo da *Bit Rate*

A codificação do MPEG é realizada utilizando uma taxa de bits constante, ou seja, todas as *frames* têm de ter o mesmo tamanho. A qualidade da matriz gerada durante a quantização pode afetar o tamanho resultante da *frame* codificada. O sistema de Controlo de *Bit Rate* previne dois tipos de situação:

- *Underflow* – preenche o espaço restante do *buffer* que não foi ocupado pela *frame* com zeros;
- *Overflow* – repete o processo de quantização com menor qualidade ou salta a *frame* caso não consiga obter uma *frame* com dimensões compatíveis num determinado número de iterações. Por vezes o sistema é pessimista para evitar repetir o processo.

4.2 FFmpeg

A implementação do *codec* MPEG-2 do FFmpeg consiste num algoritmo sequencial implementado em C99. Esta biblioteca foi desenvolvida com uma estrutura modular para que os *codecs* possam partilhar funções e dependências.

4.2.1 Algoritmo de Codificação

O processo de codificação MPEG-2 do FFmpeg consiste na aplicação sequencial, para cada macrobloco de uma *frame,* de um conjunto de algoritmos que realizam o cálculo da DCT, Quantização da matriz de frequências, VLC e escrita dos macroblocos codificados para o *bit buffer* tal como esquematizado no diagrama da Figura 46.

Algoritmo de Codificação: MPEG-2 IMX

O Anexo A contém um diagrama mais detalhado das principais chamadas de funções do algoritmo de codificação do FFmpeg para a codificação de uma *frame* e da divisão do algoritmo nas respetivas etapas de codificação.

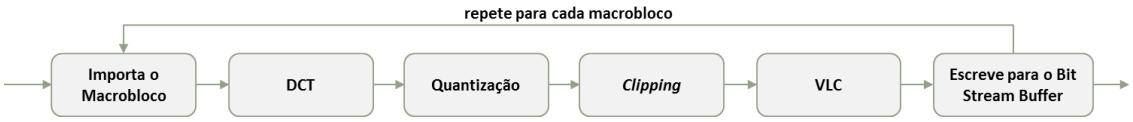


Figura 46: Diagrama da implementação do *codec* MPEG-2 IMX do FFmpeg.

4.2.1.1 Estruturas de dados

O FFmpeg utiliza estruturas de dados especificamente desenvolvidas para armazenar os dados referentes a uma *frame* ou *codec*. Estas estruturas são implementadas em C por *structs* com alinhamento de 16 *bytes* por forma a garantir a compatibilidade com instruções SIMD como MMX e SSE que possuem requisitos especiais de alinhamento dos elementos na memória e oferecendo inclusivamente em algumas arquiteturas ganhos de desempenho no acesso à memória por instruções SISD. Os requisitos de alinhamento são garantidos pela função *av_malloc* do FFmpeg que utiliza diretivas específicas do compilador e da plataforma alvo para assegurar o correto alinhamento dos bytes alocados.

A interface do *codec* é definida numa instância global da estrutura *AVCodec* que contém as configurações referentes aos formatos de cor, modos de codificação e outras capacidades suportadas pelo *codec* assim como os apontadores para as funções de inicialização, finalização e codificação do mesmo. Esta estrutura é utilizada para associar e referenciar o *codec* nas funções da API da *libavcodec*.

A *AVFrame* é o tipo de estrutura padrão utilizado no FFmpeg para armazenar e transportar os dados de uma *frame* proveniente do descodificador para os diferentes módulos. Esta estrutura contém os dados de cada componente de cor do vídeo ou canal de áudio isolados em diferentes linhas de um *array* bidimensional. No caso do MPEG-2, o *codec* recebe uma *AVFrame* contendo a imagem da *frame* subamostrada para o espaço de cor YUV 4:2:2 ou 4:2:0, com a luminância e crominâncias isoladas e a proporção entre elas especificada num elemento da estrutura.

O contexto do *codec* é armazenado numa estrutura do tipo *MpegEncContext* que é alocada apenas uma vez na inicialização do *codec* e desalocada quando o codec é fechado. Este tipo de estrutura é utilizado por todos os *codecs* da família MPEG para garantir a interoperabilidade das funções e contém todos os dados utilizados nas operações do codec como os parâmetros de qualidade da *frame*, o *buffer* de inteiros de 16 bits utilizado para a matriz de frequências da DCT e Quantização de um macrobloco e o buffer de bits utilizado para a escrita da *frame* codificada ao longo do processo de codificação.

4.2.1.2 Inicialização da frame

No início da codificação de cada *frame* o *codec* começa por detetar se tem posse exclusiva da estrutura *AVFrame* recebida e em caso negativo realiza uma cópia profunda desta estrutura por forma a prevenir eventuais colisões. Após garantir a posse exclusiva da estrutura, o *codec* estima e define a escala de quantização, nível de qualidade da quantização, e inicializa o *bit buffer* do contexto com o cabeçalho da *frame* e codifica a matriz de quantização selecionada.

4.2.1.3 Importar o macrobloco para o contexto

Nesta etapa o *codec* copia os dados de um macrobloco para um *buffer* do contexto utilizando uma função desenvolvida em *assembly* que tira partido das capacidades de processamento vetorial, SIMD, da plataforma de destino. No caso em estudo para a plataforma x86, o FFmpeg utiliza a função *ff_get_pixels_sse2* que faz recurso das instruções SSE para copiar, 128 bits de cada vez, os blocos da estrutura *AVFrame* para o *buffer* do contexto e a função *emulated_edge_mc_sse* para simular as margens para macroblocos das extremidades que devido às dimensões do vídeo possam ter menor número de blocos. A nível da dependência de dados, todos os macroblocos são independentes nesta etapa, ocorrendo no entanto colisão a nível dos *buffers* do contexto.

4.2.1.4 DCT

O FFmpeg utiliza um algoritmo de alta precisão implementado pelo Independent JPEG Group para o cálculo da DCT de bidimensional em blocos de oito por oito pixéis baseado no algoritmo "Pratical Fast 1-D DCT" descrito por C. Loeffler *et all* em 1989. Este algoritmo consiste em calcular a DCT de uma dimensão para cada linha da matriz e de seguida para cada coluna obtendo o mesmo resultado de uma DCT bidimensional direta com uma implementação mais simples como representado na Figura 47 [51]. O algoritmo implementado utiliza inteiros de 16bits para a representação da matriz e aritmética, não necessitando de recorrer à unidade de processamento de vírgula flutuante (FPU), evitando assim a troca de modo do FPU entre processamento vetorial e vírgula flutuante que é uma operação lenta em alguns processadores.

Nesta etapa apenas existe dependência de dados dentro de um bloco.

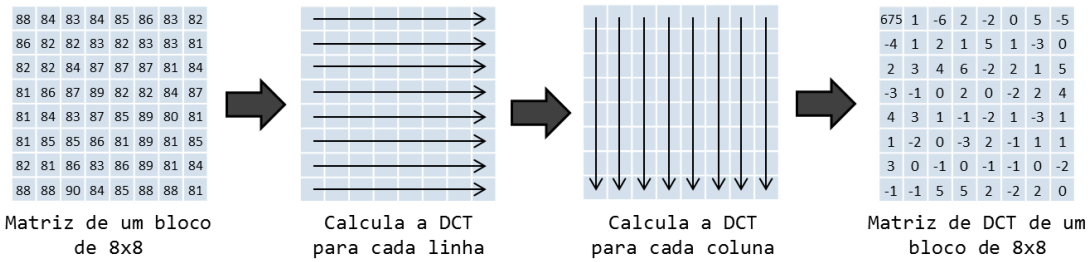


Figura 47: Diagrama do algoritmo de DCT bidimensional baseado numa passagem do algoritmo unidimensional em cada dimensão.

4.2.1.5 Quantização

O algoritmo de quantização implementado utiliza aritmética de inteiros, substituindo a divisão de vírgula flutuante e consequente arredondamento pela multiplicação pelos valores da matriz seguido de um *shift* de *n* bits à direita que equivale a uma divisão por 2^n. Esta implementação, tal como no caso da DCT, evita a utilização do FPU que poderia limitar a performance do algoritmo em alguns sistemas.

Este algoritmo é constituído por dois ciclos, o primeiro começa pelo fim da matriz e quantiza a matriz até encontrar um elemento que não seja quantizado a zero e regista a sua posição e o segundo começa no início da matriz e quantiza todos os elementos até ao último zero calculado pelo primeiro ciclo. Esta etapa tem dependência de dados a nível do bloco.

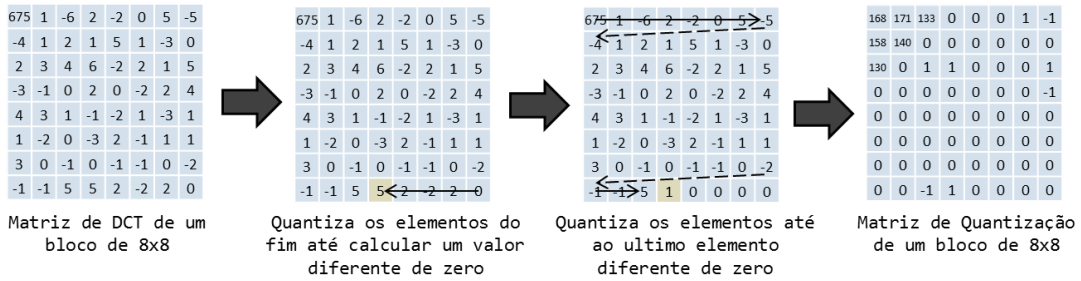


Figura 48: Diagrama do algoritmo de quantização com pesquisa do último elemento quantizado num valor diferente de zero.

4.2.1.6 Controlo de *Bit Rate*

Durante o processo de quantização o *codec* estima as dimensões finais do macrobloco medindo as variações de frequências entre os elementos da tabela e o número de zeros da matriz quantizada. Caso a estimativa indique a possibilidade da ocorrência de um *overflow* durante a codificação, devido às dimensões do macrobloco excederem o tamanho máximo por macrobloco, é realizado o *clipping* da matriz de frequências, reduzindo os valores extremos e prevenindo o *overflow* sem necessitar de repetir todo o processo de DCT e Quantização utilizando um valor de qualidade inferior.

O *clipping* consiste em reduzir a amplitude da matriz reduzindo frequências extremamente grandes para o valor de um coeficiente máximo previamente definido e aumentando frequências extremamente pequenas para o valor de um coeficiente mínimo, como exemplificado na Figura 49, reduzindo assim o tamanho do macrobloco codificado com VLC devido à codificação de todos os valores extremos como um único valor.

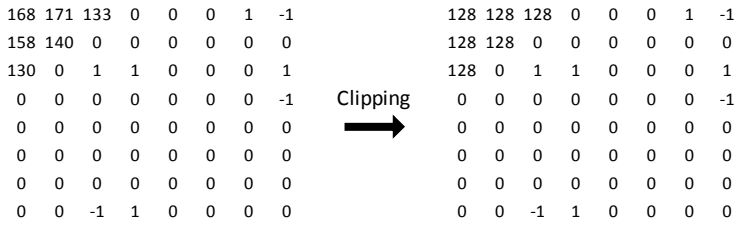


Figura 49: *Clipping* de uma matriz de quantização de um bloco.

Nesta etapa existe dependência de dados a nível do macrobloco para o controlo da Bit Rate e *clipping*.

4.2.1.7 VLC e Escrita para o Buffer

O codec de MPEG-2 do FFmpeg utiliza uma implementação própria do algoritmo de Huffman VLC escrevendo para o *buffer de bits* à medida que cada macrobloco é codificado sem a utilização de nenhuma representação intermédia dos dados binários dos macroblocos.

Nesta etapa existe dependência de dados a nível do macrobloco, de alguns parâmetros do contexto que são calculados e partilhados a nível do *slice* e do *buffer* de bits, uma vez que que cada macrobloco codificado tem de ser escrito para o buffer pela ordem correspondente à sua posição na *frame*.

4.2.2 Profiling

Para caracterizar o algoritmo e encontrar as etapas que são mais vantajosas de paralelizar, procedeu-se à análise do *codec* utilizando a ferramenta de *profiling* do Microsoft Visual Studio 2012. Os resultados obtidos encontram-se representados em percentagem no gráfico da Figura 50 e na Tabela 4. O processo de codificação utiliza um total de 55,61% do tempo total despendido pelo CPU na execução do programa. As etapas de codificação com maior percentagem de tempo do CPU são a DCT e Quantização com 25,63% e 13,63% respetivamente, a VLC requer 6,10% e as outras tarefas do programa que incluem leitura e escrita de ficheiro e descodificação do ficheiro de origem 44%.

Os resultados obtidos salientam que as etapas onde é possível um maior ganho são a DCT e Quantização que juntas constituem aproximadamente 70,60% do tempo total do algoritmo de

codificação e que cerca de 12,37% do algoritmo são tarefas sequenciais de cálculo de parâmetros do contexto e chamadas de funções que não podem ser paralelizadas.

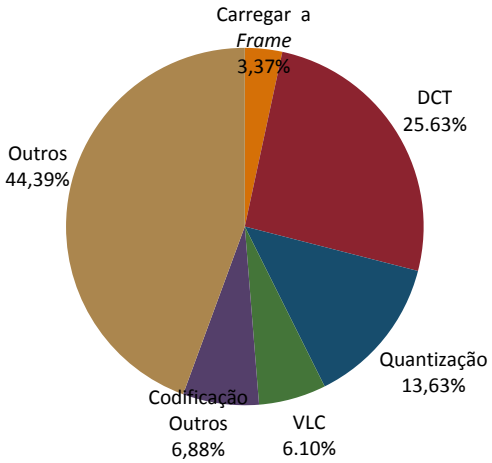


Figura 50: Gráfico representativo da distribuição de tempo despendido no programa de codificação de vídeo.

Tabela 4: Percentagem do tempo de codificação por etapa.

Profiling	*Total (%)*
Carrega a frame	3,37
DCT	25,63
Quantização	13,63
VLC	6,10
Codificação - Outros	6,88
Codificação - Total	55,61

As dependências de dados existentes relativas ao contexto do *codec* estão registadas na Tabela 5 em função do nível de dependência. Analisando as dependências da tabela conclui-se que é possível paralelizar as etapas de carregar a imagem, DCT, Quantização e *Clipping* replicando a matriz e o contador de *overflow* para cada bloco da *frame*. A etapa de VLC é mais complexa sendo necessário replicar alguns dos parâmetros de codificação ao nível do *slice* e o bit *buffer* garantindo que todos os *slices* são escritos para o *buffer* final pela ordem correta.

Tabela 5: Tabela representativa das dependências de dados do algoritmo do FFmpeg.

Variável / *Etapa*	*Parâmetros de qualidade*	*Buffer da Matriz*	*Bit Buffer*	*Contador de Overflow*
Inicialização	RW/Vídeo			
Escrita do *Header*			W/*Frame*	
Carregar *Frame*		W/Bloco		
DCT		RW/Bloco		
Quantização	R/*Frame*	RW/Bloco		W/Bloco
Clipping	R/*Frame*	RW/Bloco		R/Bloco
VLC	RW/*Slice*	R/Bloco	W/*Frame*	
Escrita da *tail*			W/*Frame*	

R – Leitura W-Escrita RW-Leitura e Escrita

Algoritmo de Codificação: MPEG-2 IMX

Recorrendo à lei de Amdahl pode calcular-se o speedup teórico de um algoritmo parcialmente paralelizável para *n* unidades de processamento conhecendo a distribuição do tempo entre as partes sequenciais e paralelizáveis do algoritmo. Aplicando a lei de Amdahl obteve-se a expressão (2) para o *speedup* teórico do programa de codificação.

$$T_{paralelo} = DCT + Quant + VLC \quad (1)$$

$$S(p) = \frac{T(1)}{T(p)} = \frac{Total}{Seq + \frac{1 - Seq}{p}} = \frac{Total}{Total - T_{paralelo} + \frac{T_{paralelo}}{p}} = \frac{100}{51{,}27 + \frac{48{,}73}{p}} \quad (2)$$

Utilizando a expressão anterior, traçou-se o gráfico de *speedup* em função do número de unidades de processamento da Figura 51. Como pode concluir-se do gráfico, a elevada quantidade de tempo despendida a ler e escrever do ficheiro e inicializar o *codec* condiciona o *speedup* máximo obtido pelo programa que tende para 1,95 para um número infinitamente grande de unidades de processamento. Este valor poderia ser melhorado utilizando um disco de maior performance, como por exemplo SSD ou hibrido, que reduziria consideravelmente o tempo gasto a ler o ficheiro *raw* do sistema.

Relativamente ao algoritmo de codificação o *speedup* de Amdahl é representado em função do número de núcleos pela expressão (3).

$$S(n) = \frac{T(1)}{T(p)} = \frac{Total_{codificação}}{Total_{codificação} - T_{paralelo} + \frac{T_{paralelo}}{p}} = \frac{55{,}61}{4{,}34 + \frac{48{,}73}{p}} \quad (3)$$

A Figura 52 representa a curva do *speedup* em função do número de unidades de processamento utilizando a expressão anterior e cujo limite tende para 12,81 para um número infinitamente grande de unidades de processamento. Como se pode constatar, o *codec* tem valores de *speedup* máximos mais elevados devido ao menor peso das tarefas sequenciais no algoritmo que aumentam a sua escalabilidade com o número de unidades de processamento.

Algoritmo de Codificação: MPEG-2 IMX

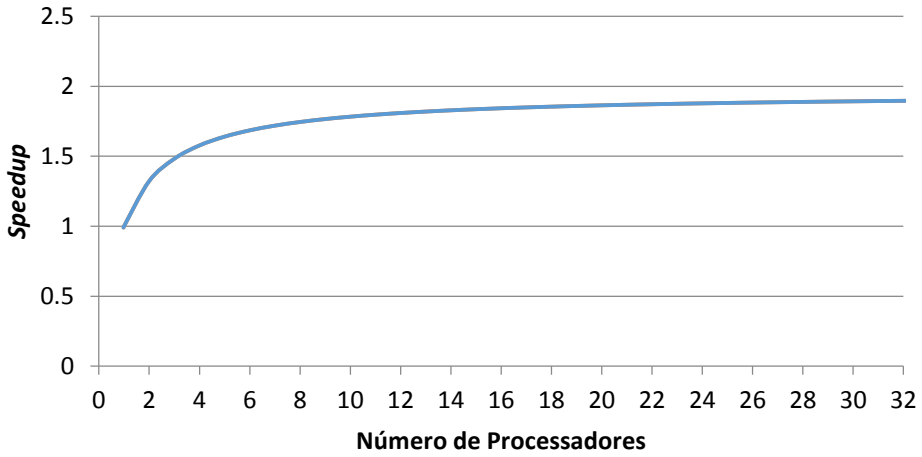


Figura 51:Gráfico do *speedup* de Amdahl do programa de codificação em função do número de unidades de processamento.

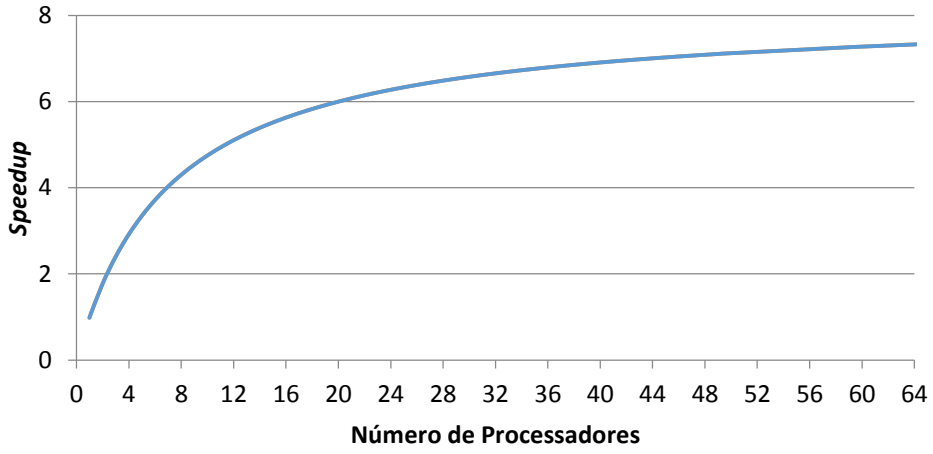


Figura 52: Gráfico do *speedup* de Amdahl do *codec* MPEG-2 em função do número de unidades de processamento.

Capítulo 5

Paralelização do Algoritmo

Neste capítulo analisam-se as especificações das soluções implementadas e alguns detalhes relevantes de implementação. A solução implementada consistiu na criação de um novo *codec* no FFmpeg, que utiliza como estrutura de dados o mesmo contexto do *codec* MPEG-1/2 já existente para garantir a interoperabilidade de funções. As soluções implementadas consistem em:

- Solução de computação paralela com o CPU – desenvolvida recorrendo ao OpenMP como API de computação paralela;
- Solução de computação heterogénea com o CPU e o GPU – desenvolvida utilizando OpenMP para paralelizar o trabalho ao nível do CPU e utilizando como plataforma de computação heterogénea o CUDA da NVIDIA;
- Ferramenta de Otimização da Performance – que permite o cálculo automático da distribuição ótima de *slices* para a execução da solução de computação heterogénea com o CPU e o GPU.

5.1 Solução utilizando Computação Paralela no CPU

A solução implementada, utilizando computação paralela com o CPU, sustenta-se numa alteração da ordem lógica da aplicação das etapas do algoritmo base do FFmpeg, dividindo o algoritmo em diferentes fases nas quais um conjunto de etapas são aplicadas a todos os macroblocos utilizando diferentes núcleos do CPU como exemplificado na Figura 53. Para implementar este algoritmo foi necessário atender às dependências de dados, estudadas no capítulo anterior, criando um conjunto de estruturas para conter os dados de cada macrobloco de

forma a utilizar um modelo de paralelismo de dados que não era possível reutilizando a mesma estrutura para todos os macroblocos. Assim distinguem-se três fases neste algoritmo:

1. Importar os macroblocos para as respetivas estruturas – os dados de cada macrobloco são copiados utilizando SSE para a matriz de frequências da estrutura que lhe corresponde;

2. Calcular DCT, Quantização e *Clipping* – é aplicada paralelamente a função de DCT, Quantização e *Clipping* a cada bloco de um macrobloco*;*

3. Aplicar VLC – os macroblocos são codificados para um *buffer* correspondente ao *thread* em que são processados.

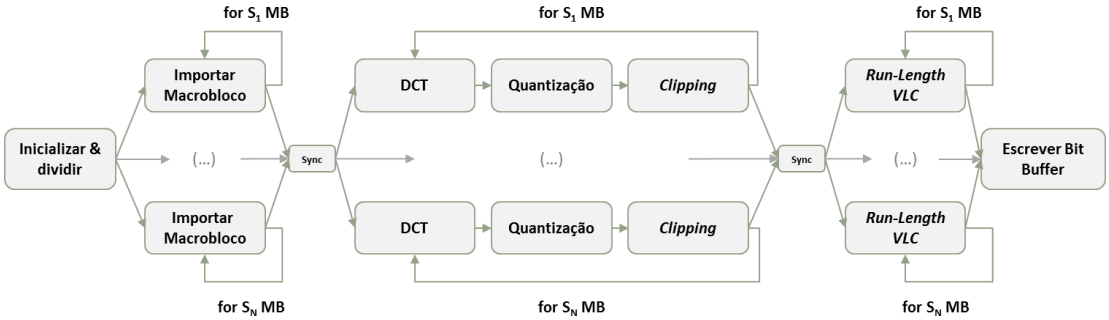


Figura 53: Diagrama do algoritmo de codificação MPEG-2 IMX implementado utilizando paralelismo de dados para N *threads* do CPU. Neste diagrama apenas estão exemplificados 2 *threads*.

5.1.1 Estrutura de dados

Na solução implementada foram desenvolvidas estruturas, C *struct,* para conter os dados do contexto do *codec* correspondentes a cada *slice*/macrobloco da *frame*. Isolando estes dados foi possível remover as dependências do contexto existentes entre as iterações de codificação e implementar o algoritmo descrito no ponto 4.2 FFmpeg. Estas estruturas são alocadas apenas uma vez na inicialização do *codec* e eliminadas quando o *codec* é encerrado para evitar o *overhead* da alocação de memória dinâmica. Todas as estruturas foram implementadas utilizando alinhamento de 128 bits. O diagrama UML da Figura 54 contém uma visão hierárquica sobre as estruturas seguidamente descritas:

- Mpeg2ImxFrame – esta estrutura contém as estruturas Mpeg2ImxSlice e as dimensões da *frame;*

- Mpeg2ImxSlice – esta estrutura contém as estruturas Mpeg2ImxMacroBlock e dados relativos ao processo de codificação do *slice*. Estes dados são calculados e atualizados ao longo da codificação do *slice* e não dependem dos dados de outros *slices*;

- Mpeg2ImxMacroBlock – esta estrutura contém as matrizes de DCT/Quantização dos blocos que fazem parte do macrobloco, as suas dimensões e os índices do último elemento de cada bloco quantizado a zero.

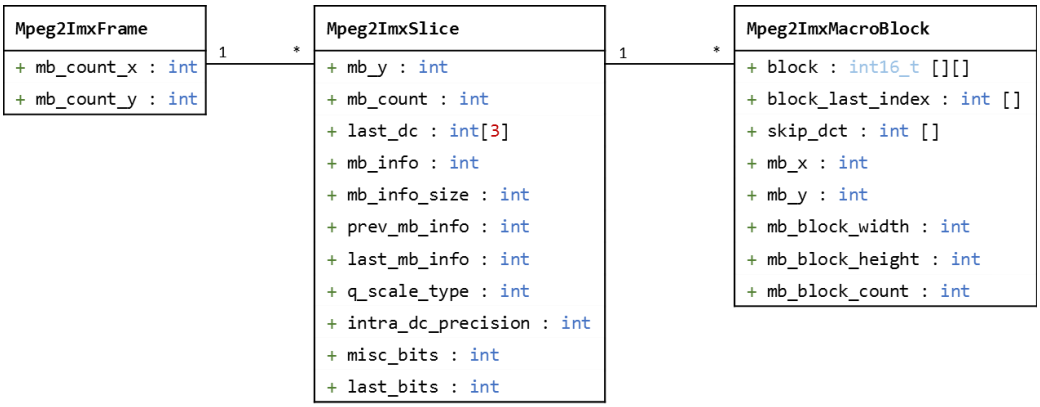


Figura 54: Diagrama UML de classes das estruturas implementadas.

5.1.2 Paralelismo e Alocação das tarefas

A API de computação paralela utilizada para implementar este algoritmo foi o OpenMP por ser flexível e de simples implementação. Para eliminar o *overhead* de criação e destruição dos *threads*, eles são criados quando o *codec* é inicializado e reutilizados para diferentes tarefas. Os *threads* têm o mesmo tempo de vida do *codec* ficando inativos sempre que não têm tarefas alocadas. A Figura 55 representa um diagrama da utilização e tempo de vida dos *threads* utilizados pelo *codec,* salientando as etapas que são paralelizadas em cada *thread*.

Paralelização do Algoritmo

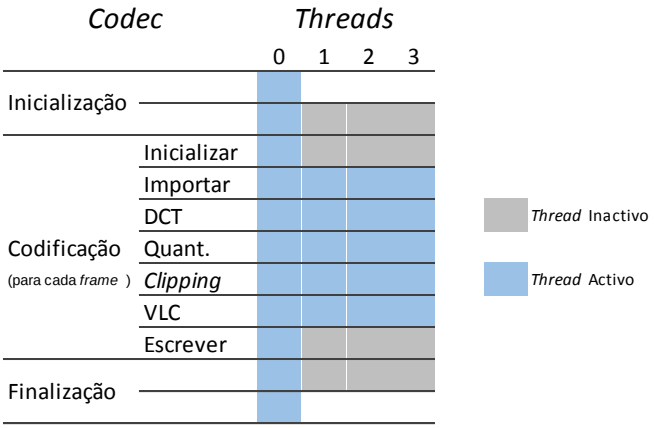


Figura 55: Diagrama da utilização e tempo de vida dos *threads* no *codec* implementado. Na figura está representado o funcionamento do *codec* num processador com quatro núcleos.

A Figura 56 contém um excerto do código utilizado para paralelizar tarefas com OpenMP. O *pragma omp parallel* permite definir um bloco de código que vai ser executado em cada *thread* como se fosse uma tarefa, enquanto o *pragma omp for* distribui automaticamente as iterações do ciclo pelos *threads* de forma estática durante a compilação, balanceando o número de iterações alocadas a cada *thread* e sincronizando todos os *threads* no final do ciclo.

```
// Define o número de threads a serem instanciados durante a execução
// omp_set_num_threads(cpu_thread_count); @ Init

// Declara a secção a ser paralelizada nos threads do OpenMP com variáveis
// privadas para cada thread, todas as variáveis não especificadas são
// partilhadas entre os threads.
// Cada thread processa uma tarefa com uma "cópia" deste bloco
#pragma omp parallel private (...)
{

// Define um ciclo "for" que vai ser paralelizado entre os threads
// dividindo as suas iterações pelos threads previamente definidos
#pragma omp for
     for (...) {
          task1(...);
     }
// Depois do omp for, os threads são automaticamente sincronizados
}
```

Figura 56: Excerto de código da implementação de um bloco de código em paralelo com OpenMP utilizando um *pragma* para distribuir as iterações de um ciclo *for* entre os *threads*.

Para distribuir o trabalho entre os *threads,* a *frame* é dividida horizontalmente em *chunks* (grupos de *slices*) da mesma dimensão podendo o último *chunk* ter um número diferente de *slices* para compensar. Cada *chunk* é alocado a um *thread* e processado. A Figura 57 representa a divisão da *frame* em *chunks*.

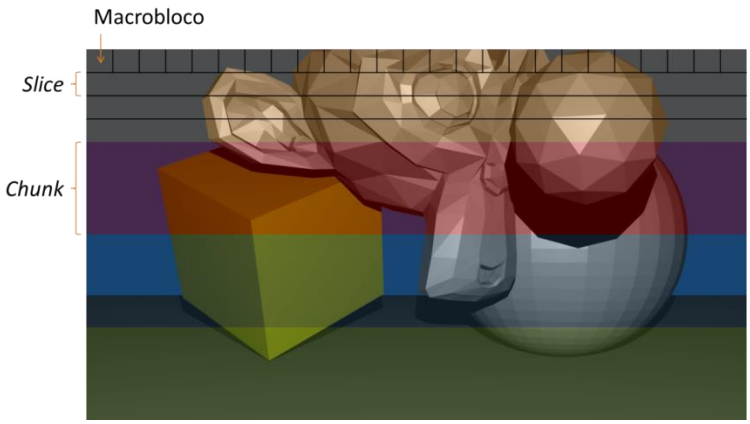


Figura 57: *Frame* dividida em 4 *chunks* constituídos por *slices* horizontais. Na imagem, o *chunk* superior está dividido numa grelha correspondente aos *slices* e macroblocos que o constituem.

5.1.3 Bit *Buffer*

O FFmpeg utiliza um *buffer* de bits intermédio para escrever a *frame* codificada antes de a exportar para o ficheiro. Uma vez que em C o tamanho mais pequeno que é suportado é um byte, escreve-se um byte de cada vez para o *buffer*, utilizando *shifts* à esquerda para colocar os bits num byte antes de o copiar para o *buffer*, como representado na Figura 58. Para permitir a paralelização da VLC replicou-se o *buffer* para cada *thread e* implementou-se no final da codificação a união destes *buffers* para obter a *frame* resultante. No MPEG-2 todos os códigos de início têm um byte de alinhamento, como tal, apenas foi necessário preencher o último byte incompleto de cada *buffer* com zeros e juntar os *buffers* como se fossem simples *arrays* de bytes.

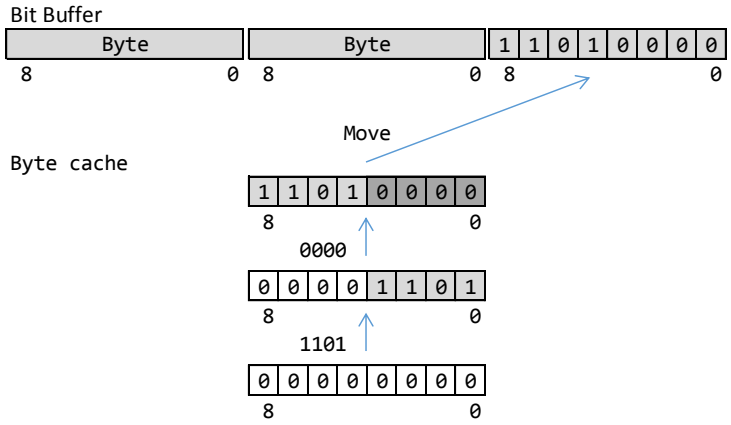


Figura 58: Diagrama exemplificativo do processo de escrita de 8 bits para o *buffer* de bits.

5.2 Solução utilizando Computação Heterogénea com o CPU e GPU

5.2.1 Algoritmo

A solução implementada utilizando computação heterogénea com o CPU e GPU foi desenvolvida sobre a solução de computação paralela com o CPU estendendo as funcionalidades dessa implementação com a utilização de uma plataforma CUDA. O GPU é utilizado para o *offload* de processamento do CPU nas etapas de maior paralelismo de dados, o cálculo da DCT e quantização, alocando parte do trabalho destas etapas para o GPU. O diagrama da Figura 59 esquematiza o algoritmo implementado e a divisão das diferentes etapas entre os dispositivos, de seguida descrito com maior detalhe:

1. Importar os macroblocos para as respetivas estruturas – os dados de cada macrobloco são copiados pelos núcleos do CPU, utilizando SSE, para a matriz de frequências da estrutura que lhe corresponde;

2. Aplicar DCT, quantização e *clipping* – O trabalho é dividido entre o CPU e o GPU nos quais é aplicada a função de DCT, quantização e *clipping* a cada bloco de um macrobloco. No CPU esta etapa consiste no processamento sequencial em cada *thread* de cada um dos blocos alocados a esse *thread*. No GPU os blocos são processados paralelamente com um *thread* por bloco;

3. Aplicar VLC – os macroblocos são codificados pelo CPU para um *buffer* correspondente ao *thread* em que são processados.

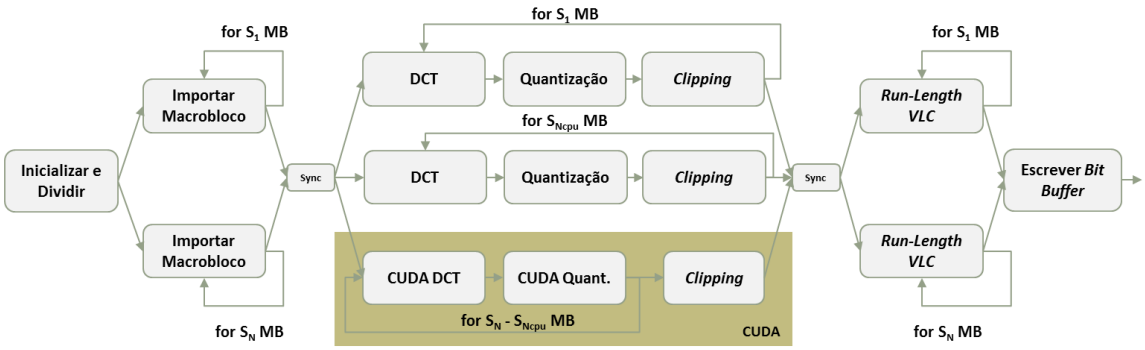


Figura 59: Diagrama do algoritmo de codificação MPEG-2 IMX implementado utilizando paralelismo de dados para N *threads* de processamento do CPU e o GPU utilizando a plataforma CUDA. A caixa verde contém operações processadas na plataforma CUDA, todas as outras são processadas por defeito pelo CPU.

5.2.2 Paralelismo e Alocação de Tarefas

Para interagir com a plataforma de computação heterogénea foi implementado um *thread* de controlo responsável por gerir e executar assincronamente funções no dispositivo. Este *thread* é ativado quando são alocadas tarefas de DCT e quantização ao GPU permanecendo inativo quando não tem tarefas alocadas, nomeadamente, durante a execução de código sequencial no CPU. O diagrama da Figura 60 representa o tempo de vida e estado de ativação dos *threads* utilizados nesta implementação. O *thread* 4 da Figura 60 corresponde ao *thread* de controlo do GPU num sistema com um CPU de quatro núcleos.

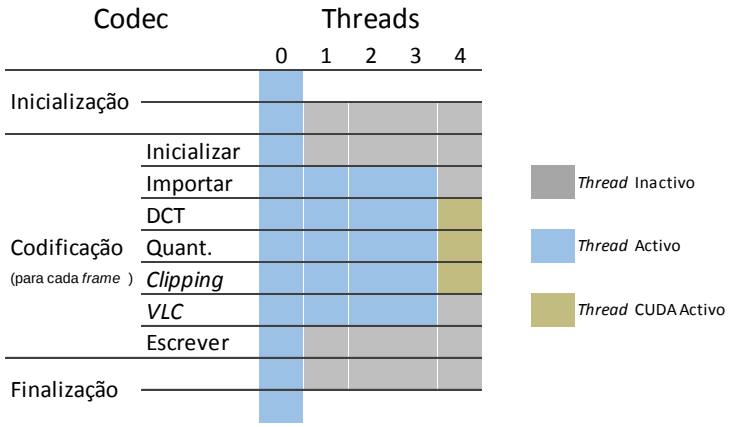


Figura 60: Diagrama da utilização e tempo de vida dos *threads* da solução implementada. A figura representa o funcionamento do *codec* num sistema com quatro processadores lógicos e um GPU CUDA controlado pelo *thread* 4.

A Figura 61 contém um excerto de código utilizado nesta solução para a implementação de tarefas processadas apenas pelo CPU e para a implementação de tarefas com *offloading* para o GPU. Nesta implementação substitui-se o *pragma omp for* pela alocação manual dos *chunks* utilizando *macros* que permite distribuir diferentes quantidades de processamento entre o CPU e GPU. Para blocos de código apenas processados pelo CPU os *chunks* são distribuídos homogeneamente pelos *threads* de processamento. Para blocos de código processados pelo CPU e o GPU o problema é dividido entre as plataformas conforme parametrizado em *runtime*, com a parte do CPU a ser dividida homogeneamente entre os *threads* de processamento enquanto a parte correspondente ao GPU é processada pelo *thread* de controlo da plataforma CUDA.

```
// Desactiva o spin-wait do scheduler do CUDA.
cudaSetDeviceFlags(cudaDeviceScheduleYield);

#pragma omp parallel private (...)
{
        // Obtém o identificador do thread que está a ser processado
        int thread_id = omp_get_thread_num();

        // Bloco apenas processado pelo CPU.
        // Verifica se o identificador do thread corresponde aos threads do CPU
        // ou se é o thread de controlo do CUDA
        if (thread_id < cpu_thread_count){
                // Caso seja um thread de processamento do CPU
                // calcula o chunk correspondente a este thread e processa-o
                GET_CHUNK_CPU(thread_id);
                cpu_task0(...);
        }
        // Sincroniza os threads
        #pragma omp barrier

        // Bloco processado pelo CPU e GPU cooperativamente.
        // Verifica se o identificador do thread corresponde aos threads do CPU
        // ou se é o thread de controlo do CUDA.
        if (thread_id < cpu_thread_count){
             // Caso seja um thread de processamento do CPU, calcula o chunk
             // do CPU e processa-o
                GET_CHUNK2_CPU(thread_id);
                cpu_task1(...);
        } else {
             // Caso seja o thread de controlo do GPU, calcula o chunk do GPU e processa-o
                GET_CHUNK2_GPU(thread_id);
                gpu_task1(...);
                asd
        }
}
```

Figura 61:Excerto de código da implementação de tarefas com etapas distribuídas pelo CPU e CPU + GPU.

5.2.3 Cópia da memória para o GPU

No desenvolvimento desta solução foram implementados três métodos de acesso aos dados da memória RAM pelo GPU, de seguida descritos:

1. Cópia síncrona de memória – consiste em alocar no GPU um bloco de memória das mesmas dimensões do bloco a copiar da memória RAM e chamar uma função que é executada pelo CPU e copia sincronamente o bloco de memória RAM para a memória do GPU;

2. Cópia por DMA de memória não paginável – consiste em alocar no GPU um bloco de memória das mesmas dimensões do bloco a copiar da memória RAM e transmitir ao sistema de DMA do CUDA os apontadores do bloco de memória da RAM e da memória do GPU para que este seja copiado assincronamente pelo DMA do GPU. Este método requer que o bloco de memória alocado na memória RAM seja declarado

como não paginável, isto é, que não possa ser movido ou copiado para um ficheiro de paginação pelo sistema operativo, garantindo assim que o apontador fornecido ao GPU é valido durante toda a execução do CUDA. Esta implementação permite a utilização de um *pipeline* com *streams*;

3. *Zero-Copy Pinned Memory* – este método, contrariamente aos anteriores, não implica a cópia do bloco para a memória do GPU mas sim permite o acesso direto do GPU à memória RAM através de DMA. No entanto, esta implementação aumenta a carga sobre o *bus* PCI-Express e revelou-se consideravelmente mais lenta que a cópia por DMA com *streams* nomeadamente por não permitir a sobreposição de operações de cópia e de *kernels*.

As imagens da Figura 62, da Figura 63 e da Figura 64 contêm excertos do código utilizado para implementar os métodos acima referidos.

```
// Cópia sincrona de memória paginável
Object * object_cpu, object_gpu;
// Aloca a variável na memória RAM
object_cpu = av_malloc(sizeof(Object));
// Aloca a variável na memória do GPU
cudaMalloc (&object_gpu, sizeof(Object));
// Copia os dados para o GPU
cudaMemcpy (object_gpu, object_cpu, sizeof(Object), cudaMemcpyHostToDevice);

// Invoca o kernel
kernel <<< BLOCKS, THREADS, SHARED_SIZE, STREAM >>> (object_gpu);
// Espera que o kernel termine antes de poder copiar os dados
cudaDeviceSynchronize()
// Copia os dados do GPU
cudaMemcpy (object_cpu, object_gpu, sizeof(Object), cudaMemcpyDeviceToHost);
```

Figura 62: Diagrama e excerto de código utilizado para a cópia síncrona de memória paginável.

```
// Cópia assíncrona por DMA de memória não paginável
Object * object_cpu, object_gpu;

// Activa o DMA
cudaSetDeviceFlags (cudaDeviceMapHost);
// Aloca a variável na memória RAM não paginável
cudaHostAlloc(&object_cpu, sizeof(Object), cudaHostAllocDefault);
// Aloca a variável na memória do GPU
cudaMalloc (&object_gpu, sizeof(Object));

// Copia assincronamente os dados para o GPU
cudaMemcpyAsync(object_gpu, object_cpu, sizeof(Object), cudaMemcpyHostToDevice, STREAM);

// Invoca assincronamente o kernel
kernel <<< BLOCKS, THREADS, SHARED_SIZE, STREAM >>> (object_gpu);

// Copia assincronamente os dados do GPU
cudaMemcpyAsync(object_cpu, object_gpu, sizeof(Object), cudaMemcpyDeviceToHost, STREAM);
```

Figura 63: Diagrama e excerto de código utilizado para a cópia assíncrona através de DMA de um endereço de memória não paginável para a memória do GPU.

```
// Zero-Copy Pinned Memory
Object * object_cpu, object_gpu;

// Activa o DMA
cudaSetDeviceFlags (cudaDeviceMapHost);
// Aloca a variável na memória RAM não paginável e mapeia-a para o GPU
cudaHostAlloc (&object_cpu, sizeof(Object), cudaHostAllocMapped);
// Obtém o apontador do objecto válido para a DMA do GPU
cudaHostGetDevicePointer(&object_gpu, object_cpu, FLAG = 0);

// Invoca assincronamente o kernel
kernel <<< BLOCKS, THREADS, SHARED_SIZE, STREAM >>> (object_gpu);
```

Figura 64: Diagrama e excerto de código utilizado para o acesso direto do GPU à memória RAM através de DMA.

Na solução desenvolvida a escolha do método de acesso e cópia de memória recaiu sobre a cópia por DMA de memória não paginável devido ao melhor desempenho e maior capacidade de sobrepor tarefas.

5.2.4 Sobreposição de tarefas / *Pipelining*

Para reduzir o *overhead* da cópia de dados entre a memória RAM e a memória do GPU implementou-se um *pipeline* utilizando a funcionalidade de *streams* do CUDA, que permite o encadeamento num *stream* de operações assíncronas de cópias de dados e execução de *kernels*. Os GPUs da plataforma CUDA podem processar simultaneamente três instruções de tipos diferentes (executar um *kernel*, realizar uma cópia de memória da RAM para o GPU e uma cópia de memória do GPU para a RAM), sobrepondo assim os *streams* à medida que os recursos são disponibilizados como representado no diagrama da Figura 65. Desta forma, a solução implementada consegue "esconder" o *overhead* das transações de memória sobrepondo-as durante a execução dos *kernels*, à exceção das primeiras cópias de memória para o GPU e das últimas cópias de memória do GPU, que não são sobrepostas.

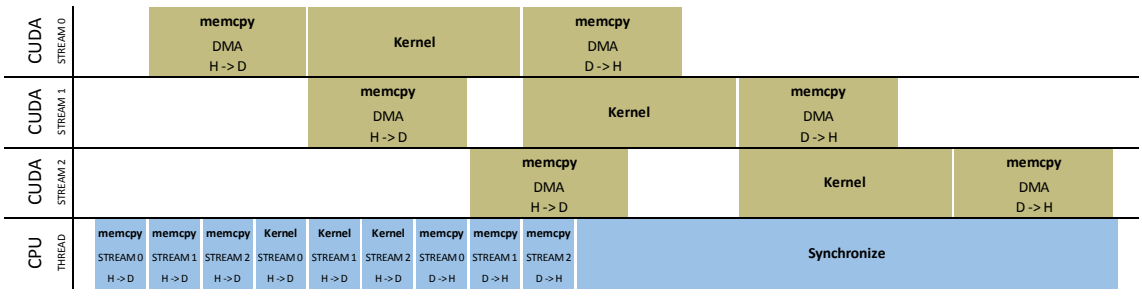


Figura 65: Diagrama de uma linha temporal de um *pipeline* de três *streams* CUDA. As operações de um *stream* são processadas sequencialmente pela ordem que são registadas. Os *streams* são processados concorrentemente à medida que os recursos são disponibilizados.

Paralelização do Algoritmo

A Figura 66 apresenta um excerto do código desenvolvido para a funcionalidade de sobreposição de tarefas. Nesta implementação dividiu-se o problema em *chunks* que podem ser processados num *stream* independente. Os *streams* são criados no *host* e as tarefas das diferentes etapas são alocadas ao *stream* respetivo ao *chunk* a ser processado. Esta implementação permite ainda que o CPU espere por cada *stream* permitindo sobrepor funções do CPU que utilizem o *chunk* que acabou de ser processado enquanto o GPU processa assincronamente os restantes *streams*.

```
// Aloca o array de variáveis na memória do GPU
Object * object_gpu;
cudaMalloc (&object_gpu, sizeof(Object) * object_count);
// Cria os streams que vão ser utilizados para processar cada objecto
cudaStream_t stream [object_count];

// Inicializa os streams
for (int i = 0; i < object_count; i++)
  cudaStreamCreate(&stream[i]);
//Copia assincronamente cada objecto para o GPU utilizando o stream correspondente
for (int i = 0; i < object_count; i++)
  cudaMemcpyAsync(object_gpu[i], object_cpu[i], sizeof(Object),cudaMemcpyHostToDevice,
stream[i]);

// Invoka o kernel para processar cada objecto utilizando o stream correspondente
for (int i = 0; i < object_count; i++)
  kernel <<<BLOCKS, THREADS, SHARED_SIZE, stream[i]>>> (object_gpu[i]);

// Copia assincronamente cada objecto do GPU utilizando o stream correspondente
for (int i = 0; i < object_count; i++)
  cudaMemcpyAsync(object_cpu, object_gpu, sizeof(Object), cudaMemcpyDeviceToHost, stream[i]);
// Espera que cada stream acabe
for (int i = 0; i < object_count; i++){
  cudaStreamSynchronize(stream[i]);
  // O CPU pode ainda reralizar operações em cada bloco enquanto os outros
  // streams são processados, sobrepondo-as ao GPU
  cpu_task0(object_gpu[i]);
}
```

Figura 66: Excerto de código utilizado para a implementação de tarefas sobrepostas na plataforma CUDA.

5.2.5 *Memory coallescing* e Memória Partilhada

O algoritmo e estruturas de dados implementados recorrem a um modelo de acesso à memória otimizado para CPUs no qual os blocos são lineares, isto é, os macroblocos contêm uma estrutura bidimensional em que cada linha corresponde a um bloco. Contrariamente ao CPU em que este modelo otimiza o funcionamento da *cache*, no GPU os *threads* de um *warp* devem aceder num ciclo a blocos de dados contíguos de forma a permitir que o controlador de memória possa realizar apenas um *coalesced access,* ou seja, possa condensar todos os acessos num único acesso de maiores dimensões aumentando a largura de banda efetiva e reduzindo a latência dos múltiplos acessos.

A Figura 67 apresenta um diagrama do modelo de acesso à memória de uma implementação utilizando paralelismo de dados para processar os blocos de um macro bloco. A performance desta implementação é reduzida devido à baixa eficiência deste padrão de acesso à memória do GPU.

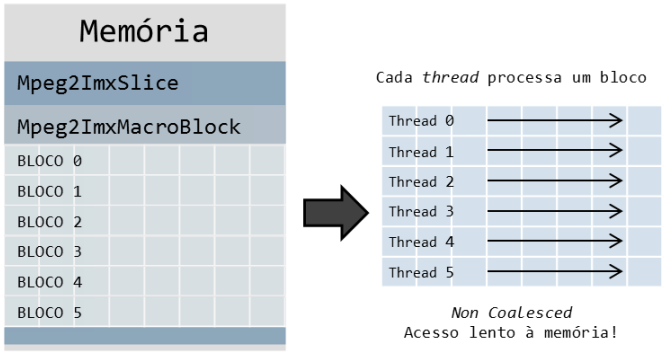


Figura 67: Diagrama do modelo de acesso não coalescido à memória pelos *threads* que processam os blocos de vídeo.

Uma das soluções estudadas para este problema foi a transposição da matriz de blocos de modo a tornar os acessos contíguos e colineares, no entanto, devido à complexidade desta solução e ao tempo despendido no processo de transposição optou-se pela utilização da memória partilhada do SMP para o processamento dos blocos. Sendo assim, a solução implementada consiste em realizar uma cópia utilizando *coalesced access* das estruturas para a memória partilhada (*on-chip* de alta velocidade) e de seguida, processar os dados da memória partilhada utilizando um padrão de acesso não coalescido uma vez que a velocidade e baixa latência da memória partilhada dispensa a necessidade de um padrão específico de acesso. O diagrama da Figura 68 representa o padrão de acesso coalescido à memória utilizado durante a cópia dos dados para a memória partilhada e o padrão de processamento não coalescido utilizando dados da memória partilhada. A Figura 69 contém um excerto do código implementado para copiar os dados entre as memórias e processa-los utilizando a memória partilhada.

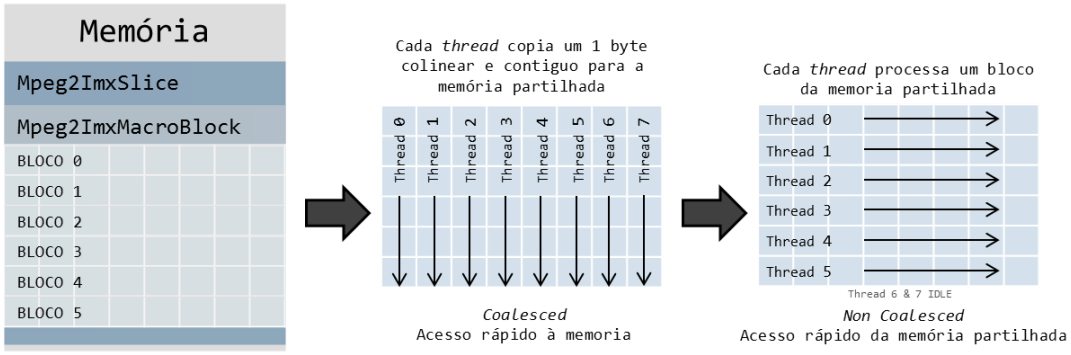


Figura 68: Diagrama do modelo de acesso coalescido à memória da solução implementada utilizando memória partilhada.

```
// Processa N objectos da shared memory em N threads

// Define o tamanho do tipo utilizado para copiar dados entre as memórias
#define COPY_TYPE uint32_t
#define COPY_SIZE sizeof(COPY_TYPE)

// Define a variável utilizada para aceder à shared memory do GPU
extern __shared__ Object cache[];

// Calcula o numero de COPY_TYPE que vão ser realmente copiados tendo em conta
// o número de objectos alocados para o kernel definidos em blockDim.x
const int copy_count = (sizeof(Object) * blockDim.x + COPY_SIZE-1) / COPY_SIZE;

// Cópia coalescida dos objectos para a memória partilhada
for(int i = threadIdx.x; i < copy_count; i += blockDim.x)
        ((COPY_TYPE*)&cache)[i] = ((COPY_TYPE*)object_gpu)[i];

// Sincroniza garantindo que todos os threads do warp estão neste ponto
__syncthreads();

// Executa uma função para cada Object
if (threadIdx.x < blockDim.x)
        gpu_task0(&cache[threadIdx.x]);

// Sincroniza garantindo que todos os threads do warp estão neste ponto
__syncthreads();

// Cópia coalescida dos objectos da memoria partilha para a memoria global
for(int i = threadIdx.x; i < copy_count; i += blockDim.x)
        ((COPY_TYPE*)object_gpu)[i] = ((COPY_TYPE*)&cache)[i];
```

Figura 69: Código utilizado para a cópia coalescida de um *array* de objetos para a *cache* e o processamento de cada um dos objetos num *thread* independente.

5.2.6 Ocupação do GPU e Alocação dos Threads

A plataforma CUDA requer que o problema seja dividido em blocos de *threads* por forma a alocar o problema às unidades de processamento da plataforma. As principais limitações à divisão do algoritmo implementado estão relacionadas com o número de registos em uso por cada *thread* e o tamanho da memória partilhada dos multiprocessadores que restringem o número de macro blocos alocados a um bloco de *threads*. Os *threads* de um bloco de *threads*

são executados sincronamente em *SPs* controlados pelo mesmo SMP. Posto isto, a implementação desenvolvida divide os conjuntos de macroblocos de um *slice* em blocos de *threads* que maximizem a utilização da memória partilhada, utilizando o menor número de blocos de *threads* possível por *slice*, com um *thread* por bloco da *frame*. O diagrama da Figura 70 contém um esquema da divisão de macro blocos por SMP descrita.

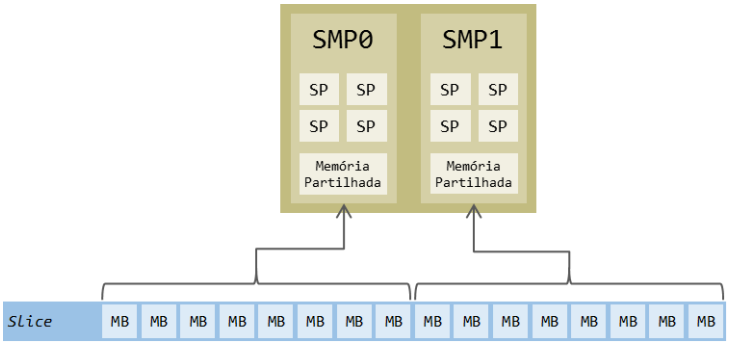


Figura 70: Divisão de um *slice* de 16 macroblocos por dois SMP cada um com capacidade para armazenar 8 macroblocos na memória partilhada. Cada bloco é processado por um *thread* independente.

5.2.7 Otimização da Performance

Atendendo ao funcionamento do algoritmo implementado utilizando sobreposição de *streams* para esconder o tempo de cópias de dados, concluiu-se que o tempo total despendido pelo GPU no cálculo da DCT e quantização varia linearmente na razão direta do número de *slices* processados pelo GPU com um valor inicial correspondente ao tempo das cópias de memória que não são sobrepostas. Assim, o tempo de processamento do GPU pode ser descrito pela função (4), em que t_{gpu} equivale ao tempo total de processamento do GPU, S_{gpu} é a variável correspondente ao número de *slices* processados no GPU, $t_{slice\ gpu}$ é um valor constante do tempo de processamento do *kernel* de cada *slice* e t_{memcpy} corresponde à soma do tempo da primeira cópia do *host* para o dispositivo e a última cópia do dispositivo para o *host*.

$$t_{gpu} = t_{slice\ gpu} * S_{gpu} + t_{memcpy\ gpu} \quad (4)$$

Posto isto, tendo em conta que o número de *slices* é constante pode-se formalizar duas expressões adicionais ao problema. A função (5) permite-nos calcular o tempo total de processamento do CPU sabendo o tempo despendido pelo CPU a processar um *slice* e o número de núcleos do CPU. Por último, a condição (6) garante que o somatório dos *slices* processados pelo CPU e pelo GPU são o total dos *slices* da *frame*.

$$t_{cpu} = t_{slice\ cpu} * S_{cpu} / N_{cores} \quad (5)$$

$$S_{total} = S_{cpu} + S_{gpu} \quad (6)$$

Com estas três condições pode-se formular um problema de otimização (7) utilizando um sistema para procurar a distribuição dos *slices* que permite explorar melhor as capacidades do sistema de computação combinada entre o CPU e o GPU que oferece o desempenho mais elevado, ou seja, minimiza a função objetivo (8).

$$\begin{cases} t_{gpu} = t_{slice\ gpu} * S_{gpu} + t_{memcpy\ gpu} \\ t_{cpu} = t_{slice\ cpu} * S_{cpu} / N_{cores} \\ S_{total} = S_{cpu} + S_{gpu} \end{cases} \quad (7)$$

$$\mathrm{T(S)} = max(t_{cpu}, t_{gpu}) \quad (8)$$

Uma vez que todas as operações do GPU são assíncronas é extremamente difícil isolar o tempo despendido exclusivamente nas cópias de dados. Utilizando a ferramenta de *profiling* do CUDA, estimou-se que as cópias de dados demoram sensivelmente o mesmo tempo de processamento de um *slice*, podendo se estimar os tempos utilizando as expressões (9).

$$t_{slice\ gpu} = t_{gpu} / (S_{gpu} + 1) \ \wedge \ t_{memcpy\ gpu} = t_{slice\ gpu} \quad (9)$$

Em termos computacionais este problema pode ser resolvido calculando o tempo médio de processamento de cada *slice* no CPU e no GPU e iterando sobre todas as combinações de *slices* possíveis, procurando a combinação que oferece o máximo desempenho, tal como implementado no excerto de código da Figura 71.

```
// Guarda melhor solução inteira
int best_n = 0;

// Procura todas as combinações
for(int n = 1; n < total_slices; n++){
        // Calcula os tempos para cada situação tendo em conta o tempo máximo
        int time = MAX(GPU_TIME(n), CPU_TIME(total_slices - n));
        int best_time = MAX(GPU_TIME(best_n), CPU_TIME(total_slices - best_n));

        // Se o tempo for melhor encontrou um novo n
        if(time < best_time)
                best_n = n;
}
```

Figura 71: Algoritmo para explorar e encontrar a distribuição ótima de *slices*.

Utilizando estas métricas e o algoritmo previamente referido implementou-se uma solução para analisar o perfil computacional de uma máquina e estimar a distribuição ótima dos *slices.* A Figura 72 apresenta um resultado da execução do programa de otimização para uma máquina de testes.

```
encoding frame 116 (size=86440)
encoding frame 117 (size=86252)
encoding frame 118 (size=86227)
encoding delayed frame 119 (size=86275)
Encoding Finished

Time spent encoding in Milis: 3364
Time spent total in Milis: 5664

CPU average per chunk: 220u
GPU average per chunk: 561u

Current CPU usage: 32/68
Current GPU usage: 32/68

Ideal CPU usage: 49/68
Ideal GPU usage: 22/68

Press any key to continue...
```

Figura 72: Exemplo de uma execução do programa de optimização de performance.

Capitulo 6

Resultados e Discussão

Um dos principais objetivos desta dissertação consistiu na avaliação dos ganhos de performance das soluções implementadas em relação à implementação original do FFmpeg. Este capítulo está subdividido nos seguintes tópicos:

- Descrição da metodologia utilizada para a medição dos resultados;
- Avaliação da solução de computação paralela com o CPU e discussão dos resultados;
- Avaliação da solução de computação heterogénea com o CPU e o GPU e discussão dos resultados;
- Avaliação dos resultados da ferramenta de otimização da distribuição ótima de *slices* entre o CPU e o GPU e discussão;
- Considerações sobre a escalabilidade das soluções implementadas.

6.1 Metodologia

Para avaliar as soluções implementadas procedeu-se à medição do tempo de codificação de um vídeo de testes em formato *raw* para MPEG-2 IMX utilizando o *codec* original do FFmpeg e as duas soluções implementadas. Atendendo ao estudo realizado no capítulo 4.2.2 Profiling, a medição dos programas de codificação realizou-se em dois níveis, num nível superior mediu-se o tempo total de execução do programa que inclui o tempo despendido na inicialização de variáveis e *threads*, o tempo da execução do algoritmo de codificação e a leitura e escrita de ficheiros enquanto num nível inferior mediu-se apenas o tempo de execução do algoritmo de

codificação como exemplificado na Figura 73. Uma vez que o código de leitura e escrita de ficheiros e da chamada à biblioteca de codificação é o mesmo para todos os casos de teste e já foi analisado no capítulo 4.2.2 *Profiling*, neste capitulo apenas consideramos a performance do algoritmo de codificação.

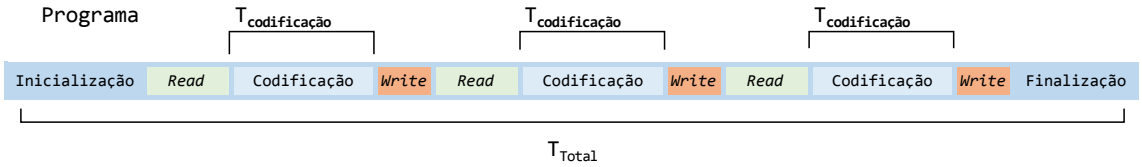


Figura 73:Diagrama do esquema utilizado para a medição do tempo total de execução e tempo de codificação.

O vídeo utilizado nestas medições consiste num vídeo *raw* em formato UYVY (*YUV 4:2:2*) com resolução *Full HD* de 1920 por 1080 pixéis e uma duração de 4,8 segundos a 25 fps, com um tamanho no disco de 470 MB (493.516.800 bytes). O vídeo contém um conjunto de figuras tridimensionais que incluem a cabeça de um macaco, um cubo e duas esferas das quais uma tem movimento, colocados sobre uma superfície preta com um fundo azul. A Figura 74 representa a primeira *frame* do vídeo de testes.

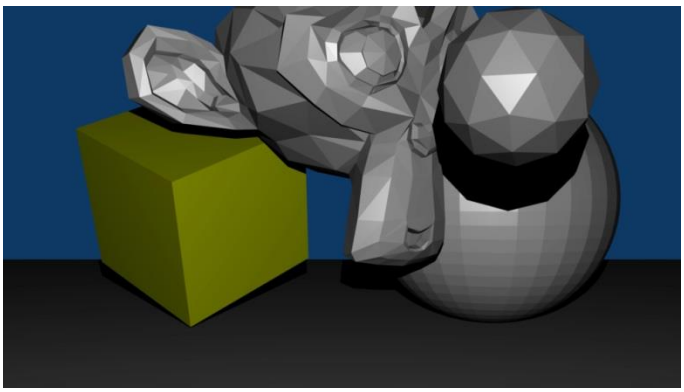


Figura 74: Primeira *frame* do vídeo de teste utilizado

O vídeo foi codificado para MPEG-2 IMX com *GOPs* de uma *frame* do tipo I, formato de cor 4:2:0 e com uma taxa de bits máxima de 18Mbps. O vídeo resultante tem 10 MB (10.551.399 bytes) o que equivale a uma taxa de compressão de 0,021 como calculado em (10).

$$Taxa\ de\ Compressão = \frac{Tamanho_{comprimido}}{Tamanho_{não\ comprimido}} = \frac{10.551.399}{493.516.800} = 0{,}021 \qquad (10)$$

Os resultados foram obtidos utilizando dois computadores cujas características estão especificadas na Tabela 6. O primeiro computador é um *desktop* com um CPU de dois núcleos e

um GPU discreto com memória dedicada. O segundo computador é um *netbook* equipado com um processador de baixo consumo de dois núcleos com *multi threading* e um GPU integrado da NVIDIA. Ambos os sistemas correm Microsoft Windows versão 6 ou superior e são compatíveis com a plataforma CUDA (*Compute Capability* 1.1 ou superior).

Sistema		*Computador 1 (PC 1)*	*Computador 2 (PC 2)*
CPU	Modelo	Intel Core 2 Duo E4400	Intel Atom 330
	Núcleos	2 @ 2000 MHz	2 w/HT @ 1600 MHz
	Cache	L1 32 KB, L2 2048 KB	L1 2x24 KB, L2 2x512 KB
RAM		2048+1024 MB DDR2-666	2x2048 MB DDR2-800
GPU	Modelo	NVIDIA GeForce 8500 GT	NVIDIA Ion (GeForce 9400M)
	Memória	512 MB DDR2-666	256 MB Partilhados
	CC	1.1	1.1
	SMP	2 @ 450 MHz	2 @ 450 MHz
	MP	16 @ 900 MHz	16 @ 1100 MHz
Disco		HDD 7200 rpm, SATA	HDD 5400 rpm, SATA
Sistema Operativo		Windows 7 Enterprise 64bits	Windows 8 Pro 32bits

Tabela 6: Especificações dos sistemas de teste utilizados.

6.2 Processamento Sequencial

Antes de comparar os resultados obtidos utilizando múltiplos núcleos, optou-se por comparar a performance sequencial dos dois algoritmos utilizando apenas um *thread* e consequentemente um núcleo do CPU. Os resultados obtidos para o tempo de processamento da solução implementada em comparação com a implementação original do FFmpeg estão representados no gráfico da Figura 75. Estes resultados revelam uma ligeira diferença entre o tempo de execução dos dois *codecs* quando é utilizado apenas uma unidade de processamento/*thread*. No computador 1 esta discrepância é favorável à implementação sequencial do FFmpeg. Tal facto é causado pela utilização de diferentes posições de memória para armazenar os dados da *frame* na solução implementada, que causa um aumento do número de *cache misses* e consequentemente um aumento do número de acessos à memória comparativamente ao *codec* original que reutiliza o *buffer*. Outro fator de perda de desempenho é a definição das etapas de codificação como um conjunto de tarefas do OpenMP que apesar de ser inicializado apenas um *thread* em *runtime* introduz no processo um *overhead* de alocação das tarefas aos *threads*. No computador 2, os resultados invertem-se devido à menor capacidade de processamento do CPU, que demora mais tempo a processar instruções, reduzindo o impacto do tempo de espera dos *cache misses* e maximizando os ganhos temporais da remoção de

operações redundantes para a codificação no formato IMX que são realizadas no *codec* FFmpeg como a predição de movimentos *inter-frame*.

A nível das máquinas de testes, o computador 2 alcançou resultados consideravelmente mais lentos que os do computador 1 derivados da menor capacidade de processamento do CPU de baixo consumo energético e das menores quantidades de *cache* L1 e L2.

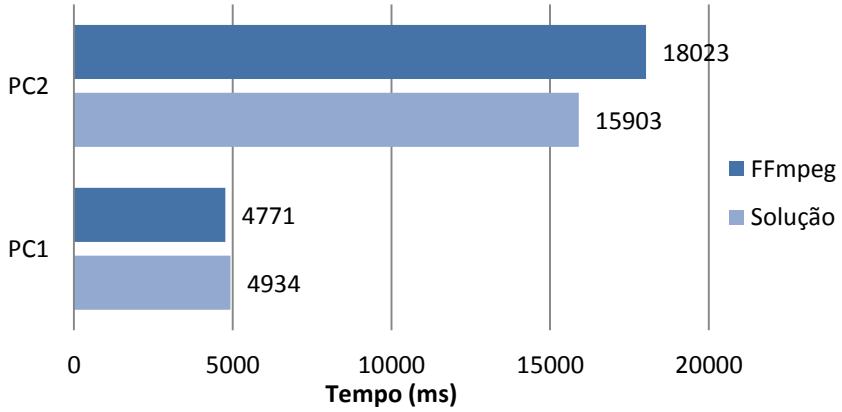


Figura 75: Comparação da performance do *codec* do FFmpeg e da solução de computação paralela implementada quando executadas sequencialmente em apenas um processador.

6.3 Processamento em Paralelo com o CPU

Para quantificar a performance de processamento em paralelo com o CPU mediram-se os tempos de codificação dos dois *codecs* utilizando múltiplos *threads* obtendo-se os resultados representados pelos gráficos da Figura 76 e Figura 77.

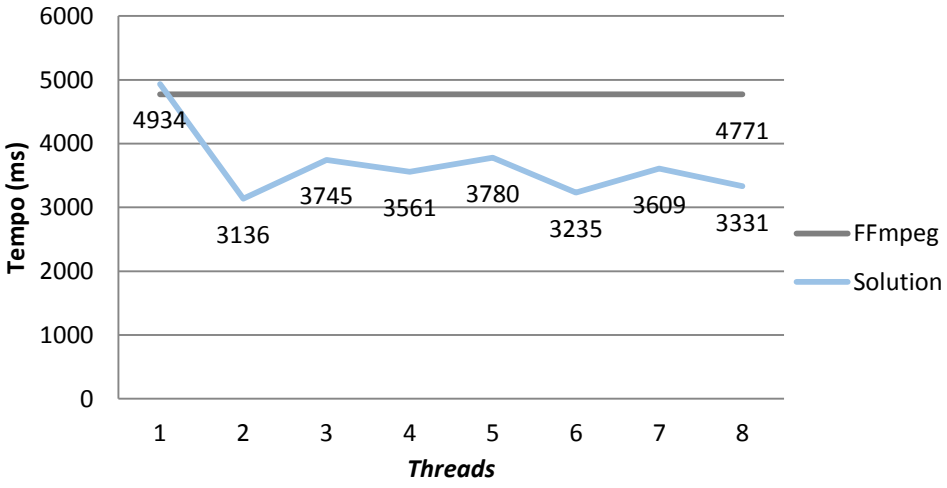


Figura 76: Tempo de codificação com múltiplos *threads* no Computador 1

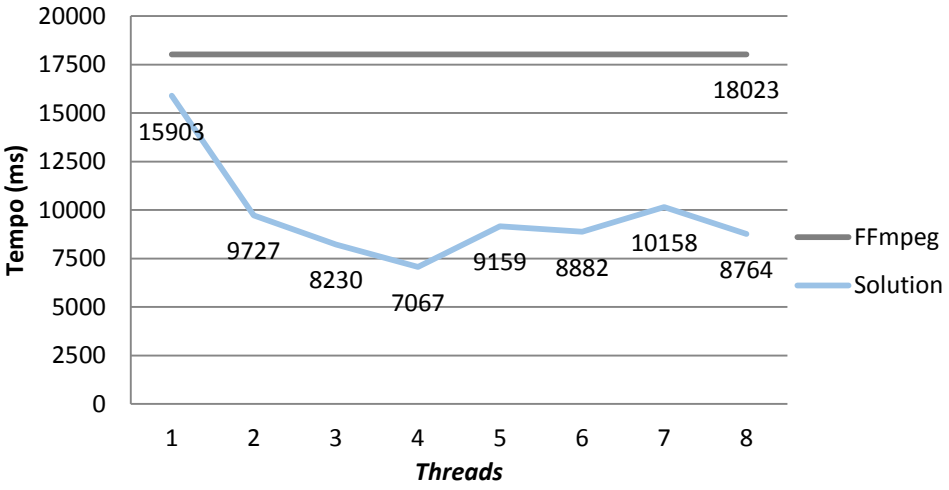


Figura 77: Tempo de codificação com múltiplos *threads* no Computador 2

Estas medições provam que existe uma melhoria de desempenho considerável quando são utilizados múltiplos *threads*. Com base nestes resultados e para melhor se avaliar os ganhos de performance, calcularam-se os valores do *speedup* do algoritmo implementado em relação ao *codec* do FFmpeg utilizando a fórmula previamente referida e representaram-se graficamente os resultados obtidos na Figura 78 e Figura 79.

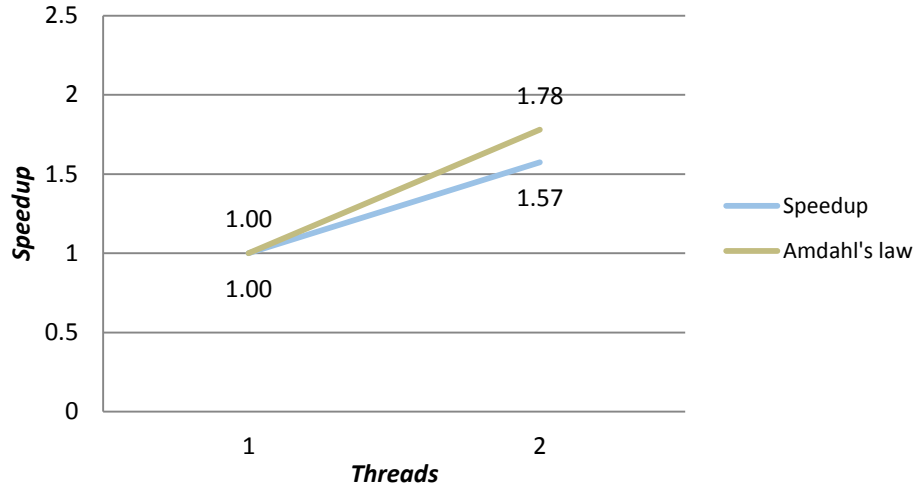


Figura 78: *Speedup* da solução implementada utilizando *n threads* no Computador 1.

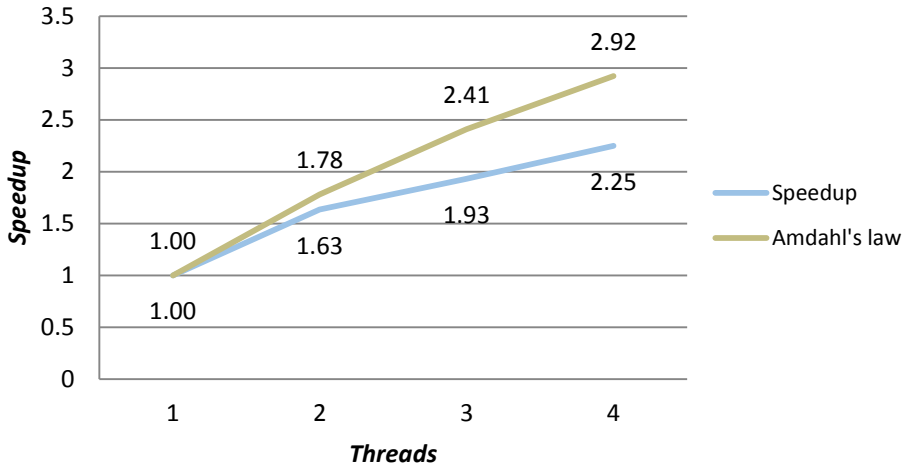


Figura 79: *Speedup* da solução implementada utilizando *n thread* no Computador 2.

Como podemos concluir, os resultados obtidos são positivos, apesar de se afastarem um pouco do *speedup* máximo de Amdahl devido ao *overhead* da alocação de tarefas e sincronização dos *threads* assim como à necessidade de realizar processamento sequencial extra para a replicação e sincronização de dados e para a união do *bit buffer* final. Os resultados permitem-nos ainda concluir que, apesar de ambos os computadores terem CPUs de dois núcleos, no Computador 2 o algoritmo continuou a ter ganhos de desempenho até quatro *threads* o que evidencia a escalabilidade do algoritmo implementado com tecnologias de *simultaneous multi-threading* como o *Hyper-Threading* da Intel, alcançando níveis de desempenho equiparáveis aos níveis obtidos com a utilização de um núcleo físico adicional por cada núcleo virtual do *Hyper-Threading*. Este ganho de performance está relacionado com o facto de o *Hyper-Threading* duplicar as unidades de execução dos núcleos permitindo a um núcleo alternar entre dois *threads,* sempre que um dos *threads* fique em espera devido a um *cache miss* ou outra operação que consuma um largo número de ciclos, sem incorrer numa perda de desempenho.

Relativamente ao comportamento do sistema, quando são instanciados mais *threads* do que unidades de processamento lógico, a performance decai principalmente quando o número de *threads* não é múltiplo do número de núcleos do CPU, uma vez que nesta situação o trabalho não é distribuído homogeneamente entre os núcleos ocorrendo uma perda de recursos dos núcleos que não têm tarefas alocadas durante uma parte da codificação. De salientar ainda que mesmo nos casos de distribuição homogénea é introduzido na codificação um *overhead* sempre que o processador troca o contexto dos *threads* que está a processar.

6.4 Processamento em Paralelo com o CPU e GPU

A solução de computação heterogénea combina a capacidade de processamento do CPU e do GPU para processar o vídeo. Para avaliar a performance desta solução, mediram-se os tempos de codificação de um vídeo utilizando diferentes números de *worker threads* do CPU e dividindo o trabalho em diferentes proporções entre o CPU e GPU.

A Figura 80 e a Figura 81 apresentam os resultados obtidos nos casos de teste executados nas duas máquinas.

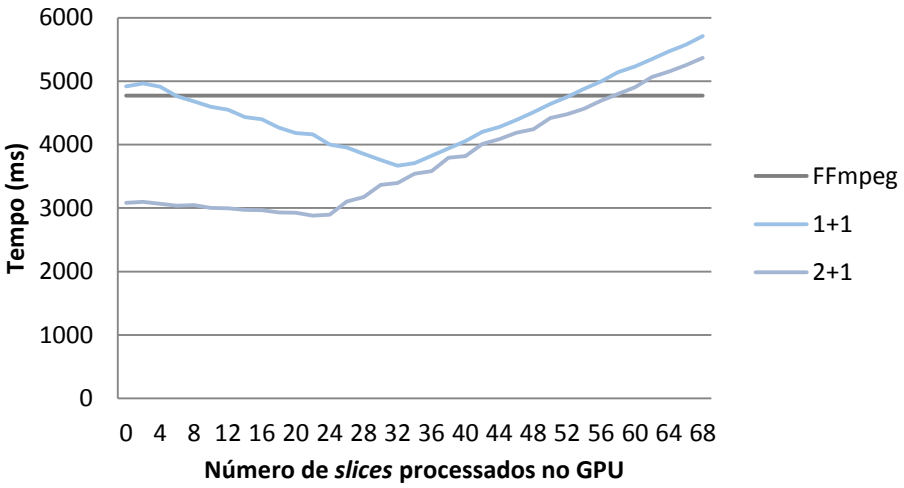


Figura 80: Tempo de codificação no Computador 1 em função do número de *slices* processados no GPU.

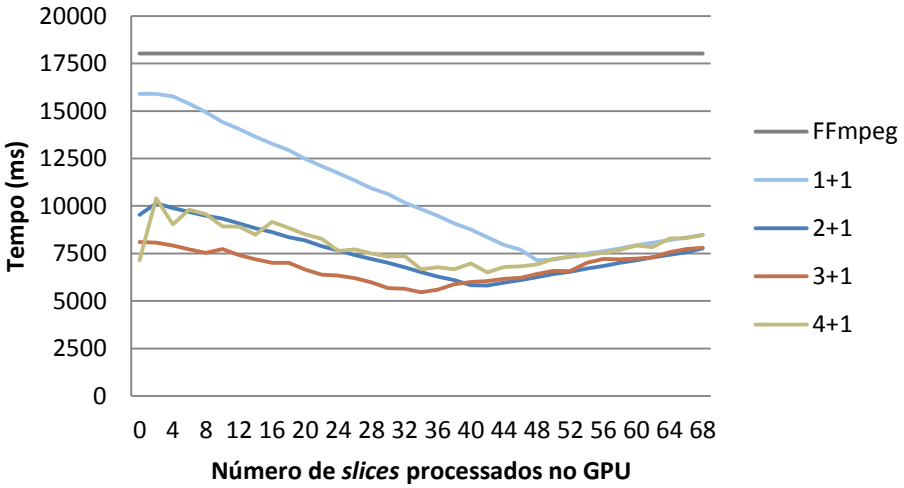


Figura 81: Tempo de codificação no Computador 2 em função do número de slices processados no GPU.

Resultados e Discussão

Analisando os resultados obtidos para cada uma das máquinas de testes:

- No Computador 1 obtiveram-se ganhos consideráveis de performance através do *offloading* de parte da *frame* para o GPU, mais acentuados nas medições em que apenas é utilizado um *thread* de processamento do CPU, para processar a DCT e quantização, as quais apresentam ganhos de performance até um máximo de 32 *slices* no GPU. A solução ótima para o PC1 é a solução de dois *threads* de processamento e um *thread* de controlo que permite um ganho de 200 milissegundos em relação à implementação com uso exclusivo do CPU com o *offloading* de 22 a 24 *slices*.

- À semelhança do Computador 1, no Computador 2 obtiveram-se ganhos de performance com a utilização do GPU em relação à implementação sem *offloading*. No entanto, a solução ótima é a solução de três *threads* de processamento mais o *thread* de controlo ao invés da solução de quatro *threads* mais o *thread* de controlo, que seria expectável dos resultados do Computador 1 e do capítulo anterior. Esta situação deve-se ao facto do *thread* de controlo da plataforma CUDA necessitar de tempo de processamento do CPU em momentos críticos e, uma vez que o HT procura maximizar o desempenho de múltiplos *threads* em detrimento da performance sequencial, causa ciclos de espera no *thread* de controlo, desperdiçando tempo de processamento do GPU. Neste computador atingiram-se ganhos de 1600 milissegundos para um *offloading* de 34 *slices* utilizando a solução anteriormente referida com 3 *threads* de processamento do CPU.

A análise dos resultados obtidos permite-nos concluir que para cada máquina, dependendo do número de *threads* de processamento, existe um número de *slices* ótimo que se pode enviar para o GPU a partir do qual ocorre uma perda de performance devido ao facto do GPU demorar mais tempo a processar o trabalho que lhe é distribuído do que os núcleos do CPU, atrasando o processo de codificação.

Os gráficos da Figura 82 e Figura 83 representam o *speedup* em função do número de *slices* alocados ao GPU para a solução de melhor desempenho para cada máquina.

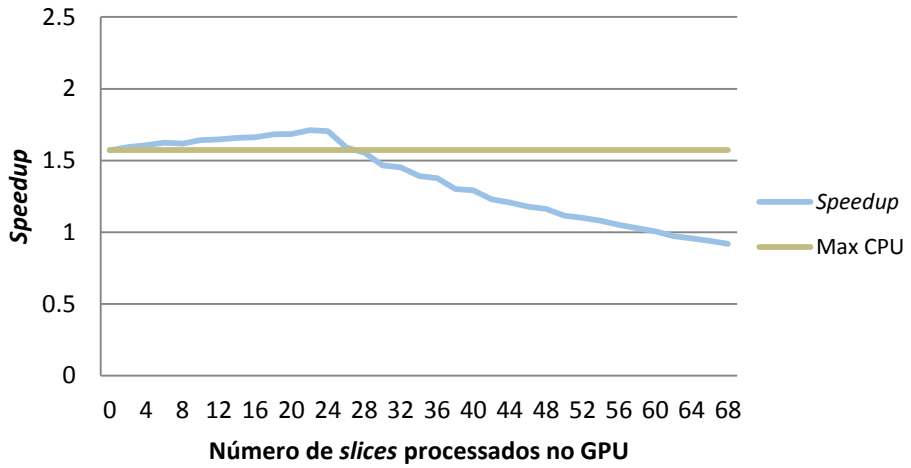


Figura 82:*Speedup* em função do número de *slices* alocados ao GPU na solução de dois *threads* de processamento no Computador 1.

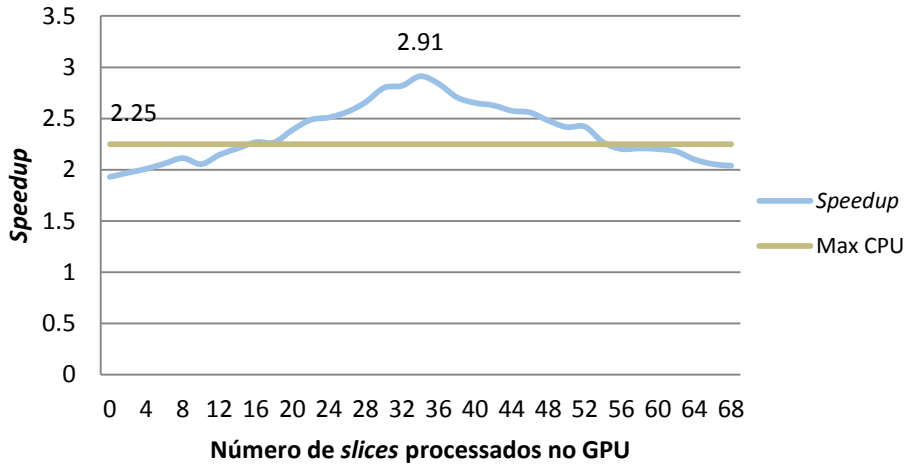


Figura 83: *Speedup* em função do número de *slices* alocados ao GPU na solução de três *threads* de processamento mais no Computador 2.

Os valores de *speedup* calculados reafirmam que em determinadas condições a divisão do trabalho entre o CPU e o GPU permite a obtenção de ganhos de performance em relação a uma abordagem que apenas faça uso do CPU. Os ganhos obtidos dependem das configurações do computador, sendo que no Computador 1, um computador com um CPU mais poderoso em relação ao GPU, os *speedups* obtidos com o GPU são 9% superiores à implementação sem recurso a computação Heterogénea. Por outro lado no Computador 2, CPU de baixa voltagem com um GPU equiparável ao do computador 1, a performance do algoritmo aumentou consideravelmente, sendo 2,91 vezes mais rápido que o algoritmo sequencial e 29% mais rápido que a implementação em paralelo sem recurso ao GPU.

6.5 Ferramenta de Otimização da Performance

Para avaliar a precisão da ferramenta de otimização desenvolvida, calcularam-se as distribuições de *slices* ótimas estando os resultados para o Computador 1 e Computador 2 representadas nos gráficos da Figura 84 e Figura 85 respetivamente.

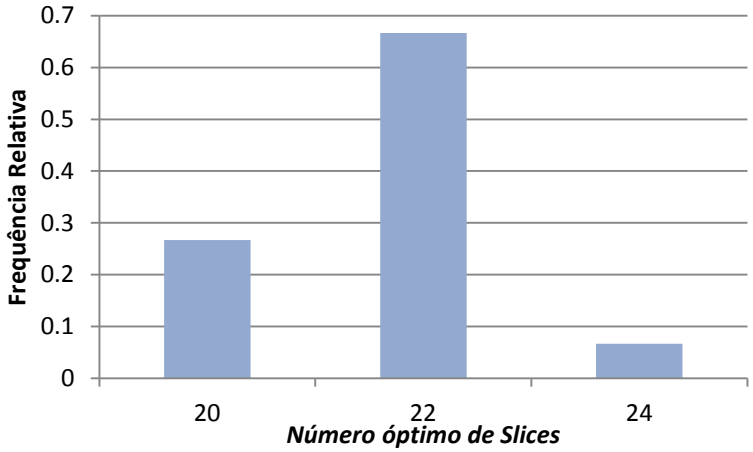


Figura 84: Resultados da ferramenta de otimização no Computador 1

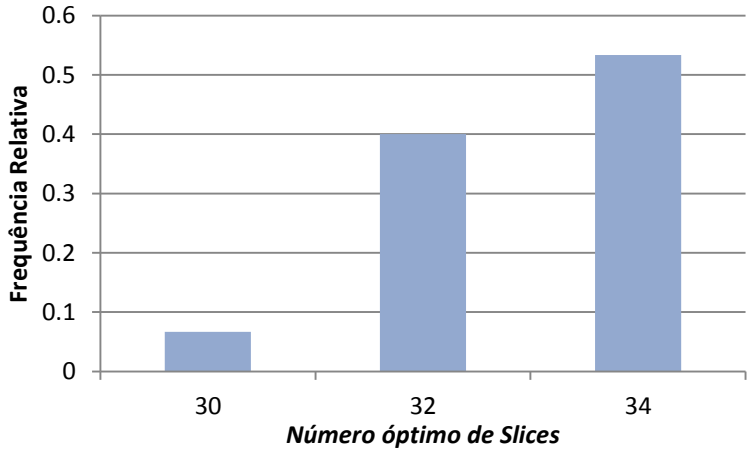


Figura 85: Resultados da ferramenta de otimização no Computador 2

Considerando os resultados obtidos, podemos concluir que não existe um resultado exato para cada computador, havendo um conjunto de resultados que são sugeridos em diferentes execuções da ferramenta. Tal deve-se ao facto do algoritmo implementado utilizar as medições dos tempos que o CPU e o GPU demoram a processar um *slice* para estimar a solução ótima, estando assim suscetível a oscilações de performance resultantes de outros programas ou o sistema operativo requererem recursos de processamento. Por outro lado, é importante voltar a

referir que na impossibilidade de se calcular o tempo exato da cópia de dados é realizada uma estimativa que pode afetar a exatidão dos resultados. É importante salientar que os resultados são precisos, isto é, apresentam desvios máximos de +/- 2 *slices* o que é consideravelmente baixo.

Analisando caso a caso, para o Computador 1 a distribuição obtida mais frequentemente foi de 22 *slices* para o GPU que coincide com o valor ótimo das medições do capítulo anterior ocorrendo desvios de +/-2 *slices* para 20 e 22 *slices* que ainda oferecem um ganho de performance em relação à implementação sem GPU. Para o Computador 2 a distribuição obtida mais frequentemente foi de 34 *slices* que equivale ao valor ótimo das medições anteriores obtendo-se com uma elevada frequência o valor de 32 *slices* que apesar de não ser ótimo, permite um ganho de 25% em relação à implementação em paralelo apenas com o CPU.

Tendo em conta a qualidade do *auto-tuning* podemos concluir que a ferramenta tem uma utilidade acrescida, podendo numa aplicação real ser executada da primeira vez que o programa é executado para caracterizar a máquina e parametrizar automaticamente a aplicação de modo a obter ganhos de performance através da computação heterogénea com o GPU de uma forma completamente transparente ao utilizador final.

6.6 Escalabilidade

6.6.1 Evolução da performance com o CPU

Atendendo ao estudo realizado pode concluir-se que a performance da solução implementada vai variar com as características do CPU. A nível da capacidade de processamento em paralelo, o algoritmo é capaz de escalar com os núcleos de processamento, físicos ou virtuais do HT, até ter um número de núcleos superior ao número de *slices* de uma *frame,* mais propriamente 68 núcleos para uma *frame* de vídeo *Full-HD*. No entanto, apesar de ser possível dividir o trabalho entre núcleos até cada núcleo processar apenas um *slice*, os dados do capítulo 4.2.2 Profiling indicam que os ganhos para este *codec* serão pouco significativos a partir de um número de núcleos superior a 32 devido ao peso das tarefas sequenciais no algoritmo. Relativamente à capacidade de processamento sequencial, a performance do algoritmo vai variar com a frequência de relógio e tamanho da *cache* uma vez que todo o algoritmo beneficia de um aumento da capacidade de processamento sequencial, desde as etapas sequenciais do algoritmo até às etapas de computação paralela em que cada *thread* será processado mais rapidamente. De ressalvar que caso seja utilizado HT combinado com o GPU o número de *threads* de processamento deve ser inferior ao número de unidades de processamento lógicas, devido ao facto do HT aumentar a latência do *thread* controlo do GPU para maximizar o rendimento do processador o que faz com que ocorra uma perda tempo de processamento do GPU tal como explicado anteriormente (Capítulo 6.4).

6.6.2 Evolução da performance com o GPU

Os GPUs são constituídos por diferentes unidades de processamento e níveis de memória. O algoritmo implementado processa um conjunto de *slices* simultaneamente criando um *thread* para cada bloco. Tendo em conta a estrutura da solução implementada e os resultados obtidos, para um GPU hipotético os principais fatores que vão permitir a escalabilidade são:

- Tamanho da Memória Partilhada – a solução implementada utiliza a memória partilhada para processar os macroblocos, copiando para esta memória de forma coalescida todos os macroblocos para serem processados. A memória partilhada é uma das maiores limitações à escalabilidade da solução num GPU restringindo o número de macroblocos de um *slice* que são alocados a um SMP e consequentemente o número de *threads* criados. Sendo o GPU um sistema otimizado para o processamento de múltiplos *threads*, quanto maior for a memória partilhada maior será o número de *threads* alocados a um SMP e consequentemente maior será a performance. Atualmente os GPUs NVIDIA com CC 2.0 ou superior têm 48KB de memória partilhada contrariamente aos 16KB dos GPUs utilizados;

- Unidades de Processamento – cada SMP contém um módulo de memória partilhada e um conjunto de *Stream Processors*. A nível de escalabilidade um maior número de SMPs vai permitir alocar um maior número de macroblocos em simultâneo num *kernel* ao GPU e consequentemente aumentar a performance do processamento. Um SP ou CUDA *Core* é a unidade que processa um *thread. Q*uanto maior for o número de SPs constituintes de um SMP mais *threads* podem ser processados simultaneamente e consequentemente maior vai ser o desempenho do algoritmo. Os GPUs mais recentes de alta performance contêm para cima de 32 SMP com mais de 1500 SPs o que permite enormes ganhos de performance;

- Velocidade do BUS PCI-E – a velocidade do BUS não deve limitar a performance do GPU garantindo que as cópias de memória são mais rápidas que a execução do *kernel* por forma a permitir a sobreposição das tarefas e assim eliminar o custo das transferências.

Posto isto, podemos concluir que para um GPU desenvolvido para GPGPU do "Estado da Arte" como o NVIDIA Tesla K20X de CC 3.5 com Paralelismo Dinâmico e 15 *Stream MultiProcessor Extreme* (SMX) com 192 SP, os ganhos de performance seriam muito superiores [52]. Em comparação com a solução atual, o aumento de memória partilhada permite o processamento de 4 vezes mais macroblocos por SMP, para um total de 80 macroblocos o que equivale a 480 *threads* por SMP a serem divididos por 192 SP, em contraste com os 120 *threads* a serem divididos por 8 SP, que para além de serem em maior número também são mais

rápidos. Ponderando todos os números, este GPU é capaz de processar simultaneamente 480 macroblocos, 180 vezes mais que os GPUs utilizados.

Capitulo 7

Conclusões e Trabalho Futuro

No processamento de vídeo o desempenho do processo de codificação tem uma importância critica. O MPEG-2 IMX é um algoritmo de codificação de vídeo muito utilizado no mundo da televisão em que todas as *frames* são codificadas independentemente com um tamanho fixo permitindo a edição simples e rápida de um stream de vídeo. Num cenário real é necessário que o processo de codificação seja suficientemente rápido para permitir a codificação de vídeo em tempo real para este formato.

O desenvolvimento de GPUs mais poderosos e com capacidades de processamento cada vez mais flexíveis possibilita a distribuição de tarefas computacionais entre o CPU e o GPU de forma a acelerar algumas tarefas de processamento mais intensivas. Os algoritmos de codificação de vídeo não são exceção sendo constituídos por etapas de processamento que podem ser processados em arquiteturas heterogéneas.

Neste projeto especificaram-se duas soluções com o objetivo de aumentar a performance do algoritmo de codificação de vídeo MPEG-2 IMX utilizando computação paralela com o CPU e computação heterogénea com o CPU e o GPU. As soluções foram implementadas utilizando OpenMP e CUDA. Os resultados obtidos foram bons tendo-se obtido para a solução que utiliza apenas o CPU um *speedup* próximo do *speedup* teórico de Amdahl e um aumento de performance face a esta solução de 10 a 30% utilizando processamento combinado. Para tornar esta solução acessível a um utilizador final, foi desenvolvida uma ferramenta de parametrização que estima a distribuição ótima do processamento entre as diferentes unidades.

Este trabalho permitiu-nos verificar que, apesar da maior parte dos sistemas industriais desta área não conterem um GPU, a computação heterogénea tem um enorme potencial para ser aplicada em tarefas de codificação de vídeo por forma a atingir o objetivo da codificação em tempo real e justificar a sua implementação na indústria televisiva.

7.1 Trabalho Realizado e Satisfação dos Objetivos

Os objetivos inicialmente estabelecidos foram atingidos. Durante o projeto foi possível definir uma estratégia de paralelismo e divisão das tarefas entre as diferentes unidades de processamento, que foi implementada com sucesso tanto para plataformas de computação paralela com múltiplos núcleos como para plataformas heterogéneas com CPU e GPU. Os resultados obtidos para estas implementações foram promissores tendo-se obtido um ganho de performance para ambas as soluções. A implementação para plataformas heterogéneas foi a solução que atingiu performances mais elevadas e demonstrou que o GPU pode ser utilizado na codificação de vídeo para reduzir a carga do CPU e consequentemente acelerar o processo de codificação.

Além dos objetivos propostos implementou-se uma ferramenta para calcular a distribuição ótima do trabalho entre os dispositivos. Os resultados desta ferramenta revelaram-se precisos o suficiente para garantir ganhos performance utilizando a distribuição calculada pela ferramenta e justificar a sua utilização por um possível utilizador final que pretenda obter ganhos de performance "*out of the box*" sem ter de configurar o sistema.

7.2 Trabalho Futuro

Este tema poderá ser futuramente explorado em maior detalhe com o estudo das aplicações deste paradigma em algoritmos de codificação de vídeo de formatos com predição de movimento entre *frames* como o MPEG-2 Long GOP ou o MPEG HEVC.

Por outro lado, atendendo ao lançamento recente de novas plataformas de coprocessadores x86, seria pertinente o estudo da performance de uma implementação heterogénea utilizando coprocessadores x86 como o Intel Xeon Phi ou o Tilera e a comparação da performance desta solução com uma implementação heterogénea utilizando GPUs destinados a GPGPU intensivo como o NVIDIA Tesla.

Referências

[1] A. B. Barney, “Introduction to Parallel Computing,” *Lawrence Livermore National Laboratory*. 2012.

[2] B. B. Schauer, “Multicore Processors – A Necessity,” no. September, pp. 1–14, 2008.

[3] W. Knight, “Two heads are better than one [dual-core processors],” *IEE Review*, vol. 51, no. 9. pp. 32–35, 2005.

[4] Randy Hyde, *The Art of Assembly Language Programming*, 2nd ed. No Starch Press, 2010.

[5] Intel, “Intel Hyper-Threading Technology Technical User’s Guide,” no. January. 2003.

[6] A. Rege, “An Introduction to Modern GPU Architecture,” vol. 34, no. 1, Jan. 2008.

[7] A. Plyer, “GPU programming,” 2012. [Online]. Available: http://www.aurelien.plyer.fr/my-resume/phd/gpu-programming/#gpuHadware. [Accessed: 30-Jan-2013].

[8] D. Luebke, “GPU Architecture: Implications & Trends,” *Supercomputing 08: Proceedings of the 2008 ACM/ ...*, 2008.

[9] O. Rosenberg, “Introduction to GPU Architecture,” 2011.

[10] P. (NVIDIA) Micikevicius, “Local Memory and Register Spilling,” 2011.

[11] M. Zahran, “Lecture 6: CUDA Memories,” *Graphics Processing Units (GPUs): Architecture and Programming*, 2012.

[12] P. Micikevicius, “Optimizing cuda,” *SC08: High Performance Computing With CUDA*, 2008.

Referências

[13] V. Lee, C. Kim, J. Chhugani, and M. Deisher, "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU," ... *SIGARCH Computer ...*, pp. 451–460, 2010.

[14] H. Processing and U. Discovery, "The Intel ® Xeon Phi ™ Coprocessor 5110P The Intel Xeon Phi Coprocessor," 2012.

[15] Tilera, "TILEncore Card – Product Brief." 2011.

[16] Intel, "Avoiding and Identifying False Sharing Among Threads," 2011.

[17] R. (Microsoft) Lämmel, "Google's MapReduce programming model — Revisited," *Science of Computer Programming*, vol. 70, no. 1, pp. 1–30, Jan. 2008.

[18] NVIDIA, "What is GPU computing?," 2012. [Online]. Available: http://www.nvidia.com/object/what-is-gpu-computing.html. [Accessed: 05-Jan-2013].

[19] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," 2012.

[20] M. (NVIDIA) Harris, "GPGPU: General-purpose computation on GPUs," *SIGGRAPH 2005 GPGPU COURSE*, 2005.

[21] G. Bilotta, E. Rustico, and A. Hérault, "General-purpose programming on GPU: From GPU to GPGPU," 2012. [Online]. Available: http://www.dmi.unict.it/~bilotta/gpgpu/notes/05-gpgpu-history.html. [Accessed: 31-Jan-2013].

[22] B. Barney, "POSIX threads programming," *Lawrence Livermore National Laboratory*. 2009.

[23] Microsoft, "Processes and Threads (Windows)," 2012. [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/ms684841.aspx. [Accessed: 10-Jan-2013].

[24] OpenMP, "Frequently Asked Questions on OpenMP." [Online]. Available: http://openmp.org/openmp-faq.html. [Accessed: 13-Jan-2013].

[25] Intel, "Intel Threading Building Blocks - Intel Developer Zone," 2012. [Online]. Available: http://software.intel.com/en-us/intel-tbb. [Accessed: 14-Jan-2013].

[26] M. V. (Intel), "Intel Threading Building Blocks, OpenMP, or native threads?," 2011. [Online]. Available: http://software.intel.com/en-us/intel-threading-building-blocks-openmp-or-native-threads. [Accessed: 14-Jan-2013].

[27] Khronos, "OpenCL - The open standard for parallel programming of heterogeneous systems." [Online]. Available: http://www.khronos.org/opencl/. [Accessed: 18-Jan-2013].

Referências

[28] O. (AMD) Rosenberg, “OpenCL Overview,” 2011. [Online]. Available: http://www.khronos.org/assets/uploads/developers/library/overview/opencl-overview.pdf. [Accessed: 12-Jan-2013].

[29] M. Ebersole, “What is CUDA?,” 2012. [Online]. Available: http://blogs.nvidia.com/2012/09/what-is-cuda-2/. [Accessed: 25-Jan-2013].

[30] Tosaka, “Processing flow on CUDA.” 2008.

[31] Microsoft, “C ++ Accelerated Massive Parallelism (C ++ AMP),” 2012. [Online]. Available: http://blogs.msdn.com/cfs-file.ashx/__key/communityserver-components-postattachments/00-10-29-86-29/cppAMPv6_2D00_gen.pdf. [Accessed: 03-Jun-2013].

[32] NVIDIA, “OpenACC | NVIDIA Developer Zone:” [Online]. Available: https://developer.nvidia.com/openacc. [Accessed: 18-Jan-2013].

[33] “The OpenACC ™ Application Programming Interface,” 2011.

[34] S. Dong, Z. Wang, R. Wang, Z. Wang, and W. Gao, “A two stage parallel encoder scheme for real time video encoder,” in *2012 IEEE International Conference on Signal Processing, Communication and Computing (ICSPCC 2012)*, 2012, pp. 280–285.

[35] N.-M. Cheung, X. Fan, O. Au, and M.-C. Kung, “Video Coding on Multicore Graphics Processors,” *IEEE Signal Processing Magazine*, vol. 27, no. 2, pp. 79–89, Mar. 2010.

[36] R. Sachdeva and K. Saha, “Parallel MPEG-2 Video Encoder,” in *2011 IEEE International Conference on Consumer Electronics (ICCE)*, 2011, pp. 159–160.

[37] W. Flohr, “Implementation of an MPEG Codec on the Tilera TM 64 Processor,” *Research Project. St. Louis: Washington University*, pp. 1–11, 2008.

[38] Intel, “Intel Media SDK 2013.” [Online]. Available: http://software.intel.com/en-us/vcsource/tools/media-sdk. [Accessed: 18-Jan-2013].

[39] T. (Intel) Craver, “Performance Interactions of OpenCL* Code and Intel Quick Sync Video on Intel HD Graphics 4000,” 2012. [Online]. Available: http://software.intel.com/en-us/articles/performance-interactions-of-opencl-code-and-intel-quick-sync-video-on-intel-hd-graphics. [Accessed: 10-Jan-2013].

[40] NVIDIA, “NVIDIA VIDEO CODEC SDK | NVIDIA Developer Zone,” 2013. [Online]. Available: https://developer.nvidia.com/nvidia-video-codec-sdk. [Accessed: 04-Jun-2013].

[41] AMD, “Accelerated Parallel Processing (APP) SDK > Samples & Demos,” 2013. [Online]. Available: http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/samples-demos/. [Accessed: 04-Jun-2013].

[42] MainConcept, “GPU Acceleration : MainConcept,” 2013. [Online]. Available: http://www.mainconcept.com/eu/products/sdks/gpu-acceleration.html. [Accessed: 20-May-2013].

Referências

[43] MPEG, “MPEG.ORG - MPEG Home,” 2012. [Online]. Available: http://www.mpeg.org/. [Accessed: 01-Jan-2013].

[44] V. U. of H. K. Lo, “A Beginners Guide for MPEG-2 Standard,” 2012. [Online]. Available: http://www.fh-friedberg.de/fachbereiche/e2/telekom-labor/zinke/mk/mpeg2beg/beginnzi.htm. [Accessed: 10-Jan-2013].

[45] Apple, “Final Cut Pro 7 Professional Formats and Workflows: About IMX,” 2009. [Online]. Available: http://documentation.apple.com/en/finalcutpro/professionalformatsandworkflows/index.html#chapter=4&section=1. [Accessed: 10-Jan-2013].

[46] J. Watkinson, *The MPEG Handbook*, 1st ed. Woburn, MA, U.S.A.: Focal Press, 2001, p. 4.

[47] Vijay, “MPEG-2 Video compression basics.” 2009.

[48] C. Poynton, “Chroma subsampling notation,” *Retrieved June*, pp. 3–5, 2002.

[49] P. Tudor, “MPEG-2 video compression,” *Electronics & communication engineering journal*, no. November 1993, 1995.

[50] J.-L. Wu, “MPEG-1 Video Codec,” 1999. [Online]. Available: http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/mpeg1/. [Accessed: 01-Feb-2013].

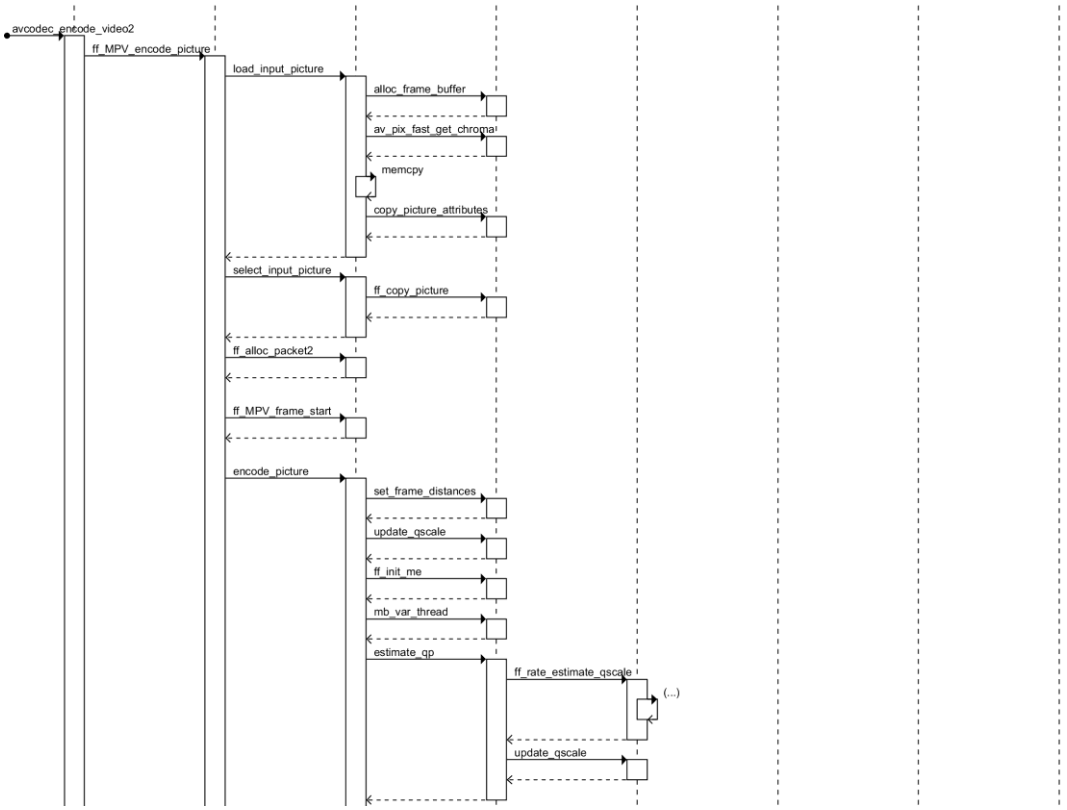
[51] C. Loeffler, A. Ligtenberg, and G. S. Moschytz, “Practical fast 1-D DCT algorithms with 11 multiplications,” in *International Conference on Acoustics, Speech, and Signal Processing,*, 1989, pp. 988–991.

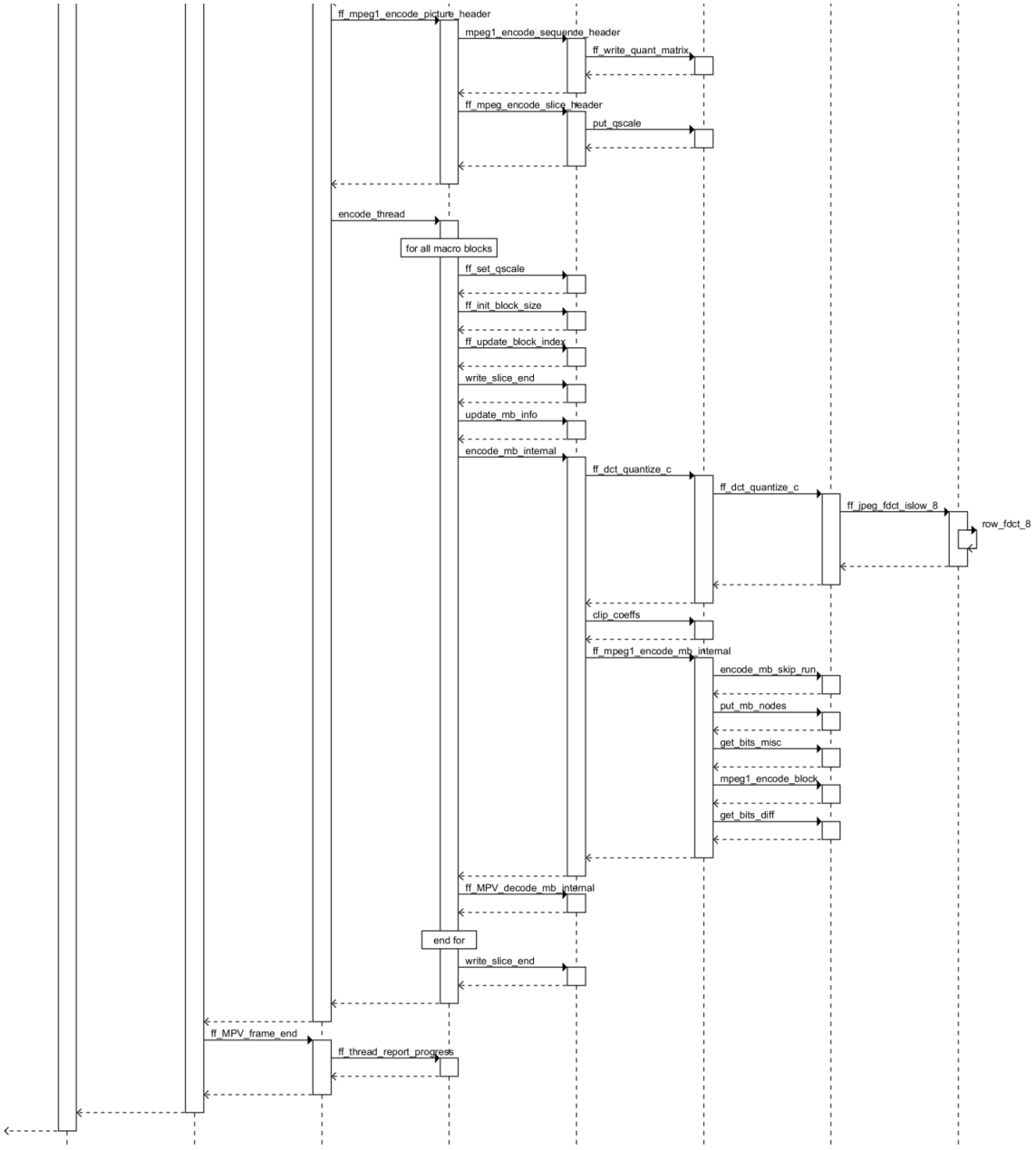
[52] NVIDIA, “Tesla Kepler GPU Accelerators,” 2013. [Online]. Available: http://www.nvidia.com/content/tesla/pdf/Tesla-KSeries-Overview-LR.pdf. [Accessed: 05-May-2013].

Anexo A

FFmpeg

9.1 Diagrama de chamada de funções da função de codificação do codec MPEG-2 do FFmpeg




ff_mpeg1_encode_picture_header
mpeg1_encode_sequende_header
ff_write_quant_matrix
ff_mpeg_encode_slice_header
put_qscale
encode_thread
for all macro blocks
ff_set_qscale
ff_init_block_size
ff_update_block_index
write_slice_end
update_mb_info
encode_mb_internal
ff_dct_quantize_c
ff_dct_quantize_c
ff_jpeg_fdct_islow_8
row_fdct_8
clip_coeffs
ff_mpeg1_encode_mb_internal
encode_mb_skip_run
put_mb_nodes
get_bits_misc
mpeg1_encode_block
get_bits_diff
ff_MPV_decode_mb_internal
end for
write_slice_end
ff_MPV_frame_end
ff_thread_report_progress