FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Abordagem de computação heterogénea para reamostragem e redimensionamento de vídeo de alto desempenho

José Pedro Soares João Pereira

VERSÃO DE TRABALHO

Mestrado Integrado em Engenharia Informática e Computação

Orientador: Jorge Manuel Gomes Barbosa

8 de Junho de 2018

Abordagem de computação heterogénea para reamostragem e redimensionamento de vídeo de alto desempenho

José Pedro Soares João Pereira

Mestrado Integrado em Engenharia Informática e Computação

8 de Junho de 2018

Resumo

A crescente popularidade dos canais de comunicação de televisão e plataformas de streaming motiva a publicação de novo conteúdo multimédia. De modo a alcançar e a agradar o maior número de consumidores, este conteúdo deve ser criado o mais rapidamente possível com a melhor qualidade disponível. Os conteúdos de multimédia de elevada qualidade são obtidos através da aplicação de várias operações de pós-produção. Visto que as operações de produção são aplicadas ao nível de cada pixel de uma imagem e imagens de alta qualidade são constituídas por uma quantidade elevada de pixeis, estas operações reclamam uma capacidade de recursos computacionais diretamente proporcionais ao tamanho das imagens a serem processadas.

Atualmente, devido ao progresso da tecnologia contida nas placas de processamento gráfico existe uma maior capacidade de processamento em relação à capacidade dos processadores, tipicamente, aliada a uma utilização mais eficiente de energia. A abordagem que combina as capacidades de processamento de um processador e de uma placa gráfica de uma máquina, a abordagem de computação heterogénea, é ideal para o desenvolvimento de software de alto desempenho.

O objetivo deste trabalho consiste em analisar e implementar uma abordagem de computação heterogénea utilizando as capacidades das placas gráficas para uma das fases de pós-produção, a fase de reamostragem e redimensionamento de vídeo sem compressão em tempo real. Especificamente, este trabalho aborda os detalhes e peculiaridades da implementação de uma abordagem que alia as capacidades de processamento do CPU e GPU durante a fase de pós-produção referida de vídeo sem codificação, assegurando que o tempo de processamento de um vídeo é menor ou igual ao seu tempo de captura. A solução implementada utiliza a ferramenta OpenMP e a plataforma de desenvolvimento CUDA para paralelização e tirar partido das capacidades de processamento dos processadores e das placas gráficas.

Os resultados obtidos pela ferramenta desenvolvida mostram que a implementação da abordagem referida permitiu ganhos de desempenho entre 48% e 57% em relação à solução mais utilizada na área de multimédia para processamento e aplicação de operações de pós-produção de vídeo, a ferramenta FFMpeg.

A partir dos resultados obtidos por este trabalho é possível concluir-se que as placas gráficas são uma ferramenta com grandes capacidades computacionais que podem ser utilizadas para extrair paralelismo e potenciar o desempenho da aplicação de operações de pós-produção de vídeo.

Abstract

The increasing popularity of communication channels such as television broadcasters and streaming platforms motivates the issue of new multimedia content. In order to reach and please the most customers, this content should be created as fast as possible with the best quality available. High quality multimedia content is obtained through the application of various post-production operations. Since post-production operations are applied at an image's pixels level and high quality images are constituted by an high quantity of pixels, these operations require computational resources directly proportional to the their size.

Nowadays, due to the advance of the technology contained in the graphics processing units there is a greater processing capability in relation to the processing capability of processors, typically, conjugated with a better power usage efficiency. The approach that combines the processing capabilities of a processor and a graphics card of a machine, the heterogeneous computing approach, is ideal to the development of high performance software.

The goal of this work is to analyze and implement an heterogeneous computing approach using the capabilities of graphics cards during one of the post-production phases, the resample and resizing phase of uncompressed video in real time. Specifically, this work tackles the details and implementation peculiarities of an approach that allies the computing capabilities of CPU and GPU during the stated post-production phase of video without encoding, guaranteeing that the processing time of a video is less or equal than its capturing time. The implemented solution uses OpenMP and the development framework CUDA to parallelize and take advantage of the computing capabilities of processors and graphics cards.

The results of the developed tool show that the implementation of the stated approach allowed a performance gain between 48% and 57% in comparison to the most used solution in multimedia area to process and application of video post-production operations, the FFMpeg tool.

From the obtained results it is possible to conclude that graphics cards are a tool with great computing capabilities that can be used to extract parallelism and potentiate the performance of the application of video post-production operations.

Conteúdo

CONTEÚDO

Lista de Figuras

LISTA DE FIGURAS

Lista de Tabelas

LISTA DE TABELAS

Abreviaturas e Símbolos

ALU	Arithmetic Logic Unit
API	Application Programming Interface
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
FHD	Full High Definition
GPGPU	General Purpose computing on Graphics Processing Units
GPU	Graphics Processing Unit
MPI	Message Passing Interface
UHD	Ultra High Definition
RAM	Random Access Memory
SM	Steaming Multiprocessors
SP	Steam Processors

Capítulo 1

Introdução

A sociedade do século vinte e um tem sofrido uma drástica alteração cultural, económica e social devido à utilização das novas tecnologias dos dispositivos eletrónicos como computadores pessoais, smartphones, e tablets, que aliada à utilização da internet permite que qualquer indivíduo possa aceder, visualizar, modificar, armazenar e partilhar conteúdos de multimédia com um alcance virtualmente global. Por esta razão as indústrias de jornalismo, educação, entretenimento, a indústria editorial e de música têm sofrido significativas transformações para acompanhar a tendência, levando à utilização de meios de comunicação diferentes dos considerados convencionais antigamente de modo a atingir os seus objetivos como a utilização de plataformas de streaming e canais de televisão. Todo o mercado de criação de conteúdos multimédia, constituído maioritariamente pelas indústrias referidas, está direcionado para a publicação de novo conteúdo multimédia o mais rapidamente possível com a melhor qualidade disponível culminando numa vantagem sobre outros negócios do mercado.

Desde a invenção dos primeiros computadores, o poder de processamento e as capacidades de armazenamento de informação têm crescido exponencialmente. Com a evolução da tecnologia surgiram diferentes abordagens para realizar os processos de pós-produção de vídeo necessários para gerar conteúdos de multimédia de elevada qualidade. A computação paralela e computação heterogénea são duas das abordagens que podem ser tidas em conta para a aplicação destes processos, pois estas abordagens permitem a execução concorrente dos mecanismos de um sistema utilizando as capacidades de processamento dos processadores e das placas de processamento gráfico de uma máquina de forma a acelerar a aplicação das operações de pós-produção de vídeo permitindo os melhores níveis de qualidade dos resultados em tempo real.

~~1.1 Contexto~~

O tema desta dissertação foi proposto pela MOG Technologies, especializada no desenvolvimento de novas plataformas tecnológicas para colocar novos produtos, sistemas e soluções ino-

vadoras que automatizam os processos de trabalho dos operadores de conteúdos de multimédia profissional, nomeadamente, produtoras de vídeo e estações de televisão. Um destes processos de trabalho é o processo de pós-produção, automatizado através do desenvolvimento de soluções de ferramentas de Ingest que manipulam e transportam um vídeo desde um ponto de origem até a um ponto de destino da cadeia de produção.

A evolução das áreas de computação paralela e computação heterogénea permitem a aplicação destas tecnologias na área das soluções de vídeo de forma a obter um melhor desempenho dos sistemas desenvolvidos e atender às restrições, cada vez mais rigorosas, impostas pela indústria.

1.2 Motivação

O processo de reamostragem e redimensionamento de imagem é um dos processos necessários a serem realizados durante a fase de pós-produção de um vídeo. Cada uma das imagens de um vídeo sem codificação, sem qualquer tipo de compressão, em formato *raw*, são constituídas por um elevado número de pixeis. Uma imagem na resolução *Full High Definition*, ou *FHD*, com uma dimensão de 1920 por 1080 pixeis, é constituída, aproximadamente, por dois milhões de pixeis. Atualmente, são cada vez mais utilizadas resoluções superiores ao *FHD*, como é o caso do *Ultra High Definition*, ou *UHD*, designadamente as resoluções *4K* e *8K*, aproximadamente constituídas, respetivamente, por nove milhões e por trinta e cinco milhões de pixeis.

Atendendo a que o processo de pós-produção mencionado opera ao nível de cada pixel que constitui uma imagem, a capacidade de processamento necessária para o realizar é diretamente proporcional à resolução da imagem. De forma a que os conteúdos de multimédia estejam disponíveis assim que possível, a aplicação dos processos de pós-produção deve ser realizada, idealmente, em tempo real. Isto é, o processamento de uma imagem de um vídeo deve ser realizado antes da captura da imagem seguinte. Por estes motivos, a aplicação dos processos de pós-produção, respeitando as restrições da indústria, torna-se cada vez mais difícil de ser efetuada sem que haja uma nova análise e uma nova implementação das soluções utilizadas até ao momento para o efeito.

1.3 Objetivos

O objetivo deste projeto é a análise das técnicas utilizadas durante a realização da fase de pós-produção de reamostragem e redimensionamento de vídeo, a implementação e descrição de uma solução para a aplicação do processo referido recorrendo a uma abordagem de computação heterogénea. Enumeram-se os seguintes objetivos para este trabalho:

- Analisar a arquitetura de unidades de processamento;
- Explorar as diferentes técnicas e ferramentas de paralelização de software;
- Estudar e descrever os diferentes algoritmos de reamostragem e redimensionamento de imagem;

- Descrever os múltiplos formatos e modelos de cor utilizados na representação de vídeo digital;
- Averiguar as diferentes soluções já existentes para o processo;
- Explicar e pormenorizar detalhes da implementação da solução realizada;
- ~~Apresentar e discutir resultados da solução implementada.~~

1.4 Estrutura da Dissertação

Este relatório contém cinco capítulos, incluindo este capítulo introdutório, contextualizando o problema. No capítulo seguinte, capítulo 2 intitulado de "Computação Paralela e Computação Heterogénea", são expostos os principais detalhes das arquiteturas dos processadores utilizados nos sistemas que abordam as áreas de computação paralela e computação heterogénea, uma descrição dos principais conceitos da área de sistemas paralelos e a análise das ferramentas e plataformas de desenvolvimento utilizadas para implementar as abordagens referidas.

No seguinte capítulo, capítulo **??** intitulado de "Processamento de Vídeo", serão analisadas as técnicas e os algoritmos respetivamente utilizados nos processos de reamostragem e redimensionamento de imagem, e uma descrição dos mesmos.

Por último, o capítulo **??** de conclusão aborda as principais conclusões alcançadas no desenvolvimento deste trabalho, a solução propostas e os resultados esperados, e as tarefas a serem realizadas durante a próxima fase deste projeto, assim como um plano de trabalho para as executar.

Capítulo 2

Computação Paralela

Neste capítulo é analisado o estado da arte referente à área de computação paralela e computação heterogénea. Nesta secção são analisadas as componente que constituem as unidades de processamento central e as unidades de processamento gráfico dos computadores atuais, estabelecendo uma comparação entre ambas. Como conclusão deste capítulo são apresentadas as diferentes ferramentas e plataformas de desenvolvimento utilizadas atualmente na implementação de sistemas de computação paralela e computação heterogénea.

2.1 Unidades de Processamento

Os computadores atuais seguem a arquitetura desenvolvida por John von Neuman, designada de máquina de von Neuman. Segundo esta arquitetura, uma máquina é dividida em três componentes principais como apresentado na figura 2.1, sendo estas: a unidade de processamento central, ou CPU, a memória de acesso aleatório, ou RAM, e a interface de entrada e saída.

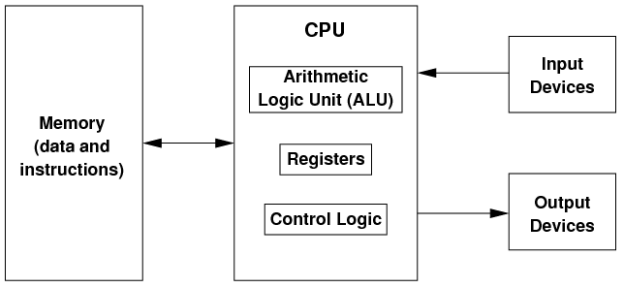

Figura 2.1: Diagrama de arquitetura de von Neuman [1].

A unidade de processamento central é a componente de um computador responsável pela execução de instruções de um determinado programa. O CPU é constituído pela unidade de lógica e aritmética, ou ALU, os registos do processador e uma unidade de controlo. A ALU é um circuito responsável pela execução de operações aritméticas e lógicas. Os registos do processador estão

encarregues de fornecer operandos à ALU e armazenar os respetivos resultados das operações. A unidade de controlo coordena a recuperação dos dados armazenados na memória e a execução das operações realizadas pela ALU, registos do processador e outras componentes. A RAM armazena as instruções a serem executadas e os dados necessário para as realizarem. A interface de entrada e saída faz a ligação entre a RAM e o CPU podendo também criar ligação com outro tipo de periféricos ou hardware. Os sistemas de computação paralela são constituídos por várias unidades que seguem a arquitetura de von Neuman o que permite executar em simultâneo múltiplas operações e conter diferentes dados armazenados.

2.1.1 Evolução dos CPUs

A lei de Moore, criada por Gordon Moore, co-fundador da Intel, é a observação que o número de transístores de um circuito integrado duplica a cada dois anos. Os transístores permitem criar circuitos complexos. Quanto maior for o número de transístores de um circuito integrado, como por exemplo um processador, maior será o número dos circuitos complexos destinados a realizarem as operações aritméticas e lógicas, e por consequência maior será a capacidade de execução de instruções de um processador [2]. Por esta razão, a capacidade de processamento de um CPU está intrinsecamente relacionada com a lei de Moore.

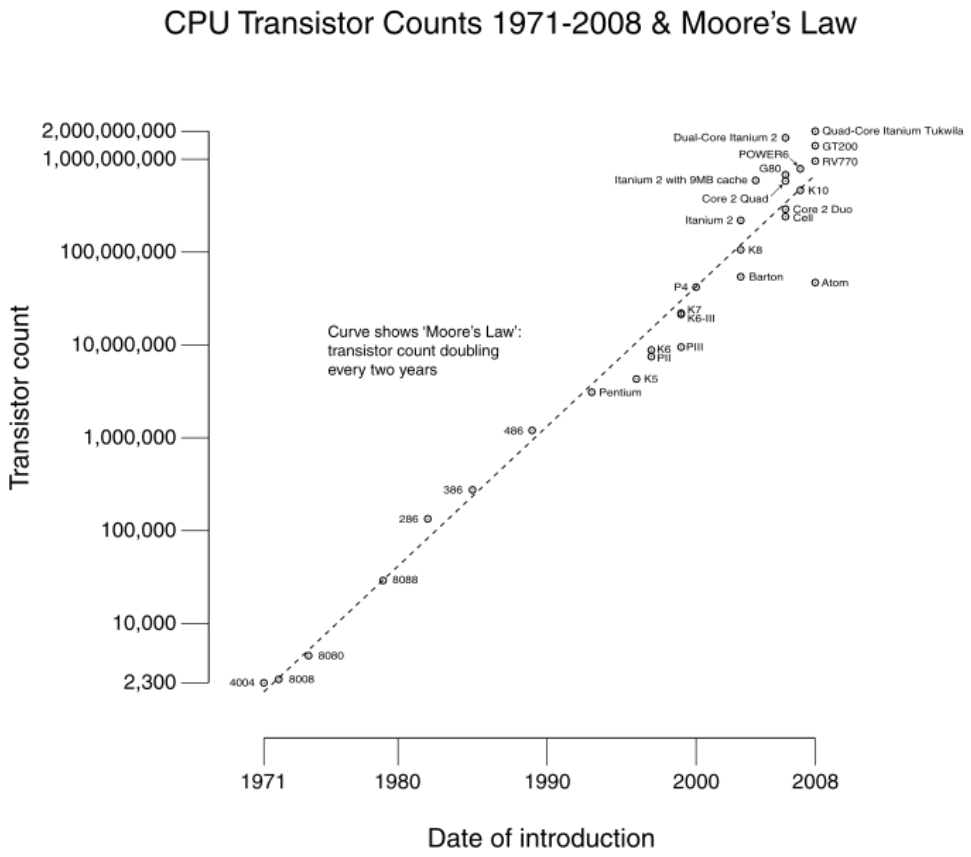

Figura 2.2: Relação entre a evolução dos processadores e a lei de Moore [2].

A utilização de um maior número de transístores implica um maior consumo de energia por parte de um processador. Até 2004, a abordagem utilizada para aumentar a capacidade de processamento de um CPU era a integração de um maior número de transístores no circuito, de modo a aumentar a frequência de relógio que dita a cadência de execução das instruções a serem realizadas. O aumento da frequência de um processador causa um maior consumo de energia devido à dissipação de calor causada. A cada aumento de 400 MHz na frequência de execução de instruções de um CPU existe um incremento de 60% da energia consumida pelo processador [7]. Esta abordagem tornou-se inviável visto que o custo da energia consumida não compensa o ganho de performance do processador.

Por esta razão outro tipo de abordagem foi tida em conta, nomeadamente, a arquitetura *multi-core*. O aumento do número de núcleos de processamento de um processador permite ter uma frequência de relógio mais lenta e aumentar o número de instruções executadas [8]. Como exemplo considera-se uma certa frequência de relógio de um processador, um núcleo de processamento a 20% dessa frequência economiza 50% da energia necessária apenas sacrificando 13% da sua performance. Ao dividir o trabalho a ser realizado por dois núcleos de processamento a 80% de frequência de relógio, é possível atingir 43% de ganho de performance pelo mesmo custo energético.

2.1.2 Hierarquia de Memória - CPU

Na arquitetura de um computador, as componentes destinadas ao armazenamento de dados são ordenadas hierarquicamente com base no seu tempo de resposta [9]. A hierarquia de memória afeta o desempenho de um computador de tal modo que a implementação de um programa de alto desempenho deve considerar as capacidades e limitações de cada uma das componentes de memória utilizadas. As componentes de memória presentes num computador atual são apresentadas, por ordem crescente de tempo de resposta e capacidade de armazenamento:

- Registos do processador - Componente da arquitetura de um computador de acesso de memória mais rápido, cerca de um ciclo de relógio do CPU, mas de reduzida capacidade de armazenamento. Esta componente de memória é normalmente utilizada para armazenar temporariamente resultados de operações intermédias e valores de variáveis utilizadas frequentemente num processo;

- Memória cache - É a componente de memória mais próxima do processador que armazena fragmentos da memória principal. A maioria dos computadores têm diferentes níveis deste tipo de memória que diferem segundo a proximidade ao núcleo do processador, capacidade de armazenamento e tempo de resposta de acessos a memória. Este tipo de memória tem um sistema próprio de gestão que prioriza o armazenamento de dados utilizados frequentemente e liberta dados armazenados obsoletos no contexto de um processo;

- Memória principal - Componente mais utilizada para conservação de dados devido à sua abundante capacidade de armazenamento. Contudo, a memória principal é a componente

de memória com maior latência de resposta de acessos a memória das componentes apresentadas;

- Memória Secundária - Este tipo de memória não está imediatamente disponível a ser utilizada pelo computador visto que necessita de interação humana para o efeito. O acesso a este tipo de memória não pode ser processado diretamente pelo CPU. Para aceder a esta memória é necessário copiar os dados armazenados na componente para memória principal. Disquetes, discos compactos e memórias flash USB são dispositivos considerados memórias secundárias.

2.2 Taxonomia de Flynn

Atualmente, as arquiteturas dos computadores evoluíram para máquinas paralelas devido ao limite prático da frequência de relógio de um CPU em relação ao seu gasto de energia. A taxonomia de Flynn é uma classificação das arquiteturas dos computadores baseada no modo como estas unidades de processamento funcionam, como a sua memória é organizada e como são realizadas as comunicações do processador [10]:

- SISD, ou *single instruction single data stream*, um único processador processa um único elemento dos dados por unidade de tempo. Este tipo de arquitetura encontra-se presente em micro-controladores e antigos computadores pessoais;

- SIMD, ou *single instruction multiple data streams*, uma mesma instrução é aplicada a múltiplos elementos de dados a cada unidade de tempo. Esta arquitetura é caraterística de processadores vetoriais e operações de processamento gráfico;

- MISD, ou *multiple instructions single data stream*, um conjunto de unidades de processamento, conectadas sequencialmente, realizam diferentes operações sobre os mesmos dados. *Systolic arrays* são um exemplo de aplicação desta arquitetura;

- MIMD, ou *multiple instructions multiple data streams*, por unidade de tempo cada processador, contendo uma unidade de controlo independente, pode executar diferentes instruções de um programa. Esta arquitetura está presente na maioria das máquinas *multi-core* e multi-computadores.

2.2.1 Arquitetura *Multi-core*

Um processador *multi-core* segue a arquitetura MIMD, ou *multiple instructions multiple data stream*. Segundo esta arquitetura o CPU pode realizar múltiplas operações sobre diferentes dados e por isso é capaz de executar um programa em paralelo. Um processador multi-core é constituído por várias unidades de processamento contendo cada uma: unidades de lógica e aritmética, e unidades de controlo independentes.

2.2.2 Arquitetura *Many-core*

Um processador *many-core* segue a arquitetura SIMD, ou *single instruction multiple data streams*, é projetado para realizar processamento paralelo de menor consumo energético à custa de latência de comunicação entre os núcleos do processador, ou *core*, e de fraco desempenho singular dos processadores.

2.3 Unidades de Processamento Gráfico

Uma unidade de processamento gráfico, também designada de GPU, é um circuito especializado em alterar e manipular a memória de maneira a que o processamento de imagens a serem apresentadas num dispositivo de exibição seja acelerado. O GPU é extremamente independente em relação as restantes componentes de uma máquina por possuir a sua própria hierarquia de memória, mecanismos de transferência e de processamento de dados. As unidades de processamento gráfico são constituídas por múltiplos *streaming multiprocessors*, ou SM, que por sua vez são constituídos por múltiplos *stream processors*, ou SP.

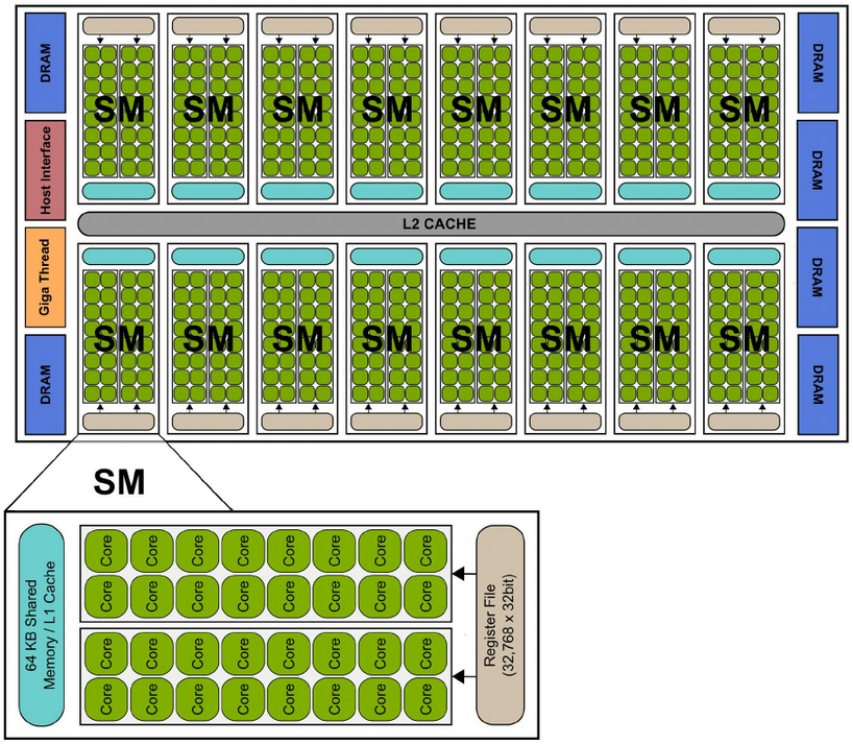

Figura 2.3: Arquitetura e composição interna de uma unidade de processamento gráfica [3].

Os *streaming multiprocessors* são unidades fundamentais de processamento dos GPUs. Os SM são constituídos por vários *stream processors* e uma unidade de controlo. A unidade de controlo é responsável por distribuir dados pelo conjunto de SPs. Esta unidade permite a execução síncrona da mesma instrução em todos os *stream processors*. Por esta razão, os *streaming multiprocessors* seguem a classificação de arquitetura SIMD.

Os *stream processors*, também designados de *cores*, são as unidades constituintes dos SMs, utilizadas efetivamente no processamento das operações atribuídas ao GPU. Os SPs seguem a arquitetura SISD. Por conseguinte, cada *stream processor* executa uma instrução sobre um conjunto específico de dados.

Devido à vasta quantidade de núcleos de processamento, à segregação dos mecanismos de acesso a memória e aos mecanismos de execução independente de instruções, os GPUs têm uma estrutura altamente paralela em comparação com os processadores, o que permite o processamento excecionalmente eficiente de blocos de informação em paralelo.

2.3.1 Hierarquia de Memória - GPU

A hierarquia de memória das unidades de processamento gráfico têm uma grande influência no desempenho das operações que esta executa. Uma estratégia de implementação que considere as diferentes caraterísticas dos múltiplos níveis de memória dos GPUs e dos seus mecanismos de gestão, permite tirar partido de um maior fluxo de dados e por consequência um melhor desempenho das operações realizadas nestas unidades.

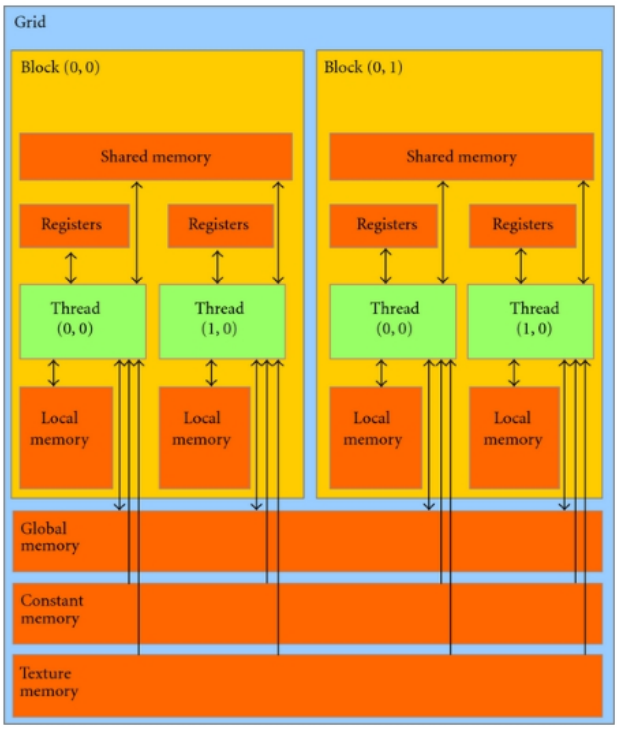

Figura 2.4: Disposição de memória de unidades de processamento gráfico [4].

Na figura 2.4 é apresentado um diagrama simplificado de um GPU constituído apenas por dois SMs, cada um com dois SPs, e as componentes de memória que constituem as unidades de processamento gráfico.

2.3.1.1 Memória Global

A memória global é componente de armazenamento principal das unidades de processamento gráfico. Toda este nível de memória é endereçável e os dados armazenados são persistentes durante toda a atividade do GPU, sendo apenas libertados ou removidos explicitamente por instruções ou pela desativação da placa gráfica.

Esta componente de memória apresenta a maior capacidade de armazenamento no GPU em relação a outros níveis de memória. Qualquer *streaming multiprocessor* do GPU pode aceder a esta memória e, por consequência, todos os SPs. Os acessos a este tipo de memória são geridos através de um sistema de cache de dois níveis, designados de cache L1 e L2. Existe uma instância de cache L1 presente em cada SM e apenas uma instância de cache L2 partilhada por todos os SMs.

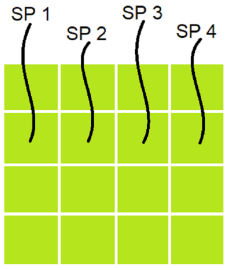

Figura 2.5: Acesso coalescido de memória pelos *stream processors*.

As respostas a acessos de memória global são serializadas pelo GPU para diminuir o tempo de transferência de dados. Assim, ao aumentar a largura de banda de transferência de dados são realizados menos acessos a memória [11]. Quando um *stream processor* realiza um acesso a memória global toda a região circundante à posição desse acesso é serializada e transferida para o sistema de cache. Para tirar o máximo proveito desta funcionalidade, o acesso a memória global de um GPU é mais eficiente, e com menor latência de resposta, quando cada SP acede a posições de memória consecutivas à posição de memória acedida pelos SPs vizinhos, como pode ser esquematizado pela figura 2.5. Este modelo de acesso a memória é denominado de acesso a memória coalescida, ou *coalesced memory acess*.

2.3.1.2 Memória Local

A memória local encontra-se ao mesmo nível da memória global devido à sua limitada velocidade de resposta a acessos de memória. A componente de memória local está associada independentemente a um SP. Este nível de memória é utilizada pelo GPU quando o *stream processor* não tem capacidade para armazenar em registos os valores das variáveis utilizadas nas suas operações. Os dados armazenados na memória local são conservados durante o tempo de execução de uma operação por parte do SP correspondente.

2.3.1.3 Memória Constante

A componente de memória constante encontra-se ao mesmo nível da memória global e, à semelhança da última, todo este nível de memória é endereçável pelos SMs e persistente durante toda a atividade do GPU. A memória constante tem o seu próprio sistema de cache que difere do sistema de cache da memória global. Todos os *stream processors* de um SM têm acesso à componente de memória constante, embora, apenas com capacidade de leitura e não de escrita. Os dados armazenados na memória constante são definidos pelo CPU antes da execução de operações do GPU.

O acesso a memória constante é extremamente rápido em comparação com os acessos a memória global e local, se todos os SPs de um SM acederem à mesma posição de memória. A latência de respostas a acessos a este nível de memória é reduzido, apresentando um desempenho semelhante a níveis de memória inferiores [12]. Esta peculiaridade deve-se a simplicidade do sistema de gestão de memória desta componente, pois não necessita de implementar mecanismos de escrita.

2.3.1.4 Memória de Textura

À semelhança da memória global, local e constante, a memória de textura pode ser acedida por todos os SMs de um GPU e encontra-se ao mesmo nível que as primeiras. A componente de memória de textura tem o seu próprio sistema de cache, embora, apenas com capacidade de leitura para o GPU enquanto o CPU é o responsável pela escrita dos dados nela armazenados.

O sistema de cache desta memória é ideal para acessos não coalescidos, o que torna este nível de memória uma opção eficaz para a impossibilidade de utilização eficiente da memória global [13]. O sistema de cache da memória de textura prioriza a localidade espacial dos acessos a memória realizado pelos *stream processors*. Isto é, o modelo de acesso coalescido compele o acesso a posições de memória consecutivas por parte de SPs consecutivos, enquanto, este sistema de cache permite o acesso eficiente de um *stream processor* a qualquer posição de memória desde que os dados acedidos pelo último se encontrem próximos de dados previamente acedidos por outros SPs.

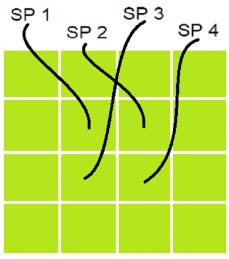

Figura 2.6: Acesso eficiente a memória de textura pelos *stream processors*.

Este padrão de acesso descrito é designado de prioridade de localidade espacial, ou *spatial locality*, e pode ser representado pelo diagrama da figura 2.6.

2.3.1.5 Memória Partilhada

A memória partilhada é uma componente de rápida resposta a acessos de memória por se encontrar integrada numa região próxima dos *stream processors* do GPU. É integrada em cada *streaming multiprocessor* de modo a possibilitar o acesso e transferências de dados entre os diferentes SPs que o constituem.

Esta memória apenas pode ser utilizada através de chamadas explicitas de métodos do GPU. Para armazenar dados da memória global nesta componente e tirar partido da reduzida latência de resposta de acessos a memória é necessário que a operação realizada em cada SP tenha explicitamente uma cópia ou escrita de dados coalescidos entre a memória global e esta componente [12].

A latência de resposta a acessos a memória partilhada é menor quando todos os *stream processors* acedem a diferentes bancos de memória (uma posição de quatro bytes de memória) ou ao mesmo endereço de memória.

2.3.1.6 Registos

Os registos são a componente de memória mais próxima dos SPs. Cada SP tem uma memória de registos própria que é independente dos restantes *stream processors*. Por estas razões, os registos são a componente de memória mais rápida das unidades de processamento gráfico [12], isto é, com menor latência de resposta a acessos de memória.

Os valores das variáveis utilizadas durante a execução de uma operação por parte de um SP são armazenadas nos seus registos correspondentes. Devido à reduzida capacidade de memória dos registos, caso não seja mais possível armazenar dados nesta componente, a memória local passará a ser utilizada. Os dados armazenados nos registos apenas são mantidos durante a atividade do SP correspondente.

Tabela 2.1: Visão geral dos níveis de memória do GPU

Memória	Permissões [1]	Acessível Por	Cache	Observações
Global	L/E	SPs, SMs e CPU	Sim	Abundante capacidade de armazenamento
Local	L/E	SPs	Sim	Reserva de armazenamento dos registos
Constante	L	SPs, SMs e CPU	Sim	Rápido acesso ao mesmo endereço
Textura	L	SPs, SMs e CPU	Sim	Localidade espacial de acesso
Partilhada	L/E	SPs, SMs	Não	Latência de transferências reduzida
Registos	L/E	SPs	Não	Acesso virtualmente instantâneo

[1] L/E - Leitura e Escrita; L - Leitura; E - Escrita;

2.4 CPU vs GPU

Em comparação, os processadores modernos são capazes de realizar processamento em paralelo de várias operações sobre múltiplos dados, de acordo com a arquitetura MIMD que seguem. Enquanto, as unidades de processamento gráfico seguem a arquitetura SIMD, implicando que estes dispositivos de processamento realizem apenas uma operação sobre múltiplos dados em paralelo.

A área de superfície dos circuitos dos CPUs é ocupada, maioritariamente, pelas unidades de controlo e o sistema de cache, restando apenas uma pequena área reservada a unidades de lógica e aritmética. Por outro lado, a área de superfície dos circuitos dos GPUs é constituída, essencialmente, por unidades de lógica e aritmética e apenas uma pequena região é dedicada a unidades de controlo e do sistema de cache.

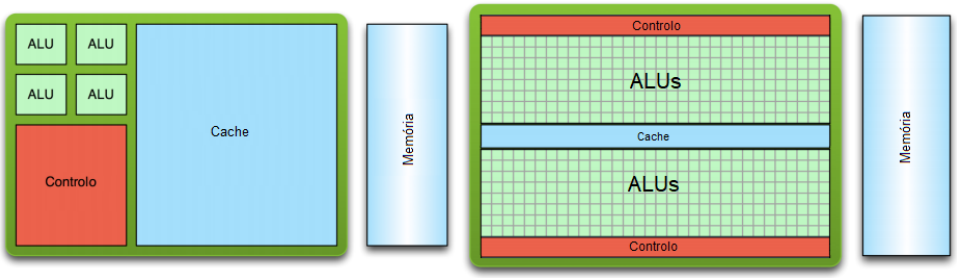

Figura 2.7: Diferenças de arquitetura entre CPUs e GPUs.

As diferenças arquiteturais destas unidades de processamento têm um impacto direto na forma como estas unidades desempenham. O GPU é mais restrito em termos de instruções executadas do que o CPU mas tem um maior potencial para realizar operações numéricas e executar problemas de processamento de dados. Outra diferença técnica entre este dois tipos de processadores é a alta largura de banda das unidades de processamento gráfico em relação ao CPU. Esta diferença é justificada pela presença dos sistemas de cache especializados das componentes de memória do GPU que permite a utilização eficiente dos seus vários níveis de memória, levando a uma menor latência de transferência de dados entre os núcleos de processamento do GPU e, por consequência, a um melhor desempenho de execução.

Os processadores mantêm o balanço entre o seu poder computacional e o propósito geral de execução das diferentes funções que realizam em paralelo. Por outro lado, as unidades de processamento gráfico proporcionam o máximo de poder computacional sofrendo das restrições que esse poder implica.

A relação entre o CPU e o GPU é um balanço entre a flexibilidade de funcionamento e a capacidade computacional. Cada uma destas diferentes unidades de processamento apresenta uma melhor eficácia na resolução de problemas em relação à outra dependendo do problema em questão. Devido as diferenças de arquitetura entre os CPUs e os GPUs é possível concluir que o CPU é uma unidade de processamento mais eficaz para processamento de múltiplas operações diferentes em paralelo em comparação com o GPU. Por outro lado, o GPU apresenta um melhor desempenho que o CPU na execução de problemas de processamento de múltiplos dados em paralelo.

2.4.1 Computação Heterogénea

Computação heterogénea refere-se a sistemas que utilizam mais que um tipo de processadores. Este tipo de sistemas têm alto nível de desempenho e eficiência de consumo de energia adicionando processadores especializados para diferentes tipos de objetivos. A arquitetura de sistemas

que implementam uma abordagem de computação heterogénea é, habitualmente, constituída por CPUs e unidades de processamento gráfico. Este tipo de sistemas encarrega a execução de tarefas de computação paralela intensiva ao GPU enquanto o resto do programa é atendido e executado pelo CPU.

2.5 Computação Paralela

Computação paralela é a designação atribuída ao mecanismo segundo o qual vários processadores executam ou processam uma determinada computação em simultâneo. A utilização de computação paralela permite realizar computações complexas dividindo a carga de trabalho por mais de uma unidade de processamento [14], todas as unidades de processamento realizam as operações ao mesmo tempo.

De modo a ser possível dividir um problema em diferentes partes, é essencial identificar o tipo de problema antes da formulação da solução paralela. Se P_D é um problema de domínio D e P_D é paralelizável, então D pode ser decomposto em k sub-problemas:

$$D = d_1 + d_2 + ... + d_k = \sum_{i=1}^{k} d_i \tag{2.1}$$

P_D é um problema de paralelização de dados se D é composto por elementos de dados e o problema é solucionado através da aplicação de uma determinada função $f()$ a todo o domínio:

$$f(D) = f(d_1) + f(d_2) + ... + f(d_k) = \sum_{i=1}^{k} f(d_i) \tag{2.2}$$

P_D é um problema de paralelização funcional se D é composto por diferentes operações e o problema é solucionado através da aplicação de cada operação a um mesmo conjunto de dados S:

$$D(S) = d_1(S) + d_2(S) + ... + d_k(S) = \sum_{i=1}^{k} d_i(S) \tag{2.3}$$

Problemas de paralelização de dados são ideais para serem solucionados recorrendo às potencialidades do GPU. A sua arquitetura SIMD é ideal para a resolução de problemas cuja solução é obtida através da aplicação da mesma instrução por parte de todos os núcleos de processamento em fragmentos de dados diferentes. Por outro lado, problemas de paralelização funcional são ideias para serem solucionados recorrendo às potencialidades do CPU, pois a arquitetura MIMD do CPU permite a execução eficiente de múltiplas tarefas diferentes em cada núcleo de processamento, ou *core*.

2.5.1 Modelos de Programação paralela

De modo a tirar partido das capacidades computacionais das unidades de processamento atuais e a implementar uma solução paralela eficaz de um problema, é necessário haver uma abstração da arquitetura de memória da plataforma de destino e dos detalhes técnicos das componentes de

processamento utilizadas. Os modelos de programação paralela fornecem paradigmas de implementação de soluções paralelas em que é possível a abstração das condições referidas [15]. Os modelos de programação paralela descrevem a interação entre a memória de uma máquina e as suas unidades de processamento: o modelo de memória partilhada e o modelo de memória distribuída.

2.5.1.1 Memória Partilhada

Segundo o modelo de memória partilhada, os processos que estão a ser executados nos núcleos de processamento, ou *threads*, interagem por variáveis declaradas no espaço de memória que partilham entre si. Com este modelo, os *threads* podem executar assincronamente sobre o mesmo conjunto de dados não havendo a necessidade de sincronização dos mesmos.

Contudo poderá ser necessária a implementação de uma secção crítica com mecanismos de controlo de leitura e escrita caso algum dos *threads* necessite de acesso exclusivo à memória. No modelo de memória partilhada cada *thread* é constituído pelo seu próprio estado interno e as variáveis globais partilhadas, definidas pelo *thread* principal.

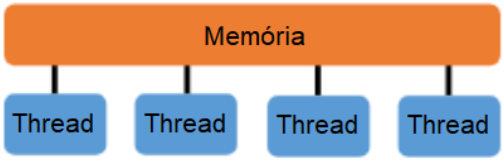

Figura 2.8: Modelo de memória partilhada.

Quando um dos *threads* acede um espaço da memória provoca que a região de memória circundante a esse local, designado de linha de cache, seja copiado para o sistema de cache do CPU. Referências e acessos subsequentes ao mesmo espaço de memória ou a dados da mesma linha de cache podem ser realizados sem recorrer à memória principal do sistema.

A linha de cache é preservada na memória cache até que o sistema determine que é necessário restituir a coerência entre esta memória e a memória principal. Alterações de elementos de uma linha de cache, realizadas por diferentes *threads*, invalida toda essa linha. A cada alteração a linha é marcada como inválida e todos os outros processos que a contém recebem o mesmo estatuto. O sistema obriga que estes processos recuperem a versão mais recente da linha invalidada a partir da memória principal de modo a manter a coerência de dados entre os *threads*.

Esta situação é denominada de *false sharing* e ocorre frequentemente em soluções baseadas no modelo de memória partilhada. O *false sharing* provoca um aumento do tráfico de comunicações dos *threads* e um maior tempo de latência de transferência de dados, o que diminui consideravelmente o desempenho de um sistema [16].

2.5.1.2 Memória Distribuída

No modelo de memória distribuída cada *thread* tem uma componente de memória independente dos restantes que contém uma cópia dos dados da tarefa a ser executada. Os *threads* in-

teragem entre si através de um sistema de mensagens utilizando a rede interna de comunicações do processador e, por essa razão, surge a necessidade de realizar operações de sincronização da execução de cada *thread* sobre os dados utilizados.

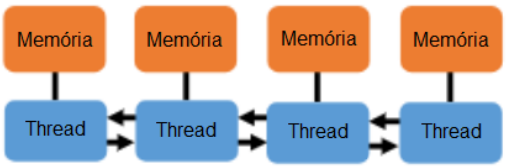

Figura 2.9: Modelo de memória distribuída.

2.5.2 Extração de Paralelismo

Ian Foster sugeriu uma metodologia utilizada para extrair paralelismo de um determinado problema e planear uma solução para o mesmo, maximizando o número de opções viáveis para o fazer e diminuindo o custo de retroceder na fase de desenvolvimento para resolver possíveis problemas que possam surgir.

Esta metodologia permite um melhor foco na fase inicial do processo de modelação de uma solução e criar uma abstração a problemas como compatibilidade de tecnologias, concorrência de execução da implementação e caraterísticas das componentes da máquina onde será integrada a solução [5].

Esta metodologia divide o processo de planeamento da arquitetura da solução em quatro fases distintas: divisão em partes ou *partitioning*, comunicação, aglomeração e mapeamento.

2.5.2.1 Divisão em Partes ou *Partitioning*

A fase de divisão em partes refere-se à decomposição em tarefas de dimensão inferior das atividades computacionais e dos dados do problema em questão. A decomposição das atividades computacionais a serem realizadas em tarefas disjuntas e dos dados é designada, respetivamente, de decomposição funcional e decomposição do domínio/dados.

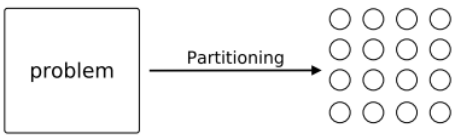

Figura 2.10: Fase de divisão do problema em partes [5].

2.5.2.2 Comunicação

A fase de comunicação concentra-se na criação do fluxo de informação entre as unidades de processamento e coordenação de execução das tarefas criadas na fase de divisão do problema em

partes através da instauração de canais de comunicação entre as diferentes divisões. A especificidade do problema e o método de decomposição realizado determina o padrão de comunicação entre as tarefas do programa paralelo.

Figura 2.11: Fase de criação dos canais de comunicação [5].

2.5.2.3 Aglomeração

A fase de aglomeração refere-se à fase de planeamento do sistema paralelo em que as tarefas idealizadas durante a fase de *partitioning* são agrupadas de modo a que a carga de trabalho necessária para as executar compensa a ocupação de uma unidade de processamento. Este aspeto é usualmente qualificado de granularidade do problema.

Um problema de granularidade fina pode ser dividido num elevado número de pequenas tarefas, enquanto um problema de granularidade grossa é dividido num número reduzido de tarefas de grande dimensão. Problemas de granularidade fina têm um alto potencial de serem paralelizáveis e apresentam uma maior latência de comunicações. Quanto aos problemas de granularidade grossa, este tipo de problemas têm uma solução fracamente paralelizável mas apresentam uma menor latência de comunicações.

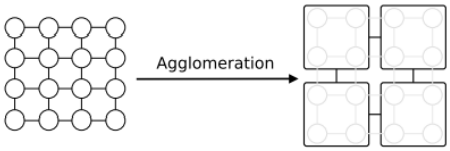

Figura 2.12: Fase de aglomeração de tarefas por unidades de processamento [5].

2.5.2.4 Mapeamento

A fase de mapeamento descreve o mecanismo de distribuição dos grupos de tarefas resultado da fase de aglomeração pelas unidades de processamento disponíveis. O mapeamento é a última fase da metodologia de Foster e pode ser conseguida através de várias estratégias. O objetivo desta fase é encontrar a melhor distribuição da carga de trabalho pelas unidades de processamento minimizando a latência de comunicações entre as últimas.

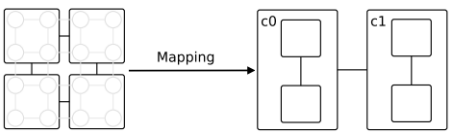

Figura 2.13: Fase de mapeamento de grupos de tarefas às unidades de processamento [5].

2.6 Ferramentas e Plataformas de Desenvolvimento

Na seguinte secção são apresentadas, descritas e comparadas as ferramentas e plataformas de desenvolvimento mais frequentemente utilizadas na atualidade para a implementação de sistemas que integram mecanismos de computação paralela e computação heterogénea.

2.6.1 OpenMP

A ferramenta OpenMP é uma API, ou *application programming interface*, multi-plataforma para as linguagens de programação C, C++ e Fortran. A utilização desta ferramenta permite paralelizar através de um mecanismo de *multi-threads* uma determinada solução.

Multi-thread é o mecanismo que permite a utilização das capacidades de processamento de todos os cores de um CPU. Com *multi-threads* é possível realizar diversos tarefas concorrentemente com total suporte do sistema operativo [6].

O OpenMP é uma ferramenta destinada ao modelo de programação paralela de memória partilhada implementando o paralelismo do sistema através de diretivas de compilador *pragma* recorrendo ao mecanismo de execução paralela designado de *fork-join*.

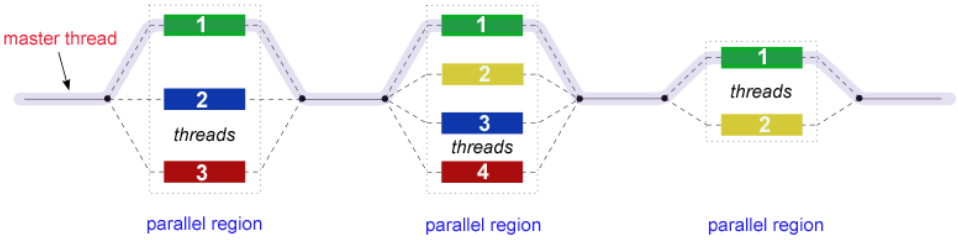

Figura 2.14: Mecanismo de paralelização *fork-join* utilizado OpenMP [6].

De acordo com o mecanismo *fork-join*, as diretivas de compilador presentes nas instruções do sistema indicam regiões paralelas que serão executadas por diversos *threads*. No começo de um programa paralelo implementado com OpenMP apenas um *thread* está tivo, também designado de *master thread*. O *master thread* ativa regiões paralelas de instruções a serem executadas sempre que as diretivas de compilador assim o indiquem.

2.6.2 *Message Passing Interface*

O *Message Passing Interface*, ou MPI é um standard de especificação de comunicação para sistemas paralelos que implementem o modelo paralelo de memória distribuída. O MPI especifica os métodos necessário para efetuar operações de comunicação e de controlo entre vários processos.

Seguindo este standard é possível criar uma rede de unidades de processamento independentes com componentes de memória individuais e capazes de executarem assincronamente diferentes operações.

2.6.3 OpenCL

O OpenCL é uma plataforma de desenvolvimento de software para o desenvolvimento de sistemas de computação heterogénea que aliam as capacidades computacionais das unidades de processamento gráfico e CPUs.

A plataforma OpenCL fornece uma interface de métodos necessários para a implementação de sistemas paralelos cujo problema é a paralelização de dados e funcional da solução. O OpenCL pode ser executado em sistemas operativos baseados em Unix e Windows, e integrado com as linguagens de programação C e C++.

Esta plataforma de desenvolvimento permite a criação de métodos que se baseiam em tarefas, designados de *kernels*, e a execução dos mesmos sobre diferentes dados por cada unidade de processamento. Cada *kernel* é colocado numa fila de espera de operações a serem realizadas e são executados logo que uma das unidades de processamento tenha disponibilidade para o fazer.

2.6.4 *Compute Unified Device Architecture*

A *Compute Unified Device Architecture*, ou CUDA, é uma plataforma de computação paralela criada pela NVidia para o desenvolvimento de sistemas paralelos recorrendo a aceleração gráfica.

A plataforma CUDA permite tirar partido das capacidades de processamento das placas gráficas para a execução de programas de propósito geral, do tipo *General Purpose computing on Graphics Processing Units*, ou GPGPU. Este tipo de designação carateriza sistemas paralelos que utilizam as unidades de processamento gráfico para operações de processamento que usualmente são realizadas pelo CPU. Estes sistemas analisam e processam dados como se de uma imagem ou outro tipo de formas gráficas se tratasse. Este paradigma atribui tarefas de intensa computação paralela ao GPU enquanto o restante programa é executado pelo CPU.

A plataforma CUDA pode ser executado em sistemas operativos baseados em Unix e Windows através das linguagens de programação C, C++ e Fortran, logo que estes sistemas contenham uma unidade de processamento gráfico da NVidia.

Em relação ao modo de funcionamento desta plataforma, a atividade da unidade de processamento gráfico é inicializada pelo CPU. O GPU opera sobre dados previamente transferidos a partir da memória da máquina para uma das componentes de memória do GPU. Após o processamento dos dados através das operações paralelas, também designadas de *kernels*, os dados resultados são transferidos da memória do GPU para a memória da máquina. Este processo liberta as capacidades de processamento do CPU para realizar outro tipo de operações enquanto o GPU se encontra em atividade. Por essa razão, esta plataforma permite tirar o máximo partido das capacidades computacionais de ambas componentes de processamento caso a solução seja corretamente implementada.

O GPU é especializado em processamento de dados através de operações bem definidas, ou *kernels*. Esta descrição compatibiliza com a descrição de um problema de paralelização de dados, logo é possível assumir que o GPU é ideal para o desenvolvimento de sistemas baseados neste tipo de problemas de paralelização [17].

2.6.5 Comparação Entre Ferramentas e Plataformas de Desenvolvimento

Considerando a abordagem de computação heterogénea desta dissertação é necessário aferir quais das ferramentas e plataformas de desenvolvimento trazem mais vantagens para as unidades de processamento utilizadas na solução do problema, o CPU e o GPU.

A paralelização do CPU pode ser efetuada através da implementação de *threads* nativos. Contudo, esta abordagem é mais trabalhosa pois todas as funcionalidades têm de ser desenvolvidas considerando as particularidades dos vários sistemas operativos onde a solução pode ser executada. Tendo em conta a redução da complexidade do código da solução foi escolhida a ferramenta OpenMP pois a sua utilização requer apenas da definição de diretivas de compilador *pragma* e é suportada pela maioria dos processadores utilizados na atualidade [6].

É possível verificar no seguinte excerto de código a fácil implementação de paralelização do CPU sem alterar drasticamente o código original, através da utilização do OpenMP. A inserção da diretiva de compilador permite que o bloco de instruções incluído no ciclo *for* seja dividido e executado pelos vários *threads* da máquina, introduzindo paralelismo de dados:

```
1 #pragma omp parallel for
2 for(int index = 0; index < N_ITERATIONS; index++){
3     foo();
4 }
```

O standar MPI é uma ferramenta versátil para o desenvolvimento de soluções paralelas de memória distribuída, especialmente quando esta solução recorre às capacidades de processamento de um grupo de diferentes máquinas, pois permite um mecanismo simples de transmissão de dados por sistemas com caraterísticas diferentes. Ainda assim, esta ferramenta não se enquadra com os objetivos desta dissertação.

O OpenCL e o CUDA são plataformas bastante semelhantes em termos de implementação de uma solução paralela, isto porque ambas necessitam da implementação de um *kernel* utilizado para o processamento de um conjunto de dados. Ambas plataformas apresentam funcionalidades idênticas o que torna o processo de portabilidade de programas entre estas plataformas simples de ser realizado.

A plataforma CUDA é destinada a implementação de soluções que utilizem as unidades de processamento gráfico da marca NVidia. Por essa razão, o CUDA apresenta um melhor desempenho de execução em todos os aspetos em comparação com a plataforma OpenCL [18]. Os aspetos referidos referem-se ao tempo de transferência de dados entre a memória da máquina e a componente de memória da unidade de processamento gráfico, o tempo de execução das operações de processamento e a complexidade de integração da ferramenta.

2.7 Conclusão

A partir do conteúdo apresentado neste capítulo podemos aferir que as diferenças das arquiteturas entre um CPU e um GPU têm implicações na sua performance. A arquitetura de um processador, devido aos níveis de memória que contém, apresenta um valor baixo de latência de acesso a memória principal por possuir vários níveis de memória cache que armazenam espaços de memória frequentemente acedidos. Os valores de tempo de acesso a memória só é afetado caso existam ocorrências de *false sharing* [16].

O GPU, por conter um número reduzido de unidades de controlo, permite a incorporação de um maior número de unidades de lógica e aritmética, o que possibilita trocar toda a flexibilidade funcional de processamento de um CPU por uma maior capacidade computacional.

Tendo em conta a abordagem de computação heterogénea utilizada na solução deste trabalho, as ferramentas que apresentam mais vantagens de utilização são a ferramenta OpenMP e a plataforma de desenvolvimento CUDA. A ferramenta OpenMP apresenta uma boa performance de execução para sistemas paralelos e é ideal para a paralelização funcional das instruções executadas no processador sem introduzir um nível de complexidade adicional ao código do programa. A plataforma de desenvolvimento CUDA destaca-se por apresentar um melhor desempenho em termos gerais em relação à plataforma OpenCL [18]. Também, a solução resultado deste trabalho será destinada a ser executada em máquinas com unidades de processamento gráfico da NVidia, o que torna a utilização do CUDA ideal nestas circunstâncias.

Referências

[1] Warren Toomey. Introduction to systems architecture. Disponível em http://minnie.tuhs.org/CompArch/Lectures/week01.html, Maio 2011.

[2] Wgsimon. Transistor count and moore's law. Disponível em https://en.wikipedia.org/wiki/File:Transistor_Count_and_Moore%27s_Law_-_2008.svg, Novembro 2008.

[3] Moisés Hernández, Ginés D Guerrero, José M Cecilia, José M García, Alberto Inuggi, Saad Jbabdi, Timothy EJ Behrens, e Stamatios N Sotiropoulos. Accelerating fibre orientation estimation from diffusion weighted magnetic resonance imaging using gpus. *PloS one*, 8(4):e61892, 2013.

[4] Mark Harris. Optimizing parallel reduction in cuda. nvidia dev. *Technology*, 2008.

[5] Ian Foster. *Designing and building parallel programs*, volume 78. Addison Wesley Publishing Company Boston, 1995.

[6] Blaise Barney. Openmp. *Website: https://computing. llnl. gov/tutorials/openMP*, 2008.

[7] William Knight. Two heads are better than one [dual-core processors]. *IEE Review*, 51(9):32–35, 2005.

[8] Philip E Ross. Why cpu frequency stalled. *IEEE Spectrum*, 45(4), 2008.

[9] Nikola Zlatanov. Computer memory, applications and management, 02 2016.

[10] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[11] Sunpyo Hong e Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. Em *ACM SIGARCH Computer Architecture News*, volume 37, páginas 152–163. ACM, 2009.

[12] Xinxin Mei e Xiaowen Chu. Dissecting gpu memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, 2017.

[13] Mark Sutherland, Joshua San Miguel, e Natalie Enright Jerger. Texture cache approximation on gpus. Em *Workshop on Approximate Computing Across the Stack*, 2015.

[14] Cristobal A Navarro, Nancy Hitschfeld-Kahler, e Luis Mateu. A survey on parallel computing and its applications in data-parallel problems using gpu architectures. *Communications in Computational Physics*, 15(2):285–329, 2014.

[15] David B. Skillicorn e Domenico Talia. Models and languages for parallel computation. *ACM Comput. Surv.*, 30(2):123–169, Junho 1998. URL: http://doi.acm.org/10.1145/280277.280278, doi:10.1145/280277.280278.

[16] Josep Torrellas, HS Lam, e John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

[17] Fedy Abi-Chahla. Nvidia's cuda: The end of the cpu?', 2008.

[18] Kamran Karimi, Neil G Dickson, e Firas Hamze. A performance comparison of cuda and opencl. *arXiv preprint arXiv:1005.2581*, 2010.