clOpenCL - OpenCL para ambiente cluster

Mário João da Costa Afonso

Relatório Final do Trabalho de Projecto apresentado à

Escola Superior de Tecnologia e Gestão

Instituto Politécnico de Bragança

para obtenção do grau de Mestre em

Sistemas de Informação

Dezembro 2012

clOpenCL - OpenCL para ambiente cluster

Mário João da Costa Afonso

Relatório Final do Trabalho de Projecto apresentado à

Escola Superior de Tecnologia e Gestão

Instituto Politécnico de Bragança

para obtenção do grau de Mestre em

Sistemas de Informação

Orientador:

Prof. Dr. Albano Alves

"Este Trabalho de Projecto não inclui as críticas e sugestões feitas pelo Júri."

Dezembro 2012

Agradecimentos

Ao meu orientador, Prof. Albano Alves, pelo seu empenho e orientação em me manter pelos caminhos certos. Especialmente pelo seu esforço cingido ao desenvolvimento.

Ao meu colega de investigação Tiago Ribeiro, que me proporcionou um ambiente de desenvolvimento prospero e um desembaraço de situações de estagnação na investigação.

Às pessoas envolvidas no projeto, Prof. António Pina e Prof. Luís Santos, pelas suas ideias e motivações que fizeram despertar todo o desenvolvimento.

Aos meus Pais e Família, que me suportaram e deram todo o seu apoio no decorrer de toda a minha vida estudantil.

Um especial agradecimento à Joana Rocha pelo seu companheirismo e suporte nas minhas escolhas ao longo deste trabalho.

Resumo

Nos dias correntes, as pessoas estão cada vez mais familiarizadas com a necessidade de um aumento dos recursos computacionais. Com isto, torna-se evidente a junção de dispositivos, como CPUs e GPUs de fabricantes diferentes, com a finalidade de preencher o mesmo propósito. Este nível de heterogeneidade é trazida pela tecnologia OpenCL.

O trabalho aqui desenvolvido abrange o High-Performance Computing e o OpenCL, com o resultado do clOpenCL. O objetivo do clOpenCL é suportar o conjunto de dispositivos OpenCL num ambiente cluster. O clOpenCL foi construído sobre duas vertentes, estando estas inteiramente ligadas ao tipo de tecnologia utilizada no processo de comunicação entre a biblioteca e o daemon. A primeira vertente da API foi construída sobre a tecnologia de comunicação socket TCP/IP e a segunda foi desenvolvida sobre o Open-MX. Ambos os modelos do clOpenCL estão ligados à tecnologia de comunicação e à forma como é suportada a comunicação ao nível de programação.

No decorrer do trabalho serão apresentadas todas as caraterísticas integradas pelo clOpenCL e todos os seus recursos abrangidos. Irá também ser analisada a performance obtida para cada vertente do clOpenCL, em termos de utilização de largura de banda e com a leitura e escrita em *buffers* remotos.

Palavras Chave: Heterogéneo, OpenCL, clOpenCL, Open-MX, socket TCP/IP.

Abstract

In the current days, people are increasingly acquainted with the need for an increase computing resources. It becomes evident the combination of devices such as CPUs and GPUs from different vendors, with the task of fulfill the same purpose. This level of heterogeneity, is brought by OpenCL technology.

This work covers the High-Performance Computing and OpenCL, with the result of clOpenCL. The main goal of clOpenCL is to support a set of OpenCL devices in a cluster environment. The clOpenCL was built in two branches, fully connected with the type of technology used in the comunication process, between the library and the daemon. The first version of the API was built on socket TCP/IP communication technology, and the second one was developed on the Open-MX tecnology. Both clOpenCL models are connected to the communication technology and the way that communication is suported at programming level.

Throughout this work, we will present all the clOpenCL integrated features and all its covered resources. We will also analyze the performance obtained for each side of clOpenCL in terms of bandwidth usage, through the reading and writing of remote buffers.

Key Word: Heterogeneous, OpenCL, clOpenCL, Open-MX, socket TCP/IP.

Prefácio

O OpenCL (Open Computer Language) é um standard de computação paralela e heterogénea desenvolvido pelo grupo Khronos. Com a introdução do OpenCL abrem-se novas perspetivas para tirar proveito do poder computacional de quaisquer dispositivos destinados à computação, como CPU e GPU, e entre outros aceleradores de gráficos. Áreas como o cálculo científico identificam-se intrinsecamente com esta perspetiva devido à exigência de processar um grande volume de dados.

É conhecido que um computador tem um número limitado de *slots* PCI para albergar aceleradores gráficos, como também de sockets para CPU, pelo que o OpenCL está confinado a ser executado neste tipo de ambiente. Todavia é nesta perspetiva que o trabalho desenvolvido aqui recai, aproveitando um conjunto de dispositivos OpenCL dispersos num ambiente cluster. Neste domínio foi desenvolvido uma API (Application Programming Interface) no âmbito do projeto de investigação "Portabilidade e Desempenho em Sistemas Paralelos Heterogéneos". O resultado final foi batizado de clOpenCL. Exposto numa forma rudimentar, o seu objetivo é alargar o OpenCL para o ambiente cluster, daí que a sua designação tenha em acréscimo "cl" à designação de OpenCL. A sua publicação encontra-se em [AA12].

Uma versão distribuída do OpenCL terá como objetivo garantir que qualquer aplicação que corra em OpenCL nativo possa correr num cluster, sem que haja necessidade de alterar o próprio código OpenCL. Para atingir esse objetivo seguiu-se de forma linear o modelo OpenCL apenas ao domínio distribuído. Para tal o clOpenCL utiliza um nível de

abstração idêntico que o próprio OpenCL utiliza, ou seja, todo o ambiente formalizado do clOpenCL tem correspondência direta ao OpenCL.

O modelo de operação adotado segue o paradigma cliente-servidor. No lado do cliente são manipuladas as primitivas OpenCL de forma adaptá-las quer às necessidades correspondentes à abrangência do cluster, quer à correspondência da própria especificação do OpenCL. Essa parte é denominada de biblioteca e exerce a função de um *wrapper* situado do lado da aplicação, manipulando todas as primitivas OpenCL invocadas.A biblioteca contém as referências das primitivas reais do OpenCL e é onde vai recair o maior esforço a nível computacional. Na aplicação principal apenas são invocados os protótipos implementados na biblioteca, que por sua vez são exatamente iguais aos protótipos das primitivas do OpenCL. O lado do servidor tem como objetivo atender pedidos remotos da biblioteca. O serviço destacado para atender esse objetivo é designado de daemon; cada máquina do cluster terá o serviço deamon a correr.

O modelo de execução do OpenCL assenta numa abordagem host-dispositivos e o mesmo acontece com o clOpenCL. O host é o componente pela qual a aplicação inicia a execução e a partir da qual se lançam e se coordena a execução dos Kernels, enquanto os dispositivos podem ser locais ou então espalhados pelo cluster. Embora os dispositivos estejam dispersos, foi introduzido o conceito de abstração da localização dos dispositivos. Com isso existe a ilusão de que todos os dispositivos estão localizados no nó local, onde se encontra a correr a aplicação.

O clOpenCL foi construído sobre duas vertentes, estando estas inteiramente ligadas ao tipo de tecnologia utilizada no processo de comunicação entre a biblioteca e o *daemon*. A primeira vertente da API foi construída sobre a tecnologia de comunicação socket TCP/IP e a segunda foi desenvolvida sobre o Open-MX. Em termos de implementação, as duas vertentes distinguem-se no procedimento da comunicação e na implementação da tecnologia.

Na API desenvolvida em Open-MX, as mensagens passadas entre a biblioteca e o

daemon são encapsuladas em dois pacotes: o pacote de transmissão e o pacote de receção. Estes pacotes são constituídos pelos parâmetros dos protótipos das primitivas OpenCL, na biblioteca o pacote de envio abrange os parâmetros de entrada da primitiva OpenCL e o de receção abrange os parâmetros de saída. No modelo socket TCP, a comunicação é efetuada de forma síncrona, entre sucessivas escritas e leituras de cada parâmetro dos protótipos das primitivas OpenCL.

O rCUDA, publicado em [JD11], emprega a mesma abordagem cliente-servidor, embora com designações diferentes para ambos os componentes. O lado do cliente foi designado por cliente *middleware* onde está especificado o *wrapper* para o CUDA *runtime*, onde são intercetados os pedidos da aplicação para o servidor. O lado do servidor foi designado como *server middleware*, sendo executado onde o GPU físico está hospedado. A ideia que fica é a verosimilhança entre as tecnologias a nível modelar.

O OpenCL e o CUDA ambos preenchem o mesmo propósito —GPGPU (general purpose GPU) —e acabam por ter abordagens bastantes idênticas, quer a nível de desenvolvimento quer a nível metodológico. Contudo o OpenCL é um standard com mais abrangência ao nível dos dispositivos. Ambas as tecnologias consistem num sistema CPU/GPU (host-device); o CPU é representado pelo host exercendo a função de estabelecer as computações entre a memória RAM e os dispositivos [DBK10]. Numa forma natural, o CUDA, sendo uma tecnologia proprietária da NVIDIA, direciona-se apenas para dispositivos NVIDIA com o suporte CUDA. Esta advertência faz com que o rCUDA fique limitado apenas a dispositivos NVIDIA.

Virtual OpenCL (VCL) é um trabalho idêntico ao clOpenCL, este é publicado no [AB11]. O VCL é mais extenso, relativamente ao clOpenCL, envolvendo três componentes; a biblioteca, o *broker* e o *back-end daemon*. A componente biblioteca tem o mesmo propósito que a biblioteca clOpenCL e que o *client-midleware* no rCUDA. O componente *broker* é um *daemon-process* com a utilidade de rastreabilidade dos dispositivos OpenCL e toda a manutenção nos nodos onde o serviço está a ser executado. O VCL, neste componente, introduz segurança na comunicação e impõe uma maior robustez na implementação, o que

induz peso na comunicação. Tal situação não é desejável, uma vez que já a própria comunicação entre nodos é um gargalo, tornando impraticável com a necessidade acrescida de efetuar operações com carga computacional muito elevada. A abordagem do componente *back-end daemon* é idêntica ao *daemon* do clOpenCL, emulando todas as operações OpenCL pedidas pela biblioteca e redireciona-as de novo.

A comunicação na tecnologia rCUDA, entre o cliente e o servidor, e no VCL, entre o *broker* e a biblioteca é efetuada através de sockets TCP/IP. No entanto, o VCL, para além do *broker* a consumir comunicação com a biblioteca, existe o *back-end daemon* a efetuar comunicação através da tecnologia MPI (Message Passing Interface).

Todas as tecnologias analisadas aqui abraçam um número de características idênticas, inclusive a tecnologia desenvolvida neste trabalho; demonstram características em termos de modelo de construção equivalentes, particularmente na interação entre cliente e multi-servidor, demonstrando assim a necessidade de materializar aspetos remotos para zonas locais. O que verdadeiramente difere das demais tecnologias, é a abordagem na implementação; todas as tecnologias abordadas utilizam para fins comunicativos, entre o cliente e o servidor, a tecnologia MPI ou sockets TCP/IP. No entanto, o clOpenCL foi desenvolvido com a tecnologia de comunicação Open-MX, embora tenha também sido usado, numa fase inicial a tecnologia TCP/IP.

Conteúdo

Agradecimentos **v**

Resumo **vi**

Abstract **vii**

Prefácio **viii**

Lista de Acrónimos **xviii**

1 Introdução **21**

1.1 Definição do problema 22

1.2 Tecnologia OpenCL 25

1.2.1 Modelo de Execução 26

1.2.2 Modelo de Memória 26

1.2.3 Primitivas OpenCL 27

1.2.4 Aplicação OpenCL 28

1.3 Descrição dos capítulos seguintes 30

2 Tecnologia adotadas **31**

2.1 Modelo clOpenCL sobre Socket TCP 31

2.1.1 Diretório Global de Visibilidade 33

2.1.1.1 Reestruturação do DGV 33

2.1.2 Sincronismo e Fragmentação 34

2.1.3 Sockets TCP 34

2.1.3.1 Aspetos na Comunicação Socket 35

2.1.3.2 Integração com o clOpenCL 35

2.2 Modelo clOpenCL Open-MX 36

2.2.1 Pacotes de Comunicação 37

2.2.2 Fragmentação de mensagens 37

2.2.3 Diretório Global de Visibilidade 37

2.2.4 Open-MX 38

2.2.4.1 Processo Comunicativo 38

2.2.4.2 Qualificação das Mensagens 39

2.3 Epílogo 39

3 Implementação e Execução **40**

3.1 Interposição de Bibliotecas 40

3.1.1 Linkagem 41

3.2 Inicialização de Recursos 43

3.2.1 Levantamento das Plataformas clOpenCL 43

3.2.2 Encapsulamento e exposição das Plataformas 45
3.3 Interação e construção aplicação/daemon 45
3.3.1 daemon clOpenCL TCP 46
3.3.1.1 Teste de Integridade 47
3.3.1.2 Efetividade do Serviço de Diretório 47
3.3.1.3 Comunicação 47
3.3.2 Biblioteca clOpenCL TCP 48
3.3.3 daemon clOpenCL Open-MX 51
3.3.4 Biblioteca clOpenCL Open-MX 56
3.4 Objetos clOpenCL 63
3.4.1 Espaço de Endereçamento 63
3.4.2 Tratamento do clGetPlatformIDs 65
3.5 Epílogo 68

4 Aspetos no desenvolvimento do clOpenCL 69

4.1 Plataformas com o clOpenCL 69
4.1.1 Disposição 70
4.1.2 Localização Física 71
4.2 Tratamento do Objeto Contexto 71
4.2.1 Localização Objeto Plataforma 72
4.2.2 Tradução do Objeto clOpenCL para OpenCL 73
4.2.2.1 Processo de Pesquisa 74

4.3 Objectos de Memória 75

4.3.1 Escritas e Leituras dos Objetos de Memória 76

4.3.1.1 Fragmentação das Mensagens 76

4.3.2 Eventos 77

4.3.2.1 Resolução de Estados 78

4.4 Objeto Programa 78

4.4.1 Encapsulação e Encaminhamento 79

4.5 Parâmetros Kernel 80

4.6 Epílogo 81

5 Comparações e Benchmarks **82**

5.1 Incidência dos testes 82

5.2 Largura de Banda utilizada 83

5.2.1 TCP vs Open-MX 83

5.3 Consumo do cpu 84

5.3.1 Primitivas Alvo 85

6 Conclusões **87**

6.1 Análise da implementação e resultados 87

6.2 Trabalho Futuro 88

Bibliografia **89**

Lista de Tabelas

3.1 Representação da tabela Open-MX em cada nó. 57

4.1 Disposição das Plataformas no OpenCL e no clOpenCL. 70

5.1 Tempo de execução em nanosegundos das primitivas em ambas os modelos. 86

Lista de Figuras

1.1 Modelo OpenCL. 23

1.2 Modelo clOpenCL. 24

2.1 Modelo de execução clOpenCL com a tecnologia TCP. 32

2.2 Modelo de execução clOpenCL com a tecnologia Open-MX. 36

3.1 Modelo de uma aplicação com o clOpenCL. 42

3.2 Tratamento das Plataformas com o clOpenCL. 44

3.3 Processo de recolha de Plataformas. 45

3.4 daemon Runtime do modelo TCP/IP. 46

3.5 Processo de execução da Biblioteca clOpenCL TCP. 48

3.6 clOpenCL Runtime do modelo Open-MX. 52

3.7 clOpenCL biblioteca Runtime do modelo Open-MX. 56

5.1 Performance de leitura e escritas em nós remotos. 84

5.2 Performance de leitura e escritas em nós remotos. 85

Lista de Acrónimos

OpenCL Open Computer Language

clOpenCL cluster Open Computer Language

AMD Advanced Micro Devices

API Application Programming Interfaces

CU Compute Unit

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

GPP General Purpose Processors

GPU Graphics Processing Unit

GPGPU General-Purpose Graphics Processing Units

HPC High-Performance Computing

MPI Message Passing Interface

OS Operative System

OpenCL Open Computing Language

PE Processing Element

SDK Software Development Kit

SMP Symetric MultiProcessing

SIMD Single Instruction, Multiple Data

SPMD Single Program, Multiple Data

VCL Virtual OpenCL

rCUDA remote Compute Unified Device Architecture

Capítulo 1

Introdução

A programação paralela tem ganho cada vez mais adeptos, sendo fácil de perceber o motivo que está por detrás do crescimento da empregabilidade deste paradigma; o mundo tem-se tornado cada vez mais competitivo, a todos os níveis, e com a competitividade é exigido um maior capacidade de processamento. Desde o nível empresarial até à vida social, a chave é obter um maior aproveitamento dos gastos face ao investimento efetuado.

A questão que se põe é: qual a necessidade de adquirir um novo hardware, se o presente não tem um aproveitamento de cem porcento? Esta questão torna-se mais pertinente com a introdução de CPUs com capacidades de executar um número significativo de threads e GPUs com suporte para um maior número de threads relativamente as CPUs. Na realidade, é um facto, se por alguma razão fosse possível desdobrar as pessoas em dois ou mais, as tarefas que tivessem que exercer torna-se-iam mais rápidas.

A programação concorrente está estritamente ligada ao aumento significativo do aproveitamento do hardware, tornando os processos mais eficazes na utilização dos recursos. Todavia, a grande revolução deste paradigma aconteceu com a introdução da alta escalabilidade dos GPUs; para termos uma ideia, o atual acelerador gráfico da NVIDIA —GTX 690 —integra 3072 unidades computacionais.

Até 2006 existia uma enorme relutância no desenvolvimento de aplicações que tirassem partido dos dispositivos gráficos. Até à data, as APIs de alto nível que suportassem os dispositivos gráficos eram poucas e com bastantes limitações; o Direct3D e o OpenGL por exemplo, na tentativa de programar estes dispositivos, era necessário um profundo conhecimento do seu hardware. As próprias APIs existentes eram utilizadas apenas para representação gráfica, tendo pouca utilidade para a execução de algoritmos pesados que exigissem poder de cálculo. Com a intenção de preencher essa lacuna, surgiu a tecnologia CUDA, criada pela companhia NVIDIA. O CUDA tira proveito do paradigma da concorrência envolvendo os dois dispositivos CPU/GPU, para tirar melhor proveito do hardware proprietário, ou seja, apenas é suportado o hardware fabricado pela NVIDIA.

É neste contexto que o OpenCL surge, de forma que o programador não tenha necessidade de reescrever o código cada vez que programe para uma plataforma diferente. O propósito do OpenCL é ser uma API standard heterogénea de maneira que se encaixe em qualquer hardware independentemente do fabricante. Especificamente, o OpenCL foi desenhado para suportar dispositivos com capacidades diferentes, inerentes a uma plataforma específica.

É nesta perspetiva que foi desenvolvido o clOpenCL, com o intuito de ser uma API direcionada para um ambiente cluster. O agregado de máquinas heterogéneas fornece um aumento significativo na performance das aplicações. A vantagem seria enriquecer o OpenCL e aproveitar o potencial abrangido pelo ambiente cluster heterogéneo. O trabalho foi desenvolvido num ambiente Linux CentOS 5 com um *front-end* e 4 nós, todos eles máquinas reais.

1.1 Definição do problema

A abordagem face ao desenvolvimento do clOpenCL, foi efetuada de duas formas diferentes, tendo em conta a forma de comunicação entre os elementos do clOpenCL, refletindo assim o peso que as comunicações têm no desenvolvimento de aplicações para ambientes

cluster. A conceção do clOpenCL não se desviou da rota do OpenCL, uma vez que o modelo de programação introduzido pelo próprio OpenCL indica uma transposição intrínseca com um ambiente alargado, conforme demonstrado na Figura 1.1.

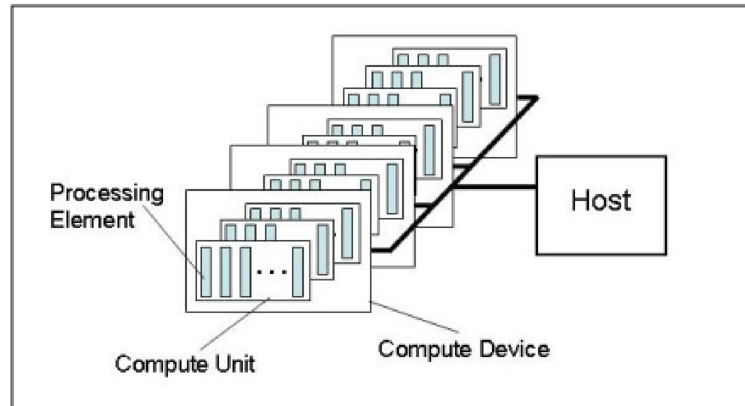

Figura 1.1: Modelo OpenCL.

O modelo de programação que está representado na Figura 1.1 é constituído por um host interligado com um ou vários dispositivos OpenCL. No modelo OpenCL, o host funciona como um manipulador dos dispositivos. Todo o fluxo de operações exigidas aos dispositivos é operado através do host. Existe uma sequência de operações necessárias na realização de uma aplicação OpenCL e todas essas operações são transportadas pelo próprio host. Operações como a criação de *buffers* de memória nos dispositivos e a inicialização do *kernel* nos dispositivos, serão abordadas com mais profundidade em capítulos posteriores.

No cumprimento do modelo apresentado pelo OpenCL, de forma a justificar a sua abrangência para um ambiente cluster, foi então desenvolvido, de forma bastante idêntica, o modelo de programação do clOpenCL. De forma a justificar a transposição implícita no modelo nativo OpenCL para o modelo clOpenCL é apresentado na Figura 1.2 o esquema global da operação.

Pode-se observar que neste modelo o host localiza-se onde a aplicação se encontra a ser executada, podendo-se verificar a existência de apenas um host para todo o cluster.

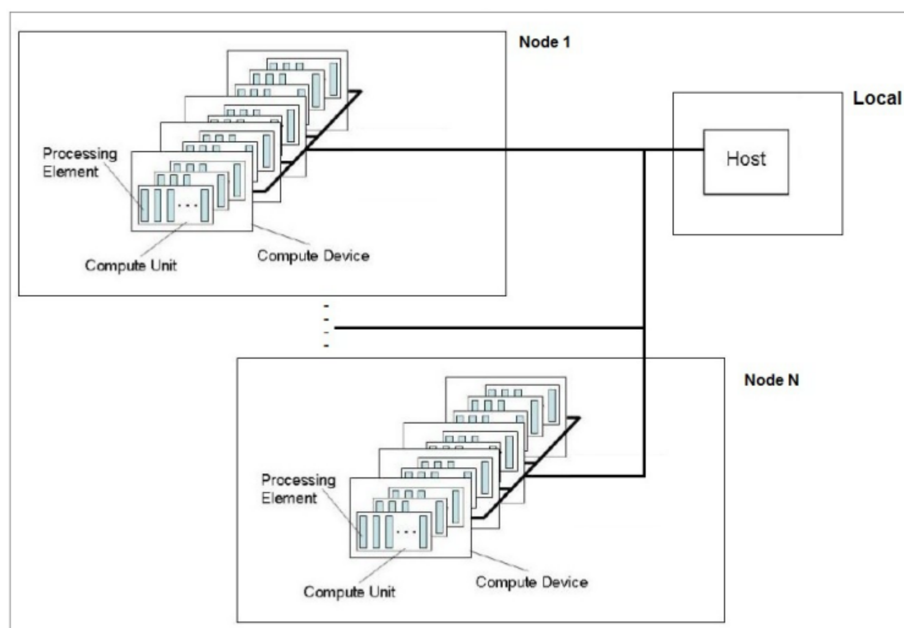

Figura 1.2: Modelo clOpenCL.

No modelo OpenCL o host é descrito apenas para um sistema isolado. A ideia inicial foi desenvolver o clOpenCL por forma a que uma aplicação nativa OpenCL executasse numa máquina específica, neste caso no *front-end*, onde a sua função se resume apenas ao acesso e não à produção de computações, para além das comunicações exercidas entre as máquinas adjacentes. Deste modo, as especificações de hardware onde a aplicação OpenCL executa serão minimalistas. Nós adjacentes ao *front-end* apenas irão ser tratados como dispositivos, como o próprio modelo OpenCL os define.

No modelo clOpenCL existe uma abstração do host em todos os nós adjacentes, dando a ideia que cada nó é tratado apenas como um ou vários dispositivos. Para tornar este modelo exequível foi necessário conceber um novo modelo de execução.

O modelo de execução do OpenCL está dividido em duas partes: 1) o *kernel* onde é executado nos dispositivos OpenCL, 2) e o programa host, onde são definidos os contextos para a execução do *kernel*. Os contextos abrangem uma série de recursos, enquanto os dispositivos OpenCL são utilizados pelo host para a execução de *kernels*. *Kernels* são

funções especificadas pelo OpenCL que irão ser executadas pelos dispositivos. Objeto programa é o código fonte e executável do *kernel*, código compilado pelo próprio OpenCL. Por fim, objetos de memória são configurações de memória geridas pelo host e visíveis pelos dispositivos OpenCL, sendo operados por instâncias do *kernel*.

Todo este conjunto de recursos OpenCL foi analisado para averiguar a sua exequibilidade no modelo clOpenCL. O peso maior no desenvolvimento correspondem ao suporte às comunicações entre as máquinas adjacentes, definindo esta vertente dois modelos de execução diferentes [Mun11]. Os demais problemas encontrados estão relacionados insoluvelmente com o suporte de *buffers* remotos em memória RAM e o suporte de contextos globalizados.

1.2 Tecnologia OpenCL

Na secção anterior foi apresentada a tecnologia OpenCL, mas apenas se demonstrou o seu modelo de plataforma. Existe agora a necessidade de compreender o que o OpenCL pode fazer e que tipo de propósitos preenche. É de realçar que o OpenCL oficialmente não é uma linguagem de programação mas sim uma especificação desenvolvida sobre a linguagem C, com uma vertente na linguagem de programação C++ (apesar de apenas ser uma *wrapper* da API desenvolvida em C). Esta API foi inicialmente proposta pela APPLE e desenvolvida pelo grupo Khronos que engloba várias APIs abertas bastante conhecidas como o OpenGL, por exemplo.

A especificação OpenCL permite que cada fabricante desenvolva plataformas para os seus próprios dispositivos, com as necessárias adaptações. É por isso que o OpenCL é uma especificação de plataformas [Mun11]. Atualmente estão disponíveis três plataformas: a AMD, Intel e NVIDIA.

1.2.1 Modelo de Execução

O modelo de execução define a efetividade representativa da execução de trabalho nos dispositivos, com recurso a uma grelha que na qual integra dois componentes: *work-groups* e *work-items*. Cada *work-group* é representado fisicamente pelas unidades computacionais e contém vários *work-items*. Os *work-items* fisicamente são representados pelos elementos de processamento. No ponto de vista de desenvolvimento, este modelo traduz um ganho no paralelismo SIMD (única instrução, múltiplos dados), em que cada *work-item* é visto como uma instância do *kernel* executando de forma concorrente, ou SPMD (programa único, múltiplos dados) no intuito de cada *work-group* manter um contador de programa.

1.2.2 Modelo de Memória

O OpenCL introduz o seu próprio modelo de memória, definindo cinco regiões de memória distintas. Estas regiões estão diferenciadas perante o conjunto de objetos do OpenCL e, de acordo com os tempos de acesso.

A região onde o host se localiza é apenas visível para o host. A especificação OpenCL apenas define como a memória do host interage com os objetos OpenCL e os seus construtores. As regiões de memória intrínsecas à execução do *kernel* são as quatro zonas restantes.

A região de memória global, na disposição hierárquica, está no fundo, sendo a zona de memória vista por todos os elementos e pode ser escrita e lida por todos os *work-items*.

A região de memória que se segue na hierarquia, apesar de ser uma zona específica da memória global é a memória constante. O OpenCL representa esta zona numa camada superior à zona de memória global. A informação é armazenada pelo host e é mantida no formato constante ao longo da execução do *kernel*. Os *work-items* apenas têm permissões de leitura na zona de memória constante.

No próximo nível hierárquico está representada a região de memória local. Esta zona é

local para um *work-group*. Pode ser utilizada para alocar variáveis partilhadas por todos os *work-items* representados pelo respetivo *work-group*. Pode ser implementada como uma região de memória dedicada ao dispositivo OpenCL.

No topo da hierarquia do modelo de memória do OpenCL está alojada a região de memória privada. Esta apenas serve para alojar as variáveis alocadas por cada *work-item* que só são visíveis pelos mesmos.

1.2.3 Primitivas OpenCL

O OpenCL apresenta um conjunto de objetos cada um com a sua particularidade na execução de um programa; no conjunto global de uma aplicação OpenCL, os objetos têm uma respetiva disposição em tempo de execução.

O objeto de Plataforma tem como objetivo inicializar recursos OpenCL e fornecer os apontadores de memória reservada para cada Plataforma; é criado através da primitiva *clGetPlatformIDs*. O objeto Dispositivo tem como objetivo vincular-se a um dispositivo referente ao objeto Plataforma para futuras computações. A obtenção do objeto Dispositivo pode ser efetuada através da primitiva *clGetDeviceIDs*. O Objeto Contexto é utilizado para fins de manuseamento do objeto Dispositivo, tendo a capacidade de agregar uma série de dispositivos para uma possível partilha de recursos. Este objeto pode ser obtido a partir da primitiva *clCreateContext*. Segue-se o objeto Command Queue com a função de manter uma configuração de comandos que ditam a ordem de execução nos dispositivos. Este objeto é obtido através da primitiva *clCreateCommandQueue*. Segue-se o objeto Programa, o qual consiste numa configuração de fontes de *kernels*, agregando uma ou mais fontes *kernel* no mesmo objeto. O objeto Kernel encapsula o binário que foi gerado para a função do *kernel*, para ser executado nos dispositivos associados e é obtido através da primitiva *clCreateKernel*.

Um dos recursos com mais peso no OpenCL é oferecido pelo objeto Buffer; este peso está diretamente relacionado com a frequência com que é utilizado e com a quantidade de

dados que manipula. O objeto Buffer tem como função desempenhar um papel fronteiriço, ou seja, a troca de dados entre o host e os dispositivos é efetuada através do objeto Buffer. Este objeto é obtido através da primitiva clCreateBuffer. Para além do objeto de memória *buffer*, o modelo OpenCL apresenta o objeto Image, com as propriedades idênticas ao objeto Buffer. O objeto Image tem como alvo armazenar imagens de uma a três dimensões, com o formato específico do OpenCL. Ainda no âmbito do tratamento de imagens, o objeto Sampler descreve como efetuar a amostra da imagem, quando esta for executada no *kernel*.

De forma a obter sincronismo, o OpenCL define o objeto Evento. Este encapsula uma série de estados agregados a uma operação específica, como a escrita de um buffer ou uma sub-configuração acionado pelo próprio utilizador.

Não é de todo necessário manusear todos estes objetos numa aplicação OpenCL, para se obter sucesso na execução, embora seja necessário tratar de um certo número de objetos OpenCL de forma obrigatória, de maneira a que a aplicação OpenCL possa ser executada.

1.2.4 Aplicação OpenCL

Na análise efetuada à execução de uma aplicação OpenCL, existe uma exigência na efetividade de determinados recursos para a concretização de uma aplicação OpenCL. Na escrita de um programa OpenCL é exigido obrigatoriamente o levantamento das plataformas através da primitiva *clGetPlatformIDs*, na perspetiva de realizar os passos posteriores. Com a obtenção da plataforma, de seguida é necessário obter os dispositivos disponíveis para exercer trabalho. Os dispositivos podem ser obtidos com a primitiva *clGetDeviceIDs*, com indicação da plataforma extraída anteriormente.

O passo seguinte seria a contextualização dos dispositivos extraídos de uma determinada plataforma. A contextualização pode ser efetuada com a primitiva *clCreateContext*; na execução desta primitiva vincula-se todos os dispositivos ou apenas um a um contexto. Um passo seguinte seria a criação de uma fila de execução do dispositivo através

da primitiva *clCreateCommandQueue*. A particularidade desta primitiva é a agregação ao objeto de contexto e a vinculação apenas a um dispositivo, ou seja, caso a solução agregue vários dispositivos seria obrigatório invocar a primitiva *CommandQueue* para cada dispositivo isoladamente. Não obrigatoriamente, neste sentido, procede-se à criação de um ou vários objetos de memória, consoante as necessidades na execução do *kernel* no dispositivo. A criação do objeto de memória é efetuada através da primitiva *clCreate-Buffer*, com a agregação ao objeto contexto. A leitura e escrita dos objetos de memória podem ser efetuadas através das primitivas *clEnqueueReadBuffer* e *clEnqueueWriteBuffer*. A agregação com o objeto contexto remete para uma partilha de buffers entre os dispositivos vinculados no objeto contexto. O OpenCL não permite criar contextos com dispositivos de diferentes plataformas e com isso o estado de partilha entre dispositivos de plataformas diferentes é desabilitado. A primitiva envolve alguma complexidade em torno do uso da memória, sendo possível explicitar a região de memória que se pretende utilizar como alvo.

Para fins de construção do *kernel* é necessário gerar o objeto programa com a primitiva *clCreateProgramWithSource*. Esta tem agregado o objeto contexto e recebe o programa através da leitura do array que contém todo o código para a construção do *kernel*. Após a geração do objeto programa procede-se à compilação do *kernel* através da primitiva *clBuildProgram*. Esta primitiva agrega o objeto Programa e a lista de dispositivos associados ao programa. O processo de compilação é efetuado consoante as especificações dos dispositivos, podendo causar alguma incompatibilidade entre dispositivos de fornecedores diferentes. Depois de construído o programa, é gerado o objeto *kernel* através da primitiva *clCreateKernel*, com a agregação ao objeto programa. Para finalizar a operação é efetuada a passagem de parâmetros para o *kernel* através da primitiva *clSetKernelArg* e o início da sua execução é despoletado através da primitiva *clEnqueueNDRangeKernel*. O processo que efetua a inicialização da execução do dispositivo tem agregado os objetos CommandQueue e o Kernel.

1.3 Descrição dos capítulos seguintes

Relativamente aos capítulos seguintes, o capítulo dois dá ênfase as tecnologias de comunicação utilizadas para o desenvolvimento de ambos os modelos do clOpenCL, demonstrando as tecnologias e características de cada tecnologia e dos modelos. O capítulo três tem a especial incidência nos aspetos de programação relativamente a ambos os modelos. No capítulo quatro descreve-se a interação ao nível de programação entre o clOpenCL e o OpenCL e toda a sua tradução. O capítulo cinco faz referência aos respetivos resultados obtidos na implementação de um exemplo, comparando os valores entre os modelos clOpenCL e o próprio OpenCL. Finalmente, o capítulo seis apresenta as apreciações finais e as perspetivas futuras.

Capítulo 2

Tecnologia adotadas

Como referido anteriormente, este trabalho foi desenvolvido em duas vertentes, sendo elas definidas pelas próprias comunicações entre o daemon e a biblioteca. Consequentemente, envergou-se por dois modelos de execução diferentes, embora com algumas semelhanças na cerne da implementação. Este capítulo foca-se primariamente nos dois modelos construídos para o clOpenCL, referindo a sua composição e técnicas abordadas como também as tecnologias de comunicação abordadas por ambos.

2.1 Modelo clOpenCL sobre Socket TCP

Inicialmente desenhou-se um modelo de execução com base na tecnologia de comunicação socket TCP. A justificação do modelo clOpenCL Socket TCP vai ao encontro de uma generalização e uma interoperabilidade elevada. Este modelo serve assim como base comparativa e para fins de demonstrar a simplicidade e generalização inserida neste projeto, dado que o clOpenCL foi desenvolvido sobre a tecnologia Ethernet.

Sendo o uso de sockets TCP/IP referido como uma tecnologia de baixo custo e com níveis de latência elevados [YTL07] [PB05], prevalece como uma base para fins comparativos. Em capítulos posteriores esta análise irá ser feita com maior profundidade. A

representação gráfica deste modelo está explícita na Figura 2.1.

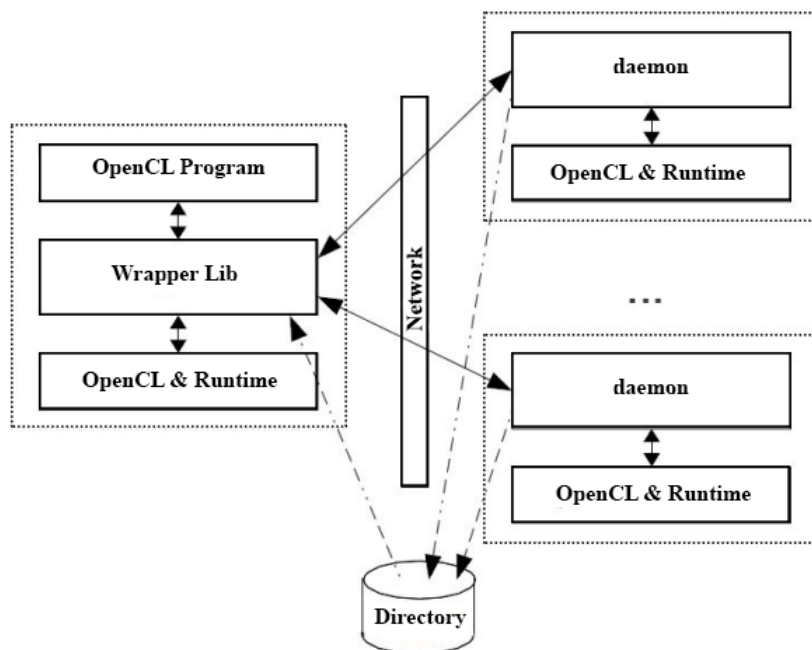

Figura 2.1: Modelo de execução clOpenCL com a tecnologia TCP.

É visível na Figura 2.1 a disposição das camadas funcionais e o fluxo entre elas. A execução de uma aplicação com o clOpenCL envolve dois níveis, separadas pela camada de rede; 1) no lado esquerdo da camada de rede está representado o espaço de endereçamento da biblioteca, 2) no lado direito à camada de rede está representado o espaço de endereçamento do daemon. No clOpenCL, a execução propriamente dita de uma aplicação ocorre no primeiro nível, que representa o nó local. A aplicação é envolvida num *wrapper* com a função de intercetar e manipular as primitivas OpenCL. Após o tratamento das primitivas estar concluído é invocada a verdadeira biblioteca e o *runtime* do OpenCL. O segundo nível está relativamente ligado ao desempenho e manutenção do serviço daemon.

2.1.1 Diretório Global de Visibilidade

O modelo de execução do clOpenCL inclui um diretório com vários fluxos direcionais, denominado de diretório global de visibilidade (DGV). O propósito do diretório é armazenar as portas de comunicação e o próprio endereço IP das máquinas onde os serviços daemon irão ser executados, usando para o efeito, uma base de dados MYSQL, facilitando assim o processo de identificação e de vinculação às portas ativas de cada serviço em execução.

No lado da biblioteca, o *wrapper* consulta o diretório uma única vez, quando é feita a inicialização do programa OpenCL. Na parte do serviço daemon é desencadeado uma operação de registo quando o daemon é colocado em funcionamento. Este mecanismo garante que a aplicação local pode obter o conhecimento dos nós nos quais se encontram serviços recetivos à comunicação com a biblioteca. Relativamente à implementação do acesso à base de dados de ambos os lados recorreu-se ao conhecido conector desenvolvido em linguagem de programação C MYSQL.

2.1.1.1 Reestruturação do DGV

Na implementação do DGV foi seguida uma implementação bastante simples. Contudo, estão presentes algumas limitações relativamente à aplicabilidade deste modelo num ambiente cluster. Será necessário uma intervenção administrativa para configurar um repositório global, visível por todos os nós. O sistema operativo usado foi o Rocks, que traz instalada uma base de dados MYSQL autoritária que apenas poder ser acedida por utilizadores devidamente credenciados. Desta forma, para tornar este modelo funcional recorreu-se à configuração de uma instalação MYSQL diferente da base de dados do próprio sistema operativo, tornando-a pública para todos os nós e sem intervenções administrativas.

2.1.2 Sincronismo e Fragmentação

O processo de comunicação entre a biblioteca e o daemon é efetuado através de sockets e com vertente síncrona. Em tempo de execução, a comunicação é efetuada conforme a necessidade da parametrização da própria primitiva. Na manipulação de primitivas OpenCL, com a necessidade de execução remota, são efetuadas sucessivas escritas e leituras, todos elas de forma síncrona, ou seja, todos as escritas que são efetuadas pela biblioteca são remetidas para o daemon como parâmetros de entrada da primitiva. As leituras efetuadas são os parâmetros resultantes da própria primitiva ou os parâmetros configurados para entrada de dados. A respetiva forma contrária encaixa-se no processo de comunicação do daemon.

Obviamente que este tipo de processo de comunicação não é efetuado da mesma forma para todas as primitivas. Existem primitivas OpenCL com exigências mais delicadas relativamente ao tratamento de dados. Por exemplo, as primitivas de escrita e leitura dos *buffers* podem lidar com uma quantidade dados bastante significativas. Para a resolução desse problema foi necessário fragmentar os parâmetros das primitivas OpenCL com um número considerável de bytes, para tamanhos de 4 MB, ou seja, parâmetros com mais de 4 MB são fragmentados e enviados através de sucessivas leituras e escritas dependendo do caso.

2.1.3 Sockets TCP

O modelo TCP/IP é um dos modelos de comunicação com mais sucesso de implementação e bastante difundido [PG05]. O protocolo TCP apresenta características favoráveis para a sua utilização. Tem a capacidade de endereçamento recebe um pacote de um emissor e redireciona-o para uma particular aplicação, com o auxílio de um conjunto de portas bem conhecidas, para fins de obter uma ligação com outro ponto de comunicação. Um recurso TCP bem conhecido e talvez a base do seu sucesso é a segurança na transmissão de dados e a fiabilidade no canal *byte-stream*. A fiabilidade e a segurança na transmissão de dados

é obtida através da deteção e recuperação da perda de dados, redundância e outro tipo de erros que possam a vir ocorrer.

2.1.3.1 Aspetos na Comunicação Socket

Do ponto de vista do programador a comunicação com sockets TCP é uma tecnologia bastante difundida, utilizada para estabelecer comunicação entre o par cliente-servidor, para construção de aplicações sobre Ethernet, herdando todos os recursos do protocolo TCP. O socket é apenas uma abstração que a aplicação utiliza para enviar e receber dados, permitindo também efetuar uma ligação à rede e estabelecer ligações com outras aplicações alojadas na mesma rede. Existem vários tipos de sockets, que estão inteiramente relacionados com o tipo de protocolo que representam. Os principais tipos de sockets são os sockets sobre TCP, denominados socket *stream*, e os sockets sobre o protocolo UDP, denominados socket *datagram*. As diferenças entre estes dois tipos de sockets estão associadas às especificações integradas por cada protocolo. Por exemplo o socket stream oferece fiabilidade na troca de dados, enquanto o socket datagram já não garante essa fiabilidade dado que se baseia no protocolo UDP (mas, em contrapartida a troca de mensagens é efetuada de forma mais rápida).

2.1.3.2 Integração com o clOpenCL

Neste trabalho as comunicações foram implementadas através de sockets stream de forma a tornar o cenário de troca de mensagens mais seguro. Este processo de comunicação envolve um par (cliente e servidor) —requer uma série de procedimentos. Para estabelecer uma ligação através de sockets stream, no lado do servidor é necessário criar o socket e estabelecer uma vinculação ao endereço IP local. O passo seguinte implica estabelecer um número máximo de ligações ao socket e, por fim, será necessário aceitar conexões. No lado do cliente, os processos envolvidos são reduzidos relativamente ao servidor; o cliente apenas necessita de criar o socket e estabelecer a ligação do mesmo socket ao endereço

onde o serviço se encontra alojado.

2.2 Modelo clOpenCL Open-MX

Os caminhos que levaram ao desenvolvimento de um segundo modelo, construído de raiz com suporte à tecnologia de comunicação Open-MX, foram vários. Entre eles a promessa de um melhor desempenho nas comunicações entre a biblioteca e o daemon e, acima de tudo, aliviar toda a carga computacional na CPU exercida pela receção e envio de pacotes sobre TCP/IP. O modelo relativamente à construção do clOpenCL sobre o Open-MX comparativamente ao modelo socket TCP, difere em dois níveis: ao nível de como é efetuada a comunicação e ao nível do conceito. O modelo clOpenCL com o Open-MX está representado graficamente na Figura 2.2.

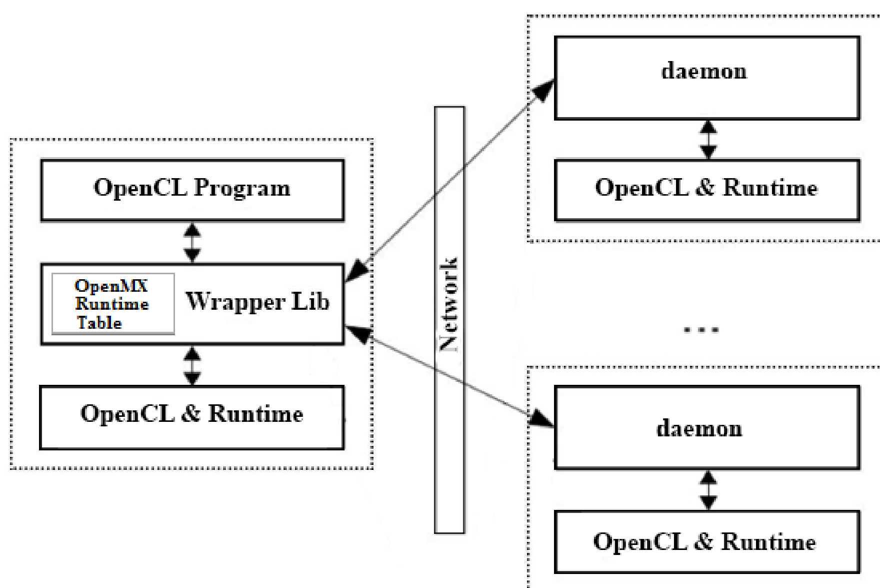

Figura 2.2: Modelo de execução clOpenCL com a tecnologia Open-MX.

2.2.1 Pacotes de Comunicação

Ao nível de comunicação foram definidos pacotes genéricos. Toda a comunicação exigida por ambas as partes é realizada por pacotes nos quais se encontram encapsuladas estruturas de dados. Os pacotes têm tamanhos variáveis; o tamanho do pacote depende dos parâmetros da primitiva a ser tratada. Na biblioteca, os pacotes são divididos em dois: o pacote de envio e o pacote de receção. O pacote de envio é constituído por um identificador, que identifica a respetiva primitiva OpenCL a ser tratada remotamente, e os respetivos campos referentes aos parâmetros de entrada da primitiva. O pacote de receção é remetido pelo daemon e é constituído pelos parâmetros de saída gerados pela primitiva. Acontece que é imprevisível estipular um tamanho exato para o pacote e, quando se trata de primitivas com especiais exigências, como já referido no modelo socket, torna-se necessário tratar individualmente parâmetros com essas configurações. Esses parâmetros são fragmentados e tratados de forma distinta dos pacotes.

2.2.2 Fragmentação de mensagens

Nas primitivas OpenCL destinadas à escrita e leitura de buffers são gerados os pacotes igualmente com os demais parâmetros, mas durante o processo de comunicação é efetuada a fragmentação de 4 MB por envio, estabelecendo assim uma comunicação síncrona com sucessivos envios e receções. Relativamente ao mecanismo de comunicação desenvolvido no modelo socket, esta solução é mais elegante e mais limpa em termos de leitura de código.

2.2.3 Diretório Global de Visibilidade

A nível de conceção, conforme apresentado na Figura 2.2, é notória a ausência do diretório global introduzido no modelo socket TCP e o acréscimo de uma sub-camada dentro da camada biblioteca. O diretório é substituído pela camada Open-MX Runtime Table. A

sub-camada Open-MX Runtime Table representada na Figura 2.2, tem como objetivo efetuar um levantamento aos dispositivos de rede, os quais se encontram registados na tabela fornecida pelo próprio Open-MX. Essa verificação é efetuada através de um script implementado em Python, que varre a tabela do Open-MX, verificando ao longo de cada identificador Open-MX (NIC) a sua vinculação ao serviço. Este processo foi designado como endereçamento global, fundamentando assim a relação com o comportamento remetido pelo modelo Socket do diretório global de visibilidade.

2.2.4 Open-MX

O Open-MX foi desenhado para fornecer uma alternativa ao Myrinet Express, sistema de baixo nível de passagem de mensagens sobre hardware proprietário Gigabit Myrinet e mesmo sobre 10-Gigabit Ethernet. O Open-MX oferece uma interface programável para facilitar a interoperabilidade no hardware, trabalhando perfeitamente sobre a camada Ethernet genérica. Assim, a pilha de rede construída pelo Open-MX oferece a interface programável do MX nativo e o seu interface binário; deste modo consegue manter os recursos do MX e a interoperabilidade com vários tipos de hardware. O suporte ao protocolo MX implica a mesma estruturação dos métodos de passagem de mensagens. A diferença está na forma como são escritos e lidos os dados, pelos controladores de rede. Enquanto o MX utiliza técnicas diretas de transferência de dados, no lado do emissor o PIO (programmed input/output) e DMA (Direct Memory Access) e no recetor apenas o DMA, o Open-MX baseia-se nos Socket Buffers manipulados por todos os controladores Ethernet. Portanto, o Open-MX emula todo um conjunto de comunicações MX utilizadas nativamente no firmware MXoE, na camada Ethernet no *kernel* do Linux [Gog11] [Gog08].

2.2.4.1 Processo Comunicativo

O processo de comunicação envolve vários elementos. O NIC é um identificador único de 64 bits ligado à rede física, utilizado para criar um ponto de ligação nas aplicações. O

resultado do processo da criação de ponto de ligação é o endpoint ID. Todo o processo de comunicação com os computadores remotos passa pelos *endpoints*. Cada NIC pode conter associados vários endpoints. A fim de não comprometer a troca de mensagens em cada *endpoint*, o Open-MX implementa no processo de criação do endpoint o endpoint filter. O *endpoint filter* é um inteiro que é atribuído pela aplicação de modo a distinguir as diferentes mensagens passadas pelo mesmo meio físico.

2.2.4.2 Qualificação das Mensagens

Foram desenvolvidos três métodos distintos no envolvimento da passagem de mensagens. Mensagens pequenas, compreendidas até aos 128 bytes, são otimizadas no lado do emissor escrevendo os dados através do NIC usando o método PIO, transferindo assim dados entre a CPU e o adaptador de rede; dado que o tamanho de dados é relativamente pequeno, há envolvimento da CPU, transmitindo-se um único pacote de dados. Mensagens médias estão compreendidas entre os 129 bytes a 32 kB e são enviadas pelo hardware através da DMA, em vários pacotes se for necessário. As mensagens grandes estão compreendidas para além do 32 kB e são processadas através do método *zero-copy*. Este método permite a transferência de dados entre a memória da aplicação ao nível do utilizador e o NIC através do DMA, permitindo assim uma redução na latência e na utilização da CPU [Gog08].

2.3 Epílogo

Estes modelos têm como propósito ligar um programa OpenCL com as várias componentes clOpenCL distribuídas num cluster, para que o utilizador da biblioteca se abstenha de todo o processo de *linkagem* e execução. Este capítulo debruçou-se sobre o modelo e funcionamento do clOpenCL e tecnologias usadas na sua implementação.

Capítulo 3

Implementação e Execução

No desenvolvimento deste projeto, o maior peso esteve do lado da implementação, para suportar um conjunto de primitivas OpenCL tornando-as utilizáveis num ambiente distribuído. Neste capítulo é feita uma abordagem mais técnica, expondo um conjunto de tecnicismos envolvidos na implementação e a própria implementação para ambos os modelos.

3.1 Interposição de Bibliotecas

A interposição de bibliotecas não é uma técnica nova, esta é abordada no desenvolvimento de mecanismos de *wrapping*, permitindo efetuar alterações numa biblioteca em tempo de execução [SS08]. A técnica de interposição está presente neste trabalho, possibilitando assim, a interação do OpenCL com o clOpenCL.

O mecanismo clOpenCL inclui parte do mecanismo suportado pelo OpenCL, dado que o comportamento ao nível da API é exatamente o mesmo. O utilizador não necessita ter conhecimentos adicionais ao OpenCL para se integrar com o clOpenCL. A biblioteca clOpenCL engloba o processo de reencaminhamento para o OpenCL; o processo de inclusão é efetuado com a *header* do próprio OpenCL. O que difere é o processo de *linkagem*

com o objeto da biblioteca. O processo de *linkagem* de uma aplicação OpenCL envolve a biblioteca "libOpenCL.so", que é uma biblioteca partilhada globalmente, reconhecida pelo sistema. A biblioteca clOpenCL funciona como um intermediário do OpenCL, ou seja, interceta as primitivas dirigidas ao OpenCL, efetua o processamento necessário e encaminha-as novamente para o OpenCL.

3.1.1 Linkagem

A biblioteca clOpenCL é concebida através de um processo de *wrapping* de nomes. Este processo, para os nomes das primitivas OpenCL, acontece em tempo de compilação da aplicação OpenCL. O conceito é aplicado através da opção do gcc –Xlinker. Por exemplo, para a primitiva clGetPlatformIDs o *wrap* é definido da seguinte forma:

- -Xlinker –wrap=clGetPlatformIDs

A flag –wrap permite fazer *wrapping* das primitivas em tempo de *linkagem*. O processo de *linkagem* através deste método irá resolver a referência para a primitiva clGetPlatformIDs com a nova chamada desta primitiva através do nome __wrap_clGetPlatformIDs. A flag –wrap origina ainda um novo símbolo referenciado por __real_, que irá resolver a primitiva original do OpenCL no formato __real_clGetPlatformIDs. Este método foi baseado na solução apresentado na publicação [DSM04]. Todas as outras primitivas OpenCL suportadas pelo clOpenCL sofrem o mesmo processo em tempo de *linkagem*. A efetividade de uma aplicação vinculada ao clOpenCL é representada na Figura 3.1, onde são definidos três objetos com elevada importância na interação do clOpenCL numa aplicação: o objeto de biblioteca do OpenCL, a biblioteca do clOpenCL e a própria aplicação. O cabeçalho cl_wrapper.h exerce a função de ponte de ligação entre o OpenCL e o clOpenCL, incorporando todas as primitivas do OpenCL e a *main* da aplicação com o prefixo __real_.

A aplicação mantém no seu espaço de endereçamento os objetos das bibliotecas clOpenCL e o OpenCL, de forma a que o clOpenCL possa comunicar com o OpenCL. A

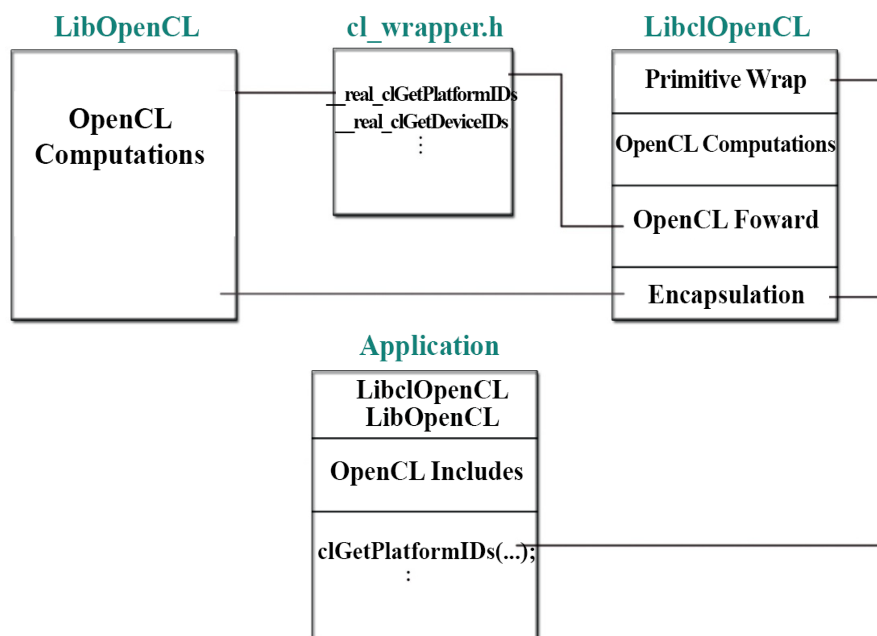

Figura 3.1: Modelo de uma aplicação com o clOpenCL.

primitiva definida conforme a especificação OpenCL é intercetada pelo clOpenCL. Assim, a biblioteca clOpenCL funciona como um intermediário, que manipula as chamadas das primitivas OpenCL na aplicação e injeta informação adicional na aplicação. No domínio da aplicação é feita a *linkagem* aos dois objetos de biblioteca clOpenCL e OpenCL e são efetuadas as inclusões da própria API do OpenCL. O processo de interceção das primitivas na aplicação acontece quando é chamada uma primitiva OpenCL.

As primitivas OpenCL são intercetadas pela camada Primitive wrap, no objeto libclOpenCL. Efetuada a interceção, são desencadeadas computações na biblioteca e de seguida a primitiva é reencaminhada através do cl_wrapper.h, para execução no espaço de endereçamento da biblioteca OpenCL. Os dados resultantes da execução da primitiva na biblioteca OpenCL são encapsulados na biblioteca clOpenCL e seguidamente são injetados na aplicação.

O processo de criação da biblioteca clOpenCL é posto em prática com a ferramenta "ar" de forma a conceber uma biblioteca estática.

3.2 Inicialização de Recursos

A especificação do OpenCL não inclui explicitamente uma primitiva para inicialização de recursos. No entanto, as APIs que estão geralmente relacionadas com a interação e computação em hardware específico têm a particularidade de providenciar ao programador primitivas para esse efeito. Como exemplo, o OpenMPI e o OpenGL expressam esse comportamento de inicialização através da primitiva MPI_Init e glinit, respetivamente.

3.2.1 Levantamento das Plataformas clOpenCL

O OpenCL tem uma abordagem peculiar relativamente ao comportamento de inicialização. Conforme a abordagem multi-plataforma do OpenCL, quando invocada a primitiva *clGetPlatformIDs*, é desencadeada uma pesquisa no sistema onde é feita uma verificação de cada fornecedor através do ICD (installable client driver loader). Portanto, a primitiva *clGetPlatfomIDs* funciona como um ponto de partida na inicialização de um programa OpenCL.

No seguimento desta particularidade do OpenCL, todas as inicializações necessárias no contexto de uma aplicação do clOpenCL são desencadeadas a partir da interceção da primitiva clGetPlatformIDs. A primitiva clGetPlatformIDs é constituída por três parâmetros e retorna uma estrutura de dados parametrizada pelo próprio OpenCL.

- cl_int ret clGetPlatformIDs (cl_uint *n_entries*, cl_platform_id **platforms*,cl_uint *num_plat*)

O primeiro parâmetro da primitiva *n_entries* tem como objetivo delimitar o número pretendido de plataformas. O parâmetro *platforms* é um apontador para uma estrutura de dados específica do OpenCL, a qual contém as plataformas armazenadas, exercendo a

função de parâmetro de saída. O último parâmetro *num_plat* exerce a funcionalidade de parâmetro de saída indicando o número de plataformas que existem no sistema na sua totalidade. Foi necessário construir um processo que agilizasse a transposição da primitiva clGetPlatformIDs nativa do OpenCL para captar a abrangência introduzida pelo clOpenCL.

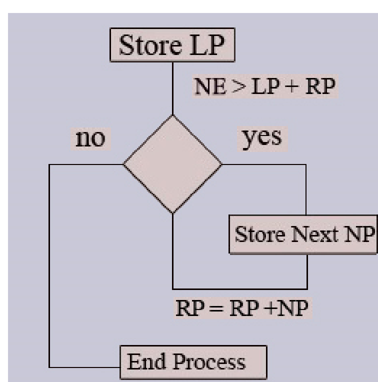

Figura 3.2: Tratamento das Plataformas com o clOpenCL.

Na Figura 3.2 está representado o processo seguido pelo clOpenCL a fim de efetuar uma recolha de plataformas localizadas pelos nós dispersos no cluster. Todo o processo contém uma série de conceitos introduzidos pelo clOpenCL, o quais não são abordados pelo OpenCL. O conceito de plataformas remotas, representado na Figura 3.2 pela nomenclatura RP, representa o resultado das plataformas remotas, recolhidas na totalidade. O conceito plataformas do nó, representado pela nomenclatura NP (Node Plataform), é o número de plataformas recolhidas no nó emergente. São conceitos que naturalmente têm peso na abordagem do clOpenCL relativamente à materialização do OpenCL e, completamente desconhecidos para o próprio OpenCL. Todo o processo é realizado localmente com o protagonismo da variável NE (número de entradas). O NE, no ponto de vista do clOpenCL, age equivalentemente ao OpenCL apenas para execuções locais, fazendo-se em primeiro lugar o levantamento das LP (plataformas locais). Consequentemente, o clOpenCL consegue economizar verificações desnecessárias, evitando lançar uma pesquisa aos nós. A obtenção de plataformas é efetuada uma a uma, em cada nó adjacente, até

que seja preenchido o máximo de NE entradas.

3.2.2 Encapsulamento e exposição das Plataformas

O conjunto de plataformas recolhidas, quer locais quer remotas, será encapsulado numa única estrutura de dados, composto por um apontador fabricado pelo clOpenCL e pela respetiva indicação relativamente à localização da plataforma para fins comunicativos. O cenário final produzido pela primitiva clGetPlatformIDs está representado na Figura 3.3.

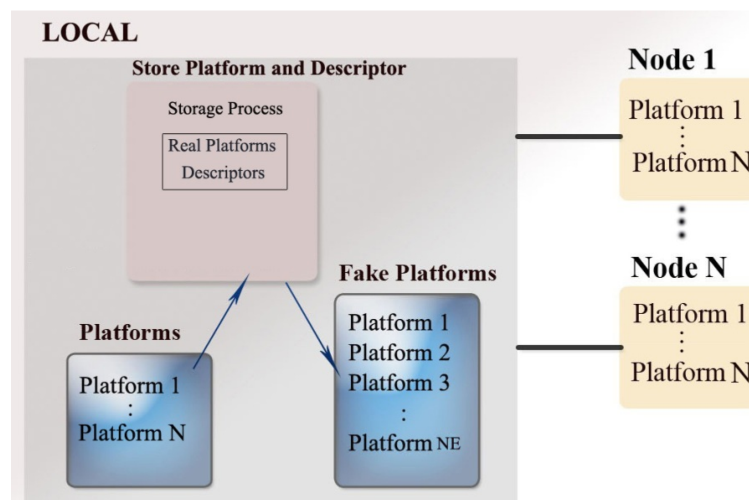

Figura 3.3: Processo de recolha de Plataformas.

O processo de recolha de plataformas é efetuado em vários estados, explícitos na Figura 3.3. Os estados estão definidos desde as plataformas locais ao processo de armazenamento até à organização final representada no bloco Fake Plataforms. Este bloco final irá ser acedido durante a execução de uma aplicação OpenCL.

3.3 Interação e construção aplicação/daemon

A abordagem seguida é baseada na interação cliente-servidor; o servidor é representado pelo daemon e o cliente é representado pela biblioteca. Ambos os modelos seguem esta

filosofia e cada tecnologia implementa de forma diferente a filosofia cliente-servidor.

3.3.1 daemon clOpenCL TCP

O serviço daemon é todo ele construído sobre a tecnologia socket TCP/IP, fazendo com que seja necessário levar a cabo uma série de instruções para tornar o serviço funcional, segundo os procedimentos do socket TCP/IP e realizar o propósito do alargamento do OpenCL. O processo efetuado pelo clOpenCL a fim de estabelecer uma abertura ao exterior para fins comunicativos já foi devidamente exposto no segundo capítulo. O que ficou por abordar foi a técnica por detrás do próprio processo.

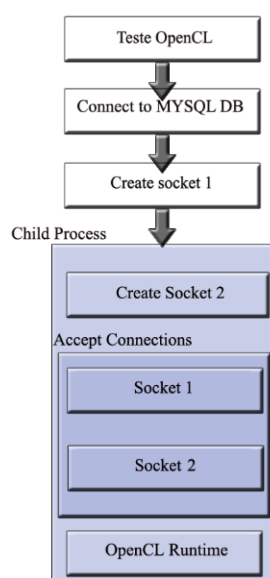

Figura 3.4: daemon Runtime do modelo TCP/IP.

A Figura 3.4 representa o fluxo do *runtime* do serviço daemon e as componentes relativas a cada processo, o qual está dividido em dois. No decorrer da thread principal do serviço são desencadeadas as rotinas de inicialização e, no processo derivado da thread principal indicado pelo bloco Child Process na Figura 3.4, são delegadas as rotinas que tratam de transportar as computações do OpenCL para a localização da aplicação. Todo

o mecanismo é efetuado em tempo de execução do serviço daemon, como a a) verificação do OpenCL, b) a ligação à base de dados MYSQL, c) a criação e o estabelecimento dos sockets e d) o próprio mecanismo de registo do daemon.

3.3.1.1 Teste de Integridade

A verificação da existência de plataformas OpenCL localmente ao serviço é necessária de forma tornar o serviço útil. Este é denominado de teste de integridade no nó ao OpenCL. É procedido através de uma rotina que faz uso da primitiva clGetPlatformIDs, tentando extrair as plataformas existentes no nó e com isso expor a sua integridade de maneira a tornar possível a interação com a biblioteca.

3.3.1.2 Efetividade do Serviço de Diretório

A fim de dinamizar o acesso à base de dados MYSQL, é executada uma rotina que faz uso da API desenvolvida para a linguagem C de acesso à base de dados MYSQL e que tem como objetivo estabelecer ligação à base de dados e retornar o conetor.

3.3.1.3 Comunicação

Na comunicação e interação com a biblioteca são usados dois sockets de comunicação. Estes sockets, quer no serviço quer na biblioteca, servem propósitos diferentes. O primeiro socket destina-se às ocorrências normais desencadeadas através de pedidos da biblioteca i.e o processamento de primitivas OpenCL que suportam como comportamento uma execução procedimental e que não dependam de fatores assíncronos para a sua execução. Este socket representa a implementação síncrona do OpenCL, enquanto o segundo socket representa a implementação assíncrona. O bloco referenciado na Figura 3.4 por Accept Connections é suportado pelo *runtime* do OpenCL, ou seja, as computações realizadas a pedido da biblioteca provêm do primeiro socket, a fim de serem processadas pelo *runtime*

do OpenCL. Conforme a descrição da primitiva, relativamente ao seu fluxo de execução, os seus resultados são reencaminhados através dos respetivos sockets.

3.3.2 Biblioteca clOpenCL TCP

O clOpenCL é uma API com o objetivo de expandir a implementação do OpenCL, a fim de suportar um conjunto de mecanismos do OpenCL. Num ambiente computacional distribuído, através do auxílio de um serviço ativo em cada elemento computacional. A sequência de operações aplicáveis ao clOpenCL é representado na Figura 3.5.

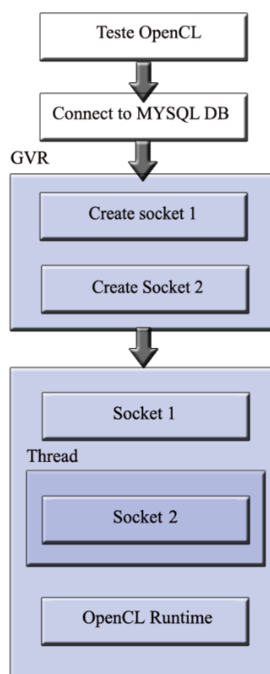

Figura 3.5: Processo de execução da Biblioteca clOpenCL TCP.

A main thread desencadeada pela aplicação vinculada à biblioteca clOpenCL é uma cascata de intercetações acionadas por primitivas OpenCL. O clOpenCL, quando interceta a função main da aplicação escrita em OpenCL, efetua um conjunto de operações antes de transportar as próprias computações para o OpenCL. Dentro da interceção do main são efetuadas as inicializações do MYSQL e o teste de integridade do OpenCL, da mesma

forma que acontece no daemon. O bloco que se segue é configurado segundo o diretório global de visibilidade, ou seja, o socket 1 é gerado com base nas propriedades do socket stream e é vinculado aos serviços registados na base de dados através da descoberta do seu endereço de rede e da porta efetiva do serviço. A vinculação do socket 2 é uma resultante do socket 1. O socket 2 é vinculado enviando através do socket 1 a descrição da porta do serviço. Os dois sockets têm os mesmos objetivos evidenciados para o serviço; o socket 1 realiza as comunicações síncronas e o socket 2 é utilizado para as comunicações assíncronas.

O bloco que se segue ao DGV é referente ao tratamento das primitivas OpenCL. Naturalmente, nem todas as primitivas têm a necessidade de comunicar com o serviço daemon; neste caso as computações passam diretamente para o *runtime* OpenCL, evitando-se a utilização dos canais de comunicação. Contudo, o socket 2 poderá ter sempre alguma utilização, com a perspetiva de atender aspetos assíncronos do OpenCL, sem intervenção da thread main. Para a manipulação do socket 2, usaram-se POSIX threads, por forma a suportar um complexo conjunto de eventos assíncronos do OpenCL. Estes eventos estão devidamente mapeados em memória através de uma tabela de *hash*, em que cada posição da tabela tem um apontador para a região de memória usada pelo evento a tratar e o respetivo identificador.

A criação da thread é efetuada com o procedimento bem conhecido das POSIX thread pthread_create:

- pthread_create(&handler_thread, NULL,callback_handler_th,NULL)

O resultado da criação do thread envolve a vinculação da região de memória apontada por *callback_handler_th*, na qual se encontra a rotina onde irá ser realizado o processo de comunicação e o tratamento das vertentes assíncronas do OpenCL. O processo comunicativo dentro do thread é um pouco diferente das abordagens até aqui analisadas. Enquanto a rotina *callback_handler_th* é executada concorrentemente ao resto do processo da biblioteca, esta fica em modo *standby* até que seja detetado algum tipo de comunicação no

socket 2. No processo de inicialização do thread é efetuada a recolha de todas as portas vinculadas ao funcionamento assíncrono.

```
for(d=0; d<daemon_count; d++){
        FD_SET(daemon_sckt_d2[d], &read_fds);
        maxfd = max(maxfd, daemon_sckt_d2[d]);
}
```

O ciclo é necessário para percorrer as portas individualmente armazenadas num bloco de memória contínuo. Procede-se até encontrar o descritor com o valor mais elevado relativamente aos demais. O thread é mantido num estado estacionário à espera de comunicações efetuadas através das portas de cada um dos daemons.

```
while(select(maxfd + 1, &read_fds, NULL, NULL, NULL) != -1){
        for(d=0; d<daemon_count; d++)
                if(FD_ISSET(daemon_sckt_d2[d], &read_fds)){
                        sckt_d2 = daemon_sckt_d2[d];
                        break;
                }
        read(sckt_d2, &callback_type, sizeof(int));
        do_callback[callback_type](sckt_d2);
}
```

A função *select* dentro do ciclo *while*, descrita no código em cima, é a condição que torna a thread estacionária. O *select* bloqueia o thread até que seja detetado algum fluxo de comunicação dentro da gama dos descritores socket, sendo o descritor maior representado por maxfd +1. Entretanto, quando é detetado algum tipo de comunicação na gama de sockets, é necessário extrair o socket em causa através de um ciclo que percorre a região de memória onde as portas então alojadas e verifica com a rotina *FD_ISSET* a porta efetiva do daemon que desencadeou a comunicação. Depois de efetuada a descoberta, procede-se à leitura da porta extraindo o identificador proveniente do daemon, com a

finalidade de identificar a rotina assíncrona a ser tratada.

A particularidade deste modelo está no processo de comunicação; todo o processo é efetuado através de sucessivos *read* e *write*.

3.3.3 daemon clOpenCL Open-MX

A natureza do daemon construído para suportar a utilização do Open-MX não é completamente diferente da natureza do daemon TCP; a filosofia de construção cliente-servidor mantém-se. A abordagem metodologicamente seguida é idêntica à do modelo anterior.

O processo referente à criação do daemon sobre o Open-MX é demonstrado na Figura 3.6. No decorrer do processo é chamada a rotina que efetua o teste de integridade do OpenCL, rotina já abordada no modelo TCP. Os procedimentos seguintes são referentes ao comportamento do Open-MX. Antes que outras funções Open-MX possam ser chamadas, a biblioteca tem que ser inicializada através da função *omx_init*.

É visível a ausência de um diretório como evidenciado no modelo anterior. Neste modelo, o cliente é que vai ao encontro do serviço, sendo desnecessário providenciar um mecanismo para futuramente o serviço ser descoberto pelo cliente. Comparativamente ao modelo de sockets, o Open-MX abre um canal de comunicação que pode ser visto como um descritor socket, mas o processo de criação é ligeiramente diferente.

- omx_open_endpoint(OMX_ANY_NIC, OMX_ANY_ENDPOINT, FILTER, NULL, 0,&endpoint)

O processo de criação é realizado apenas com recurso à função omx_open_endpoint. Ao contrário do modelo socket, onde é necessário vincular o socket a um endereço físico de rede, o Open-MX mantém um serviço ativo no qual apresenta uma tabela onde descreve o NIC referente ao hardware de rede e o seu respetivo endpoint. O primeiro parâmetro descreve precisamente o NIC pretendido para a vinculação do endpoint; através da constante *OMX_ANY_NIC* definida pelo próprio Open-MX o programador não necessita conhecer

endereços, pois a API verifica a tabela descrita anteriormente e associa o primeiro NIC encontrado. Para além desta facilidade introduzida pelo Open-MX, o parâmetro *FILTER*, é também uma novidade comparando com o modelo socket; este parâmetro torna possível distinguir as mensagens passadas pelo *enpoint*. Apesar de o Open-MX estar preparado para abrir múltiplos endpoints em cada NIC, o uso de um filtro permite que um único endpoint seja usado para múltiplos fins; a comunicação entre dois pontos exige que ambos conheçam e usem o mesmo filtro.

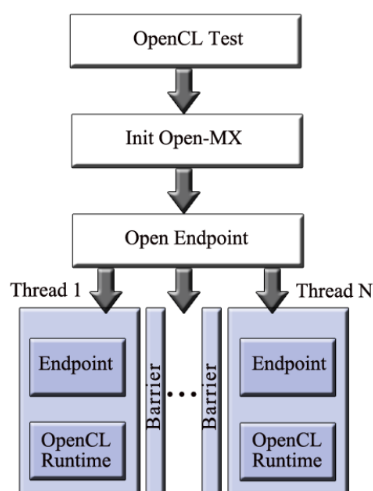

Figura 3.6: clOpenCL Runtime do modelo Open-MX.

Outra diferença visível no esquema da Figura 3.6 é o tratamento das mensagens. Enquanto no modelo socket TCP o transporte dos pedidos, para o próprio processamento no *runtime* do OpenCL, é efetuado através do processo filho gerado pelo processo principal do serviço daemon, neste modelo é usada uma abordagem multithread, em que cada thread tem a tarefa de analisar cada pedido efetuado e transpô-lo para o OpenCL. O resultado será a independência interprocessual de cada cliente, tornando a abordagem escalável ao ponto de instituir um ambiente onde é possível que dois clientes utilizem o clOpenCL. O ponto negativo na abordagem multicliente é a necessidade de espera enquanto uma thread está a ser executada. Isto acontece pelo simples facto de não haver maneira de controlar os processos do OpenCL. O OpenCL não está devidamente preparado para

dividir trabalho por múltiplos processos, não proporcionando um mecanismo que efetue um rastreamento dos *work-items* ocupados. Como tal, foi implementado, no modelo Open-MX, um conjunto de barreiras construidas com os mutexes e localizadas entre a execução das threads, como é apresentado na Figura 3.6.

A receção e envio de dados é efetuada através das primitivas *omx_irecv* e *omx_isend*, que exibem um comportamento assíncrono, seguindo-se o uso da primitiva *omx_wait*, que permite aguardar pela conclusão da receção ou envio.

```
void _ccl_omx_recv_packet(void *data, size_t max_length, uint64_t tag,
    uint64_t mask, omx_status_t *status){
        uint32_t result;
        omx_request_t request;
        omx_status_t _status, *__status;
        __status = (status == NULL) ? &_status : status;
        if(omx_irecv(_ccl_local_endpoint, data, max_length, tag, mask, NULL,
            &request) != OMX_SUCCESS)
                _ccl_perror_and_exit("omx_irecv");
        if(omx_wait(_ccl_local_endpoint, &request, __status, &result,
            OMX_TIMEOUT_INFINITE) != OMX_SUCCESS)
                _ccl_perror_and_exit("omx_wait");
}

void _ccl_omx_send_packet(void *data,size_t length,omx_endpoint_addr_t
    addr,uint64_t tag){
        uint32_t result;
        omx_request_t request;
        omx_status_t status;
        if(omx_issend(_ccl_local_endpoint, data, length, addr, tag, NULL,
            &request))
                _ccl_perror_and_exit("omx_issend");
```

```
        if(omx_wait(_ccl_local_endpoint, &request, &status, &result,
            OMX_TIMEOUT_INFINITE))
                _ccl_perror_and_exit("omx_wait");
}
```

Relativamente aos parâmetros das primitivas de receção e envio, o primeiro parâmetro é a estrutura de dados que irá receber o pacote de dados enviados pelo cliente. A estrutura de dados comum a todas as leituras é a seguinte:

```
struct{
        char id;
        char data[MAX_REQ_PACKET_SIZE];
} ccl_request;
```

Esta estrutura tem como objetivo encapsular os dados enviados pelo cliente, que são estruturados pela biblioteca, juntamente com um *id*. O identificador *id* indica a primitiva OpenCL a ser tratada. Ainda na thread do daemon, é guardado numa região de memória contínua o apontador para a rotina correspondente ao *id* enviado pela biblioteca. O processo de desempacotamento é efetuado da seguinte forma:

```
struct{
                char id;
                cl_uint num_entries;
                char plat_null;
        } *ccl_request=request;
```

Por exemplo, no seguimento de um levantamento de plataformas, em que são necessários os respetivos parâmetros para executar a primitiva, a thread recebe o pacote e efetua o destacamento para a rotina onde se encontra a resolução da primitiva identificada pelo id. A rotina recebe como parâmetro a estrutura com os dados *request*, e faz-se com que o apontador para a estrutura de dados aponte para o *request*; assim o número de

bytes é dividido consoante as exigências de cada identificador. Posto isto, a estrutura de dados que contém a rotina tem que encaixar de forma exata com a estrutura de dados do lado da biblioteca.

Ainda relativamente à rotina de receção, o parâmetro *TAG_REQUEST* é o filtro do Open-MX adjacente ao serviço e à biblioteca para validação de mensagens e o parâmetro *ALL_EXCEPT_SBTG* é a mascara de 32 bits, para seleção de mensagens provenientes de outros endpoints remotos. Esta máscara oferece um mecanismo de autenticação, por parte da biblioteca, no caso de esta estar a ser utilizada por múltiplas instâncias. Em suma, o daemon vai ter a necessidade de descobrir todos os elementos específicos do endpoint remoto que efetuou a comunicação. Para atingir esse fim é ainda usado o parâmetro *status* na rotina de receção. O *status* é uma estrutura do Open-MX, que é devidamente preenchida na receção de pacotes; com o parâmetro *status* é possível extrair informação, como o endpoint do emissor e o respetivo filtro. Todo este processo é repetido nas manipulações das primitivas OpenCL, apresentando-se de seguida o exemplo da primitiva _cclGetPlatformIDs.

- void _cclGetPlatformIDs(void *request,omx_endpoint_addr_t addr,uint32_t subtag)

A sua parametrização está em torno dos dados necessários para a execução da primitiva em causa —parâmetro request —e dos dados necessários para endereçar a resposta ao cliente —parâmetros addr e subtag.

- omx_decompose_endpoint_addr(addr, &remote_nic_id, &remote_endpoint_id)
- omx_connect(local_endpoint,remote_nic_id,remote_endpoint_id, FILTER_OPENCCL, TIMEOUT_CONNECT, &remote_endpoint_addr)

Estas duas funções Open-MX são executadas para efetuar a vinculação ao endpoint remoto. Este processo é apenas realizado uma única vez e é remetido pela biblioteca. Na extração, do endereço por vias do parâmetro status, na rotina de receção, é usada a rotina

Open-MX *omx_decompose_endpoint_addr* que permite descobrir o endpoint e o NIC do respetivo endereço passado no parâmetro addr. Em seguida à descoberta do NIC remoto e o endpoint remoto, é efetuada a vinculação dirigida pela rotina Open-MX *omx_connect*.

3.3.4 Biblioteca clOpenCL Open-MX

A biblioteca desenhada em Open-MX tem diferenças consideráveis relativamente à biblioteca desenhada sobre os socket TCP. Para além dos aspetos ligados à própria implementação, ao nível de comunicações, a abordagem multithread é já um aspeto que induz uma séria mudança.

Relativamente aos aspetos construtivos, há um conjunto de passos idênticos ao daemon Open-MX. Todo o processo é vincado através da captação da main da aplicação que faz uso da biblioteca clOpenCL. Os primeiros três passos demonstrados na Figura 3.7

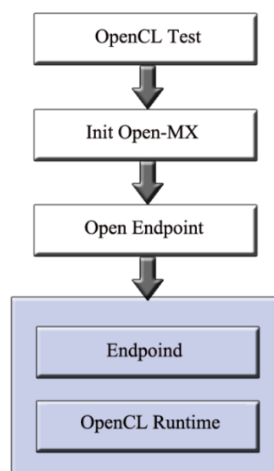

Figura 3.7: clOpenCL biblioteca Runtime do modelo Open-MX.

correspondem ao processo normal de inicialização de recursos e testes de integridade já anteriormente abordados. O conjunto de manipulações para a construção do endpoint já foi abordado na apresentação do deamon. O que é desconhecido é o processo de reconhecimento do serviço, por parte da biblioteca. A necessidade explícita introduzida

pelo modelo Open-MX de efetivar uma ligação ponto-a-ponto, ou seja, a procura de um parceiro geograficamente disperso é eminente. Daí se ter implementado uma solução que executa um plano de descoberta dos daemons ativos, para operacionalizar o Diretório Global de Visibilidade.

Esta solução é baseada na utilização da tabela Open-MX que cada nó computacional detém num ambiente HPC. Em cada nó que contenha o serviço Open-MX a correr é possível executar o binário *omx_info*, disponível na instalação do Open-MX, o qual descreve o número de nós que contêm o serviço Open-MX a correr com o respetivo NIC e a designação do nó. A Tabela 3.1 é o exemplo do que pode ser obtido a partir do binário *omx_info*; neste caso particular, encontram-se 4 nós a correr o Open-MX.

Tabela 3.1: Representação da tabela Open-MX em cada nó.

	NIC	Nó Computacional
0	00:00:5a:9c:33:8e	compute-4-2.local:0
1	00:00:5a:9b:1f:c4	compute-4-1.local:0
2	00:00:5a:9b:49:8a	compute-4-3.local:0
3	00:00:5a:9b:1f:d2	compute-4-0.local:0

Este mecanismo permite identificar o número de nós que contem o serviço e o respetivo NIC. Contudo, fica por responder a questão do endpoint. Não basta apenas efetuar a descoberta do NIC, sendo também necessário descobrir se algum destes NIC possui algum endpoint e se se este encaixa no perfil comunicativo do clOpenCL. No intuito de descobrir o endpoint em cada NIC, o Open-MX também dispõe de um binário para esse efeito. O binário é designado de *omx_endpoint_info* e tem como objetivo listar o número de endpoints abertos no momento. Ambos os binários apresentados são interpretados pela *bash*, em cada nó daí a necessidade que criar um mecanismo de leitura deste binário dentro do espaço de endereçamento da biblioteca. Estas necessidades levaram ao desenvolvimento de uma script produzida na linguagem Python. Para tornar funcional a inserção do resultado da script no mesmo espaço de endereçamento do clOpenCL, foi utilizado um canal de comunicação, designado de *pipe*, com o processo gerado pela script.

A rotina que desencadeia o processo de vinculação do endpoint aos demais daemos é a *connect_to_daemons*. Esta rotina tem o objetivo de proceder à união de todos os elementos analisados anteriormente. A abertura do *pipe* para a realização das comunicações inter-processos é procedida com a função fopen, com os parâmetros nome da script em Python e modo de leitura. A script contém duas rotinas, nas quais estão definidos os tratamentos dos binários Open-MX referidos anteriormente. A função *omx_hosts* é definida para extrair a informação da tabela Open-MX e organiza-la de forma a poder ser processada. A função é descrita da seguinte forma:

```
def omx_hosts():
        hosts =[]
        (status, output) =
            commands.getstatusoutput('/opt/open-mx/bin/omx_info')
        if status == 0:
                start = 0
                for line in output.splitlines():
                        if start == 1:
                                hosts.append((line.split()[1],
                                    line.split()[2].split(':')[0]))
                        elif line == '====================================':
                                start = 1
        return hosts
```

É utilizada a função getstatusoutput, disponível na linguagem Python, para executar o binário *omx_info* na Shell local. O resto do procedimento corresponde à separação dos dados úteis e ao armazenamento no array de dados *hosts*. Depois de efetuada a recolha do conteúdo descrito na tabela do Open-MX, é efetuada a análise em cada um dos NIC respetivos, para verificar a existência de algum endpoint efetivo para comunicações. A rotina que efetua esse tratamento é *omx_find_endpoints*, descrita da seguinte forma:

```
def omx_find_endpoints(host, executable):
```

```
    endpoints = []
    (status, output) = commands.getstatusoutput('ssh %s
       /opt/open-mx/bin/omx_endpoint_info'%(host))
    if status == 0:
            start = 0
            for line in output.splitlines():
                    if start == 1:
                            if line.split()[5] == '(%s)'%(executable):
                                    endpoints.append(line.split()[0])
                    elif line =='======================================':
                            start = 1
    return endpoints
```

O binário *omx_endpoint_info* é um processo local, ou seja, a sua efetividade apenas está direcionada ao sistema local. Posto este facto, com a execução local deste binário não seria possível obter a informação pretendida acerca dos nós adjacentes. A fim de obter um resultado amplo com a manipulação do binário, este é marcado como alvo de execução no respetivo host transportado através do mecanismo Secure Shell. Assim, é analisado em cada host, através de um procedimento remoto, a sua disponibilidade de abarcar comunicações com a biblioteca. O corpo seguinte da rotina *omx_find_endpoint* efetua a interpretação do output de maneira a tornar legível o endpoint capturado através do binário. Um aspeto muito importante neste processo de levantamento é distinguir o originário do endpoint, com a comparação do serviço que se quer filtrar. Para este efeito, introduz-se descritivamente o endpoint pelo qual se quer efetuar a vinculação e o respetivo serviço.

Estas duas rotinas são combinadas da seguinte forma, na script Python:

```
host_list = omx_hosts()
for (addr, host) in host_list:
        endpoint_list = omx_find_endpoints(host, 'ccl_daemon')
```

```
    for endpoint in endpoint_list:
        print '0x%s; %s'%(addr.replace(':', ''), endpoint)
```

No seguimento da execução da script é efetuada a interpretação dos resultados, na biblioteca clOpenCL, na linguagem C, e a vinculação do endpoint da biblioteca aos endpoints dos daemon. O conjunto de procedimentos é descrito nas seguintes linhas de código:

```
while(fscanf(fp, "%lx; %d\n", &remote_nic_id, &remote_endpoint_id) > 0){
    daemon_info = realloc(daemon_info, sizeof(struct _daemon_info) *
        ++count);
    if(omx_connect(_ccl_local_endpoint, remote_nic_id, remote_endpoint_id,
        FILTER_OPENCCL, TIMEOUT_CONNECT, &remote_endpoint_addr))
            _ccl_perror_and_exit("omx_connect");
    daemon_info[count-1].addr = remote_endpoint_addr;
    daemon_info[count-1].nic_id = remote_nic_id;
    daemon_info[count-1].endpoint_id = remote_endpoint_id;
    mysubtag = _omx_next_subtag();
    _ccl_omx_send_packet(&ccl_request, sizeof(ccl_request),
        daemon_info[count-1].addr, TAG_REQUEST|mysubtag);
    _ccl_omx_recv_packet(NULL, 0, TAG_REPLY|mysubtag, ALL_BITS, NULL);
}
```

Esta interpretação é efetuada dentro de um ciclo, com uma leitura inter-processo, através do *file pointer* resultante da abertura do *pipe*, ou seja, o endpoint remoto e o NIC remoto provenientes da script em Python são tratados sequencialmente. Nas anteriores linhas de código está representada uma estrutura de dados *daemon_info* declarada numa zona de memória global, com o objetivo de armazenar informações vitais à comunicação com os endpoint remotos, para futuramente estabelecer vinculações aos objetos manipulados do

OpenCL. Uma vez descobertos todos os elementos referentes à comunicação sobre o Open-MX, é efetuada através da primitiva *omx_connect* a vinculação do endpoint produzido na biblioteca com os demais endpoints e ainda com o filtro *FILTER_OPENCCL*.

Ainda no mesmo seguimento é apresentado o mecanismo que garante uma autenticidade na abordagem multi-processo. Quando existe uma necessidade do clOpenCL ser dividido por vários processos, é oferecida segurança em termos de fluidez de dados, sem que haja colisões nos endpoints possivelmente provocadas pelas comunicações cruzadas dos processos. O mecanismo está dividido em duas partes. A primeira parte está do lado do daemon e trata-se da abordagem efetuada na vinculação dos endpoints. Esta abordagem compreende a manipulação de filtros dentro de uma gama de valores nos quais as mensagens estão compreendias, provenientes do lado da biblioteca. A segunda parte cabe à biblioteca, onde são definidos os filtros de comunicação; antes de efetuar qualquer comunicação é adicionado ao filtro base um conjunto de bits produzido pela seguinte função:

```
uint32_t _omx_next_subtag(){
       uint32_t st;
       pthread_mutex_lock(&register_mutex);
       st = ++subtag;
       pthread_mutex_unlock(&register_mutex);
       return(st);
}
```

Com o auxílio do mutex é obtido um filtro único, originado antes de estabelecer qualquer tipo de comunicação efetuada por parte da biblioteca.

O processo comunicativo através da rede é despoletado quando são usadas plataformas OpenCL remotas. Se apenas for usada a plataforma local, como já referido em capítulos prévios, através do levantamento das plataformas OpenCL, as comunicações na rede não são efetuadas e as computações exigidas pelo utilizador são redirecionadas de forma direta

da camada biblioteca clOpenCL para a própria camada OpenCL. Ainda assim, a biblioteca clOpenCL estabelece a ligação aos daemons através dos endpoints abertos, apesar de depois não existirem comunicações pela rede.

Estruturalmente, a nível de comunicações, todas as primitivas são tratadas através de pacotes de comunicação, quer para receção quer para envio, e os pacotes são construídos consoante as necessidades de cada primitiva OpenCL. Recorrendo à representação da construção dos pacotes da primitiva clGetPlatformIDs:

```
struct{
        char id;
        cl_uint num_entries;
        char plat_null;
} ccl_request={.id=0x01,.num_entries=num_entries,.plat_null='n'};
struct{
        cl_int result;
        cl_uint num_platforms;
        cl_platform_id platform[MAX_NUM_PLATFORMS];
} ccl_reply;
```

Esta primitiva contem três parâmetros: um parâmetro de entrada e dois de saída. Para a execução remota é necessário enviar o parâmetro de entrada juntamente com o identificador da função a exercer no lado do daemon. O pacote com este conjunto de campos é representado pela estrutura *ccl_request*. O *ccl_request* é enviado para o respetivo daemon para ser tratado consoante os parâmetros enviados. De seguida é necessário obter os resultados referentes à execução da primitiva no daemon. Estes resultados —os parâmetros de saída da primitiva —uma vez concluída a execução da primitiva no daemon, são enviados num pacote de envio com o mesmo conjunto de bytes da estrutura *ccl_reply*. Genericamente, a estrutura *ccl_reply* apresenta o resultado devolvido pela execução da primitiva e os demais parâmetros de saída que são preenchidos. A estrutura dos pacotes de envio e de receção de todas as primitivas tratadas no clOpenCL é genericamente a representada

em cima.

3.4 Objetos clOpenCL

Até ao momento apenas nos debruçámos sobre a descrição das tecnologias usadas na compactação das comunicações sobre a rede e todos os seus mecanismos em volta das inicializações, para proceder à tarefa de tornar o OpenCL operacional num ambiente cluster. Nesta secção irá ser descrito como os objetos do OpenCL são manipulados e todos os seus mecanismos oferecidos por ambos os modelos do clOpenCL para tal efeito. Conceptualmente, a manipulação dos objetos OpenCL é idêntica em ambos os modelos. A diferença é notável ao nível de programação.

3.4.1 Espaço de Endereçamento

No espaço de memória da biblioteca clOpenCL é alocada uma zona de memória onde são mantidos os objetos OpenCL, com a informação correspondente. Para tornar possível a manipulação de objetos OpenCL, o clOpenCL faz uso de uma zona de memória específica, que contém uma sequência de bytes, traduzidos em campos identificativos do objeto OpenCL. No caso particular do modelo Open-MX, o registo do objeto é efetuado com o seguinte procedimento:

```
void * register_object(void *object, omx_endpoint_addr_t addr, uint64_t
    nic_id, uint32_t endpoint_id){
        int i;
        pthread_mutex_lock(&register_mutex);
        for(i=0; i<num_registered_objects; i++)
                if((registered_object[i].object == object) &&
                    (registered_object[i].nic_id == nic_id) &&
                    (registered_object[i].endpoint_id == endpoint_id))
```

```
                    break;
        if(i == num_registered_objects){
                if(num_registered_objects == max_registered_objects){
                        max_registered_objects += 128;
                        registered_object = realloc(registered_object,
                            sizeof(struct _registered_object) *
                            max_registered_objects);
                }
                registered_object[i].object = object;
                registered_object[i].addr = addr;
                registered_object[i].nic_id = nic_id;
                registered_object[i].endpoint_id = endpoint_id;
                num_registered_objects++;
        }
                pthread_mutex_unlock(&register_mutex);
        return((void *)(long)i);
}
```

No caso do modelo clOpenCL Open-MX, o registo é efetuado consoante as necessidades que o Open-MX possui para desencadear as comunicações. A diferença relativamente ao modelo clOpenCL TCP são apenas os elementos essenciais à comunicação. O tratamento da biblioteca para suportar múltiplos fios de execução está explícito na utilização do mutex *register_mutex*. Na operação de registo são armazenados o apontador real do objeto OpenCL, o endereço físico da rede, o NIC e o endpoint, usando-se, para tal efeito, um array dinâmico.

O ciclo representado no excerto de código é utilizado para autenticar o objeto na zona de memória global da biblioteca clOpenCL e encontrar o identificador para o novo registo. A estrutura de dados que representa a zona de memória global, onde são armazenados os objetos OpenCL e os respetivos elementos de comunicação, só é alocada

conforme as necessidades de utilização. Com esta abordagem evitam-se alocações de memória desnecessários. Na finalização do processo de registo é devolvido um apontador com propriedades de identificação do objeto na estrutura de dados. O apontador originado pela biblioteca explicita o resultado do processo de registo e corresponde ao índice do array usado para armazenar os objetos. É utilizado como substituto do verdadeiro apontador, devolvido pelo OpenCL em tempo de execução da aplicação. Consequentemente, numa futura utilização desse apontador, consegue-se facilmente obter a informação da localização do objeto em causa. O registo está diretamente ligado com a execução das primitivas OpenCL.

3.4.2 Tratamento do clGetPlatformIDs

Quando é estabelecida a *linkagem* de uma aplicação OpenCL à biblioteca clOpenCL, todas as primitivas suportadas são tratadas no momento de interceção, sendo efetuado o encaminhamento para o nó adequado. A fim de efetuar o levantamento das plataformas do OpenCL é utilizada a primitiva clGetPlatformIDs; todos os objetos de plataforma capturados localmente e em cada nó computacional onde se encontra em execução o serviço daemon são extraídos e armazenados dentro do espaço de memória da biblioteca clOpenCL. O procedimento para levar a cabo o levantamento das plataformas é executado em duas partes: a primeira localmente e a segunda em cada um dos nós remotos. A execução local é efetuada com as seguintes linhas de código:

```
if((ccl_reply.result = __real_clGetPlatformIDs(num_entries, platforms,
    &_num_platforms)) != CL_SUCCESS)
                return(ccl_reply.result);
        num_returned_platforms = min(_num_platforms, num_entries);
        if(platforms != NULL){
                for(p=0; p<num_returned_platforms; p++)
                        platforms[p] = register_object(platforms[p], NO_ADDR,
                            NO_NIC_ID, NO_ENDPOINT_ID);
```

```
        ccl_request.num_entries -= num_returned_platforms;
    }
```

Na execução local, os objetos plataforma devolvidos pela primitiva clGetPlatformIDs também são tratados no clOpenCL; com a execução da primitiva obtêm-se os objetos de plataforma locais, que são armazenados no espaço de endereçamento da biblioteca. No processo de registo, o apontador para as plataformas referenciado pela variável platforms é substituído por um "apontador falso" do processo de registo e para fins de identificação local essas plataformas são registadas com o conjunto de indicadores pré-definidos na biblioteca —NO_ADDR, NO_NIC_ID e NO_ENDPOINT_ID.

Se o número de plataformas obtidas localmente for inferior ao número de entradas definido pelo utilizador na primitiva clGetPlatformIDs, então é iniciada a recolha das plataformas nos daemon remotos. A recolha de plataformas remotas é efetuada da seguinte forma:

```
for(d=0; d<daemon_count; d++){
    mysubtag = _omx_next_subtag();
    _ccl_omx_send_packet(&ccl_request, sizeof(ccl_request),
        daemon_info[d].addr, TAG_REQUEST|mysubtag);
    _ccl_omx_recv_packet(&ccl_reply, sizeof(ccl_reply),
        TAG_REPLY|mysubtag, ALL_BITS, NULL);
    if(ccl_reply.result != CL_SUCCESS)
        return(ccl_reply.result);
    _num_returned_platforms = min(ccl_reply.num_platforms,
        ccl_request.num_entries);
    for(p=0; p<_num_returned_platforms; p++)
        if(platforms != NULL)
```

```
                    platforms[num_returned_platforms++] =
                        register_object(ccl_reply.platform[p],
                        daemon_info[d].addr, daemon_info[d].nic_id,
                        daemon_info[d].endpoint_id);
        ccl_request.num_entries -= _num_returned_platforms;
        if(num_platforms != NULL)
                *num_platforms += ccl_reply.num_platforms;
}
```

Para efeito de recolha de plataformas remotas é definido um ciclo que percorre todos os daemon registados pela biblioteca. No processo de registo, as plataformas remotas são registadas conforme as necessidades de preenchimento de entradas, ou seja, as plataformas obtidas a partir de um daemon, são registadas uma a uma e o número de entradas é decrementado conforme efetuado o registo. Apenas é contactado o daemon seguinte se existirem entradas disponíveis.

Uma vez estabelecida a relação com os objetos plataforma, nos demais objetos não existe a necessidade intrínseca de percorrer todos os nós computacionais à procura dos seus respetivos dados; basta apenas efetuar o processo de registo para tal efeito. Exemplificando, a obtenção dos dispositivos com a primitiva clGetDevicesIDs está estritamente ligada ao objeto Plataforma, pelo que, para a identificação e registo é apenas necessário recorrer à região de memória com o índice do apontador da plataforma a fim de identificar a localidade de execução. Depois de efetuada a localização e executada a primitiva, o resultado do objeto Dispositivo é registado com a função de registo. Todos os demais objetos são tratados de igual forma.

3.5 Epílogo

O presente capítulo apresenta a abordagem que foi usada para a resolução da integração do OpenCL no clOpenCL. Demonstrou-se os aspetos peculiares na integração das duas bibliotecas e todos os processos relevantes na interação clOpenCL com o OpenCL. A especial atenção manteve-se nos mecanismos de ambos os modelos clOpenCL, do ponto de vista de programação, abordando-se assim o conjunto de tecnologias de comunicação utilizadas em ambos os modelos.

Capítulo 4

Aspetos no desenvolvimento do clOpenCL

O desejável seria a não manipulação do produto OpenCL, mas, com as divergências notadas ao longo do desenvolvimento, tal não foi possível. Este capítulo trata dos aspetos formalizados face à resolução do conjunto de primitivas OpenCL, que envolveram operações com relevo suficiente para serem abordadas no domínio do clOpenCL.

4.1 Plataformas com o clOpenCL

Existe uma série de objetos mínimos do OpenCL que é necessariamente obrigatório manipular na realização de uma aplicação OpenCL. O objetivo seguido no desenho do clOpenCL, foi oferecer um conjunto mínimo de primitivas do OpenCL, que permitisse a execução de aplicações. Os objetos focados foram os objetos Plataforma, Dispositivo, Contexto, Command Queue, Programa, Kernel e Buffer. É também referido neste trabalho que, para a criação de cada objeto, o OpenCL oferece caminhos diferentes. Apesar dessa vertente do OpenCL, no trabalho aqui abordado, é seguido um conjunto mínimo de primitivas que permite a execução de uma aplicação OpenCL. O peso no desenvolvimento

do clOpenCL encontra-se na resolução remota das primitivas, pois a resolução local das primitivas é uma passagem direta para o OpenCL.

4.1.1 Disposição

O ambiente de programação do clOpenCL apresenta uma filosofia equivalente ao OpenCL; as diferenças encontram-se ao nível da execução. Na realidade, o OpenCL é uma API desenhada para ser executada num sistema computacional isolado, enquanto o clOpenCL abrange a mesma filosofia adicionalmente à vertente remota. Por exemplo, na extração de todas as plataformas existentes no sistema, através do OpenCL ou com o clOpenCL com apenas um daemon ativo, o resultado é demonstrado na Tabela 4.1.

Tabela 4.1: Disposição das Plataformas no OpenCL e no clOpenCL.

Plataforma	OpenCL	clOpenCL
1	AMD Accelerated Parallel Processing	AMD Accelerated Parallel Processing
2	NVIDIA CUDA	NVIDIA CUDA
3		AMD Accelerated Parallel Processing
4		NVIDIA CUDA

Verifica-se no clOpenCL, que a descrição das plataformas são duplicadas, e cada uma delas terá um papel diferente, ou seja, mesmo que tenham a mesma descrição são tratadas como objetos distintos. A abordagem assim definida toma uma posição diferente ao OpenCL, a vinculação da plataforma dita toda a linha de recursos abrangida, que vão desde a recolha à execução de trabalho nos dispositivos. De facto, o OpenCL usa apenas os recursos de uma única plataforma, enquanto que o clOpenCL tem uma abordagem multiplataforma —multinó.

4.1.2 Localização Física

Os aspetos no desenvolvimento do clOpenCL estão estritamente ligados à concordância dos recursos abrangidos pelas plataformas. Face a este paradigma, houve necessidade de explicitar o nó computacional ao qual está associada determinada plataforma. Para esse efeito foi estendida a primitiva clGetPlatformIDs do OpenCL com um atributo especial de recolha de informação CL_PLATFORM_HOSTNAME [AA12].

4.2 Tratamento do Objeto Contexto

Com o clOpenCL não é possível misturar recursos de diferentes plataformas, mesmo que estas sejam identificadas pelo mesmo fornecedor. Esta metodologia não significa uma limitação mas sim a integração da metodologia do próprio OpenCL. Os dispositivos extraídos de diferentes plataformas não podem ser associados num mesmo contexto, de forma a partilhar recursos entre eles. Estes têm que ser necessariamente contextualizados de forma separada para o efeito.

O processo de contextualização dos dispositivos pode ser realizado através de uma das duas primitivas fornecidas pelo OpenCL. A primitiva clCreateContext é utilizada quando se faz o levantamento dos dispositivos de uma dada plataforma. Em contrapartida, na primitiva clCreateContextFromType só é necessário passar explicitamente, através do primeiro parâmetro, o objeto plataforma, de forma a estabelecer uma associação aos seus dispositivos.

- cl_context clCreateContext(const cl_context_properties *, cl_uint, const cl_device_id *, void (CL_CALLBACK *)(const char *, const void *, size_t, void *), void *, cl_int *)

- cl_context clCreateContextFromType(const cl_context_properties *, cl_device_type, void (CL_CALLBACK *)(const char *, const void *, size_t, void *), void *, cl_int *)

4.2.1 Localização Objeto Plataforma

O clOpenCL oferece suporte para as primitivas *clCreateContextFromType* e *clCreateContext*. A implementação das duas primitivas envolve o cumprimento da tabela 4.4 da especificação OpenCL 1.1, na qual se realça que o apontador para as propriedades apenas aponta para um elemento com a respetiva plataforma. É com recurso a essa plataforma que se faz a extração da localização da plataforma.

```
If(properties != NULL)
       while(properties[num_properties+=2] != 0);
if((num_properties > MAX_NUM_PROPERTIES-1) || (num_devices > MAX_NUM_DEVICES))
       ccl_reply.errcode_ret = CL_OUT_OF_HOST_MEMORY;
else{
       for(p=0; p<num_properties; p+=2){
              ccl_request.properties[p] = properties[p];
              if(properties[p]== CL_CONTEXT_PLATFORM)

              ccl_request.properties[p+1] = (cl_context_properties)
                 registered_object[(int)(long)properties[p+1]].object;
       }
}
```

O excerto de código em cima representado, apresenta a característica de ambas as primitivas na extração da localização da plataforma. Diretivas como *MAX_NUM_PROPERTIES* são inicializadas no clOpenCL para fins de controlo de código conduzido pelo utilizador. Neste caso, o *MAX_NUM_PROPERTIES* impede que o utilizador fuja da especificação OpenCL e se assim o fizer é redirecionado para o erro conhecido pelo espaço de nomes do OpenCL *CL_OUT_OF_HOST_MEMORY*. O ciclo é utilizado para percorrer a região de memória apontada pelo apontador *properties*, com o intuito de descobrir o objeto de plataforma associado à execução das primitivas.

4.2.2 Tradução do Objeto clOpenCL para OpenCL

Uma vez que os apontadores reais dos objetos OpenCL são substituídos pelos apontadores do clOpenCL no processo de *wrapping*, é necessário recuperar o apontador real do objeto OpenCL. Este processo é efetuado com uma consulta à região de memória referenciada pelo apontador falso. Este passo não é tão crítico para a primitiva clCreateContext como para a clCreateContextFromType, uma vez que esta depende única e exclusivamente desta tradução para a vinculação efetiva de todos os dispositivos da plataforma passada pelo parâmetro *properties*, enquanto o clCreateContext obriga a identificar os dispositivos que são passados pelo parâmetro *devices*.

Na primitiva clCreateContext a identificação e a tradução do apontador falso para o real são efetuadas com a seguinte operação:

```
for(d=0; d<num_devices; d++){
        addr = registered_object[(int)(long)devices[d]].addr;
        remote_nic_id = registered_object[(int)(long)devices[d]].nic_id;
        remote_endpoint_id =
            registered_object[(int)(long)devices[d]].endpoint_id;
        if((remote_nic_id == NO_NIC_ID) && (remote_endpoint_id ==
            NO_ENDPOINT_ID))
                num_local_dev++;
                ccl_request.devices[d] =
                    registered_object[(int)(long)devices[d]].object;
}
```

Os elementos representativos na comunicação são extraídos individualmente, identificando-se a existência de algum dispositivo local dentro do array local representado pelo apontador *devices*. O clOpenCL dá prioridade ao dispositivo local; caso haja uma mistura de dispositivos de diferentes plataformas evita-se proceder à comunicação para executar o código, uma vez que o clOpenCL não permite a criação de contextos com dispositivos de

diferentes plataformas e nós.

A primitiva *clCreateContextFromType* tem um comportamento diferente; a localização baseia-se no objeto plataforma passado pelo parâmetro *properties*. Os objetos dispositivos são vinculados através de um processo automático do OpenCL, que exige uma especial atenção por parte do clOpenCL. Com a invocação do *clCreateContextFromType*, os dispositivos sofrem uma associação implícita com o contexto, o que implica a existência de um processo implicitamente intrínseco na separação do contexto e dispositivo. Para efeito, o OpenCL especificou a primitiva clGetContextInfo, com o objetivos de extrair os dispositivos OpenCL.

- cl_int clGetContextInfo(cl_context, cl_context_info, size_t, void *, size_t *)

4.2.2.1 Processo de Pesquisa

Nas primitivas como o *clGetContextInfo* é requerida uma especial atenção por parte do clOpenCL. No caso da extração dos dispositivos, o OpenCL devolve os apontadores reais dos dispositivos. Portanto o clOpenCL tem que transformar os apontadores reais dos objetos dispositivos para os apontadores clOpenCL. O seguinte excerto de código é usado na função que captura o *clGetContextInfo*, para tal efeito.

```
If((param_name == CL_CONTEXT_DEVICES) && (param_value != NULL) &&
    (ccl_reply.result == CL_SUCCESS)){
num_devices = ((param_value_size_ret == NULL) ? param_value_size :
    min(param_value_size, *param_value_size_ret)) / sizeof(cl_device_id);
        for(d=0; d<num_devices; d++){
_param_value = lookup_object(((cl_device_id *)param_value)[d],
    remote_nic_id,remote_endpoint_id);
               if(_param_value==NULL)
((cl_device_id *)param_value)[d]= register_object(((cl_device_id
    *)param_value)[d],addr,remote_nic_id,remote_endpoint_id);
               else
```

```
                ((cl_device_id *)param_value)[d]=_param_value;
    }
}
```

Depois da execução da primitiva no runtime do OpenCL é efectuado o controlo no clOpenCL do parâmetro efetivo para a operação de recolha de dispositivos. A ausência de um indicador explícito com o número de dispositivos associados obriga à verificação através de outros meios. Através do número de bytes devolvidos pela primitiva no parâmetro *param_size_ret*, é possível obter o número exato de dispositivos associados ao contexto em causa. Obtido o número exato de dispositivos o controlo individual é facilitado. Nesta operação é apresentada a técnica de *lookup*, que representa a tradução do apontador do objeto real para o apontador gerado pelo clOpenCL, ou seja, o mecanismo de *lookup* é uma pesquisa à região de memória alojada pelo clOpenCL onde se encontram todos os objetos relacionados com o OpenCL. Em termos práticos, a pesquisa é efetuada à tabela de *hash* do clOpenCL, convertendo-se o apontador clOpenCL para o apontador OpenCL. A pesquisa é lançada com o apontador manipulado pelo clOpenCL, dos objetos em causa, e o seu conjunto de elementos comunicativos. Dentro do ciclo individual de controlo dos dispositivos é verificado a sua integridade com o clOpenCL e, caso não se encontrem registados, o registo é efetuado. Este mecanismo é desencadeado nas primitivas informativas do OpenCL que exijam um registo dos objetos gerados e a sua entidade.

4.3 Objectos de Memória

No desenvolvimento do clOpenCL surgiram algumas dificuldades na manipulação da primitiva clCreateBuffer. Verificou-se que o OpenCL permite a interação dos objetos de memória com a memória alocada na RAM de várias formas. Uma das formas que o clOpenCL não suporta é do mapeamento em RAM, ou seja, o objeto de memória gerado pela primitiva utiliza a zona de memória mapeada pelo apontador gerado no host em

memória RAM; toda a memória tem que ter como alvo o próprio dispositivo vinculado ao objeto de memória. Suportar buffers remotos em memória RAM implicaria a utilização de sistemas de gestão de memória distribuída partilhada. O suporte seria um risco no desenvolvimento e um peso adicional no clOpenCL, tendo-se optado por não suportar esta funcionalidade e direcionar os esforços para outros desenvolvimentos mais importantes.

4.3.1 Escritas e Leituras dos Objetos de Memória

Ainda no seguimento da manipulação dos objetos de memória, aparecem destacadas as primitivas de leitura e de escrita para os buffers. Estas primitivas desempenham um papel bastante importante no OpenCL, transportando os dados dos dispositivos para o host e vice-versa.

Os aspectos mais relevantes que estas primitivas introduzem dizem respeito à incidência do sincronismo e tratamento dos dados em tempo de execução de uma aplicação OpenCL. As primitivas de leitura e escrita têm a capacidade de embutir mecanismos explicitados pelo utilizador, que permitem um bloqueio na thread principal do programa a fim de realizar a operação ou estabelecer uma execução concorrente à thread principal através de mecanismos implícitos no OpenCL. As primitivas de leitura e escrita do OpenCL têm o seguinte protótipo:

- cl_int clEnqueueReadBuffer(cl_command_queue, cl_mem, cl_bool, size_t, size_t, void *, cl_uint, const cl_event *, cl_event *)
- cl_int clEnqueueWriteBuffer(cl_command_queue, cl_mem, cl_bool, size_t, size_t, const void *, cl_uint, const cl_event *, cl_event *)

4.3.1.1 Fragmentação das Mensagens

A natureza das primitivas de leitura e escrita remete para a manipulação de objetos de memória com tamanhos variáveis. Para o clOpenCL, estas primitivas exercem um

peso acrescido, devido ao fluxo de dados que uma aplicação pode efetuar no seu tempo de vida. A fim de otimizar as comunicações nas duas primitivas, foi foi implementado um mecanismo de fragmentação de mensagens, onde, mensagens superiores a 4 MB são divididas e enviadas em pedaços de 4MB, sincronamente. O seguinte excerto de código representa a abordagem seguida na fragmentação das mensagens, nas primitivas de leitura e de escrita.

```
for(b=0; b<ccl_request.cb; b+=MAX_OMX_PACKET_SIZE){
        cb2 = min(MAX_OMX_PACKET_SIZE, ccl_request.cb - b);
        _ccl_omx_recv_packet((void *)ptr+b, cb2, TAG_DATA_PACKET, ALL_BITS,
            NULL);
}
```

A escolha do tamanho do fragmento de 4MB deve-se ao facto de, na rede do *testbed* utilizado no âmbito do trabalho, se atingir um máximo de utilização de largura de banda com a fragmentação de 4MB.

4.3.2 Eventos

Um dos conceitos ainda não tratados neste trabalho, e que são introduzidos nas primitivas que manipulam aspetos assíncronos, é o objeto evento. A particularidade do objeto evento é fazer referência ao estado da execução da primitiva em causa. O objeto evento permite dar a conhecer a outras primitivas o estado em que se encontra a operação que desencadeou o evento e, se for o caso, esperar pela conclusão da operação. Os três últimos parâmetros das primitivas de leitura e escrita estão associados ao tratamento de eventos.

No clOpenCL, os eventos são tratados como qualquer outro objeto OpenCL; é efetuado o registo, com recurso ao mecanismo de tradução de apontadores. A utilização dos eventos só se torna crítica quando a leitura desse evento implica a interação multi-nó. Para isso é necessário criar um mecanismo de atualização de estados dos eventos em modo

assíncrono. O mesmo acontece com o aspeto bloqueante que é acionado explicitamente num dos parâmetros das primitivas de leitura e escrita. Num cenário em que é despoletada uma operação não bloqueante e o buffer em causa é de natureza remota, exige-se uma assincronia entre o nó computacional adjacente com a localização da aplicação que o fez desencadear.

4.3.2.1 Resolução de Estados

A estratégia usada no clOpenCL para a resolução dos aspetos assíncronos do OpenCL baseia-se numa solução multithread. É definida uma estrutura de dados para todas as primitivas que têm um papel assíncrono, a fim de manter os endereços de memória devidamente mapeados para futuras alterações efetuadas pela thread. A responsabilidade de recuperar os dados que são processados assincronamente é inteiramente da responsabilidade do utilizador. Esta opção está de acordo com o modelo OpenCL; por exemplo, se utilizador efetuar uma operação assíncrona sobre uma determinada zona de memória e seguidamente efetuar qualquer tipo de operação, como a leitura dessa zona de memória, corre o risco de obter dados errados, dado que a operação assíncrona pode ainda estar a acontecer. Daí que o clOpenCL também não tenha a obrigação de providenciar algum tipo de mecanismo que inverta essa situação.

4.4 Objeto Programa

No tratamento do kernel, o OpenCL oferece dois mecanismos distintos para o processo de criação do programa que irá ser executado nos dispositivos OpenCL. O processo de criação do programa com a primitiva clCreateProgramWithSource implica carregar o código fonte para memória, referenciado por um endereço de memória local. Contudo, o programa necessita de ser compilado posteriormente através da primitiva clBuildProgram, enquanto a primitiva clCreateProgramWithBinary prescinde de uma compilação no tempo de vida da

aplicação; esta primitiva carrega o binário compilado exteriormente à aplicação, podendo logo passar à execução do *kernel*. Para evitar compilar o código fonte do OpenCL fora da aplicação host, optou-se por apenas suportar a primitiva clCreateProgramWithSource. A primitiva clCreateProgramWithSource é parametrizada da seguinte forma:

- clCreateProgramWithSource(cl_context context, cl_uint count, const char **strings,const size_t * lengths, cl_int *errcode_ret)

Os parâmetros *strings* e *lengths* estão relacionados com a vinculação do código fonte apontado pelo endereço strings e o respetivo tamanho armazenado pelo apontador *lengths*. Através do parâmetro *strings* é possível passar mais do que um ficheiro com código fonte. O tamanho de cada bloco de código fonte é armazenado no array *lengths*. O OpenCL oferece outra forma distinta de especificar implicitamente o tamanho de cada ficheiro passado pelo parâmetro *strings*, adicionando-se no final de cada bloco o caractere especial '\0'; desta maneira não é necessário tornar explícito o tamanho do ficheiro através do parâmetro *lengths*.

4.4.1 Encapsulação e Encaminhamento

No modelo clOpenCL TCP, os dados apontados pelo parâmetro *string* são enviados (byte por byte) e são recebidos dessa mesma forma no daemon, enquanto no modelo Open-MX esses dados são encapsulados num array de tamanho fixo. A limitação desta solução recai no limite da *stack* admitida pelo próprio sistema operativo; a quantidade de bytes apontada por strings apenas pode conter o número de bytes inferior à quantidade suportada pela *stack*.

```
p=ccl_request.strings;
if(count <= MAX_NUM_LENGTHS)
        for(s=0; s<count; s++){
                ccl_request.lengths[s] = (lengths != NULL) ? lengths[s] :
                    strlen(strings[s] + 1);
```

```
            total_length += ccl_request.lengths[s];
            if(total_length < MAX_STRINGS_SIZE)
                   memcpy(p, strings[s], ccl_request.lengths[s]);
            p += ccl_request.lengths[s];
      }
```

O excerto de código aqui representado mostra de que forma é efetuada o varrimento ao número de ficheiro mantidos por *strings*, sendo a informação contida em cada um dos ficheiros transportada pelo apontador p de maneira a armazenar todos os códigos fonte na variável que representa o pacote de dados, inerente ao envio de informação para o exterior.

4.5 Parâmetros Kernel

A primitiva clSetKernelArg dá acesso à definição dos parâmetros do *kernel*. A especificação OpenCL indica que, para um funcionamento normal da execução do *kernel* no dispositivo, esta primitiva seja invocada consoante o número de parâmetros definidos no *kernel*.

- clSetKernelArg(cl_kernel kernel, cl_uint arg_index, size_t arg_size, const void *arg_value)

Na representação da primitiva em cima é especificado que toda a sua parametrização está associada a um *kernel* e a um parâmetro específico desse *kernel* passado pelo parâmetro *arg_value*. O *arg_value* tem como papel passar o endereço do parâmetro alocado na região de memória do host. O *arg_value* pode conter objetos específicos do OpenCL, como objetos de memória, entre outros, e variáveis definidas pela linguagem C. Essa peculiaridade exige uma especial atenção para a região de memória apontada pelo clOpenCL; antes de efetuar a execução da primitiva real, é necessário efetuar verificações relativamente ao tipo de argumento passado pelo parâmetro *arg_value*. Como o clOpenCL tem um endereçamento

específico para cada objeto OpenCL, é necessário efectuar a tradução do clOpenCL para OpenCL.

4.6 Epílogo

Este capítulo teve a especial incidência nos aspetos ligados ao desenvolvimento, referindo-se assim as características funcionais dos modelos do clOpenCL e a própria integração com o OpenCL. A abordagem do clOpenCL foi mantida dentro do leque de opções do OpenCL, ou seja, seguiu à risca a abordagem conceptual do OpenCL. Com isso surgiram algumas dificuldades, que foi necessário contornar dado que a abordagem do OpenCL é apenas local.

Capítulo 5

Comparações e Benchmarks

O presente capítulo tem como objetivo a avaliação do clOpenCL em cenários de teste. Aqui são testados os dois modelos clOpenCL nas vertentes comunicativas e no envolvimento da CPU, ao nível da comunicação. Ambos os testes envolvem as GPUs e CPUs nas operações.

5.1 Incidência dos testes

O domínio de incidência dos testes efetuados foi ao nível do aproveitamento da largura de banda e da sobrecarga da CPU, provocada pela comunicação inter-nó. Foram também efetuados testes isolados com as primitivas OpenCL, relacionando o tempo de execução entre os modelos e o OpenCL.

Embora as tecnologias de comunicação tenham sofrido notáveis desenvolvimentos ao longo dos tempos, ao nível do tempo de resposta e da largura de banda, não podem ainda competir diretamente com o desempenho obtido no barramento, ou seja, com aplicações que são otimizadas para execução local. Em contrapartida, numa solução distribuída tira-se partido de um maior aproveitamento e aglomeração de dispositivos. É com está analogia que são desempenhados os testes nesta secção.

5.2 Largura de Banda utilizada

A fim de realizar o teste de largura de banda utilizada na rede Gigabit Ethernet, foram usadas as primitivas de escrita e leitura dos objetos de memória do OpenCL. Estas primitivas têm a capacidade de movimentar um número considerável de dados e, no ponto de vista do clOpenCL, conduzirem a um peso considerável nas aplicações. Tendo em conta que, no desenvolvimento de qualquer tipo de aplicação, num ambiente cluster, as comunicações são uma grande limitação, não basta olhar apenas para a especificação do hardware que suporta as comunicações sendo também importante o protocolo utilizado.

5.2.1 TCP vs Open-MX

Neste teste analisa-se o aproveitamento da largura de banda, por parte dos dois modelos clOpenCL. Embora os dois mecanismos de comunicação usados em ambos os modelos utilizem a camada Ethernet, englobam protocolos distintos nas outras camadas de comunicação. Foi nesta perspetiva que foi desenhada uma aplicação de teste, que enverga um conjunto de requisitos mínimos OpenCL a fim de estabelecer sucessivas leituras e escritas através das primitivas clEnqueueReadBuffer e clEnqueueWriteBuffer, para transportar dados pela rede, envolvendo os dispositivos CPU e GPU separadamente. A relevância destas primitivas, para este efeito, está na quantidade de dados que podem envolver, transmitindo um peso considerável para o clOpenCL.

Os testes foram realizados fazendo variar a quantidade de dados lidos/escritos. Os resultados obtidos são demonstrados na Figura 5.1.

A quantidade de dados movimentados depende da capacidade do hardware; de maneira que é possível distinguir no gráfico patamares distintos entre CPU e GPU; enquanto a CPU tem a capacidade de armazenar um total de 1,6 GB, a GPU utilizada apenas consegue armazenar aproximadamente 1 GB.

A proporcionalidade entre o tamanho dos dados e o tempo de conclusão das primitivas

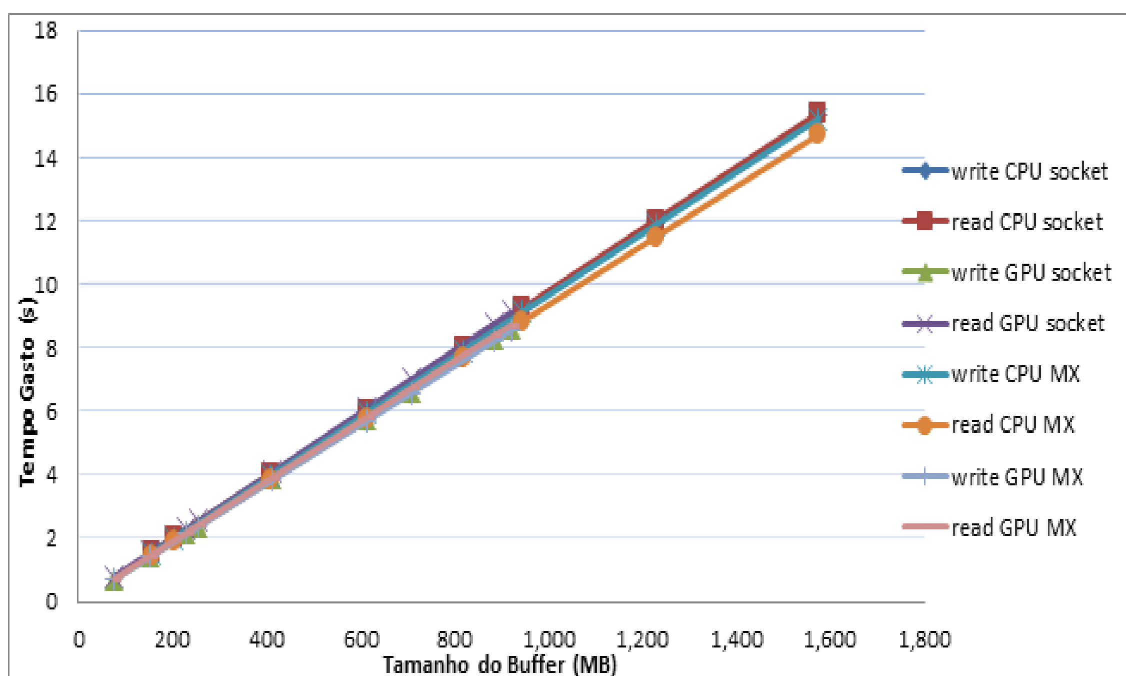

Figura 5.1: Performance de leitura e escritas em nós remotos.

verifica-se no domínio de cada tecnologia de comunicação. Ambas as tecnologias apresentam tempos de operações semelhantes, embora se possa verificar uma ligeira vantagem no desempenho obtido com o Open-MX.

5.3 Consumo do cpu

Num ambiente cluster é de todo essencial libertar os recursos computacionais, para garantir maior desempenho às aplicações. O teste aqui efetuado é referente à sobrecarga da CPU nas comunicações efetuadas com o uso das primitivas de leitura e escrita do clOpenCL. A Figura 5.2 apresenta o resultado do consumo da CPU enquanto as mensagens são trocadas.

São efetuados testes entre os dois modelos de leitura e escrita com os dispositivos OpenCL GPU e CPU. Os dispositivos utilizados encontravam-se em locais remotos.

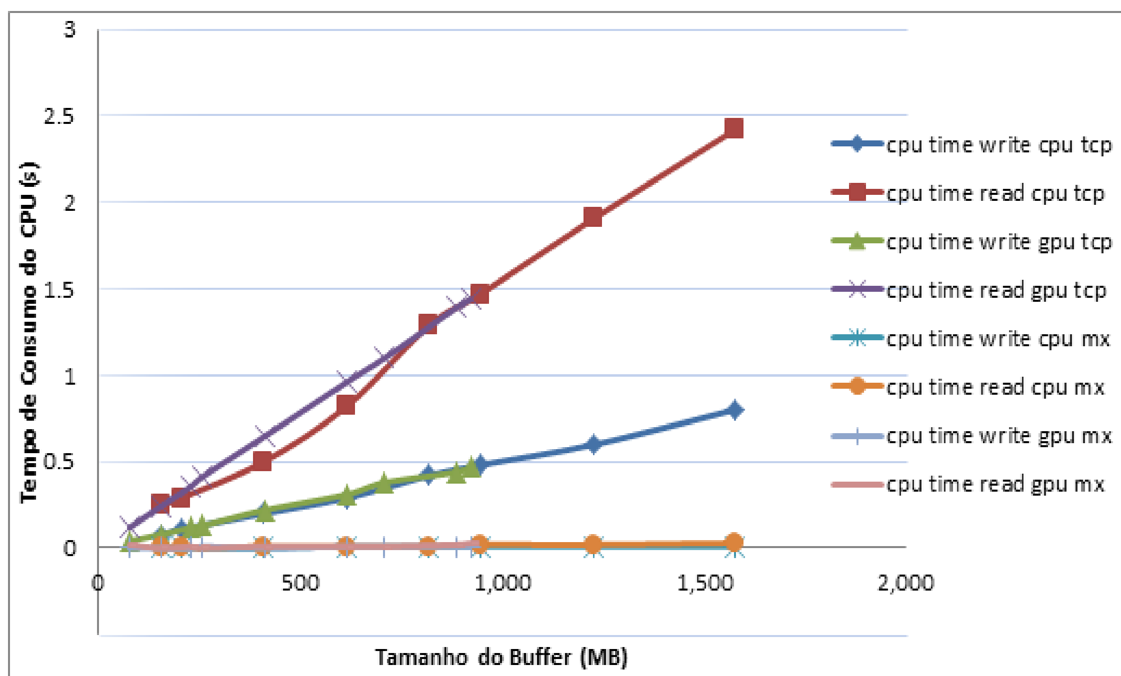

Figura 5.2: Performance de leitura e escritas em nós remotos.

Tendo em conta que o Open-MX utiliza mecanismos que libertam a CPU enquanto as mensagens são trocadas entre os diversos nós, o que não acontece no modelo socket TCP, que recorre à CPU para efetuar as comunicações, a implementação clOpenCL sobre o Open-MX, apesar de não oferecer um desempenho significativamente melhor, permite libertar a CPU para execução de código aplicacional, o que é uma grande vantagem. No entanto, poderiam ser usados mecanismos do género das apresentadas em [PB05], como TOE (TCP offload engines), com o objetivo de tornar possível trocar mensagens sem passar pelo processador, usando exclusivamente a pilha TCP/IP.

5.3.1 Primitivas Alvo

Nesta secção são apresentadas as primitivas do OpenCL que permitem a realização de um programa básico, a fim de expor o impacto que cada primitiva induz em ambos os modelos e no OpenCL nativo. Os resultados apresentados foram obtidos com uma

aplicação simples, desenvolvida em linguagem C e envolvendo o OpenCL com o propósito de efetuar uma multiplicação de arrays. A aplicação não se destina a testar a solidez dos modelos nem o próprio OpenCL; serve apenas para avaliar a aplicabilidade dos modelos, comparativamente com o OpenCL.

O parâmetro principal de avaliação é o tempo de execução de cada primitiva, medido desde a chamada da primitiva até à sua conclusão. O conjunto de primitivas avaliadas na aplicação de multiplicação de arrays em OpenCL é apresentado na Tabela 5.1.

Tabela 5.1: Tempo de execução em nanosegundos das primitivas em ambas os modelos.

Primitivas	OpenCL	clOpenCL TCP		clOpenCL Open-MX	
		Remoto	Local	Remoto	Local
clGetPlatformIDs	8	80002	8	243	8
clGetDevicesIDs	7	79964	8	128	13
clCreateContext	17	108885	56	149	27
clCreateCommandQueue	6	79968	7	164	11
clCreateBuffer	20	79782	9	143	12
clCreateProgramWithSource	12	79806	6	148	18
clBuildProgram	77868	80449	95663	87584	95837
clCreateKernel	13	6	7	148	18
clSetKernelArg	4	21	3	134	8
clEnqueNDRangeKernel	10	943	29	142	26

Na tabela é indicado a localização referente à execução das primitivas e dos seus recursos. Desde já, o OpenCL nativo foi incluído no cenário a fim de representar a *linkagem* da aplicação multiplicação de arrays diretamente ao objeto de biblioteca do próprio OpenCL, sem envolver o clOpenCL. De entre todas as primitivas distingue-se a primitiva clGetPlatformIDs; para ambos os modelos clOpenCL não é possível especificar a localidade da primitiva, dado que a extração das plataformas envolve todos os nós computacionais ativos; para executar esta primitiva apenas localmente, seria necessário remover os serviços daemon ativos nos nós.

Capítulo 6

Conclusões

Neste capítulo é feito um balanço final dos resultados alcançados e são apontados linhas para desenvolvimentos futuros.

6.1 Análise da implementação e resultados

A abrangência deste trabalho permite descrever o peso nas comunicações nas aplicações construídas para tirar proveito de um sistema distribuído; a parte mais sensível num sistema distribuído são as comunicações, devido à limitação das velocidades face ao hardware utilizado. Todavia, não é apenas o hardware o fator limitador; no contexto do clOpenCL, isto torna-se percetível, com a comparação efetuada aos dois modelos de comunicação. É percetível que, na experimentação dos modelos no mesmo hardware Ethernet Gigabit e nas mesmas condições, as diferenças são esmagadoras para o modelo construído sobre o Open-MX, face ao modelo sobre socket TCP. Os melhores resultados ao nível do desempenho são conseguidos com a implementação Open-MX quer no envolvimento da CPU nas comunicações, quer no tempo de resposta. Contudo, também existem contrariedades na utilização do Open-MX, como a obrigatoriedade da instalação do serviço em cada um dos nós computacionais e, a impossibilidade de efetuar *routing*, ou seja, só é possível o

uso do Open-MX em máquinas que estejam ligadas através do mesmo Switch.

6.2 Trabalho Futuro

Devido ao facto do OpenCL ser uma especificação ainda recente e existindo um interesse acrescido no seu desenvolvimento, há uma necessidade constante de aumentar o seu nível de suporte. No espaço de um ano, o OpenCL foi alvo de duas novas versões, cada uma suportando muitos mais recursos e melhoramentos ao nível da performance relativamente à anterior. Portanto, o clOpenCL depende do desenvolvimento do OpenCL, sendo necessário adaptar o clOpenCL sempre que o OpenCL lance novas primitivas e novos recursos.

A questão das tecnologias utilizadas para comunicação e compactação, seria uma possível escolha para tentar aumentar a performance do clOpenCL. As evidências recolhidas neste trabalho, indicam que a tecnologia utilizada na comunicação tem um grande impacto nos resultados finais, mesmo estando em igualdade relativamente ao hardware utilizado. Um melhoramento evidente seria a implementação da tecnologia socket sobre o UDP. Este apresenta caracterizas melhores para o HPC, em relação ao socket TCP. Para além desta possibilidade, tendo em conta as características de ambos as tecnologias, seria enriquecedor construir um clOpenCL híbrido, fazendo uso do melhor das tecnologias de comunicação, com o propósito de melhorar o desempenho, juntando mecanismos que evitam a sobrecarga da comunicação no CPU.

Relativamente à comunicação, para além do suporte a novas especificações do OpenCL, um próximo trabalho seria o suporte do DMA remoto; haveria vantagens em conseguir mapear memória com mais eficácia, podendo assim fazer-se uso de *flags* específicas da primitiva clCreateBuffer e, inclusivamente, suportar operações assíncronas do OpenCL.

Bibliografia

[AA12] Antonio Pina Luis Paulo Santos Albano Alves, Jose Rufino. clopencl - supporting distributed heterogeneous computing in hpc clusters. *Tenth International Workshop HeteroPar - Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms*, pages 391–407, 2012.

[AB11] A. Shiloh A. Barak. The virtual opencl (vcl) cluster platform. *Intel European Research and Innovation Conf.*, page 196, 2011.

[DBK10] Wen-mei W. Hwu David B. Kirk. *NVIDIA Programming Massively Parallel Processors*. ELSEVIER, 2010.

[DSM04] Adam L. Bazinet Daniel S. Myers. Intercepting arbitrary functions on windows, unix, and macintosh os x platforms. Technical report, UMIACS, 2004.

[Gog08] Brice Goglin. Design and implementation of open-mx: High-performance message passing over generic ethernet hardware. *Workshop on Communication Architecture for Clusters*, 37, 2008.

[Gog11] Brice Goglin. High-performance message passing over generic ethernet hardware with open-mx. *Journal of Parallel Computing Symposium*, pages 85–100, 2011.

[JD11] Federico Silla Juan C. Fernandez Rafael Mayo Enrique S. Quintana-Ort Jose Duato, Antonio J. Pena. A new approach to rcuda. *XXII Jornadas de Paralelismo*, (978-84-694-1791-1):341–346, 2011.

[Mun11] Aaftab Munshi. The opencl specification. Technical report, Khronos OpenCL Working Group, 2011.

[PB05] Q. Gao R. Noronha W. Yu D. K. Panda P. Balaji, W. Feng. Head-to-toe evaluation of high-performance sockets over protocol offload engines. *IEEE International Conference on Cluster Computing*, 38(0-7803-9486-0):1–10, 2005.

[PG05] Arthur B. Maccabe Patricia Gilfeather. Connection-less tcp. *19th IEEE International Parallel and Distributed Processing Symposium*, (0-7695-2312-9), 2005.

[SS08] Alan Morris David Cronk Sameer Shende, Allen Malony. Observing parallel phase and i/o performance using tau. *DoD HPCMP Users Group Conference, 2008. DOD HPCMP UGC*, pages 431–346, 2008.

[YTL07] Robert N. Shorten Yee-Ting Li, Douglas Leith. Experimental evaluation of tcp protocols for high-speed networks. *IEEE/ACM Transactions on networking*, 15(0018-9219):879–899, 2007.