rOpenCL: uma Ferramenta para Acesso de Aplicações Heterogéneas a Co-Processadores Remotos

Rui Alexandre Coelho Alves - a32045

Dissertação apresentada à Escola Superior de Tecnologia e de Gestão de Bragança para obtenção do Grau de Mestre em Sistemas de Informação.

Trabalho orientado por:

José Carlos Rufino Amaro

Esta versão da dissertação contempla as críticas e sugestões feitas pelo Júri.

Bragança

2019-2020

rOpenCL: uma Ferramenta para Acesso de Aplicações Heterogéneas a Co-Processadores Remotos

Rui Alexandre Coelho Alves - a32045

Dissertação apresentada à Escola Superior de Tecnologia e de Gestão de Bragança para obtenção do Grau de Mestre em Sistemas de Informação.

Trabalho orientado por:

José Carlos Rufino Amaro

Esta versão da dissertação contempla as críticas e sugestões feitas pelo Júri.

Bragança

2019-2020

Dedicatória

"Se queres chegar onde os outros não chegam, faz o que os outros não fazem."

- Ives Lamarck

.

Agradecimentos

Pelo empenho, dedicação e confiança, mas fundamentalmente pela elevada disponibilidade, mesmo em momentos que não eram os mais propícios, um agradecimento muito grande ao Prof. José Carlos Rufino Amaro, orientador deste trabalho.

Em seguida, um agradecimento especial à minha família, especialmente à minha mãe, à minha irmã e ao meu pai, que sempre tentaram dar-me as melhores condições para alcançar o maior sucesso possível, compreendendo a minha ausência em inúmeros momentos familiares, e ficando a promessa de os compensar devidamente.

Por último, deixo uma palavra de apreço a todos que direta ou indiretamente contribuíram para o sucesso deste trabalho, mas que deixo no anonimato para não correr o risco de me esquecer de alguém.

A todos, muito obrigado.

Resumo

Há cerca de uma década, o panorama da arquitetura dos sistemas de computação registou um salto evolutivo, com o aparecimento de *sistemas heterogéneos*. Nestes sistemas, à unidade central de processamento (CPU), talhada para uso genérico, juntaram-se dispositivos co-processadores, como GPUs e FPGAs, de diferentes arquiteturas. Originalmente concebidos para fins muito específicos (como processamento gráfico ou de sinal), estes co-processadores passaram a ser vistos como elementos auxiliares de processamento, capazes de acelerar a execução de aplicações computacionalmente exigentes.

Para permitir a exploração eficiente de sistemas heterogéneos, e garantir portabilidade do código, definiram-se standards abertos, como o OpenCL, suportando co-processadores de virtualmente qualquer tipo. Noutros casos, passaram a existir *frameworks* proprietárias, orientadas a dispositivos de fabricantes específicos, como a *framework* CUDA para GPUs da NVIDIA. Comum a todas estas abordagens é o facto de, originalmente, apenas preverem a utilização de co-processadores locais, ligados a um único sistema hospedeiro, não possibilitando a exploração de aceleradores ligados a outros sistemas, acessíveis via rede, limitando assim o potencial de aceleração das aplicações.

O trabalho desenvolvido nesta dissertação dá resposta a esta limitação. Consistiu na criação do *remote OpenCL* (rOpenCL), *middleware* e serviços que, em conjunto, permitem que uma aplicação OpenCL (mesmo pré-compilada), explore de forma transparente e eficiente o conjunto de aceleradores disponíveis num ambiente distribuído de sistemas Linux, recorrendo a comunicação portável assente em sockets BSD. A abordagem é validada recorrendo a *benchmarks* OpenCL de referência, que provam a conformidade do rOpenCL com a especificação OpenCL 1.2, bem como a robustez e escalabilidade da implementação.

Palavras-chave: OpenCL, sockets, C, sistemas heterogéneos, sistemas distribuídos.

Abstract

About a decade ago, the landscape of computer systems architecture registered an evolutionary leap, with the appearance of *heterogeneous systems*. In these systems, the central processing unit (CPU), designed for generic use, was joined by co-processor devices, such as GPUS and FPGAS, of different architectures. Originally designed for very specific purposes (such as graphic or signal processing), these co-processors came to be seen as auxiliary processing elements, capable of accelerating the execution of computationally demanding applications.

To allow efficient exploitation of heterogeneous systems, and to ensure portability of code, open standards were defined, such as OpenCL, supporting coprocessors of virtually any type. In other cases, there have been proprietary *frameworks* oriented to devices from specific manufacturers, such as the CUDA *framework* for NVIDIA GPUs. Common to all these approaches is that they originally only provide for the use of local co-processors, which are connected to a single host system, and do not allow the exploitation of accelerators connected to other systems, accessible via the network, thereby limiting the potential for application acceleration.

The work developed in this dissertation responds to this limitation. It consisted of the creation of *remote OpenCL* (rOpenCL), middleware and services that allow an OpenCL application (even pre-compiled) to transparently and efficiently explore the set of accelerators available in a distributed Linux system environment, using portable BSD sockets for communication. The approach is validated using reference OpenCL benchmarks, which prove the rOpenCL compliance with the OpenCL 1.2 specification, as well as the robustness and scalability of the implementation.

Keywords: OpenCL, sockets, C, heterogeneous systems, distributed systems.

Conteúdo

1 Introdução **1**

1.1 Contribuições 4

1.2 Relação com Trabalho Anterior 5

1.3 Estrutura do Documento 6

2 Estado da Arte **7**

2.1 OpenCL 7

2.2 Hybrid OpenCL (2010) 10

2.3 VCL (2011-2017) 11

2.4 dOpenCL (2012-2013) 13

2.5 clOpenCL (2012) 15

2.6 SnuCL (2012-2015) 17

2.7 clusterCL (2020) 19

2.8 Síntese Comparativa 22

3 Arquitetura e Implementação **27**

3.1 Prólogo 27

3.2 Versão Inicial 28

3.3 Paralelizar no *host* 30

3.4 Paralelizar no serviço 32

3.5 Perdas de dados na troca de mensagens 35

3.5.1 Código de erro CL_ERROR_NETWORK 36

3.6 Comunicação com recurso a TCP . . . 37
3.7 Integração com o ICD-Loader . . . 38
3.7.1 Gestão dos apontadores associados ao ICD-Loader . . . 39
3.7.2 Atributo de plataforma CL_PLATFORM_IP . . . 44
3.8 Gestão de conexões TCP . . . 45
3.8.1 Gestão de conexões TCP no *driver* . . . 46
3.8.2 Gestão de conexões TCP nos serviços . . . 48
3.8.3 Efeito da gestão de conexões TCP duradouras . . . 48
3.9 *Thread Safety* . . . 49
3.10 Estágio Atual do rOpenCL . . . 50

4 Análise de Desempenho **53**
4.1 Prólogo . . . 53
4.2 *Benchmarks* OpenCL . . . 54
4.3 Ambiente de Teste . . . 59
4.4 Cenários de Teste . . . 60
4.4.1 Testes Mono-Cliente . . . 60
4.4.2 Testes Multi-Cliente . . . 61
4.5 Metodologia e Métricas . . . 63
4.6 Resultados dos Testes Mono-Cliente . . . 65
4.6.1 BabelStream . . . 65
4.6.2 cl-mem . . . 66
4.6.3 clpeak . . . 68
4.6.4 FinanceBench . . . 69
4.6.5 PolyBench . . . 70
4.6.6 Rodinia . . . 73
4.6.7 Hashcat . . . 75
4.6.8 Discussão . . . 76
4.7 Resultados dos Testes Multi-Cliente . . . 77

4.7.1 FinanceBench 78
4.7.2 cl-mem 80
4.8 Comparação com Outras Abordagens 85
4.9 Comparação com CPU Local 88

5 Conclusão **91**
5.1 Trabalho Futuro 92

A Publicação Científica **A1**

Lista de Tabelas

2.1 Comparação de abordagens OpenCL distribuídas (1/2). 23
2.2 Comparação de abordagens OpenCL distribuídas (2/2). 23

3.1 *Parâmetros Principais* das primitivas OpenCL (1/2). 42
3.2 *Parâmetros Principais* das primitivas OpenCL (2/2). 43
3.3 *Profiling* da função *clSetKernelArg* com UDP e TCP (tempos em ms). . . 46
3.4 *Profiling* de *clSetKernelArg* com TCP sem (A) e com (B) conexões duradouras. 49
3.5 Cobertura da especificação 1.2 do OpenCL pelo rOpenCL. 51

4.1 Lista de *benchmarks* OpenCL identificados (a **negrito**, os selecionados). . . 56
4.2 Lista de *benchmarks* selecionados. 57
4.3 Especificações de hardware dos servidores de virtualização. 59
4.4 Especificações das máquinas virtuais. 60
4.5 Combinações de GPUs para os testes com Hashcat. 76
4.6 Desacelerações para o cl-mem no cenário 1G. 81
4.7 Multiplicação de Matrizes: rOpenCL vs abordagens alternativas. 88
4.8 Execução do BabelStream e do cl-mem: CPU local vs GPU remota. 89

Lista de Figuras

2.1 Pogramação Tradicional vs OpenCL (in `https://www.khronos.org/opencl/`). 8

2.2 Relações entre diferentes recursos no OpenCL [32]. 9

2.3 Alguns elementos do modelo de programação e operação do OpenCL [33]. . 10

2.4 Organização do Hybrid OpenCL[32]. 11

2.5 Arquitetura do VCL [38]. 12

2.6 Arquitetura do dOpenCL [39]. 13

2.7 Requisição de dispositivos ao *device manager* no dOpenCL [39]. 14

2.8 Arquitetura do clOpenCL [26]. 15

2.9 Arquitetura do SnuCL [43]. 17

2.10 Organização do *SnuCL runtime* [43]. 18

2.11 Organização do clusterCL [46]. 19

2.12 Principais componentes e módulos do clusterCL. 20

3.1 Arquitetura da versão 1.0 do rOpenCL. 29

3.2 Arquitetura da versão 1.1 do rOpenCL. 30

3.3 Diagrama de sequência de duas transações em simultâneas. 31

3.4 Mensagens UDP de um pedido rOpenCL com 2070 *bytes* de dados OpenCL. 33

3.5 Teste com *iperf3*: envio de 1 *GByte* por 1 *thread*. 35

3.6 Teste com *iperf3*: envio de 4 *GBytes* por 1 *thread*. 35

3.7 Teste com *iperf3*: envio de 1 *GByte* por 4 *threads*. 36

3.8 Teste com *iperf3*: envio de 4 *GBytes* por 4 *threads*. 36

3.9 Integração do rOpenCL com o *ICD-Loader*. 39

3.10 Saída do comando *clinfo* gerada pelo rOpenCL. 44
3.11 Gestão de conexões TCP duradouras no rOpenCL. 47
3.12 Arquitetura final do rOpenCL. 50

4.1 Cenário 1G para análise da escalabilidade dos serviços rOpenCL. 62
4.2 Cenário 2G para análise de escalabilidade dos serviços. 63
4.3 Cenário 4G para análise de escalabilidade dos serviços. 64
4.4 Resultados globais do *benchmark* BabelStream. 66
4.5 Resultados individuais dos sub-*benchmarks* do BabelStream. 66
4.6 Resultados globais do *benchmark* cl-mem. 67
4.7 Resultados individuais dos sub-*benchmarks* do cl-mem. 67
4.8 Resultados do *benchmark* clpeak (parte1). 68
4.9 Resultados do *benchmark* clpeak (parte2). 68
4.10 Resultados globais do *benchmark* FinanceBench. 69
4.11 Resultados individuais dos sub-*benchmarks* do FinanceBench. 69
4.12 Resultados globais do *benchmark* PolyBench. 70
4.13 Resultados dos 9 sub-*benchmarks* mais demorados do PolyBench. 71
4.14 Resultados dos 12 sub-*benchmarks* mais rápidos do PolyBench. 72
4.15 Resultados globais do *benchmark* Rodinia. 73
4.16 Resultados dos 12 sub-*benchmarks* mais demorados do Rodinia. 74
4.17 Resultados dos 8 sub-*benchmarks* mais rápidos do Rodinia. 75
4.18 Resultados dos testes com Hashcat. 77
4.19 Tempos de execução do FinanceBench no cenário 1G. 78
4.20 Carga máxima da GPU usada no cenário 1G com o FinanceBench. 79
4.21 Temperatura máxima da GPU usada no cenário 1G com o FinanceBench. . 79
4.22 Tempos de execução do cl-mem no cenário 1G. 81
4.23 Carga máxima da GPU no cenário 1G com o cl-mem. 82
4.24 Temperatura máxima da GPU no cenário 1G com o cl-mem. 82
4.25 Tempos de execução do cl-mem nos cenário comparáveis 1G e 2G. 83

4.26 Carga máxima das GPUs nos cenários comparáveis 1G e 2G com o cl-mem. 84
4.27 Tempos de execução do cl-mem nos cenário comparáveis 1G, 2G e 4G. . . . 84
4.28 Carga máx. das GPUs nos cenários comparáveis 1G, 2G e 4G com o cl-mem. 85
4.29 Multiplicação de Matrizes: rOpenCL vs clOpenCL (tempos em s). 86
4.30 Multiplicação de Matrizes: rOpenCL vs VCL (tempos em s). 87
4.31 Multiplicação de Matrizes: rOpenCL vs dOpenCL (tempos em s). 87
4.32 Tempos (ms) da execução do BabelStream em CPU local vs GPU remota. 89
4.33 Tempos (ms) da execução do cl-mem em CPU local vs GPU remota. . . . 90

Siglas

clOpenCL cluster OpenCL. 5, 6, 15–17, 23–25, 28, 29, 38, 50, 85, 86, 92

CUDA Compute Unified Device Architecture. 2–4, 57

dOpenCL distributed OpenCL. 13, 14, 22, 24, 25, 44, 85, 91

DSPs Digital Signal Processors. 7

FPGAs Field-Programmable Gate Arrays. 2, 7, 8

GPU Graphics Processing Unit. 3, 17, 79, 80

ICD Installable Client Driver. 8, 11–13, 16, 18, 22, 38, 39, 41, 44, 45, 92

MPI Message Passing Interface. 3, 18–20, 25, 57

MTU Maximum Transmission Unit. 29, 31, 33, 35, 85

OpenCL *Open Computing Language*. 2–18, 21–25, 27–41, 44–46, 48–58, 61, 66, 91–93

OpenGL Open Graphics Library. 51

POCL Portable Computing Language. 8, 28

RDMA Remote Direct Memory Access. 3

rOpenCL *remote* OpenCL. 4–7, 9, 13–16, 22–25, 27, 28, 30, 32, 37–41, 44–47, 49–54, 59, 61, 64, 65, 91–93

SMPD Single Program Multiple Data. 1, 7, 9

SnuCL Seoul National University CL. 17, 18, 25, 85

VCL Virtual CL. 11–14, 23, 25, 85, 92

Capítulo 1

Introdução

Um sistema computacional *heterogéneo* congrega elementos de processamento de diferentes arquiteturas, acoplando à CPU de uso genérico um ou mais co-processadores originalmente concebidos para fins específicos (processamento gráfico, processamento digital de sinal, etc.). O objetivo dessa composição é a aceleração da execução de aplicações computacionalmente exigentes, delegando nos co-processadores partes da computação cujo perfil encaixe nas capacidades computacionais específicas dos aceleradores. Por exemplo, uma aplicação do tipo Single Program Multiple Data (SMPD) é tipicamente adequada à execução em GPUs, dotadas de centenas/milhares de núcleos de execução, que executam o mesmo código em sintonia, operando sobre dados diferentes. Aplicações deste tipo, aceleradas em GPUs, são já relativamente frequentes [1]–[3].

Hoje em dia, boa parte dos sistemas computacionais são heterogéneos. Simples computadores portáteis ou de secretária têm frequentemente GPUs discretas, destinadas a jogos ou a CAD, mas que podem ser usadas para co-processamento. As próprias GPUs integradas, que partilham o mesmo chip com o CPU, são também usáveis, nalgumas situações, para o mesmo fim [4]. Na mesma linha, muitos sistemas móveis atuais possuem GPUs suficientemente poderosas para aceleração computacional [5], [6]. Esta heterogeneidade, e correspondente potencial de aceleração, é também observável em sistemas embebidos [7], [8], incluindo computadores de pequeno formato de tipo *single-board* [9].

Mas para além de plataformas de utilização comum, ou dos sistemas embebidos, os sistemas heterogéneos estão também perfeitamente estabelecidos nos *datacenters* modernos. Os operadores de *cloud computing*, por exemplo, já há algum tempo que providenciam máquinas virtuais com GPUs ou Field-Programmable Gate Arrays (FPGAs) acopladas [10], [11], suportando aplicações computacionalmente exigentes, fora dos domínios tradicionais dos *clusters HPC*. Por seu turno, estes *clusters*, tipicamente compostos por nós homogéneos, exibem heterogeneidade intra-nó, ao incorporar co-processadores, principalmente GPUs. Por exemplo, em Novembro de 2019, cerca de 40% do poder computacional agregado dos super-computadores da listagem TOP500, tinha origem em sistemas dotados de GPUs [12]; e, em Junho de 2020, 6 dos 10 supercomputadores mais poderosos dessa classificação estavam equipados também com GPUs [13]. Adicionalmente, o uso de co-processadores é considerado essencial no trajeto em direcção à computação *exascale*.

Sob um ponto de vista programático, a exploração aplicacional de sistemas heterogéneos implica que as aplicações sejam desenvolvidas utilizando paradigmas de programação específicos. Algumas dessas abordagens são de baixo nível, como a programação com a API do Compute Unified Device Architecture (CUDA) da NVIDIA [14], ou com a API do *standard* aberto *Open Computing Language* (OpenCL) [15]. Nestes casos, uma aplicação heterogénea desdobra-se em diferentes componentes de código, direccionados às diferentes arquitecturas presentes. Esses componentes são tipicamente escritos em C (com eventuais extensões) ou C++ (usando os *bindings* fornecidos para esta linguagem), e focam-se separadamente nos co-processadores e no sistema hospedeiro. Tipicamente, o código do sistema hospedeiro serve para criar as estruturas de dados necessárias para interagir com os co-processadores e realizar essas interacções (trocas de dados, emissão de código para os co-processadores, sincronização com estes, etc.); por vezes, o sistema hospedeiro também concorre para a tarefa delegada nos co-processadores.

Em alternativa, existem abordagens de mais alto nível, em que o código da aplicação heterogénea é unificado (*single-source*), sem separação explícita do código do sistema hospedeiro e dos co-processadores. É o caso de abordagens como o SYCL [16], C++ AMP [17], OpenACC [18] e OpenMP (desde a versão 4) [19], que oferecem um maior grau de

abstração: podem usar C++ completamente standard (SYCL), construções de alto nível (C++ AMP), ou diretivas/anotações ao código (OpenMP e OpenACC), de forma que os compiladores geram automaticamente estruturas de dados e operações de baixo nível (por vezes expressos em CUDA ou OpenCL). Embora haja *nuances* que as distinguem, todas estas abordagens melhoram a produtividade do desenvolvimento e a portabilidade do código, em relação às de mais baixo nível. Todavia, estas últimas ainda se revestem de bastante importância, devido às maiores possibilidade de afinação do código tendo em vista a extracção de níveis acrescidos de desempenho.

Outro denominador comum às abordagens de baixo nível, como o CUDA e o OpenCL, é que, no seu modelo aplicacional original, uma aplicação heterogénea que arranca num certo *host* apenas usufruirá dos co-processadores diretamente ligados a esse *host*. Isto é uma limitação importante ao potencial de aceleração da aplicação, dado que é também necessariamente limitado o número de co-processadores ligáveis a um único sistema, mesmo em cenários mais sofisticados em que se utilizam barramentos proprietários [20].

Para ultrapassar esta limitação, foram desenvolvidas tecnologias, ao nível do *hardware* e seus *drivers*, que permitem comunicação direta (sem recurso à RAM dos *hosts*) entre co-processadores, nomeadamente GPUs, de diferentes nós de um *cluster*. Essa comunicação assenta no protocolo Remote Direct Memory Access (RDMA) sobre uma rede Infiniband, sendo esta a base sobre a qual operam as abordagens GPUDirect-RDMA [21] da NVIDIA, e ROCn-RDMA [22] da AMD. Em particular, a abordagem da NVIDIA foi explorada, a um nível mais alto, por bibliotecas de passagem de mensagens conformes ao standard Message Passing Interface (MPI) [23], permitindo o desenvolvimento de aplicações heterogéneas distribuídas assentes nesse modelo [24]. Notavelmente, não se conhecem, à data, aproximações equivalentes para OpenCL, o que limita este tipo de abordagem a co-processadores do tipo Graphics Processing Unit (GPU) e de um só fabricante.

Uma alternativa para explorar co-processadores dispersos por vários nós interligados em rede, é manter o modelo original de programação do CUDA e do OpenCL, mas fornecer às aplicações, através do ambiente de execução, acesso a dispositivos remotos de forma transparente, como se fossem locais. Isto permite, inclusivamente, arrancar aplicações

heterogéneas em nós sem qualquer co-processador local. Permite também complementar a prestação de co-processadores locais com o contributo de co-processadores remotos.

Naturalmente, o redirecionamento de chamadas às APIs CUDA/OpenCL para co-processadores remotos importará sempre numa penalização, introduzida pela troca mensagens na rede. No entanto, dependendo da natureza e das especificidades das aplicações heterogéneas, essa penalização poderá ser facilmente amortizada. Por exemplo, uma aplicação computacionalmente intensiva, com poucas trocas de dados com o sistema hospedeiro, ou entre co-processadores, colherá mais facilmente os benefícios da distribuição da carga por co-processadores dispersos. Complementarmente, o programador, tendo noção da natureza local *versus* remota dos co-processadores, e das eventuais diferenças na capacidade computacional dos mesmos, poderá implementar os seus próprios mecanismos de balanceamento de carga, adaptados a cada cenário em particular. Outra possibilidade é a agregação de todos os co-processadores num sistema de imagem única, com mecanismos automáticos de balanceamento de carga, à margem da intervenção do programador.

1.1 Contribuições

Esta dissertação documenta o desenho, implementação e avaliação do *remote* OpenCL (rOpenCL), uma solução que permite a uma aplicação heterogénea, assente em OpenCL, usar co-processadores remotos, expostos de forma individualizada. O prosseguimento desta abordagem com o OpenCL, em detrimento do CUDA, deve-se ao fato do OpenCL ser um standard aberto e suportar co-processadores diversificados, e não apenas GPUs.

A solução desenvolvida assenta num *driver* rOpenCL, a instalar no sistema onde a aplicação OpenCL é lançada, e num serviço rOpenCL, a executar nos sistemas remotos cujos co-processadores se pretendem explorar. O *driver* rOpenCL co-existe e integra-se perfeitamente com os *drivers* OpenCL dos fabricantes dos co-processadores, permitindo à aplicação continuar a utilizar os co-processadores locais sem qualquer limitação ou penalização. Assim, o *driver* rOpenCL intervém apenas quando a aplicação pretende usar os co-processadores remotos, tratando de toda a interação com os serviços rOpenCL.

A inserção plena do *driver* rOpenCL no ambiente de execução do OpenCL permite que aplicações pré-compiladas possam tirar partido de co-processadores remotos de forma completamente transparente, não sendo necessário ter acesso ao código fonte para re-compilar ou modificar as aplicações OpenCL. A única exceção prende-se com a situação particular em que as aplicações desejam conhecer a localização dos co-processadores remotos, para utilizarem essa informação em balanceamento de carga; neste caso, as aplicações podem ser modificadas para recorrerem a uma extensão do rOpenCL à API do OpenCL, que revela, precisamente, a localização (endereço IP) dos co-processadores.

A comunicação entre o *driver* rOpenCL e os serviços rOpenCL assenta em TCP (podendo ser utilizado o UDP para contextos aplicacionais mais simples), tendo sido programada com recurso a sockets BSD. Esta opção permite executar o rOpenCL sobre virtualmente qualquer tecnologia de rede, dada a universalidade da pilha TCP/IP e do suporte dos sistemas operativos ao modelo de programação baseado em sockets BSD.

A escalabilidade da implementação assenta no recurso ao modelo de fios-de-execução, quer ao nível do *driver*, quer dos serviços rOpenCL. Internamente, o *driver* desdobra-se em diferentes fios-de-execução, para diferentes conexões TCP com serviços rOpenCL, que são reutilizadas tanto quanto possível. O *driver* rOpenCL é também compatível com o uso de múltiplos fios-de execução pelas próprias aplicações OpenCL, oferecendo as necessárias garantias de *thread safety*. Do lado dos serviços, a adoção de múltipos fios-de-execução permite também um atendimento eficiente e escalável de múltiplos pedidos rOpenCL, potencialmente oriundos de múltiplas aplicações OpenCL. Estas qualidades do rOpenCL são plenamente demonstradas com base na avaliação apresentada nesta dissertação.

Refira-se ainda que este trabalho teve já uma primeira validação por pares: um artigo [25] aceite no *workshop* WAMCA 2020, indexado no Scopus, reproduzido no apêndice A.

1.2 Relação com Trabalho Anterior

Esta dissertação retoma investigação realizada no âmbito do projeto FCT FCOMP-01-0124-FEDER-010067, que culminou, em 2012, com o aparecimento do cluster OpenCL

(clOpenCL) [26], uma das primeiras abordagens à extensão do OpenCL a um ambiente distribuído. Na altura, o grau de transparência atingido não foi o desejável, obrigando à recompilação das aplicações OpenCL. A cobertura da API OpenCL era também inferior à que foi agora atingida pelo rOpenCL. E, mais importante, o clOpenCL dependia, para troca de mensagens, de um protocolo de baixo nível e de utilização limitada ao mesmo segmento de rede (*Open-MX*), que entretanto foi descontinuado. O rOpenCL procurou assim dar resposta a estas lacunas e ir mais além, mas tendo o seu desenvolvimento beneficiado do *know-how* adquirido com o trabalho anterior em torno do clOpenCL.

1.3 Estrutura do Documento

Este documento está estruturado em 5 capítulos e 1 apêndice, com o seguinte conteúdo:

Capítulo 1: Introduz a temática e a motivação para a dissertação, e resume as principiais contribuições da mesma.

Capítulo 2: Apresenta conceitos fundamentais do OpenCL e concentra-se na descrição e análise de outras abordagens que procuram resolver o mesmo tipo de problemas que os atacados por esta dissertação.

Capítulo 3: Descreve os elementos da arquitetura do rOpenCL e os aspetos mais importantes do seu desenvolvimento e implementação. Faz também referência às tecnologias e ferramentas utilizadas.

Capítulo 4: Apresenta os resultados da avaliação e validação do rOpenCL em cenários experimentais com *benchmarks* OpenCL de referência e por contra-posição com outras abordagens do género.

Capítulo 5: Conclui o documento, com uma análise crítica ao trabalho realizado e delineando vias deixadas em aberto para trabalho futuro.

Apêndice 1: Reproduz a publicação científica que resultou deste trabalho.

Capítulo 2

Estado da Arte

Uma fase crucial do trabalho foi o levantamento do estado da arte, permitindo contextualizar o rOpenCL no universo de abordagens semelhantes, compreender até onde tinham conseguido chegar e quais poderiam ser as contribuições do rOpenCL. Este capítulo apresenta um resumo dessas abordagens, por ordem cronológica do seu aparecimento, terminando com uma comparação com o rOpenCL em aspetos considerados importantes. Antes, porém, introduzem-se conceitos fundamentais e terminologia do OpenCL.

2.1 OpenCL

O OpenCL [15] é um standard aberto para programação de aplicações heterogéneas, ou seja, aplicações com componentes de código que executam em dispositivos computacionais de diferentes arquiteturas, co-existentes no mesmo sistema hospedeiro. Além da CPU tradicional, herdeira do modelo de *von Neumman*, esses dispositivos incluem co-processadores sob a forma de GPUs, nalguns casos FPGAs e, mais raramente, Digital Signal Processors (DSPs). O recurso a co-processadores faz-se com o objetivo de tirar partido de características arquiteturais específicas, que permitem acelerar certos blocos de código. Por exemplo, código do tipo SMPD, adequa-se particularmente à execução em GPUs, onde o mesmo código, atuando sobre diferentes dados, é executado em sintonia por milhares de núcleos de processamento. Assim, no modelo de programação do OpenCL,

uma aplicação desdobra-se num *host program*, com a parte do código a executar na CPU genérica do hospedeiro, e num conjunto de *kernels*, rotinas específcas para executação nos co-processadores – ver Figura 2.1.

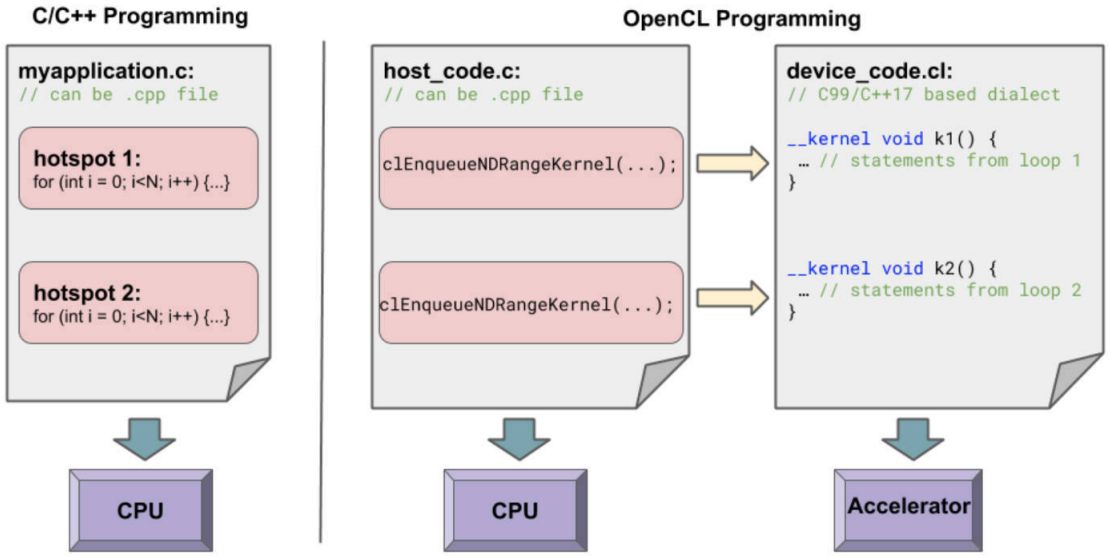


Figura 2.1: Pogramação Tradicional vs OpenCL (in `https://www.khronos.org/opencl/`).

No dialeto do OpenCL, os co-processadores designam-se por *dispositivos*, sendo agregados por *plataforma*. Uma plataforma corresponde a uma implementação particular do *standard* OpenCL, normalmente fornecida por fabricantes de dispositivos, sob a forma de um Installable Client Driver (ICD), ou simplesmente *driver*, que é basicamente uma *shared library*. Atualmente, as principais plataformas OpenCL são fornecidas pela NVIDIA para os seus GPUs (*NVIDIA GPU drivers* [27]), pela AMD para os seus GPUs e CPUs (*ROCm OpenCL runtime* [28]), e pela Intel para os seus CPUs, GPUs e FPGAs (*Intel OpenCL SDK* [29]). Existem também plataformas independentes de fabricantes, caso do Portable Computing Language (POCL) [30], [31], principalmente direccionado a CPUs e com suporte parcial a GPUs.

O *workflow* típico de uma aplicação OpenCL passa por descobrir que plataformas e dispositivos estão disponíveis, criar objetos OpenCL chamados *contextos* agregando dispositivos da mesma plataforma, criar *filas de comandos* (*command queues*) para cada dispositivo dos contextos, criar *objetos de memória* (*memory objects*) a partilhar com os

dispositivos, submeter programas (*program objects*) com rotinas (*kernels*) aos dispositivos para execução, e definir *handlers* a serem invocados quando ocorrem determinados *eventos*. A figura 2.2 mostra as relações válidas entre estes tipos de recursos.

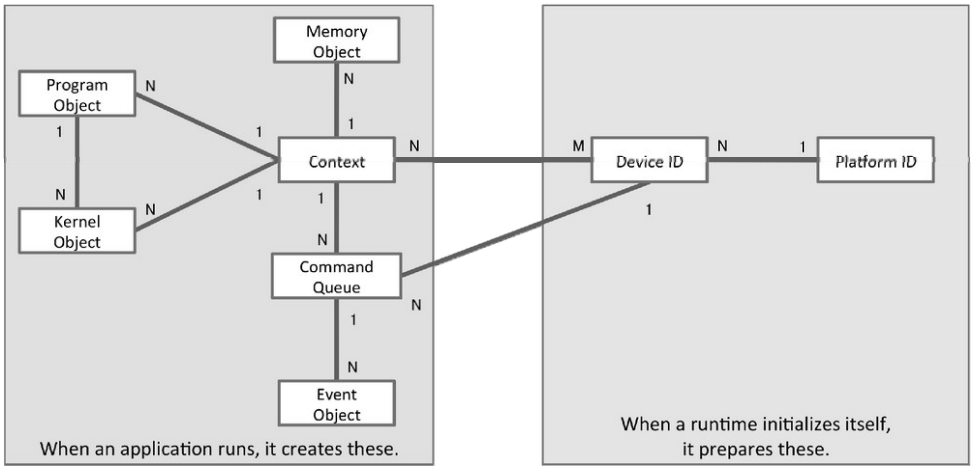


Figura 2.2: Relações entre diferentes recursos no OpenCL [32].

A submissão de programas aos dispositivos co-processadores envolve a definição de um espaço de índices (*NDRange*) usado pelas rotinas para acederem a dados de forma particionada (suporte ao modelo SMPD). Depois da submissão é necessário monitorizar a conclusão da execução das rotinas, e copiar os dados produzidos, da memória dos dispositivos para a do sistema hospedeiro. Ao nível dos dispositivos, a execução de um *kernel* corresponde a um *work-item*, sendo executadas várias instâncias em simultâneo (*work-group*). Alguns dos elementos desta descrição estão representados na figura 2.3.

O modelo de programação original do OpenCL assume um único sistema hospedeiro e respectivo dispositivos co-processadores directamente ligados a esse sistema. Não previu, portanto, a possibilidade de uma aplicação heterogénea ser distribuída por dispositivos de diferentes sistemas inter-ligados em rede. O preenchimento dessa lacuna, com o objetivo de potenciar o desempenho das aplicações heterogéneas, esteve na base de um conjunto de abordagens que a seguir se descrevem, de forma resumida, e numa perspetiva comparativa com o rOpenCL, a abordagem do mesmo domínio apresentada nesta dissertação.

A primeira versão do OpenCL (1.0) surgiu em Dezembro de 2008, contemplando 71

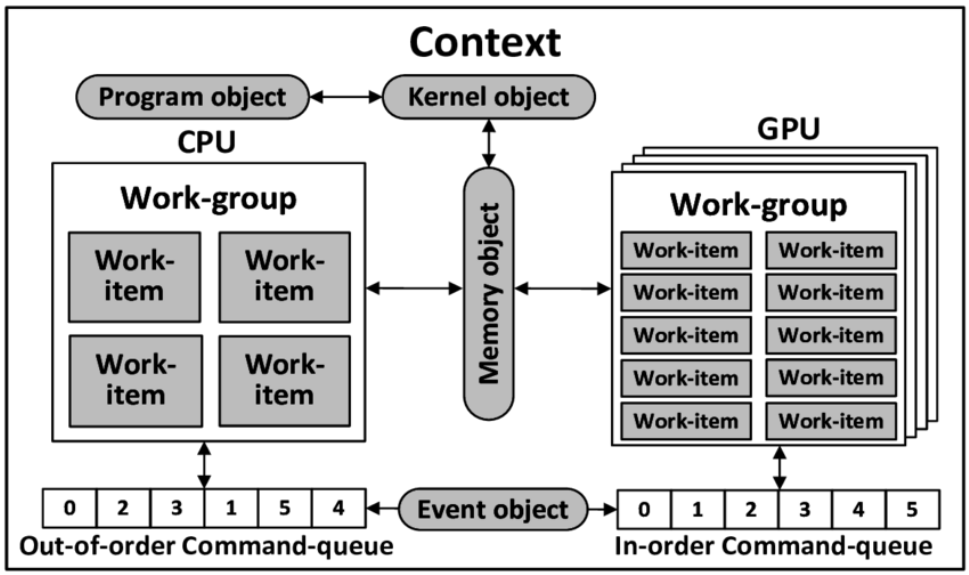


Figura 2.3: Alguns elementos do modelo de programação e operação do OpenCL [33].

primitivas, mas seria a versão 1.1 a primeira a ter um suporte mais generalizado dos fabricantes. Atualmente, a especificação do OpenCL vai já na versão 3.0 [34]. No entanto, no âmbito desta versão, apenas é mandatório o suporte às facilidades previstas na versão 1.2 [35], sendo todos os módulos das versões 2.x e 3.0 considerados opcionais.

2.2 Hybrid OpenCL (2010)

O Hybrid OpenCL [32] foi a primeira proposta para execução distribuída de código OpenCL publicada na literatura científica da área. Esta abordagem combina os conceitos de *API-forwarding* e *source-to-source translation*.

Assim, no sistema hospedeiro, as aplicaçõesOpenCL ligam-se ao *Hybrid OpenCL runtime*, uma versão do *FOXC (Fixstars OpenCL Cross Compiler) OpenCL runtime*, alterada para incluir uma camada de rede baseada em sockets. Este *runtime* direcciona as chamadas OpenCL para dispositivos locais ou remotos, conforme escolhidos pela aplicação. Nos sistemas remotos, um *Bridge Program* recebe os pedidos do sistema hospedeiro e submeto-os aos dispositivos subjacentes. A Figura 2.4 mostra esta organização.

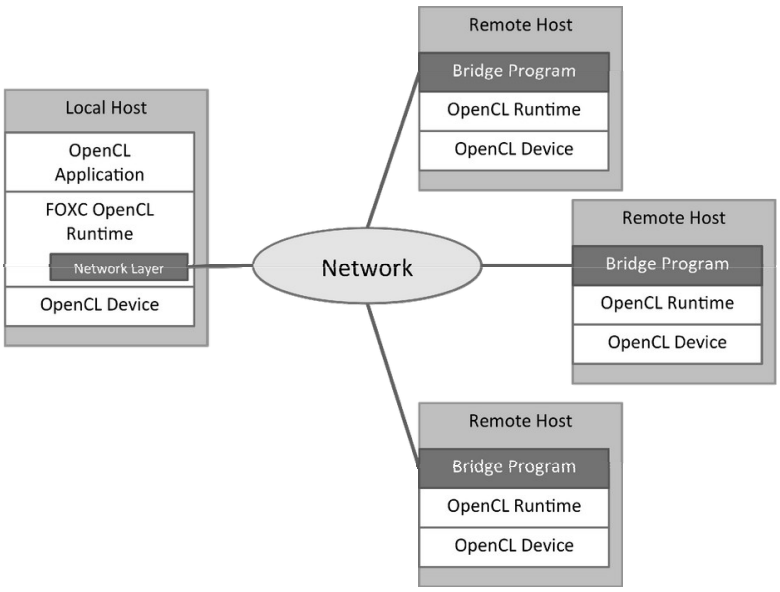


Figura 2.4: Organização do Hybrid OpenCL[32].

A característica *source-to-source translation* implica ter acesso ao código fonte original da aplicação OpenCL, de forma que o código dos *kernels* OpenCL seja traduzido para código C com funções que usam instruções vectoriais SSE IA-32, visando apenas dispositivos x86 *multi-core* (excluindo assim GPUs ou outros co-processadores).

Outra característica importante do Hybrid OpenCL é que oferece às aplicações OpenCL uma só plataforma OpenCL, que unifica todos os dispositivos disponíveis, locais e remotos, usáveis em qualquer combinação. Porém, o Hybrid OpenCL não se encaixa no mecanismo *standard* de carregamento dos *drivers* OpenCL (ICD-Loader), não suporta *callbacks* para tratamento de eventos, nem contempla balanceamento automático de carga. Define ainda os *hosts* participantes na plataforma global de forma estática, num ficheiro de texto.

2.3 VCL (2011-2017)

Outra das abordagens primordiais à execução remota de primitivas OpenCL foi o Virtual CL (VCL) [36], originalmente uma camada do MOSIX [37], um sistema de gestão de trabalhos de computação paralela para ambientes do tipo *cluster*. Tal como no Hybrid OpenCL, é fornecida a abstracção de uma plataforma global, neste caso combinando não

apenas CPUs de vários nós mas também GPUs e outros aceleradores. Com o VCL é portanto possível a criação de *contexts* que englobam dispositivos de plataformas diferentes, sejam do nó de arranque (escolha por omissão), sejam de outros nós; no entanto, a verdadeira localização dos dispositivos é escondida das aplicações, pois a escolha desses dispositivos é determinada, de forma transparente, por um *broker* externo.

Utilizando o VCL, as aplicações OpenCL conseguem ser executadas sem modificações, mas têm de ser recompiladas, para serem ligadas a uma biblioteca de *wrappers* (*VCL Library*), que é *thread-safe*. No nó de arranque (sistema hospedeiro) da aplicação OpenCL, corre um serviço *broker*, que monitoriza, gere e reserva os aceleradores dos vários nós computacionais envolvidos, redireccionando também para eles os pedidos OpenCL originários da aplicação. Em cada um dos nós participantes corre um serviço de *backend*, que recebe os pedidos do *broker* e os entrega aos dispositivos OpenCL locais (de notar que cada dispositivo é usado exclusivamente para servir cada solicitação recebida, não sendo possível a sua partilha por diferentes aplicações). A comunicação via rede entre os vários componentes assenta em *sockets* TCP/IP. A Figura 2.5 representa a arquitetura do VCL.

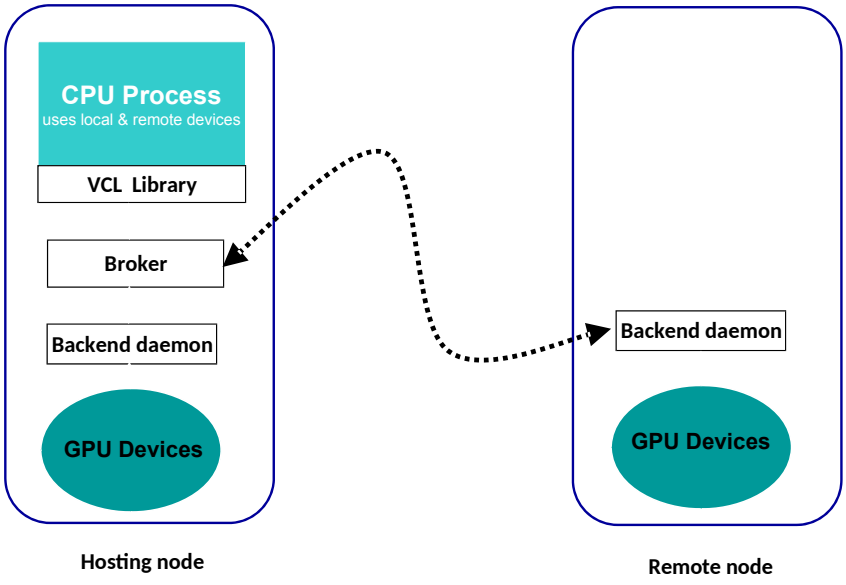


Figura 2.5: Arquitetura do VCL [38].

Devido à forma como foi originalmente desenhado, em articulação com o MOSIX, o VCL não se integra no mecanismo ICD-Loader de gestão de *drivers* do OpenCL. Quanto

ao suporte a eventos OpenCL, é no *hosting node* que é manipulado esse mecanismo, mas não há referências relacionadas com suporte do mecanismo de *callbacks*.

Uma particularidade interessante do VCL é que, embora tendo sido historicamente a segunda abordagem do género a ser publicada, o seu desenvolvimento/manutenção prolongou-se até recentemente (2017), sendo ainda possível executá-lo em sistemas atuais, o que permitiu incluí-lo na comparação de desempenho com o rOpenCL.

2.4 dOpenCL (2012-2013)

O distributed OpenCL (dOpenCL) [39] é a implementação distribuída de OpenCL mais completa, apesar de o seu desenvolvimento ter cessado em 2013. A figura 2.6 apresenta a arquitetura respetiva.

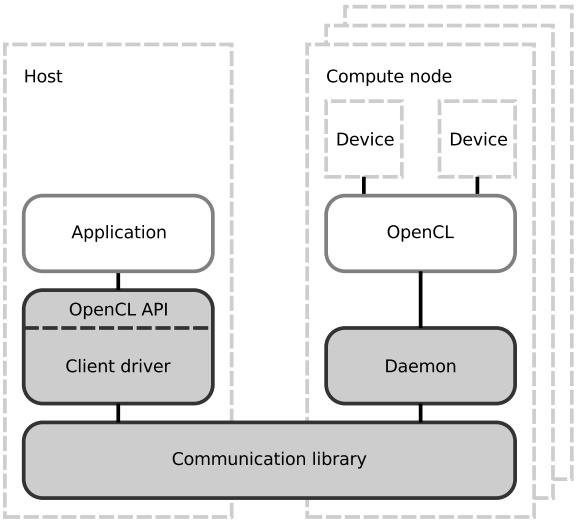


Figura 2.6: Arquitetura do dOpenCL [39].

Assim, através de um *client driver* instalado no *host* onde a aplicação OpenCL arranca (ver figura 2.6), passa a ser possível o acesso desta a co-processadores remotos. No entanto, este *client driver* não é compatível com o mecanismo ICD-Loader de gestão de *drivers* do OpenCL, operando de forma separada (não integrada com aquele mecanismo). Em cada nó de computação remoto, que oferece os préstimos dos co-processadores aí instalados, executa um *Daemon*, que submete os pedidos OpenCL recebidos da rede, para

os dispositivos OpenCL locais. A comunicação via rede entre o *client driver* e os *Daemons* assenta em TCP, recorrendo a funcionalidades de uma *Generic Communication Framework (GFC)*, que faz parte de uma *Real-Time Framework (RTF)* criada para comunicação de alto desempenho em contextos distribuídos [40] (nomeadamente jogos on-line com número massivo de participantes). Estas *frameworks* são de código fechado, existindo uma versão alternativa do dOpenCL, com comunicação assíncrona baseada na biblioteca C++ Asio [41], e que foi a usada para comparação de desempenho com o rOpenCL.

Tal como o Hybrid OpenCL e o VCL, o dOpenCL implementa uma plataforma OpenCL virtual, reunindo todos os dispositivos remotos numa só plataforma global. Essa unificação viabiliza um controlo centralizado dos dispositivos disponíveis (figura 2.7) bem como mecanismos de balanceamento de carga automáticos e transparentes para as aplicações OpenCL. Esta transparência estende-se à possibilidade de executar aplicações OpenCL binárias (pré-compiladas), o que não era suportado pelas abordagens anteriores.

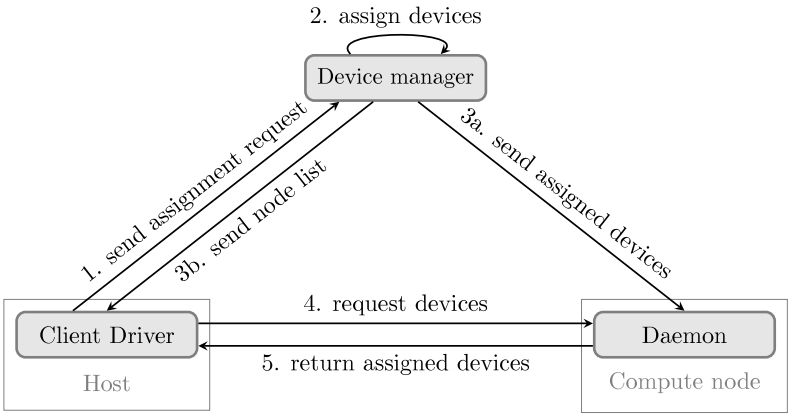


Figura 2.7: Requisição de dispositivos ao *device manager* no dOpenCL [39].

A criação e utilização de uma plataforma OpenCL global, com dispositivos oriundos de diferentes nós de computação (e possivelmente de diferentes plataformas OpenCL nativas), levanta desafios especiais. Em particular, implica gerir e manter a consistência de réplicas de objetos (como contextos OpenCL e objetos subordinados) espalhados por vários nós de computação, sendo este problema devidamente gerido pelo dOpenCL.

Outra característica importante do dOpenCL é o suporte de *callbacks*, o que permite

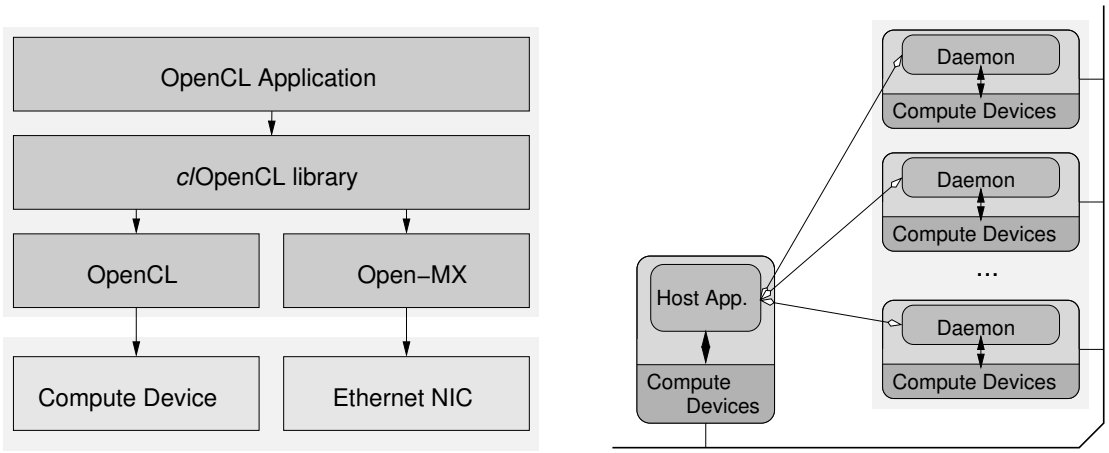
o funcionamento do mecanismo de eventos previsto noOpenCL.

Quanto à composição do sistema distribuído, a mesma é definida de forma simples, com base num ficheiro de texto que enumera o endereço IP dos vários *hosts* participantes.

2.5 clOpenCL (2012)

Como já se referiu, existe uma ligação histórica entre o rOpenCL e uma das primeiras abordagens do mesmo tipo: o clOpenCL [26], resumido nesta seção.

Assim, o clOpenCL consiste numa biblioteca de *wrappers* OpenCL (clOpenCL *library*), presente no sistema hospedeiro onde são lançadas as aplicações OpenCL, e num conjunto de serviços (*Daemons*), executando em sistemas remotos com aceleradores, inter-ligados em rede com o sistema hospedeiro. Estes elementos estão ilustrados na figura 2.8.



(a) camadas no sistema hospedeiro (b) componentes distribuídos

Figura 2.8: Arquitetura do clOpenCL [26].

A biblioteca do clOpenCL expõe às aplicações as várias plataformas e dispositivos disponíveis, quer locais, quer remotos, mas de forma separada, sem a preocupação de unificar tudo numa só plataforma virtual, ao contrário das abordagens anteriores. Isto simplifica bastante a implementação, mas passa para as aplicações a responsabilidade da escolha das plataformas e dispositivos a usar, no que são ajudadas por uma extensão à API do OpenCL que fornece a localização (endereço IP) das plataformas. Com essa informação,

as aplicações poderão realizar as suas próprias decisões de alocação e balanceamento.

Embora não tenham de ser modificadas, as aplicações OpenCL têm de ser recompiladas e ligadas com a biblioteca do clOpenCL para acederem a dispositivos remotos. Graças a diretivas especiais de ligação na compilação (`-Xlinker -wrap` no GCC), a biblioteca intercepta todas as chamadas a primitivas OpenCL, decidindo então como redirecioná-las: ou para os dispositivos OpenCL locais, ou para os serviços clOpenCL que facultam acesso a dispositivos remotos. O acesso a estes últimos faz-se por intermédio do Open-MX [42], uma biblioteca de comunicações de baixo nível, diretamente assente na camada Ethernet.

No que toca à gestão de objetos remotos, o clOpenCL cria apontadores falsos ao nível da biblioteca OpenCL local, utilizando-os como índices num vetor que os mapeia em apontadores verdadeiros, válidos no contexto dos serviços remotos.

Esta abordagem, relativamente simples, provou ser eficaz na exploração de aceleradores remotos por aplicações OpenCL, embora de pouca complexidade (*e.g.*, multiplicação de matrizes) e variedade de chamadas OpenCL (devido ao suporte limitado do clOpenCL à API do OpenCL), e para as quais o código fonte estava disponível ou foi desenvolvido de raíz. No entanto, precisamente devido à sua arquitetura de software, assente em *wrappers* que intercetam as chamadas OpenCL, e obriga à recompilação das aplicações, o clOpenCL tem utilidade e transparência aquém do desejável. A falta de transparência nota-se também na ausência de um *driver* conforme ao mecanismo ICD-Loader.

Adicionalmente, a dependência do Open-MX é problemática; este deixou de ser desenvolvido e mantido há vários anos, não sendo suportado por versões do *kernel* do Linux posteriores à 4.8. Além disso, o recurso ao Open-MX restringe o âmbito de utilização do clOpenCL ao mesmo segmento de rede, pois o Open-MX não faz encaminhamento.

O clOpenCL faz parte do conjunto das primeiras abordagens a um OpenCL distribuído, mas acabou por ser uma iniciativa que não teve continuidade. Apesar disso, foi possível montar um ambiente de execução com um *kernel* suficientemente recente para viabilizar uma comparação de desempenho com o rOpenCL.

2.6 SnuCL (2012-2015)

O Seoul National University CL (SnuCL) [43] representa a última das abordagens de 1ª geração à distribuição do OpenCL, tendo-se o seu desenvolvimento prolongado até 2015.

Partilha com essas abordagens (à excepção do clOpenCL) a visão da plataforma OpenCL virtual, que congrega todos os dispositivos do ambiente distribuído (ver figura 2.9), mas permitindo, ao mesmo tempo, criar partições destes, com certas restrições: um dispositivo pode ser uma GPU ou um subconjunto dos núcleos de CPU de um nó (incluindo o *host node*), não suportando outros tipos de dispositivos de computação.

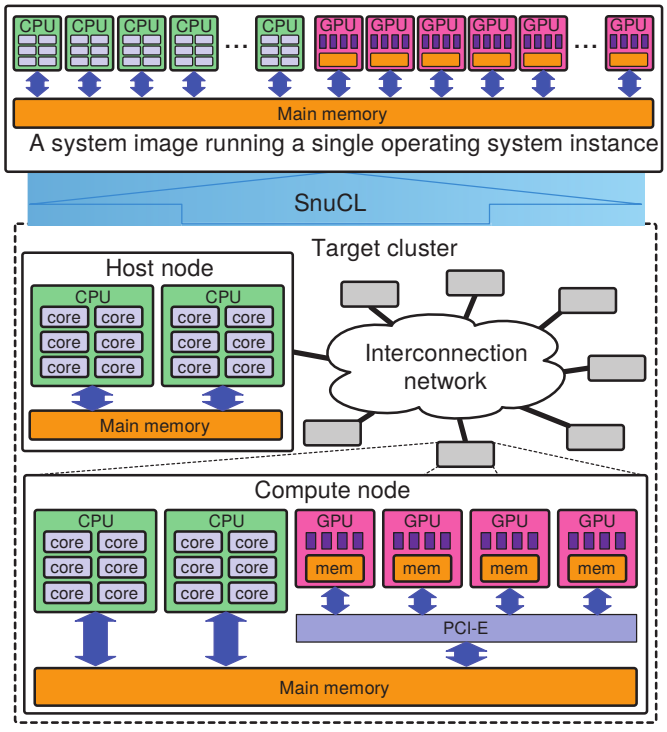


Figura 2.9: Arquitetura do SnuCL [43].

Entretanto, e recuperando a técnica *source-to-source translation* do Hybrid OpenCL, o SnuCL depende da sua própria implementação de OpenCL: aplica um tradutor *OpenCL-C-to-CUDA-C* a *kernels* direcionados a GPUs (apenas NVIDIA) e um tradutor *OpenCL-C-to-C* a *kernels* que visam CPUs; estas traduções são realizadas no *host node*, sendo

posteriormente o código traduzido e enviado para os *compute nodes* remotos onde é compilado nativamente para cada dispositivo em questão. Desta forma, o SnuCL necessita do código fonte das aplicaçõesOpenCL (embora não para o modificar), não sendo capaz de executar aplicações OpenCL binárias, nem se integrando com o mecanismo ICD-Loader.

Também contrariamente às abordagens até aqui apresentadas, o SnuCL não segue a aproximação *API-forwarding*, ou seja, não redireciona sistematicamente todas as chamadas OpenCL para sistemas computacionais remotos. Ao invés, todas as chamadas OpenCL de uma aplicação são executadas no *host node* até ao momento em que comandos são adicionados a *command queues*; a partir daí, os comandos em causa são escalonados para execução em dispositivos OpenCL de outros nós computacionais (ver figura 2.10).

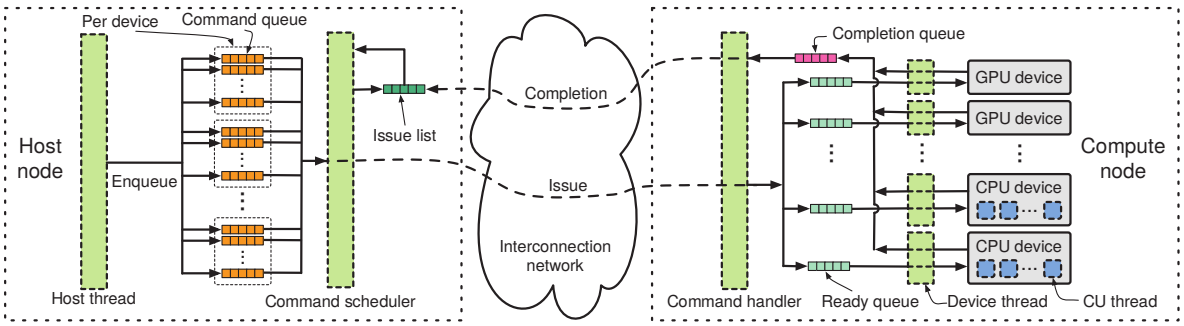


Figura 2.10: Organização do *SnuCL runtime* [43].

A comunicação entre um *host node* e os restantes nós de computação é realizada com recurso a MPI (pelo que os nós participantes são identificados de forma estática, com base num *machinefile*). Adicionalmente, o OpenCL é enriquecido com operações de comunicação coletiva entre *buffers*. Outra característica importantes no SnuCL é o suporte para a sincronização de comandos através do mecanismo de eventos do OpenCL.

Para um número elevado de nós, o escalonamento centralizado do SnuCL prejudica o desempenho, tendo sido desenvolvido o SnuCL-D [44] como solução para este problema.

2.7 clusterCL (2020)

O clusterCL[45] é a abordagem mais recente de todas as apresentadas nesta secção. Foi desenhado para suportar distribuição de carga em *clusters* heterogéneos assimétricos [1], tendo em consideração a minimização do tempo de execução e do consumo energético, e assumindo que a ordem de execução dos *kernels* não é relevante.

A figura 2.11 apresenta a arquitetura de alto nível respetiva, composta por um nó de *front-end* e um conjunto de nós de computação que alojam co-processadores de qualquer tipo. Os vários nós estão interligados em rede (*Infiniband/Ethernet*) e a comunicação entre eles é baseada em passagem de mensagens conforme ao standard MPI.

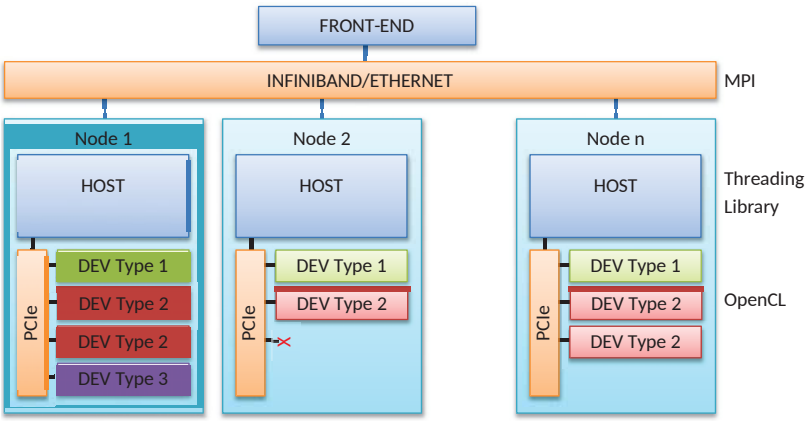


Figura 2.11: Organização do clusterCL [46].

O funcionamento do clusterCL assenta em três componentes de software principais, com vários módulos – ver figura 2.12. O *Supervisor Component* executa no nó de *front-end*, e os *Coordination Component* e *Computation Component* nos nós de computação.

Assim, cabe ao *Supervisor Component* inicializar o processo de computação. Um dos passos desse processo é a criação de uma tabela de descritores de aceleradores, com as principais características dos co-processadores do *cluster*. Essa tabela, juntamente com a estrutura do *kernel* fornecida pelo programador, permite ao módulo *Partitioner* realizar uma análise cuidada, distribuindo a carga de trabalho e construindo *Work Packages* ajustados consoante a capacidade dos dispositivos de computação.

[1] *clusters* que possuiem nós remotos com diferentes números de aceleradores

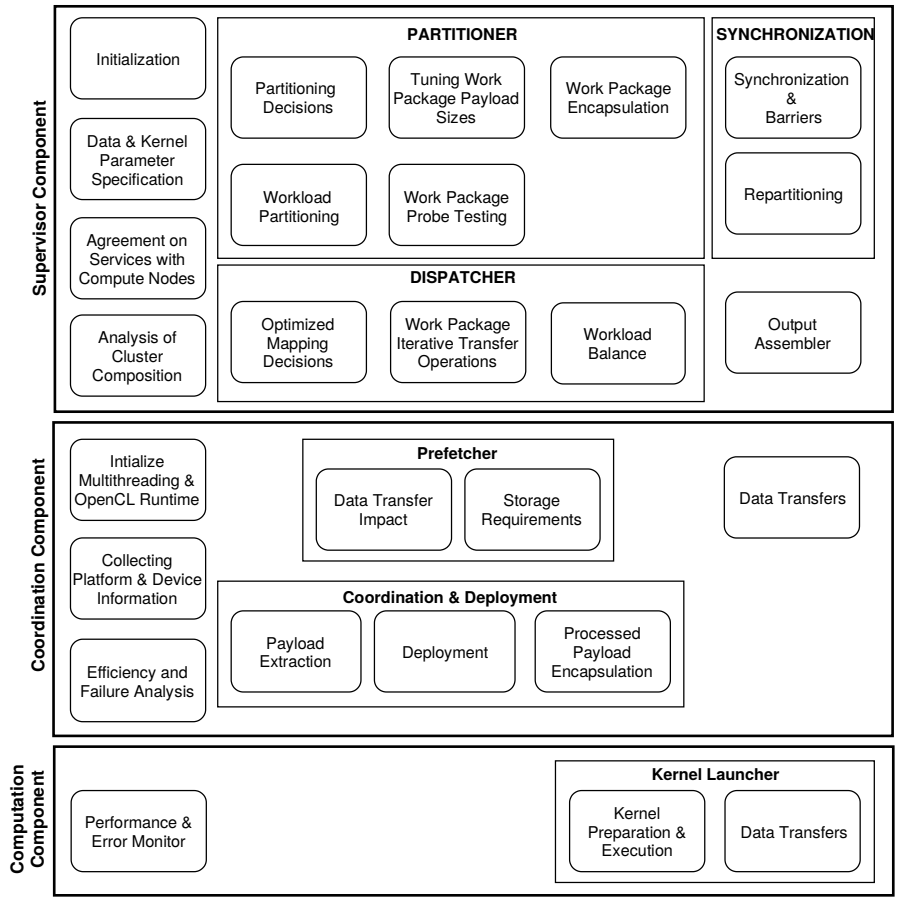


Figura 2.12: Principais componentes e módulos do clusterCL.

O envio efetivo dos *Work Packages* para os dispositivos de computação é realizado pelo módulo *Dispatcher*. Todos os *Work Packages* enviados são marcados com um identificador único que permite fazer o seu *tracking* após a sua submissão. Após essa operação, é adicionada à tabela de descritores informação como o estado de cada *Work Package* e o nó de destino. É também comum o módulo *Dispatcher* receber informações do *Coordination Component* de cada nó de computação, sobre o estado dos seus co-processadores, atualizando a informação na tabela de descritores de aceleradores.

Quando um pedido é recebido num nó de computação, são instanciadas no mesmo vários *threads*: por cada co-processador local é criada um *thread* para executar o *Coordination Component* e é criado uma *thread* adicional que lidará com os pedidos MPI. O *Coordination Component* é o intermediário entre os dispositivos de computação locais e o

front-end. Genericamente, possui como funções principais: a receção de *Work Packages* e a coordenação da submissão destes para execução nos dispositivos de computação.

Como a comunicação entre o *frontend* e os restantes nós pode tornar-se facilmente num *bottleneck*, um módulo *Prefetcher* minimiza esse impacto, trazendo para os nós *Work Packages* extras enquanto os dispositivos de computação estão ocupados. Para tal, o *Prefetcher* negoceia com o *Dispatcher* a quantidade de *Work Packages* transferidos de cada vez. Essa quantidade é definida inicialmente com base na capacidade do dispositivo de computação do nó de computação, podendo ser alterada após algumas execuções.

Os *Work Packages*, ao serem recebidos pelo *Prefetcher*, são reencaminhados para o módulo *Coordination and Deployment* para serem distribuídos pelos *Computation Components*, que têm como função básica orientar o procedimento de lançamento dos *kernels* e recolher os resultados do processo de computação para cada *Work Package* submetido.

Como não é conhecido de antemão o tempo que um dispositivo leva para executar um *Work Package*, é realizada uma análise rápida ao desempenho de cada dispositivo de computação, sendo este calculado e comparado com os outros dispositivos existentes. Se o desempenho de um determinado acelerador é muito menor do que outros, o módulo *Efficiency and Failure Analysis* sugere ao *Coordination Component* a exclusão do dispositivo do processo de computação. Esta decisão é baseada, fundamentalmente, nas considerações disponibilizadas pelo programador, que especifica se o tempo de execução ou o consumo de energia deve ser o parâmetro guia para a tomada de decisões. Estas decisões são comunicadas ao módulo *Prefetcher*, para que seja ajustado o envio de *Work Packages* entre o *Supervisor Component* e o *Coordination Component*.

O *Computation Component* é também responsável por configurar todos os recursos e objetos OpenCL que são necessários para lançar um *kernel*, assim como pelo arranque do *loop* que itera sobre comandos OpenCL na execução do *kernel*.

O resultado da computação deve ser organizado pelo *Supervisor Component* com base nos resultados parciais que são retornados pelo *Coordination Component* de cada nó.

2.8 Síntese Comparativa

Nesta secção faz-se uma síntese das abordagens descritas numa perspetiva comparada, incluindo já o rOpenCL. Os parâmetros considerados relevantes para a comparação são:

- **Driver OpenCL** - *Driver* integrado com o mecanismo ICD-Loader no sistema onde arranca a aplicação OpenCL.
- **Plataforma Global** - Unificação dos co-processadores do sistema distribuído numa plataforma OpenCL global.
- **Gestão de Objetos** - Forma de gestão de objetos remotos OpenCL (usando apontadores locais emparelhados com apontadores remotos; ou permitindo replicação).
- **Comunicação** - Biblioteca de comunicação utilizada para troca de mensagens entre os vários intervenientes do sistema distribuído.
- ***Kernels* 5.x** - Capacidade de executar com *kernels* recentes do Linux.
- **Continuidade** - O código da abordagem é mantido/atualizado regularmente.
- **Definição dos *hosts*** - Mecanismo de identificação dos nós remotos.
- **Linguagem de Desenvolvimento** - Linguagem principal da implementação.
- ***Callbacks*** - Suporta o mecanismo de *callbacks* do OpenCL para tratar eventos.

As tabelas 2.1 e 2.1 sintetizam a comparação entre as várias abordagens. Essa comparação é desenvolvida nos parágrafos seguintes.

Relativamente ao primeiro parâmetro da lista, verifica-se que apenas o dOpenCL possui um *driver* OpenCL no *host*. Contudo, esse *driver* não é compatível com o mecanismo ICD-Loader [47] nativo do OpenCL, levando a que no momento da instalação do dOpenCL, ficheiros da instalação nativa do OpenCL sejam reescritos, o que não acontece com o *driver* existente no rOpenCL, que é totalmente integrável com esse mecanismo.

Característica	Hybrid OpenCL	VCL	dOpenCL	clOpenCL
Driver OpenCL	Não	Não	Sim	Não
Plataforma Global	Sim	Sim	Sim	Não
Gestão de Objectos	Acesso Remoto	Replicação	Replicação	Acesso Remoto
Comunicação	Sockets	Sockets	Asio / GFC	Open-MX
Kernels 5.x	(código indisponível)	Executa	Não Executa	Não Executa
Continuidade	Não	Sim	Não	Não
Definição dos Hosts	Machinefile	Broker	Machinefile	EndPoints Open-MX
Linguagem Desenv.	C e C++	-	C e C++	C
Callbacks	Não	Sim	Sim	Não

Tabela 2.1: Comparação de abordagens OpenCL distribuídas (1/2).

Característica	SnuCL	clusterCL	rOpenCL
DriverOpenCL	Não	Não	Sim
Plataforma Global	Sim	Não	Não
Gestão de Objectos	Acesso Remoto	Acesso Remoto	Acesso Remoto
Comunicação	MPI	MPI	Sockets BSD
Kernels 5.x	Executa	Executa	Executa
Continuidade	Sim	Sim	Sim
Definição dos Hosts	Machinefile	Machinefile	Machinefile
Linguagem Desenv.	-	-	C
Callbacks	-	-	Não

Tabela 2.2: Comparação de abordagens OpenCL distribuídas (2/2).

Analisando a forma como os dispositivos são apresentados ao programador, apenas o clOpenCL e o clusterCL seguiram a mesma abordagem do rOpenCL, optando pela não unificação das plataformas, deixando assim a tarefa da gestão da carga de trabalho atribuída a cada co-processador para a lógica de programação das aplicações. Refira-se que um dos riscos decorrentes do uso de uma plataforma global é a alocação de carga a co-processadores remotos, apesar de existirem co-processadores locais, acessíveis através do OpenCL nativo, suficientemente capazes de oferecer o desempenho necessário. Se esse factor é levado ou não em conta pelas abordagens baseadas em plataforma global, é algo merecedor de investigação mais aprofundada, remetida para trabalho futuro.

A gestão dos objectos OpenCL é realizada, na maioria das implementações apresentadas, mantendo no nó *host* um registo dos apontadores remotos, que referenciam os objetos OpenCL instanciados nos vários serviços. No entanto, o dOpenCL e o VCL vão

mais longe, ao permitir a replicação de alguns desses objetos em vários serviços (como contextos OpenCL que podem incluídos aceleradores de diferentes) nós remotos.

A forma de comunicação entre o nó *host* e os nós remotos é crucial nestas abordagens, pois uma percentagem considerável do tempo consumido na execução de um pedido remoto é gasto na comunicação. O rOpenCL utiliza *sockets* BSD sobre TCP/IP para troca de mensagens, não sendo assim dependente de tecnologias de comunicação de nicho (por exemplo, Infiniband), mas podendo tirar proveito delas quando disponíveis (desde que suportem TCP/IP). As bibliotecas usadas pelas outras abordagens são diversificadas. Assim, o Hybrid OpenCL e o VCL utilizam também *sockets* para transmissão de mensagens, uma tecnologia antiga mas bastante competente a nível de desempenho, como será visível na seção 4.8 para o VCL. Noutra direcção, a comunicação no dOpenCL assenta no uso de uma *framework* de código fechado (GFC), embora tivesse começado por recorrer a comunicação assíncrona com base na biblioteca C++ Asio [41], uma tecnologia que, embora descontinuada pelo dOpenCL, mantém um bom desempenho – ver secção 4.8. Também o Open-MX, usado pelo clOpenCL, a um nível protocolar mais baixo, foi já descontinuado; adicionalmente, na avaliação feita neste trabalho (ver secção 4.8), exibiu um desempenho aquém do esperado, provavelmente devido às condicionantes do ambiente de avaliação, assente em máquinas virtuais. Finalmente, uma alternativa de comunicação de uso mais recente, e continuidade garantida, é a protagonizada pelo MPI, usado nas abordagens SnuCL e clusterCL. Esta poderia também ter sido a opção do rOpenCL; no entanto, entendeu-se prosseguir uma abordagem mais purista, focada essencialmente numa comunicação de mais baixo nível que aquela que seria a oferecida pelo MPI.

A capacidade de execução em *kernels* do Linux recentes está fortemente ligada com a frequência das atualizações realizadas às várias abordagens estudadas e com a tecnologia de comunicação utilizada. O rOpenCL, sendo recente, e o VCL, SnuCL e clusterCL, tendo atualizações recentes/regulares, têm todos a capacidade de executar em *kernels* atuais. Adicionalmente, a tecnologia de comunicação usada por todos eles é plenamente suportada nos *kernels* mais recentes. Com exeção do Hybrid OpenCL, que não tem o código fonte disponível, as restantes abordagens (dOpenCL e clOpenCL), como bibliotecas

de comunicação obsoletas (GFC e Open-MX) apenas são suportadas em *kernels* mais antigos, tornando inviável a sua execução em ambientes de computação mais recentes.

A forma como se definem os *hosts*/serviços que disponibilizam os co-processadores remotos varia também nas várias abordagens estudadas. Assim, apenas o VCL e o clOpenCL não realizam a identificação dos *hosts* remotos com recurso a um ficheiro do tipo *machinefile*. No caso do VCL, a identificação dos serviços remotos assenta numa associação (*bind*) a uma porta específica (255) onde todos os nós remotos devem estar ligados, o que permite a sua identificação pelo nó *host*. Um mecanismo semelhante é utilizado pelo clOpenCL, que faz uso da listagem disponível dos *endpoints* do Open-MX para contatar cada *host* remoto, conseguindo compreender se está ativo o *daemon* clOpenCL correspondente. Para o caso do SnuCL e clusterCL, o *machinefile* corresponde ao ficheiro de *hosts* que é comum usar quando se recorre a MPI para realizar a passagem de mensagens entre o nó inicial e os restantes nós. O *machinefile* existente no Hybrid OpenCL e dOpenCL é muito semelhante ao do rOpenCL, onde os *hosts* que devem fazer parte do *cluster* são definidos num ficheiro de texto, podendo ser adicionadas outras informações (porta de comunicação dos serviços remotos, tamanho máximo da trama de dados, etc.).

O suporte para o mecanismo de *callbacks* para tratamentos de eventos apenas é garantido no VCL e dOpenCL, sendo que no SnuCL e *clusterCL* não foi possível confirmar a existência deste mecanismo. Suportar *callbacks* permite que aplicações que façam uso destas abordagens distribuídas consigam executar funçõesOpenCL de forma assíncrona e detetar a ocorrência de determinados eventos. No caso do rOpenCL, o suporte para este mecanismo foi remetido para trabalho futuro, como será assumido no capítulo 5.

Por fim, e no que respeita às linguagens de programação usadas no desenvolvimento das várias abordagens estudadas neste capítulo, na generalidade (salvo o VCL, SnuCL e clusterCL, cuja situação particular não foi possível confirmar), todas foram desenvolvidas com recurso à linguagem C, C++, ou um misto de ambas. Esta opção é compreensível, não só por questões de desempenho, associadas ao facto de serem linguagens tradicionalmente usadas em programação de sistemas (em particular o C), como também devido ao próprio standard OpenCL, que define originalmente as suas primitivas em C e *bindings* para C++.

Capítulo 3

Arquitetura e Implementação

Feita a análise a abordagens congéneres no capítulo anterior, descreve-se agora a arquitetura do rOpenCL e os aspetos mais importantes do seu desenvolvimento e implementação.

3.1 Prólogo

Uma análise cuidada das soluções apresentadas no capítulo 2 permitiu compreender o estado geral das implementações que suportam um modelo distribuído do OpenCL, possibilitando que o rOpenCL fosse desenhado para responder a algumas das limitações identificadas e acrescentar as suas próprias contribuições. Assim, foi possível concluir que a principal característica que diferencia o rOpenCL é a transparência: aplicações OpenCL não precisam de ser recompiladas para conseguir usar e explorar co-processadores remotos como se de locais se tratassem. Desta forma, com rOpenCL instalado no nó *host*, uma aplicação OpenCL é capaz de usar qualquer combinação de plataformas/dispositivos locais e plataformas/dispositivos remotos. Para atingir este nível de transparência, o desenho e desenvolvimento do rOpenCL assentou em quatro pilares fundamentais:

- oferecer as camadas de *software* necessárias à execução remota de primitivas OpenCL;
- permitir que qualquer aplicação OpenCL possa tirar partido dos dispositivos remotos sem necessitar de ser alterada ou ligada a bibliotecas adicionais;

- garantir a semântica original das primitivas OpenCL executadas remotamente;
- assegurar compatibilidade com as mais conhecidas implementações OpenCL (*NVIDIA*, *AMD*, *Intel Runtime*, POCL).

3.2 Versão Inicial

Como já referido, o rOpenCL possui uma forte ligação com o clOpenCL: o estudo do código fonte deste, nomeadamente no que diz respeito à gestão de objetos remotos e troca de mensagens, foi fulcral no desenvolvimento do primeiro protótipo do rOpenCL.

Desta forma, o passo inicial consistiu em revisitar o clOpenCL, procurando reconstruir a execução remota das primitivas por ele implementadas, agora com recurso a *sockets* BSD e TCP/IP. De facto, como referenciado no capítulo 2, o *Open-MX*, utilizado pelo clOpenCL para a troca de mensagens entre os componentes da sua arquitetura, apenas se encontra suportado em versões do *kernel* do *Linux* até à geração 4.

Assim para manter a ordem de grandeza dos tempos de execução [26] das aplicações clOpenCL produzidos pelo uso do *Open-MX*, fizeram-se alterações ao código do clOpenCL para se passar a utilizar o protocolo UDP na troca de mensagens; este, sendo um protocolo *connectionless*, permitiu reduzir bastante a porção do tempo consumido na comunicação. A figura 3.1 apresenta a arquitetura do rOpenCL modificado desta forma, e que corresponde à primeira iteração do rOpenCL.

Nesta iteração, quando a aplicação OpenCL arranca, para além da intersecção das funções OpenCL, a função de arranque *main* também é intersectada, permitindo que seja feito o reconhecimento dos nós disponíveis para receber pedidos de execução remota. Esse reconhecimento começa pelo envio, por difusão (*broadcast*), de um pedido específico, através da *interface* de rede utilizada para a troca de mensagens com os nós remotos. Quando os serviços de execução remota iniciam nesses nós, ligam-se (*bind*) a duas portas: a primeira atende pedidos recebidos via *broadcast* (pedidos de ligação), e a segunda recebe os pedidos diretos (*unicast*) de execução de funções OpenCL.

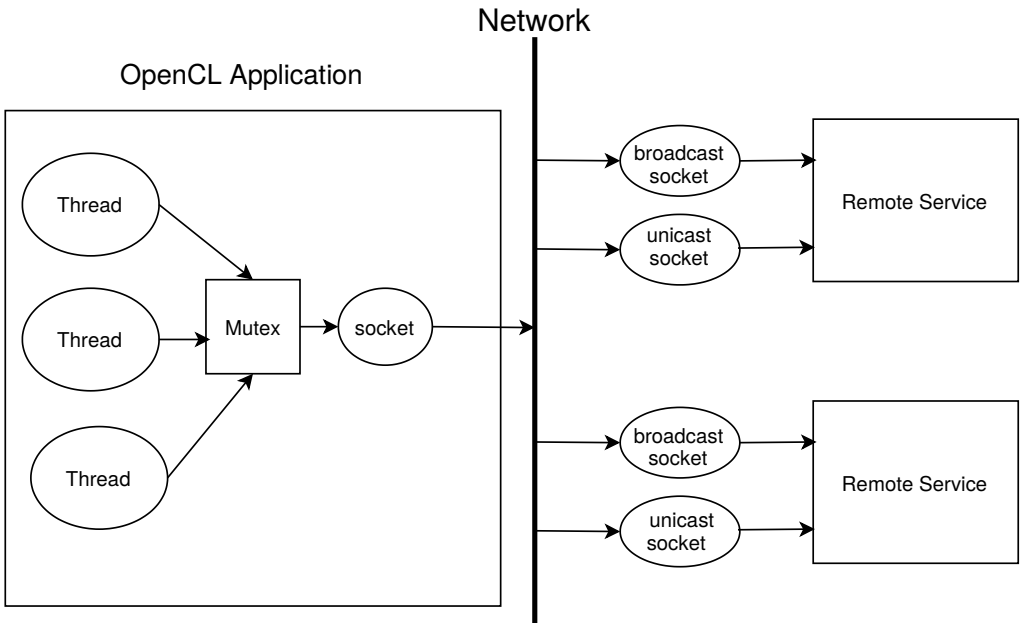


Figura 3.1: Arquitetura da versão 1.0 do rOpenCL.

Portanto, no momento da interseção da função *main*, a aplicação *host* envia um pedido, por *broadcast*, para identificar quais os nós disponíveis para execução remota. Todos os nós à escuta respondem, informando da sua disponibilidade. Ao analisar as respostas, a aplicação *host* regista, para cada nó remoto, o seu endereço IP e a porta *unicast*, para posterior envio direto de pedidos de execução OpenCL. Nesta fase, para além do reconhecimento dos nós remotos, é também criado o único descritor de *socket* que será usado por todas as *threads* da aplicação *host* na troca de mensagens.

Apesar das modificações na camada de comunicação, esta arquitetura mantém algumas limitações fundamentais do clOpenCL: i) apenas uma aplicação OpenCL pode interagir com os serviços remotos; ii) a execução remota de algumas primitivas OpenCL continua a obrigar a que os dados associados tenham que ser enviados em tramas separadas, uma vez que, para certos contextos de execução, o tamanho total pode exceder o valor da Maximum Transmission Unit (MTU) da rede; esta fragmentação é agora justificada pelo uso do protocolo UDP para o envio de mensagens, sendo recomendável [48] o envio de mensagens com tamanho inferior à MTU.

Testes realizados evidenciaram ainda que a arquitetura proposta na figura 3.1 não era

satisfatória sob o ponto de vista do desempenho, uma vez que gerava grande contenção no acesso ao *socket* único usado pela aplicação OpenCL. Tal efeito deve-se à presença do *mutex* no acesso ao descritor de *socket*, garantido que cada *thread* progride só após receber a resposta ao pedido que submeteu. Percebeu-se ainda que o mecanismo de reconhecimento de nós poderia não funcionar corretamente, uma vez que limitar o tráfego da interface de *broadcast* é uma prática comum em muitas configurações de rede.

3.3 Paralelizar no *host*

Para tentar ultrapassar os problema de desempenho da abordagem inicial, optou-se por garantir que cada *thread* dispõe de um *socket*, criado *just-in-time*, para cada transação com um serviço remoto. Para o mecanismo de reconhecimento dos nós, o recurso a um *hostfile* foi a solução encontrada, ou seja, quando uma aplicação OpenCL arranca, na interseção da função *main* contactam-se apenas os nós registados no *hostfile*. A figura 3.2 ilustra a arquitetura da segunda iteração do rOpenCL, baseada nestas modificações.

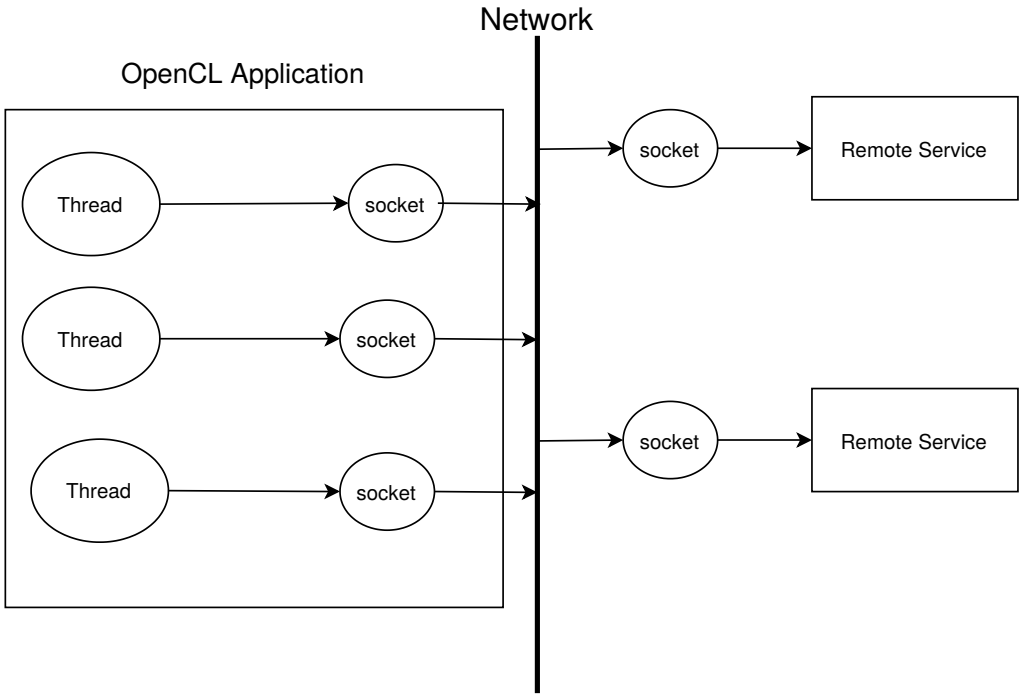


Figura 3.2: Arquitetura da versão 1.1 do rOpenCL.

Assim, sempre que um *thread* necessita de realizar uma transação com um serviço remoto, cria um novo *socket*, que decarta logo após a conclusão da transação (recepção da reposta correspondente). Genericamente, se uma aplicação invocar funções OpenCL n vezes, serão utilizados outros tantos *sockets*. No entanto, apesar desta alteração ter resolvido os problemas de contenção da versão anterior, trouxe consigo novos problemas.

Na primeira arquitetura (figura 3.1), refreados pela presença do *mutex* no controle do acesso ao *socket*, todos os pedidos submetidos para os serviços remotos são serializados logo no lado *host* da aplicação OpenCL, garantindo que nenhum serviço remoto recebe mais do que um pedido ao mesmo tempo, e também que não há mais que um serviço a processar pedidos em simultâneo. Porém, na segunda arquitetura (figura 3.2), passou a ser possível à mesma aplicação OpenCL enviar vários pedidos, em paralelo, seja para um mesmo serviço remoto, seja até para diferentes serviços. Ora, isto revelou a incapacidade dos serviços remotos processarem de forma correta esses pedidos, devido à forma como ainda estavam a lidar com a sua recepção, igual à seguida na primeira versão da arquitetura.

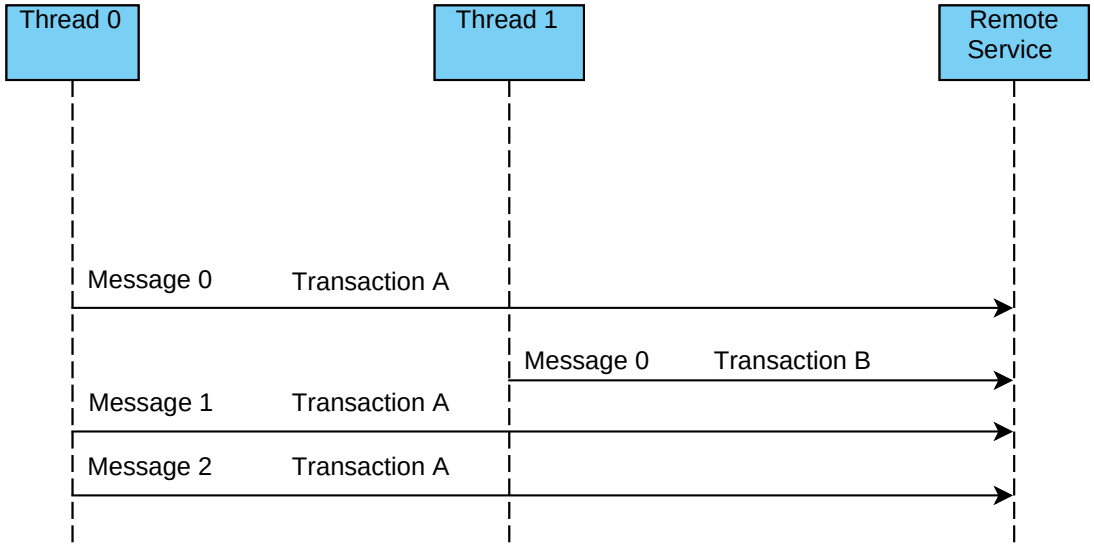


Figura 3.3: Diagrama de sequência de duas transações em simultâneas.

Para compreender a limitação em causa, atente-se na figura 3.3, onde se podem ver dois *threads* a enviar tramas para o mesmo serviço remoto. Recorde-se ainda que, como já explicado na secção 3.2, para pedidos cujo tamanho exceda o valor da MTU *default*, é

necessário fragmentar o pedido e enviá-lo em tramas / mensagens diferentes.

Assim, o pedido correspondente à transação A tem de se desdobrar em várias mensagens (M0, M1 e M2), ao passo que para a transação B o pedido é enviada apenas com base numa só mensagem (M0). Analisando a sequência de envios da figura 3.3, verifica-se que, devido à fragmentação, é possível que mensagens de pedidos diferentes se misturem na recepção, caso da mensagem 0 da transação B, que será interpretada como sendo a mensagem 1 da transação A. Isto acontece porque quando a mensagem 0 da transação A é recebido em primeiro lugar, a mesma indica um tamanho total a receber, para o qual vão contribuir mensagens que deveriam chegar logo a seguir (mensagem 1 e 2 da mesma transação); desta forma, a mensagem 0 da transação B é vista como parte da transação A dado que a receção das mensagens desta ainda está em curso.

Outro problema potencial deriva do próprio UDP: mensagens da mesma transação podem ser recebidas num serviço por ordem diferente da do envio e nesta versão da arquitetura não está acautelada a re-ordenação. Esta limitação é herdada da implementação assente no *Open-MX*, que assumia execução numa rede local e de elevada fiabilidade.

3.4 Paralelizar no serviço

A resolução dos problemas exibidos pela segunda versão da arquitetura do rOpenCL implicou algumas alterações significativas, para poder continuar a usar o protocolo UDP:

- No lado da aplicação *host*: marcar todas as tramas enviadas, acrescentando um identificador único de transação, assim como o número total de tramas da transação.
- No serviço remoto: acrescentar a capacidade de reconhecer cada trama como parte de uma transação, armazenar temporariamente as tramas até completar a recolha de todas as tramas da mesma transação e, se necessário, reordenar as tramas pela ordem correcta (que pode não coincidir com a ordem de receção).

Assim, quando uma aplicação *host* pretende iniciar uma transação OpenCL com um serviço remoto, é necessário aferir se o pedido da transação irá necessitar apenas de uma

ou de várias mensagens. A figura 3.4 oferece um exemplo em que um pedido de transação, envolvendo 2070 *bytes* de dados da camada aplicacional, incorre em 3 mensagens.

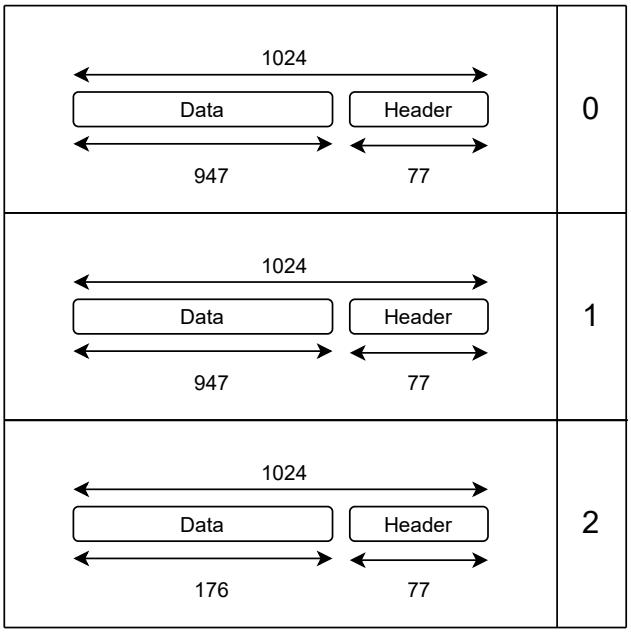


Figura 3.4: Mensagens UDP de um pedido rOpenCL com 2070 *bytes* de dados OpenCL.

No exemplo fornecido, é visível que cada trama possui um tamanho de 1024 *bytes*, dos quais 77 são consumidos por um cabeçalho (*header*), restando 947 *bytes* para o transporte dos dados aplicacionais. De notar que a tudo isto se acrescenta o próprio cabeçalho UDP, sendo que, em qualquer caso, a dimensão total da trama UDP não será, garantidamente, superior ao valor mínimo do MTU (habitualmente assumido como sendo 1500 *bytes*).

Os campos que compõem o cabeçalho de cada mensagem são os da estrutura da listagem 3.1, incluindo: o identificador único da transação (*id_transaction*); a posição da trama enviada (*part_of_packets*) em relação ao número total de tramas (*total_packets*) da transação; o identificador único (*id_func*) da função OpenCL a executar; a dimensão total dos dados (parâmetros) a passar à função OpenCL (*size_buffer*).

O identificador da transação inclui 4 sub-campos: endereço IP; identificador PID do processo da aplicação OpenCL cliente; identificador TID do thread [49] da mesma aplicação; um número auto incremental. Note-se que o uso do endereço IP do *host* da aplicação OpenCL, na geração do identificador de uma transação, acomoda a possibilidade

Listagem 3.1: Estrutura do cabeçalho de uma trama.

```
struct{
    char id_transaction [SIZE_TRANSACTION_ID];
    int part_of_packets;
    int total_packets;
    char id_func;
    int size_buffer;
}  rOpenCL_header_udp;
```

de as transações terem origem em diferentes aplicações OpenCL de diferentes *hosts*, o que representa outra evolução importante face à iteração anterior da arquitetura.

Nos serviços remotos, houve a necessidade de alocar estruturas de dados de acesso eficiente para armazenar os pedidos, tantos os incompletos (em construção, aguardando pela chegada de todas as mensagens associadas), como os completos (prontos a executar ou já em execução). Várias estruturas foram consideradas, nomeadamente tabelas de *hashing* e árvores binárias, tendo a opção recaído nesta última, especificamente na implementação de *Red-Black Trees* oferecida pela *libc*, devido ao bom desempenho exibido.

Adicionalmente, o processamento dos pedidos desdobrou-se em dois *threads* distintos:

- *thread 0*: recebe as mensagens da rede, identifica a transação de que fazem parte, regista-a (se for nova) numa *árvore 0* e acrescenta nessa mesma estrutura de dados as mensagens relativas à mesma transação; uma vez reassemblada a totalidade de um pedido, move-o para uma *árvore 1* e notifica a *thread 1* para o executar;

- *thread 1*: executa os pedidos completos inseridos pela *thread 0* na *árvore 1*.

A sincronização entre as *threads* assenta num contador de pedidos completos, protegido por uma *variável mutex* e uma *variável de condição*, pois usaram-se POSIX *Threads*.

Com todos estes melhoramentos, passou a ser possível conduzir várias transações, em paralelo, entre um nó *host* de uma aplicação OpenCL e os serviços remotos. Por seu turno, os serviços remotos ficaram habilitados a realizar transações com múltiplas aplicações OpenCL, provenientes de apenas um *host* ou de vários.

Todavia, apesar do uso de uma rede isolada, os testes realizados após estas alterações revelaram perca de dados em algumas transações, criando situações de espera indefinida, por parte das aplicações OpenCL, da resposta aos seus pedidos.

3.5 Perdas de dados na troca de mensagens

Como abordado na seção anterior, apesar das precauções tomadas, garantindo que cada mensagem encaixa no MTU da rede, e que a rede usada nos testes era uma rede isolada, registou-se perda de pacotes UDP. Nesta secção documentam-se os esforços realizados no sentido de compreender melhor o problema e procurar soluções para o mesmo.

Assim, realizaram-se testes com o *iperf3* [50], uma ferramenta que permite medir a largura de banda máxima possível em redes IP. Os testes foram feitos recorrendo a duas máquinas e 4 cenários, combinando dois valores diferentes para a quantidade de dados a enviar (1 *GByte* ou 4 *GBytes*), com dois valores diferentes para o nº de *threads* envolvidas no envio (1 *thread* ou 4 *threads*). As figuras 3.5 a 3.8 mostram o *output* dos quatro testes.

```
Accepted connection from 192.168.45.40, port 41290
[  5] local 192.168.45.45 port 1050 connected to 192.168.45.40 port 57518
[ ID] Interval           Transfer     Bandwidth       Jitter    Lost/Total Datagrams
[  5]   0.00-1.00   sec  59.9 MBytes   502 Mbits/sec  0.017 ms  669/8333 (8%)
[  5]   1.00-1.73   sec  0.00 Bytes  0.00 bits/sec  0.017 ms  0/0 (0%)
```

Figura 3.5: Teste com *iperf3*: envio de 1 *GByte* por 1 *thread*.

```
Accepted connection from 192.168.45.40, port 41294
[  5] local 192.168.45.45 port 1050 connected to 192.168.45.40 port 51787
[ ID] Interval           Transfer     Bandwidth       Jitter    Lost/Total Datagrams
[  5]   0.00-1.00   sec  32.4 MBytes   271 Mbits/sec  0.017 ms  741/4885 (15%)
[  5]   1.00-2.00   sec  0.00 Bytes  0.00 bits/sec  0.017 ms  0/0 (0%)
```

Figura 3.6: Teste com *iperf3*: envio de 4 *GBytes* por 1 *thread*.

As figuras revelam perda de dados nos quatro cenários testados, variando entre 8% a 18%. A ocorrência destas perdas faz com que os serviços remotos fiquem com transações incompletas pendentes por tempo indefinido.

Com base em pesquisas [51], concluiu-se que o *kernel* do *Linux* não está otimizado para tirar partido de redes de 10 Gbps na transmissão de dados com recurso a UDP. As percas

```
Accepted connection from 192.168.45.40, port 41298
[  5] local 192.168.45.45 port 1050 connected to 192.168.45.40 port 58380
[  6] local 192.168.45.45 port 1050 connected to 192.168.45.40 port 38361
[  8] local 192.168.45.45 port 1050 connected to 192.168.45.40 port 54523
[ 10] local 192.168.45.45 port 1050 connected to 192.168.45.40 port 59945
[ ID] Interval           Transfer     Bandwidth       Jitter    Lost/Total Datagrams
[  5]   0.00-1.00   sec  5.55 MBytes  46.5 Mbits/sec  0.057 ms  159/870 (18%)
[  6]   0.00-1.00   sec  5.56 MBytes  46.6 Mbits/sec  0.057 ms  158/870 (18%)
[  8]   0.00-1.00   sec  5.50 MBytes  46.1 Mbits/sec  0.050 ms  164/868 (19%)
[ 10]   0.00-1.00   sec  5.54 MBytes  46.4 Mbits/sec  0.054 ms  159/868 (18%)
[SUM]   0.00-1.00   sec  22.2 MBytes   186 Mbits/sec  0.055 ms  640/3476 (18%)
```

Figura 3.7: Teste com *iperf3*: envio de 1 *GByte* por 4 *threads*.

```
Accepted connection from 192.168.45.40, port 41296
[  5] local 192.168.45.45 port 1050 connected to 192.168.45.40 port 54289
[  6] local 192.168.45.45 port 1050 connected to 192.168.45.40 port 33289
[  8] local 192.168.45.45 port 1050 connected to 192.168.45.40 port 53125
[ 10] local 192.168.45.45 port 1050 connected to 192.168.45.40 port 55019
[ ID] Interval           Transfer     Bandwidth       Jitter    Lost/Total Datagrams
[  5]   0.00-1.00   sec  5.42 MBytes  45.4 Mbits/sec  0.065 ms  152/846 (18%)
[  6]   0.00-1.00   sec  5.41 MBytes  45.3 Mbits/sec  0.062 ms  154/846 (18%)
[  8]   0.00-1.00   sec  5.47 MBytes  45.8 Mbits/sec  0.054 ms  146/846 (17%)
[ 10]   0.00-1.00   sec  5.47 MBytes  45.8 Mbits/sec  0.061 ms  145/845 (17%)
[SUM]   0.00-1.00   sec  21.8 MBytes   182 Mbits/sec  0.060 ms  597/3383 (18%)
```

Figura 3.8: Teste com *iperf3*: envio de 4 *GBytes* por 4 *threads*.

detetadas na transmissão acontecem devido ao esgotamento dos *buffers* de receção e envio de dados associados aos *sockets* no *kernel*. De forma mais sucinta, analisando o código do *kernel* [52], verifica-se que o mesmo começa a descartar pacotes automaticamente (e silenciosamente) quando os *buffers* de receção e/ou envio ficam saturados.

Uma abordagem possível para mitigar o problema passa por um conjunto de parâmetros [53] que permite aumentar a capacidade de receção/envio dos *buffers* dos *sockets*. Contudo, esta estratégia é insuficiente, pois existe um tamanho máximo para cada *buffer*, e esse tamanho pode ficar aquém das necessidades de determinadas transações (tudo depende da quantidade máxima de dados passada como parâmetro às primitivas OpenCL).

3.5.1 Código de erro CL_ERROR_NETWORK

A fim de contornar o problema da espera indefinida, referido no final da secção 3.4, explorou-se um mecanismo de *timeout* no *socket* usado com a primitiva *recvfrom*, sendo o valor do *timeout* (500 s) definido através da primitiva *setsockopt*.

Assim, quando o *timeout* expira sem ter chegado nenhuma resposta ao pedido da transacção em curso, a primitiva *recvfrom* desbloqueia e torna-se possível informar a camada aplicacional do ocorrido. Essa informação corresponde ao retorno do erro CL_ERROR_NETWORK, que representa uma extensão ao *standard* do OpenCL.

Note-se, porém, que a extensão CL_ERROR_NETWORK resolveu apenas o bloqueio indefinido das aplicações, e não a perda de mensagens. Com a adição deste código de erro, fica assegurado que, apesar da existência das perdas (pelo motivos já explicados), todas as aplicações que façam uso do rOpenCL, configurado com o protocolo UDP, conseguem terminar e informar o programador da ocorrência de possíveis perdas. Tal ocorrência na troca de mensagens impedia que a implementação desenvolvida fosse submetida a testes mais rigorosos, uma vez que, mesmo com aumento do tamanho dos *buffers* de receção e envio (referido na secção 3.5), em contextos que envolviam troca de grandes quantidades de dados era inevitável a ocorrência de perdas.

3.6 Comunicação com recurso a TCP

A resolução do problema das perdas de mensagens, que são inevitáveis quando se usa UDP num contexto de trocas muito rápidas de grandes quantidades de dados, acabou por ser resolvida com recurso ao protocolo TCP. Os detalhes associados a essa opção são discutidos na secção 3.8. Refira-se, no entanto, que o código do rOpenCL reteve a capacidade de poder usar o UDP, se pretendido.

Permitir a escolha do protocolo de comunicação levou à reformulação de uma parte da implementação já criada. A primeira alteração passou por isolar as funções responsáveis pelo envio das mensagens. Tal justifica-se pelo fato de no protocolo UDP não existir a necessidade de invocar a primitiva *connect* da API dos *sockets* BSD para estabelecer previamente a ligação; por seu turno, no caso do protocolo TCP, não existe a necessidade de acautelar fragmentação na camada aplicacional.

Nos serviços remotos também foram necessárias alterações para acomodar a comunicação recorrendo ao TCP. Assim, adicionaram-se dois *threads*, em cada serviço, com

funções semelhantes às dos *threads* usados com UDP (rever seção 3.4), mas simplificando a tarefa do *thread 0*, que deixou de ter necessidade de lidar com pedidos fragmentados.

De qualquer forma, a opção entre o UDP e o TCP não pode ser feita em tempo de execução, pela aplicação OpenCL; em vez disso, tem de ser feita no ato da compilação do *driver* e serviços do rOpenCL, uma vez que o código de suporte ao UDP e ao TCP é compilado, de forma mutuamente exclusiva, com recurso a simples diretivas #ifdef/#endif. Isto poupa algum tempo durante a execução, pois evita a necessidade de estar constantemente a verificar qual o protocolo ativo, para seguir o trajeto respetivo no código.

3.7 Integração com o ICD-Loader

Resolvida a questão da comunicação, partiu-se para a melhoria da usabilidade. De facto, à semelhança do clOpenCL, neste estágio do desenvolvimento, a utilização do rOpenCL por uma aplicação OpenCL implicaria ligar o código desta aplicação, em tempo de compilação, com a biblioteca do rOpenCL, prevenindo o uso deste por aplicações pré-compiladas.

Após pesquisas sobre o assunto [54], percebeu-se que um dos mecanismos fundamentais de uma implementação do OpenCL em conformidade com o respetivo standard é ICD [47]. Este mecanismo permite a coexistência de implementações OpenCL de múltiplos *vendors* no mesmo sistema. A integração do rOpenCL com este mecanismo tornou-se a solução para o problema da usabilidade identificado.

Adicionalmente, ao contrário do que acontecia até aqui, em que os pedidos de execução OpenCL associados a dispositivos locais passavam também pela biblioteca do rOpenCL, com esta integração o rOpenCL passa apenas a expor e a gerir dispositivos remotos através do seu *driver*. Esta separação é claramente visível na figura 3.9.

Apesar desta solução resolver o problema da usabilidade do rOpenCL, possibilitando a qualquer aplicação OpenCL tirar partido do rOpenCL sem necessidade de ser recompilada ou alterada, implicou mudar a forma de gestão dos apontadores para objetos OpenCL. Isto porque a forma como essa gestão era feita no clOpenCL, e que foi adoptada num primeiro momento pelo rOpenCL, era incompatível com a integração com o *ICD-Loader*.

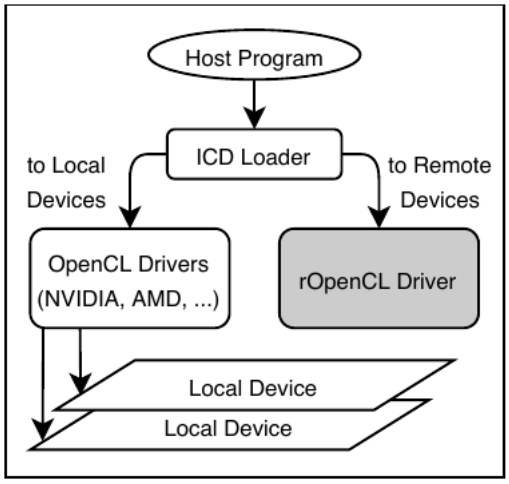


Figura 3.9: Integração do rOpenCL com o *ICD-Loader*.

3.7.1 Gestão dos apontadores associados ao ICD-Loader

Para que o mecanismo *ICD-Loader* consiga lidar corretamente com as transacções (pedidos e respostas) associadas ao rOpenCL, é necessário um mecanismo de gestão de objetos OpenCL compatível com o *ICD-Loader*. Esse mecanismo efetua o mapeamento de apontadores locais para objetos OpenCL (visíveis à aplicação OpenCL no *host* de arranque) em apontadores para objetos remotos equivalentes (válidos nos serviços remotos onde as chamadas OpenCL são efetivamente executadas). Esta associação, expressa através da estrutura da listagem 3.2, acontece à medida que as invocações OpenCL são colocadas ao rOpenCL, sendo apenas necessário para os tipos de dados abstratos do OpenCL[1].

A estrutura de mapeamento inclui: um apontador para um objeto OpenCL local (*object_local*), um apontador para o objeto OpenCL remoto correspondente (*object_remote*), um apontador para um objeto usado para identificar o serviço rOpenCL onde o apontador remoto é válido (*daemon*) e um par de apontadores (*ptr_local* e *ptr_remote*) que permitem referenciar *buffers* adicionais, garantindo o correto funcionamento de um grupo restrito de funções OpenCL (e.g., *clCreateBuffer*). A necessidade do apontador *daemon* deriva de poder haver vários mapeamentos em que o serviço remoto é o mesmo, evitando-se assim replicar o mesmo endereço desse serviço remoto em diferentes mapeamentos.

[1]Ver `https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/dataTypes.html`

Listagem 3.2: Estrutura de mapeamento entre objetos OpenCL locais e remotos.

```
typedef struct _r_opencl_object_t {
        remote_daemon_t* daemon;
        void* object_local;
        void* object_remote;
        void* ptr_local;
        void* ptr_remote;
} r_opencl_object_t;
```

Listagem 3.3: Endereço de um serviço remoto.

```
typedef struct _remote_daemon_t {
    struct sockaddr_in addr;
    char ip_addr[16];
} remote_daemon_t;
```

Todos os mapeamentos entre objetos locais e remotos são geridos através de uma *Red-Black Tree* (árvore de mapeamento) mantida pelo *driver* do rOpenCL. Essa estrutura de dados começa a ser preenchida aquando da invocação de funções que envolvam a identificação de plataformas OpenCL[2], e que habitualmente são as primeiras primitivas OpenCL a serem invocadas por uma aplicação OpenCL. Quando essa invocação acontece, o *driver* rOpenCL contacta os serviços remotos dos nós listados no seu *hostfile*. As funções de identificação das plataformas são executadas nesses nós, sendo retornados ao *driver* os apontadores remotos criados pelos serviços durante a identificação. De seguida, o *driver* determina os apontadores locais correspondentes (com base num mecanismo explicado mais adiante) e insere, na árvore de mapeamento, as correspondências entre apontadores locais e remotos, atualizando também o campo *daemon* em conformidade.

Após a inicialização da árvore de mapeamento de objetos, o procedimento seguido para as restantes primitivas OpenCL é comum: cada primitiva OpenCL apresentada ao *driver* do rOpenCL terá vários parâmetros, alguns dos quais de tipos de dados abstratos do OpenCL; todos os apontadores locais para parâmetros desse género são usados como

[2]Explicitamente, no caso de *clGetPlatformIDs*, ou implicitamente, no caso de *clGetDeviceIDs*.

chaves de procura na árvore de mapeamento; mas um desses parâmetros é considerado parâmetro *principal* (ver tabelas 3.1, 3.2), sendo obrigatória a existência do mapeamento no respetivo apontador remoto para que o pedido OpenCL seja encaminhado para o serviço remoto correspondente; para os restantes parâmetros abstratos, considerados *secundários*, a ausência de mapeamento na árvore implica a definiação do valor *NULL* para o apontador remoto; caso não seja possível mapear o parâmetro *principal*, o *driver* retorna imediatamente para a camada aplicacional, com o erro *CL_INVALID_VALUE*.

No serviço remoto, recebido um pedido, o mesmo é desserializado, identificando-se a primitiva OpenCL a executar bem como os parâmetros a passar-lhe; estes incluem os apontadores remotos (agora locais) que foram determinados pelo *driver* do rOpenCL, no lado *host* da aplicação. Terminada a execução da primitiva, é construída a mensagem de resposta e enviada ao originador do pedido, completando assim o processo de invocação remota (transação). Neste ponto, é importante salientar que os serviços não mantêm qualquer estado que relacione uma transação com outra; essa possível relação, a existir, é mantida a um nível mais baixo, nomeadamente assente em *buffers* dos dispositivos OpenCL, cujo conteúdo pode ser reutilizado por diferentes primitivas OpenCL.

Após a receção da resposta a um pedido, o *driver* rOpenCL desserializa a mensagem recebida e começa a construir a resposta a devolver para a camada aplicacional. Essa resposta terá de envolver apontadores para objectos locais e poderá implicar definir novos mapeamentos, nomeadamente quando há objectos remotos abstractos recém-criados, logo ainda por mapear. Nesta última situação, o mapeamento de um objecto remoto num local que seja reconhecível como válido pelo *ICD-Loader* corresponde a definir um apontador local como um dos campos da estrutura rOpenCL_dispatch da listagem 3.4. O registo de um novo mapeamento implicará, ainda, a definição do campo *daemon* em conformidade.

De referir ainda que na fase do desenvolvimento do rOpenCL em que ocorreu a integração com o *ICD-Loader*, cada transação entre o *driver* rOpenCL e um serviço remoto começava com a criação de um *socket* de suporte a essa transacção e terminava com o fecho desse *socket* uma vez concluída a transação. Esta abordagem viria a ser substituída por outra mais eficiente, descrita na secção 3.8.1.

Tabela 3.1: *Parâmetros Principais* das primitivas OpenCL (1/2).

Primitiva OpenCL	**Parâmetro Principal**
clGetPlatformIDs	-
clGetPlatformInfo	cl_platform_id platform
clGetDeviceIDs	-
clGetDeviceInfo	cl_device_id device
clCreateSubDevices	cl_device_id in_device
clRetainDevice	cl_device_id device
clReleaseDevice	cl_device_id device
clCreateContext	cl_device_id* devices
clCreateContextFromType	cl_context_properties *properties
clRetainContext	cl_context context
clReleaseContext	cl_context context
clGetContextInfo	cl_context context
clGetExtensionFunctionAddressForPlatform	cl_platform_id platform
clCreateCommandQueue	cl_context context
clRetainCommandQueue	cl_command_queue command_queue
clReleaseCommandQueue	cl_command_queue command_queue
clGetCommandQueueInfo	cl_command_queue command_queue
clCreateBuffer	cl_context context
clCreateSubBuffer	cl_mem buffer
clCreateSubBuffer	cl_mem buffer
clEnqueueReadBuffer	cl_command_queue command_queue
clEnqueueReadBufferRect	cl_command_queue command_queue
clEnqueueWriteBuffer	cl_command_queue command_queue
clEnqueueWriteBufferRec	cl_command_queue command_queue
clEnqueueFillBuffer	cl_command_queue command_queue
clEnqueueCopyBuffer	cl_command_queue command_queue
clEnqueueCopyBufferRect	cl_command_queue command_queue
clEnqueueMapBuffer	cl_command_queue command_queue
clRetainMemObject	cl_mem memobj
clReleaseMemObject	cl_mem memobj
clSetMemObjectDestructorCallback	cl_mem memobj
clEnqueueUnmapMemObject	cl_command_queue command_queue
clEnqueueMigrateMemObjects	cl_command_queue command_queue
clGetMemObjectInfo	cl_mem memobj
clCreateProgramWithSource	cl_context context
clCreateProgramWithBinary	cl_context context
clCreateProgramWithBuiltInKernels	cl_context context
clRetainProgram	cl_program program
clReleaseProgram	cl_program program
clBuildProgram	cl_program program

Tabela 3.2: *Parâmetros Principais* das primitivas OpenCL (2/2).

Primitiva OpenCL	**Parâmetro Principal**
clCompileProgram	cl_program program
clLinkProgram	cl_program program
clUnloadPlatformCompiler	cl_platform_id platform
clGetProgramInfo	cl_program program
clGetProgramBuildInfo	cl_program program
clCreateKernel	cl_program program
clCreateKernelsInProgram	cl_program program
clRetainKernel	cl_kernel kernel
clReleaseKernel	cl_kernel kernel
clSetKernelArg	cl_kernel kernel
clGetKernelInfo	cl_kernel kernel
clGetKernelWorkGroupInfo	cl_kernel kernel
clGetKernelArgInfo	cl_kernel kernel
clEnqueueNDRangeKernel	cl_command_queue command_queue
clEnqueueTask	cl_command_queue command_queue
clEnqueueNativeKernel	Não implementada
clCreateUserEvent	cl_context context
clSetUserEventStatus	cl_event event
clWaitForEvents	const cl_event *event_list
clGetEventInfo	cl_event event
clSetEventCallback	cl_event event
clRetainEvent	cl_event event
clReleaseEvent	cl_event event
clEnqueueMarkerWithWaitList	cl_command_queue command_queue
clEnqueueBarrierWithWaitList	cl_command_queue command_queue
clGetEventProfilingInfo	cl_event event
ClCreateImage	cl_context context
clGetSupportedImageFormats	cl_context context
clEnqueueReadImage	cl_command_queue command_queue
clEnqueueWriteImage	cl_command_queue command_queue
clEnqueueFillImage	cl_command_queue command_queue
clEnqueueCopyImage	cl_command_queue command_queue
clEnqueueCopyImageToBuffer	cl_command_queue command_queue
clEnqueueCopyBufferToImage	cl_command_queue command_queue
clEnqueueMapImage	cl_command_queue command_queue
clGetMemObjectInfo	cl_mem memobj
clGetImageInfo	cl_mem memobj
clCreateSampler	cl_context context
clRetainSampler	cl_sampler sampler
clReleaseSampler	cl_sampler sampler
clGetSamplerInfo	cl_sampler sampler

Listagem 3.4: Objectos do rOpenCL compatíveis com o ICD-Loader.

```
struct _cl_icd_dispatch  rOpenCL_dispatch =
{
        &IPname(clGetPlatformIDs),
        &IPname(clGetPlatformInfo),
        &IPname(clGetDeviceIDs),
        &IPname(clGetDeviceInfo),
        &IPname(clCreateContext),
        &IPname(clCreateContextFromType),
        &IPname(clRetainContext),
         (...)
};
```

3.7.2 Atributo de plataforma CL_PLATFORM_IP

Após a integração do rOpenCL com o *ICD-Loader* percebeu-se que não seria possível ao rOpenCL expor também as plataformas locais, além das remotas, porque isso implicaria modificar o código do próprio *ICD-Loader* (opção seguida pelo dOpenCL). Essa modificação foi considerada indesejável porque iria no sentido contrária da transparência que se pretendia: o *ICD-Loader* modificado correria sempre o risco de ser substituído durante uma atualização do sistema; e, sempre que surgisse uma nova versão oficial do *ICD-Loader*, teria de ser modificado para incorporar as alterações específicas do rOpenCL.

Uma vez assumido que o rOpenCL expõe apenas plataformas remotas, torna-se conveniente expor também a sua localização, traduzida pelo endereço IP do sistema remoto respetivo. Essa informação passa ser visível, por exemplo, quando se invoca o utilitário *clinfo* [55], uma das ferramentas mais usadas para consulta de um conjunto de parâmetros relacionado com a instalação do OpenCL. Assim, na figura 3.10 é possível observar o endereço IP de uma plataforma remota (i.e., do seu nó) embebido no nome da plataforma:

```
  Platform Name                                   NVIDIA CUDA [192.168.45.40]
Number of devices                                 2
  Device Name                                     GeForce RTX 2080 Ti
  Device Vendor                                   NVIDIA Corporation
```

Figura 3.10: Saída do comando *clinfo* gerada pelo rOpenCL.

Adicionalmente, o endereço IP de uma plataforma remota ficou também acessível através da função *clGetPlatformInfo*, que foi extendida para suportar o novo atributo CL_PLATFORM_IP no campo *param_name*. Esta simples extensão é a base para que os programadores de aplicações OpenCL, que queiram explorar o rOpenCL, possam implementar mecanismos de balanceamento, distribuindo os pedidos OpenCL por plataformas de diferentes *hosts*. Essa distribuição poderá ser eventualmente afinada, incorporando conhecimento prévio sobre as capacidades relativas dos vários dispositivos remotos.

3.8 Gestão de conexões TCP

O mecanismo de gestão de objectos OpenCL, em conjunção com a integração do rOpenCL com o *ICD-Loader*, tornou possível começar a testar de forma mais séria o rOpenCL, recorrendo a aplicações mais pesadas e *benchmarks* de referência. Esses testes evidenciaram a robustez e fiabilidade da implementação conseguida, mas também expuseram alguma degradação de desempenho provocada pela utilização do TCP em detrimento do UDP.

Uma análise à forma como o TCP estava a ser usado permitiu perceber a origem do problema: por cada invocação de uma função OpenCL, o *driver* do rOpenCL criava um novo *socket* e estabelecia uma conexão com um serviço remoto, sendo que o tempo gasto na conexão era suficientemente relevante no contexto do tempo global de execução de funções OpenCL mais simples (i.e., envolvendo pequenas quantidades de dados), algumas das quais invocadas com muita frequência. A título ilustrativo, veja-se o caso da função *clSetKernelArg*, podendo-se comparar na tabela 3.3 o tempo consumido (médias de 3 amostras) em cada fase da comunicação quando se usa UDP vs TCP, para um teste envolvendo uma pequena quantidade de dados transferidos (menos que 1024 *bytes*)[3].

Observando a tabela 3.3 percebe-se a existência de uma grande diferença na fase *Binding/Connect* entre o uso de UDP e TCP, sendo os tempos das restantes fases da comunicação muito próximos. Esta diferença motivou uma nova abordagem à gestão de conexões TCP, ilustrada na figura 3.11 e explicada na secções seguintes.

[3]Esta quantidade foi suficiente para evitar perdas com UDP.

Tabela 3.3: *Profiling* da função *clSetKernelArg* com UDP e TCP (tempos em ms).

Fase	***Tempo com UDP***	***Tempo com TCP***
Serialização do Pedido	9	10.33
Criação de Socket	11	11
Binding (UDP)/Connect (TCP)	12.33	370
Envio e Receção	404.33	365
Deserialização da Resposta	7.33	6.33
Total por Transacção	443.99	762.33

3.8.1 Gestão de conexões TCP no *driver*

Como referido anteriormente, a primeira abordagem à gestão das conexões entre o *driver* do rOpenCL e os serviços remotos revelou-se ineficiente, por assentar na criação de um *socket* e sua conexão a um serviço remoto, por cada transação. Ora, sendo certo que durante a execução de uma aplicação OpenCL que tire partido de co-processadores remotos por intermédio do rOpenCL, haverá potencialmente muitas invocações a primitivas OpenCL, o ideal seria que a aplicação estabelecesse uma só conexão TCP com cada serviço remoto e que esta fosse reutilizada durante a vida da aplicação. Nesta perspetiva, importa no entanto acautelar a possibilidade de a aplicação ser baseada em múltiplos *threads*, sendo então necessário garantir que cada *thread* utiliza uma conexão própria com cada serviço remoto, não partilhada com outros *threads* da mesma aplicação OpenCL.

Refira-se, porém, que o conceito de *thread* a explorar neste contexto não corresponde às POSIX *Threads*, que foram usadas na implementação do serviço remoto, mas sim às *threads* do *kernel* do *Linux*, que dispõem de um identificador TID [49] global único. Esse TID permite identificar, de forma única, qualquer invocador de funções OpenCL que passe pelo *driver* do rOpenCL, seja esse invocador um processo pesado *single-threaded*, ou um de vários *threads* de um processo *multi-threaded*. Esta característica permite implementar um novo mecanismo de gestão de conexões, conforme descrito a seguir.

Assim, quando uma chamada OpenCL entra no *driver* rOpenCL, haverá primeiro que determinar o serviço remoto a contactar (rever secção 3.7.1). Depois, o TID do *thread* que entrou no *driver*, e o endereço IP do serviço alvo, são usados como chave de pesquisa

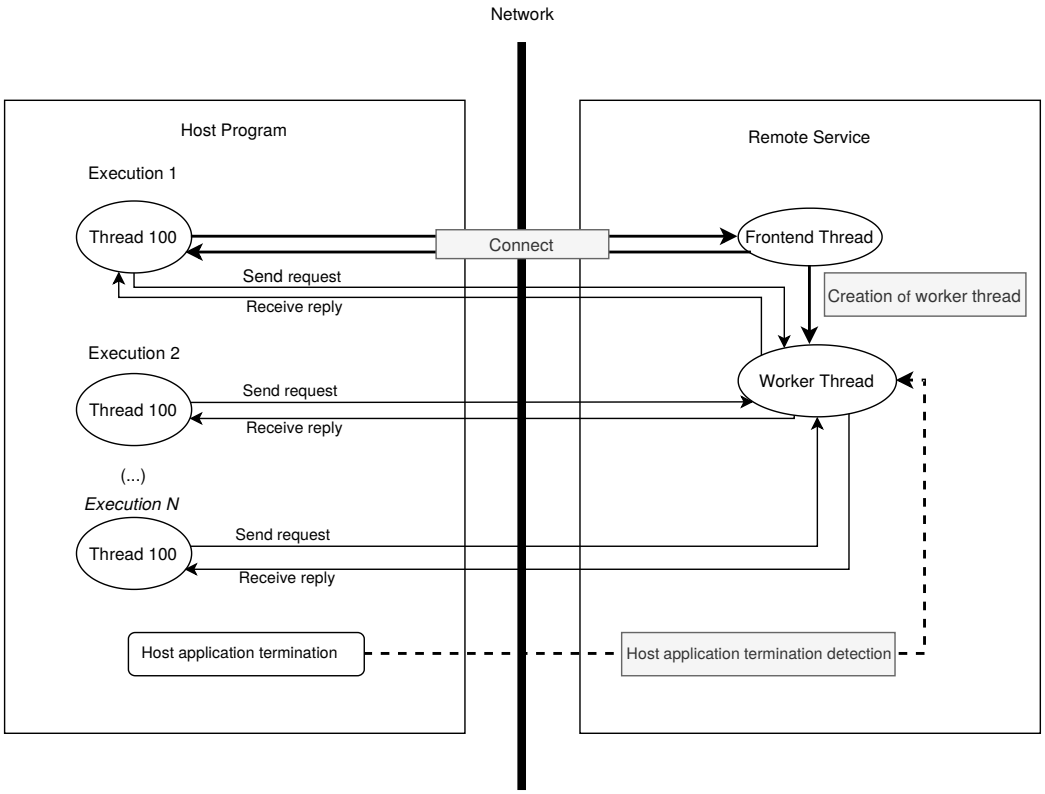


Figura 3.11: Gestão de conexões TCP duradouras no rOpenCL.

numa estrutura de dados que mapeia esse par num *socket* conectado ao serviço remoto; essa estrutura de dados é novamente baseada numa *Red-Black Tree*, doravante designada por árvore de conexões; se o par <TID,IP> em causa não for encontrado na árvore de conexões, isso indica a necessidade de criar um novo *socket* TCP, conectá-lo ao serviço remoto, e registar na árvore o novo mapeamento do par <TID,IP> no *socket* recém-criado; das próximas vezes que o mesmo *thread* necessitar de contactar o mesmo serviço remoto, poderá reutilizar a conexão já estabelecida previamente por intermédio daquele *socket*.

Por fim, é importante salientar que, apesar de não existir especificamente um fecho dos descritores destes *sockets* dentro do código do *driver* do rOpenCL, os mesmos são fechados automaticamente quando a aplicação OpenCL em questão termina.

3.8.2 Gestão de conexões TCP nos serviços

Nos serviços remotos, o mecanismo de gestão de pedidos UDP (descrito na secção 3.4) foi complementado com um mecanismo de gestão de conexões TCP.

Assim, aos dois *threads* usados para receber e processar pedidos recebidos por UDP, foram acrescentados: i) um *frontend thread*, responsável por aceitar pedidos de conexão TCP, e ii) um *worker thread*, criado pelo *frontend thread* por cada conexão recebida, que irá ser responsável por suportar todas as transações realizadas através dessa conexão, as quais têm origem sempre no mesmo *thread* do lado do *driver*; essas transacções implicam a recolha de pedidos de execução de primitivas OpenCL oriundos do *driver*, sua submissão a dispositivos OpenCL do nó do serviço, e retorno dos resultados ao *driver*.

Cada *worker thread* ficará emparelhado com um *thread* do lado *host* da aplicação OpenCL, enquanto esta aplicação não terminar. Quando essa terminação acontecer, os *worker threads* nos vários serviços, emparelhados com as *threads* da aplicação, irão desbloquear da primitiva *recvfrom*, por via da recepção de um sinal; essa primitiva retorna o erro *EINTR* e os *worker threads* concluem que também devem terminar.

3.8.3 Efeito da gestão de conexões TCP duradouras

Após a implementação e validação das alterações descritas nas duas secções anteriores, testou-se novamente o tempo de execução da função *SetKernelArg* para perceber o efeito que as alterações surtiram nos tempos das várias fases de execução da função. A tabela 3.4 permite comparar os resultados obtidos anteriormente (A) com uma conexão TCP por cada transação (rever tabela 3.3), com os obtidos com conexões TCP duradouras (B).

Analisando a tabela, pode verificar-se que a grande diferença de tempos entre os dois cenários, A e B, está na fase *Connect*, sendo o tempo de Criação de Socket no cenário A (11ms) comparável com o tempo de *Pesquisa de Socket* na árvore de conexões no cenário B. Em termos globais, o *speedup* conseguido pela reutilização de conexões TCP parece modesto (apenas 1,53), mas representa, afinal um ganho de mais de 50% de desempenho, que ganhará importância em situações com muitas primitivas OpenCL

Tabela 3.4: *Profiling* de *clSetKernelArg* com TCP sem (A) e com (B) conexões duradouras.

Fase	***Tempo A (ms)***	***Tempo B (ms)***
Serialização do Pedido	10.33	9.33
Criação de Socket	11	-
Connect	370	-
Pesquisa de Socket	-	8.67
Envio e Receção	365	473
Deserialização da Resposta	6.33	6.33
Total por Transação	762.66	497.33

invocadas. Adicionalmente, e em rigor, é preciso levar em conta que mesmo no cenário B é necessário criar um *socket* e inseri-lo na árvore de conexões, mas esses tempos são de pequena dimensão e rapidamente amortizados a partir da segunda transação.

3.9 *Thread Safety*

Um dos pontos importantes no que toca à execução de funções OpenCL em contexto concorrente/distribuído, é compreender até que ponto essas funções podem ser apresentadas em simultâneo, por vários *threads*, aos mesmos dispositivos remotos, sem que os *threads* tenham de acautelar mecanismos *user-level* de controle de concorrência, nomeadamente de exclusão mútua. Ora, de acordo com a especificação do OpenCL, uma implementação do standard deve garantir que todas as funções são *thread safe*, com uma única excepção, representada pela função *clSetKernelArg*, para a qual os programadores devem garantir serialização quando invocada concorrentemente para um mesmo parâmetro *cl_kernel*.

Desta forma, a implementação atual do rOpenCL com recurso ao protocolo TCP assume que, em todos os serviços remotos, a invocação concorrente, por parte das *worker threads*, de funções OpenCL, é inerentemente *thread safe*, acautelando apenas a situação particular da função *clSetKernelArg*. Para esta função, a implementação atual, por simplicidade, tira partido de uma *variável mutex* (POSIX), global ao serviço, para serializar todas as invocações à função, independentemente do valor do parâmetro *cl_kernel*. Uma

alternativa, com menos contenção, prevista para trabalho futuro, é o registo dos objetos *cl_kernel* numa *Red-Black Tree*, cada um acompanhado de uma *variável mutex*.

Quando o protocolo UDP é usado, o problema da função *clSetKernelArg* não se coloca atualmente, pois as chamadas OpenCL nos serviços são inerentemente serializadas, dado existir apenas um *thread* responsável pela sua invocação (rever secção 3.4). De futuro, caso se venha a optar por um modelo diferente (e.g., ter uma *pool* de *threads* para execução das chamadas OpenCL), o caso particular da função *clSetKernelArg* terá de ser também acautelado, seguindo, por exemplo, a mesma estratégia prevista com o uso de TCP.

3.10 Estágio Atual do rOpenCL

Após todas as alterações implementadas, especificadas nas seções anteriores, foi possível ao rOpenCL atingir um funcionamento estável e eficiente. A arquitetura final da implementação está representada na figura 3.12, que apesar de ser muito semelhante à figura 2.8, acabou por se distanciar em vários aspetos da sua abordagem precursora (clOpenCL).

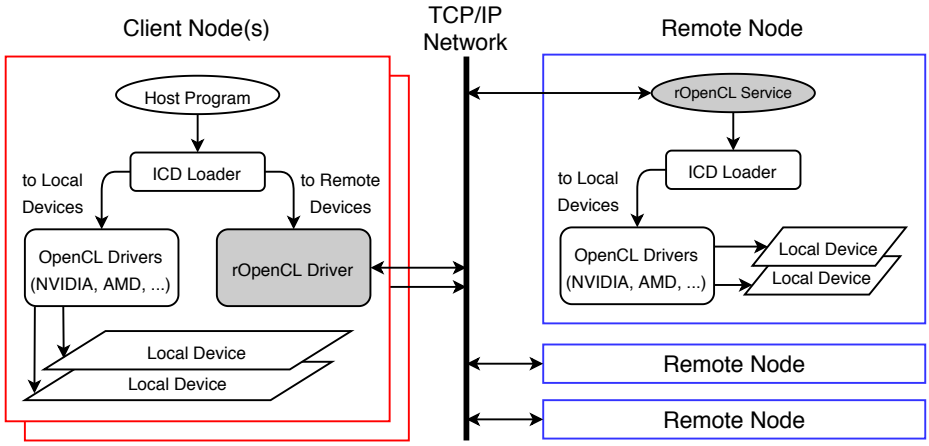


Figura 3.12: Arquitetura final do rOpenCL.

No seu estágio atual, o rOpenCL apenas suporta a versão 1.2 [56] do OpenCL, "limitação" herdada das implementações da maioria dos *vendors* OpenCL, que se ficam por essa versão. Além disso, nem todas as funções OpenCL podem ser executadas remotamente pelo rOpenCL. A tabela 3.5 clarifica a cobertura atual do OpenCL 1.2 pelo rOpenCL.

Tabela 3.5: Cobertura da especificação 1.2 do OpenCL pelo rOpenCL.

Grupo de Funções	***Suportadas***	***Não Suportadas***
The OpenCL Platform Layer	13	0
The OpenCL Runtime	4	0
Buffer Objects	13	3
Program Objects	11	0
Kernel and Event Objects	22	1
Image Objects	7	3
Sampler Objects	4	0
OpenGL Sharing	0	9
Direct3D 10 Sharing	0	6
DX9 Media Surface Sharing	0	4
Direct3D 11 Sharing	0	6

A escolha das funções a suportar pelo rOpenCL foi condicionada pelo contexto de desenvolvimento (orientado a computação de alto desempenho, e não a aceleração gráfica) e ambiente de execução previsto. Assim, os grupos de funções *Direct3D 10 Sharing*, *DX9 Media Surface Sharing* e *Direct3D 11 Sharing*, foram automaticamente excluídos, dado o rOpenCL ser direccionado a ambientes *Unix/Linux* e os grupos anteriores serem responsáveis pela integração do OpenCL com o *DirectX*, que é uma tecnologia para Windows.

Adicionalmente, o grupo de funções *OpenGL Sharing* não foi considerado por necessitar de dados gerados pela API do Open Graphics Library (OpenGL), isto é, este grupo realiza a integração do OpenCL com o OpenGL, sendo que uma parte considerável das funções deste grupo contém parâmetros provenientes de funções do OpenGL; assim, a inclusão destas funções tornar-se-ia incompatível com os objetivos desta dissertação.

Como no rOpenCL ainda não foi implementada gestão de réplicas, não é suportada a partilha de memória entre dispositivos de nós diferentes via rede. Consequentemente, algumas das funções de determinados grupos da tabela 3.5, como os grupos *Image Objects* e *Buffer Objects*, não são suportadas ou são-o parcialmente. É o caso da função *clEnqueueCopyBuffer*, que permite a cópia de um *buffer* de memória mas, para já, apenas dentro da mesma máquina. É também algo que acontece com a função *clEnqueueImageBuffer*, que ainda só permite a cópia de um *ImageBuffer* dentro da mesma máquina.

Quanto ao grupo *Kernel and Event Objects*, a função *clEnqueueNativeKernel* é a única que não é suportada. Tal acontece porque, para a correta execução desta primitiva, é necessário disponibilizar nos serviços remotos uma função definida no *host* da aplicação.

A resolução destas limitações levaria à inclusão de outras tecnologias e/ou soluções de *software* no rOpenCL, que não foram consideradas para não tornar o presente trabalho demasiado extenso. Por razões semelhantes, a implementação do mecanismo *pthread notify*, presente num sub-conjunto de funções, foi deixado para trabalho futuro, uma vez que seria necessário adicionar novas camadas de *software* para garantir o seu funcionamento.

Por último, discute-se a gestão de contextos OpenCL num ambiente distribuído. Originalmente, o OpenCL possibilita aos programadores a criação de contextos que envolvam dispositivos de plataformas diferentes. Além disso, através do mecanismos de eventos, é também possível sincronizar a execução de comandos associados a determinados aceleradores, permitindo uma melhor organização no envio de comandos para as diferentes *commands queues* existentes. Seria assim expectável que o rOpenCL permitisse a criação de contextos com dispositivos das múltiplas plataformas remotas, assim como a sincronização de comandos entre esses dispositivos. No entanto, resultante da ausência de suporte total à partilha de memória, assumida anteriormente, não é possível a criação de contextos tão generalistas, nem a sincronização de execução de comandos, através dos mecanismos de eventos, entre máquinas de nós diferentes. Fica apenas salvaguardada a normal criação de contextos e o uso do mecanismo de eventos entre dispositivos situados no mesmo nó.

Capítulo 4

Análise de Desempenho

Realizada a apresentação da arquitetura do rOpenCL e dos aspetos mais relevantes da sua implementação, apresenta-se neste capítulo o resultado da sua avaliação e validação, com base em *benchmarks* OpenCL de referência e contraposição com outras abordagens do género. O capítulo termina com resultados de alguns *benchmarks* assentes em CPU local que evidenciam os méritos da execução em aceleradores remotos através do rOpenCL.

4.1 Prólogo

Apesar das limitações referidas na secção 3.10, a implementação realizada permitiu levar o rOpenCL a um patamar onde suporta a execução de *benchmarks* OpenCL bem conhecidas, possibilitando a validação do rOpenCL e o estudo de diversos parâmetros de desempenho.

A introdução de operação distribuída no modelo do OpenCL comporta custos de desempenho. A rede que interliga todas as máquinas participantes, por mais rápida que seja, acrescenta atrasos na execução de qualquer aplicação OpenCL através do rOpenCL. Contudo, considerar a porção de tempo gasta no envio/receção dos dados através da rede não é o único fator a ter em conta numa avaliação do desempenho do rOpenCL. Para compreender melhor o desempenho da implementação realizada é igualmente importante avaliar: a escalabilidade dos serviços rOpenCL remotos; comparar o desempenho do rOpenCL com algumas das abordagens apresentadas no capítulo 2; perceber o nível de desempenho

oferecido pelo uso de dispositivos remotos (fundamentalmente GPUs) expostos pelo rOpenCL, em comparação com o desempenho obtido quando se utiliza uma CPU *multicore* local por não haver outros acelerados locais disponíveis.

De forma sucinta, a avaliação realizada procurou responder às seguintes questões:

- Qual o comportamento dos serviços remotos em contexto de elevada carga.
- Que funções do OpenCL é que o rOpenCL suporta e executa de forma correta.
- Que percentagem de tempo é gasto com o uso do rOpenCL em execuções que apenas utilizem um único dispositivo.
- Quais as vantagens do rOpenCL sobre o OpenCL em ambientes onde podem ser usados vários dispositivos.
- Que desempenho oferece o rOpenCL face a abordagens apresentadas no capítulo 2.
- De que forma compensa utilizar GPUs remotas em detrimento do uso da CPU local.

Através de um conjunto de *benchmarks* seleccionado de forma criteriosa, foi possível avaliar o rOpenCL em diversos cenários. Tal possibilitou obter respostas a estas questões e retirar outras conclusões igualmente importantes, como explicado nas próximas secções.

4.2 *Benchmarks* OpenCL

O universo de *benchmarks* OpenCL é relativamente amplo, levando à necessidade de definir um conjunto de parâmetros de caracterização dos *benchmarks*, de forma a auxiliar a selecção dos mais adequados para avaliar o rOpenCL. Os parâmetros considerados foram:

- *Khronos* (Sim/Não): o *benchmark* é ou não recomendado [57] pela entidade responsável pela especificação do OpenCL;
- Tipo (Cálculo, Memória, Gráficos, Híbrido): qual o foco de avaliação do *benchmark*, sendo considerado Híbrido quando avalia pelos menos duas das restantes categorias;

- Número de Dispositivos (Mono/Multi): o *benchmark* tira partido apenas de um dispositivo (Mono) ou é capaz de tirar partido de vários (Multi).

- Linux (Sim/Não): o *benchmark* é ou não executável em ambiente *Linux*.

- *Open Source* (Sim/Não): o código fonte está ou não disponível; essa disponibilidade é muito importante, para permitir afinações/modificações ao código do *benchmark* a fim de automatizar a sua execução no âmbito da bancada de testes montada, e corrigir *bugs* identificados no processo.

- Último *Commit*: data da última atualização do *benchmark* realizada pelos autores; a atualidade/manutenção do código do *benchmark* é importante, reforçando a confiança nos resultados produzidos, nomeadamente pela correcção de *bugs*;

Para compreender se determinado *benchmark* deve ou não ser considerado, para além da sua análise à luz dos parâmetros acima definidos, também foi necessário levar em conta: a forma como os resultados são apresentados; o mecanismo de compilação; as dependências de outros softwares, que é necessário pré-instalar.

Numa primeira fase, identificaram-se os vários *benchmarks* enumerados na tabela 4.1, tendo todos eles o código fonte disponível e sendo compiláveis em Linux. Entretanto, uma análise mais ponderada levou à selecção de um conjunto mais restrito desses *benchmarks*.

Assim, o Hetero-Mark, OpenDwarfs e Luxmark excluíram-se por não ter sido possível a sua compilação no ambiente de testes montado (ver secção 4.3), mesmo depois de se seguirem as instruções de compilação disponibilizadas pelos autores.

Por seu turno o Cf40cl não foi seleccionado por ser essencialmente uma *framework* em C para desenvolvimento de aplicações OpenCL, possuindo uma componente de *benchmarking* cujas funções OpenCL utilizadas já eram avaliadas noutros *benchmarks*.

Do grupo dos *benchmarks* excluídos fazem também parte o Mandelgpu, Smallpt-gpu, Mandelbulbgpu e o Parboil, dado possuírem uma data de Último *Commit* muito antiga, o que explica algumas incompatibilidades com *kernel* dos sistemas Linux usados nos testes

Tabela 4.1: Lista de *benchmarks* OpenCL identificados (a **negrito**, os selecionados).

Nome	**Khronos**	**Tipo**	**Número de Dispositivos**	**Último Commit**
FinanceBench [58]	Sim	Cálculo	Mono	2016-09-25
BabelStream [59]	Sim	Memória	Mono	2019-08-08
Hetero-Mark [60]	Sim	Híbrido	Mono	2019-01-17
Mixbench [61]	Sim	Híbrido	Mono	2019-05-06
OpenDwarfs [62]	Sim	Híbrido	Mono	2019-06-04
PolyBench [63]	Sim	Híbrido	Mono	2015-12-14
cl-mem [64]	Sim	Memória	Mono	2016-12-21
Luxmark [65]	Sim	Gráfico	Multi	2019-10-07
Cf40cl [66]	Sim	Híbrido	Mono	2019-08-28
Juliagpu [67]	Não	Gráfico	Mono	2018-04-07
Mandelgpu [68]	Não	Gráfico	Mono	2014-04-11
Smallpt-gpu [69]	Não	Gráfico	Mono	2010-01-08
Mandelbulbgpu [70]	Não	Gráfico	Mono	2013-11-06
clpeak [71]	Não	Híbrido	Mono	2019-08-24
Rodinia [72]	Não	Híbrido	Mono	2017-10-23
Parboil [73]	Não	Híbrido	Mono	2014-02-21
Hashcat [74]	Não	Cálculo	Multi	2016-06-23
viennaCL [75]	Sim	Cálculo	Mono	2016-01-20

assim como com a versão do OpenCL usada. Adicionalmente, percebeu-se ainda que as funções OpenCL avaliadas nestes *benchmarks* estavam presentes em outros mais recentes.

Os Mixbench e o ViennaCL também acabaram por não se utilizar, dado que nos *benchmarks* seleccionados (ver a seguir) o foco (Tipo) da avaliação (híbrido e cálculo, respetivamente), bem como todas as funções OpenCL avaliadas, estão contemplados.

Por fim, apesar de não ter sido considerado mais nenhum *benchmark* de tipo gráfico, o *juliagpu* acabou por se excluído porque não utiliza as funções OpenCL associadas ao processamento de imagens (*clCreateImage, clEnqueueWriteImage*), não acrescentado assim funções OpenCL novas à lista de funções testadas com recurso a uma *benchmark*.

Os *benchmarks* seleccionados para a avaliação do rOpenCL resumem-se então àqueles cujo nome está em **negrito**, na tabela 4.1. Em termos gerais, verifica-se que não foram considerados para avaliação quaisquer *benchmarks* de tipo Gráfico, mas os restantes tipos (Cálculo, Memória e Híbrido) estão representados de forma relativamente equilibrada. Para uma mais fácil identificação, a tabela 4.2 reúne apenas os *benchmarks* selecionados.

Tabela 4.2: Lista de *benchmarks* selecionados.

Nome	Khronos	Tipo	Número de Dispositivos	Último Commit
BabelStream [59]	Sim	Memória	Mono	2019-08-08
cl-mem [64]	Sim	Memória	Mono	2016-12-21
clpeak [71]	Não	Híbrido	Mono	2019-08-24
FinanceBench [58]	Sim	Cálculo	Mono	2016-09-25
PolyBench [63]	Sim	Híbrido	Mono	2015-12-14
Rodinia [72]	Não	Híbrido	Mono	2017-10-23
Hashcat [74]	Não	Cálculo	Multi	2016-06-23

Nos parágrafos seguintes fornece-se um mínimo de informação sobre cada um, podendo-se obter mais detalhes consultando as respetivas referências bibliográficas.

BabelStream Trata-se de um *benchmark* utilizado para medir as taxas de transferência internas nas memórias (*memory bandwitdth*) dos dispositivos OpenCL durante a execução de cinco *kernels* (*copy*, *mul*, *add*, *triad* e *dot*). Não inclui, portanto, tempos de transferência através do barramento *PCIe* entre o *host* e os dispositivos a ele acoplados.

cl-mem Tal como o BabelStream, realiza operações sobre a memória dos dispositivos. Conta com 3 sub-testes (*write, read e copy*) destinados a avaliar a velocidade de escritas, leituras e cópias em sequência (de 128 GB de dados); estas operações são todas efetuadas em paralelo, por grupos de *threads*, no dispositivo testado; não necessita de transferências de dados para o dispositivo, dado que os dados necessários são gerados *on-the-fly*.

clpeak É um *benchmark* híbrido, que mede capacidades de pico dos dispositivos em processamento e largura de banda da memória. Inclui 6 sub-testes: *Global memory bandwidth* (GBPS), *Single-precision compute* (GFLOPS), *Double-precision compute* (GFLOPS), *Integer Compute* (GIOPS), *Transfer bandwith* (GBPS) e *Kernel launch latency* (μs). Os valores obtidos, gerados por operações vetoriais, não representam um caso de uso real.

FinanceBench Direccionado essencialmente a cálculo, inclui 4 sub-testes (*Black-Scholes, Monte-Carlo, Bonds, Repo*), todos relacionados com aplicações financeiras reais, e que podem ser executados recorrendo a diversas tecnologias (CUDA, MPI, OpenCL). Embora

apenas 2 sub-testes usem OpenCL (*Black-Scholes* [76] e *Monte-Carlo* [77]), estes ganham especial importância para a avaliação do rOpenCL, por demonstrarem a sua capacidade de execução de aplicações com utilidade prática efetiva, e não só de *benchmarks* sintéticos.

PolyBench Consiste num conjunto de 30 sub-testes de cálculo numérico, cobrindo várias domínios de aplicação (álgebra linear, estatística, programação dinâmica, simulações físicas, processamento de imagens, etc.), sendo que apenas 21 tiram partido de OpenCL:

- correlation
- covariance
- 2mm
- 3mm
- atax
- bicg
- doitgen
- gemm
- gemver
- gesummv
- mvt
- syr2k
- syrk
- gramschmidt
- lu
- adi
- 2DConvolution
- 3DConvolution
- fdtd2d
- jacobi1D
- jacobi2D

Rodinia É um conjunto de *benchmarks* direcionados a plataformas de computação heterogéneas e que tira partido de tecnologias como OpenMP, OpenCL e CUDA. Dos 23 programas fornecidos, só 20 podem ser executados com OpenCL, cobrindo áreas ligadas à imagiologia médica, bioininformática, dinâmica de fluídos, álgebra linear, etc:

- backprop
- bfs
- cfd
- dwt2d
- gaussian
- heartwall
- hotspot
- hotspot3D
- hybridsort
- kmeans
- lavaMD
- leukocyte
- lud
- myocyte
- nn
- nw
- particlefilter
- pathfinder
- srad
- streamcluster

Hashcat Este *benchmark* é o único concebido à partida para tirar partido de vários dispositivos em simultâneo. Na prática, não é propriamente um *benchmark*, mas sim uma aplicação de quebra de chaves criptográficas. A sua inclusão na lista de *benchmark* é prova de que o rOpenCL pode ser usado com ferramentas que lidam com casos reais. Na secção 4.6.7 são fornecidos detalhes adicionais sobre a forma como o Hashcat foi usado.

4.3 Ambiente de Teste

Todos os testes, envolvendo os *benchmarks* seleccionados, foram realizados em máquinas virtuais de um *cluster* oVirt [78], com alguns servidores munidos de GPUs, e com capacidade de disponibilizar máquina virtuais com GPUs através de mecanismos de *passthrough*. A tabela 4.3 sumaria as características destes servidores.

CPU	2 x AMD EPYC 7351 16-Core 2.4/2.9GHz
RAM	256 GB ECC DDR4 2666 MHz
Rede	10Gbps Ethernet
GPUs	2 x NVIDIA RTX 2080 Ti

Tabela 4.3: Especificações de hardware dos servidores de virtualização.

Desta forma, há que levar em conta que os resultados dos *benchmarks*, apresentados mais adiante, são certamente inferiores (em termos de desempenho) aos que se conseguiriam obter em ambiente *bare-metal*, não só devido à sua execução em máquinas virtuais, como também pelo facto de, por limitações do *hypervisor* usado (QEMU+KVM), as GPUs passadas às máquina virtuais estarem limitadas à velocidade 1x no barramento PCIe 3.0, em vez de poderem operar à velocidadde 16x. Independentemente de tudo isto, os resultados obtidos são, como se verá, demonstrativos do potencial da solução desenvolvida.

Quanto às várias máquinas virtuais usadas, e salvo excepções assinaladas quando oportuno, as suas características estão elencadas na tabela 4.4.

Foi sempre garantido que as máquinas virtuais estavam localizadas em diferentes servidores, para assegurar o uso da rede física na comunicação (em vez de memória partilhada, como seria com máquinas virtuais alojadas no mesmo servidor). O MTU usado foi sempre

vCPU	1 x AMD EPYC 7351 8-Core 2.4/2.9GHz
vRAM	16 GB ECC DDR4 2666 MHz
vNIC	10Gbps Ethernet
vGPU	1 ou 2 x NVIDIA RTX 2080 Ti
Sistema Operativo	Ubuntu 18.04.3 LTS
POCL	versão 1.3
Intel OpenCL SDK	versão 18.1.0.0920
NVIDIA Driver	versão 430.50

Tabela 4.4: Especificações das máquinas virtuais.

de 1500 bytes, com excepção dos testes de comparação do rOpenCL com abordagens afins, em que o MTU foi de 9000 bytes, dado ser este o MTU recomendado para o clOpenCL.

Adicionalmente, houve o cuidado de realizar os testes em períodos de tipicamente menor carga nos servidores do *cluster* e na sua rede, designadamente períodos noturnos e dias não-úteis, uma vez que o *switch* de 10Gbps que interliga os vários servidores é também parte integrante da rede de *core* do *datacenter* do IPB.

4.4 Cenários de Teste

Após a escolha dos *benchmarks*, e a caracterização do ambiente computacional para a sua execução, levanta-se a necessidade de definir os diferentes cenários de teste, tendo em conta as contingências desse ambiente, nomeadamente: i) a existência de apenas 2 servidores com GPUs; ii) cada um desses servidores ter apenas 2 GPUs; iii) o número máximo de GPUs de uma máquina virtual ser 2; iv) o número máximo de GPUs envolvidos num teste ser 4. Estas limitações, juntamente com o número máximo de dispositivos suportados por cada *benchmark*, levaram à definição de dois tipos de testes, descritos a seguir.

4.4.1 Testes Mono-Cliente

Nestes testes é executada uma só instância em simultâneo de um *benchmark*, daí designarem-se "testes mono-cliente". Cada teste desde tipo é realizado com GPUs locais e repetido com GPUs remotas. O objetivo é perceber, para cada *benchmark*, qual a sobrecarga do

rOpenCL na sua execução (indo esse estudo um puoco mais além no caso do Hashcat).

Estes testes usam duas máquinas virtuais: uma cliente, onde o *benchmark* arranca; uma servidora, onde executa o serviço rOpenCL. Em ambas as máquinas virtuais existem 2 GPUs e todos os benchmarks, excepto o Hashcaht, usam uma só GPU (ou local, ou remota). Para o Hashcat, o nº de GPUs usados varia de 1 a 4, podendo incluir simultaneamente GPUs remotas e locais, pois além da avaliação da sobrecarga do rOpenCL com um só GPU remoto, também se fez um estudo de escalabilidade com várias GPUs.

Para automatizar os vários testes mono-cliente foram construídas scripts (em Python), acauteladndo pausas apropriadas, entre execuções sucessivas, a fim de permitir o arrefecimento das GPUs, antes de cada teste, como forma de favorecer o desempenho.

Nos primeiros ensaios iniciais deste tipo de testes, apesar de todos os esforços realizados para garantir um ambiente o mais isolado possível, registaram-se algumas inconsistências nos tempos obtidos: para *benchmarks* de muito curta duração, a sua execução remota através do rOpenCL era mais rápida que a execução com um plataforma OpenCL local. Como sugerido na literatura [79], uma parte da explicação para este aparente paradoxo reside em efeitos de *caching*: entre execuções sucessivas do mesmo *benchmark*, os serviços remotos não são reiniciados; dessa forma, os *drivers* OpenCL das plataformas usadas são carregados uma só vez, no arranque dos serviços, persistindo carregados; pelo contrário, do lado do *host*, esse *driver* tem de ser carregado de cada vez que o *benchmark* é executado.

4.4.2 Testes Multi-Cliente

O objetivo destes testes é o estudo da escalabilidade dos serviços rOpenCL, através da execução de várias instâncias do mesmo *benchmark*. Estas instâncias tiram partido dos GPUs remotos disponíveis no *cluster*, em diferentes cenários. Estes cenários usam um máximo de 8 máquinas virtuais "clientes" onde os *benchmarks* arrancam, e até 2 máquinas virtuais "servidoras" com um serviço rOpenCL e até 2 GPUs cada. Os cenários multi-cliente explorados (cenários 1G, 2G e 4G, usando 1, 2 e 4 GPUs) são apresentados adiante.

Entretanto, é importante referir que a execução de testes multi-cliente implicou a construção de *scripts* (em Python) capazes de, em qualquer cenário, lançar simultâneamente os vários clientes, detectar a sua terminação, e contabilizar corretamente o tempo de execução do conjunto de clientes. Isso implicou também, nalguns casos, pequenas modificações ao código dos *benchmarks*, o que só foi possível devido à disponibilidade do código fonte. Adicionalmente, também nessas scripts foi necessário levar em conta o impacto da temperatura das GPUs, instrumentando pausas adequadas entre as execuções dos testes.

Cenário 1G Neste cenário, representado na figura 4.1, são usadas até 9 máquinas virtuais; uma delas (servidora) aloja um serviço rOpenCL associado a uma só GPU (1G); o número das outras (clientes) faz-se variar de 1 a 8, originando 8 sub-cenários diferentes; em cada máquina virtual cliente usada executa uma instância de um *benchmark*; todas as instância irão recorrer ao mesmo GPU, exposto pelo único serviço rOpenCL disponível.

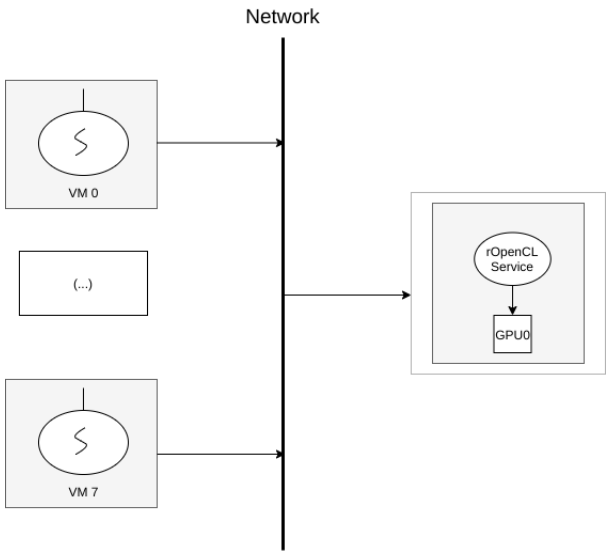


Figura 4.1: Cenário 1G para análise da escalabilidade dos serviços rOpenCL.

Cenário 2G No cenário 2G, representado na figura 4.2, continua a haver um só serviço rOpenCL, mas agora expondo 2 GPUs (2G). Adicionalmente, as 8 máquinas virtuais clientes são divididas em dois grupos, de 4 máquinas cada. O alvo de cada grupo é uma

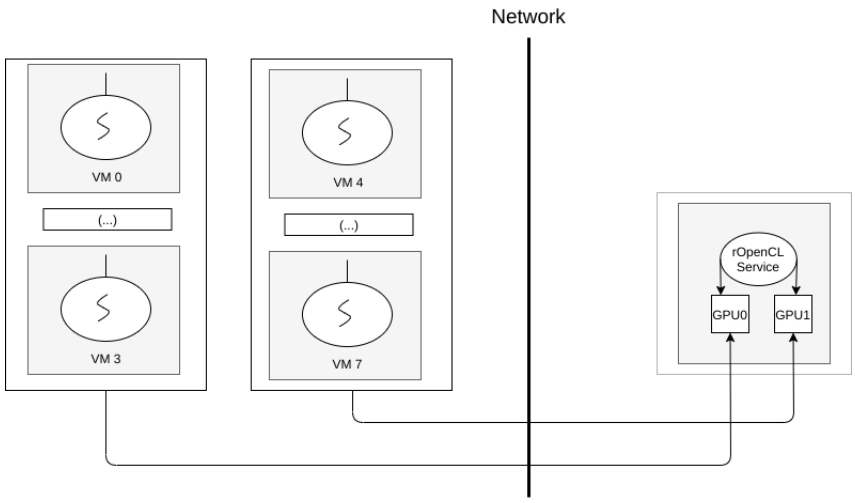


Figura 4.2: Cenário 2G para análise de escalabilidade dos serviços.

GPU diferente, de forma que a carga gerada em cada GPU seja similar. Assim, cada grupo contribui com igual número de instâncias de *benchmarks*, originando 4 sub-cenários de testes, com um total de 2 (1+1), 4 (2+2), 6 (3+3) ou 8 (4+4) *benchmarks* em execução.

Cenário 4G Este cenário, ilustrado na figura 4.3, usa todas as 4 GPUs disponíveis e um total de 10 máquinas virtuais (8 clientes e 2 servidoras). As máquinas clientes são dividas em 4 grupos, e o alvo de cada grupo é uma GPU diferente (4G). Assim, são viáveis apenas 2 sub-testes: i) um sub-teste, com uma máquina virtual cliente por grupo, com um total de 4 (1+1+1+1) *benchmarks* em execução; ii) outro sub-teste, com duas máquinas virtuais clientes por grupo, com um total de 8 (2+2+2+2) *benchmarks* em execução.

4.5 Metodologia e Métricas

Para os testes mono-cliente, os resultados apresentados correspondem a médias de 10 amostras (10 repetições do sub-cenário específico de teste em causa). Isto porque tanto na máquina cliente, como na máquina servidora, existem 2 GPUs, tendo-se recolhido 5 amostras usando uma dessas GPUs, e outras 5 amostras usando a outra GPU, de forma a assimilar possíveis variações de resultados com diferentes GPUs.

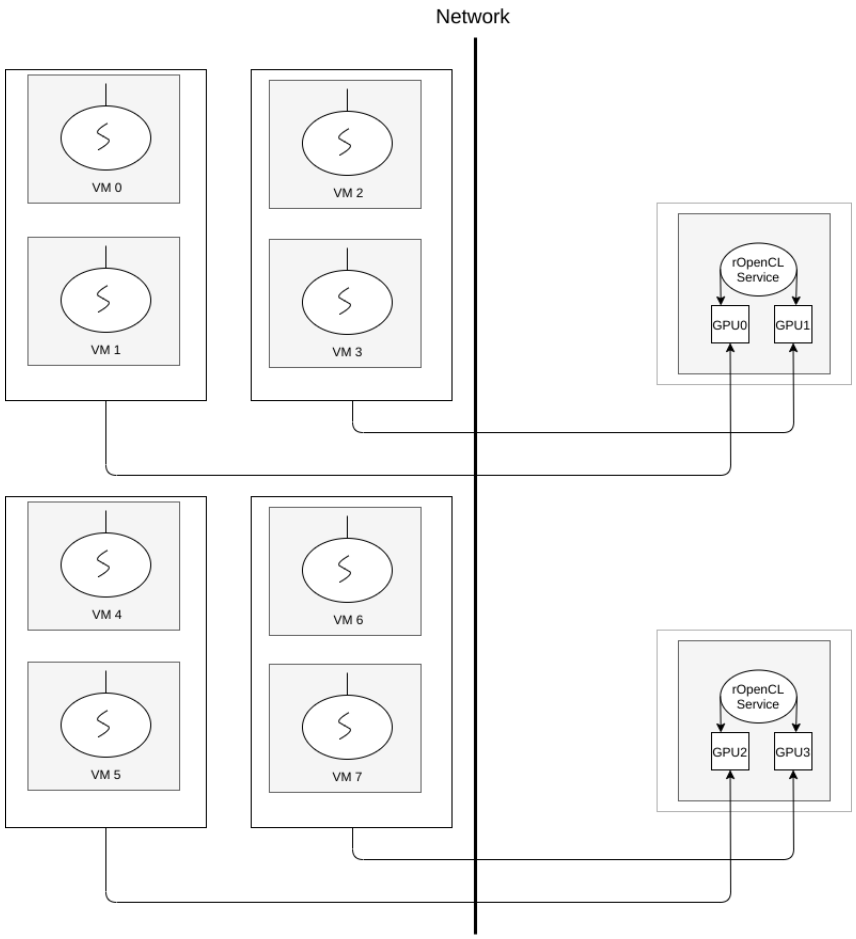


Figura 4.3: Cenário 4G para análise de escalabilidade dos serviços.

A média, apesar de ser uma métrica simples, considerou-se adequada porque, após uma análise prévia dos dados obtidos pelos vários testes percebeu-se que, independentemente do teste em questão, os valores das 10 amostras não apresentavam discrepâncias relevantes.

Para os vários testes mono-cliente são então apresentados tempos médios de execução, bem como a sobrecarga (desaceleração) introduzida pelo rOpenCL. Esta é dada, em termos percentuais, pela expressão $(\overline{T_r}/\overline{T_l} - 1) \times 100$, onde $\overline{T_r}$ é a média dos tempos da execução remota e $\overline{T_l}$ corresponde à média dos tempos da execução local.

Novamente o Hashcat representa a excepção: apenas se apresentam tempos de execução e *speedups*, e os testes não foram repetidos com diferentes GPUs.

Nos testes multi-cliente, que avaliam a escalabilidade, o recurso à média de todos os

tempos registados, em todos os clientes, não permite percepcionar correctamente o efeito da variação do número de *benchmarks* clientes. De facto, haverá sempre clientes mais rápidos, e outros mais lentos, mas quando está em jogo a execução de um conjunto de clientes, o que conta, em termos de percepção, é o momento em que o último cliente do grupo (o cliente mais lento) termina a execução do *benchmark*; esse momento marca o fim da execução do grupo. Desta forma, nos testes multi-cliente, é considerada a média dos piores tempos obtidos de 5 amostras. É mostrada também a evolução da carga das GPUs usadas e das suas temperaturas, obtidas com base no utilitário *nvidia-smi*.

4.6 Resultados dos Testes Mono-Cliente

Nesta secção são apresentados os resultados de todos os testes mono-cliente. Para cada teste é apresentado um gráfico e uma breve discussão em torno do mesmo.

4.6.1 BabelStream

O BabelStream, recorde-se (rever secção 4.2), é um *benchmark* direccionado à medição da largura de banda da memórias dos dispositivos, sendo no entanto os resultados aqui apresentados expressos em tempos de execução (ms). A figura 4.4 fornece uma visão global desses resultados. Com base nessa figura, verifica-se que executar este *benchmark* com recurso ao rOpenCL incorre numa penalização de 336.1% no tempo total de execução.

O gráfico da figura 4.5 oferece uma visão de pormenor dos vários sub-*benchmarks*, sendo que os tempos globais da figura 4.4 resultam da soma dos tempos da figura 4.5.

Analisando o comportamento dos diferentes sub-*benchmarks*, verifica-se que a sobrecarga da execução remota é apreciável, situando-se na maior parte dos casos entre cerca de 250% e 300%, e com um caso em que está um pouco acima dos 600%. A leitura do código dos sub-*benchmarks* revela que cada um deles lida com 33554432 elementos de tipo *double* e que executa um *kernel* 100 vezes. No entanto, no caso do *dot*, há também operações de cópia de dados na memória do dispositivo. Esta cópia é realizada através de invocação de

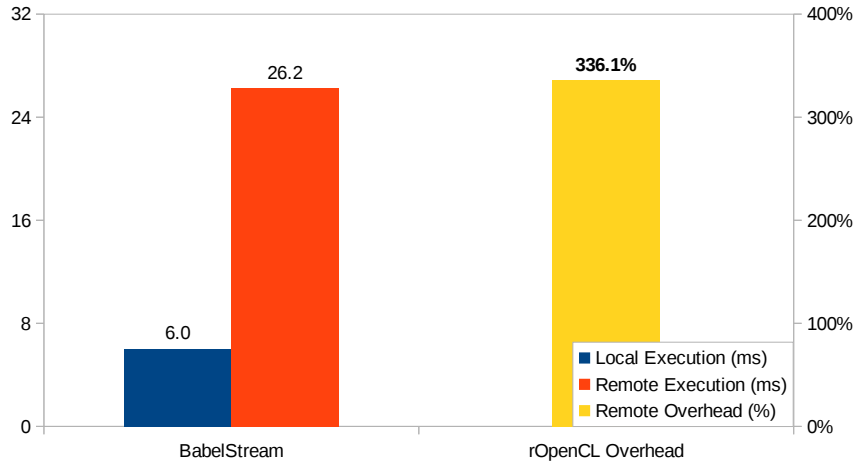


Figura 4.4: Resultados globais do *benchmark* BabelStream.

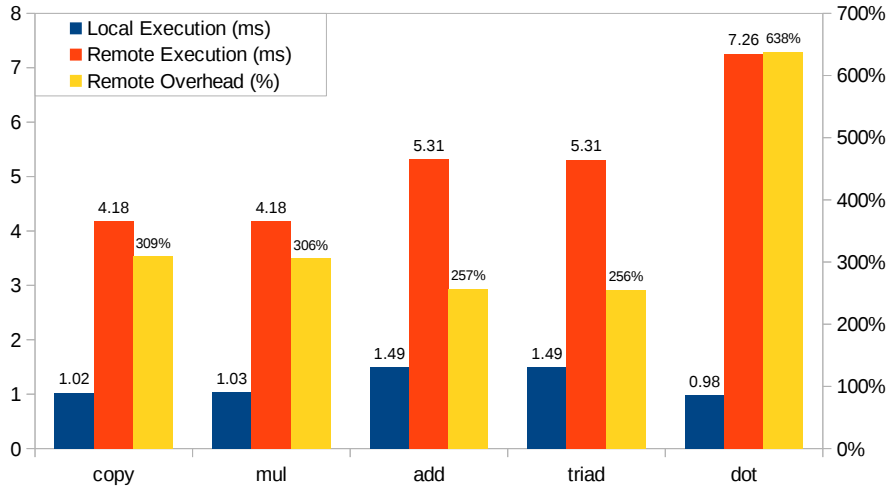


Figura 4.5: Resultados individuais dos sub-*benchmarks* do BabelStream.

funções OpenCL, e não no código do *kernel*, o que explica a elevada sobrecarga (638%) deste sub-*benchmarks* em particular.

4.6.2 cl-mem

O cl-mem é também um *benchmark* focalizado em operações sobre a memória dos dispositivos, neste caso operações mais fundamentais (*write*, *read*, e *copy*). Os resultados globais são apresentados na figura 4.6 e os resultados de cada sub-*benchmark* na figura 4.7.

Em termos globais, a sobrecarga do rOpenCL é negligenciável (0,12%). Isso acontece

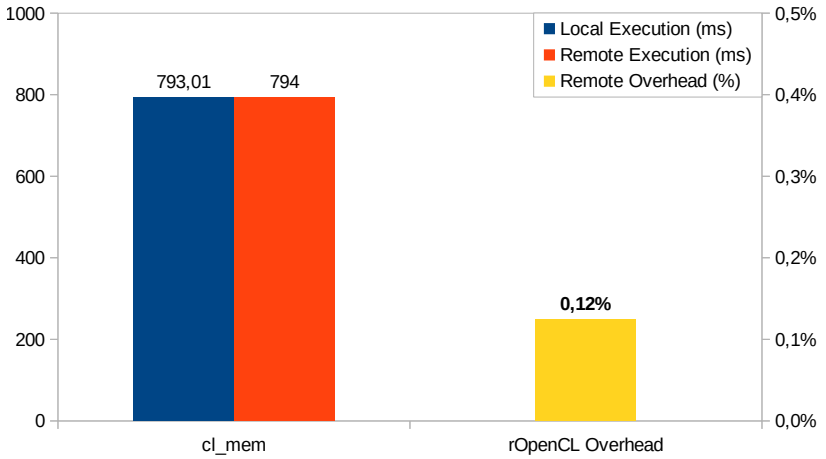


Figura 4.6: Resultados globais do *benchmark* cl-mem.

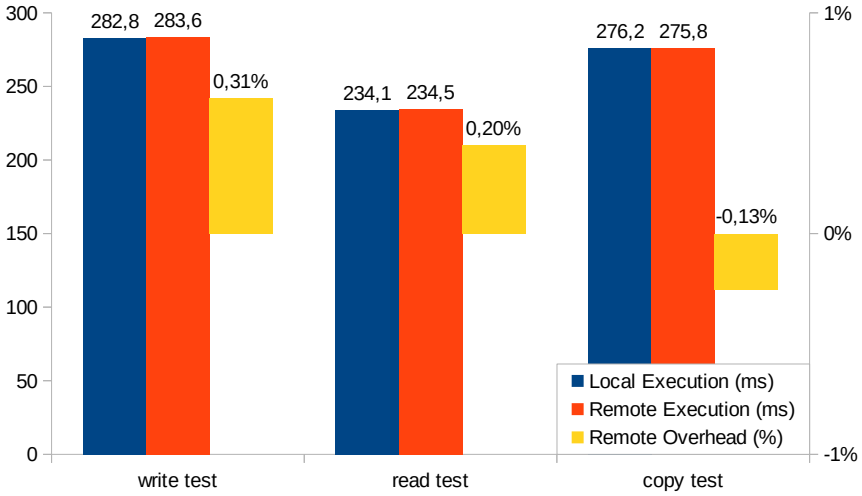


Figura 4.7: Resultados individuais dos sub-*benchmarks* do cl-mem.

porque o cl-mem apenas contabiliza o tempo de execução das primitivas OpenCL relacionadas com as operações sobre a memória, realizada por cada sub-*benchmark*. Além disso, como se referiu na secção 4.2, os dados usados pelos vários sub-*benchmarks* são gerados *on-the-fly*, no decurso dos testes, não sendo transferidos a partir do *host*.

A semelhança nos resultados globais da execução local e remota, estende-se aos resultados individuais dos sub-*benchmarks*, embora com uma nota digna de menção: o *copy test* exibe uma sobrecarga negativa (execução remota mais rápida que a local); dada a sua dimensão muito reduzida em termos absolutos, esta melhoria não é considerada estatisticamente relevante, sendo atribuível aos efeitos de *caching* referidos na secção 4.4.1.

4.6.3 clpeak

Como já foi referido, o clpeak é um *benchmark* híbrido que mede capacidades de pico dos dispositivos, em termos de processamento e largura de banda da memória. Os resultados obtidos para este *benchmark* podem ser consultados nos gráficos das figuras 4.8 e 4.9.

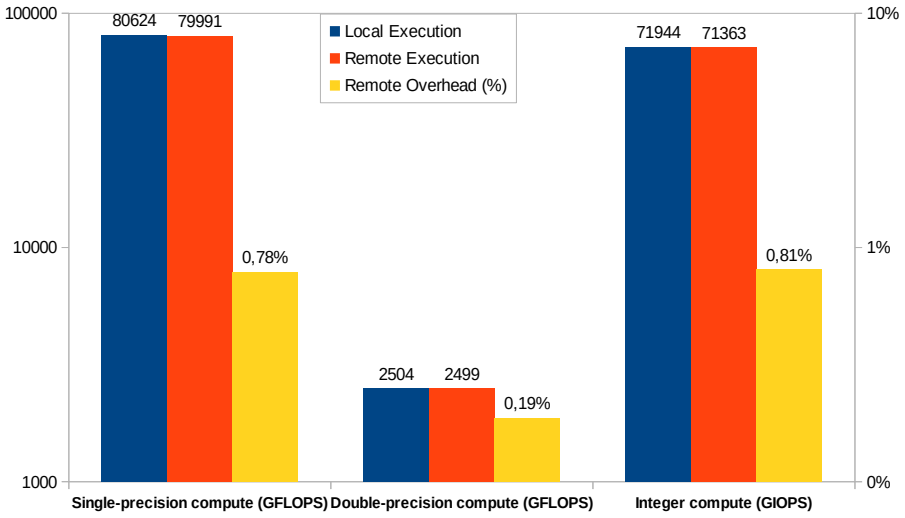


Figura 4.8: Resultados do *benchmark* clpeak (parte1).

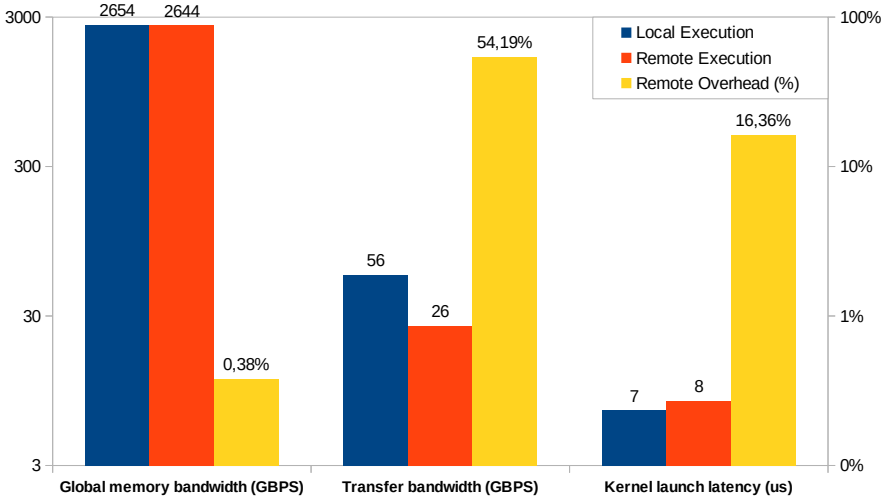


Figura 4.9: Resultados do *benchmark* clpeak (parte2).

Analisando os dois gráficos, constata-se que a sobrecarga da execução remota é quase nula em 4 dos 6 sub-*benchmarks*, muito modesta num deles (*Kernel launch latency*) e mais

expressiva naquele em que há uma quantidade maior de dados transferidos entre o *host* e o serviço rOpenCL (*Transfer bandwidth*), caindo o desempenho para metade.

4.6.4 FinanceBench

O FinanceBench é um *benchmark* baseado em aplicações de cálculo financeiro (rever secção 4.2), incluindo dois sub-*benchmarks* assentes em OpenCL (*Black-Scholes* e *Monte-Carlo*). Os resultados globais da execução do FinanceBench, e os resultados individuais dos 2 sub-*benchmarks* OpenCL, são apresentados nas figuras 4.10 e 4.11 respetivamente.

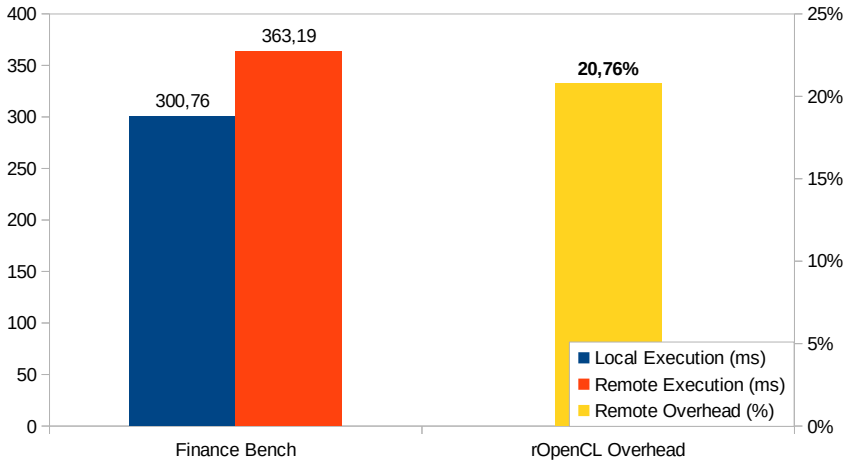


Figura 4.10: Resultados globais do *benchmark* FinanceBench.

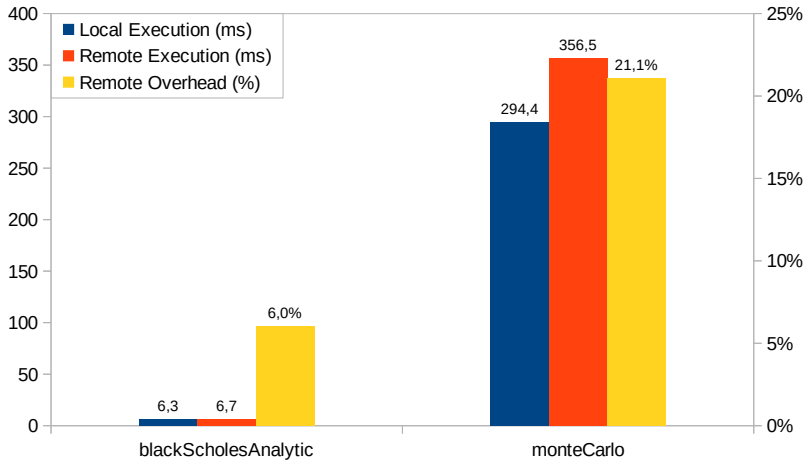


Figura 4.11: Resultados individuais dos sub-*benchmarks* do FinanceBench.

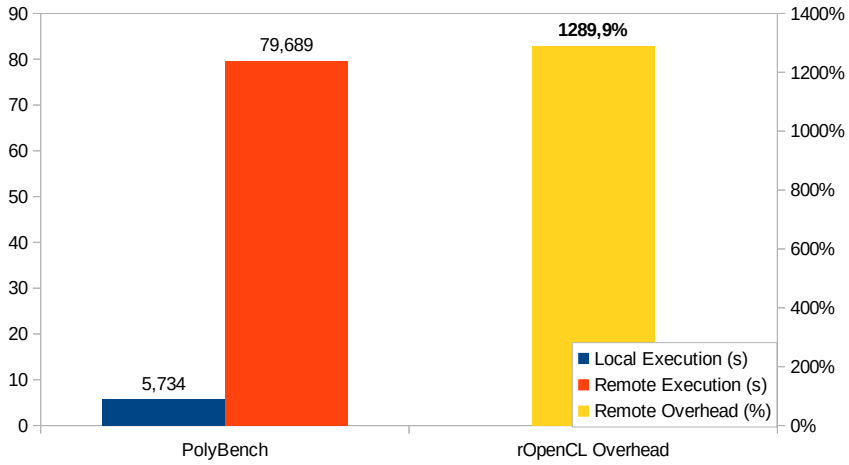


Figura 4.12: Resultados globais do *benchmark* PolyBench.

Em termos globais o impacto da execução remota é modesto (acréscimo de 20% no tempo de execução), devendo-se essencialmente ao sub-*benchmark Monte-Carlo* (com um acréscimo de 21,1%), já que no caso do *Black-Scholes* a sobrecarga é de apenas 6%. Este valores justificam-se pelo número contido de primitivas OpenCL envolvidas, em especial das que envolvem transferências de dados entre o *host* e o serviço rOpenCL.

4.6.5 PolyBench

O *PolyBench* é um *benchmark* de cálculo multifacetado, com 21 sub-*benchmarks* baseados em OpenCL (rever secção 4.2). Os resultados globais do *PolyBench* são apresentados na figura 4.12 (tempos em segundos), e os resultados dos vários sub-*benchmarks* estão distribuídos pelas figuras 4.13 (tempos em segundos) e 4.14 (tempos em milissegundos).

Globalmente, constata-se que neste *benchmark* a sobrecarga do rOpenCL é bastante elevada (≈1290%), sendo aliás a mais elevada de todos os *benchmarks* considerados. No entanto, é preciso observar os resultados individuais dos vários sub-*benchmarks* para ter a noção de que há uma grande variabilidade da sobrecarga provocada pelo rOpenCL.

Assim, nos sub-*benchmarks* mais demorados, de duração medida em segundos (figura 4.13), há 4 casos com sobrecargas muito pequenas (2%, 3%, 6% e 10%), e 5 com sobrecargas elevadas ou mesmo muito elevadas (786%, 2461%, 7383%, 10930% e 13621%).

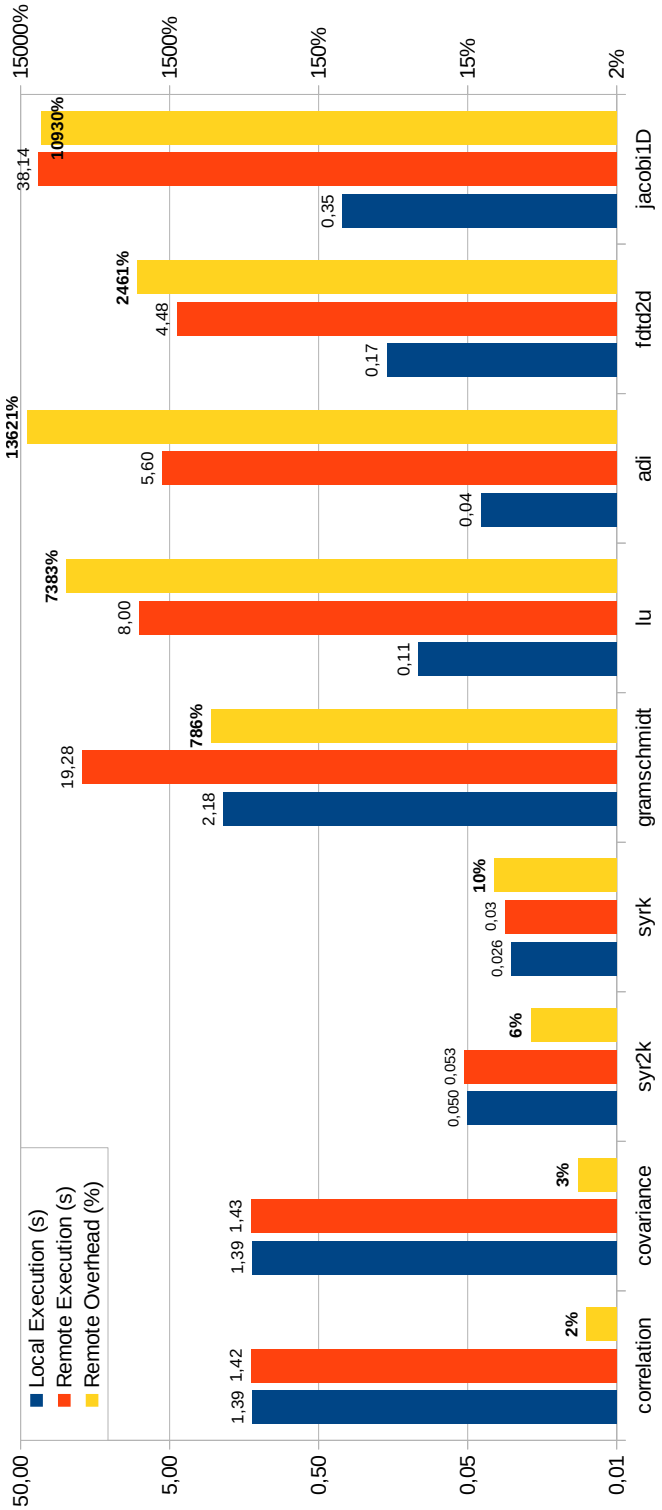
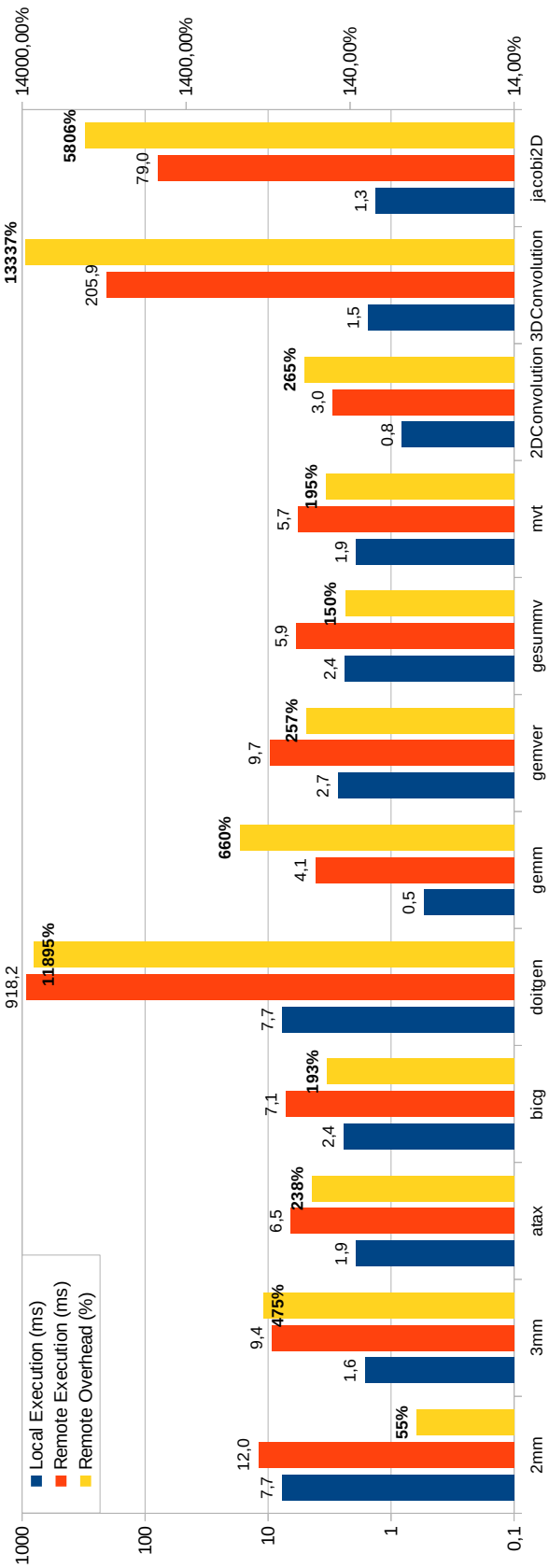


Figura 4.13: Resultados dos 9 sub-*benchmarks* mais demorados do PolyBench.



72 Figura 4.14: Resultados dos 12 sub-*benchmarks* mais rápidos do PolyBench.

Por outro lado, nos sub-*benchmarks* mais rápidos, de duração expressa em milissegundos (figura 4.14), as menores sobrecargas tendem a ser à partida elevadas, havendo-as também muito elevadas. Porém, como se trata de sub-*benchmarks* muito rápidos (mesmo quando executados através do rOpenCL), a sua contribuição no computo global é muito reduzida. De facto, contabilizando apenas os tempos da figura 4.13, a execução local do PolyBench levaria 5,701s e a remota 78,423s, traduzindo-se numa sobrecarga de 1275,6%; ou seja, na figura 4.12, só 14,3% de sobrecarga deriva dos sub-benchmarks mais rápidos.

Uma análise ao código dos sub-*benchmarks* revelou que grande parte manuseiam uma quantidade considerável de dados, fazendo sobressair com mais evidência o impacto das transações de rede realizadas pelo rOpenCL, traduzindo-se nas sobrecargas verificadas.

4.6.6 Rodinia

O *benchmark* Rodinia inclui 20 sub-*benchmarks* OpenCL, cobrindo diversos domínios de cálculo científico (rever secção 4.2). Nas figuras 4.15, 4.16 e 4.17 podem-se observar os resultados globais e individuais da sua execução, sendo os tempos da figura 4.17 expressos em milissegundos, e os das restantes figuras em segundos.

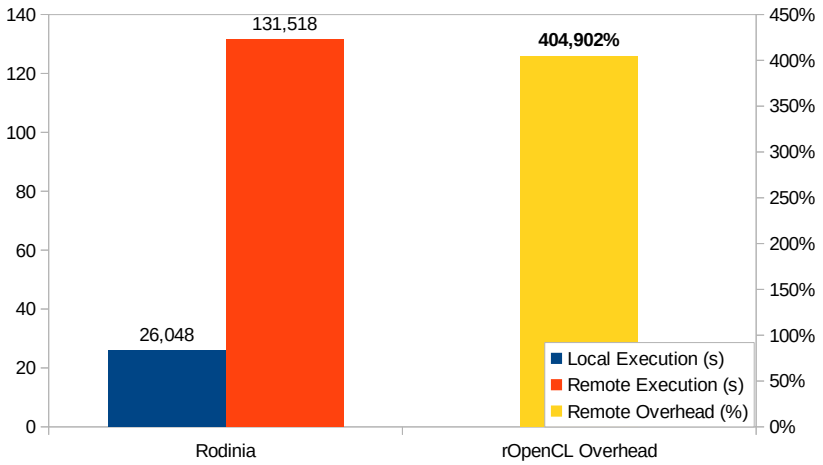


Figura 4.15: Resultados globais do *benchmark* Rodinia.

Verifica-se, portanto, que o *benchmark* completo demora 5 vezes mais tempo a executar através do rOpenCL (sobrecarga global de 405%). No entanto, tal como no PolyBench,

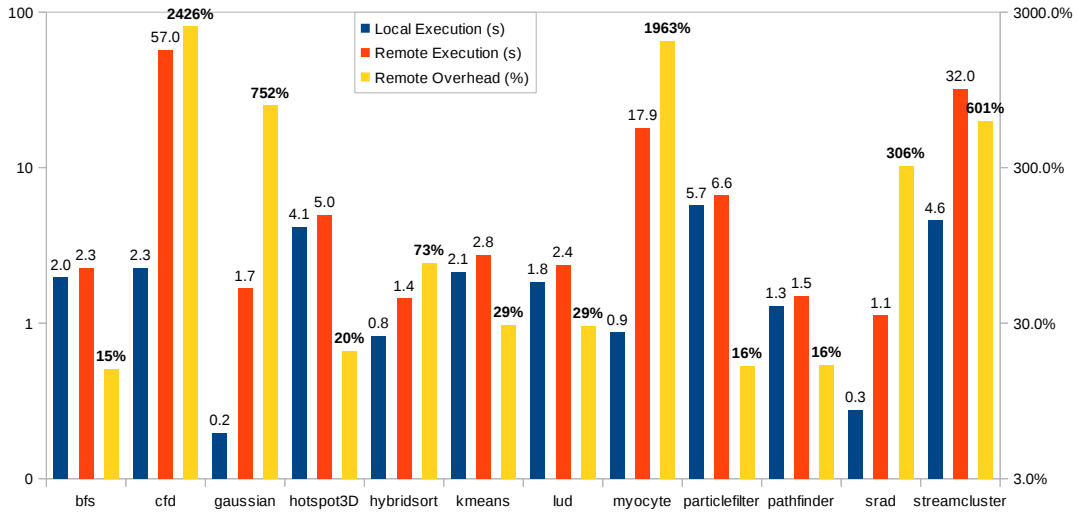


Figura 4.16: Resultados dos 12 sub-*benchmarks* mais demorados do Rodinia.

importa analisar o que se passa nos vários sub-*benchmarks*, pois também estes no Rodinia se distribuem por dois grupos, consoante a ordem de grandeza da sua duração.

Assim, para os 12 sub-*benchmarks* mais demorados (figura 4.16), as sobrecargas e peso no tempo total variam bastante: 6 deles (bfs, hotspot3D, kmeans, lud, particlefilter e pathfinder) exibem sobrecargas abaixo dos 30%, mas por terem uma duração inferior a 7s o seu impacto no tempo total é reduzido; outros exibem sobrecargas mais elevadas (hybridsort, com 73%; srad, com 306%; gaussian, com 752%), mas a sua duração é inferior a 2s; as maiores sobrecargas estão no entanto ligadas aos sub-*benchmarks* mais demorados (cfd, myocyte e streamcluster), com o consequente impacto no tempo total.

Quanto aos sub-*benchmarks* mais rápidos (figura 4.17), as suas sobrecargas variam menos e são mais mais contidas. Porém, o peso destes sub-*benchmarks* é residual. Levando só em conta os tempos da figura 4.16, a execução local do Rodinia levaria 26,046s (menos 0,002s) e a remota 131,514s (menos 0,004s), sendo a sobrecarga semelhante (404,929%).

Tal como no PolyBench, concluiu-se também que as movimentações de dados são as principais responsáveis pelas maiores sobrecargas observadas. Isto sugere que com tecnologias de rede mais rápidas que as usadas durante os testes, os resultados tenderiam a melhorar (incluindo para os outros *benchmarks*), algo a comprovar em trabalho futuro.

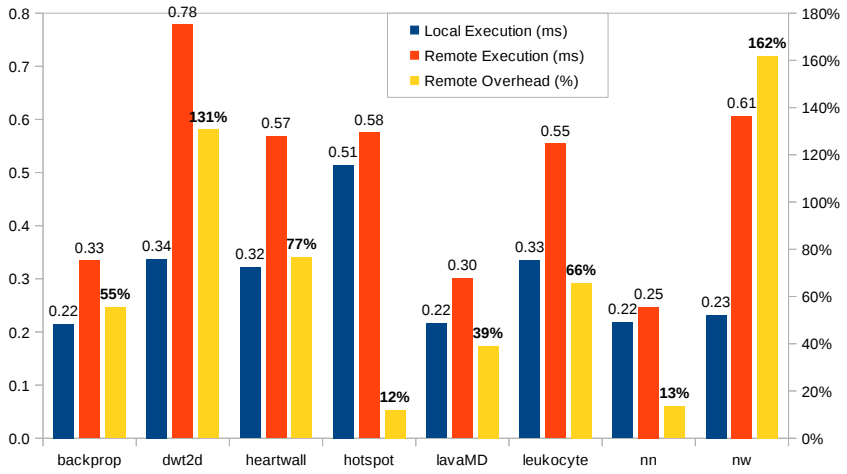


Figura 4.17: Resultados dos 8 sub-*benchmarks* mais rápidos do Rodinia.

4.6.7 Hashcat

O *hashcat* assume um papel especialmente importante nos testes mono-cliente, pois é o único teste com capacidade de utilizar mais que um GPU, permitindo explorar de forma completa os 4 GPUs disponíveis no ambiente de teste e perceber não só a sobrecarga da execução remota, como também o seu efeito na escalabilidade (para os restantes *benchmarks*, a única hipótese de explorar vários GPUs é a execução simultânea de várias instâncias, como acontece nos cenários multi-cliente discutidos nas secção 4.7).

De referir que o *hashcat* possui uma funcionalidade de *benchmark* integrada (um ataque de força bruta a um só *hash*). Contudo, envolve muito poucas transferências de dados entre o *host* e as GPUs, pois o espaço dos *hashes* é conhecido *a priori* e dividido por igual entre os GPUs usados. Assim, em alternativa, utilizou-se um exemplo [80] que recorre a um dicionário com regras, para quebrar 28 passwords encriptadas com a cifra *MD5*.

Os cenários testados são os apresentados na tabela 4.5, envolvendo diferentes combinações de GPUs locais e remotas, conforme permitido pelo ambiente de teste à data.

Os resultados (tempos e *speedup*) dos testes estão representados no gráfico da figura 4.18. Estes resultados mostram que: i) usando apenas uma GPU remota (cenário 0+1), é claramente vantajoso adicionar-lhe outra GPU remota (cenário 0+2), pois o *speedup* é quase ideal ($494/249 = 1{,}98 \approx 2$), atestando a boa escalabilidade do rOpenCL neste caso

Tabela 4.5: Combinações de GPUs para os testes com Hashcat.

Cenário de Teste	**GPUs Locais**	**GPUs Remotas**
C1	0	1
C2	0	2
C3	1	0
C4	1	1
C5	1	2
C6	2	0
C7	2	1
C8	2	2

particular com o Hashcat (para confirmar essa escalabilidade, seria necessário fazer testes com mais de 2 GPUs remotas); ii) uma GPU local (cenário 1+0) consegue, ainda assim, oferecer mais desempenho que 2 GPUs remotas; iii) adicionar a uma GPU local uma ou duas GPUs remotas (cenários 1+1 e 1+2) compensa notoriamente, o que mostra que o rOpenCL ajuda a melhorar efetivamente o desempenho, complementando GPUs locais com remotas; iv) essa melhoria ainda se regista quando a duas GPUs locais (cenário 2+0) se juntam 1 ou mais GPUs remotas (cenários 2+1 e 2+2), embora a melhoria já seja, compreensivelmente, menos expressiva, pois quanto mais GPUs locais, menor será a quantidade de trabalho entregue a GPUs remotas (isto para uma distribuição de trabalho *on-demand*, como é o caso do exemplo testado, e não para uma distribuição estática).

Em suma, o gráfico da figura 4.18 comprova que quanto mais GPUs forem utilizadas, maior é o *speedup*, e que é sempre benéfico juntar GPUs remotas às locais. Este exemplo é também um claro demonstrador da utilidade do rOpenCL na aceleração de casos reais.

4.6.8 Discussão

Os resultados apresentados nas seções anteriores foram obtidos a partir das configurações *default* tanto dos *benchmarks* como das máquinas virtuais que foram utilizadas, existindo no entanto, margem para obter melhores resultados. Essa melhoria pode ser alcançada recorrendo ao ajustes de parâmetros que alguns dos *benchmarks* permitem (como o FinanceBench) e/ou reajustando parâmetros de rede das máquinas virtuais.

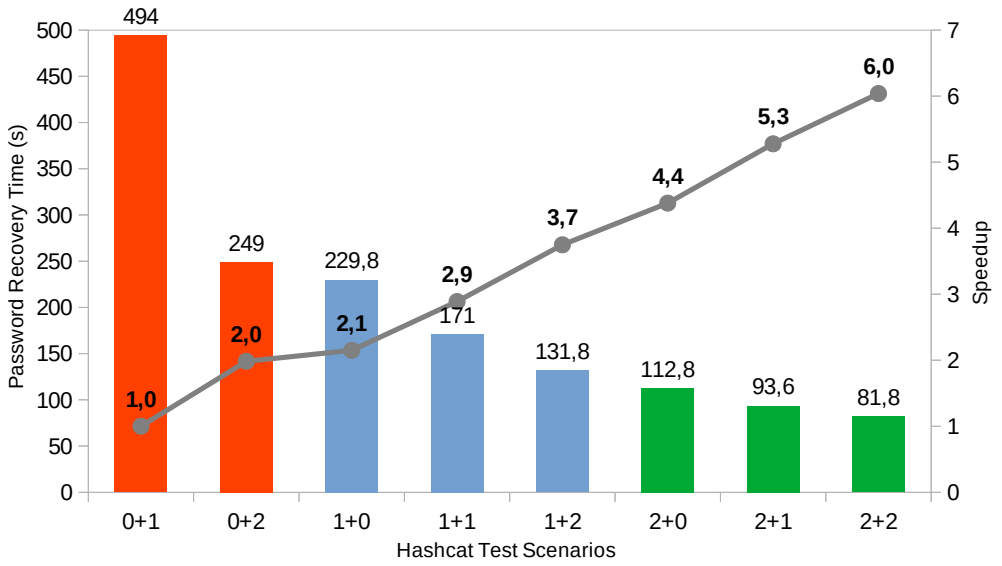


Figura 4.18: Resultados dos testes com Hashcat.

Todos os resultados obtidos estão dentro do esperado, existindo porém, algumas considerações que se impõem. Assim, a ordem de grandeza da sobrecarga obtida nas execuções remotas está fortemente ligada com a quantidade de dados que é transferida, ou seja, quanto maior for o conjunto de dados a enviar maior será a sobrecarga registada. Outra conclusão importante retirada é a influência do tempo de espera (pausa) entre cada execução: tempos maiores ajudam a reduzir a temperatura das GPUs, minimizando o impacto do *throttling* da frequência dos núcleos destas nos resultados obtidos.

Após a análise dos resultados obtidos, percebe-se que o impacto das transações de rede levadas a cabo pelo OpenCL varia consoante o tipo e o contexto da aplicação heterogénea que se execute. Contudo, em cenários em que não existam no *host* co-processadores, é claramente vantajoso explorar os aceleradores remotos, como será reforçado na secção 4.9.

4.7 Resultados dos Testes Multi-Cliente

Os resultados dos testes multi-cliente são apresentados nesta secção. Estes resultados cingem-se aos *benchmarks* FinanceBenck e cl-mem, o primeiro apenas no âmbito do cenário 1G, e o segundo no âmbito dos 3 cenários definidos na secção 4.4.2. Estes benchmarks

são representativos de duas situações diferentes: i) múltiplas instâncias de um mesmo *benchmark* provocam pequenos acréscimos de carga nas GPUs, porque o *benchmark* é demasiado rápido ou devido à própria natureza das tarefas que solicita às GPUs; ii) várias instâncias do mesmo *benchmark* resultam num crescimento linear da carga nas GPUs.

4.7.1 FinanceBench

O gráfico 4.19 mostra os tempos da execução dos dois sub-*benchmarks* do FinanceBench no cenário 1G, quando o número de instâncias simultâneas de cada sub-*benchmark* varia de 1 a 8, executando-se essas instâncias em máquinas virtuais diferentes, e tendo como alvo uma nona máquina, onde corre o serviço rOpenCL e existe um só GPU.

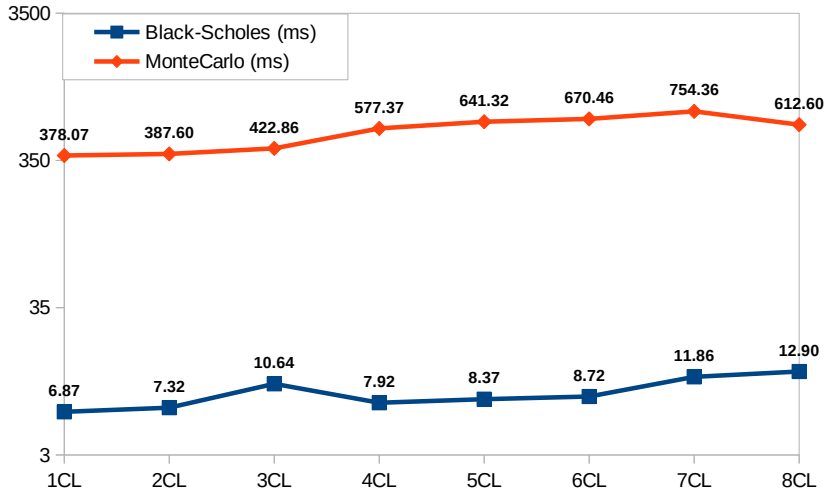


Figura 4.19: Tempos de execução do FinanceBench no cenário 1G.

Como se pode observar, os tempos de execução remota apresentam uma tendência crescente (visualmente pouco pronunciada, porque a escala é logarítmica no eixo dos tempos), salvo casos pontuais. Mas esse crescimento não é diretamente proporcional ao número de clientes. Por exemplo, o rácio entre o tempo com 7 clientes e com 1 cliente é de (754.36+11.86)/(378.07+6.87)=1.99, longe de 7, valor correspondente a uma proporcionalidade direta. Isto sugere que a taxa de utilização da GPU deverá ser baixa, pois os tempos de execução aumentam pouco de cada vez que se aumenta o número de clientes ativos, motivando a análise das métricas de carga e temperatura da GPU

(habitualmente com tendências correlacionadas), em busca de explicações. Os gráficos 4.20 e 4.21 mostram, precisamente, a evolução dessas métricas para o cenário em causa.

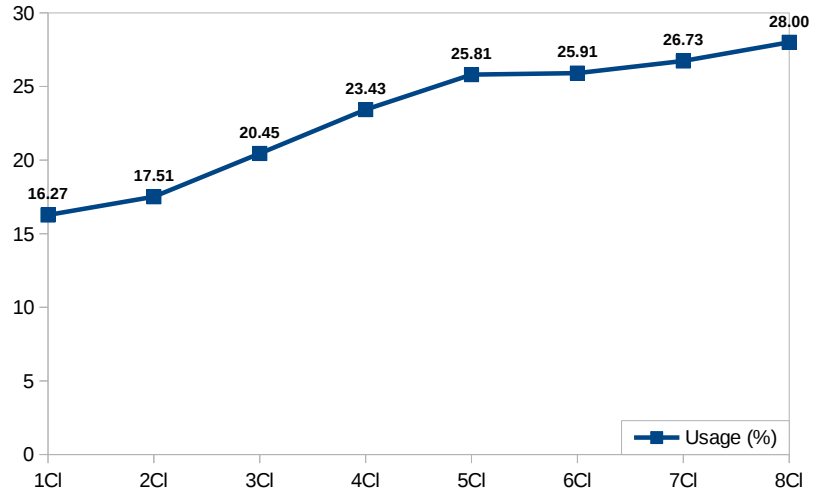


Figura 4.20: Carga máxima da GPU usada no cenário 1G com o FinanceBench.

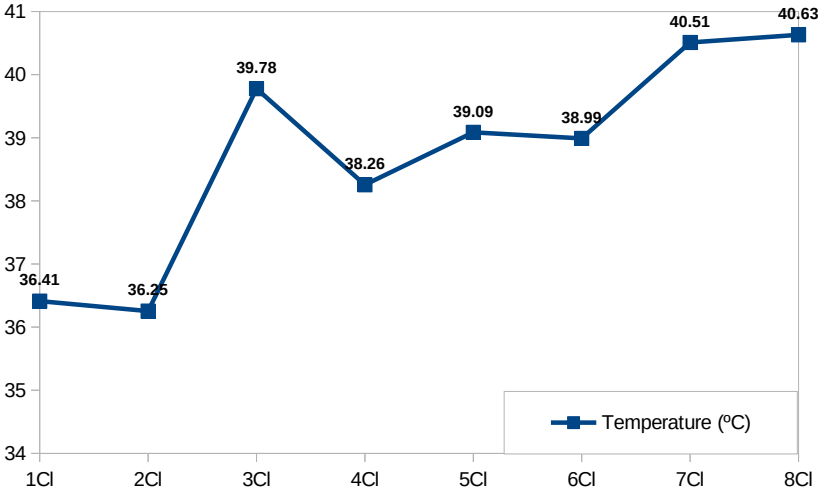


Figura 4.21: Temperatura máxima da GPU usada no cenário 1G com o FinanceBench.

O gráfico da figura 4.20 revela então que a carga da GPU aumenta efetivamente com o aumento do número de *benchmarks* simultâneos, mas aumenta pouco, variando entre os 16.27% (1 cliente) e os 28% (8 clientes), com uma razão de apenas 1,72 entre estes dois valores. A não introdução de carga suficiente na GPU faz com que esta consiga responder aos pedidos recebidos de forma relativamente rápida, levando assim a que o aumento do tempo de execução, à medida que o número de clientes aumenta, seja muito contido.

Quanto à temperatura, o gráfico da figura 4.21 mostra que acompanha a tendência crescente da carga, mas esse crescimento faz-se a uma razão inferior à do aumento da carga: entre 1 e 8 clientes ativos, esse rácio é de apenas 40,62/36,41=1,12, correspondendo a uma amplitude térmica de apenas 40,63°-36,41°=4,22°. As próprias temperaturas observadas são relativamente baixas, sendo menos provável uma redução no *clock* dos núcleos da GPU, favorecendo assim a obtenção dos tempos registados. Um factor que contribui para a contenção das temperaturas é a inclusão de tempos de pausa entre as 5 amostras colhidas para cada número diferente de clientes. Essa pausa acaba por beneficiar o desempenho.

Adicionalmente, o FinanceBench exerce em si pouca carga computacional sobre a GPU, executando por surtos curtos, da ordem de poucos milissegundos. Assim, apesar de todos os clientes começarem ao mesmo tempo o teste, o efeito de pico na carga pode ficar diluído, levando a que o aumento de clientes não seja tão sentido nos tempos obtidos.

4.7.2 cl-mem

Face ao comportamento observado no cenário 1G, optou-se por não testar o FinanceBench nos cenários 2G e 4G. Em vez disso, procurou-se um outro *benchmark* que gerasse carga suficiente para justificar o uso de 2 e 4 GPUs. O cl-mem foi o *benchmark* que se identificou possuir essas características, sendo os resultados dos seus testes multi-cliente apresentados de seguida, para os 3 sub-*benchmarks* que o compõem (*write test*, *read test* e *copy test*).

Cenário 1G Os resultados do cenário 1G para o cl-mem constam do gráfico da figura 4.22. Ao contrário do observado com o FinanceBench, verifica-se uma progressão linear, nos tempos de execução dos vários sub-*benchmarks*. De facto, esses tempos crescem de forma diretamente proporcional ao número de clientes ativos, como demonstram os valores da tabela 4.6, correspondentes à divisão de cada tempo pelo obtido com um só cliente.

A carga gerada na GPU pode ser observada no gráfico da figura 4.23. Os valores começam agora em 38,36% com 1 cliente, e atingem 80,55% com 8 clientes, correspondendo a um rácio de 80,55/38,36=2,15. Em comparação com o FinanceBench, a carga gerada pelo cl-mem é portanto francamente superior, reflexo das especificidades deste teste.

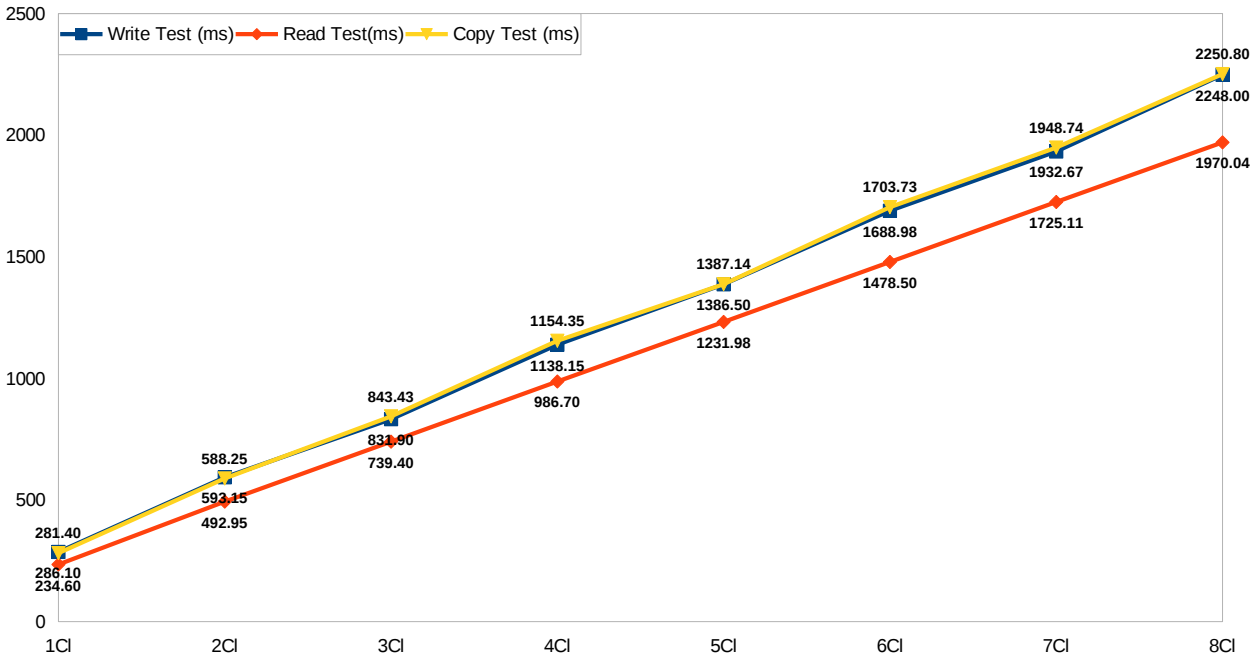


Figura 4.22: Tempos de execução do cl-mem no cenário 1G.

Tabela 4.6: Desacelerações para o cl-mem no cenário 1G.

clientes	**desaceleração(write)**	**desaceleração(read)**	**desaceleração(copy)**
1	-	-	-
2	2,07	2,10	2,09
3	2,91	3,15	3,00
4	3,98	4,21	4,10
5	4,85	5,25	4,93
6	5,90	6,30	6,05
7	6,76	7,35	6,93
8	7,86	8,40	8,00

Ainda em relação à carga, é possível adivinhar uma capacidade de encaixe de clientes adicionais, dado que a carga máxima ficou em torno dos 80%. A descoberta do número de clientes que conduz à saturação da GPU ficou, no entanto, para trabalho futuro.

A nível da evolução da temperatura, mostrada no gráfico da figura 4.24, a mesma cresce sempre (e de forma suave), à medida que aumenta o número de clientes ativos. Em

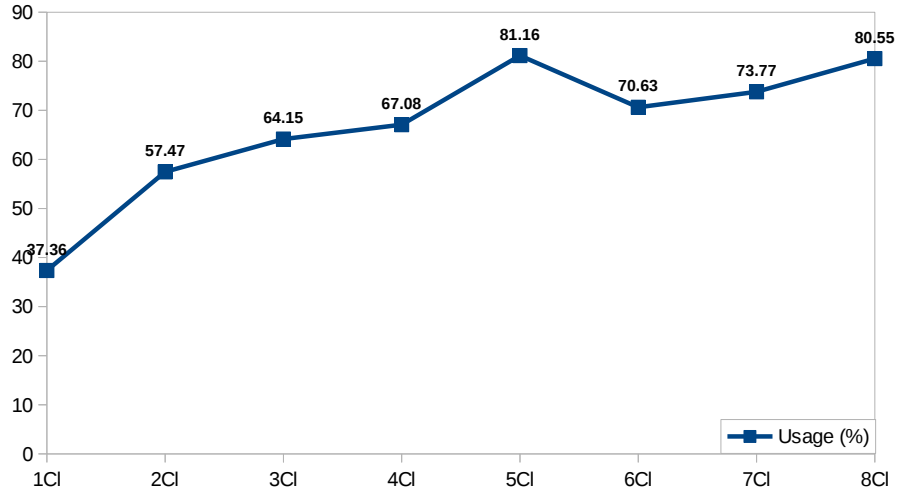


Figura 4.23: Carga máxima da GPU no cenário 1G com o cl-mem.

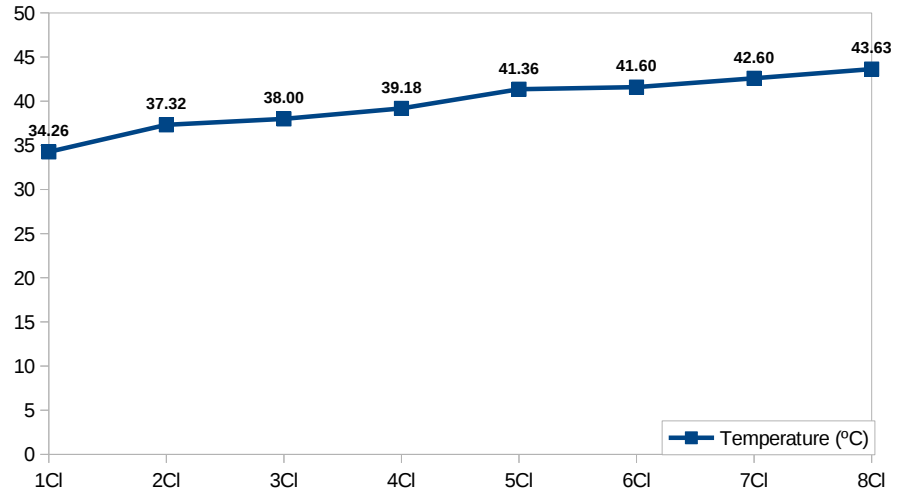


Figura 4.24: Temperatura máxima da GPU no cenário 1G com o cl-mem.

comparação com o FinanceBench, a amplitude térmica é sensivelmente o dobro (43,63°-34,26=9,37°), mas a temperatura máxima é superior em apenas 3°, mantendo-se as temperaturas relativamente baixas, tendo em conta a carga substancialmente mais elevada. No cenários 2G e 4G não se fornece o gráfico da temperatura, devido à evolução similar.

Cenário 2G Neste cenário, um só serviço rOpenCL expõe 2 GPUs, garantindo-se *a priori* uma distribuição uniforme dos clientes pelas GPUs. Assim, o número de clientes é necessariamente múltiplo do número de GPUs (2, 4, 6 e 8 clientes) e cada um escolhe

explicitamente uma das 2 GPUs, dividindo-se os clientes pelas GPUs de forma equitativa. Adicionalmente, usando-se 2 GPUs, é suposto que o tempo de execução do cenário 2G com n clientes seja similar ao do cenário 1G com $n/2$ clientes. Por esse motivo, na figura 4.25, que mostra o gráfico dos tempos de execução, os cenários 1G que é suposto demorarem o mesmo tempo que os correspondentes cenários 2G, surgem ao lado (antes) destes.

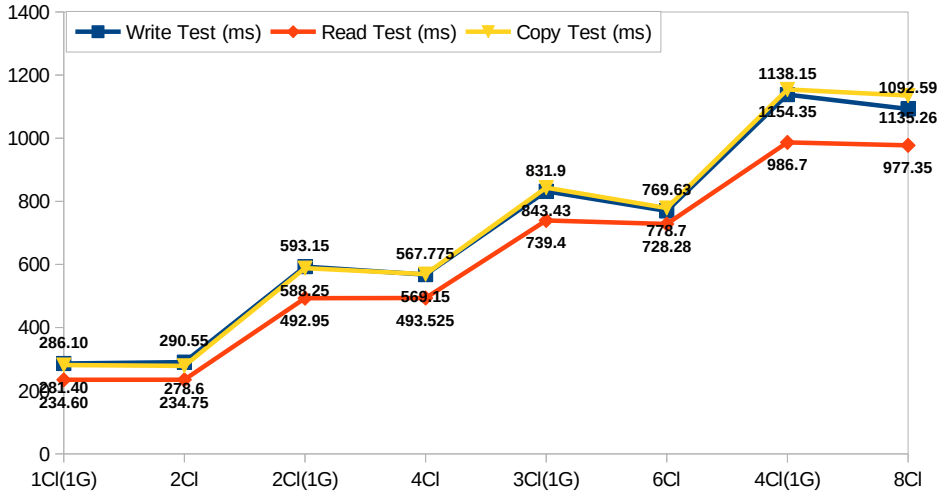


Figura 4.25: Tempos de execução do cl-mem nos cenário comparáveis 1G e 2G.

A observação do gráfico da figura 4.25 comprova precisamente a equivalência esperada: desde que o rácio do número de clientes pelo número de GPUs seja igual, o tempo de execução nos cenários 1G e 2G é semelhante. Verificam-se, no entanto, alguns tempos de execução ligeiramente menores nos cenários 2G, cuja causa não foi possível apurar. Ainda assim, o comportamento observado aponta para uma boa escalabilidade dos serviços rOpenCL, quando sujeitos à necessidade de operar com todas as GPUs disponíveis.

Quanto à carga nas GPUs, o gráfico da figura 4.26 mostra que as cargas na GPU0, usada em ambos os cenários 1G e 2G, são similares para os casos comparáveis desses cenários (sendo que nalgumas situações a GPU0 parece ter uma carga ligeiramente menor nos cenários 2G). Para a GPU1, os valores da carga andam próximos dos da GPU0, como seria de esperar de uma distribuição equitativa dos *benchmarks* clientes pelas duas GPUs.

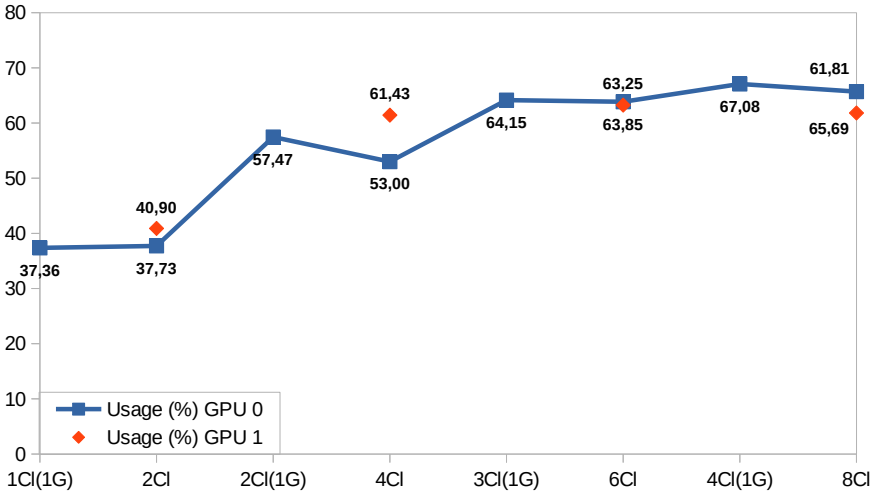


Figura 4.26: Carga máxima das GPUs nos cenários comparáveis 1G e 2G com o cl-mem.

Cenário 4G No cenário 4G, dois serviços rOpenCL, em máquinas distintas, expõem 2 GPUs cada um, disponibilizando um total de 4 GPUs. Para que todos os GPUs sejam usados em simultâneo, de forma equitativa, são necessários 4 ou 8 clientes, sendo garantido à partida, para cada número de clientes, a sua distribuição uniforme pelos 4 GPUs. Com este número de GPUs, o tempo de execução do cenário 4G com n = clientes (4 ou 8) deverá ser semelhante ao do cenário 1G com $n/4$ clientes (1 ou 2), e ao do cenário 2G com $n/2$ clientes (2 ou 4). Essa semelhança pode-se comprovar no gráfico da figura 4.27.

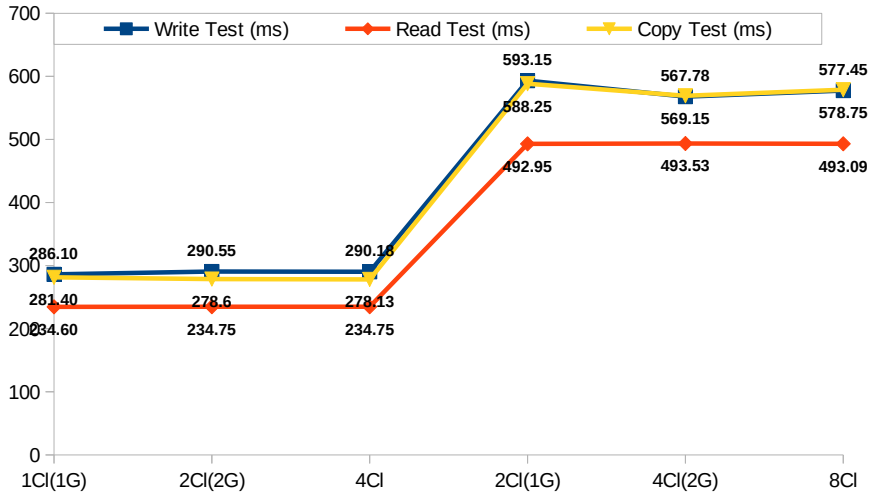


Figura 4.27: Tempos de execução do cl-mem nos cenário comparáveis 1G, 2G e 4G.

Novamente, são similares os tempos de execução, nas situações em que, independentemente do cenário (1G, 2G e 4G), a razão do número de clientes ativos pelo número de GPUs disponíveis é igual, reforçando o argumento da escalabilidade dos serviços rOpenCL.

No que diz respeito à carga, o gráfico da figura 4.28 representa a sua evolução, para os cenários 1G, 2G e 4G comparáveis, podendo-se observar a sua semelhança, como aliás é esperado, tendo em conta os valore antes observados dos tempos de execução.

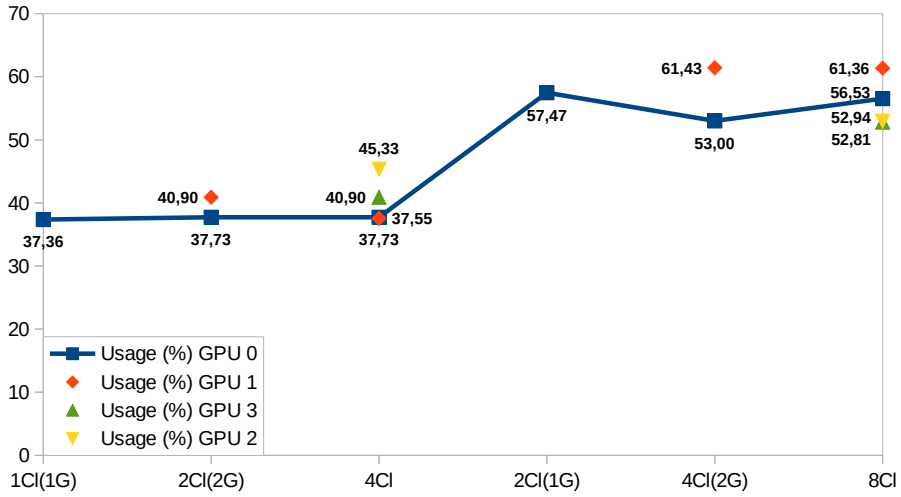


Figura 4.28: Carga máx. das GPUs nos cenários comparáveis 1G, 2G e 4G com o cl-mem.

4.8 Comparação com Outras Abordagens

Outro aspeto importante da análise ao desempenho do rOpenCL é a sua comparação com abordagens apresentadas no capítulo 2. Apesar de terem sido apresentadas 6 abordagens, apenas foi possível fazer uma comparação de desempenho com 3 delas: clOpenCL, VCL e dOpenCL (versão assente na biblioteca C++ Asio [41]). De facto, o código fonte do clusterCL e do Hybrid OpenCL não se encontra disponível. Já no caso do SnuCL, apesar da disponibilidade do código, não foi possível executar o exemplo de teste selecionado.

Para realizar esta comparação, foram usadas três novas máquinas virtuais, com algumas alterações face ao ambiente de teste descrito na secção 4.3. Assim, para avaliar o clOpenCL, foi necessário recuar à versão 4.8 do *kernel*, a última com suporte à biblioteca de comunicação Open-MX usada pelo clOpenCL. O valor do MTU teve também de ser

modificado, de 1500 para 9000 bytes. A duas das três máquinas virtuais foram também associadas GPUs, ficando cada uma dessas máquinas com 2 GPUs.

A aplicação OpenCL que serviu de base à comparação foi um exemplo de multiplicação de matrizes [81], originalmente usado no desenvolvimento do clOpenCL. Neste exemplo multiplicam-se duas matrizes quadradas com *floats*, de ordem *size* (ou seja, com *size* × *size* números), dividindo o trabalho equitativamente pelos vários co-processadores disponíveis, com base em fatias (*slices*) das matrizes. Usando como guia a publicação de referência do clOpenCL [26], testaram-se apenas três combinações <*size,slice*>: <8k,1k>,<16k,2k>,<24k,4k>. Para cada combinação variou-se o número de GPUs usadas, de 1 a 4, executando-se assim 3 × 4 = 12 testes para cada abordagem testada.

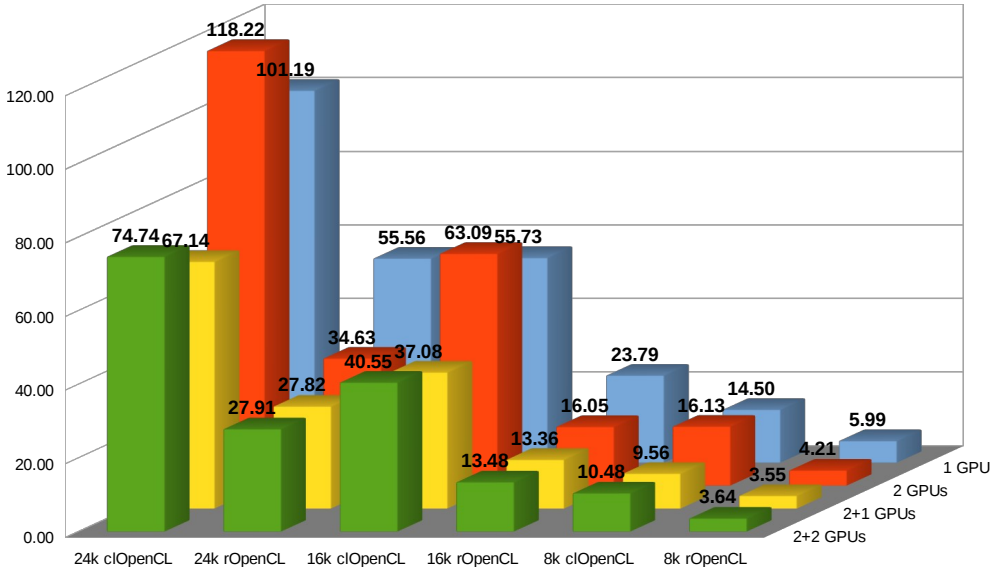


Figura 4.29: Multiplicação de Matrizes: rOpenCL vs clOpenCL (tempos em s).

Os gráficos das figuras 4.29, 4.30, e 4.31 mostram os resultados dos testes, permitindo comparar o rOpenCL com cada uma das 3 abordagens alternativas. A observação dos gráficos permite identificar algumas tendências gerais: i) o rOpenCL apresenta sempre melhor desempenho que o clOpenCL e que o VLC: ii) pelo contrário, o dOpenCL exibe sempre melhor desempenho que o rOpenCL, mas as diferenças são pequenas quando estão em causa matrizes de pequena dimensão, e acentuam-se com matrizes maiores; iii) em geral, com mais GPUs usadas, o tempo de execução diminui com toda as abordagens;

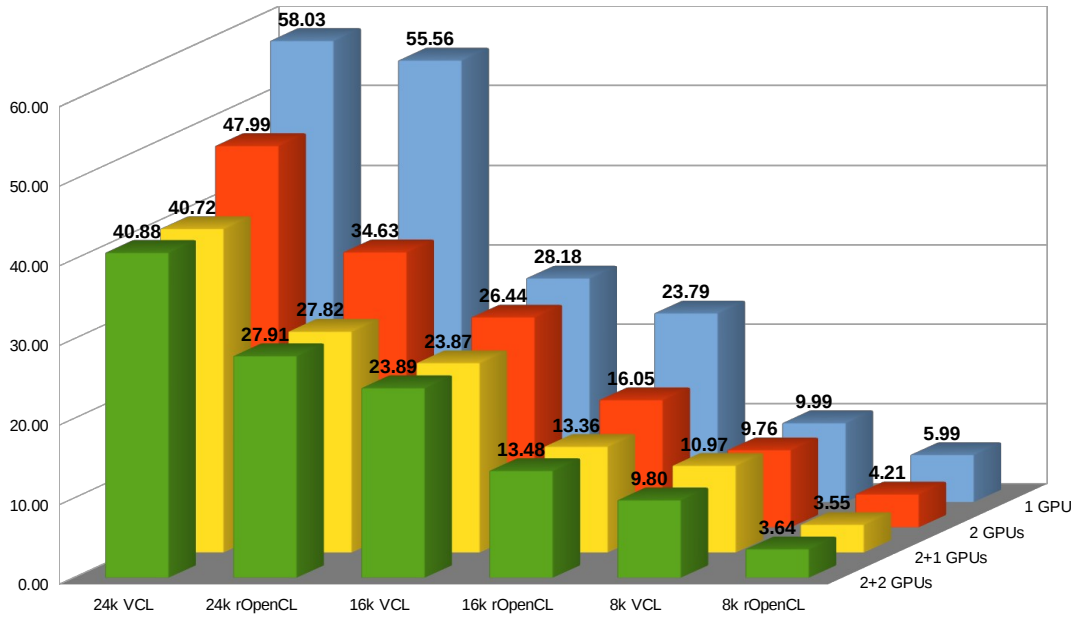


Figura 4.30: Multiplicação de Matrizes: rOpenCL vs VCL (tempos em s).

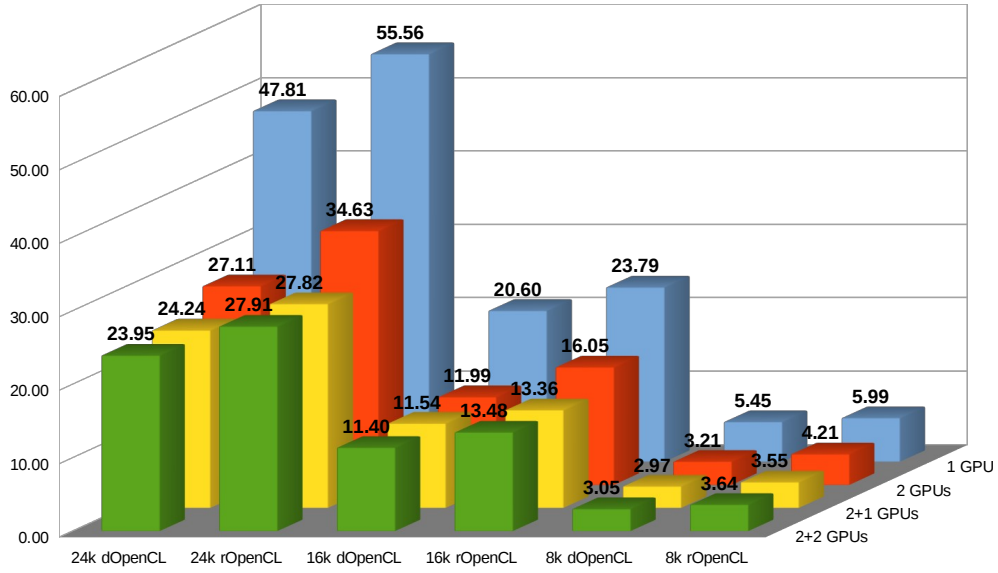


Figura 4.31: Multiplicação de Matrizes: rOpenCL vs dOpenCL (tempos em s).

no entanto, o clOpenCL apresenta alguns desvios importantes a esta regra, na passagem de 1 para 2 GPUs, e de 3 para 4 GPUs, não revelando escalabilidade nessas situações; iv) com o dOpenCL e o rOpenCL, que são as abordagens mais rápidas, os principais ganhos de desempenho estão na passagem de 1 para 2 GPUs, sendo os ganhos mais marginais

Tabela 4.7: Multiplicação de Matrizes: rOpenCL vs abordagens alternativas.

Métrica	**rOpenCL**	**clOpenCL**	**VCL**	**dOpenCL**
Tempo Total (s)	229,97	608,40	330,50	193,32
Speedup do rOpenCL	-	2,645	1,437	0,84

com 3 e 4 GPUs (e em especial com matrizes maiores); aliás, na passagem de 3 para 4 GPUs, o rOpenCL exibe sempre uma pequeno aumento do tempo de execução, levando a crer que atingiu, para este exemplo, o limite da escalabilidade.

A tabela 4.8 permite uma comparação, em termos globais, do desempenho exibido pelas quatro abordagens, tomando como ponto de referência o rOpenCL.

Tendo como pano de fundo o caso prático explorado, o rOpenCL oferece um desempenho claramente superior ao clOpenCL, posiciona-se de forma confortável perante o VCL, mas é ultrapassado, ainda que por pouco (em 16%) pelo dOpenCL. A este facto não será alheio o uso de comunicação assíncrona pelo dOpenCL, com base na biblioteca Boost.Asio [82], uma abordagem que poderá também ser explorada, futuramente, pelo rOpenCL.

4.9 Comparação com CPU Local

As avaliações já efetuadas ao rOpenCl demonstraram a sua estabilidade através da capacidade em suportar a execução remota de vários *benchmarks* de referência, provaram a escalabilidade dos seus serviços com um número variável de GPUs e clientes, e mostraram também os seus méritos relativos em comparação com outras abordagens do género.

No entanto, isto terá pouca importância se os custos da execução em co-processadores remotos forem demasiado elevados, comparados com a execução em CPUs locais, nomeadamente CPUs com elevado número de núcleos, cada vez mais frequentes nos sistemas de computação. Naturalmente, a opção pela execução em CPUs locais *multi-core* apenas se colocará na ausência de co-processadores locais. Mas é também para uma situação dessas que o rOpenCL foi concebido, ao permitir aceder a co-processadores remotos. Se compensa ou não, face à execução em CPUs locais, dependerá de cada caso em concreto.

Nesta secção, são apresentados tempos de execução do BabelStream e do cl-mem,

em CPU local e GPU remota. Estes *benchmarks* em particular foram escolhidos para esta comparação porque são os únicos, dos considerados nesta tese, com capacidade de execução em CPU, além de GPU. Os da execução em GPU remota são repescados das figuras 4.6.1 e 4.6.2. Os resultados da execução em CPU local são produzidos por duas plataformas OpenCL diferentes, com capacidade de explorar CPUs *multi-core*: a plataforma POCL [31] e a da Intel [29]. Para a execução em CPU, maximizaram-se os recursos da máquina virtual usada, atribuindo-lhe uma vCPU de 64 *núcleos* virtuais (na prática, 16+16=32 núcleos, cada um com suporte a 2 *threads*), e 256 GB de RAM.

Os gráficos das figuras 4.32 e 4.33 representam os resultados obtidos, permitindo uma análise e comparação do desempenho de cada sub-*benchmark* do BabelStream e do cl-mem. A tabela 4.8 sintetiza os tempos totais dos dois *benchmarks* e apresenta também o *speedup* fornecido pelo rOpenCL, no uso de uma GPU remota, face à execução dos *benchmarks* em CPU local com ambas as plataformas OpenCL consideradas.

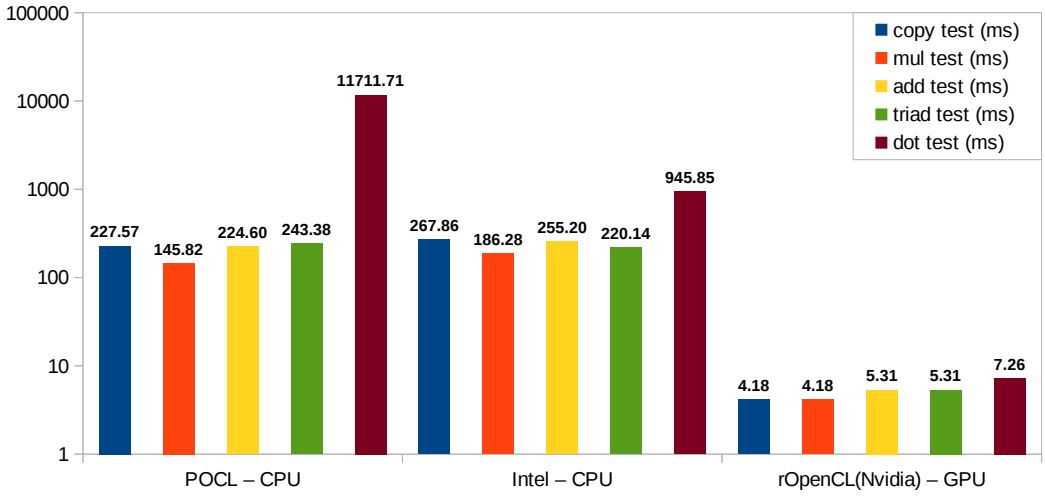


Figura 4.32: Tempos (ms) da execução do BabelStream em CPU local vs GPU remota.

Tabela 4.8: Execução do BabelStream e do cl-mem: CPU local vs GPU remota.

Benchmark	**Métrica**	**rOpenCL**	**POCL**	**Intel Runtime**
BabelStream	Tempo Total (ms)	26,24	12553,08	1875,33
	Speedup do rOpenCL	-	478,39	71,47
cl-mem	Tempo Total (ms)	793,9	567678,94	457843,10
	Speedup do rOpenCL	-	715,05	576,7

Da observação dos gráficos e da tabela, podem-se extrair as seguintes conclusões: i)

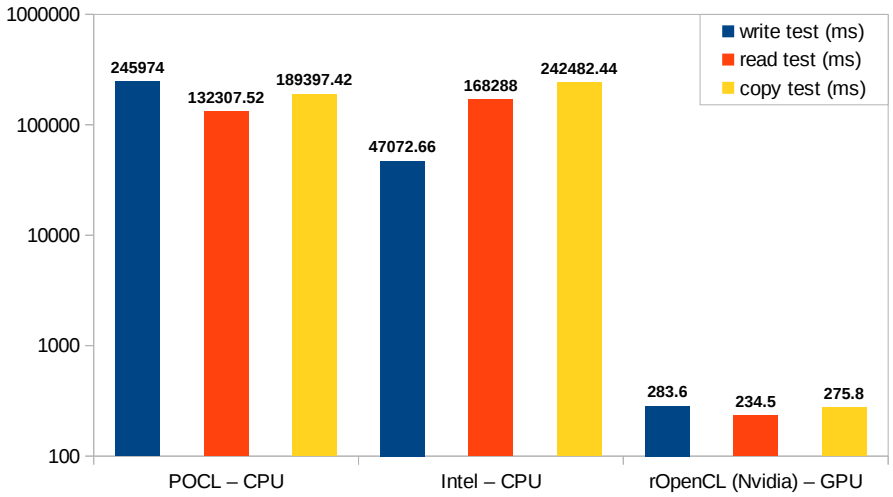


Figura 4.33: Tempos (ms) da execução do cl-mem em CPU local vs GPU remota.

os *speedups* exibidos pelo rOpenCL não deixam dúvidas sobre a vantagem, no caso destes *benchmarks*, em executá-los em GPU remota, em vez de em CPU local; as acelerações conseguidas chegam a ser da ordem das centenas; ii) a execução em GPU, ainda que remota, importa em tempos de execução sem diferenças significativas entre os vários sub-*benchmarks* do BabelStream e do cl-mem; já na execução em CPU local, a duração do teste *dot* do BabelStream destaca-se face aos outros e essa duração é muito diferente entre as plataformas POCL e Intel; por seu turno no no cl-mem, a plataforma de CPU usada (POCL ou Intel) influencia os *rankings* dos sub-*benchmarks*, nomeadamente do *write test*.

Em suma, os resultados da comparação levada a cabo nesta secção são particularmente encorajadores, realçando a validade do projeto desenvolvido nesta tese e o bom desempenho conseguido pela implementação realizada. Por explorar, ficam cenários práticos em que a GPUs remotas se soma também a contribuição de CPUs remotos, no pressuposto de que estes são consideravelmente mais céleres do que os locais, de forma a esconder o impacto da rede. A exploração desses cenários depende, no entanto, da existência de aplicações ou *benchmarks* capazes de tirar partido de múltiplos co-processadores de múltiplos tipos (CPU e GPU) em simultâneo (algo que o Hashcat, por exemplo, não é capaz de fazer, dado que apenas explora múltiplas GPUs). A identificação ou programação de aplicações eficientes desse género é em si um desafio, que fica para trabalho futuro.

Capítulo 5

Conclusão

Nesta dissertação apresentou-se o rOpenCL, uma nova solução que permite que aplicações OpenCL tenham acesso a co-processadores remotos, desde que acessíveis por TCP/IP, o que cobre virtualmente todos os cenários de rede relevantes, seja no domínio especializado do HPC, seja fora dele, e mesmo que haja *routing* envolvido.

A cobertura de mais de 70% das funções OpenCL 1.2, associada à forma como o rOpenCL foi implementado, permite que uma grande variedade de aplicações OpenCL pré-compiladas dele tirem partido, sem necessidade de acesso ou alterações ao código fonte. Este nível de transparência foi alcançado através da integração do *driver* rOpenCL no *runtime* do OpenCL, permitindo o acesso a um cenário distribuído de execução remota de pedidos OpenCL, com a mesma facilidade com que se invocam para execução local.

A estabilidade e escalabilidade do rOpenCL foram demonstradas pelos resultados dos *benchmarks* realizados, permitindo retirar conclusões importantes sobre as vantagens do uso do rOpenCL. Em particular, percebeu-se que apesar da rede poder representar um *bottleneck* neste tipo de soluções, aplicações heterogéneas "embaraçosamente paralelas", com poucas trocas de dados com a componente *host*, tirarão bom partido do rOpenCL.

Ficou também demonstrado o mérito do rOpenCL face a outras abordagens do género. É certo que o dOpenCL conseguiu um desempenho 16% superior, muito provavelmente devido ao uso de comunicação assíncrona, mas esta abordagem deixou de ser desenvolvida e mantida desde 2013, deixando de ser aconselhável a sua adoção, devido ao risco de

incompatibilidade com novas versões do *kernel* do Linux. Algo semelhante, no que toca à obsolescência, acontece com o clOpenCL, cuja camada de comunicação apenas funciona em *kernels* até à versão 4.8. Como alternativa ao rOpenCL resta apenas o VCL, com uma cobertura da API do OpenCL similar à do rOpenCL, e que ainda é mantido atualmente; porém, é quase 50% mais lento que o rOpenCL, e bastante menos transparente (as aplicações necessitam de ser recompiladas e não se integra com o mecanimo ICD Loader).

Numa situação em que não existem co-processadores locais, o normal é recorrer apenas à CPU local para fornecer capacidade de processamento às aplicações heterogéneas. Porém, ficou demonstrado nesta dissertação que, apesar da sobrecarga introduzida pela rede, existem aplicações para as quais compensa utilizar co-processadores (GPUs, no caso testado) remotos em alternativa (no fundo, essa é a razão de ser do rOpenCL).

O rOpenCL apenas fornece suporte para funções da versão 1.2 da norma OpenCL. No entanto, a versão 3.0 [83], lançada em 27 de Abril de 2020, apenas obriga a que os *vendors* suportem as funcionalidades da versão 1.2, tornando opcionais todas as funções das versões 2.x. Desta forma, o rOpenCL, acaba por estar alinhado com aquilo que a versão mais recente da especificação considera fundamental.

5.1 Trabalho Futuro

O estágio atual do rOpenCL permite afirmar que os objetivos propostos para esta dissertação foram plenamente alcançados (incluindo a validação por pares através de uma publicação científica). Contudo, foram sendo identificados outros pontos a desenvolver, e caminhos a percorrer, que apenas não o foram agora para não alongar o trabalho para lá do razoável, no contexto de uma dissertação de mestrado. Assim, algumas tarefas que ficaram para trabalho futuro (entre outras identificadas no corpo deste documento) são:

- Revisitar a comunicação com UDP, procurando mitigar os problemas de perda de pacotes identificados, a fim de tornar o UDP numa alternativa ao TCP, robusta e de melhor desempenho (pelo menos em segmentos de rede sem *routing* envolvido).

- Avaliar a possibilidade de recorrer a comunicação assíncrona, seja com base em sockets BSD, seja com base noutras *frameworks* (e.g., Boost.Asio).

- Optimizar o código relacionado com gestão de objetos, mensagens e conexões, eventualmente explorando outras estruturas de dados alternativas às *Red-Black Trees*, mais *cache-friendly*; isto implica realizar um estudo de *profiling* do código, com ferramentas adequadas, que permitam identificar os eventuais *hotspots*.

- Alargar o grau de cobertura da especificação OpenCL 1.2 e resolver o maior número possível das limitações da cobertura atual, identificadas na secção 3.10.

- Realizar testes de escalabilidade com mais clientes e/ou mais GPUs, procurando identificar os verdadeiros limites da implementação actual, bem como recolher *feedback* experimental que possa ser usado na melhoria do código do rOpenCL.

- Avaliar o acesso partilhado de máquinas virtuais, através do rOpenCL, a GPUs instalados no próprio sistema hospedeiro:

  - GPUs que cooperam com um *hypervisor*, a fim de poderem ser partilhadas por várias máquinas virtuais no mesmo sistema hospedeiro, são muito dispendiosas; por outro lado, GPUs mais acessíveis não dispõem dessa funcionalidade de partilha, suportando apenas o *passthrough* a uma máquina virtual, que depois usa a GPU em exclusividade; com o rOpenCL torna-se possível partilhar uma destas GPUs por várias máquinas virtuais, instalando um serviço rOpenCL no sistema hospedeiro e o *driver* rOpenCL em cada máquina virtual alojada nesse sistema; as transações de rede entre os *drivers* e o serviço serão feitas via memória partilhada, com um potencial de desempenho que merece ser avaliado.

- Realizar pelo menos mais uma publicação científica, tirando partido do trabalho realizado e consolidado desde a primeira publicação.

Bibliografia

[1] S. Mittal e J. Vetter, «A survey of CPU-GPU heterogeneous computing techniques,» *ACM Computing Surveys*, vol. 47, jul. de 2015.

[2] A. Cano, «A survey on graphic processing unit computing for large-scale data mining,» *WIREs Data Mining and Knowledge Discovery*, vol. 8, n.º 1, e1232, 2018.

[3] H. Wang, H. Peng, Y. Chang e D. Liang, «A survey of GPU-based acceleration techniques in MRI reconstructions,» *Quantitative Imaging in Medicine and Surgery*, vol. 8, pp. 196–208, mar. de 2018.

[4] M. Daga, A. M. Aji e W. Feng, «On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing,» em *2011 Symposium on Application Accelerators in High-Performance Computing*, jul. de 2011, pp. 141–149.

[5] J. Ross, D. Richie, S. Park, D. Shires e L. Pollock, «A case study of OpenCL on an Android mobile GPU,» *2014 IEEE High Performance Extreme Computing Conference, HPEC 2014*, fev. de 2015.

[6] A. Acosta, C. Merino e J. Totz, «Analysis of OpenCL Support for Mobile GPUs on Android,» mai. de 2018, pp. 1–6, ISBN: 978-1-4503-6439-3.

[7] D. Kaeli e D. Akodes, «The convergence of HPC and embedded systems in our heterogeneous computing future,» em *2011 IEEE 29th International Conference on Computer Design (ICCD)*, out. de 2011, pp. 9–11.

[8] M. Ditty, T. Architecture, J. Montrym e C. Wittenbrink, «NVIDIA'S Tegra K1 system-on-chip,» em *2014 IEEE Hot Chips 26 Symposium (HCS)*, ago. de 2014, pp. 1–26.

[9] A. Olofsson, T. Nordström e Z. Ul-Abdin, «Kickstarting high-performance energy-efficient manycore architectures with Epiphany,» em *2014 48th Asilomar Conference on Signals, Systems and Computers*, nov. de 2014, pp. 1719–1726.

[10] C. Kachris e D. Soudris, «A survey on reconfigurable accelerators for cloud computing,» em *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, ago. de 2016, pp. 1–10.

[11] J. Prades e F. Silla, «A Live Demo for Showing the Benefits of Applying the Remote GPU Virtualization Technique to Cloud Computing,» mai. de 2017, pp. 735–738.

[12] NVIDIA. (). «Record 136 NVIDIA GPU-Accelerated Supercomputers Feature in TOP500 Ranking,» URL: `https://blogs.nvidia.com/blog/2019/11/19/record-gpu-accelerated-supercomputers-top500/` (acedido em 19/11/2019).

[13] TOP500.org. (). «June 2020 | TOP 500,» URL: `https://www.top500.org/lists/top500/2020/06/`.

[14] NVIDIA. (). «CUDA Driver API :: CUDA Toolkit Documentation,» URL: `https://docs.nvidia.com/cuda/cuda-driver-api/index.html` (acedido em 22/01/2020).

[15] The Khronos Group. (). «OpenCL Overview,» URL: `https://www.khronos.org/opencl/` (acedido em 22/01/2020).

[16] ——, (). «SYCL Overview,» URL: `https://www.khronos.org/sycl/` (acedido em 22/01/2020).

[17] K. D. Gregory e A. Miller, *C++ AMP: Accelerated Massive Parallelism with Microsoft Visual C++*, 1st, sér. Developer Reference. Microsoft Press, out. de 2012.

[18] OpenACC-Standard.org. (). «The OpenACC Application Programming Interface (version 3.0),» URL: `https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC.3.0.pdf` (acedido em 22/01/2020).

[19] OpenMP Architecture Review Board. (). «OpenMP Application Programming Interface,» URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf (acedido em 22/01/2020).

[20] P. T. /. T. N. Platform. (). «Building Bigger, Faster GPU Clusters Using NVSwitches,» URL: https://www.nextplatform.com/2018/04/13/building-bigger-faster-gpu-clusters-using-nvswitches/ (acedido em 13/04/2018).

[21] NVIDIA. (). «Developing a Linux Kernel Module using GPUDirect RDMA,» URL: https://docs.nvidia.com/cuda/pdf/GPUDirect%5C_RDMA.pdf (acedido em 11/2019).

[22] Advanced Micro Devices. (). «ROCnRDMA: ROCm Driver RDMA Peer to Peer Support,» URL: https://github.com/rocmarchive/ROCnRDMA (acedido em 2016).

[23] Message Passing Interface Forum. (). «MPI: A Message-Passing Interface Standard - Version 3.1,» URL: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf (acedido em 04/06/2015).

[24] NVIDIA. (). «An Introduction to CUDA-Aware MPI,» URL: https://devblogs.nvidia.com/introduction-cuda-aware-mpi/ (acedido em 2013).

[25] R. Alves e J. Rufino, «Extending heterogeneous applications to remote co-processors with rOpenCL,» English, em *Proceedings - Symposium on Computer Architecture and High Performance Computing*, vol. 2020-September, 2020, pp. 305–312. URL: www.scopus.com.

[26] A. Alves, J. Rufino, A. Pina e L. Santos, «clOpenCL - Supporting Distributed Heterogeneous Computing in HPC Clusters,» jan. de 2013, pp. 112–122. DOI: 10.1007/978-3-642-36949-0_14.

[27] NVIDIA. (). «OpenCL NVIDIA Developer,» URL: https://developer.nvidia.com/opencl.

[28] Advanced Micro Devices. (). «AMD ROCm Open Ecosystem,» URL: https://www.amd.com/en/graphics/servers-solutions-rocm.

[29] Intel. (). «Intel SDK for OpenCL Applications,» URL: https://%20software.intel.com/content/www/us/en/develop/tools/opencl-sdk.html.

[30] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala e H. Berg, «pocl: A Performance-Portable OpenCL Implementation,» English, *Int. Journal of Parallel Programming*, vol. 43, n.º 5, pp. 752–785, 2015, ISSN: 0885-7458. URL: http://dx.doi.org/10.1007/s10766-014-0320-y.

[31] P. developers, *Portable Computing Language*, http://portablecl.org/, (accessed Abril 07, 2020).

[32] R. Aoki, S. Oikawa, T. Nakamura e S. Miki, «Hybrid OpenCL: Enhancing OpenCL for Distributed Processing,» em *2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications*, 2011, pp. 149–154.

[33] J.-H. Hong, J.-Y. Park e K. Chung, «Parallel LDPC Decoding on a Heterogeneous Platform using OpenCL,» *KSII Transactions on Internet and Information Systems*, vol. 10, pp. 2648–2668, jun. de 2016. DOI: 10.3837/tiis.2016.06.011.

[34] The Khronos Group. (). «Khronos Group Releases OpenCL 3.0,» URL: https://www.khronos.org/news/press/khronos-group-releases-opencl-3.0 (acedido em 27/04/2020).

[35] ——, (). «The OpenCL Specification 1.2,» URL: https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf (acedido em 08/06/2020).

[36] A. Barak e A. Shiloh. (). «The VirtualCL (VCL) Cluster Platform,» URL: http://www.mosix.cs.huji.ac.il/vcl/VCL%5C_wp.pdf (acedido em 2011).

[37] A. Barak e A. Shiloh. (). «The MOSIX Cluster Management System for Distributed Computing on Linux Clusters and Multi-Cluster Private Clouds,» URL: http://www.mosix.cs.huji.ac.il/pub/MOSIX%5C_wp.pdf (acedido em 2014).

[38] A. Barak e A. Shiloh, «The MOSIX Virtual OpenCL (VCL) Cluster Platform,» em *Proceedings of the Intel European Research and Innovation Conference*, out. de 2011, p. 196. URL: http://developer.amd.com/wordpress/media/2013/06/2913_2_final.pdf.

[39] P. Kegel, M. Steuwer e S. Gorlatch, «dOpenCL: Towards a Uniform Programming Approach for Distributed Heterogeneous Multi-/Many-Core Systems,» em *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, 2012, pp. 174–186.

[40] F. Glinka, A. Ploss, J. Müller-Iden e S. Gorlatch, «RTF: A real-time framework for developing scalable multiplayer online games,» jan. de 2007, pp. 81–86. DOI: 10.1145/1326257.1326272.

[41] C. M. Kohlhoff. (). «Asio C++ library,» URL: http://think-async.com/Asio/ (acedido em 2020).

[42] I. B. R. Centre, *Open-MX*, http://open-mx.gforge.inria.fr/, (accessed Abril 07, 2020).

[43] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo e J. Lee, «SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters,» em *Proceedings of the 26th ACM International Conference on Supercomputing*, sér. ICS '12, San Servolo Island, Venice, Italy, jun. de 2012, pp. 341–352, ISBN: 9781450313162.

[44] J. Kim, G. Jo, J. Jung, J. Kim e J. Lee, «A distributed OpenCL framework using redundant computation and data replication,» em *PLDI '16: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, jun. de 2016, pp. 553–569.

[45] V. Raca e E. Mehofer, «clusterCL: comprehensive support for multi-kernel data-parallel applications in heterogeneous asymmetric clusters,» *The Journal of Supercomputing*, mar. de 2020. DOI: 10.1007/s11227-020-03234-w.

[46] V. Raca e E. Mehofer, «Device-Sensitive Framework for Handling Heterogeneous Asymmetric Clusters Efficiently,» em *2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2015, pp. 178–185. DOI: 10.1109/SBAC-PAD.2015.15.

[47] T. K. Group, *OpenCL™ ICD Installation Guidelines*, https://www.khronos.org/registry/OpenCL/specs/2.2/html/OpenCL_ICD_Installation.html, (accessed Junho 22, 2019).

[48] S. Notes, *Chapter 10. User Datagram Protocol (UDP) and IP Fragmentation*, https://notes.shichao.io/tcpv1/ch10/, (accessed Abril 07, 2020).

[49] M. Pages, *GETTID*, http://man7.org/linux/man-pages/man2/gettid.2.html, (accessed Abril 07, 2020).

[50] *iperf3*, https://iperf.fr/, (accessed Abril 07, 2020).

[51] M. J. Christensen e T. Richter, *Achieveing reliable UDP transmission at 10 Gb/s using BSD socket for data acquisition systems*, jun. de 2017.

[52] L. Torvalds, *udp*, https://github.com/torvalds/linux/blob/master/net/ipv4/udp.c, (accessed Abril 07, 2019).

[53] packagecloud, *Monitoring and Tuning the Linux Networking Stack: Sending Data*, https://blog.packagecloud.io/eng/2017/02/06/monitoring-tuning-linux-networking-stack-sending-data/?fbclid=IwAR1TvrtD163uYp8j-K3IwbdIQY2lQfuQQu9aacHWRJC
1eU8OO595RG0, (accessed Março 17, 2019).

[54] S. COMMUNICATIONS, *OpenCL basics: Multiple OpenCL devices with the ICD.* https://streamhpc.com/blog/2015-08-14/opencl-basics-multiple-opencl-devices-with-the-icd/, (accessed Junho 17, 2019).

[55] Oblomov, *clinfo*, https://github.com/Oblomov/clinfo, (accessed Junho 22, 2019).

[56] T. K. Group, *OpenCL API 1.2 Reference Guide*, https://www.khronos.org/files/opencl-1-2-quick-reference-card.pdf, (accessed Março 30, 2020).

[57] K. Group, *OpenCL Benchmarks*, https://www.iwocl.org/resources/opencl-benchmarks/, (accessed Julho 22, 2019).

[58] cavazos-lab, *FinanceBench*, https://github.com/cavazos-lab/FinanceBench, (accessed Outubro 23, 2019).

[59] UoB-HPC, *BabelStream*, http://uob-hpc.github.io/BabelStream/, (accessed Outubro 23, 2019).

[60] NUCAR-DEV, *Hetero-Mark*, https://github.com/NUCAR-DEV/Hetero-Mark, (accessed Agosto 22, 2019).

[61] ekondis, *mixbench*, https://github.com/ekondis/mixbench, (accessed Setembro 23, 2019).

[62] vtsynergy, *OpenDwarfs*, https://github.com/vtsynergy/OpenDwarfs, (accessed Agosto 23, 2019).

[63] the Ohio State University, *polybench*, https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/, (accessed Outubro 23, 2019).

[64] nerdralph, *clmem*, https://github.com/nerdralph/cl-mem, (accessed Outubro 23, 2019).

[65] J.-F. '. Romang, *luxmark*, http://www.luxmark.info/, (accessed Setembro 23, 2019).

[66] fakenmc, *cf4ocl*, https://github.com/fakenmc/cf4ocl, (accessed Setembro 23, 2019).

[67] D. Bucciarelli, *juliagou*, http://davibu.interfree.it/opencl/juliagpu/juliaGPU.html, (accessed Setembro 23, 2019).

[68] NatTuck, *MandelGPU-v1.3*, https://github.com/NatTuck/cakemark/tree/master/benchmarks/mandelbrot/MandelGPU-v1.3, (accessed Setembro 23, 2019).

[69] Knio, *SmallptGPU*, https://github.com/Knio/SmallptGPU, (accessed Setembro 23, 2019).

[70] wolfviking0, *mandelbulbGPU-v1.0*, https://github.com/wolfviking0/webcl-davibu/tree/master/mandelbulbGPU-v1.0, (accessed Setembro 23, 2019).

[71] krrishnarraj, *clpeak*, https://github.com/krrishnarraj/clpeak, (accessed Outubro 23, 2019).

[72] yuhc, *rodinia*, https://github.com/yuhc/gpu-rodinia, (accessed Outubro 23, 2019).

[73] abduld, *Parboil*, https://github.com/abduld/Parboil, (accessed Setembro 23, 2019).

[74] hashcat, *hashCat*, https://hashcat.net/hashcat/, (accessed Outubro 23, 2019).

[75] K. Rupp, *ViennaCL*, http://viennacl.sourceforge.net/, (accessed Setembro 23, 2019).

[76] investopedial, *Black Scholes Model*, https://www.investopedia.com/terms/b/blackscholes.asp, (accessed Outubro 23, 2019).

[77] ——, *Monte Carlo Simulation Definition*, https://www.investopedia.com/terms/m/montecarlosimulation.asp, (accessed Outubro 23, 2019).

[78] oVirt. (). «oVirt,» URL: http://www.ovirt.org (acedido em 2020).

[79] K. Tausche, M. Plauth e A. Polze, *dOpenCL – Evaluation of an API-Forwarding Implementation*, nov. de 2016. DOI: 10.13140/RG.2.2.16598.24641.

[80] Jake, *Hashcat Tutorial – The basics of cracking passwords with hashcat*, https://laconicwolf.com/2018/09/29/hashcat-tutorial-the-basics-of-cracking-passwords-with-hashcat/, (accessed Outubro 3, 2020).

[81] IPB. (). «source code of a distributed Matrix Multiplication:» URL: http://www.ipb.pt/~rufino/clopencl/matrix-v13-split-v1_2012-11-10.tgz (acedido em 2020).

[82] C. Kohlhoff. (). «Boost.Asio,» URL: https://www.boost.org/doc/libs/1_74_0/doc/html/boost_asio.html (acedido em 2020).

[83] Khronos Group. (). «Khronos Group Releases OpenCL 3.0,» URL: https://www.khronos.org/news/press/khronos-group-releases-opencl-3.0 (acedido em 2020).

Apêndice A

Publicação Científica

Extending heterogeneous applications to remote co-processors with rOpenCL

Rui Alves
Instituto Politécnico de Bragança,
Campus de Santa Apolónia,
5300-253 Bragança, Portugal
rui.alves@ipb.pt

José Rufino
Research Centre in Digitalization and Intelligent Robotics (CeDRI),
Instituto Politécnico de Bragança,
Campus de Santa Apolónia,
5300-253 Bragança, Portugal
rufino@ipb.pt

***Abstract*—In heterogeneous computing systems, general purpose CPUs are coupled with co-processors of different architectures, like GPUs and FPGAs. Applications may take advantage of this heterogeneous device ensemble to accelerate execution. However, developing heterogeneous applications requires specific programming models, under which applications unfold into code components targeting different computing devices. OpenCL is one of the main programming models for heterogeneous applications, set apart from others due to its openness, vendor independence and support for different co-processors.**

In the original OpenCL application model, a heterogeneous application starts in a certain host node, and then resorts to the local co-processors attached to that host. Therefore, co-processors at other nodes, networked with the host node, are inaccessible and cannot be used to accelerate the application. rOpenCL (remote OpenCL) overcomes this limitation for a significant set of the OpenCL 1.2 API, offering OpenCL applications transparent access to remote devices through a TPC/IP based network. This paper presents the architecture and the most relevant implementation details of rOpenCL, together with the results of a preliminary set of reference benchmarks. These prove the stability of the current prototype and show that, in many scenarios, the network overhead is smaller than expected.

***Index Terms*—OpenCL, heterogeneous computing, API forwarding, remote execution, parallel and distributed computing**

I. Introduction

The last decade saw the emergence of *heterogeneous* computing systems, where different architectures co-exist, in addition to the main architecture embodied by the familiar general purpose CPU. In such systems, traditional multi-core CPUs are coupled with auxiliary co-processor devices, like Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), and Digital Signal Processors (DSPs). These devices are exploitable beyond their primary original goal (i.e., graphics/signal processing), in order to accelerate the execution of computationally demanding applications [1], [2].

For heterogeneous systems to be efficiently exploited, applications must be developed using special programming models. With low-level programming approaches, like the vendor-specific NVIDIA CUDA Driver API [3] or the OpenCL open standard [4], heterogeneous applications are non-single source, unfolding explicitly into different code components per architecture. These components are written in C (with some extensions) or in C++ (using the bindings provided), and target separately the co-processor devices and the host system where devices are attached to; typically, the host-side code creates the necessary data structures to interact with the devices and triggers the relevant operations (data exchanges with / issuing code to / synchronization with the devices); sometimes, host code solves parts of the problem, in parallel with the devices.

Another common denominator to CUDA and OpenCL is that their original application model is intra-node centric, that is, an heterogeneous application launched on a certain host can only exploit the co-processor devices directly attached to that host. This necessarily implies a limited number of locally usable co-processors, due to physical and logical constraints (power, cooling, chassis, maximum number of devices supported by the drivers and the system BIOS, etc.).

However, scaling-out execution of heterogeneous applications to co-processors in multiple nodes has been possible, from some time now, with NVIDIA GPUDirect-RDMA [5] or AMD ROCn-RDMA [6], that rely on inter-node GPU communication over Infiniband. Basically, Infiniband RDMA-enabled network cards have direct access (via the PCIe bus) to GPU memory, allowing for data exchanges between GPU memory in different nodes, bypassing the main memory subsystem. At a higher-level, these RDMA-based approaches, namely NVIDIA GPUDirect-RDMA, have been taken advantage of by popular MPI-compliant [7] message passing libraries that can be combined with CUDA in developing distributed parallel hybrid applications [8]; in essence, with CUDA-aware MPI implementations, the MPI library can send and receive GPU buffers directly, thus optimizing data exchanges (notably, opposed to the availability of several CUDA-aware MPI implementations, there's currently no OpenCL equivalent).

An alternative to exploit inter-node multi-accelerator configurations, while being able to use the original programming models of CUDA and OpenCL, is to provide, through its runtime systems, access to remote accelerator devices (that is, devices attached to other nodes than the one in which the host-side of the heterogeneous applications is launched), as if they were local, effectively creating a unified view of all the available accelerators. This even allows to start heterogeneous

This work has been supported by FCT – Fundação para a Ciência e Tecnologia within the Project Scope: UIDB/05757/2020.

applications in nodes where accelerators are absent, provided nodes with co-processors are networked with the starting node. Of course, issuing CUDA/OpenCL API calls and kernels for remote execution, or exchanging data with or between remote devices, is expected to exhibit less performance, compared to using only local co-processors. However, this may not always be the case: depending on the computing capabilities differential between local and remote devices, and also on the behavior of the heterogeneous application (namely its workload distribution algorithm), it may pay off to offload certain tasks to remote devices, despite the penalty introduced by the network. Moreover, if enough transparency is ensured, such transparency may be regarded as a key advantage to counteract the expected performance limitations (ideally, pre-compiled binary code should be transparently executed in the overall set of devices exposed to the application by the unified runtime, irregardless of the devices location). On the other hand, when developing new applications, or when the source code is available, if the unified runtime offers extensions that allow to determine whether a device is local or remote, such information (together with the results of previous benchmarks) may be used during the distribution of the workload.

This paper introduces rOpenCL (remote OpenCL), as an approach that tackles the previous scenario for the specific case of distributed OpenCL heterogeneous applications and TCP/IP networks where hosts support the traditional BSD sockets mechanism (meaning virtually any networked host). Thus, rOpenCL does not depend on niche network technologies (e.g., Infiniband), but takes advantage of them when available (provided they support TCP/IP). Also, rOpenCL (and similar approaches, including CUDA-related – see section IV), is not meant to compete with dense intra-node multi-accelerator systems, or multi-node multi-accelerator clusters built on specialized communication fabrics; it is a complementary multi-node multi-accelerator approach that may be deployed on commodity hardware and networks, or on high-performance infrastructures when available. Finally, in opting to extend OpenCL to a distributed environment, in detriment of alternatives like CUDA, the key factors considered were i) standard openness, ii) application portability, and iii) support for a wide range of co-processors (and not only GPUs).

The remainder of the paper is organized as follows: section II covers the software architecture of rOpenCL and important implementation details; section III provides results of a preliminary evaluation of the stability and performance of the rOpenCL prototype; section IV offers a brief survey of the most relevant approaches comparable to rOpenCL; section V concludes and defines directions for future work.

II. Architecture and Implementation

A. Overview

OpenCL is a specification that allows applications to take advantage of different architecture devices [4]. Such *heterogeneous applications* are composed of *host code* and a set of *kernels*. A kernel is a specific core function of an OpenCL application. Typically, a kernel is meant to be executed in a co-processor that is faster (regarding the specific kernel operations) than the processor(s) where the host code runs; exceptionally, a kernel may also execute on the same processor(s) as the host code, e.g., if no co-processors are available.

The OpenCL programming model is currently supported by four major implementations of the official specification, targeting different device types: NVIDIA GPU drivers [9]; ROCm [10] OpenCL runtime for AMD CPUs and GPUs; Intel OpenCL SDK [11] for Intel CPUs, GPUs and FPGAs; POCL [12], vendor-agnostic and mainly CPU-oriented, though with some GPU support. In OpenCL parlance, an implementation of the specification is a *platform*. A platform typically supports a limited set of devices, accessible by the platform runtime.

However, despite all the features offered by the OpenCL model for the building of heterogeneous applications, and the variety of platforms, OpenCL applications that run on top of conventional OpenCL platforms (those conforming to the official specification) can only use devices from the machine where they run. To circumvent this limitation, allowing network-reachable devices from other machines to be used, several alternatives were developed (see section IV for a brief survey), including rOpenCL, introduced in this paper.

The main feature that sets rOpenCL apart, when compared with similar approaches, is its transparency: OpenCL applications do not need to be recompiled in order to use rOpenCL to reach out and exploit remote platforms and their devices; this is because rOpenCL exposes remote platforms to applications as if they were local platforms, that is, rOpenCL is an *aggregator* of remote platforms; also, rOpenCL doesn't expose any local devices (such concerns only to the local conventional platforms); thus, with rOpenCL installed in its host node, an OpenCL application is able to use any mix of local platforms/devices and remote platforms/devices.

TABLE I
OpenCL 1.2 API coverage of rOpenCL.

Function Categories	Implemented	Not Implemented
OpenCL Platform Layer	13	0
OpenCL Runtime	4	0
Buffer Objects	13	3
Program Objects	11	0
Kernel and Event Objects	22	1
Image Objects	7	3
Sampler Objects	4	0
OpenGL Sharing	0	9
Direct3D 10 Sharing	0	6
DX9 Media Surface Sharing	0	4
Direct3D 11 Sharing	0	6

In its current stage, rOpenCL supports $\approx$ 71% of the OpenCL 1.2 API [13] – see Table I. Most image/graphic processing primitives (categories in *italic*) were left out, as usual in other distributed OpenCL implementations, where the main focus is also pure computing. Thus, considering only the computing-related primitives, rOpenCL coverage of the OpenCL 1.2 API is $\approx$ 94%, with the following functions yet to be supported: `clEnqueueCopyBuffer` (partially implemented; doesn't support copies between de-

vices of different machines); `clEnqueueCopyBufferRect`, `clEnqueueMigrateMemObjects` and `clEnqueueNativeKernel` (none support). Nevertheless, the primitives already supported by rOpenCL are enough to conduct a comprehensive range of OpenCL benchmarks (see section III). Moreover, by targeting OpenCL 1.2, rOpenCL is in line with the recently released OpenCL 3.0 specification [14], which only mandates full support for OpenCL 1.2 (though with more focus in C++).

B. Architecture

rOpenCL consists of two main components: a client driver (rOpenCL Driver) and a set of remote services (rOpenCL Services) responsible for executing OpenCL requests in the underlying devices. Figure 1 provides a representation of these components and their relations with the OpenCL environment.

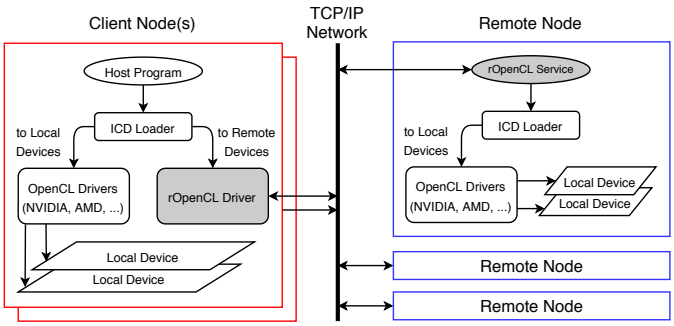


Fig. 1. rOpenCL architecture

At a host node, where an OpenCL application starts, the rOpenCL Driver is installed, like if it were another vendor driver; such drivers are known as installable client drivers (ICDs), and so rOpenCL has its own ICD. This way, the OpenCL runtime (the ICD Loader), is able to locate the ICD for rOpenCL and redirect OpenCL calls to it. In a Linux system, where rOpenCL was developed, this means there is a `/etc/OpenCL/vendors/rOpenCL.icd` file containing the path to an rOpenCL shared library where OpenCL calls are redirected to. Thus, by providing its own ICD, rOpenCL offers OpenCL applications full transparent access to the remote platforms and devices beneath the rOpenCL Services.

An OpenCL application may, however, want to know about the location of a specific remote OpenCL platform (and corresponding devices). This knowledge may be used, for instance, to implement client-side load balancing mechanisms, where an OpenCL application does not blindly assign requests to remote platforms that may happen to co-exist in the same remote node (thus overloading its devices); instead, it may decide to spread the requests among platforms in different nodes, by following a round-robin distribution, or other algorithms. rOpenCL makes this possible by extending the OpenCL `clGetPlatformInfo` primitive to support the new value `CL_PLATFORM_IPADDR` for its parameter `param_name`.

Another important architectural consideration is how rOpenCL manages OpenCL *contexts* in a distributed environment. A context is used by the OpenCL runtime to manage several interrelated objects (like command-queues – used to submit commands to devices –, memory buffers, program and kernel objects), and also to execute kernels on the devices specified for that context; these devices may even be of different types (e.g., CPUs and GPUs); however, they all must belong to the same OpenCL platform; in turn, an OpenCL platform object is, per the OpenCL standard, specific to a certain node, and rOpenCL complies with this; as such, rOpenCL doesn't replicate contexts (and subordinated objects) among different nodes; avoiding the burden of replica management simplifies the implementation; however, it prevents the creation of inter-node contexts, based on virtual platforms encompassing devices from different nodes; such objects, with a broader view of the nodes device set, could perform some form of automatic load-balancing; instead, with rOpenCL, that task is left to the programmer, by exploiting the `CL_PLATFORM_IPADDR` extension, as already stated.

Figure 1 also reveals that all network communication between the rOpenCL components happens via TCP/IP; such is because one of the goals for rOpenCL was network portability; this, coupled with the need for maximal network efficiency, lead to the choice of C-based BSD sockets as the programming framework for network communication; there are lower-level alternatives [15], but usually they do not support routing and so can only be used in the same LAN segment (thus limiting the remote device set that can be reached); moreover, the rOpenCL code base was designed from the very beginning to support the choice between TCP and UDP; currently, only the TCP code path is considered sufficiently robust, with the advantage of being usable with any network topology; UDP support is still experimental and meant to be used only on local network segments in order to minimize the probability of packet loss.

C. Connections Management

When an OpenCL application starts at the host node, OpenCL requests begin to be forwarded to the various OpenCL drivers configured, including the rOpenCL Driver. At minimum, this driver will be queried for platforms (and devices), in accordance with the usual workflow of OpenCL applications; after this inquiry phase, the application decides which platforms (and devices) to use, and further OpenCL requests will follow, directed to the drivers behind the platforms chosen.

To honor the platforms and device queries, as well as any other subsequent OpenCL request, the rOpenCL Driver will forward those requests to the appropriate rOpenCL Services. In order to make an efficient use of the communication layer, the driver minimizes the number of TCP connections and reuses them as much as possible. Every time the driver receives an OpenCL request, it checks the unique OS kernel thread ID (TID) of the requester process/thread, as well as a particular OpenCL parameter of the request (the *main* parameter – see section II-D) that, together with the TID, is mappable on the IP address of a rOpenCL service, using a Red-Black tree (RBT_1); it then uses the TID and the remote IP address to find out if it is necessary to create a new socket descriptor and establish a connection between the requester and the remote rOpenCL

Service; subsequent requests from the same process/thread to the same remote service will reuse the previously established connection; the inventory of TCP connections, by local TID and remote IP address, is hold on another Red-Black tree (RBT_2). To understand the importance of connection reuse, it was found, at an early stage of the development of rOpenCL, that establishing a TCP connection per each OpenCL call would consume around 30% of the execution time of the call (for OpenCL functions that involved small network transfers).

At each rOpenCL Service, POSIX threads are used. A frontend thread accepts connection requests from client threads (at host nodes), and assigns each new connection to a new worker thread. A worker thread will be responsible to forward the OpenCL requests received from a client thread, to the proper underlying OpenCL platform. When an OpenCL application at the host node ends, all open sockets descriptors will be closed by the operating system; this makes all remote worker threads that were previously paired with the application to unblock from the closed connections and to self terminate.

The startup of an OpenCL application in a host node also involves the discovery of rOpenCL Services. This is done simply, by reading a hostfile, that may be user-specific (default hostfile) or system-wide (in the absence of the former). The file contains the IP addresses and ports of the rOpenCL Services, as well as the local interface that the host node should use to communicate with those services (useful if the host node is a multi-homed system). The initial discovery process is conducted by a certain thread of the OpenCL application, that is usually (but not necessarily) the main thread; that discovery triggers the creation of an initial set of connections to each rOpenCL Service; at this initial stage, they are used to inquiry the remote platforms and devices; latter, those connections may be reused by the same client thread, if it needs to conduct new OpenCL transactions with the same rOpenCL Services.

D. Objects Management

OpenCL applications at the host node are supposed to deal with local pointers, referencing local memory areas where the various OpenCL objects, necessary to the application, are stored. However, when an OpenCL application makes use of the rOpenCL Driver, all local pointers must be mapped into remote pointers, for the same kind of objects, referencing memory zones of the nodes where rOpenCL Services run.

Every time an OpenCL primitive is called, the ICD Loader redirects that call to the proper driver. It does so by taking advantage of function pointers that the ICD Loader learned from the driver. That redirection submits and receives back certain OpenCL objects that must be well-formed (with an internal valid structure), once they may be passed along to other primitives by the ICD Loader. This means that, at the driver level, is risky to return fake pointers to the ICD Loader, once they may have and unpredictable effect. Fake pointers are used used by other distributed OpenCL implementations as equivalents to remote pointers; they can do so because their approach to the forwarding of OpenCL primitives to remote nodes is higher-level (wrapper-based), and not as lower level (driver level) as the one followed by rOpenCL in order to attain maximum transparency. Thus, instead of returning fake pointers, the rOpenCL Driver does indeed create real local OpenCL objects; however, it treats them as "hollow" objects, without any further use than misleading the ICD Loader. The real usefulness lies on the twin remote object that rOpenCL Services create for each local object. Once both objects are created, the rOpenCL Driver must register their equivalence. It does so in the RBT_1 Red-Black tree, that maps <TID, local pointer> keys into <remote pointer, IP address> values.

When the rOpenCL Driver receives a OpenCL request, one of its parameters (usually the 1st) is considered the *main* parameter, meaning its local address is already valid and registered in the RBT_1 tree; the local address of the main parameter, together with the requester TID, are used to search this tree, for the remote pointer and its IP address; local addresses of other parameters are also searched in the RBT_1 tree, for their remote addresses (some local addressees may already have been mapped into remotes, and others may not); the TID and IP address for the main parameter are used to search the RBT_2 tree for a previously open connection to the remote service; then, a message is sent to this service, carrying all necessary remote addresses; that message may result in the creation of new remote objects; the addresses of these objects are returned to the client thread; the twin local objects are then created and the new mappings registered in the RBT_1 tree.

E. Concurrency Management

At the client side, in the rOpenCL Driver, there's concurrent access to the RBT_1 and RBT_2 trees by different OS kernel threads; a MultipleReaderOneWriter (MROW) lock (`rw_semaphore`) is used, per tree, to ensure its consistency.

At the services side, there are several worker threads, each one dealing with a separate connection from a client thread. A worker thread unmarshalls the requests received from its client thread, triggers appropriate actions in the OpenCL layer, and replies accordingly to the client thread. Thus, at any time, there may be multiple worker threads issuing OpenCL calls. This rises the question of thread-safety. Per the OpenCL 1.2 specification [16], all OpenCL API calls are thread-safe except `clSetKernelArg` and even this function is not thread-safe only when called from multiple threads on the same `cl_kernel` object at the same time. Even though this is a corner case, it is dealt with, at cost of some overhead. Currently, a single MROW lock (a POSIX `pthread_rwlock_t` object), shared by all worker threads, serializes all calls to `clSetKernelArg`; an alternative, with less contention, planned for future work, is to register all `cl_kernel` objects in a tree-like structure (like the Red-Black trees used in the rOpenCL Driver), together with a MROW lock per object.

III. Preliminary Evaluation

This section provides results of a preliminary evaluation of rOpenCL. The tests conducted prove that rOpenCL is able to sustain the full execution of a set of well-know OpenCL

reference benchmarks, in various experimental conditions. Besides asserting the compliance of rOpenCL with the OpenCL standards, the tests also measure the overhead of the remote execution over local execution. One of the tests, using many devices, provides an initial insight into the scalability of rOpenCL. The tests selected are listed in Table II.

TABLE II
OPENCL BENCHMARKS USED TO EVALUATE ROPENCL

OpenCL Benchmark	Benchmark Profile		
	Memory	*Compute*	*Multi-Device*
BabelStream [17]	✓		
cl-mem [18]	✓		
clpeak [19]	✓	✓	
FinanceBench [20]		✓	
Hashcat [21]		✓	✓

The choice of this particular set of benchmarks is due to the following reasons: i) they stress different device subsystems (compute vs memory), ii) they are all open-source (some instrumentation code was necessary to automate the benchmarks), iii) they have been actively maintained in the last few years, iv) they run in Linux (the OS environment targeted by rOpenCL), v) at least one of them (Hashcat) is able to issue OpenCL calls to multiple devices in parallel.

A. Experimental Conditions

The tests were conducted between two nodes of an HPC cluster at CeDRI/IPB, each with the HW & SW of Table III.

TABLE III
SPECIFICATIONS OF EACH TEST NODE

CPUs	2 x AMD EPYC 7351 16-Core 2.4/2.9GHz
RAM	256 GB ECC DDR4 2666 MHz
Network	10Gbps Ethernet
GPUs	2 x NVIDIA RTX 2080 Ti
OS	Linux Ubuntu 18.04.3 LTS 64 bits
OpenCL	NVIDIA Driver 430.50

All benchmarks were repeated 5 times. The times presented in the charts of the next section are averages of the times measured in all 5 runs (the time of the 1st run was observed to be similar to the others). The execution overhead (deacceleration), in percentage, is given by $(\overline{T_r}/\overline{T_l} - 1) \times 100$, where $\overline{T_r}$ is a remote execution average time and $\overline{T_l}$ is its corresponding local execution average time. Also, to remove the start latency caused by the loading of the GPU driver at the beginning of each run, the NVIDIA Persistence Daemon [22] was configured on the two test nodes (the absence of this service would only affect the client-side of the benchmark – and would be only noticeably on short-duration benchmarks – once an rOpenCL Service forces the pre-load of the driver).

B. Results

The results for the BabelStream benchmark are shown in Figure 2. BabelStream provides a measure of what memory bandwidth performance can be attained when executing five kernels (copy, mul add, triad and dot). Although the benchmark is quick to execute, it is the one tested with the highest remote execution overhead, varying between 256% and 638%, depending on the sub-benchmarks, and with an overhead average of 353%. So, on average, BabelStream is 3,5 times slower when using a remote GPU via rOpenCL.

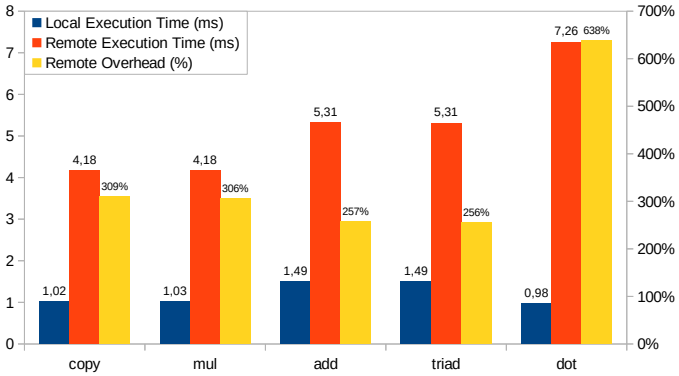


Fig. 2. BabelStream results (copy, mult, add, triad, dot).

cl-mem is a very simple benchmark with three different OpenCL kernels, to test the speed of sequential write, read and copy operations (of 128 GB of data, by default); these operations are executed in parallel, by groups of threads, in the device being tested; also, it does not require the previous transfer of the data to the device memory, once the data used in the test is generated on-the-fly; thus, cl-mem makes very little use of the network connection between the rOpenCL Driver in the client node and the rOpenCL Service in the remote node. This is clearly visible in the results presented in Figure 3.

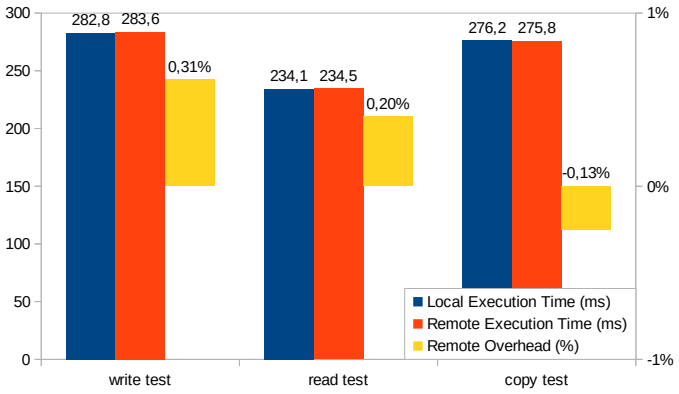


Fig. 3. cl-mem results (write, read and copy).

These results show very little differences between the local and the remote execution. And, in fact, for the copy kernel test, the remote execution turns out to be slightly faster. We have tried to uncover the reason for this, but that investigation was inconclusive; a possible explanation may lie in the fact that the nodes used for benchmarking have a NUMA architecture and the attachment of GPUs to the PCIe bus may be different.

Another benchmark conduced was clpeak. This is a simple hybrid benchmark, stressing both device memory and processing elements. It measure peak capabilities achieved using

vector operations and does not represent a real-world use case. Thus, we restrain from showing the peak memory bandwidth and compute power; instead, Figure 4 presents only the kernel launch latency; for the remote execution, it is interesting to see the impact introduced by the network on such a small operation; the overhead measured was modest: just an excess of 16%, compared to the latency of a local kernel launch.

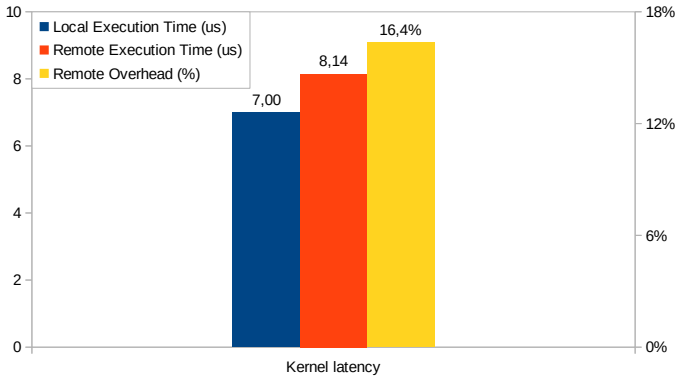


Fig. 4. clpeak results (kernel launch latency).

The last mono-device benchmark executed was FinanceBench, more compute focused. Even though four specific tests are available (Black-Scholes, Monte-Carlo, Bonds, Repo), all of which are financial applications, only two have an OpenCL version (Black-Scholes and Monte-Carlo), and so results are only presented for these two – see Figure 5. For one of the test (Black-Scholes) the remote overhead is minimal; for the other (Monte-Carlo), it is rather modest (21% more).

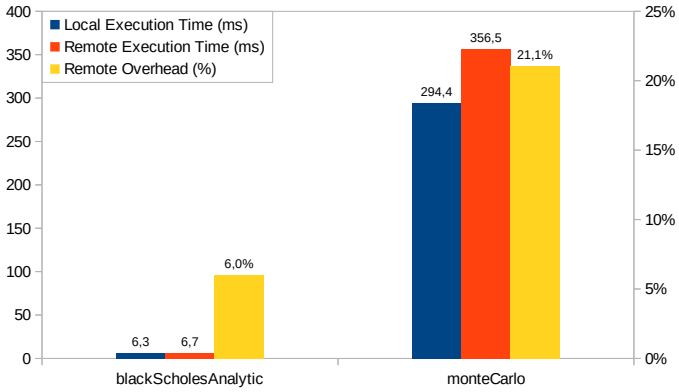


Fig. 5. FinanceBench results (Black-Scholes and Monte-Carlo).

The last benchmark results presented are for the Hashcat application, a well-known advanced password recovery tool [21]. It contains a built-in benchmark option – a single hash brute force attack; however, this benchmark involves very little data transfers between the host program and the devices, once the hash space is known a prior and is evenly divided by the used GPUs; instead, a dictionary attack with rules [23] (recovering 28 password from their MD5 hash) was chosen as the Hashcat benchmark. Moreover, once Hashcat is able to use multiple GPUs, this created the opportunity to put rOpenCL to the test with 2 GPUs (the maximum number of remote GPUs available in our test bed). Table IV shows all GPU combinations tested, and Figure 6 shows the respective execution times; depending on the specific combination of local and remote GPUs, these times are for pure local cracking, pure remote cracking, or include both alternatives.

TABLE IV
GPU COMBINATIONS FOR THE HASHCAT TEST.

Test Scenario	Local GPUs	Remote GPUs
T1	0	1
T2	0	2
T3	1	0
T4	1	1
T5	1	2
T6	2	0
T7	2	1
T8	2	2

The test results show that: i) when working only with remote GPUs, it certainly pays off to add an extra GPU (speedup is 494/249 = 1,98 $\approx$ 2); ii) the moment a local GPU is added, adding 1 remote GPU has little benefit (speedup is 249/229,8 = 1,08), and only by adding a 2nd remote GPU does the gain become noticeable (speedup is 249/171 = 1,45); iii) with 2 local GPUs, the benefit of 1 or 2 remote GPUs is again noticeable (speedup with 1 remote GPU is 112,8/93,6 = 1,20, and speedup with 2 remote GPUs is 112,8/81,8 = 1,38). The general conclusion is that the more GPUs, the better, and having local GPUs together with remote GPUs is always beneficial. Moreover, rOpenCL shows to be a viable tool to improve the performance of real-world OpenCL applications.

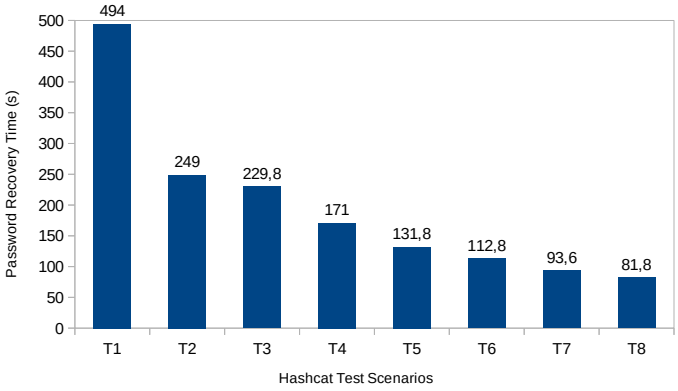


Fig. 6. Hashcat results (dictionary attack with rules on 25 MD5 hashes).

IV. Related Work

Vendors released the first public OpenCL implementations in 2009. Soon after, distributed variants started to be developed [24], [25]. This section surveys the most relevant, and includes comparisons with correlated rOpenCL features.

dOpenCL [26] is probably the most thorough distributed OpenCL implementation to date, and to which a comparison with rOpenCL is more meaningful. Both approaches build on a client driver at the host system and a set of services deployed

in the nodes of the distributed system with co-processors. Transparency is also a primary goal that both share, with the intent of allowing to run existing OpenCL applications in a heterogeneous distributed environment without any modifications. There are, however, important differences.

dOpenCL is represented by one dOpenCL platform object that gathers all the nodes and respective devices; this allows for a device management mechanism with transparent load balancing and exclusive access to specific remote devices by client applications; a single global unified platform implies, however, the need to maintain consistency of several shared distributed objects, like OpenCL contexts built on devices from different nodes (and such need extends to objects subordinated to contexts, like command queues and buffers). In turn, rOpenCL exposes the remote platforms and devices separately; so, it is up to the client application to choose which ones to use and to do its own load balancing; also, because different applications may share the same remote devices, rOpenCL services must perform some basic concurrency control in the particular situations where thread-safety is not ensured; moreover, because OpenCL contexts are restricted to use devices from the same platform, and platforms are separate per node, there's no need in rOpenCL to do replica management as in dOpenCL; this implies, though, that an application wanting to exploit devices from different nodes will have to use as many contexts at minimum (there may be more than one platform per node), while in dOpenCL one context suffices.

dOpenCL doesn't support the ICD loader mechanism; this implies that before running an OpenCL application on top of dOpenCL, the OpenCL system loader must be replaced by the dOpenCL loader, or the dOpenCL ICD must be explicitly preloaded; in this regard, rOpenCL is more transparent, once its ICD is fully integrated with the OpenCL system loader. In both, the client driver only exposes remote devices (unless a service to expose local nods is also installed in the host node).

Finally, in dOpenCL the client driver and services communicate using a Generic Communication Framework (GFC), part of a Real-Time Framework (RTF) developed for high-performance communication in distributed real-time applications, namely massively multi-player online computer games [27]. Though GFC supports TCP and UDP communication, dOpenCL opts for TCP. Likewise, rOpenCL also uses TCP communication, but via BSD sockets, a more portable and lighter approach (though arguably less feature-rich) than GFC.

Another approach to a distributed OpenCL implementation is clOpenCL (*cluster* OpenCL) [28]. It's architecture is similar to rOpenCL's, separately exposing the remote platforms, but it was not as transparent: legacy OpenCL applications needn't to be modified, but they needed to be recompiled in order to be linked with a library of wrapper functions that intercepted the OpenCL API calls. Moreover, in an effort to increase the performance of message passing between client applications and remote services, it relied on the low-level Open-MX library [15], directly above the Ethernet layer, thus restricting its use to non-routed LAN scenarios (e.g., HPC clusters).

Also tailored to heterogeneous cluster environments, SnuCL [29] is another distributed OpenCL framework that provides a single unified view of all the cluster compute devices and, at the same time, supports different partitions of the cluster device set: a compute device may be a GPU, or any sub-set of the CPU cores of a cluster node (including the host node, where the host program is launched); thus, other compute devices besides CPUs and GPUs are not supported. SnuCL relies on its own implementation of OpenCL: it uses a OpenCL-C-to-CUDA-C translator for kernels that target GPUs (NVIDIA only) and an OpenCL-C-to-C translator for kernels that target CPU devices; these translations are performed in the host node, the translated code is sent to compute nodes and there it's compiled with the native compiler for each compute device; thus, SnuCL requires the original OpenCL source code of applications to be available, although no modifications to it. Notably, SnuCL is not an API-forwarding approach: the OpenCL primitives of an OpenCL application are executed in the host node up to the point in which commands are enqued to command-queues; commands are then schedule across compute devices in the cluster. Communication between the components of the SnuCL framework is done via MPI and OpenCL is enriched with collective communication operations between buffers. For large scale clusters, the centralized task scheduling model of SnuCL degrades performance; SnuCL-D [30], a recent evolution of SnuCL, tackles this problem.

API-forwarding and source-to-source translation (thus requiring the original source code of OpenCL applications) were combined in Hybrid OpenCL [24], one of the first distributed OpenCL proposals. In Hybrid OpenCL, OpenCL code is translated to readable C code that uses embedded IA-32 SSE functions, thus targeting only multi-core x86 devices and excluding GPUs or other co-processors. At the host node, OpenCL applications link with the Hybrid OpenCL runtime, a version of the FOXC (Fixstars OpenCL Cross Compiler) OpenCL runtime, modified to include a network layer (though the communication protocol used is not disclosed). The Hybrid OpenCL runtime is able to submit request to the underlying OpenCL x86 devices or to forward them to remote nodes. Each remote node runs a networked bridge service on top of a local OpenCL runtime (that may be other than FOXC, as long as it allows to use the local CPUs as devices), to which it relays requests coming from the host node. Hybrid OpenCL offers to client applications a single OpenCL platform with all the x86 devices of the involved nodes, and applications may use any combination of those devices.

The abstraction of a global OpenCL platform combining all compute devices (CPUs, GPUs and other accelerators) available in a set of networked nodes is also provided by the VirtualCL (VCL) cluster platform [31], originally a MOSIX [32] layer. With VCL, most OpenCL applications run unmodified, starting in a single node, and take advantage, transparently, of the cluster device set; applications may create OpenCL contexts that mix devices from different nodes, or are restricted to intra-node devices (the default); the real location of the devices in the cluster is hidden from applications, but environment variables allow to tune the device allocation policies. VCL

includes i) a front-end thread-safe wrapper library to which OpenCL applications must link (thus requiring its source code to be available), ii) a broker daemon for cluster resource monitoring, reporting and allocation, that runs on the node where the application starts, and iii) a back-end daemon, per each cluster node with compute devices, that relays OpenCL requests from the client applications to local vendor-specific OpenCL libraries (with each device being exclusively locked by one application at a time). Communication between the cluster nodes involved standard TCP/IP sockets. VCL is still used and maintained (but only binaries are available).

V. Conclusions

This paper introduced rOpenCL, a novel implementation of an API-forwarding layer that allows OpenCL applications to take advantage of OpenCL devices accessible via a TCP/IP network. This puts the co-processor devices of virtually any networked system (even if routing is involved) within the reach of an OpenCL application. Moreover, the way rOpenCL was architected and implemented allows for pre-compiled OpenCL applications to take immediate advantage of it; this level of transparency, achieved by the perfect integration of the rOpenCL Driver in the OpenCL runtime, is a feature that we believe will make rOpenCL an attractive choice for scenarios where a distributed OpenCL layer is necessary. The stability of the current implementation, and its usefulness for a real-world application, were also shown by the benchmark results presented (a subset of the benchmarks already conducted).

In the future, we will enhance rOpenCL in several areas, including: further optimizing the network transfers, making UDP a robust alternative for local networks, and increasing the OpenCL 1.2 API coverage. We also intend to do tests with more computing nodes, thus supporting more simultaneous clients and rOpenCL services, with the goal of assessing the scalability of the later. Finally, a comparison with previous distributed OpenCL implementations is also planned (this should be viable at least with clOpenCL, dOpenCL and VCL).

References

[1] A. Cano, "A survey on graphic processing unit computing for large-scale data mining," *WIREs Data Mining and Knowledge Discovery*, vol. 8, no. 1, p. e1232, 2018.

[2] H. Wang, H. Peng, Y. Chang, and D. Liang, "A survey of gpu-based acceleration techniques in mri reconstructions," *Quantitative Imaging in Medicine and Surgery*, vol. 8, pp. 196–208, March 2018.

[3] NVIDIA. CUDA Driver API :: CUDA Toolkit Documentation. [Online]. Available: https://docs.nvidia.com/cuda/cuda-driver-api/index.html

[4] The Khronos Group. OpenCL Overview. [Online]. Available: https://www.khronos.org/opencl/

[5] NVIDIA. Developing a Linux Kernel Module using GPUDirect RDMA. [Online]. Available: https://docs.nvidia.com/cuda/pdf/GPUDirect_RDMA.pdf

[6] Advanced Micro Devices. ROCnRDMA: ROCm Driver RDMA Peer to Peer Support. [Online]. Available: https://github.com/rocmarchive/ROCnRDMA

[7] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard - Version 3.1. [Online]. Available: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[8] NVIDIA. An Introduction to CUDA-Aware MPI. [Online]. Available: https://devblogs.nvidia.com/introduction-cuda-aware-mpi/

[9] NVIDIA. OpenCL NVIDIA Developer. [Online]. Available: https://developer.nvidia.com/opencl

[10] Advanced Micro Devices. AMD ROCm Open Ecosystem. [Online]. Available: https://www.amd.com/en/graphics/servers-solutions-rocm

[11] Intel. Intel SDK for OpenCL Applications. [Online]. Available: https://software.intel.com/content/www/us/en/develop/tools/opencl-sdk.html

[12] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, "pocl: A performance-portable opencl implementation," *Int. Journal of Parallel Programming*, vol. 43, no. 5, pp. 752–785, 2015. [Online]. Available: http://dx.doi.org/10.1007/s10766-014-0320-y

[13] The Khronos Group. OpenCL 1.2 Overview. [Online]. Available: https://www.khronos.org/files/opencl-1-2-quick-reference-card.pdf

[14] The Khronos Group. Khronos Group Releases OpenCL 3.0. [Online]. Available: https://www.khronos.org/news/press/khronos-group-releases-opencl-3.0

[15] B. Goglin, "Design and implementation of open-mx: High-performance message passing over generic ethernet hardware," in *2008 IEEE Int. Symposium on Parallel & Distributed Processing*, April 2008, pp. 1–7.

[16] The Khronos Group. The OpenCL Specification 1.2. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf

[17] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, "Evaluating attainable memory bandwidth of parallel programming models via BabelStream," *International Journal of Computational Science and Engineering*, vol. 17, no. 3, pp. 247–262, 10 2018.

[18] nerdralph. CL-Mem. [Online]. Available: https://github.com/nerdralph/cl-mem

[19] krrishnarraj. CL-Peak. [Online]. Available: https://github.com/krrishnarraj/clpeak

[20] S. Grauer-Gray, W. Killian, R. Searles, and J. Cavazos, "Accelerating financial applications on the gpu," in *6th Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU-6)*, 03 2013, pp. 127–136. [Online]. Available: https://github.com/cavazos-lab/FinanceBench

[21] hashcat. hashcat overview. [Online]. Available: https://hashcat.net/hashcat/

[22] NVIDIA. Driver Persistence. [Online]. Available: https://docs.nvidia.com/deploy/driver-persistence/index.html/

[23] Jake Miller. Hashcat Tutorial – The basics of cracking passwords with hashcat. [Online]. Available: https://laconicwolf.com/2018/09/29/hashcat-tutorial-the-basics-of-cracking-passwords-with-hashcat/

[24] R. Aoki, S. Oikawa, R. Tsuchiyama, and T. Nakamura, "Hybrid OpenCL: Connecting Different OpenCL Implementations over Network," in *2010 10th IEEE International Conference on Computer and Information Technology*, June 2010, pp. 2729–2735.

[25] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh, "A package for OpenCL based heterogeneous computing on clusters with many GPU devices," in *2010 IEEE International Conference On Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS)*, October 2010, pp. 1–7.

[26] P. Kegel, M. Steuwer, and S. Gorlatch, "dopencl: Towards uniform programming of distributed heterogeneous multi-/many-core systems," *Journal of Parallel and Distributed Computing*, vol. 73, p. 1639–1648, December 2013.

[27] F. Glinka, A. Ploss, J. Müller-Iden, and S. Gorlatch, "Rtf: A real-time framework for developing scalable multiplayer online games," in *Proceedings of the 6th ACM SIGCOMM Workshop on Network and System Support for Games, NetGames '07*, January 2007, pp. 81–86.

[28] A. Alves, J. Rufino, A. Pina, and L. P. Santos, "clOpenCL: Supporting Distributed Heterogeneous Computing in HPC Clusters," in *Proceedings of the 18th Int. Conference on Parallel Processing Workshops*, ser. Euro-Par'12. Berlin, Heidelberg: Springer-Verlag, 2012, p. 112–122.

[29] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12, New York, NY, USA, June 2012, pp. 341–352.

[30] J. Kim, G. Jo, J. Jung, J. Kim, and J. Lee, "A distributed OpenCL framework using redundant computation and data replication," in *PLDI '16: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2016, pp. 553–569.

[31] A. Barak and A. Shiloh, "The MOSIX Virtual OpenCL (VCL) Cluster Platform," in *Proceedings of the Intel European Research and Innovation Conference*, October 2011, p. 196.

[32] A. Barak and A. Shiloh. The MOSIX Cluster Management System for Distributed Computing on Linux Clusters and Multi-Cluster Private Clouds. [Online]. Available: http://www.mosix.cs.huji.ac.il/pub/MOSIX_wp.pdf