Universidade do Estado do Rio de Janeiro
Centro de Tecnologia e Ciências
Faculdade de Engenharia

André Luiz Rocha Tupinambá

DistributedCL: *middleware* de processamento distribuído em GPU com interface da API OpenCL

Rio de Janeiro
2013

André Luiz Rocha Tupinambá

DistributedCL: *middleware* de processamento distribuído em GPU com interface da API OpenCL

Dissertação apresentada, como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Engenharia Eletrônica, da Universidade do Estado do Rio de Janeiro. Área de concentração: Redes de Telecomunicações.

Orientador: Prof. Dr. Alexandre Sztajnberg

Rio de Janeiro

2013

CATALOGAÇÃO NA FONTE

UERJ / REDE SIRIUS / BIBLIOTECA CTC/B

T928 TUPINAMBÁ, André Luiz Rocha.

DistributedCL: middleware de processamento distribuído em GPU com interface da API OpenCL / André Luiz Rocha Tupinambá. - 2013.

89fl.: il.

Orientador: Alexandre Sztajnberg.

Dissertação (Mestrado) – Universidade do Estado do Rio de Janeiro, Faculdade de Engenharia.

1. Engenharia Eletrônica. 2. Processamento eletrônico de dados - Processamento distribuído – Dissertações. I. Sztajnberg Alexandre. II. Universidade do Estado do Rio de Janeiro. III. Título.

CDU
004.72.057.4

Assinatura Data

André Luiz Rocha Tupinambá

DistributedCL: middleware de processamento distribuído em GPU com interface da API OpenCL

Dissertação apresentada, como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Engenharia Eletrônica, da Universidade do Estado do Rio de Janeiro. Área de concentração: Redes de Telecomunicações.

Aprovada em 10 de julho de 2013

Banca examinadora:

Prof. Dr. Alexandre Sztajnberg (Orientador)
Instituto de Matemática e Estatística – UERJ

Profª. Drª. Noemi de La Rocque Rodriguez
Departamento de Informática – PUC-Rio

Prof. PhD. Eugene Francis Vinod Rebello
Instituto de Computação – UFF

Prof. PhD. Felipe Maia Galvão França
COPPE – UFRJ

Rio de Janeiro

2013

DEDICATÓRIA

Suddenly – you were gone
From all the lives you left your mark upon

Neil Peart

À saudosa memória de Luis de Bonis e Margarida Liguori.

AGRADECIMENTOS

Realizar um trabalho de pós-graduação exige não só esforço, mas também a ajuda de uma série de pessoas que acabam envolvidas. Assim, gostaria de deixar meus agradecimentos aos professores, colegas de trabalho e amigos que, sem eles, este trabalho não seria possível.

Primeiramente, ao Prof. Alexandre Sztajnberg, que foi meu professor na graduação, supervisor de estágio na DINFO/UERJ e agora orientador de mestrado. Tenho que agradecer não só pela ótima orientação, mas também boa parte da minha formação como profissional.

Agradeço também ao Jorge Luiz Sued, amigo e antigo gerente na Petrobras, que me incentivou a voltar à vida acadêmica, dizendo literalmente "você devia fazer um mestrado", e possibilitou que eu realizasse este trabalho. Da mesma forma agradeço ao Sebastião César, meu atual gerente na Petrobras, pelo incentivo e pela compreensão das necessidades de horário que um mestrado necessita, principalmente nos últimos meses.

Devo agradecimentos também ao LNCC e à Faperj pelo laboratório de GPUs do LabIME, disponibilizado através do projeto Cooperação entre as Pós-Graduações de Computação Científica LNCC-UERJ / Faperj. Neste laboratório realizei as avaliações de desempenho deste trabalho, que enriqueceram muito esta dissertação.

Ao amigo Rafael Gustavo devo agradecer às longas discussões sobre este trabalho, desde a ideia inicial, até a definição da arquitetura e alternativas na construção do código.

À Fernanda Rabelo, minha amiga e namorada, pela compreensão e paciência para aguentar os dias de mau humor e cansaço.

Meus agradecimentos finais vão para meus avós Luis de Bonis e Margarida Liguori que se alegraram e acompanharam a minha volta à vida acadêmica, mas não vão estar comigo neste momento final.

RESUMO

TUPINAMBÁ, André Luiz Rocha. *DistributedCL: middleware de processamento distribuído em GPU com interface da API OpenCL*. 2013. 89f. Dissertação (Mestrado em Engenharia Eletrônica) – Faculdade de Engenharia, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, 2013.

Este trabalho apresenta a proposta de um *middleware*, chamado DistributedCL, que torna transparente o processamento paralelo em GPUs distribuídas. Com o suporte do *middleware* DistributedCL uma aplicação, preparada para utilizar a API OpenCL, pode executar de forma distribuída, utilizando GPUs remotas, de forma transparente e sem necessidade de alteração ou nova compilação do seu código. A arquitetura proposta para o *middleware* DistributedCL é modular, com camadas bem definidas e um protótipo foi construído de acordo com a arquitetura, onde foram empregados vários pontos de otimização, incluindo o envio de dados em lotes, comunicação assíncrona via rede e chamada assíncrona da API OpenCL. O protótipo do *middleware* DistributedCL foi avaliado com o uso de *benchmarks* disponíveis e também foi desenvolvido o *benchmark* CLBench, para avaliação de acordo com a quantidade dos dados. O desempenho do protótipo se mostrou bom, superior às propostas semelhantes, tendo alguns resultados próximos do ideal, sendo o tamanho dos dados para transmissão através da rede o maior fator limitante.

Palavras-chave: OpenCL, GPGPU, GPU, *middleware*, processamento distribuído.

ABSTRACT

TUPINAMBÁ, André Luiz Rocha. *DistributedCL: middleware de processamento distribuído em GPU com interface da API OpenCL*. 2013. 89f. Dissertação (Mestrado em Engenharia Eletrônica) – Faculdade de Engenharia, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, 2013.

This work proposes a middleware, called DistributedCL, which makes parallel processing on distributed GPUs transparent. With DistributedCL middleware support, an OpenCL enabled application can run in a distributed manner, using remote GPUs, transparently and without alteration to the code or recompilation. The proposed architecture for the DistributedCL middleware is modular, with well-defined layers. A prototype was built according to the architecture, into which were introduced multiple optimization features, including batch data transfer, asynchronous network communication and asynchronous OpenCL API invocation. The prototype was evaluated using available benchmarks and a specific benchmark, the CLBench, was developed to facilitate evaluations according to the amount of processed data. The prototype presented good performance, higher compared to similar proposals. The size of data for transmission over the network showed to be the biggest limiting factor.

Keywords: OpenCL, GPGPU, GPU, middleware, distributed systems.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
FIFO	First In First Out
GPGPU	General-Purpose computation on Graphics Processing Units
GPU	Graphics Processing Unit
ICD	Installable Client Driver
IDL	Interface Description Language
IP	Internet Protocol
MPI	Message Passing Interface
OpenCL	Open Computing Language
OpenGL	Open Graphics Library
RPC	Remote Procedure Call
SHOC	Scalable HeterOgeneous Computing benchmark suite
SIMD	Single Instruction, Multiple Data
TCP	Transmission Control Protocol

SUMÁRIO

INTRODUÇÃO

Com o intuito de realizar pesquisas com reconstrução tomográfica, o grupo ASTRA montou com sucesso um computador com oito GPUs, que chegou à capacidade de processamento equivalente ao cluster de processamento utilizado pelo grupo (ASTRA TEAM, 2008).

Para utilizar um algoritmo mais complexo de reconstrução, o mesmo grupo montou com sucesso um segundo computador, desta vez com treze GPUs (ASTRA TEAM, 2009). No entanto, o funcionamento correto deste equipamento só foi possível após alterações feitas na BIOS do computador e no *kernel* do Linux, pois os computadores e os sistemas operacionais normalmente não são preparados para esta quantidade de GPUs simultâneas.

Ainda que nesse caso tais problemas tenham sido resolvidos, existe um limite físico de quantas placas podem ser adaptadas em um único computador. Por exemplo, na placa mãe ASUS P6T7 WS Supercomputer, utilizada pelo projeto, existe espaço para no máximo sete placas de vídeo, cada uma contendo no máximo duas GPUs. Torna-se então necessária outra abordagem para utilizar o poder de processamento de um grande conjunto de GPUs.

Uma alternativa é utilizar as GPUs disponíveis nos computadores interligados através de uma rede e, para isso, o natural seria recorrer a mecanismos de programação distribuída conhecidos, tais como o MPI. Porém, neste caso, o desenvolvedor terá que conciliar à distribuição da aplicação através do framework escolhido e o desenvolvimento em GPU, aumentando a complexidade da aplicação, tendo que se coordenar a utilização de ambas às tecnologias.

Este trabalho apresenta a proposta de um *middleware*, chamado DistributedCL, que torna transparente o processamento em GPUs distribuídas. Com o suporte deste *middleware*, uma aplicação preparada para utilizar a API OpenCL pode executar de forma distribuída, explorando o uso de GPUs remotas, de forma transparente e sem necessidade de alteração ou nova compilação do seu código. Com o uso do *middleware*, o desenvolvedor de uma aplicação precisa tratar somente do modelo de programação para GPU usando OpenCL, sem a necessidade de tratar as questões de distribuição.

O *middleware* DistributedCL é definido com uma arquitetura modular, com camadas bem definidas e aproveita o comportamento das funções da API OpenCL para melhorar seu desempenho. Um protótipo foi construído seguindo a arquitetura proposta com foco na otimização das diversas camadas, com destaque no tratamento assíncrono das chamadas da API OpenCL, transmissão de dados em lote e paralelismo entre a transferência dados e o

processamento na GPU.

A avaliação do protótipo do *middleware* DistributedCL foi feita com o uso de diferentes aplicações *benchmark* com suporte ao OpenCL disponíveis. O objetivo foi verificar a transparência do uso do protótipo pelas aplicações, analisar o desempenho com diferentes processamentos e volume de dados e também realizar testes comparativos com propostas similares.

Para a avaliação também foi desenvolvida uma ferramenta de análise de desempenho para utilizar múltiplas GPUs de forma concorrente e com diferentes volumes de dados. Esta ferramenta, chamada CLBench, foi desenvolvida de forma independente do *middleware*, acessando diretamente a API OpenCL e pode ser utilizada para avaliação de quaisquer dispositivos OpenCL.

Organização do Texto

O presente texto está organizado em seis capítulos, apresentando o trabalho desenvolvido. O Capítulo 1 traz uma introdução sobre o processamento genérico em GPUs (GPGPU), apresenta o padrão OpenCL, trata as necessidades de um *middleware* de processamento distribuído em GPU e discute as características da plataforma OpenCL. O Capítulo 2 discute a possibilidade de uso dos frameworks RPC e MPI como infraestrutura para um *middleware* de processamento em GPU e também trabalhos relacionados. O Capítulo 3 apresenta a arquitetura do *middleware* DistributedCL, as características de suas camadas internas e o protocolo de rede. O Capítulo 4 apresenta o protótipo desenvolvido, a estrutura do seu código, a sua instalação, configuração e as otimizações introduzidas. O Capítulo 5 apresenta a avaliação de desempenho do protótipo e a comparação com propostas similares utilizando três *benchmarks* disponíveis para OpenCL e também com uma aplicação de avaliação desenvolvida utilizando a API OpenCL. Finalmente, no Capítulo 6, são apresentadas as considerações finais e propostas para trabalhos futuros.

1 PROCESSAMENTO EM GPU

Até a popularização das placas aceleradoras 3D, pouco antes do ano 2000, poucas interfaces de vídeo eram capazes de algum processamento gráfico. Hoje as GPUs possuem um poder de processamento genérico na ordem de *TeraFlops*, superando os processadores de uso geral, que estão na ordem de centenas de *GigaFlops* (NVIDIA CORPORATION, 2010c).

Originalmente, este poder não estava disponível para processamento de uso geral, as GPUs evoluíram para outro propósito: a geração de gráficos 3D para jogos de computador e desenhos de engenharia. Tendo então que efetuar esses gráficos, sua arquitetura interna foi desenvolvida para realizar cálculos vetoriais e matriciais, de ponto flutuante e com alto grau de paralelismo.

No entanto, esses cálculos matemáticos são usados em muitas outras áreas de conhecimento, assim, surgiram pesquisas para o uso desse processamento em cálculos genéricos. Os primeiros trabalhos nessa área utilizaram as APIs existentes para geração de gráficos, como em (THOMPSON, 2002), que já permitiam um bom desempenho, mesmo com alguma perda de precisão devida à falta de uma API específica.

A utilização de uma plataforma específica para processamento em GPUs surgiu principalmente com a empresa NVIDIA e sua plataforma CUDA. Esta plataforma detalha a arquitetura da GPU, os modelos de memória e uma linguagem de programação própria (NVIDIA CORPORATION, 2010a). Posteriormente outras empresas, como AMD e IBM, também disponibilizaram plataformas equivalentes.

Uma dessas iniciativas foi a plataforma genérica de acesso a GPUs da Apple, posteriormente padronizada pelo Khronos Group, o OpenCL (KHRONOS GROUP, 2010). O OpenCL define um modelo processamento em dispositivo genérico (podendo ser GPU, CPU ou um acelerador, como um FPGA ou o IBM Cell/B.E.) que também apresenta arquitetura específica, modelos de memória, API de acesso e linguagem de programação própria.

O presente trabalho está baseado na plataforma OpenCL, assim as próximas seções detalham a mesma e discutem a possibilidade de sua utilização de forma distribuída.

1.1 OpenCL

O OpenCL é um padrão aberto, definido pelo Khronos Group, para programação em dispositivo genérico. Hoje ele é suportado pelos principais fornecedores de GPUs (NVIDIA, AMD e, recentemente, Intel) e CPUs (Intel, AMD e IBM); e a tendência é que outros processadores venham a ter suporte, pois já existem chips para celulares homologados, como

o CPU ARMv7 com Mali-T604 GPU (KHRONOS GROUP, 2012), e outros chips, como o FPGA da empresa Altera (ALTERA, 2012), em desenvolvimento.

O framework OpenCL é composto por uma linguagem, uma API, bibliotecas e um ambiente de suporte para o desenvolvimento. A linguagem é baseada no padrão C99 (ISO/IEC 9899:1999) com algumas extensões para suportar os modelos de memória e execução do OpenCL.

O Khronos Group separa os conceitos do OpenCL em quatro modelos: modelo de plataforma, modelo de execução, modelo de memória e modelo de programação.

1.1.1 Modelo de plataforma

O modelo de plataforma é a definição da organização dos dispositivos OpenCL. Apesar existirem diferentes dispositivos, o modelo de plataforma propõe uma abstração para o seu tratamento. De uma forma geral, o modelo de plataforma do OpenCL é semelhante ao modelo de arquitetura de hardware da GPU.

O modelo de plataforma (Figura 1) descreve uma arquitetura composta de um computador (*host*) conectado em um ou mais dispositivos OpenCL (*compute device*). Cada dispositivo é dividido em uma ou mais unidades de computação (*compute units*), sendo que estes também são divididos em elementos de processamento (*processing elements*).

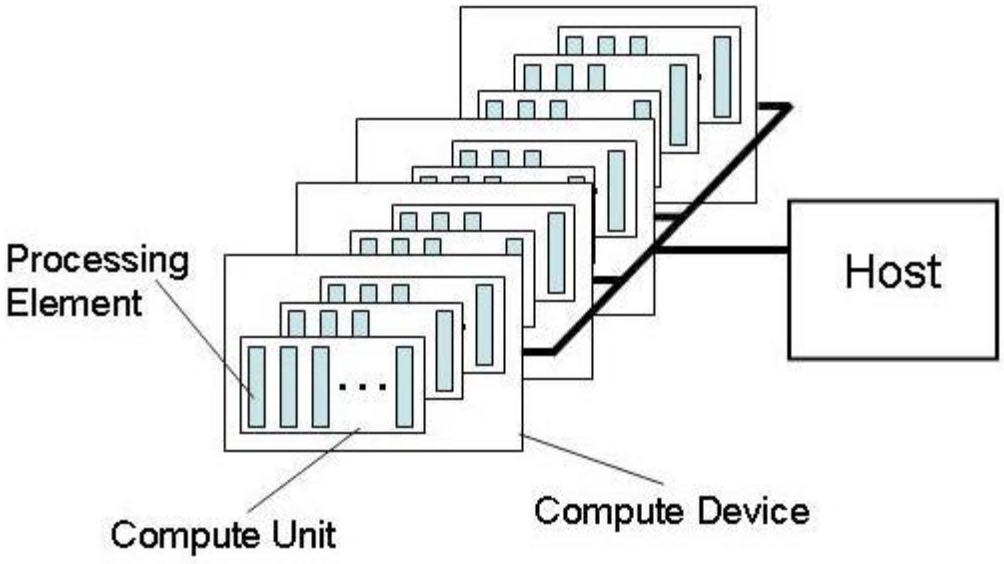

Figura 1 – Modelo de plataforma do OpenCL

Fonte: KHRONOS OPENCL WORKING GROUP. *The OpenCL Specification - Version 1.1*. Khronos Group. [S.l.], p. 385. 2010.

A execução é realizada pelas unidades de computação e elementos de processamento

do dispositivo. Nas GPUs esses dois papéis são distintos e bem definidos. As unidades de computação são grupos de elementos de processamento (os elementos são chamados de *CUDA Cores* na plataforma NVIDIA e *Stream Processors* na plataforma AMD) que mantém o mesmo ponto de execução, normalmente utilizando diferentes dados, em uma estrutura SIMD (FLYNN, 1972). Nas CPUs esses dois papéis se confundem, sendo normalmente sinônimos.

As aplicações OpenCL sempre rodam no computador *host* e enviam comandos para os dispositivos realizarem operações e cálculos específicos. Neste aspecto, o *host* e o dispositivo se assemelham ao modelo de processamento distribuído com memória independente, pois toda a comunicação é realizada por troca de mensagens, sendo que somente o *host* pode iniciar uma comunicação.

1.1.2 Modelo de execução

O modelo de execução é a definição de como aplicações OpenCL são executadas dentro dos dispositivos. Ele define as *threads* executáveis dentro dos dispositivos, chamadas de *work-items*, e como elas estão organizadas. Assim como o modelo de plataforma, o modelo de execução do OpenCL é similar ao modelo de execução em GPU. Na Figura 2 temos a ilustração do manual OpenCL sobre a organização dos componentes do modelo de execução.

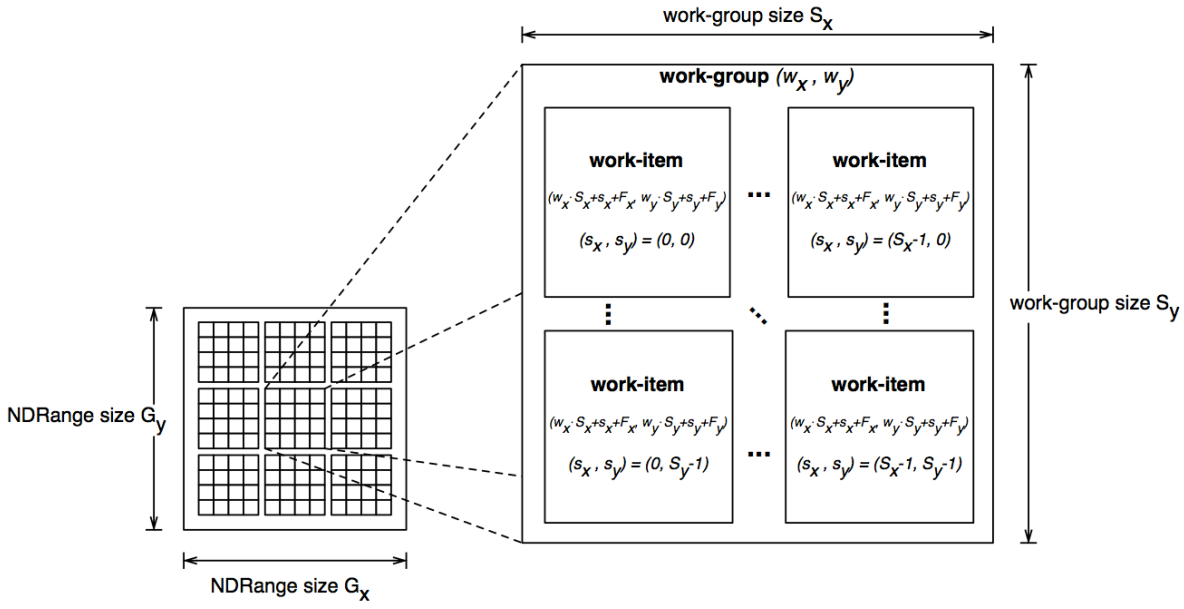

Figura 2 – Modelo de execução do OpenCL

Fonte: KHRONOS OPENCL WORKING GROUP. *The OpenCL Specification - Version 1.1*. Khronos Group. [S.l.], p. 385. 2010.

Para iniciar uma execução, a aplicação deve enviar para o dispositivo a quantidade de

work-items que deverão ser iniciados. Esta quantidade é definida como uma matriz de *work-items* que serão executados em paralelo. Esta matriz pode ser unidimensional, bidimensional ou tridimensional, assim a quantidade de *work-items* é definida passando um vetor de um, dois ou três números inteiros. O OpenCL denomina este vetor como *NDRange*.

Durante sua execução, cada *work-item* recebe um índice diferente dos demais dentro do *NDRange*. Este índice é um vetor de um, dois ou três números inteiros e é utilizado pelo *work-item* de acordo com a lógica interna da aplicação. Vale lembrar que este índice não possui significado intrínseco para o OpenCL, o significado deste índice é definido somente de acordo com o uso que a aplicação faz dele. Na Figura 3 está um exemplo de um *kernel*, onde o índice é lido na linha 3, através da função `get_global_id` da API OpenCL, e utilizado na linha 4 para definir qual posição dos vetores `vec` e `ret` o *work-item* deverá tratar. O OpenCL denomina este índice como *global ID*.

```
1:  __kernel void triple( __global double* vec, __global double* ret )
2:  {
3:      int i = get_global_id(0);
4:      ret[i] = 3 * vec[i];
5:  }
```

Figura 3 – Código utilizando o *global ID* como índice de um vetor

No entanto, o OpenCL permite separar o *NDRange* em conjuntos menores de *work-items*, fazendo que cada *work-item* receba dois índices, um índice de grupo (chamado *group ID*) sendo a posição do conjunto no *NDRange*, e um índice local (denominado *local ID*), sendo a posição do *work-item* dentro do conjunto. Também é responsabilidade da aplicação passar o tamanho dos conjuntos de execução, no entanto, em cada dispositivo existe um limite máximo para o tamanho deste conjunto. Na Figura 4 está o exemplo de um *kernel*, onde o índice é calculado, na linha 3, utilizando o *group ID* e o *localID*, e usado na linha 4 para definir qual posição dos vetores `vec` e `ret` o *work-item* deverá tratar. O OpenCL denomina o conjunto de *work-items* como *work-group*.

```
1:  __kernel void triple( __global double* vec, __global double* ret )
2:  {
3:      int i = get_group_id(0) * get_local_size(0) + get_local_id(0);
4:      ret[i] = 3 * vec[i];
5:  }
```

Figura 4 – Código utilizando o *group ID* e *local ID*

É importante notar que os *work-groups* sempre são criados, mesmo que a aplicação não indique a separação do *NDRange*. Neste caso, a biblioteca OpenCL realiza a separação automática do *NDRange* de acordo com o tamanho máximo do *work-group*.

1.1.3 Modelo de memória

O OpenCL define quatro tipos diferentes de memória: memória global, memória de constantes, memória local e memória privada. Na Figura 5 temos a ilustração do manual do OpenCL sobre os tipos de memória existentes.

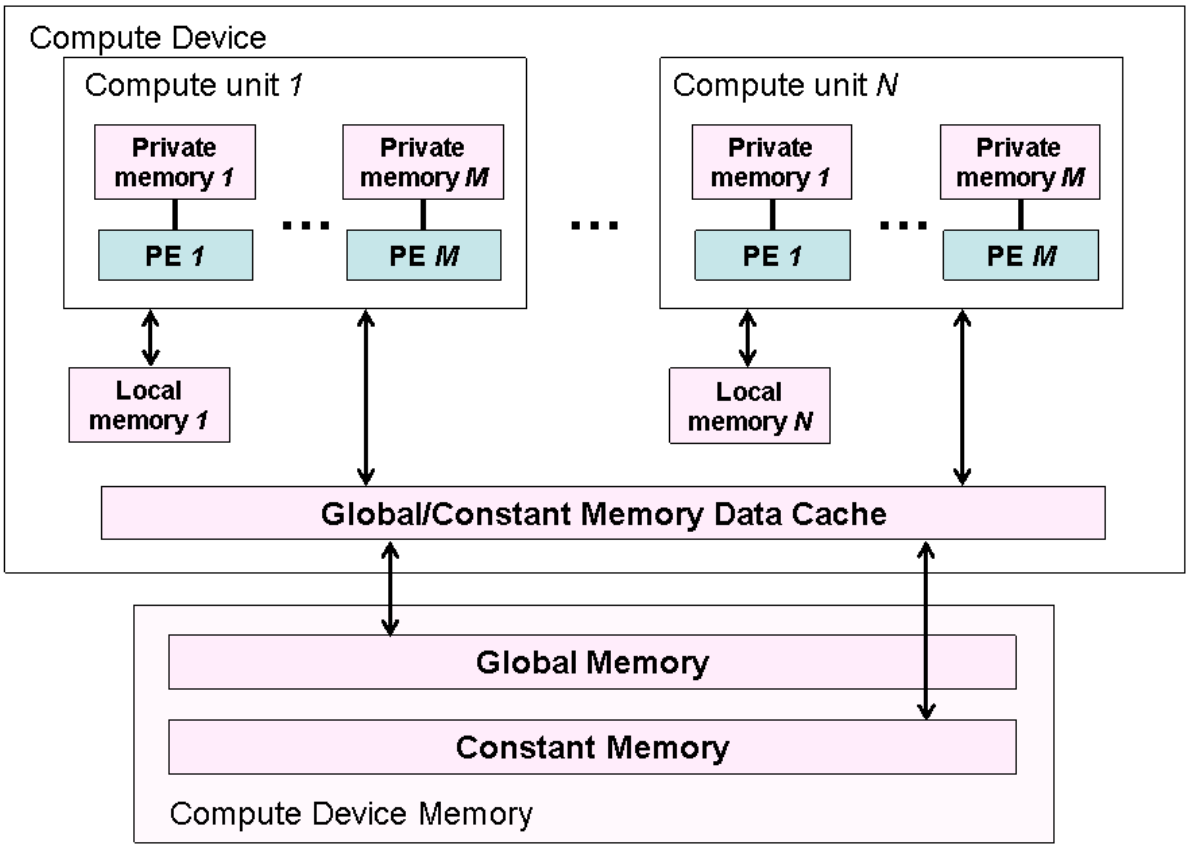

Figura 5 – Modelo de memória do OpenCL

Fonte: KHRONOS OPENCL WORKING GROUP. *The OpenCL Specification - Version 1.1*. Khronos Group. [S.l.], p. 385. 2010.

A memória global e a memória de constantes são utilizadas livremente por todos os *work-items* em todos os *work-groups* e são as únicas acessíveis para a aplicação no *host*. É responsabilidade de a aplicação alocar e preencher os dados previamente para a utilização dos *work-items*. Na GPU, no entanto, a memória global e de constantes são consideradas lentas.

A memória local é utilizada pelos *work-items* de um único *work-group*, no entanto o *host* não tem acesso à mesma. Na GPU, normalmente, esta memória é menor (normalmente de 16k nas placas NVIDIA) e mais rápida do que a memória global e de constantes. Como se trata de um recurso escasso e de alta velocidade, a maximização do uso da memória local é um importante fator para a definição do tamanho dos *work-groups*.

A memória privada é utilizada exclusivamente por cada *work-item*, sendo também

considerada rápida e inacessível à aplicação no host.

A aplicação utiliza a API OpenCL para criar, ler, escrever e apagar a memória global e de constantes, que estão inacessíveis de outro modo. Existem extensões à API OpenCL que permitem acesso direto à memória em determinados casos, como por exemplo, quando a aplicação está utilizando a CPU como dispositivo.

1.1.4 Modelo de programação

O OpenCL define uma linguagem própria, baseada no C99, e dois modelos de programação, com dados paralelos, mais comum, e com tarefas paralelas.

A linguagem de programação do OpenCL é derivada da linguagem C99, mas com algumas extensões para suportar a sua arquitetura. À linguagem C99, foram acrescidos novos tipos de dados, escalares e vetoriais, e novas palavras reservadas para qualificar as funções e também o tipo e o controle de acesso à memória.

Dos tipos de dados da linguagem C99, a linguagem OpenCL não suporta os tipos `long long` e `long double`, porém fornece 19 novos tipos de dados, escalares e vetoriais. Na Tabela 1 estão os novos tipos de dados suportados pelo OpenCL.

Tabela 1 – Novos tipos da linguagem OpenCL

Tipo	Descrição
`uchar`	Mesmo que `unsigned char`.
`ushort`	Mesmo que `unsigned short`.
`uint`	Mesmo que `unsigned int`.
`ulong`	Mesmo que `unsigned long`.
`half`	Número de ponto flutuante de 16 bits.
`char`*n*	Um vetor de *n* números inteiros de 8 bits com sinal.
`uchar`*n*	Um vetor de *n* números inteiros de 8 bits sem sinal.
`short`*n*	Um vetor de *n* números inteiros de 16 bits com sinal.
`ushort`*n*	Um vetor de *n* números inteiros de 16 bits sem sinal.
`int`*n*	Um vetor de *n* números inteiros de 32 bits com sinal.
`uint`*n*	Um vetor de *n* números inteiros de 32 bits sem sinal.
`long`*n*	Um vetor de *n* números inteiros de 64 bits com sinal.
`ulong`*n*	Um vetor de *n* números inteiros de 64 bits sem sinal.
`float`*n*	Um vetor de *n* números de ponto flutuante de 32 bits.
`double`*n*	Um vetor de *n* números de ponto flutuante de 64 bits.
`image2d_t`	Um objeto do tipo imagem 2D.
`image3d_t`	Um objeto do tipo imagem 3D.
`sampler_t`	Um objeto do tipo *sampler*.
`event_t`	Um objeto do tipo evento.

Os tipos vetoriais podem ter tamanho de 2, 3, 4, 8 ou 16 números. Por exemplo, existem tipos como double2, float3, uint4, ulong8 e char16.

As novas palavras reservadas da linguagem OpenCL são utilizadas para tratar as questões específicas da arquitetura do OpenCL e seus modelos de memória. Na Tabela 2 estão

os qualificadores desta linguagem.

Tabela 2 – Novos qualificadores da linguagem OpenCL

Qualificador	Descrição
`__global` ou `global`	Indica que o objeto de memória (buffer ou imagem) está armazenado na memória global do dispositivo.
`__local` ou `local`	Indica que a variável está armazenada na memória local do dispositivo e é visível por todos os *work-items* de um mesmo *work-group*.
`__constant` ou `constant`	Indica que a variável está armazenada na memória de constantes do dispositivo.
`__private` ou `private`	Indica que a variável está armazenada na memória privada do dispositivo e só é visível pelo próprio *work-item*.
`__kernel` ou `kernel`	Indica que a função pode se transformar em um *kernel* e ser executada pelo *host* através das funções `clEnqueueNDRangeKernel` ou `clEnqueueTask`.
`__read_only` ou `read_only`	Indica que o objeto imagem foi passado como somente leitura.
`__write_only` ou `write_only`	Indica que o objeto imagem foi passado como somente escrita.

Existem algumas limitações na linguagem. Os *kernels* só podem receber ponteiros `__global`, `__constant` ou `__local`, não podem receber ponteiros para ponteiros, não podem receber um `event_t` e só podem retornar `void`. Não são suportados: recursão, *bitfields*, ponteiros para funções, vetores sem tamanho definido, macros e funções com número indefinido de parâmetros (com reticências) e as palavras reservadas `extern`, `static`, `auto` e `register`.

No modelo de programação com dados paralelos, o mesmo código é executado em *work-items* diferentes, em paralelo, mas operando em dados diferentes. Na Figura 6 está um exemplo de um *kernel*, com dados paralelos, na linguagem OpenCL. Este modelo é desenhado para a execução em GPU, que normalmente possuem este tipo de organização do hardware e é acionado ao iniciar a execução de um *kernel* através da chamada `clEnqueueNDRangeKernel` da API OpenCL.

```
1: __kernel void
2: vecmul(__global double* vectorA, __global double* vectorB,
3:         __global double* result_vector )
4: {
5:     int i = get_global_id(0);
6:     double a = vectorA[i];
7:     double b = vectorB[i];
8:     result_vector[i] = a * b;
9: }
```

Figura 6 – Exemplo de um código OpenCL com dados paralelos

Na linha 1, o qualificador `__kernel` indica que a função `vecmul` pode ser chamada do *host*, e que ela retorna `void`, que é o único tipo possível. Na linha 2 e 3 estão os parâmetros `vectorA`, `vectorB` e `result_vector` que são ponteiros, para o tipo `double`, que estão na

memória global do dispositivo. Na linha 5 é lido o *global ID* deste *work-item*, que é usado como índice para a leitura dos dados nas linhas 6 e 7. Na linha 8 é feita a multiplicação dos dois valores lidos e o resultado é salvo.

O modelo de programação com tarefas paralelas permite que códigos diferentes sejam executados em *work-items* diferentes e operem em quaisquer dados. Na Figura 7 está um exemplo de um *kernel*, com tarefas paralelas, na linguagem OpenCL. Para acionar este modo, deve-se utilizar a chamada `clEnqueueTask` da API OpenCL para criar somente um *work-item*. Porém este modelo não é muito utilizado, mesmo com dispositivos CPU.

As linhas 1, 2 e 3 são similares ao código anterior, com a diferença que na linha 3 o parâmetro `size` é passado como um inteiro de 32 bits sem sinal. Na linha 5 é criado um loop `for` com a variável `i` indo de 0 até o valor de `size`, para executar as linhas 7, 8 e 9 que possuem o mesmo significado que as linhas 5, 6 e 7 do código anterior.

```
1: __kernel void
2: vecmul(__global double* vectorA, __global double* vectorB,
3:         __global double* result_vector, uint size )
4: {
5:     for( uint i = 0; i < size; ++i )
6:     {
7:       double a = vectorA[i];
8:       double b = vectorB[i];
9:       result_vector[i] = a * b;
10:    }
11: }
```

Figura 7 – Exemplo de um código OpenCL com tarefas paralelas.

Um detalhe interessante é que a compilação do código OpenCL é feito em tempo de execução da aplicação no *host* com o código fonte passado à biblioteca, que deve compilá-lo antes de enviar o código objeto ao dispositivo. Isto permite que a aplicação seja capaz de executar em diferentes dispositivos sem a necessidade de recompilação ou conhecimento prévio do modelo do dispositivo.

1.2 Processamento distribuído

Para realizar o processamento distribuído de uma maneira menos complexa ao desenvolvedor, deve-se encontrar um modelo capaz de realizar as tarefas tanto de distribuição do programa através da rede, como a distribuição do problema entre as GPUs.

Analisando o modelo de processamento em GPU, pode-se dizer que ele é similar ao processamento distribuído sem memória compartilhada. Cada GPU é tratada como uma unidade de processamento isolada, sua memória não é acessível à CPU ou outras GPUs e a comunicação entre eles é realizada através de troca de mensagens. Isso é um indicativo que

este modelo poderia ser viável para distribuir também o processamento entre computadores.

A plataforma OpenCL foi definida de modo a ser genérica. Ela é a mesma para a GPU de qualquer fabricante; para outros dispositivos aceleradores, como o chip IBM Cell/B.E.; e até mesmo para processadores de uso geral, como o x86. Ela também prevê que todos estes dispositivos podem ser utilizados concomitantemente, mas não foi prevista nenhuma interação entre os dispositivos. A plataforma OpenCL deixa a cargo da aplicação a seleção dos dispositivos que serão utilizados em cada processamento e a sincronização entre eles. Assim, a aplicação é obrigada a dividir o processamento especificamente para cada dispositivo, que executa sua tarefa de forma isolada dos demais.

Como, pelo modelo da plataforma OpenCL, cada GPU trabalha de forma isolada das demais e sua comunicação com a aplicação é realizada somente através de troca de mensagens, utilizar a API OpenCL como interface de um *middleware* de processamento distribuído em GPUs é viável, como foi feito pelo Mosix VirtualCL (BARAK, 2010), que será apresentado na Seção 2.4.

1.2.1 Características da API OpenCL

Para fins de análise, a interface de programação da OpenCL pode ser classificada em três categorias: de plataforma, de comunicação e de contexto.

A API de plataforma provê às aplicações um meio para identificar os dispositivos disponíveis e suas características. No OpenCL os dispositivos podem ser uma GPU, CPU ou um chip acelerador.

A API de comunicação provê às aplicações o meio de transferência de dados e comandos entre o *host* e o dispositivo. Ela permite a criação da fila de comando exclusiva de cada dispositivo, a transferência dos dados dos objetos e a execução de *kernels* nos dispositivos. Esta API também é responsável pelas primitivas de sincronização entre processamento dentro e fora dos dispositivos. Uma característica interessante é que toda a comunicação é efetivamente realizada pela fila de comando que é tratada pelo dispositivo de forma assíncrona e, dependendo da sua capacidade, fora de ordem.

A API de contexto provê a criação dos objetos manipulados pelos dispositivos e mantém o contexto de execução. Objeto é um termo genérico da API OpenCL para todos os dados manipulados. São tratados como objetos: os buffers de memória, as imagens 2D e 3D, os *samplers*, os programas e os *kernels*. Uma característica interessante é que os buffers de dados precisam ser enviados previamente ao dispositivo e somente seu ponteiro é passado como parâmetro para a execução. Isto permite que o conjunto de dados seja enviado ao

dispositivo, mesmo que ele não seja tratado de uma só vez.

1.2.2 Acesso remoto à API OpenCL

Um modelo similar ao do RPC pode ser o suficiente para transferir as chamadas da API OpenCL através da rede. Um *client stub* pode receber as chamadas da aplicação e transmitir estas via rede para o *server stub*, que recebe a mensagem e chama a OpenCL para o acesso ao dispositivo. No entanto, é interessante analisar melhor a API para tirar proveito de suas características.

A API de plataforma é somente um meio para a aplicação descobrir quais dispositivos estão disponíveis para processamento. Como as características físicas do equipamento não se alteram, os valores retornados por ela são constantes. Assim, é possível criar um *cache* com essas informações, evitando transmissões de dados via rede.

Conforme discutido, a API de comunicação é responsável pela transmissão de dados e comandos para a GPU. Para um modelo distribuído, ela deve ser preparada para transmitir as chamadas para a fila de comando através da rede. Como as filas são todas assíncronas, essa API pode criar um buffer para armazenar os comandos e aguardar um comando de sincronização para envio de todos os comandos ao *stub* via rede. Especificamente para transferência de memória, enviar os dados ao dispositivo de forma assíncrona é uma boa estratégia para as aplicações, segundo o guia de melhores práticas de programação OpenCL da NVIDIA (NVIDIA CORPORATION, 2010b). Podemos assumir então que as aplicações terão este comportamento, assim os dados poderiam ser enviados pela rede de forma assíncrona com a preparação de comandos para execução dos *kernels*.

A API de contexto permite a criação e manipulação de objetos. Na maior parte é necessário executar as chamadas desta API de imediato, porém existe a oportunidade de encontrar algumas chamadas de API que normalmente são executadas em sequência para cada objeto, por exemplo, a passagem de parâmetros para o *kernel* e em seguida a execução do mesmo para um dispositivo. Essa API pode aguardar conjuntos de comandos recorrentemente executados em sequência para transmissão em lote pela rede, minimizando o tráfego.

1.2.2.1 Diversos dispositivos remotos

Apesar de funcionar corretamente, um modelo estilo RPC obrigaria a aplicação gerenciar várias instâncias da biblioteca OpenCL para realizar processamento em vários computadores, algo que aumentaria sua a complexidade. Para manter o nível de complexidade baixo, é necessário também um modelo que suporte várias comunicações de rede, com vários

dispositivos diferentes, mas com uma única biblioteca para a aplicação.

O OpenCL prevê o uso de múltiplos dispositivos, logo é natural para as aplicações acessar múltiplos dispositivos através de sua interface. Assim um modelo de processamento distribuído poderia apresentar todos os dispositivos disponíveis, locais ou remotos, através de sua interface.

2 TRABALHOS SOBRE PROCESSAMENTO DISTRIBUÍDO EM GPU

Com a necessidade de distribuir o processamento realizado em GPU, muitos trabalhos foram feitos utilizando os métodos de distribuição existentes. Uma alternativa recorrente é utilizar as plataformas mais conhecidas, como o *Message Passing Interface* (MPI) em (FRIEDEMANN A. RÖßLER e ERTL, 2007), (FAN, 2004), (PANETTA, TEIXEIRA, *et al.*, 2009) e (PENNYCOOK, 2010).

Porém, ao utilizar esta abordagem é necessário combinar dois modelos diferentes de distribuição em um mesmo programa. Um específico da GPU, que necessita a divisão do problema em pequenos blocos de execução independente, e o da MPI, que executa o programa em um grupo de computadores interligados. No entanto, a plataforma MPI, assim como o *Remote Procedure Call* (RPC), poderia ser utilizada como base por um *middleware* de processamento. Logo é necessário avaliar os prós e contras de utilizar uma dessas plataformas em comparação a utilizar somente um transporte de rede confiável, como o TCP.

Nas próximas seções são discutidos alguns aspectos relacionados ao uso do RPC e do MPI como infraestrutura para um *middleware* de processamento em GPU e também as propostas rCUDA e o Mosix VirtualCL, que são plataformas para processamento distribuído transparente em GPU, sendo que o rCUDA utiliza a plataforma CUDA, enquanto o Mosix VirtualCL utiliza o OpenCL.

2.1 RPC

Como a API OpenCL é disponibilizada às aplicações como uma biblioteca dinâmica e com uma interface bem definida, o mecanismo RPC seria um meio para a realização de uma chamada remota a uma API de processamento de GPU.

O uso da estrutura do RPC tem uma estrutura considerada robusta e com versões bastante testadas e utilizadas. Como a interface da API OpenCL é bem definida, seria possível construir o arquivo IDL para a criação dos *stubs* do cliente e do servidor para a construção de uma camada de acesso remoto.

Porém, o RPC se propõe a ser genérico, portanto não existe nenhuma capacidade de aproveitar as características da API OpenCL, como as descritas na Subseção 1.2.2, para minimizar o uso da rede e melhorar o desempenho. Assim, não seria possível se aproveitar das filas de comando assíncronas do OpenCL ou de realizar cache local com as informações dos dispositivos disponíveis.

Outro problema é o caso específico da função `clSetKernelArg` da API OpenCL. Esta

função é responsável por definir os parâmetros do *kernel* a ser executado dentro do dispositivo e é chamada uma vez para cada parâmetro. Os parâmetros do *kernel* podem ser números inteiros e de ponto flutuante ou objetos do contexto do OpenCL, porém a função `clSetKernelArg` somente recebe qual é o número do parâmetro do *kernel*, um buffer, contendo o parâmetro, e o tamanho deste buffer. A biblioteca OpenCL de um dispositivo conhece a estrutura do *kernel* e sabe que, de acordo com a ordem do parâmetro recebido, qual o formato de seu conteúdo. Porém, o RPC não seria capaz de analisar o buffer recebido e, como os objetos de contexto da API OpenCL são ponteiros opacos (`void*` na linguagem C), eles precisariam ser transformados durante a passagem de dados através da rede. Assim, o uso do RPC, sem nenhuma camada de compatibilidade especialmente construída, seria inviável nessa configuração.

Desta forma, por não ser capaz de aproveitar as características da API OpenCL e ter necessidade de um tratamento especial para a função `clSetKernelArg`, o RPC não é uma boa alternativa como plataforma para construção do *middleware*.

2.2 MPI

O MPI é um *framework* de processamento distribuído muito utilizado, inclusive em conjunto com o processamento em GPU, no entanto, o uso das duas tecnologias em conjunto aumenta a complexidade do programa como um todo. A programação da GPU e o MPI possuem modelos distintos de tratamento de dados, sincronização e execução que precisam ser conciliados.

O pseudocódigo (Figura 8) mostra uma aplicação que utiliza MPI e OpenCL para acessar múltiplas GPUs conectadas em um ou mais computadores. Por utilizar MPI, a aplicação é dividida em vários processos que se comunicam através das primitivas do MPI. Dentre os processos da aplicação, existe um processo principal, chamado de *Master*, que controla todos os outros processos, chamados de *Slave*. O processo *Master* e os *Slave* podem ser executados um mesmo computador, ou em vários computadores conectados através de uma rede. Nas linhas 3 a 9 está o pseudocódigo do processo com o papel de *Master* e nas linhas 13 a 23 está o pseudocódigo do processo com o papel de *Slave*.

Na linha 3, o processo *Master* executa um loop para enviar um conjunto de dados para processamento para cada processo *Slave*, através da linha 4. De forma análoga, nas linhas 7 e 8, o processo *Master* aguarda os dados de retorno de cada *Slave*.

Na linha 13, o processo *Slave* recebe os dados do *Master*. Após o recebimento dos dados, o processo *Slave* passa a utilizar o OpenCL para o processamento dos dados através

das linhas 15 a 21. Na linha 15 a aplicação cria o *kernel* que será executado na GPU. Na linha 16, os dados são copiados para GPU, retornando o ponteiro `gpu_data` que será usado na linha 17 como o parâmetro do *kernel*. Na linha 18 é passado o parâmetro a área de memória onde o *kernel* irá salvar o resultado. Na linha 19 o *kernel* é efetivamente executado e o resultado do processamento é lido na linha 20. Na linha 21 a aplicação aguarda todos os comandos enviados anteriormente serem executados na GPUs. Após o término da execução no processamento na GPU, o processo *Slave* volta a utilizar o MPI, na linha 23, para enviar os dados de retorno para o processo *Master*.

```
1: // MPI Master
2:
3: foreach( node ) {
4:     MPI_Send( some_data );
5: }
6:
7: foreach( node ) {
8:     MPI_Recv( response_data );
9: }
10:
11: // MPI Slave
12:
13: MPI_Recv( some_data )
14:
15: kernel = clCreateKernel( source_code );
16: gpu_data = clEnqueueWriteBuffer( gpu, some_data );
17: clSetKernelArg( kernel, 0, gpu_data );
18: clSetKernelArg( kernel, 1, gpu_response );
19: clEnqueueNDRangeKernel( gpu, kernel );
20: response_data = clEnqueueReadBuffer( gpu, gpu_response );
21: clFinish();
22:
23: MPI_Send( response_data );
```

Figura 8 – Pseudocódigo de uma aplicação MPI e OpenCL

Como comparação, o pseudocódigo na Figura 9 mostra uma aplicação que utiliza somente o OpenCL para acessar múltiplas GPUs conectadas em um computador. Os passos são equivalentes às linhas 15 a 21 da Figura 8, com a diferença que somente um *kernel* é criado, na linha 1, e é feito um loop na linha 2 para executar os passos 4 a 8. Na linha 10 a aplicação aguarda todas as GPUs terminarem a execução para podem continuar.

```
1: kernel = clCreateKernel( source_code );
2: foreach( gpu )
3: {
4:     gpu_data = clEnqueueWriteBuffer( gpu, some_data );
5:     clSetKernelArg( kernel, 0, gpu_data );
6:     clSetKernelArg( kernel, 1, gpu_response );
7:     clEnqueueNDRangeKernel( gpu, kernel );
8:     response_data = clEnqueueReadBuffer( gpu, gpu_response );
9: }
10: clFinish();
```

Figura 9 – Pseudocódigo de uma aplicação OpenCL

Neste caso, a aplicação consiste de apenas um processo sendo executado e nele está todo o controle do processamento nas GPUs. A aplicação envia os dados e comandos diretamente a cada GPU e aguarda o seu resultado. Não existe nenhum tipo de mecanismo de sincronização além dos previstos pelo OpenCL. No entanto, somente é possível acessar as GPUs conectadas no computador, enquanto no código da Figura 8 é possível acessar GPUs instaladas em outros computadores.

Esses pseudocódigos são modelos simplificados de uma aplicação utilizando MPI com OpenCL e outra somente OpenCL, porém servem para ilustrar como o uso de duas tecnologias distintas em conjunto possui complexidade maior que uma aplicação utilizando somente uma delas.

O MPI pode ser utilizado como *framework* para um *middleware* de processamento distribuído em GPU. Ele possui versões específicas de diversos fabricantes de hardware, como, por exemplo, da Intel (INTEL, 2013), e versões em código livre, como o OpenMPI (OPENMPI, 2013). Além disso, existe bom conjunto de *drivers* para suporte de redes de alta velocidade, como o Infiniband (INFINIBAND TRADE ASSOCIATION, 2013).

Porém, o MPI possui diversas funções para suporte de aplicações distribuídas, além do transporte de dados via rede, e, das funções suportadas pelo MPI, um *middleware* de processamento se restringiria a utilizar somente as de comunicação de dados via rede.

Assim, apesar de viável e robusto, o uso do MPI para um *middleware* de processamento distribuído em GPUs não agrega muita funcionalidade sobre um transporte de rede confiável.

2.3 rCUDA

O rCUDA (DUATO, IGUAL, *et al.*, 2010b) é um *framework* de processamento distribuído em GPU que utiliza a plataforma CUDA como interface de acesso às GPUs remotas. Ele foi desenvolvido originalmente na *Universidad Politécnica de Valencia* na Espanha e sua concepção foi feita a partir da ideia de diminuir o número de GPUs em um cluster de alto processamento para a redução do consumo de energia elétrica (DUATO, PENA, *et al.*, 2010a).

A arquitetura do rCUDA é composta de um Cliente e um Servidor. O Cliente é uma biblioteca utilizada pela aplicação, de forma transparente, e provê as chamadas do CUDA através de suas funções. Estas funções transmitem os valores passados pela aplicação para o Servidor através da rede, que recebe as informações, executa os comandos na GPU e envia a resposta para o Cliente novamente pela rede.

No entanto, por ser uma arquitetura proprietária, o CUDA possui funções internas e não documentadas pela NVIDIA, dificultando a simples interceptação das chamadas de sua biblioteca. A solução encontrada pelo rCUDA, até a versão 3, foi criar um *framework* para compilação dos códigos existentes em CUDA, que utiliza somente chamadas documentadas, ao invés de utilizar um processo transparente a uma aplicação já compilada. Ou seja, para utilizar o rCUDA era necessário compilar a aplicação utilizando o seu *framework*.

Com a versão 4, o rCUDA não possui mais essa limitação, sendo possível apenas substituir a biblioteca de acesso ao CUDA para realizar o acesso a GPUs remotas de forma transparente para a aplicação.

Atualmente o rCUDA é disponibilizado gratuitamente, porém seu código é proprietário. Existem versões específicas para a rede TCP/IP e Infiniband, comum em ambientes de clusters de alto desempenho.

2.4 Mosix VirtualCL

O Mosix VirtualCL (BARAK, 2010) é um *middleware* de processamento distribuído transparente em GPU que utiliza o OpenCL como interface à GPUs remotas. Ele é parte integrante do *Mosix Cluster Operating System* (BARAK e SHILOH, 1999).

A arquitetura do Mosix VirtualCL é composta pela biblioteca Cliente, o *Broker* e o Servidor. A biblioteca Cliente intercepta as chamadas OpenCL da aplicação e envia as solicitações para um serviço no próprio computador chamado *Broker*. O *Broker* é responsável por centralizar a comunicação de diversas aplicações do computador, ele recebe as solicitações dos Clientes e envia as chamadas para os diversos Servidores através da rede. O Servidor é responsável por receber as chamadas dos *Brokers* e executar as chamadas correspondentes na GPU, retornado o resultado novamente para o *Broker* correspondente que a reenvia para a biblioteca Cliente.

Por utilizar o padrão OpenCL, onde todas as chamadas da API são bem documentadas, o Mosix VirtualCL pode agir de forma transparente para aplicação. Porém, para melhorar o desempenho do processamento remoto, ele possui uma estrutura proprietária para o envio de múltiplas chamadas do Cliente até o Servidor. Essa estrutura é chamada de SuperCL.

```
1: cl_int clSuper( cl_command_queue command_queue,
2:                 struct super_sequence* sequence,
3:                 cl_uint num_events_in_wait_list,
4:                 const cl_event* event_wait_list,
5:                 cl_event* event);
```

Figura 10 – Função `clSuper` do Mosix VirtualCL

O SuperCL é composto de uma única chamada, apresentada na Figura 10, onde é

possível passar uma lista de comandos compatíveis com o OpenCL para serem executados em ordem no Servidor.

A chamada `clSuper` é similar às chamadas `clEnqueue*` da API OpenCL, os parâmetros `command_queue`, `num_events_in_wait_list`, `event_wait_list` e `event` possuem o mesmo significado. O parâmetro `sequence` é um ponteiro para uma lista de comandos do SuperCL que serão enviados em uma só mensagem e executados em ordem no Servidor. Esta lista deve ser terminada com um código específico, informando ao SuperCL o término dos comandos.

Por possuir um esquema próprio, fora do padrão OpenCL, para enfileirar as chamadas e assim minimizar o tráfego de dados, as aplicações necessitam ser modificadas para suportar o SuperCL e terem acesso ao uso mais eficiente da rede.

O Mosix VirtualCL possui uma proposta similar ao do presente trabalho, ele utiliza a API OpenCL como interface e possibilita o acesso remoto a GPUs de forma transparente. No entanto, diferente do trabalho atual, necessita de API fora do padrão OpenCL, o SuperCL, para realizar otimizações, enquanto este trabalho utiliza otimizações também transparentes às aplicações, como descrito na Seção 4.3.

3 SISTEMA PROPOSTO: *MIDDLEWARE* DISTRIBUTEDCL

O DistributedCL é um *middleware* de processamento distribuído em GPU com a interface da API OpenCL, que considera as características discutidas na Seção 1.2. Seu objetivo é prover, de forma transparente, o acesso às GPUs existentes em computadores interligados através de uma rede a uma aplicação que utiliza OpenCL.

Para utilizar o *middleware* DistributedCL, as aplicações não necessitam nenhuma alteração ou novo processo de compilação, toda comunicação de rede e configuração necessárias são realizadas dentro da estrutura do *middleware*. Assim, aplicações de mercado, sem nenhuma modificação, podem utilizar múltiplas GPUs remotamente através do *middleware* DistributedCL, desde que sejam compatíveis com o padrão OpenCL.

Internamente, o *middleware* DistributedCL possui uma arquitetura modular com camadas bem definidas, onde cada camada é responsável por prover suas funcionalidades às camadas superiores.

Um protótipo, descrito no Capítulo 4, foi desenvolvido com base nesta arquitetura onde foram introduzidas várias otimizações, que por sua vez influenciaram refinamentos na própria arquitetura. Alguns trechos do seu código são usados nas próximas seções para ilustrar aspectos importantes da arquitetura.

3.1 Arquitetura

A arquitetura do *middleware* DistributedCL é composta por camadas (Figura 11) com comunicação e papéis bem definidos: camada de interface, camada de composição, camada de acesso local, camada de acesso remoto, camada de mensagem, camada de rede e camada de servidor.

Estas camadas possuem uma similaridade com as camadas do NFS (TANENBAUM e STEEN, 2006). Assim, da mesma forma que o NFS procura preservar a semântica da API do sistema de arquivos do Unix não impondo ao programador um novo modelo, a abordagem do *middleware* DistributedCL também não requer do programador uma mudança no programa. Diferente do NFS, entretanto, o *middleware* DistributedCL não utiliza RPC como transporte de dados através da rede.

3.1.1 Camada de interface

A camada de interface é responsável por prover a API OpenCL às aplicações e traduzir as chamadas das aplicações para a camada de composição. Ela também é responsável

pela contagem de referência dos objetos da OpenCL e por suportar a extensão ICD (KHRONOS GROUP, 2011) definida pelo Khronos Group.

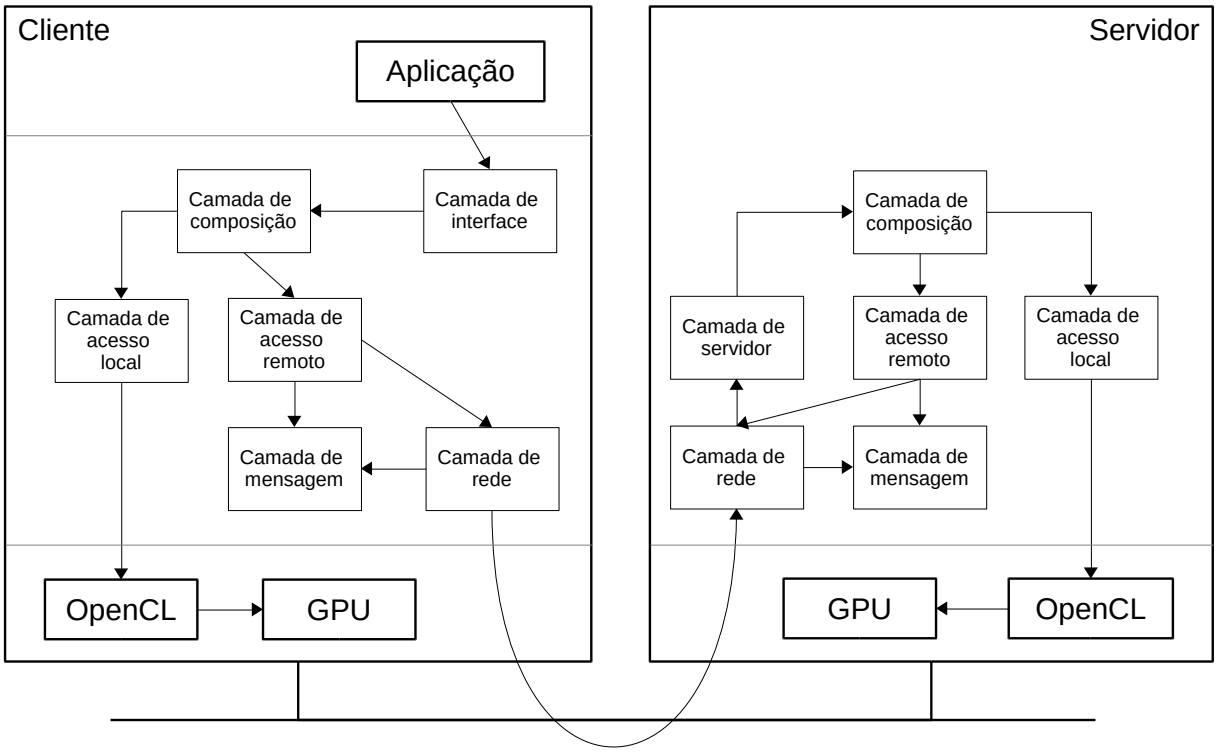

Figura 11 – Arquitetura do middleware DistributedCL

Esta camada é construída como uma biblioteca dinâmica e provê as funções definidas no padrão da OpenCL para acesso das aplicações. Esta camada possui todas as funções da API OpenCL, que simplesmente fazem a tradução das chamadas e parâmetros passados pela aplicação para as chamadas equivalentes da camada de composição. Estas funções também verificam os parâmetros passados de acordo com a especificação da OpenCL e as características do *middleware* DistributedCL.

Esta camada também é responsável por suportar a extensão ICD definida pelo Khronos Group. Como definido por esta extensão, todos os objetos OpenCL devem ser ponteiros para uma estrutura que deve possuir como o primeiro membro um ponteiro para uma tabela onde estão todas as funções da API OpenCL suportadas pela biblioteca.

```
1:  struct cl_object
2:  {
3:     _cl_icd_dispatch_table* table;
4:     uint32_t dcl_type;
5:     uint32_t reference_count;
6:     void* dcl_object;
7:  };
```

Figura 12 – Estrutura `cl_object` retornada pela camada de interface

No protótipo todos os objetos retornados pela camada de interface para a aplicação são

ponteiros para a estrutura apresentada na Figura 12. Na linha 3, como especificado no padrão, está o ponteiro para a tabela de funções da API OpenCL; na linha 4 está o tipo de objeto que está sendo representado; na linha 5 está a contagem de referência para este objeto; e na linha 6 o ponteiro para o objeto da camada de composição.

Esta camada também é responsável por controlar a contagem de referência dos objetos. O protótipo utiliza a mesma estrutura da extensão ICD para o controle dessas referências, assim o código da camada de interface também é responsável por manter o número de referências de todos os objetos.

```
1:  clCreateCommandQueue( cl_context context, cl_device_id device,
2:                        cl_command_queue_properties properties,
3:                        cl_int* errcode_ret )
4:  {
5:    if((properties & CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE) != 0)
6:    {
7:      if( errcode_ret != NULL )
8:        *errcode_ret = CL_INVALID_QUEUE_PROPERTIES;
9:      return NULL;
10:   }
11:   try
12:   {
13:     icd_object_manager& icd = icd_object_manager::get_instance();
14:     composite_context* context_ptr =
          icd.get_object_ptr< composite_context >( context );
15:     composite_device* device_ptr =
          icd.get_object_ptr< composite_device >( device );
16:     composite_command_queue* queue_ptr =
          context_ptr->create_command_queue( device_ptr, properties );
17:     if( errcode_ret != NULL )
          *errcode_ret = CL_SUCCESS;
18:     return icd.get_cl_id<composite_command_queue>( queue_ptr );
19:   }
20:   catch( dcl::library_exception& ex )
      {
21:     if( errcode_ret != NULL )
          *errcode_ret = ex.get_error();
22:     return NULL;
23:   }
24: }
```

Figura 13 – Função `clCreateCommandQueue`

Como exemplo de código da camada de interface, a Figura 13 contém a função `clCreateCommandQueue` do protótipo que cria uma fila de comando. Na linha 1 tem a assinatura da função de acordo com o padrão. A linha 5 verifica o parâmetro `properties`, procurando pelo valor `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE`, não suportado pelo protótipo para as filas de comando, resultando no erro retornado na linha 8. As linhas 14 e 15 fazem a tradução dos parâmetros `context` e `device` recebidos, em objetos da camada de composição `context_ptr` e `device_ptr`, respectivamente. Na linha 16 a fila de comando é criada através de uma chamada à camada de composição, através do objeto `context_ptr`.

Nas linhas 17 e 18 são retornados o código de erro e o objeto criado, indicando o sucesso da operação. As linhas 11 e 20 a 22 são utilizadas para tratamento de exceções, todas as chamadas dentro desta região podem gerar erros que serão capturados pela estrutura de exceção do C++ para tratamento. Caso aconteça uma exceção o erro será retornado para a aplicação.

3.1.2 Camada de composição

A camada de composição é responsável por criar a abstração de uma única plataforma OpenCL com múltiplos dispositivos para a camada de interface, no cliente, e para a camada de servidor, no servidor. Ela é responsável por manter as referências de todos os dispositivos, locais e remotos conectados.

Ela possui todas as funcionalidades da API OpenCL e repassa as chamadas recebidas para as camadas de acesso local e acesso remoto. Esta camada também tem como atribuição iniciar e configurar as camadas acesso local e acesso remoto. Para isso, esta camada provê às camadas superiores o serviço de adicionar uma biblioteca OpenCL local e conectar a um servidor remoto.

Esta camada possui similaridades com a camada *Virtual File System (VFS)* no NFS. Assim como a camada VFS cria a abstração da localização do arquivo, direcionando as chamadas para a leitura local ou através da rede, a tarefa da camada de composição é direcionar as chamadas vindas das camadas de interface e servidor para as camadas de acesso local e acesso remoto, de acordo com o caso.

A escolha de quais dispositivos serão acionados é feita de duas maneiras, uma é repassando a chamada para todos os dispositivos conectados, locais ou remotos, e a outra é selecionando qual dispositivo terá a chamada repassada, sendo que esta escolha depende exclusivamente de qual é a chamada.

```
1:  void composite_kernel::execute(
            const generic_command_queue* queue_ptr,
            const ndrange& offset, const ndrange& global,
            const ndrange& local, events_t& wait_events,
            generic_event** event_ptr )
2:  {
3:     const generic_context* ctx = queue_ptr->get_context();
4:     generic_kernel* kernel_ptr = find( ctx );
5:     kernel_ptr->execute( queue_ptr, offset, global, local,
                            wait_events, event_ptr );
6:  }
```

Figura 14 – Método `composite_kernel::execute`

As chamadas relativas aos dispositivos e às filas de comando, inclusive às de

enfileiramento de objetos (as funções `clEnqueue*` da API OpenCL), são executadas sempre fazendo uma seleção de qual dispositivo a chamada deverá ser repassada. Como exemplo, na Figura 14 está um método do protótipo com esta característica. Na linha 1 temos a assinatura do método `execute` da classe `composite_kernel` que provê a funcionalidade de executar um *kernel* em um dispositivo, na API OpenCL isto é feito enfileirando o pedido em uma fila de comando. Na linha 3 descobre-se qual o contexto associado à fila de comando selecionada, passada pelo parâmetro `queue_ptr`. Este contexto é utilizado para selecionar, na linha 4, qual o objeto *kernel*, do acesso local ou acesso remoto, está associado a este mesmo contexto. Conhecendo então o objeto da classe de acesso correta, repassa-se a chamada recebida, como mostrado na linha 5.

As chamadas relativas à criação e manipulação dos objetos da OpenCL, inclusive o próprio contexto, são sempre repassadas a todos os dispositivos. Como exemplo, na Figura 15 está um método do protótipo com esta característica. Na linha 1 temos a assinatura do método `create_kernel` da classe `composite_program` que cria um objeto *kernel*, relativo à função com o nome passado por `kernel_name`, a partir de um objeto *program*. A linha 3 cria o objeto `composite_kernel` que será retornado, no mesmo contexto do programa, mas ainda vazio, sem nenhuma referência a classes de acesso local e acesso remoto. A linha 4 enumera todas as instâncias das classes de acesso local e acesso remoto contidas na camada de composição. A linha 6 cria um objeto *kernel* em cada biblioteca carregada pelo acesso local e cada servidor conectado pelo acesso remoto, e inclui este *kernel* no objeto composto que será retornado, como mostrado na linha 7. O novo objeto é retornado na linha 9.

```
1:  generic_kernel*
    composite_program::create_kernel( const std::string& kernel_name )
2:  {
3:     composite_kernel* kernel_ptr =
          new composite_kernel( get_context(), kernel_name );
4:     for( iterator it = begin(); it != end(); ++it )
5:     {
6:        generic_kernel* gk = it->second->create_kernel( kernel_name );
7:        kernel_ptr->insert_context_object( it->first, gk );
8:     }
9:     return kernel_ptr;
10: }
```

Figura 15 – Método `composite_program::create_kernel`

3.1.3 Camada de acesso local

A camada de acesso local é responsável por acessar as bibliotecas OpenCL, utilizando API OpenCL para prover suas funcionalidades. Ela é responsável por carregar as bibliotecas e possui a mesma interface das camadas de composição e de acesso remoto.

A camada de acesso local provê a funcionalidade inversa da camada de interface, ou seja, recebe requisições da camada de composição e transforma em chamadas de uma biblioteca OpenCL previamente carregada. Como exemplo, na Figura 16 está um código do protótipo com o construtor da classe `context`, que recebe como parâmetro a plataforma e o tipo de dispositivo. Na linha 5 é preparada uma estrutura passando a plataforma e na linha 6 a chamada à biblioteca OpenCL para criação do contexto. As linhas 7 e 8 fazem o tratamento de erro através de exceções caso ocorra algum erro na execução do comando `clCreateContextFromType`. Na linha 9 é salvo o objeto OpenCL retornado pela função, este objeto será utilizado em outros métodos da classe para endereçar o mesmo objeto.

```
1:  context::context( platform& pform, cl_device_type type )
2:  {
3:    cl_int error_code;
4:    cl_context ctx;
5:    cl_context_properties properties[3] = { CL_CONTEXT_PLATFORM,
        reinterpret_cast<cl_context_properties>( pform.get_id() ), 0};
6:    ctx = opencl_.clCreateContextFromType( properties, type, NULL,
        static_cast<void*>( NULL ), &error_code );
7:    if( error_code != CL_SUCCESS )
8:      throw library_exception( error_code );
9:    set_id( ctx );
10: }
```

Figura 16 – Construtor da classe `context`

3.1.4 Camada de acesso remoto

A camada de acesso remoto é responsável por realizar a comunicação com a camada de servidor em outro computador através da rede para prover as funcionalidades da API OpenCL. A camada de acesso remoto possui a mesma interface das camadas de composição e de acesso local. A camada de acesso remoto recebe as requisições da camada de composição, as transformam em mensagens, usando a camada de mensagens, e as enviam através da camada de rede. Também é responsabilidade desta camada a decisão se o envio da mensagem do cliente para o servidor será síncrono ou assíncrono.

A principal funcionalidade da camada de acesso remoto é a transformação das chamadas recebidas em mensagens, utilizando a camada de mensagens, e chamar a camada de rede para transporte dessas mensagens para a camada de servidor em outro computador. Comparando com a arquitetura do NFS, esta camada faz papel similar ao *NFS Client*.

É decisão da camada de acesso remoto o envio imediato da mensagem, decorrente de uma chamada feita, ou enfileiramento da mesma para posterior envio em lote. Esta decisão é tomada principalmente pelo tipo de chamada. As chamadas da API de plataforma do OpenCL são efetuadas com *cache*, ou seja, na primeira solicitação todas as informações do dispositivo

ou plataforma são solicitadas e armazenadas localmente. As chamadas da API de contexto são executadas diretamente, com exceção dos parâmetros passados para execução do *kernel*, neste caso os parâmetros são passados junto com a chamada para a execução do mesmo. As chamadas da API de comunicação são, em sua maioria, armazenadas para posterior envio. O envio dessas chamadas é realizado quando outra chamada, que necessita de resposta imediata, é feita. Este processo é mais bem detalhado na descrição de sua implementação no protótipo (Seção 4.3).

Na Figura 17 está, como exemplo, o código do protótipo para o método `create_kernel` da classe `remote_program`, responsável por criar um novo *kernel* em outro computador. Na linha 3 o método cria uma nova mensagem de criação de *kernel*, que é preenchido nas linhas 4 e 5. Vale a pena notar na linha 5 que somente o identificador de rede do programa é colocado na mensagem. Na linha 7 a mensagem então é enviada através da rede de forma síncrona, através do método `send_message` da camada de rede, sendo então, executada no servidor e sua resposta carregada no mesmo objeto mensagem utilizado como parâmetro. Na linha 8 um novo objeto remoto é criado, no mesmo contexto do programa, e na linha 9 seu identificador é colocado a partir da mensagem retornada pelo servidor. Na linha 10 o objeto é retornado. Note que não existe nenhum tratamento de erro neste método, qualquer erro deverá ser tratado, na forma de exceção, pela classe que comandou a chamada.

```
1:  generic_kernel* remote_program::create_kernel( string& name )
2:  {
3:     dcl_message< msgCreateKernel >* msg_ptr =
          new dcl_message< msgCreateKernel >();
4:     msg_ptr->set_name( name );
5:     msg_ptr->set_program_id( get_remote_id() );
6:     boost::shared_ptr< base_message > message_sp( msg_ptr );
7:     session ref .send message( message sp );
8:     remote_kernel* kernel_ptr = new remote_kernel( context_, name );
9:     kernel_ptr->set_remote_id( msg_ptr->get_remote_id() );
10:    return reinterpret_cast< generic_kernel* >( kernel_ptr );
11: }
```

Figura 17 – Método `remote_program::create_kernel`

3.1.5 Camada de mensagens

A camada de mensagens é responsável pelos dados do protocolo de comunicação passado pela rede. As estruturas de controle das mensagens e dos pacotes serão discutidas mais detalhadamente junto do protocolo de comunicação na Seção 3.2.

A comunicação entre as camadas de acesso remoto e camada de servidor é feita com troca de mensagens, que são construídas através da camada de mensagens. Para o envio dos dados, as mensagens são agrupadas em pacotes, que podem conter até 65.535 mensagens. A

camada de mensagens também é responsável por criar e tratar estes pacotes de mensagens. O processo de comunicação entre as camadas de acesso remoto, mensagem, rede e servidor está exemplificado na Figura 18.

No passo 1 a camada de acesso remoto monta uma requisição através da camada de mensagem, enfileira a mensagem no passo 2 e solicita a transmissão à camada de rede no passo 3. No passo 4 a camada de rede pede à camada de mensagens a criação de um pacote de transmissão e a serialização dos dados, e no passo 5 transfere os dados através da rede. Chegando ao servidor, a camada de rede solicita, no passo 6, à camada de mensagem o tratamento do pacote e a criação dos objetos da camada de mensagem relativos àquele conjunto de dados e no passo 7 passa o controle para a camada de servidor. O retorno das chamadas é realizado de maneira análoga.

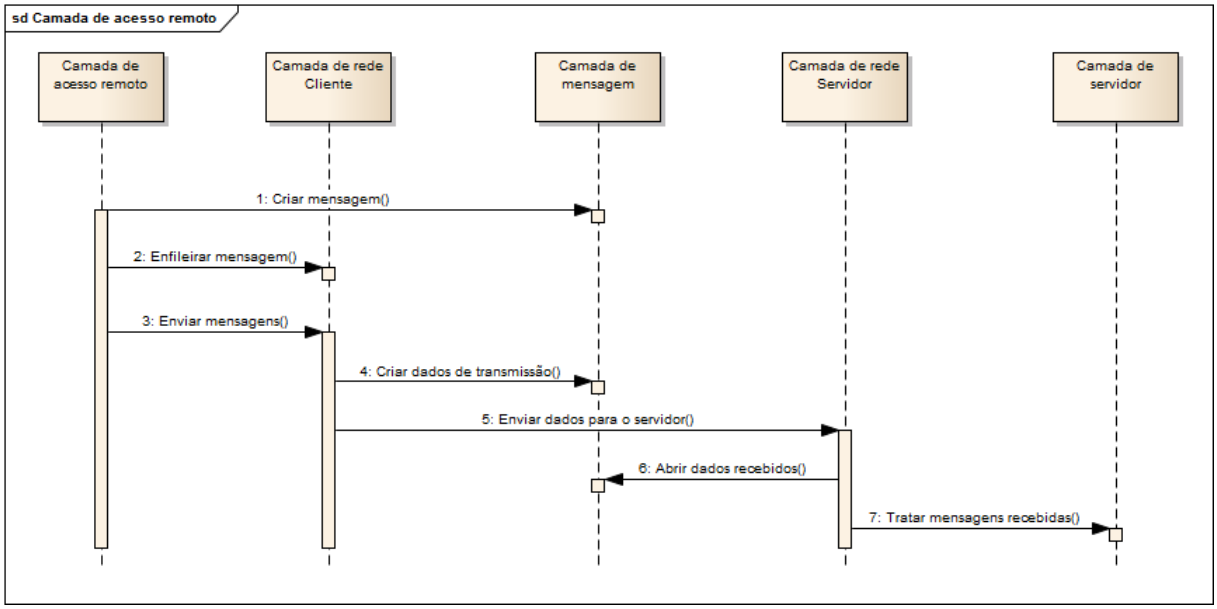

Figura 18 – Camadas de acesso remoto, mensagem, rede e servidor.

No protótipo, a camada de mensagens é composta por classes equivalentes às chamadas da API OpenCL, mas não necessariamente com uma relação direta com elas. Uma mesma mensagem pode tratar mais de uma chamada da API OpenCL ou ter um comportamento único para os diferentes parâmetros passados. Um exemplo deste segundo caso é o retorno de informações sobre o dispositivo. Na API OpenCL a chamada `clGetDeviceInfo` retorna somente uma informação sobre o dispositivo em cada chamada, enquanto a mensagem `msgGetDeviceInfo` equivalente trata todas as informações em uma única chamada.

Para a camada de mensagens, todos os objetos OpenCL manipulados são mapeados em um identificador de rede pela camada de acesso remoto. As classes da camada de

mensagem manipulam somente estes identificadores de rede, não tendo nenhum acesso às classes de acesso remoto.

No protótipo, as classes da camada de mensagens possuem chamadas para serialização e desserialização dos dados de requisição e resposta. Na Figura 19 está o exemplo da serialização e desserialização dos dados da requisição de criação de uma fila de comando através dos métodos `create_request` (linha 1) e `parse_request` (linha 10), respectivamente. Na linha 3, assim como na linha 12, o ponteiro passado é convertido no ponteiro da estrutura que será utilizada para o transporte de dados. Nas linhas 5, 6 e 7 os atributos `device_id_`, `context_id_` e `properties_` são passados para os dados serializados. Note que antes de colocar os dados na estrutura de transporte os mesmos são devidamente convertidos para ordem de bytes *little-endian*, essa conversão é feita pela função `host_to_network`. De forma análoga, as linhas 14, 15 e 16 os mesmos atributos são preenchidos com os dados serializados na estrutura, com a conversão de *little-endian* para a arquitetura do processador através da função `network_to_host`.

```
1:  void dcl_message<msgCreateCommandQueue>::create_request(void* ptr)
2:  {
3:     msgCreateCommandQueue_request* request_ptr =
          reinterpret_cast< msgCreateCommandQueue_request* >( ptr );
4:
5:     request_ptr->device_id_ = host_to_network( device_id_ );
6:     request_ptr->context_id_ = host_to_network( context_id_ );
7:     request_ptr->properties_ = host_to_network( properties_ );
8:  }
9:
10: void dcl_message<msgCreateCommandQueue>::parse_request(void* ptr)
11: {
12:    msgCreateCommandQueue_request* request_ptr =
          reinterpret_cast< msgCreateCommandQueue_request* >( ptr );
13:
14:    device_id_ = network_to_host( request_ptr->device_id_ );
15:    context_id_ = network_to_host( request_ptr->context_id_ );
16:    properties_ = network_to_host( request_ptr->properties_ );
17: }
```

Figura 19 – Métodos `create_request` e `parse_request` da classe `dcl_message<msgCreateCommandQueue>`

3.1.6 Camada de rede

A camada de rede é responsável por enviar e receber os pacotes de mensagens. Ela é dividida em cliente e servidor e pode utilizar diferentes protocolos de transporte confiável, como o TCP. Também é responsabilidade da camada de rede o estabelecimento e manutenção da sessão.

No início da conexão, o cliente e o servidor estabelecem um acordo inicial (*handshake*) para a conexão, criando o identificador da sessão e os números de sequência das mensagens.

A descrição deste acordo inicial é tratada com mais detalhes na Seção 3.2.

No protótipo, o cliente e o servidor desta camada são construídos como classes *templates* que recebem como parâmetro a classe de tratamento do protocolo de rede, sendo hoje suportado apenas o TCP/IP. O servidor da camada de rede é construído como uma classe que abre uma porta de acesso e inicia uma *thread* para recebimento de conexões. A cada nova conexão estabelecida uma nova *thread* é criada para tratá-la, esta *thread* fica continuamente recebendo as mensagens e repassando para a camada de servidor, até o término da conexão. Uma vez terminada a conexão, a *thread* da conexão é terminada. O cliente desta camada é construído como uma classe que abre uma conexão com o servidor e mantêm esta conexão. O cliente oferece os serviços de enfileirar uma mensagem, para posterior envio, e o envio direto de uma mensagem. No entanto, quando é solicitado o envio de uma mensagem, sempre toda fila é enviada.

3.1.7 Camada de servidor

A camada de servidor é responsável por tratar as mensagens recebidas pela camada de rede e executá-las através de chamadas à camada de composição, de forma similar como o *NFS Server* invoca a camada *Virtual file system* na arquitetura do NFS quando recebe uma chamada remota.

Também é responsabilidade desta camada a decisão da execução síncrona ou assíncrona das mensagens recebidas. Para a maioria das mensagens recebidas, a execução é feita de forma síncrona. Porém, as mensagens relativas às funções `clSetKernelArg` e as funções para a fila de comando da API OpenCL (as funções `clEnqueue*`) são normalmente executadas de forma assíncrona. Este processo é mais bem detalhado na descrição de sua implementação no protótipo (Seção 4.3).

No protótipo, as classes da camada de servidor são correspondentes às classes da camada de mensagens, cada mensagem possui uma classe que irá tratá-la e chamar as classes necessárias da camada de composição para a execução da mesma. As classes desta camada utilizam o padrão de projeto *Command* do livro (GAMMA, HELM, *et al.*, 1995), recebendo o objeto com a mensagem no construtor e tendo somente o método `execute()` que trata a mensagem e salva o resultado da execução no próprio objeto recebido.

Para realizar esta execução assíncrona, o protótipo mantém uma *thread* de execução com duas filas de mensagens associadas, uma de entrada e outra de saída. Ao chegar uma mensagem que possa ser executada de forma assíncrona, a mesma é colocada na fila de entrada da *thread*, e o controle é devolvido para o cliente. A *thread* executa as mensagens da

fila de entrada e salva os resultados, se houverem, na fila de saída de mensagens. Em uma próxima chamada, a camada de servidor verifica as mensagens finalizadas na fila de saída de mensagens e retorna os resultados para o cliente.

Uma função no protótipo desta camada é apresentada na Figura 20, esta chamada cria um *kernel* a partir de um programa previamente compilado. A linha 4 carrega o identificador do programa passado na mensagem, que é traduzido para o ponteiro para o objeto do tipo `composite_program` na linha 5. Na linha 6 é criado o novo *kernel*, através de uma chamada à camada de composição e que é atribuído a ele um identificador na linha 8. Na linha 9 o identificador do *kernel* é colocado na mensagem para ser enviado como retorno para a camada de acesso remoto no cliente.

```
1:  void msgCreateKernel_command::execute()
2:  {
3:     server_platform& server = server_platform::get_instance();
4:     remote_id_t program_id = message_.get_program_id();
5:     composite_program* program_ptr =
          server.get_program_manager().get( program_id );
6:     generic_kernel* gen_krnl_ptr =
          program_ptr->create_kernel( message_.get_name() );
7:     composite_kernel* kernel_ptr =
          reinterpret_cast< composite_kernel* >( gen_krnl_ptr );
8:     remote_id_t id = server.get_kernel_manager().add( kernel_ptr );
9:     message_.set_remote_id( id );
10: }
```

Figura 20 – Método `msgCreateKernel_command::execute`

3.2 Protocolo de comunicação

A comunicação do *middleware* DistributedCL possui um protocolo de comunicação específico, tendo como um dos seus objetivos principais minimizar o uso da rede. Na Figura 21 está o diagrama de estado que descreve os estados do protocolo.

O servidor inicia no estado Fechado e abre uma porta para receber conexões, passando para o estado Escutando. A comunicação é iniciada pelo cliente, passando do estado Fechado para o Conectado, realizando o primeiro passo do acordo inicial (*handshake*), enviando a mensagem *Hello* para o servidor e passando ao estado Aguardando ACK. O servidor recebe a conexão, passando do estado Escutando para o Aguardando *Hello*, que ao receber a mensagem realiza o segundo passo do acordo inicial enviando a mensagem ACK, passando ao estado Aguardando Pacote. O cliente recebe o ACK passa para o estado Pronto, terminando o acordo inicial.

Posteriormente a cada transmissão o cliente envia um pacote que contém uma ou mais mensagens a serem executados no servidor, passando do estado Pronto para o Aguardando Resposta. O servidor recebe este pacote, passando de Aguardando Pacote para Aguardando

Execução, executa as mensagens síncronas na GPU, enfileira as mensagens assíncronas e retorna um pacote para o cliente, com as mensagens de retorno dos comandos executados via a biblioteca OpenCL, novamente voltando do estado Aguardando Execução para o Aguardando Pacote. O cliente recebe o pacote com as respostas enviadas pelo servidor e passa do estado Aguardando Resposta para o estado Pronto.

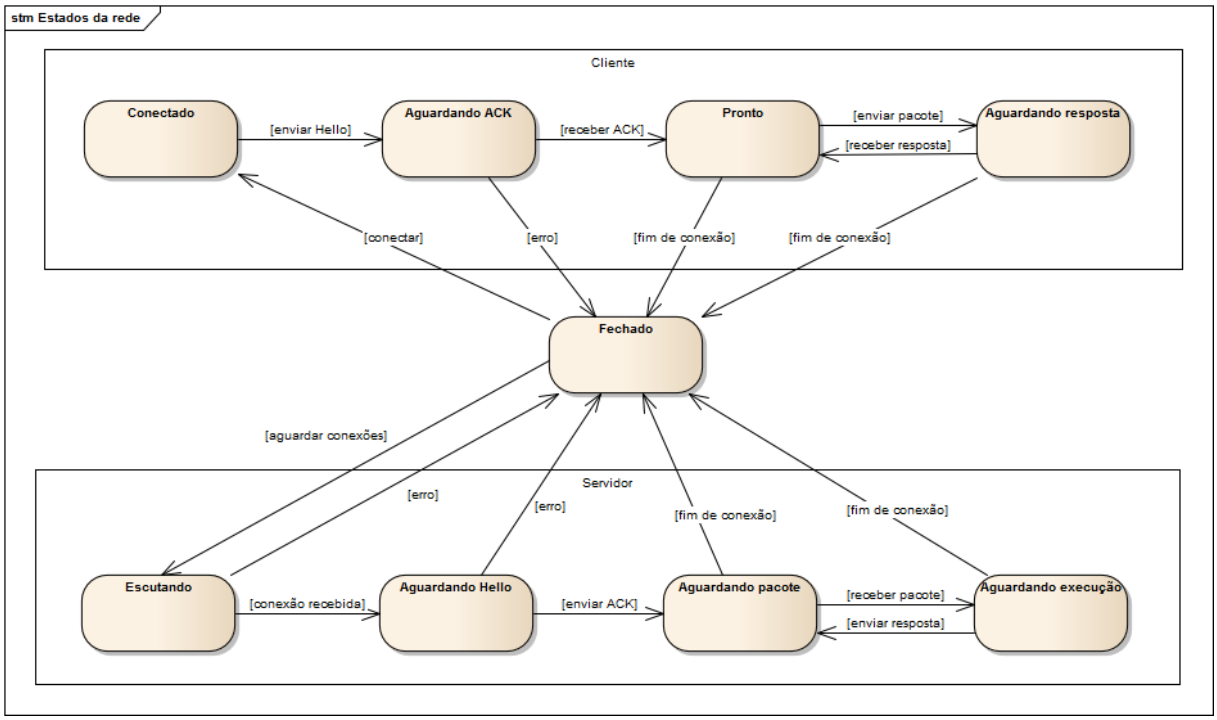

Figura 21 – Estados do protocolo de comunicação

3.2.1 Mensagem

Uma mensagem representa uma instrução entre o cliente e o servidor, em ambas as direções. O maior conjunto de mensagens existente é o das derivadas das funções da API OpenCL, mas também existem as mensagens de retorno, que são uma resposta do servidor às instruções enviadas pelo cliente do *middleware* DistributedCL, e as mensagens de controle, que são utilizadas em casos especiais, como no acordo inicial.

Todas as mensagens são estruturadas da mesma forma, possuem um cabeçalho (Figura 22) onde são informadas as características comuns e em seguida os campos específicos de cada mensagem.

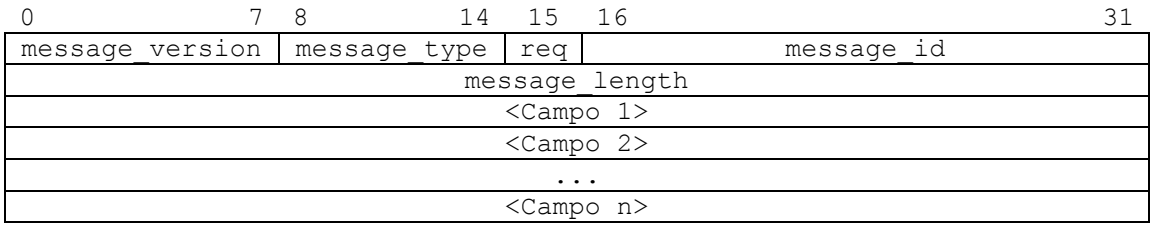

Figura 22 – Estrutura de uma mensagem

O `message_version` é um identificador que indica a versão do formato da mensagem, ele é usado para manter a compatibilidade futura de outros formatos de mensagem, este valor é fixo e igual a 16 (0x10). O `message_type` contém o tipo de mensagem que o cliente está solicitando ao servidor, ou que o servidor está respondendo ao cliente. O `req` indica se a mensagem é uma requisição ou uma resposta.

O `message_id` é um identificador da mensagem, ele é criado no cliente e retornado de volta com o mesmo valor na mensagem de resposta pelo servidor, e é utilizado pelo cliente para reconhecer mensagens que foram executadas de forma assíncrona no servidor. As respostas às mensagens com tratamento assíncrono não retornam imediatamente, sendo retornadas em outro momento pelo servidor, assim o cliente necessita armazenar as mensagens enviadas e que foram executadas de forma assíncrona. Ao receber outro pacote de mensagens de resposta, o cliente utiliza este identificador para reconhecer a mensagem enviada anteriormente e que estava pendente de resposta.

O `message_length` guarda o tamanho total da mensagem em bytes, incluindo o cabeçalho. Após o cabeçalho, os `campos 1` a `n` são utilizados por cada mensagem do protocolo, inserindo seus dados específicos, detalhados nas próximas subseções.

3.2.1.1 Mensagem `msgCreateImage2D`

A maioria das funções da API OpenCL possui uma mensagem equivalente para a serialização dos seus parâmetros e envio através da rede. Como exemplo de mensagem, está a mensagem `msgCreateImage2D`, que é relativa à função `clCreateImage2D` da API OpenCL.

A função `clCreateImage2D`, cujo formato é apresentado na Figura 23, é responsável por criar um objeto do tipo imagem na memória da GPU e iniciá-la com uma imagem passada. A mensagem `msgCreateImage2D`, que possui a estrutura apresentada na Figura 24, é utilizada pelo *middleware* DistributedCL para a transmissão dessa chamada do cliente para o servidor.

```
1:  cl_mem clCreateImage2D( cl_context context,
2:                          cl_mem_flags flags,
3:                          const cl_image_format *image_format,
4:                          size_t image_width,
5:                          size_t image_height,
6:                          size_t image_row_pitch,
7:                          void *host_ptr,
8:                          cl_int *errcode_ret);
```

Figura 23 – Assinatura da função `clCreateImage2D`

Cada parâmetro da função `clCreateImage2D` de alguma forma aparece na estrutura da `msgCreateImage2D`, porém a mensagem possui uma estrutura mais compacta para reduzir o tráfego através da rede. O mapeamento, apresentado na Tabela 3, mostra a relação entre os

parâmetros da função e os campos da estrutura, mostrando também o tamanho de cada um na memória.

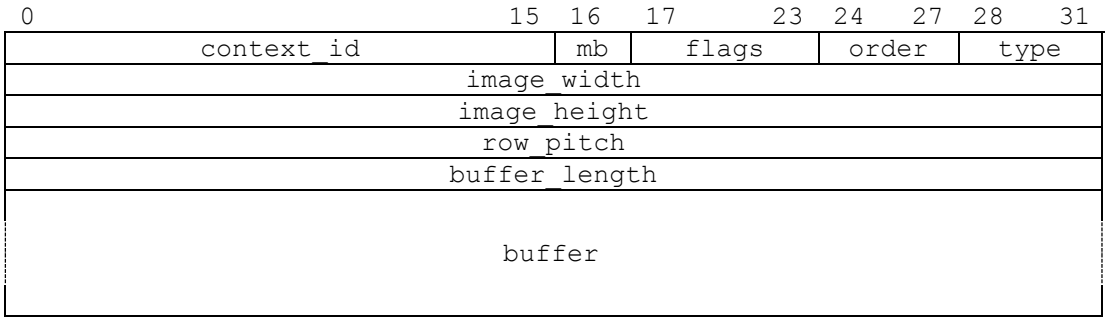

Figura 24 – Estrutura da mensagem `msgCreateImage2D`

Tabela 3 – Mapeamento entre `clCreateImage2D` e `msgCreateImage2D`

clCreateImage2D		**msgCreateImage2D**	
Parâmetro	**Tamanho**	**Campo**	**Tamanho**
`context`	32 ou 64 bits	`context_id`	16 bits
`flags`	64 bits	`flags`	7 bits
`image_format`	32 ou 64 bits	`order`	4 bits
		`type`	4 bits
`image_width`	32 ou 64 bits	`image_width`	32 bits
`image_height`	32 ou 64 bits	`image_height`	32 bits
`image_row_pitch`	32 ou 64 bits	`image_row_pitch`	32 bits
`host_ptr`	32 ou 64 bits	`mb`	1 bit
		`buffer_length`	32 bits
		`buffer`	Variável
`errcode_ret`	32 ou 64 bits	–	

O parâmetro `context` é o identificador do contexto da API OpenCL, ele é implementado como um ponteiro opaco à aplicação. Este ponteiro precisa ser traduzido para ser enviado como um identificador de rede, que na mensagem é o campo `context_id`.

O parâmetro `flags` é um inteiro de 64 bits, porém a função `clCreateImage2D` suporta apenas alguns valores, podendo ser representado em 7 bits na estrutura da mensagem.

O parâmetro `image_format` é um ponteiro para uma estrutura que descreve o formato da imagem e possui dois campos de 32 bits, os componentes da imagem (`channel_order`) e o tipo de dados (`channel_data_type`). Na estrutura da mensagem, este parâmetro é a fonte de dois campos, o `order` e `type`, ambos com 4 bits.

Os parâmetros `image_width`, `image_height` e `image_row_pitch` descrevem a largura, altura e o tamanho em bytes de uma linha da imagem respectivamente e são representados diretamente na estrutura. Vale notar que em ambientes de 64 bits estes parâmetros possuem 64 bits de tamanho, sendo que, na prática, 32 bits são suficientes para endereçar o tamanho máximo de imagem suportado pelas GPUs atuais, assim o *middleware* DistributedCL mantém

estes campos em 32 bits.

O parâmetro `host_ptr` contém o ponteiro para os dados da imagem a ser criada. Ele não pode ser simplesmente passado para o servidor pela rede, necessitando então que seus dados sejam copiados na mensagem. O campo `buffer` contém os dados da imagem a ser criada, e possui o tamanho passado pelo campo `buffer_length`. A API OpenCL faz distinção se o parâmetro `host_ptr` é nulo (`NULL`) ou se o tamanho da imagem é igual à zero. Para ter a mesma representação, a mensagem utiliza o campo `mb`, que indica se o `host_ptr` foi passado nulo ou não.

O parâmetro `errcode_ret` é utilizado como retorno da função não é representado na mensagem `msgCreateImage2D` que é utilizada para requisição, o código de retorno é transmitido pela mensagem de retorno desta chamada.

O tamanho do buffer apontado no parâmetro `host_ptr` não é passado como parâmetro da chamada e precisa ser calculado de acordo com os outros parâmetros da função. Para chegar ao valor do campo `buffer_length` o *middleware* DistributedCL utiliza a fórmula (1).

$$buffer\_length = image\_height * image\_row\_pitch \quad (1)$$

$$image\_row\_pitch = image\_width * element\_size \quad (2)$$

Tabela 4 – Tamanho de um elemento da imagem

element_size		channel_order												
		CL_R	CL_A	CL_RG	CL_RA	CL_Rx	CL_RGB	CL_RGx	CL_RGBA	CL_BGRA	CL_ARGB	CL_RGBx	CL_INTENSITY	CL_LUMINANCE
Channel_data_type	CL_UNORM_SHORT_565	2	2	2	2	2	2	2	2	2	2	2	2	2
	CL_UNORM_SHORT_555	2	2	2	2	2	2	2	2	2	2	2	2	2
	CL_UNORM_INT_101010	4	4	4	4	4	4	4	4	4	4	4	4	4
	CL_SNORM_INT8	1	1	2	2	2	3	3	4	4	4	4	4	4
	CL_UNORM_INT8	1	1	2	2	2	3	3	4	4	4	4	4	4
	CL_SIGNED_INT8	1	1	2	2	2	3	3	4	4	4	4	4	4
	CL_UNSIGNED_INT8	1	1	2	2	2	3	3	4	4	4	4	4	4
	CL_SNORM_INT16	2	2	4	4	4	6	6	8	8	8	8	8	8
	CL_UNORM_INT16	2	2	4	4	4	6	6	8	8	8	8	8	8
	CL_SIGNED_INT16	2	2	4	4	4	6	6	8	8	8	8	8	8
	CL_UNSIGNED_INT16	2	2	4	4	4	6	6	8	8	8	8	8	8
	CL_HALF_FLOAT	2	2	4	4	4	6	6	8	8	8	8	8	8
	CL_SIGNED_INT32	4	4	8	8	8	12	12	16	16	16	16	16	16
	CL_UNSIGNED_INT32	4	4	8	8	8	12	12	16	16	16	16	16	16
	CL_FLOAT	4	4	8	8	8	12	12	16	16	16	16	16	16

Entretanto, segundo a especificação da API OpenCL o parâmetro `image_row_pitch`

pode ser passado igual à zero, cabendo à biblioteca calculá-lo. Neste caso o tamanho de uma linha da imagem em bytes é calculado pela fórmula (2), utilizando a Tabela 4 de tamanho de um elemento de acordo com o `channel_data_type` e `channel_order`.

3.2.1.2 Mensagem de retorno

As funções da API OpenCL possuem quatro tipos de retorno: retorno de um objeto de contexto, retorno de informação sobre objeto de contexto, retorno de dados de objeto de memória e retorno de código de erro. Para cada um desses retornos, o *middleware* DistributedCL usa uma abordagem diferente para a mensagem de retorno.

As mensagens que retornam de um objeto de contexto, são geradas somente na criação dos objetos de contexto. Neste caso a mensagem de retorno é gerada contendo somente o identificador do objeto recém-criado.

Para mensagens que retornam informações sobre objeto de contexto, a mensagem de retorno contém todas as informações sobre o objeto, não só a solicitada pela chamada da API OpenCL. Assim é realizado um *cache* com todos os dados no cliente do *middleware* DistributedCL para resposta mais rápida à aplicação em uma chamada futura.

As mensagens que retornam dados de objetos de memória são utilizadas como resposta das chamadas `clEnqueueRead*` de transferência de dados da GPU e para o *host*. Neste caso a mensagem de retorno contém o tamanho do buffer e todos os dados para transmissão via rede.

Praticamente todas as funções da API OpenCL possuem um código de retorno. Porém, nem todas as mensagens do *middleware* DistributedCL retornam mensagens com códigos de erro. No caso das mensagens anteriores o retorno do resultado é o suficiente para o cliente identificar que o comando foi executado com sucesso e retornar esta informação para a aplicação.

As mensagens com código de erro só são retornadas em dois casos: quando a uma das chamadas à API OpenCL não foi bem sucedida ou quando todas as mensagens de um pacote não tinham retorno de informação, dados ou do objeto de contexto. Em ambos os casos é retornada a mensagem `msg_error_message` contendo o código de erro, no primeiro caso o erro retornado pela API OpenCL e no segundo o código `CL_SUCCESS` indicando que todos os comando do pacote foram executados com sucesso. É importante notar que somente uma mensagem com código de erro é retornada, mesmo que o pacote de requisição contenha muitas mensagens.

3.2.1.3 Mensagem de controle

O *middleware* DistributedCL também possui mensagens de controle que são utilizadas em situações específicas. Estas mensagens estão listadas na Tabela 5, assim como a sua utilização.

Tabela 5 – Mensagens de controle do *middleware* DistributedCL

Mensagem	**Descrição**
`msg_handshake_hello`	Mensagem do acordo inicial enviada pelo cliente.
`msg_handshake_ack`	Mensagem do acordo inicial retornada pelo servidor.
`msg_error_message`	Mensagem de erro retornada pelo servidor.
`msg_flush_server`	Mensagem enviada pelo cliente para solicitar ao servidor que execute toda a lista de comandos pendente. Esta mensagem é enviada de forma assíncrona.
`msg_get_context`	Mensagem enviada pelo cliente para solicitar o identificador interno do contexto e preparar o servidor para a criação de outra conexão para uma nova fila de comando.
`msg_attach_context`	Mensagem enviada pelo cliente para o servidor identificar que a conexão é relativa a um contexto existente.

3.2.2 Pacote

No protocolo de comunicação do *middleware* DistributedCL, o conjunto de mensagens enviadas em uma única transmissão de rede é chamado de pacote. No protocolo sempre é enviado um pacote de mensagens, mesmo no caso do envio de uma única mensagem.

Um pacote contém um cabeçalho de controle e um conjunto de mensagens que serão transmitidas entre o cliente e o servidor do *middleware* DistributedCL. A estrutura básica de um pacote está apresentada na Figura 25.

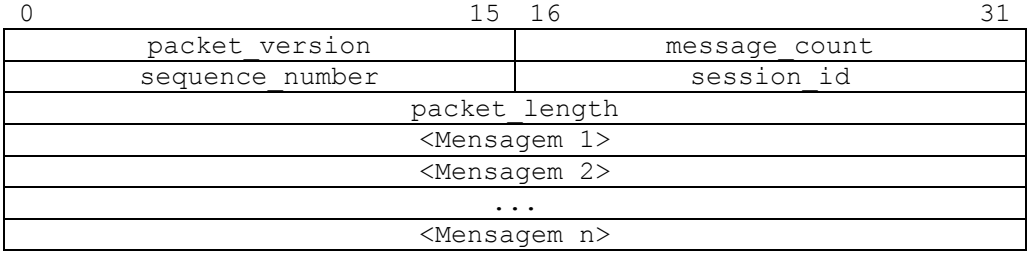

Figura 25 – Estrutura de um pacote

O `packet_version` é um identificador que indica a versão do formato do pacote, ele é usado para manter a compatibilidade futura de outros formatos de pacote, este valor é fixo e igual a 16 (0x10). O `message_count` contém o número total de mensagens no pacote. O `sequence_number` contém o número de sequência do pacote, que é acrescido de um em cada envio. O `session_id` contém a identificação da sessão, que é único para cada conexão. O `packet_lenght` contém o tamanho total do pacote em bytes.

Ao receber um pacote, tanto o cliente quanto o servidor verificam os campos

`sequence_number` e `session_id` para certificar que o pacote está na ordem esperada e que a sessão está correta.

Após o cabeçalho, as `Mensagens 1` a `n` são colocadas na ordem que devem ser tratadas e com a estrutura descrita na Subseção 3.2.1.

3.2.3 Acordo inicial (*handshake*)

Como todas as transmissões do *middleware* DistributedCL, o acordo inicial é realizado através de uma troca de mensagens de controle. O acordo inicial é composto de dois passos, uma requisição do cliente e uma resposta do servidor.

O acordo é iniciado pelo cliente, após o estabelecimento da conexão, que envia a mensagem `msg_handshake_hello`, com a estrutura descrita na Figura 26. O campo `protocol_version` é um identificador que indica a versão do protocolo, ele é usado para manter a compatibilidade futura de outros formatos de mensagem, este valor é fixo e igual a 16 (0x10). O campo `client_sequence_number` é o número de sequência dos pacotes e é escolhido aleatoriamente pelo cliente.

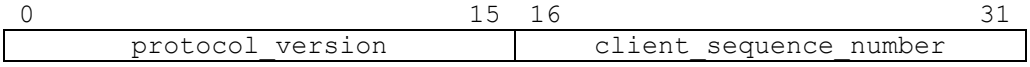

Figura 26 – Mensagem `msg_handshake_hello`

O servidor recebe essa mensagem, verifica se possui suporte para o protocolo solicitado, guarda o número de sequência do cliente e responde com a mensagem `msg_handshake_ack`, com a estrutura descrita na Figura 27, indicando que a conexão foi estabelecida. De forma similar ao cliente, o servidor escolhe aleatoriamente o número de sequência inicial e o identificador da sessão para retornar esses dados nos campos `server_sequence_number` e `session_id` respectivamente. O cliente recebe os dados do servidor, guarda o número de sequência do servidor e o número de sessão.

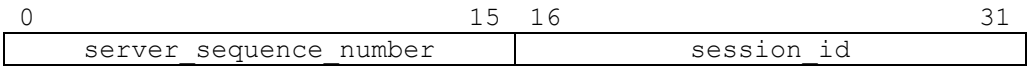

Figura 27 – Mensagem `msg_handshake_ack`

Caso o servidor não reconheça a mensagem de requisição ou não suporte o formato do protocolo, ou o cliente não reconheça os dados da resposta, a conexão é encerrada.

3.2.4 Conexão para filas de comando

Além da conexão inicial, o cliente do *middleware* DistributedCL abre uma nova conexão com o servidor para cada fila de comando criada pela aplicação. Isto é feito para

aumentar o paralelismo entre as filas de comando, tanto no cliente como no servidor, com o objetivo de melhorar o desempenho global. Esta nova conexão faz parte do mesmo contexto OpenCL da conexão anterior e deve compartilhar os dados existentes, porém, como o servidor não tem nenhuma informação que ele possa usar para associar esta nova conexão a um contexto existente, cabe ao cliente passar esta informação ao servidor.

Neste caso o cliente envia a mensagem `msg_get_context` na conexão original para resgatar o identificador interno do contexto e repassa este identificador na nova conexão através da mensagem `msg_attach_context`. Ao receber a mensagem `msg_attach_context`, o servidor valida o identificador interno e passa a considerar a nova conexão como sendo do mesmo contexto OpenCL da conexão original.

A partir da criação da nova conexão para envio de comandos para a fila, as mensagens relativas às funções `clSetKernelArg`, `clFlush`, `clFinish`, `clWaitEvent` e todas `clEnqueue*` da API OpenCL são enviadas por esta nova conexão. As mensagens relativas às outras funções continuam sendo transmitidas pela conexão original.

4 PROTÓTIPO DO *MIDDLEWARE* DISTRIBUTEDCL

O protótipo do *middleware* DistributedCL (TUPINAMBÁ e SZTAJNBERG, 2012) implementa a arquitetura descrita no Capítulo 3 e procura aproveitar as características da API OpenCL descritas na Subseção 1.2.1. Por exemplo, o protótipo faz uso de *cache* para as informações da API de plataforma, que são constantes, e possui processamento assíncrono específico para as filas de comando da API de comunicação, que são assíncronas pelo padrão OpenCL.

O protótipo é escrito na linguagem C++, com a preocupação em ser portável nos ambientes Windows e Linux e possuir bom desempenho. A comunicação utiliza o protocolo descrito na Seção 3.2 e utiliza o transporte de dados via TCP/IP. Ele é distribuído gratuitamente como software livre através do site *Sourceforge* (TUPINAMBÁ, 2011).

4.1 Estrutura do código

Uma série de direcionamentos norteia a construção do protótipo do *middleware* DistributedCL. O primeiro é a compatibilidade entre os sistemas operacionais Windows e Linux, em ambientes de 32 e 64 bits. O segundo é a modularidade de suas camadas, onde as responsabilidades e as interfaces são bem definidas. E o terceiro é o desempenho do código, seguindo a arquitetura proposta.

Para ser compatível com processadores de várias arquiteturas, vários sistemas operacionais e ainda possuir um bom desempenho, o protótipo utilizou a linguagem C++. No entanto, esta linguagem, na sua versão C++03, não fornece ferramentas suficientes para abstrair as diferentes bibliotecas e configurações específicas de cada ambiente.

Utilizar comandos do pré-processador, como o `#ifdef`, resolvem alguns problemas mais simples, mas para o uso de funções específicas de cada ambiente, como o de *threads* e de comunicação entre processos, é mais interessante utilizar classes específicas. Para endereçar essas questões de ambiente, o protótipo utiliza as bibliotecas Boost[1] (DAWES, ABRAHAMS e RIVERA, 2004), que possui esse suporte.

Outro ponto de atenção é sobre a arquitetura do processador e sistema operacional. Para a questão de arquitetura do processador, o código do protótipo não utiliza os tipos típicos `int` e `long`, dando preferência os novos tipos definidos pelo C99 que indicam exatamente o

[1] A Boost é um conjunto de bibliotecas que provêm uma série de funcionalidades, entre elas *smart pointers*, tratamento de data e hora, *threads* e comunicação entre processos. Em 2011 algumas de suas bibliotecas foram incorporadas à biblioteca padrão do C++11.

número de bit, como o `int32_t` e `uint64_t`.

Uma característica importante da codificação está na estrutura das classes das camadas de acesso local, acesso remoto e de composição. Essas três camadas oferecem o mesmo serviço, prover as características da API OpenCL através de suas classes. Para isso, suas classes herdam das mesmas classes abstratas, mantendo uma interface nas classes dessas três camadas. Assim todo o código trata o acesso de forma genérica sem a necessidade de conhecer qual camada está sendo efetivamente executada: de acesso local, de acesso remoto ou de composição. Como exemplo, a Figura 28 apresenta a classe abstrata `generic_program` que possui três descendentes, a classe `program` na camada de acesso local, a classe `remote_program` na camada de acesso remoto e a classe `composite_program` na camada de composição.

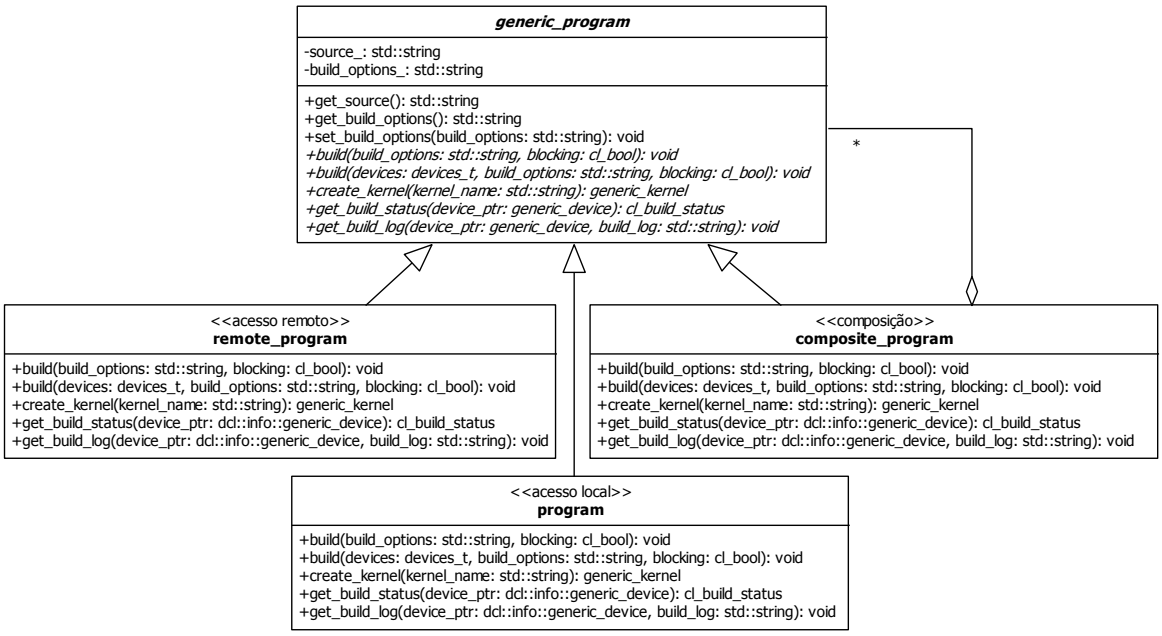

Figura 28 – Herança das classes que representam um programa OpenCL

4.2 Instalação

A instalação e configuração do protótipo se dão de maneiras diferentes no cliente e no servidor. Para a instalação do cliente é necessário que a aplicação carregue a biblioteca cliente do protótipo ao invés da biblioteca OpenCL do sistema, para isso existem diferentes mecanismos nos sistemas operacionais Windows e Linux. Por não estar previsto no padrão OpenCL, o servidor é mais flexível e pode ser mais facilmente instalado e configurado.

4.2.1 Cliente

Para manter a compatibilidade com a plataforma OpenCL, o cliente do protótipo deve utilizar mecanismos de acesso similares às de outras bibliotecas OpenCL.

O cliente do protótipo é uma biblioteca dinâmica, um *shared object* (`.so`) no Linux ou uma *dynamic link library* (`.dll`) no Windows. Porém, apesar de suas estruturas internas estarem preparadas, ele não é totalmente compatível com a extensão ICD (KHRONOS GROUP, 2011) da OpenCL não podendo utilizar sua forma de instalação.

Para que a aplicação carregue a biblioteca cliente, ao invés da biblioteca OpenCL existente no sistema, é necessário realizar configurações no ambiente, que são diferentes no Windows e no Linux. No ambiente Windows, o cliente do protótipo deve ser copiado para o mesmo diretório da aplicação e deve possuir o nome `OpenCL.dll`. No Linux, o cliente deve ter o nome de `libOpenCL.so.1.0.0`, com os links `libOpenCL.so.1` e `libOpenCL.so`, e a variável de ambiente `LD_LIBRARY_PATH` deve ser inicializada com o diretório onde se encontra o arquivo.

Para ser compatível com a plataforma OpenCL, o cliente do protótipo não pode ter configuração específica na API. No entanto, ele precisa ser informado quais os servidores deverá se conectar e se deve ou não utilizar as outras bibliotecas OpenCL disponíveis no sistema. A configuração é feita através de um arquivo de configuração, como o da Figura 29.

```
1: local=0
2: server=10.0.0.2:4791
3: server=10.0.0.3:4791
4: server=10.0.0.4:4791
```

Figura 29 – Configuração da biblioteca cliente do protótipo

Na linha 1 a variável `local` indica se o cliente deve utilizar as bibliotecas OpenCL existentes no sistema. Os valores possíveis são 0 e 1, caso seja 0 não devem ser utilizadas as bibliotecas locais, caso seja 1 elas serão utilizadas. As linhas 2 a 4 estão as configurações da variável `server`, que indica quais servidores o cliente deverá se conectar. Podem existir várias linhas para a variável `server`, e cada uma possui o formato `IP:PORTA`. No exemplo da Figura 29 está a configuração com três servidores.

O arquivo deve possuir o nome `libdcl.conf` e ser gravado em diretório específico. No Windows ele deve ser gravado no diretório `%ALLUSERSPROFILE%\DistributedCL`, que no Windows 7 fica `C:\ProgramData\DistributedCL`. No Linux o mesmo arquivo deve ser gravado no diretório `/etc/dcl`.

4.2.2 Servidor

Por não ser previsto na plataforma OpenCL, o servidor do protótipo possui maior liberdade para controlar sua configuração. O servidor é um programa executável e pode ser gravado em qualquer diretório em ambos ambientes Windows e Linux.

Sua configuração pode ser feita através de arquivo de configuração, de forma similar ao cliente. O arquivo deve possuir o nome `dcld.conf`, estar gravado nos diretórios descritos para o arquivo de configuração do cliente e possuir o formato descrito na Figura 30.

```
1: local=1
2: port=4791
3: server=10.0.0.2:4791
4: server=10.0.0.3:4791
5: server=10.0.0.4:4791
```

Figura 30 – Configuração do servidor *middleware* DistributedCL

As variáveis `local` e `server` possuem o mesmo significado que a configuração do cliente do protótipo do *middleware* DistributedCL, logo em um servidor é possível acessar outros servidores do protótipo, porém esta situação não é explorada neste trabalho e pode ser tema de trabalhos futuros.

Na linha 2, a variável `port` indica qual a porta do TCP/IP que o servidor deverá esperar as conexões vindas dos clientes.

Por se tratar de um programa executável, é possível também passar os parâmetros de configuração através da linha de comando. Os parâmetros estão na Figura 31 e são equivalentes aos do arquivo de configuração.

```
1: bash$ dcld --help
2: DistributedCL Server 0.1
3: Loading config file: /etc/dcl/dcld.conf
4: DistributedCL server parameters:
5:   -h [ --help ]             Show this message
6:   -l [ --local ] arg        Load local OpenCL libraries
7:   -s [ --server ] arg       Connect DistributedCL servers (ip:port)
8:   -p [ --port ] arg (=4791) Connection port
9: bash$
```

Figura 31 – Parâmetros do servidor do *middleware* DistributedCL

4.3 Otimizações inseridas na implementação do protótipo

O protótipo do *middleware* DistributedCL possui uma série de otimizações aproveitando o conhecimento da API OpenCL e seu comportamento. A principal otimização é tratamento assíncrono de algumas funções da API, pois possibilita o paralelismo entre a execução do programa na CPU, a transmissão dos dados via rede e o processamento nas GPUs remotas.

No padrão OpenCL a execução das funções `clEnqueue*` é feita de forma assíncrona na fila de comando, assim o protótipo tenta melhorar sua resposta através de tratamento assíncrono dessas funções. Os dois pontos principais de tratamento assíncrono no protótipo são o envio das mensagens em lotes, no cliente, e a execução assíncrona das mensagens relativas a essas funções do OpenCL, no servidor.

Também fazem parte das otimizações as estruturas compactas de mensagens, *cache* de dados no cliente, comunicação sem cópia de dados entre as camadas e conexão de rede exclusiva para cada fila de comando.

Todas as características descritas nesta seção são exclusivamente de controle interno do protótipo, logo totalmente transparente para a aplicação. Assim todas as otimizações estão disponíveis para qualquer aplicação que tenha suporte à API OpenCL, sem necessidade de alteração no código ou recompilação.

4.3.1 Envio de mensagens em lotes

O protótipo do *middleware* DistributedCL posterga o envio de algumas mensagens através da rede, enfileirando-as para posterior envio. Do ponto de vista da aplicação, o cliente do protótipo tem um retorno rápido, pois nenhuma transmissão de dados ou execução foi realizada, somente a guarda dos dados para posterior envio. Em relação à rede, minimiza-se o uso, pois os dados das funções chamadas pela aplicação são enviados em conjunto do cliente para o servidor, na forma de mensagens, em uma única transmissão de rede. As transmissões via rede são efetivamente efetuadas quando o cliente recebe uma chamada de função que necessita de uma resposta imediata, não podendo ser postergada.

As chamadas da API de comunicação do OpenCL possuem a característica de terem o processamento efetuado de forma assíncrona. Portanto, o protótipo procura aproveitar o comportamento esperado de cada função para postergar sua transmissão via rede, e por consequência sua execução.

As funções que utilizam o envio de mensagens em lotes são `clSetKernelArg` e as funções `clEnqueue*`. As funções `clFlush` e `clFinish` são utilizadas para solicitar o envio das mensagens. As funções da API de plataforma e API de contexto do OpenCL necessitam de resposta imediata, portanto não é possível utilizar envio em lotes.

A função `clSetKernelArg` é chamada para a passagem de parâmetros para um *kernel*. Esta função informa à biblioteca OpenCL os parâmetros que deverão ser passados para o *kernel* quando a aplicação solicitar a sua execução. Neste caso, o protótipo armazena essa informação no cliente e somente envia as mensagens relativas a essa função no momento do

envio da mensagem da função `clEnqueueNDRangeKernel` ou `clEnqueueTask` respectiva ao *kernel*.

As funções chamadas genericamente de `clEnqueue*` são responsáveis por enviar comandos aos dispositivos através da fila de comando do OpenCL. Essas funções possuem execução assíncrona por padrão e são armazenadas pelo protótipo para envio em lote. No entanto, em alguma dessas funções a decisão do armazenamento ou do envio imediato da mensagem é feito através do parâmetro `blocking` existente na chamada dessas funções que indica se ela deve ser síncrona ou assíncrona.

Caso o parâmetro `blocking` seja passado falso, o protótipo trata a função como assíncrona do mesmo modo das outras funções. Caso o parâmetro seja passado verdadeiro, o cliente enfileira a mensagem relativa à função e aciona a camada de rede para enviar todas as mensagens pendentes em um só pacote de forma síncrona.

As funções `clFlush` e `clFinish` são utilizadas pela aplicação para solicitar a execução da fila de comando pela biblioteca OpenCL. No protótipo estas funções disparam o envio das mensagens armazenadas no cliente para a execução no servidor, porém o comportamento de ambas é diferente. A `clFlush` força o envio das mensagens de forma assíncrona, ou seja, o cliente retorna imediatamente o controle para a aplicação enquanto envia as mensagens armazenadas através de uma *thread* separada. Enquanto na função `clFinish` o cliente envia todas as mensagens armazenadas de forma síncrona, esperando o retorno do servidor.

4.3.2 Execução assíncrona de mensagens

O protótipo do *middleware* DistributedCL também procura postergar a execução dos comandos no servidor para terem sua execução em paralelo à transmissão de dados via rede. Assim, o servidor tem uma resposta mais rápida ao cliente, pois a execução ainda não foi realizada na GPU, somente a guarda dos dados para posterior execução.

O servidor realiza o tratamento das mensagens relativas às funções `clSetKernelArgs` e as `clEnqueue*` de forma assíncrona, ou seja, envia imediatamente uma mensagem de retorno ao cliente enquanto executa os comandos na GPU relativo às mensagens recebidas através de uma *thread* separada.

Para manter a ordem das mensagens, o protótipo mantém uma fila FIFO de mensagens no servidor. Esta fila é alimentada pelas mensagens recebidas pelo servidor e é consumida por uma *thread* específica para a execução das mesmas. Da mesma forma que o envio assíncrono

de mensagens, a exceção é feita às mensagens às funções que possuem o parâmetro `blocking` da API OpenCL passado pela aplicação e transmitido ao servidor.

Caso o parâmetro `blocking` seja passado falso, o protótipo trata a mensagem com o mesmo tratamento assíncrono. Caso seja passado verdadeiro, todas as mensagens pendentes na fila são executadas antes, a mensagem é executada de forma síncrona e uma mensagem de retorno é enviada para o cliente.

4.3.3 Processamento assíncrono

Combinando o envio de mensagens em lotes e a execução assíncrona das mensagens, temos a situação que o protótipo retorna o controle rapidamente para a aplicação, que pode continuar seu processamento paralelamente ao envio das mensagens ao servidor e a execução na GPU. O retorno do controle à aplicação de forma rápida permite que a mesma possa enviar mais comandos à GPU enquanto o protótipo está tratando as mensagens anteriores. Este modelo pode ser exemplificado com o diagrama de sequência da Figura 32.

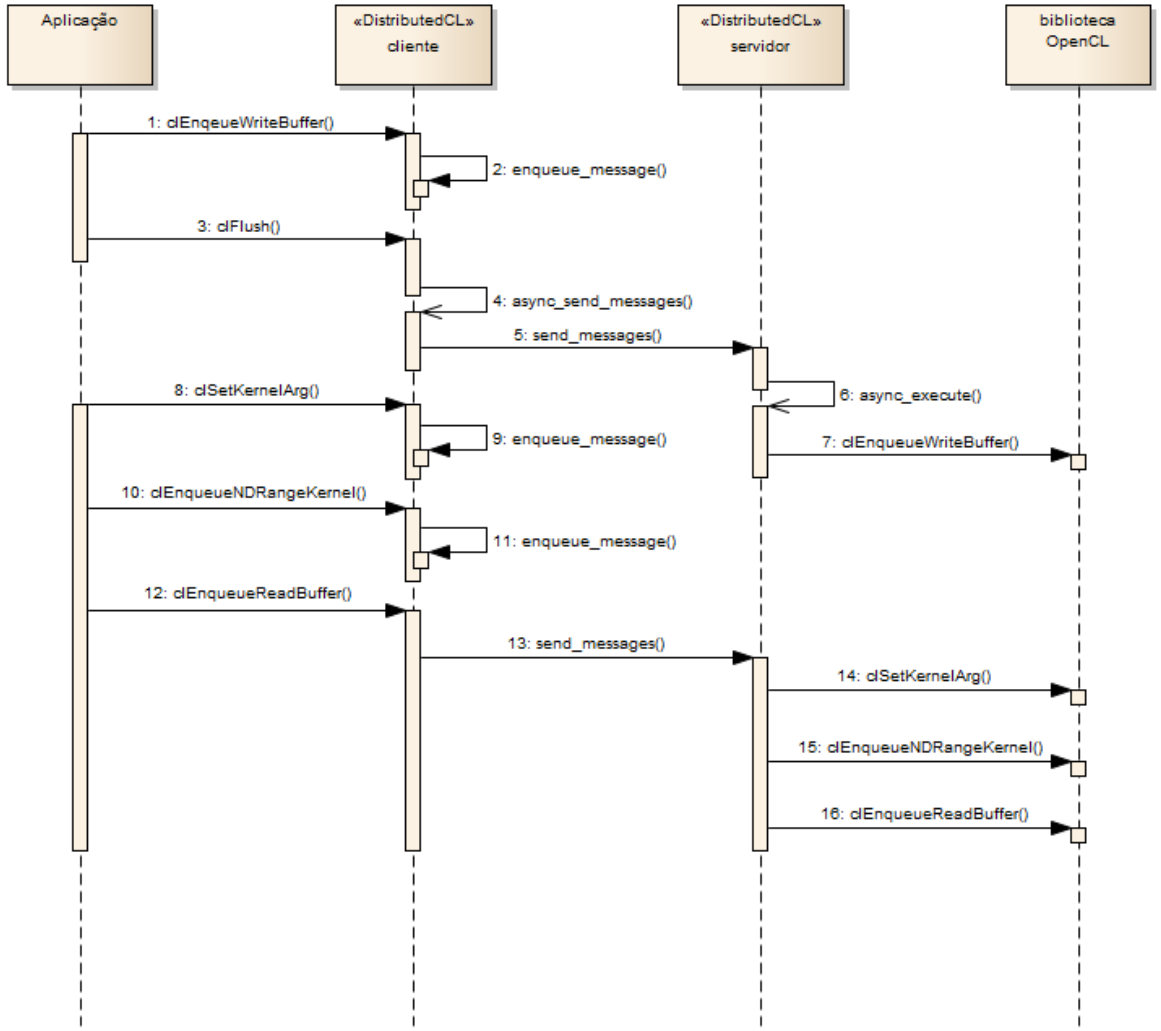

Figura 32 – Diagrama de sequência do processamento assíncrono

O diagrama mostra uma sequência típica de execução de uma aplicação OpenCL passando pelo protótipo. Ele descreve uma aplicação fazendo a gravação um buffer de memória da GPU, a execução de um *kernel* para tratar estes dados e a posterior leitura dos resultados.

O processamento se inicia no passo 1, quando a aplicação chama o cliente do protótipo solicitando a cópia de dados para a memória da GPU e o cliente, no passo 2, guarda esta informação na forma de uma mensagem. A aplicação chama novamente o cliente no passo 3 solicitando que os comandos enfileirados na fila de comando sejam executados, o que faz que o cliente execute a *thread* de envio assíncrono de dados no passo 4, retorne o controle para a aplicação e efetivamente envie as mensagens de forma assíncrona para o servidor do protótipo, no passo 5. O servidor recebe a mensagem e aciona a *thread* de execução assíncrona de mensagens no passo 6, retornando o controle ao cliente. Em paralelo, a *thread* efetivamente chama a biblioteca OpenCL no passo 7, repassando o comando recebido da aplicação.

Entretanto, após o passo 4 o controle é retornado à aplicação que pode então continuar seu processamento e realizar outra solicitação ao cliente. No passo 8 a aplicação solicita a configuração de um parâmetro de execução do *kernel* e o cliente enfileira a mensagem no passo 9. Logo em seguida a aplicação solicita a execução do *kernel* no passo 10 e o cliente novamente enfileira a mensagem no passo 11. No passo 12 a aplicação solicita então a leitura do buffer da GPU de forma síncrona, o que força o envio em lote de todas as mensagens armazenadas pelo cliente no passo 13, gerando a execução das chamadas equivalentes nos passos 14, 15 e 16 pelo servidor.

4.3.4 Outras otimizações

Outras características também contribuem para melhorar o desempenho do protótipo. As estruturas compactas de mensagens e o *cache* de dados no cliente auxiliam na diminuição do tráfego de rede, a comunicação sem cópia de dados entre as camadas torna o código mais eficiente e conexão de rede exclusiva para cada fila de comando aumenta o paralelismo entre as filas.

As estruturas das mensagens possuem a preocupação de ocupar o mínimo de espaço, mas mantendo todos os dados necessários para a transmissão das funções recebidas pelo protótipo. Como as estruturas das mensagens são reduzidas, existem menos dados a serem transmitidos pela rede, reduzindo assim o tráfego de rede.

As informações sobre as plataformas e dispositivos são constantes e são armazenadas em um *cache* no cliente do protótipo. Assim as chamadas da aplicação para o retorno dessas informações são efetuadas localmente, sem necessidade de transmitir mensagens através da rede.

Segundo (SVOBODOVA, 1989), a cópia de buffers entre as camadas de um protocolo não é boa estratégia, assim o código do protótipo foi criado de forma que a cópia de dados é evitada. Esta estratégia é usada principalmente em dois casos, o primeiro nas chamadas das camadas de acesso local, acesso remoto e composição que utilizam sempre os ponteiros passados, sem efetuar a cópia dos dados. O segundo é a montagem dos pacotes de transmissão de dados que são feitos de forma a ter somente uma cópia. Neste caso o buffer de transmissão de dados é alocado pela camada de rede, que passa o ponteiro para a classe de tratamento do pacote para preencher os seus dados. Por sua vez, a classe de tratamento do pacote passam ponteiros para a mesma área de memória para cada mensagem preencher seus dados específicos na posição correta para transmissão, eliminando assim a necessidade de processamento posterior.

Como definido no protocolo de comunicação, na Subseção 3.2.4, o protótipo cria uma nova conexão do cliente para o servidor para cada fila de comando criada pela aplicação. Esta nova conexão permite que cada fila de comando seja utilizada de forma independente, de forma paralela.

5 AVALIAÇÃO DE DESEMPENHO

Neste capítulo estão apresentadas as avaliações de desempenho realizadas com o protótipo do *middleware* DistributedCL proposto neste trabalho. O laboratório de GPUs[2] do Laboratório do Instituto de Matemática e Estatística da UERJ (LabIME) foi a infraestrutura utilizada para os testes, salvo indicado, todos os testes utilizaram este ambiente.

As aplicações utilizadas nestes testes foram o SHOC, LuxMark, BFGMiner e CLBench, sendo que este último construído especialmente para testes com o protótipo *middleware* DistributedCL. Os testes são compostos pela comparação da execução dessas aplicações acessando a GPU local do computador e através do protótipo com diversas quantidades de GPUs remotas. Além disso, o desempenho do protótipo é comparado com o rCUDA e Mosix VirtualCL que possuem propostas similares.

5.1 Configuração dos testes

O laboratório de GPUs do LabIME é composto por oito computadores com a mesma configuração. Cada computador possui: placa mãe Intel DX58SO, processador Intel Core i7-940, 8GB DDR3 de memória principal, placa de vídeo NVIDIA GeForce GTX 480 com 1.5GB GDDR5, disco rígido Seagate ST31000528AS com capacidade para 1TB e duas placas de rede *Ethernet gigabit*. Em cada computador está instalado o Linux Ubuntu 12.04 *long-term support* 64bits, *kernel* 3.2.0-36, os *drivers* para o vídeo da NVIDIA versão 304.43 e com os seus outros pacotes nas versões mais atuais. Uma placa de rede dos computadores está ligada à rede da UERJ, enquanto a outra está interligando somente os computadores do laboratório através de um *switch* de rede *gigabit* 3Com 3C1671600A.

5.2 SHOC

O *Scalable HeterOgeneous Computing benchmark suite* (SHOC) (DANALIS, MARIN, *et al.*, 2010) é um *benchmark* para OpenCL que utiliza um conjunto de testes de desempenho, listados na Tabela 6, avaliando diferentes características do dispositivo testado. Ele tem suporte às plataformas CUDA e OpenCL, podendo acessar qualquer dispositivo deste.

Os testes com o protótipo do *middleware* DistributedCL utilizam todos os testes do SHOC v1.1.4, com a menor configuração de tamanho e executado 50 vezes cada teste. O resultado dos testes executando em uma GPU remota e diretamente na GPU local sem passar

[2] O laboratório de GPUs do LabIME/IME/UERJ foi disponibilizado pelo projeto Cooperação entre as Pós-Graduações de Computação Científica LNCC-UERJ / Faperj

pelo protótipo, são comparados. Na Figura 33 está a tela do SHOC, com a execução do teste com uma GPU remota.

Tabela 6 – Testes do SHOC

Teste	**Descrição**
BusSpeedDownload	Testa a transferência de dados do *host* para o dispositivo.
BusSpeedReadback	Testa a transferência de dados do dispositivo para o *host*.
MaxFlops	Testa a capacidade em FLOPS do dispositivo.
DeviceMemory	Testa a transferência de dados dentro do dispositivo.
KernelCompile	Testa a compilação de diferentes *kernels*.
QueueDelay	Testa a latência das filas de comando.
BFS	Testa o desempenho do algoritmo de busca em largura em grafos (*breadth-first search*).
FFT	Testa o desempenho de uma transformada rápida de Fourier 1D.
SGEMM	Testa o desempenho de uma rotina SGEMM do BLAS.
MD	Testa o desempenho de um potencial de Lennard-Jones de dinâmica molecular.
Reduction	Testa o desempenho de uma redução em soma.
Scan	Testa o desempenho de uma soma cumulativa.
Sort	Teste o desempenho de um *radix sort*.
Spmv	Testa o desempenho de uma multiplicação entre uma matriz esparsa e um vetor denso.
Stencil2D	Testa o desempenho de uma aplicação de filtro 2D de 9 pontos.
Triad	Versão para GPU do teste Stream (MCCALPIN, 1995) Triad de largura de banda.
S3D	Testa o desempenho da simulação de uma combustão turbulenta.

5.2.1 Resultados

Os resultados dos testes do SHOC com ponto flutuante de precisão simples estão na Tabela 7, comparando os resultados obtidos com a GPU local e com uma GPU remota. O intervalo de confiança é de 95%.

Os testes BusSpeedDownload e BusSpeedReadback, no maior buffer de dados, apresentam um menor desempenho na execução na GPU remota em relação à GPU local. Este resultado é esperado, pois este teste mede a velocidade de transmissão de dados entre o *host* e a GPU, que na GPU remota é acrescido do tempo de transmissão via rede.

Os testes MaxFlops e DeviceMemory apresentam um resultado similar entre a GPU local e remota, o que é condizente com a expectativa, pois o teste se baseia em pontos específicos e internos, sem analisar a transmissão de dados entre o *host* e a GPU.

No teste KernelCompile houve uma grande queda no desempenho, com a eficiência do protótipo sendo somente 3,4%. Esta queda é esperada, pois na configuração local o código é

compilado totalmente na CPU do *host* e não há nenhuma transmissão de dados envolvida, enquanto no protótipo é necessária à transmissão do programa via rede para ser compilado. No entanto, esta queda de desempenho não é significativa na execução normal de uma aplicação OpenCL, porque normalmente este passo é realizado apenas uma vez na aplicação para preparar o *program*. Após este passo os *kernels* resultantes podem ser executados diversas vezes, sem necessidade de passar novamente pelo processo de compilação. Portanto, apesar do tempo de compilação do *kernel* ser maior, o impacto geral em uma aplicação é reduzido.

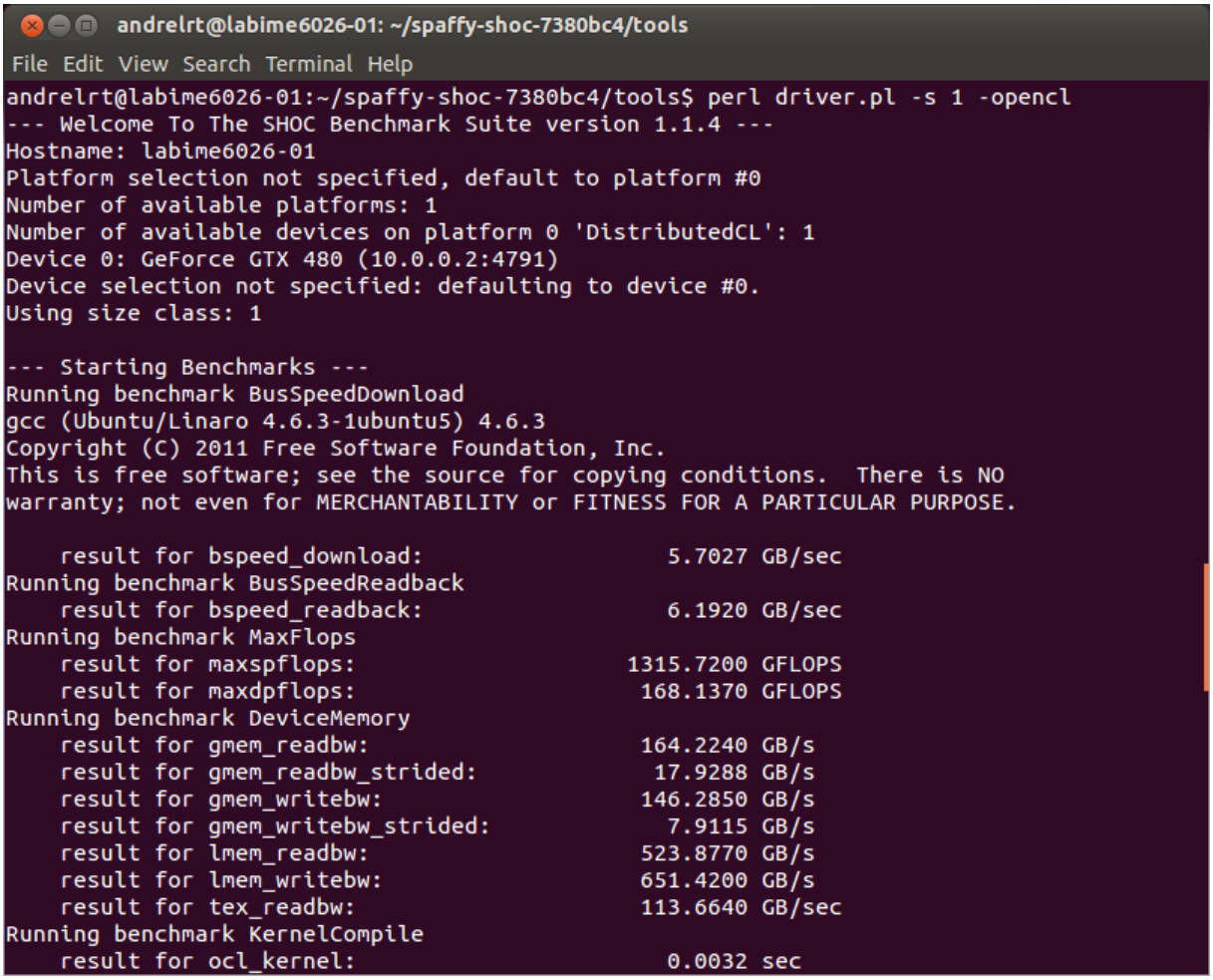

Figura 33 – Tela do SHOC rodando com uma GPU remota

O teste QueueDelay apresenta uma eficiência de 85,5%. Esta queda no desempenho é esperada, pois além da latência da fila, existe também à latência de rede.

Os testes FFT, SGEMM, MD e S3D apresentam eficiência superior a 97% em todos os casos. Fato em comum nesses testes é a pouca transmissão de dados entre o *host* e a GPU.

Os testes BFS, Reduction, Scan, Sort e Spmv, por outro lado, necessitam de mais dados sendo transferidos entre o *host* e a GPU, apresentando então uma eficiência em torno de

90%.

Tabela 7 – Resultado do teste SHOC com GPU local e remota

Teste	Unidade	1 GPU Local		1 GPU Remota		Eficiência
		Média	Conf.	Média	Conf.	
BusSpeedDownload	GB/s	5,993	1,44E-03	4,047	6,58E-02	67,5%
BusSpeedReadback	GB/s	6,482	1,37E-05	4,119	3,07E-02	63,6%
MaxFlops	GFLOPS	996,180	0,104	997,043	0,142	100,1%
DeviceMemory	GB/s	520,551	4,705	518,448	5,631	99,6%
KernelCompile	ms	0,106	1,15E-03	3,075	4,21E-02	3,4%
QueueDelay	ms	4,33E-03	1,12E-06	5,07E-03	1,82E-05	85,5%
BFS	GB/s	9,53E-02	1,03E-03	8,99E-02	8,10E-04	94,4%
FFT	GFLOPS	54,985	0,198	54,222	0,186	98,6%
SGEMM	GFLOPS	333,643	1,022	324,910	3,463	97,4%
MD	GFLOPS	40,425	0,306	39,446	1,27E-02	97,6%
Reduction	GB/s	51,788	0,540	44,568	0,480	86,1%
Scan	GB/s	18,459	9,00E-02	12,510	0,132	67,8%
Sort	GB/s	0,246	1,66E-03	0,215	1,42E-03	87,3%
Spmv	GB/s	2,456	3,10E-03	2,232	7,30E-03	90,9%
Stencil2D	GFLOPS	66,345	0,377	6,626	4,77E-02	10,0%
Triad	GFLOP/s	0,979	7,94E-04	0,505	7,98E-03	51,6%
S3D	GFLOPS	65,394	0,211	64,273	2,556	98,3%

Os testes Triad e Stencil2D apresentam uma maior queda no desempenho, pois utilizam um grande conjunto de dados transmitidos, afetando assim o tempo total necessário para sua execução, tendo então menor eficiência.

Analisando todos os resultados com o SHOC, é possível verificar que na maioria dos casos houve perda de desempenho, com diferentes proporções. Isso era esperado devido à inclusão de uma transferência de rede na comunicação entre o *host* e a GPU. Também é possível verificar a existência de uma relação entre o desempenho e o volume de dados transmitidos via rede. Nos testes com baixa transferência de dados, como MaxFlops, DeviceMemory, BFS, FFT, SGEMM, MD, Reduction, Scan, Sort, Spmv e S3D, houve uma pequena, ou nenhuma, queda no desempenho do uso do protótipo em relação à GPU local. No entanto, nos testes com alta transferência de dados, como BusSpeedDownload, BusSpeedReadback, Triad e, principalmente, Stencil2D, houve uma grande redução do desempenho.

O SHOC é capaz de realizar seus testes em um conjunto de GPUs, porém o resultado é dado pela média do desempenho de cada GPU e não com o desempenho combinado de múltiplas GPUs. Assim não é possível para o SHOC analisar o ganho no uso de múltiplos dispositivos através do protótipo do *middleware* DistributedCL, apesar de mostrar a sua

transparência. Portanto é necessário encontrar um *benchmark* capaz de combinar o resultado de todos os dispositivos disponíveis para avaliar o desempenho.

5.3 LuxMark

O LuxMark (LUXRENDER PROJECT, 2012) é uma ferramenta de software livre para teste de desempenho de dispositivos OpenCL, parte integrante da biblioteca LuxRender (LUXRENDER PROJECT, 2013) para renderização de imagens. Ele é capaz de executar em qualquer dispositivo OpenCL, inclusive CPU, e distribui o trabalho em todos os dispositivos disponíveis.

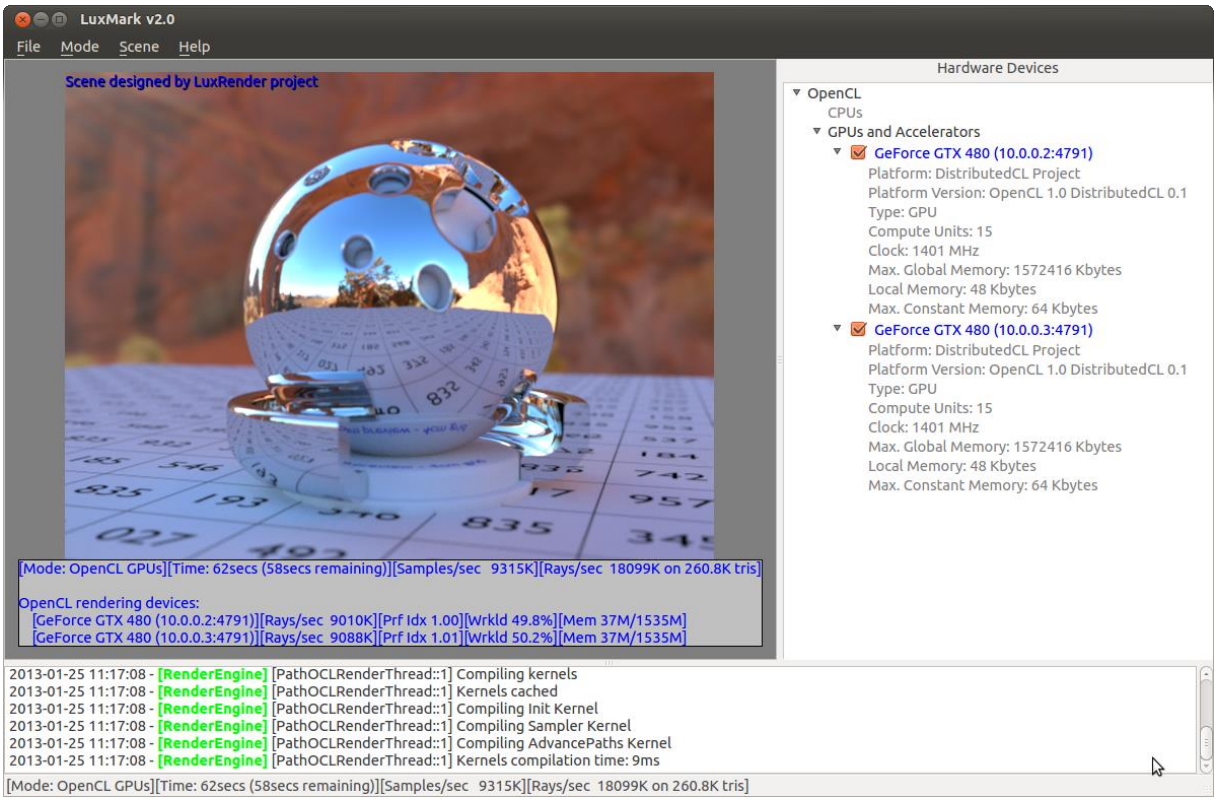

Figura 34 – Tela do LuxMark com duas GPUs remotas

O teste de desempenho do LuxMark consiste em renderizar, utilizando o algoritmo de traçado de raios (*Ray Tracing*), uma mesma cena em diversos dispositivos OpenCL por 2 minutos, medindo o número de Raios/segundo e *Samples*/segundo de cada dispositivo e geral do ambiente. Ele possui três imagens para renderização, com diferentes complexidades: “LuxBall HDR”, mais simples com 262K triângulos; “Sala”, de complexidade média com 488K triângulos; e “Room”, de maior complexidade com 2.016K triângulos.

Os testes com o protótipo do *middleware* DistributedCL utilizam o LuxMark v2.0 64 *bits* com a imagem “LuxBall HDR”, rodando de uma a sete GPUs remotas e também diretamente na GPU local, sem passar pelo protótipo, para comparação. Na Figura 34 está a

tela do LuxMark, executando com duas GPUs remotas.

5.3.1 Resultados

Na Figura 35 está o gráfico com o número de *Samples*/segundo com diferentes quantidades de GPU remotas configuradas no protótipo do *middleware* DistributedCL, o intervalo de confiança é de 95%.

É interessante notar que o gráfico apresenta comportamentos diferentes entre uma e duas GPUs remotas em relação às demais. O desempenho de duas GPUs remotas é próximo do dobro de uma GPU remota, indicando uma boa evolução, porém este desempenho diminui fortemente entre três e quatro GPUs remotas e se mantém em um patamar abaixo da GPU local.

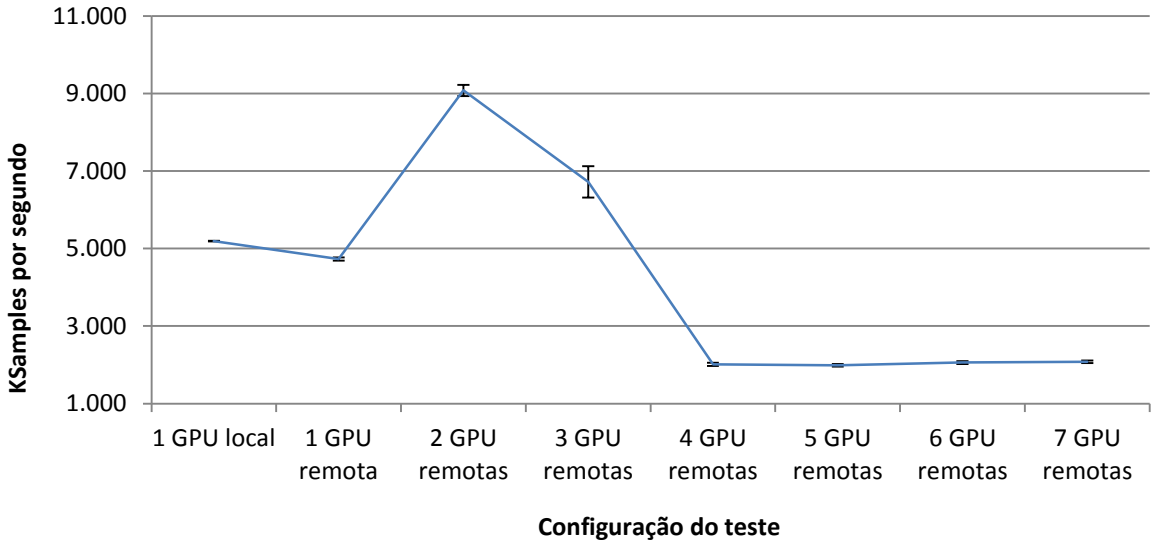

Figura 35 – LuxMark: resultado da avaliação de desempenho

Para explicar esta queda repentina no desempenho é interessante analisar o tráfego de rede, pois, como visto no teste da Seção 5.2, ela é um importante ponto de gargalo para as GPUs remotas. Para relacionar a carga que o LuxMark impõe à rede com o desempenho encontrado, na Tabela 8 está a comparação entre o número de *Samples*/segundo, o *Speedup* e o uso da rede medido no computador cliente, onde rodou o LuxMark.

A capacidade nominal de uma *Ethernet gigabit* é cerca de 120 MiB/s, logo é possível observar que com duas GPU remotas o uso da rede está perto da metade de sua capacidade. Com três GPUs remotas o uso da rede está praticamente no limite da capacidade, gerando então um gargalo, limitando o desempenho.

No entanto, somente o gargalo na rede não explica o motivo que o desempenho diminui na configuração a partir de três GPUs remotas. Acredita-se que a resposta esteja no

escalonador de tarefas do LuxMark, que deve levar em consideração o tempo de resposta de cada GPU, diminuindo a carga do mesmo. O aprofundamento deste motivo foge ao escopo do trabalho atual e deve ser tema de estudos futuros.

Tabela 8 – Resultados dos testes de desempenho utilizando o LuxMark

Configuração	*Samples* / segundo	*Speedup*	Uso da rede no cliente	
			Entrada	Saída
1 GPU local	5.195,23	1,00	–	–
1 GPU remota	4.729,03	0,91	24 MiB/s	180 KiB/s
2 GPUs remotas	9.078,14	1,75	51 MiB/s	480 KiB/s
3 GPUs remotas	6.719,96	1,29	115 MiB/s	820 KiB/s
4 GPUs remotas	2.015,26	0,39	117,7 MiB/s	1,1 MiB/s
5 GPUs remotas	1.991,76	0,38	117,7 MiB/s	1,3 MiB/s
6 GPUs remotas	2.060,43	0,40	117,7 MiB/s	1,6 MiB/s
7 GPUs remotas	2.080,95	0,40	117,7 MiB/s	1,8 MiB/s

Devido ao gargalo encontrado nos testes com o LuxMark, é necessário utilizar outra ferramenta para testes de desempenho que tenha um conjunto de dados pequeno para a avaliação de como o protótipo do *middleware* DistributedCL escala horizontalmente.

5.4 BFGMiner

O BFGMiner (LUKE-JR, 2012) é uma ferramenta para mineração de Bitcoins (THE BITCOIN FOUNDATION, 2012). Bitcoin um sistema de transação financeira online, onde as transações precisam ser validadas, ou "mineradas" no jargão do Bitcoin, a partir de cálculos baseados em algoritmos de resumo criptográficos (*hash*), para que possam ser confirmadas através de um protocolo ponto a ponto (*peer-to-peer*).

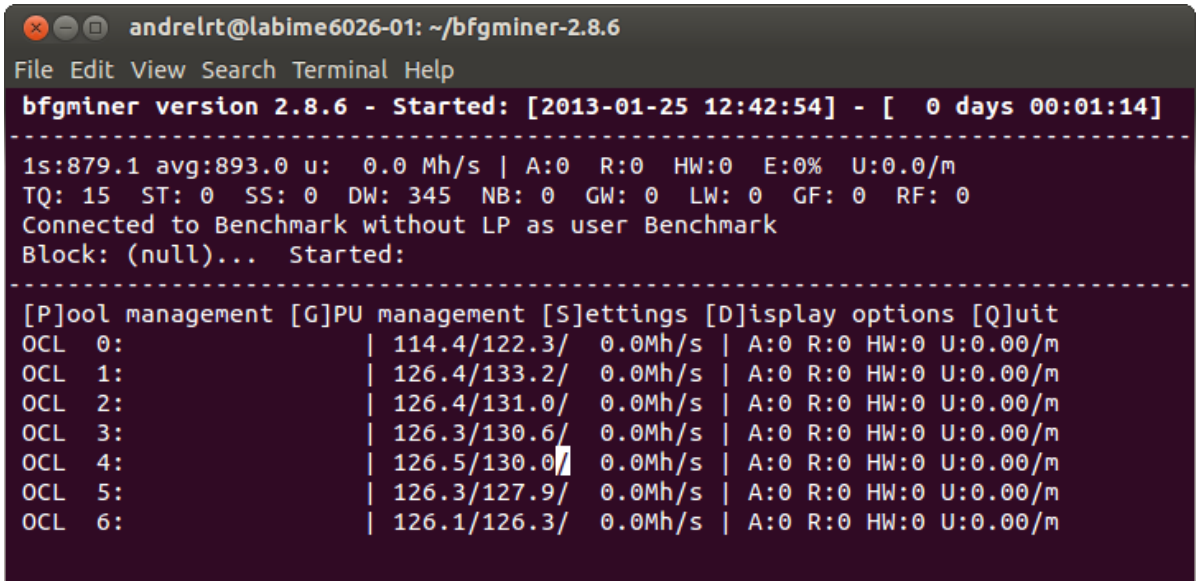

Figura 36 – Tela do BFGMiner com sete GPUs remotas

O BFGMiner é uma ferramenta que utiliza CPUs, GPUs e FPGAs para a mineração de Bitcoins e utiliza OpenCL para acessar todos os dispositivos disponíveis. Além disso, necessita pouca transferência de dados, pois os resumos criptográficos normalmente possuem tamanho pequeno. Apesar de não ter, a princípio, finalidade acadêmica, é possível utilizar a ferramenta para rodar somente um teste de desempenho, sem efetuar a mineração de Bitcoins.

Os testes com o protótipo do *middleware* DistributedCL utilizam o BFGMiner v2.8.6, na configuração de teste de desempenho, rodando de uma a sete GPUs remotas e também diretamente na GPU local, sem passar pelo protótipo, para comparação. Na Figura 36 está a tela do BFGMiner, executando com sete GPUs remotas.

5.4.1 Resultados

O gráfico com número de resumos verificados por segundo em cada conjunto de GPUs remotas configuradas no protótipo do *middleware* DistributedCL está na Figura 37.

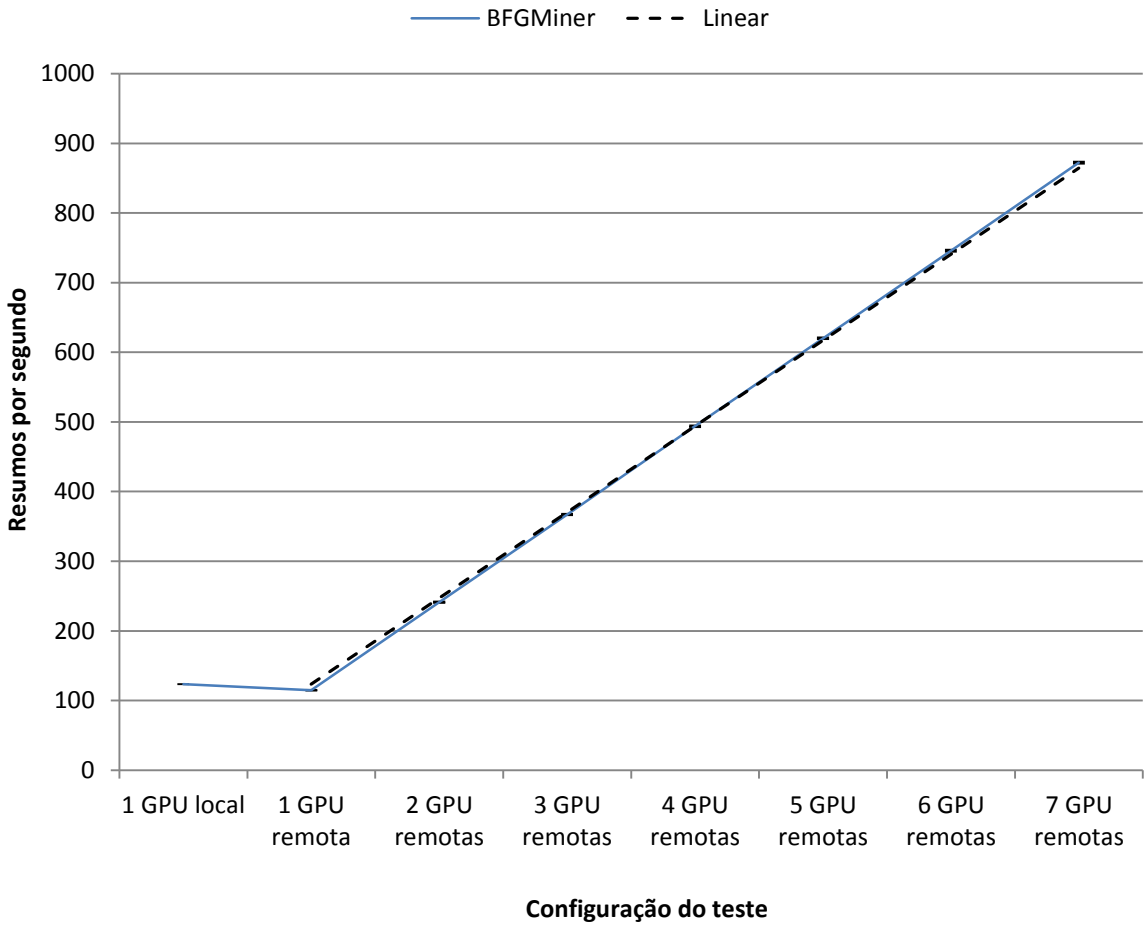

Figura 37 – BFGMiner: resultado da avaliação de desempenho

Diferente dos resultados teste da Seção 5.3, o gráfico mostra um crescimento praticamente linear no desempenho da ferramenta BFGMiner com o uso do protótipo. O

desempenho chega a ser ligeiramente superior ao crescimento linear do número de GPUs remotas. Esta linha está representada pontilhada no gráfico com a legenda Linear.

Outro detalhe importante é que, apesar ter mais de 300 dados coletados por configuração, o intervalo de confiança de 99% se apresenta muito pequeno. Isto mostra a consistência dos dados e do experimento, além da estabilidade do protótipo do *middleware* DistributedCL.

Porém, um crescimento linear, ou um pouco acima como foi o caso, é um fato incomum em sistemas distribuídos. De forma similar aos testes da Seção 5.2 e 5.3, a rede é um fator importante para entender como o desempenho se comportou desta forma.

Uma analise os pacotes transmitidos pelo BFGMiner através do protótipo, mostra que os dados úteis transmitidos da GPU para o programa BFGMiner possuem apenas 64 bytes. Assim, o uso da rede é bastante diminuído em relação ao teste com o LuxMark, garantindo assim um bom desempenho, mesmo em GPUs remotas. Porém somente o pouco uso da rede não é suficiente para explicar o ganho de desempenho maior.

Este ganho pode ser explicado pelo tratamento assíncrono realizado pelo protótipo. Como os dados são transmitidos através da rede em paralelo com a execução no servidor, o tempo perdido na latência de rede é compensado pelo tempo de execução da mineração de dados feita na GPU, escondendo esta latência. Pela transmissão de dados em lote e de forma assíncrona, o controle é retornado à aplicação antes do efetivo envio dos dados ao servidor, permitindo que o BFGMiner possa enviar mais comandos, aumentando assim o uso concorrente da CPU local, da rede, da GPU remota. Este processo é mais bem detalhado na descrição de sua implementação no protótipo (Seção 4.3).

Como na Seção 5.3, este caso indica que o uso da rede é um ponto importante na análise de desempenho do protótipo, pois com um volume reduzido de dados transmitidos via rede o desempenho se manteve ideal.

Para avaliar melhor a relação do uso da rede com o desempenho do protótipo do *middleware* DistributedCL, é necessário então realizar uma avaliação com uma ferramenta que distribua o trabalho nas múltiplas GPUs e que utilize diferentes cargas na GPU e no transporte de dados.

5.5 CLBench

O CLBench é uma ferramenta especialmente construída, no escopo deste trabalho, para realizar a avaliação de desempenho do protótipo do *middleware* DistributedCL com duas características principais: utiliza todas as GPUs disponíveis e testa cada GPU com diferente

volume de dados. Assim é capaz de endereçar as questões de uso da rede, desde uma pequena carga até onde ela se torna um gargalo para a aplicação e mostra a escalabilidade do protótipo de acordo com a carga colocada.

O CLBench utiliza um cálculo de multiplicação de vetores para testar o processamento em GPU. Dois vetores são transmitidos para a GPU somente uma vez, e a execução da multiplicação e a posterior leitura dos dados é executada em loop para avaliar o tempo de execução associado ao tempo de transmissão dos dados, sendo que a execução da multiplicação é realizada várias vezes em loop para gerar mais carga à GPU em cada iteração.

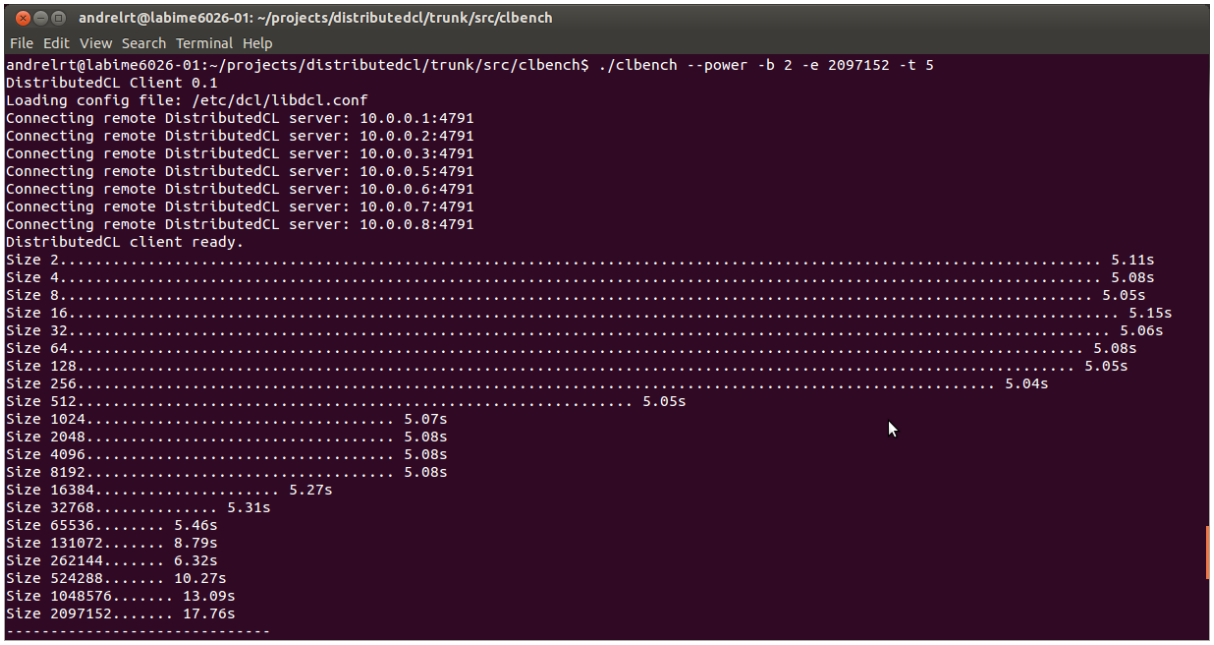

Figura 38 – Tela do CLBench com sete GPUs remotas

O teste possui dois modos de execução: por tempo definido e por carga definida. A execução por tempo definido a mesma carga de dados é enviada para todas as GPUs por um tempo, passado como parâmetro pela linha de comando. A execução por carga definida o volume de dados é dividido entre todas as GPUs disponíveis executando em loop por 1.000 vezes. Além disso, a carga a ser aplicada é definida em uma faixa de tamanhos, passados por parâmetros na linha de comando, e evoluir em potências de dois ou de forma linear com saltos definidos.

Os testes com o protótipo do *middleware* DistributedCL utilizam o CLBench nos dois modos de execução, por tempo definido e por carga definida, com o vetor de números em ponto flutuante de precisão simples de tamanho 2 até 2.097.152, evoluindo em potência de dois. Os testes utilizaram configurações de uma a sete GPUs remotas e também diretamente na GPU local, sem passar pelo protótipo, para comparação. Na Figura 38 está a tela do CLBench, executando com sete GPUs remotas.

5.5.1 Resultados do modo por tempo definido

O gráfico com o resultado dos testes com o CLBench (Figura 39) tem no eixo X o tamanho vetor e no eixo Y o número de operações por segundo do conjunto de GPUs testadas. Apesar do intervalo de confiança ser de 99%, ele se apresenta muito pequeno, menor que os marcadores que identificam cada curva. É importante notar que o CLBench entende como uma operação a multiplicação de todo o vetor e o transporte dos dados da GPU até o programa, o que nas GPUs remotas inclui o tempo de transmissão pela rede.

Neste gráfico temos quatro regiões bem definidas: a primeira dos vetores de tamanho 2 a 128, a segunda de 128 a 1.024, a terceira de 1.024 a 8.192 e a quarta de 8.192 em diante. Mesmo executando localmente, este mesmo padrão se repete, sendo então uma característica do teste em relação à GPU.

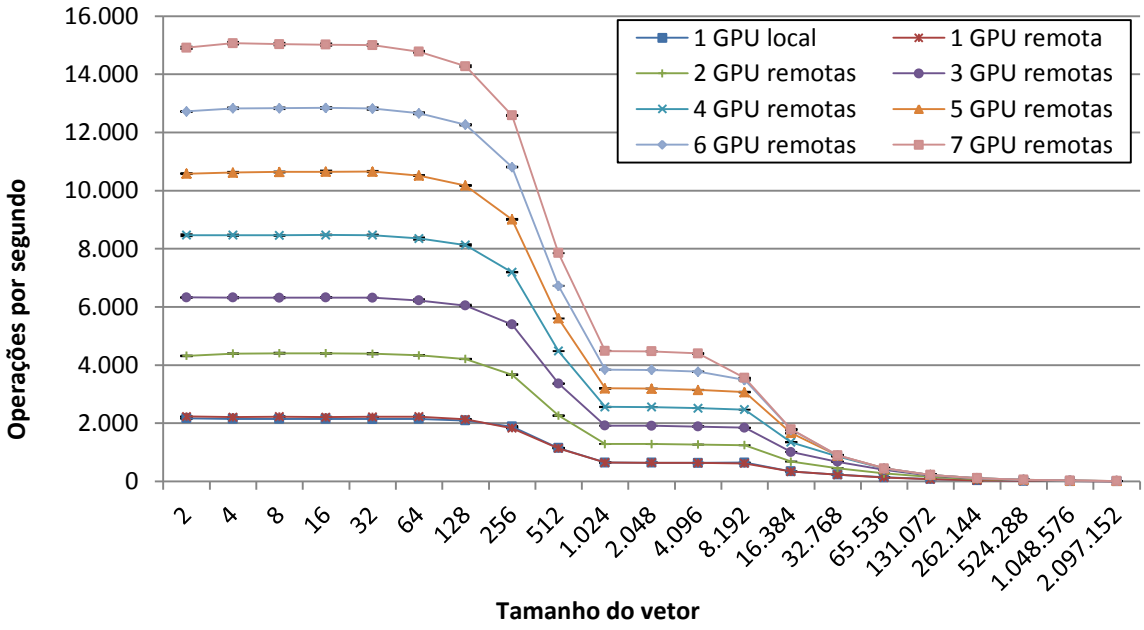

Figura 39 – CLBench: operações por segundo

Com os vetores de tamanho 2 a 128, o gráfico se mantém estável em cada uma das configurações. Isso mostra que para um conjunto pequeno de dados a capacidade máxima de processamento da GPU não é alcançada, tendo outros pontos, como a transmissão de dados entre o *host* e a GPU, como gargalo.

Com os vetores de tamanho 128 a 1.024 ocorre uma queda no desempenho, mesmo na GPU local, em relação à região anterior. Como a GPU local tem o mesmo comportamento, a rede não é o fator preponderante, como ocorrido nas seções anteriores, tendo que existir então outra explicação.

Para investigar a razão do comportamento do CLBench, o mesmo foi reexecutado em

outros modelos de GPU. No gráfico da Figura 40 está a comparação da execução local do CLBench da GTX480 com outras três GPUs: NVIDIA 8800GT, NVIDIA GT240 e NVIDIA GT630. Este teste foi realizado em um computador com outra configuração, mas o intuito é analisar a curva de desempenho gerada por cada GPU no teste CLBench e não a comparação do desempenho entre as GPUs.

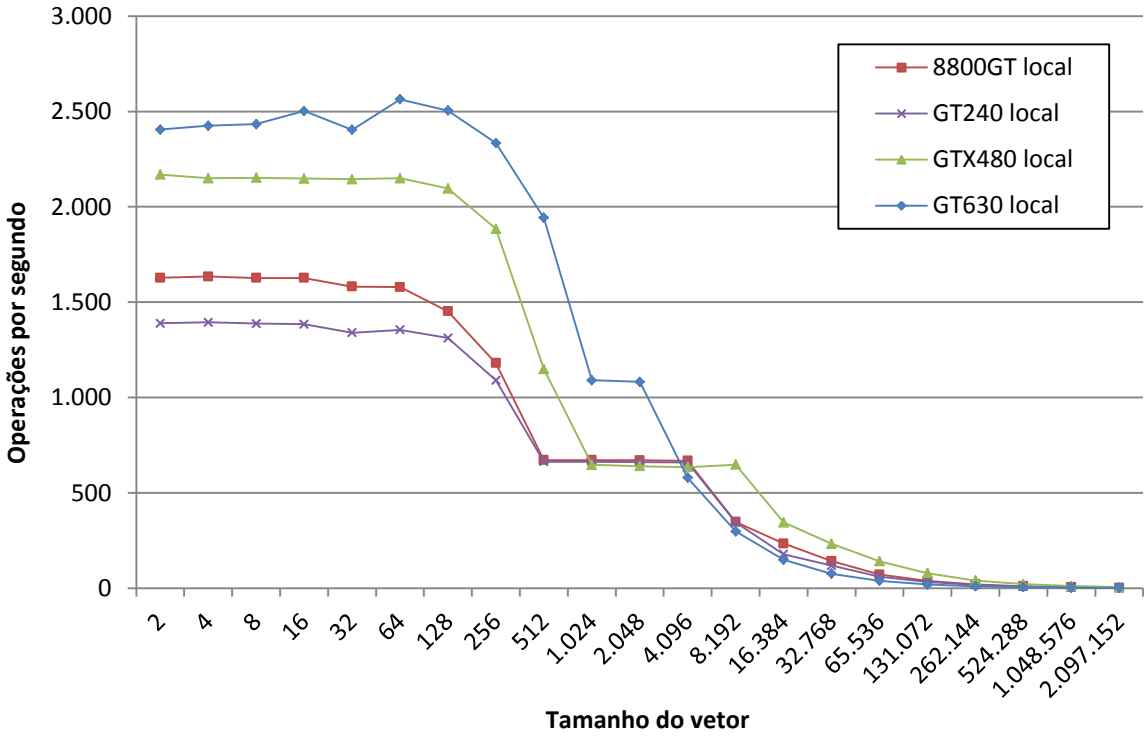

Figura 40 – CLBench: comparação entre diferentes GPUs no CLBench

Nota-se que os gráficos de todas as GPUs possuem um comportamento similar: iniciam de forma estável, têm uma queda, voltam a ficar estáveis para depois diminuir. Porém nas GPUs 8800GT e GT240 uma queda grande ocorre após tamanho 128 e a estabilização se dá no tamanho 512, enquanto que a GTX480 e a GT630, apesar de uma pequena queda entre 128 e 256, a queda maior no ocorre após tamanho 256 e a estabilização no tamanho 1.024.

Analisando as características das GPUs, verifica-se que as GPUs 8800GT e GT240 possuem o tamanho máximo de *work-items* em um *work-group* de 512, enquanto a GTX480 e GT630 possuem o tamanho igual a 1.024. Como o tamanho máximo do *work-group* coincide com o tamanho do vetor onde há a estabilização do desempenho na GPU, é possível que esta característica interna seja a responsável por este padrão no resultados do CLBench. No entanto, o aprofundamento desta característica foge ao escopo do trabalho atual e deve ser tema de trabalhos futuros.

Voltando aos resultados do primeiro teste, com os vetores de tamanho 1.024 a 8.192 existe outro período de estabilidade do desempenho. Mas vale notar que no teste com 6 e 7

GPUs remotas o desempenho começa a diminuir a partir do vetor com tamanho 8.192. Analisando o comportamento da rede nestes testes vemos que a carga no computador rodando o CLBench é em torno de 110 MiB/s, ou seja, praticamente toda a banda da *Ethernet gigabit*.

Como nos vetores de tamanho maiores que 8.192 os valores de operações por segundo diminuem com um comportamento logarítmico, é melhor analisar estes resultados através de um gráfico com escala log2 (Figura 41).

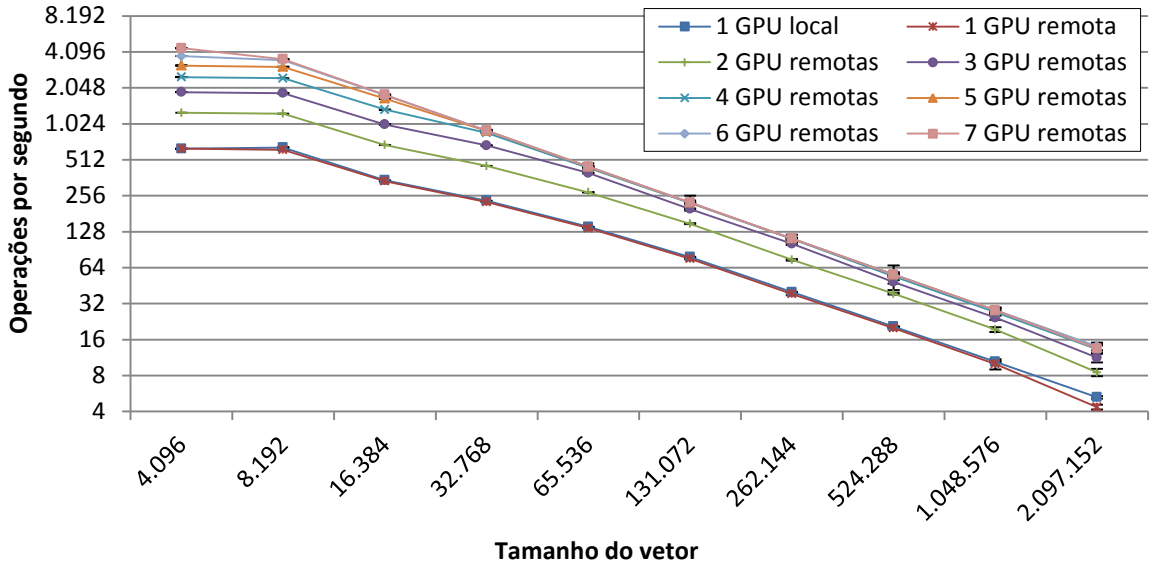

Figura 41 – CLBench: operações por segundo em escala log2

É possível observar que para os vetores com maiores tamanhos, um pequeno número de GPUs remotas ainda gera ganho até o uso de 3 GPUs. A partir de 4 GPUs em vetores maiores ou iguais a 65.536 as linhas se tornam muito próximas e os intervalos de confiança se sobrepõem. Assim não é possível afirmar que existe uma melhora no desempenho nesses casos.

Este mesmo experimento pode ser também observado através dos gráficos da Figura 42 e da Figura 43 que apresentam o *Speedup* de cada conjunto de GPUs remotas sobre o uso da GPU local em cada tamanho de vetor. Na linha pontilhada está a evolução ideal do desempenho da GPU local vezes o número de GPUs utilizadas.

É interessante notar no primeiro gráfico (Figura 42) que para tamanhos do vetor até 4.096 o desempenho é muito próximo do linear, mostrando que o protótipo do *middleware* DistributedCL escala de forma consistente até sete GPUs remotas. No entanto, o segundo gráfico (Figura 43) mostra que para tamanhos maiores ou iguais a 8.192 o desempenho não melhora com na mesma proporção com o número de GPUs remotas, chegando até, no pior caso, a não melhorar praticamente, como no caso de vetores maiores e iguais a 32.768 com 4

a 7 GPUs remotas.

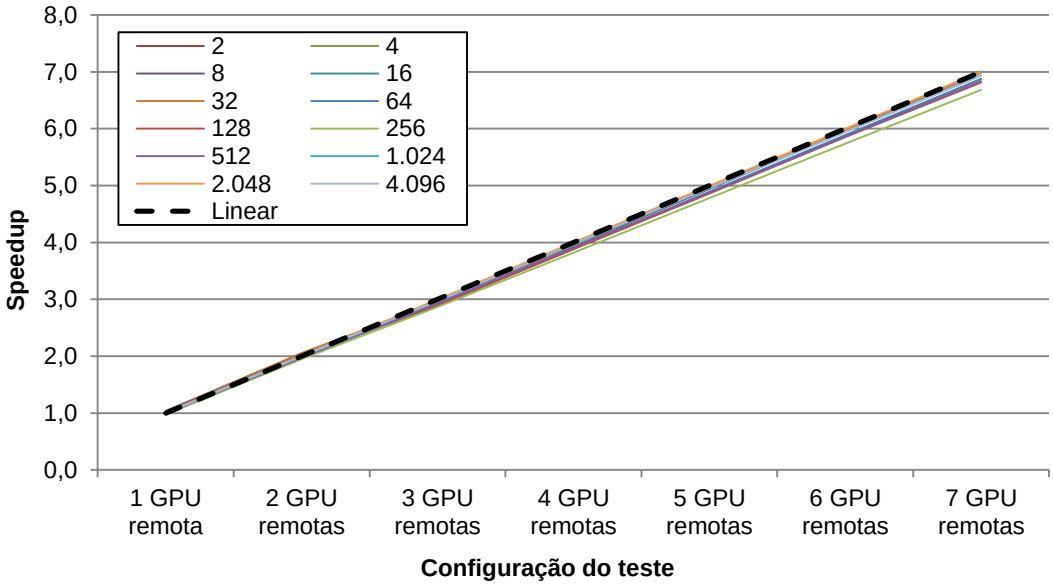

Figura 42 – CLBench: *Speedup* com vetores de tamanho de 2 a 4.096

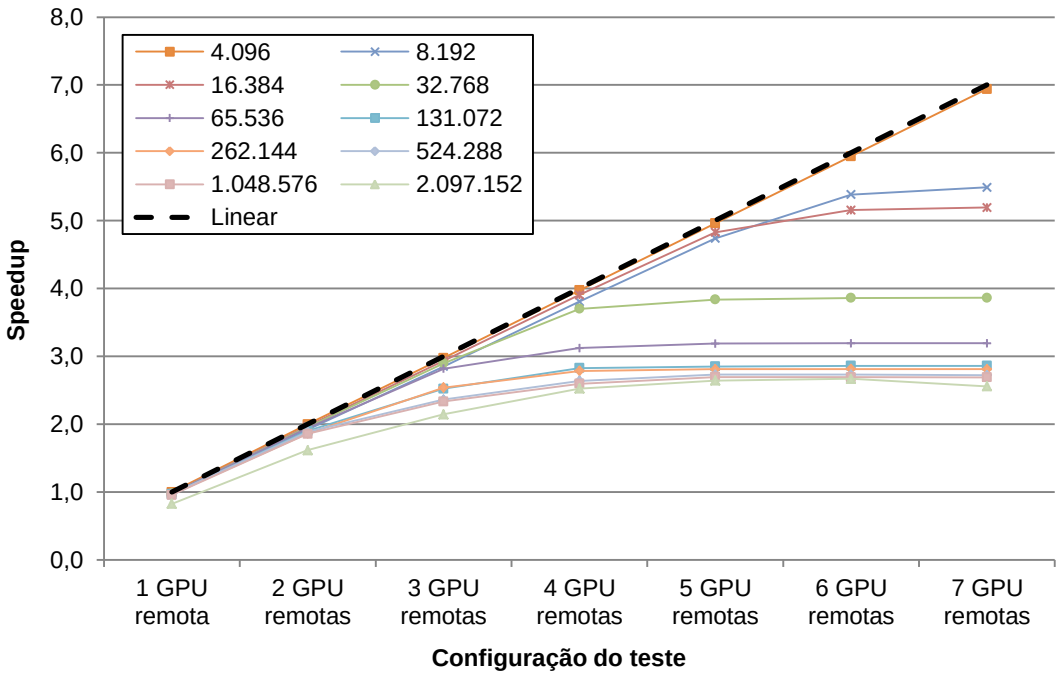

Figura 43 – CLBench: *Speedup* com vetores de tamanho de 4.096 a 2.097.152

5.5.2 Resultado do modo por carga definida

O gráfico na Figura 44 apresenta o *Speedup* do CLBench no modo por carga definida, nas configurações de uma a sete GPUs remotas e a GPU local.

O gráfico se apresenta diferente entre a configuração com uma GPU remota e as demais configurações. A configuração com uma GPU remota se mantém com o resultado inferior a 1, ou seja, com desempenho inferior ao da GPU local, em todo o gráfico, como

esperado, e com uma pequena redução de acordo com o tamanho do vetor.

As demais configurações possuem diferentes níveis de melhora do resultado de acordo com o tamanho do vetor. Para melhor análise é possível dividir o gráfico em quatro áreas, uma com vetores de tamanho 2 até 256, a segunda do tamanho 256 ao 1.024, a terceira do tamanho 1.024 ao 8.192 e a quarta do tamanho 8.192 em diante.

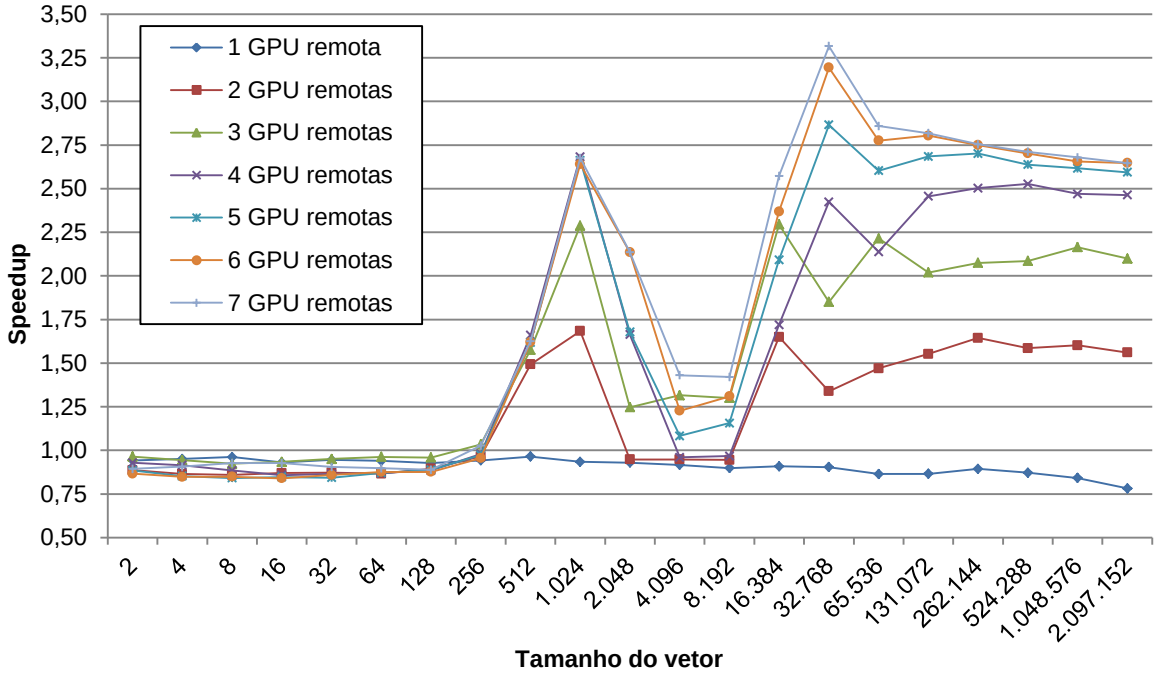

Figura 44 – CLBench: *Speedup* de uma a sete GPUs remotas

Nos vetores de tamanho 2 a 256 o gráfico se mostra estável e com resultado abaixo de 1, inferior ao desempenho da GPU local. Outra característica é que, apesar o aumento do número de GPUs remotas, não existe melhora no resultado do teste. Isto indica que para vetores pequenos não há vantagem no uso de múltiplas GPUs remotas no CLBench, sendo a melhor opção o uso da GPU local ou, em casos que o computador não possua GPU, o uso de somente uma GPU remota.

Nos vetores de tamanho 256 a 1.024 o gráfico mostra um aumento no desempenho. É interessante notar que nesta região do gráfico, todas as configurações tiveram *Speedup* em torno de 1,5 com o tamanho 512 e as configurações iguais ou maiores de 3 GPUs tiveram *Speedup* entre 2,25 e 2,75 no tamanho 1.024.

Com a observação do número de operações por segundo na faixa de 256 a 1.024 no gráfico da Figura 39 é possível notar que existe uma forte queda no desempenho para a GPU local. Esta queda do desempenho da GPU local é a principal causa da melhora no tempo de execução de múltiplas GPUs remotas neste caso. Como neste teste a carga é dividida entre as GPUs, cada GPU recebe um vetor menor, ainda na faixa de máximo desempenho entre 2 e

256. Por exemplo, no teste com 4 GPUs remotas e vetor de tamanho 1.024, cada GPU recebe um vetor de tamanho 256.

Nos vetores de tamanho 1.024 a 8.192 o gráfico apresenta uma redução do desempenho em todas as configurações, chegando a ter um valor abaixo de 1 no caso de 2 e 3 GPUs remotas e o melhor tempo na configuração com 7 GPUs, próximo de 1,5 de *Speedup*. Como neste teste o vetor é divido entre as GPUs, cada GPU pode receber um vetor na faixa entre 256 e 1.024. Verificando novamente a Figura 39, ela mostra que na faixa de 256 a 1.024 há a queda no desempenho em todas as configurações, portanto o aumento no tempo de execução, que é diferente em cada caso.

Nos vetores de tamanho 8.192 em diante o gráfico apresenta novamente uma ganho de desempenho em relação à GPU local e se estabiliza. No caso de 2 GPUs remotas a melhora fica em torno de 1,5, no caso de 3 GPUs remotas em torno de 2 e nos demais casos em torno de 2,5, chegando ao máximo de 3,3 no caso de 7 GPUs remotas e o vetor de tamanho 32.768. É possível verificar também, na Figura 41, que na faixa de 8.192 em diante o número de operações por segundo decresce em uma escala logarítmica. Assim a divisão do vetor em tamanhos menores gera ganhos pelo melhor desempenho de cada GPU.

Tabela 9 – Uso da rede em MiB/s no CLBench com carga definida

Tamanho	3 GPU remotas		4 GPU remotas		5 GPU remotas		6 GPU remotas		7 GPU remotas	
	Entrada	Saída	Entrada	Saída	Entrada	Saída	Entrada	Saída	Entrada	Saída
8.192	60,5	0,8	80,9	1,1	101,0	1,5	115,5	2,0	116,7	2,3
16.384	66,6	0,6	88,5	0,8	109,0	1,2	116,7	1,8	117,5	2,1
32.768	88,0	0,6	112,0	1,1	117,0	1,4	117,6	1,7	117,7	2,0
65.536	105,0	0,6	115,0	1,0	117,5	1,3	117,7	1,7	117,7	1,9
131.072	104,0	0,6	116,5	1,0	117,7	1,3	117,7	1,6	117,7	1,8
262.144	106,0	0,6	116,8	0,9	117,7	1,3	117,7	1,6	117,7	1,8
524.288	100,0	0,6	117,7	0,9	117,7	1,3	117,7	1,6	117,7	1,7
1.048.576	102,0	0,5	114,0	0,9	117,7	1,2	117,7	1,5	117,7	1,7
2.097.152	95,0	0,5	111,5	0,7	115,0	1,0	117,7	1,3	117,7	1,6

No entanto, mesmo com o aumento do número de GPUs remotas, após 4 GPUs não houve grande melhora no desempenho. Como no caso da Seção 5.3, a rede pode ser um bom indicativo para o entendimento desta situação. É possível observar na Tabela 9 que a rede fica sobrecarregada após 4 GPUs, limitando a melhora do tempo com um número maior de GPUs.

Assim, apesar de um ganho no desempenho no uso de múltiplas GPUs, o uso da rede se torna o fator limitante para o ganho de desempenho geral do processamento.

5.6 rCUDA

O SHOC possui suporte para as plataformas CUDA e OpenCL, utilizadas pelo rCUDA e o *middleware* DistributedCL respectivamente. No entanto, não é possível comparar

diretamente o resultado dos testes do SHOC nessas duas tecnologias, pois os resultados encontrados, para o mesmo hardware, são diferentes. Esta diferença pode ser explicada, pois o código dos testes do SHOC é diferente em cada caso, com diferentes otimizações, além das próprias bibliotecas CUDA e OpenCL da NVIDIA, que também possuem diferentes implementações.

Assim, a avaliação comparativa de desempenho entre o rCUDA e o protótipo do *middleware* DistributedCL é feita sobre o *overhead* gerado pelas duas propostas utilizando uma GPU remota em comparação com o teste SHOC executado na GPU local.

Os testes com o rCUDA utilizam as mesmas configurações utilizadas nos testes anteriores, sendo que, além da configuração descrita na Seção 5.1, utiliza também o CUDA Toolkit versão 5.0.35. O teste SHOC utiliza o teste com menor tamanho com uma GPU remota.

5.6.1 Resultados

O resultado dos testes da Tabela 10 apresenta o desempenho do rCUDA em todos os testes do SHOC. Os testes DeviceMemory e BFS não executaram corretamente com o rCUDA, portanto não são considerados nesta avaliação.

Tabela 10 – SHOC: *overhead* do rCUDA

Teste	Unid.	1 GPU Local		1 GPU Remota		Eficiência
		Média	Conf.	Média	Conf.	
BusSpeedDownload	GB/s	5,999	0,001	0,117	0,000	2,0%
BusSpeedReadback	GB/s	6,482	0,000	0,117	0,001	1,8%
MaxFlops	GFLOPS	986,584	0,105	983,393	0,335	99,7%
FFT	GFLOPS	105,873	1,120	13,702	0,514	12,9%
SGEMM	GFLOPS	310,527	0,174	1,699	0,028	0,5%
MD	GFLOPS	111,486	0,153	104,360	2,523	93,6%
Reduction	GB/s	95,105	0,330	45,085	1,740	47,4%
Scan	GB/s	27,307	0,051	17,302	0,550	63,4%
Sort	GB/s	1,796	0,002	1,111	0,058	61,9%
Spmv (Padded Vector)	GB/s	3,093	0,001	1,692	0,011	54,7%
Stencil2D	GFLOPS	79,051	0,727	14,421	0,236	18,2%
Triad	GFLOP/s	0,978	0,000	0,019	0,000	2,0%
S3D	GFLOPS	50,396	0,043	50,196	0,048	99,6%

Nos testes BusSpeedDownload e BusSpeedReadback, com o maior buffer de dados, o rCUDA apresenta uma baixa eficiência, o que é esperado devido à transferência de dados pela rede. No teste MaxFlops, por tratar de questões internas na GPU desconsiderando a transferência de dados, o rCUDA mantém um desempenho equivalente à GPU local. Nos testes FFT, SGEMM, Stencil2D e Triad houve uma grande perda de desempenho, mantendo uma eficiência de 12,9%, 0,5%, 18,2% e 2% respectivamente. Nos testes Reduction, Scan e Spmv a queda foi um pouco menor, tendo uma eficiência de 47,4%, 63,4% e 54,7%

respectivamente. Interessante notar que, nos testes MD e S3D, a queda no desempenho é relativamente mais baixa que nos outros testes, com eficiência de 93,6% e 99,6% respectivamente. Não foi encontrada uma relação direta na utilização da rede que pudesse explicar o porquê da queda no desempenho em cada um desses casos. Porém, esta comparação é feita somente no *overhead* das duas propostas, sem entrar nos detalhes dos motivos que levam os testes com rCUDA possuírem esta característica.

O gráfico na Figura 45 mostra a comparação da eficiência da GPU remota através do rCUDA e do protótipo do *middleware* DistributedCL em relação ao desempenho do medido com a GPU local.

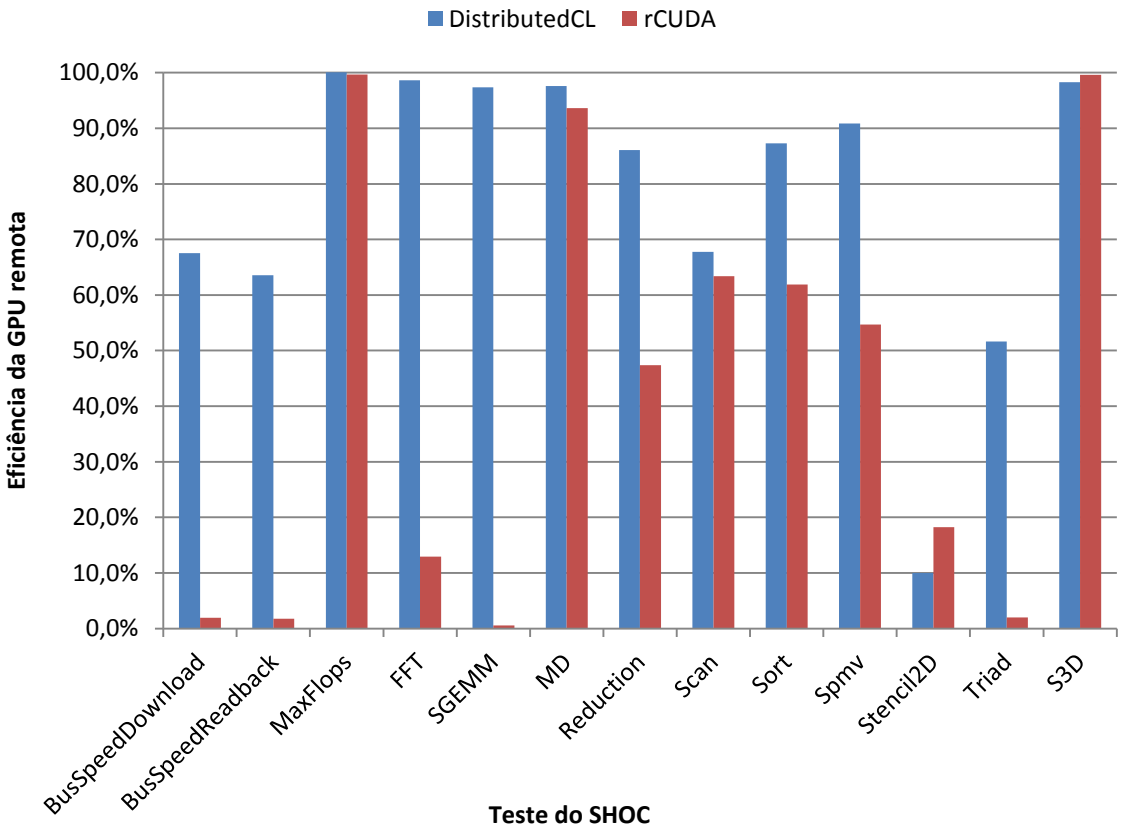

Figura 45 – SHOC: comparação dos *overheads* do protótipo e do rCUDA

No gráfico é possível perceber que nos testes MaxFlops, MD, Scan e S3D o rCUDA e o protótipo do middleware DistributedCL tiveram uma eficiência similar, sendo que no teste S3D o rCUDA foi ligeiramente superior. Nos testes Reduction, Sort e Spmv o protótipo teve uma eficiência um pouco mais alta e com resultados com diferença entre 25% e 40%. O resultado dos testes BusSpeedDownload, BusSpeedReadback, FFT, SGEMM e Triad mostra uma eficiência muito superior do protótipo, principalmente no teste SGEMM que o protótipo teve um resultado próximo da GPU local, enquanto o rCUDA teve uma eficiência de 0,5% do

desempenho da GPU local. Somente no teste Stencil2D o rCUDA conseguiu uma eficiência superior, conseguindo manter uma eficiência 18,2% da GPU local, enquanto o protótipo teve próximo de 10%.

Esta comparação mostra que no *benchmark* SHOC, de uma forma geral, o protótipo do *middleware* DistributedCL possui um *overhead* menor que o rCUDA.

5.7 Mosix VirtualCL

Os testes comparativos do Mosix VirtualCL com o protótipo do *middleware* DistributedCL utilizam os testes SHOC, LuxMark e CLBench. Foram realizados mais testes comparativos neste caso, devido à aproximação entre as propostas. O teste BFGMiner não executou corretamente com o Mosix VirtualCL, portanto não foi considerado na comparação.

Os testes com o Mosix VirtualCL utilizam as mesmas configurações utilizadas nos testes anteriores. O teste com SHOC utiliza o teste com menor tamanho com uma GPU remota. O teste com LuxMark utiliza a imagem "LuxBall HDR" com duas GPUs remotas. O teste com CLBench por tempo definido com sete GPUs remotas.

5.7.1 Resultados com SHOC

O resultado da execução do teste SHOC utilizando o Mosix VirtualCL está Tabela 11.

Tabela 11 – Resultado do teste SHOC com Mosix VirtualCL

Teste	Unidade	1 GPU Local		Mosix VirtualCL		Eficiência
		Média	Conf.	Média	Conf.	
BusSpeedDownload	GB/s	5,993	1,44E-03	6,022	2,56E-03	100,5%
BusSpeedReadback	GB/s	6,482	1,37E-05	6,196	2,01E-03	95,6%
MaxFlops	GFLOPS	996,180	0,104	763,794	11,639	76,7%
DeviceMemory	GB/s	520,551	4,705	356,749	7,391	68,5%
KernelCompile	ms	0,106	1,15E-03	51,954	0,157	0,2%
QueueDelay	ms	4,33E-03	1,12E-06	–	N/A	N/A
BFS	GB/s	9,53E-02	1,03E-03	8,35E-03	3,84E-04	8,8%
FFT	GFLOPS	54,985	0,198	0,669	0,308	1,2%
SGEMM	GFLOPS	333,643	1,022	2,773	1,498	0,8%
MD	GFLOPS	40,425	0,306	1,928	3,69E-02	4,8%
Reduction	GB/s	51,788	0,540	0,619	6,32E-02	1,2%
Scan	GB/s	18,459	9,00E-02	9,92E-02	3,89E-02	0,5%
Sort	GB/s	0,246	1,66E-03	2,62E-03	3,56E-05	1,1%
Spmv	GB/s	2,456	3,10E-03	0,127	2,31E-03	5,2%
Stencil2D	GFLOPS	66,345	0,377	0,396	0,180	0,6%
Triad	GFLOP/s	0,979	7,94E-04	1,67E-02	9,39E-05	1,7%
S3D	GFLOPS	65,394	0,211	0,389	2,66E-02	0,6%

Nos testes BusSpeedDownload e BusSpeedReadback, com o maior buffer de dados, o Mosix VirtualCL apresenta desempenho similar ao da GPU local, com uma eficiência próxima dos 100%. Nos testes MaxFlops e DeviceMemory, apesar de tratarem questões internas na GPU, o Mosix VirtualCL apresenta uma eficiência menor, 76,7% e 68,5% respectivamente. O teste KernelCompile há uma queda no desempenho, com eficiência muito baixa, mas, como discutido na Seção 5.2, um pior desempenho neste teste não implica um pior desempenho geral de uma aplicação. O teste QueueDelay não gerou resultado com o Mosix VirtualCL, retornando todos os dados com o valor zero, sendo então retirado da avaliação. Os testes BFS, FFT, SGEMM, MD, Reduction, Scan, Sort, Spmv, Stencil2D, Triad e S3D apresentaram uma grande queda no desempenho do Mosix VirtualCL, tendo uma eficiência abaixo de 10% em todos os casos, chegando a ser somente 0,5% no teste Scan.

Os testes BusSpeedDownload e BusSpeedReadback medem a capacidade de transmissão de dados entre o host e o dispositivo. Nesses testes o Mosix VirtualCL apresenta uma eficiência próxima a 100%, porém em outros testes, que também incluem a transferência de rede, os resultados são muito baixos, com eficiência abaixo de 10%. Isso levanta uma suspeita sobre o resultado do teste.

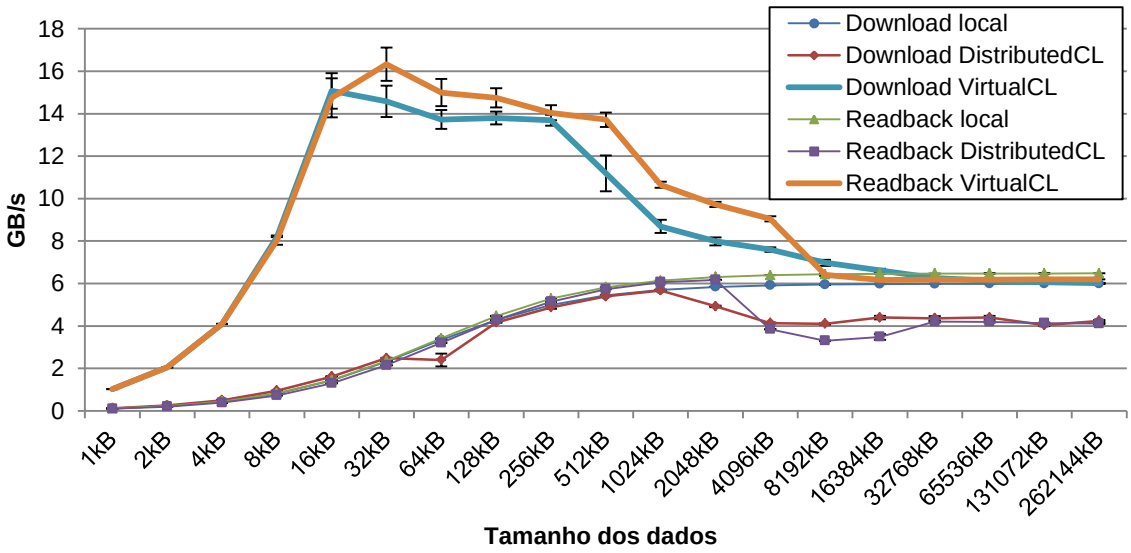

Figura 46 – Comparação do BusSpeedDownload e BusSpeedReadback com GPU local, *middleware* DistributedCL e Mosix VirtualCL

A avaliação de desempenho com os testes BusSpeedDownload e BusSpeedReadback é feita somente com o maior tamanho de dados transferidos. Porém, analisando o resultado do desses testes com todos os tamanhos é possível identificar, no gráfico da Figura 46, uma divergência nos resultados. O desempenho do teste entre tamanhos 1KiB e 8.192KiB se apresenta muito superior ao da GPU local, o que indica que o Mosix VirtualCL possui algum

problema ou otimização para este caso, impossibilitando que os testes façam a avaliação do desempenho. É possível notar que, na mesma faixa, os testes através do protótipo apresentam um desempenho com mesmo comportamento da GPU local.

Desta forma, dada à divergência de resultados, os testes BusSpeedDownload e BusSpeedReadback não são considerados na avaliação de desempenho do Mosix VirtualCL.

Comparando graficamente a eficiência do protótipo do *middleware* DistributedCL e o Mosix VirtualCL (Figura 47) é possível verificar que o protótipo possui eficiência superior em todos os casos. Na maioria dos casos o protótipo possui eficiência superior a 90%, enquanto o Mosix VirtualCL possui a maioria abaixo de 10%.

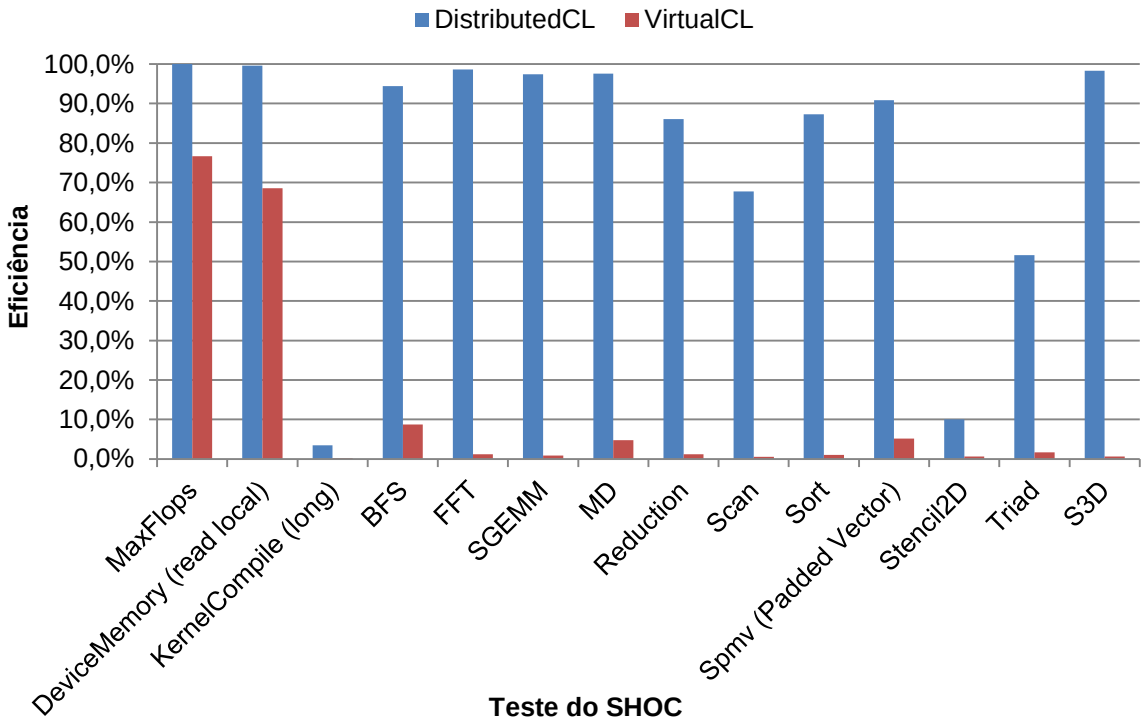

Figura 47 – SHOC: comparação da eficiência entre DistributedCL e VirtualCL

Quando comparado os valores, verifica-se que os testes através do protótipo mantém desempenho superior ao Mosix VirtualCL, como, por exemplo, no teste S3D, que executa a 0,389 GFLOPS neste, enquanto o mesmo teste através do protótipo executa a 64,273 GFLOPS, 165 vezes mais rápido.

Deste modo, é possível constatar que o protótipo do *middleware* DistributedCL provê um desempenho expressivamente superior ao Mosix VirtualCL no teste SHOC.

5.7.2 Resultados com LuxMark

O resultado do teste com o LuxMark está no gráfico da Figura 48 com o intervalo de

confiança de 95%.

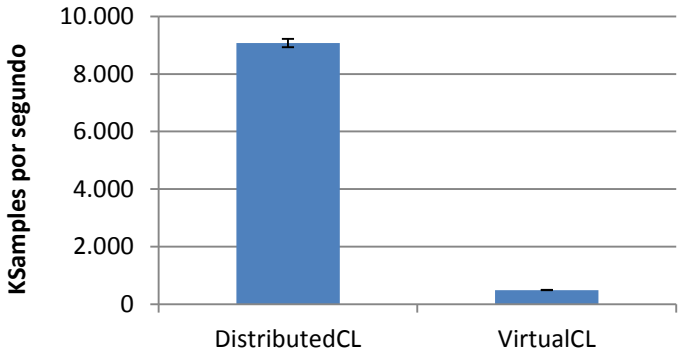

Figura 48 – LuxMark: comparação de resultados entre DistributedCL e VirtualCL

O LuxMark manteve uma velocidade de desenho de 486 KSamples/s através do Mosix VirtualCL, mostrando assim um desempenho inferior ao teste com o *middleware* DistributedCL, que manteve a velocidade de 9.078 KSamples/s, quase 19 vezes mais rápido.

5.7.3 Resultados com CLBench

A comparação do teste CLBench através do protótipo do *middleware* DistributedCL e o Mosix VirtualCL utiliza a média da eficiência das GPUs em cada configuração. Esta média é calculada considerando a eficiência das GPUs em cada configuração com todos os tamanhos de vetores do teste CLBench.

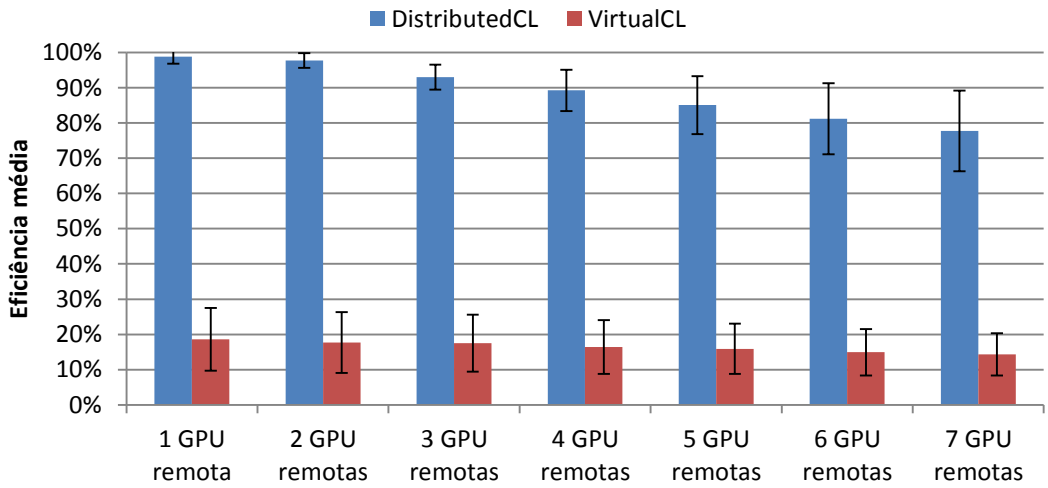

Figura 49 – CLBench: comparação da eficiência entre DistributedCL e VirtualCL

O gráfico comparativo da eficiência média da GPU entre protótipo do *middleware* DistributedCL e o Mosix VirtualCL está na Figura 49. Ele está com o intervalo de confiança é de 95% e mostra que, em todas as configurações, o protótipo do *middleware* DistributedCL apresentou uma eficiência média superior no CLBench em relação ao Mosix VirtualCL.

Com a comparação dos resultados, através da Tabela 12, é possível verificar que a

eficiência média do protótipo é mais que cinco vezes maior, em todas as configurações.

Tabela 12 – CLBench: comparação entre DistributedCL e VirtualCL

Configuração	DistributedCL		VirtualCL		Var.
	Média	Conf.	Média	Conf.	
1 GPU remota	98,85%	1,97%	18,58%	8,91%	5,32
2 GPU remotas	97,72%	2,09%	17,66%	8,64%	5,53
3 GPU remotas	93,01%	3,55%	17,48%	8,10%	5,32
4 GPU remotas	89,27%	5,85%	16,40%	7,62%	5,44
5 GPU remotas	85,08%	8,25%	15,91%	7,17%	5,35
6 GPU remotas	81,22%	10,05%	14,94%	6,59%	5,44
7 GPU remotas	77,77%	11,47%	14,31%	6,00%	5,43

O protótipo do *middleware* DistributedCL se mostrou com o desempenho superior ao Mosix VirtualCL em todos os testes executados. No SHOC o teste S3D executou 165 vezes mais rápido através do protótipo; com o LuxMark, ele foi quase 19 vezes mais rápido; e com o CLBench um pouco mais de 5 vezes mais rápido. Isso mostra que a proposta do *middleware* DistributedCL possui um desempenho consistente e que sua arquitetura e tratamentos assíncronos são capazes de ter um bom desempenho em comparação a propostas similares.

6 CONCLUSÃO

O *middleware* DistributedCL possibilita a execução de aplicações OpenCL acessando diversas GPUs em computadores remotos, de forma transparente, sem necessidade de modificação ou nova compilação do código. Com o protótipo desta arquitetura, aplicações como o SHOC, LuxMark e BFGMiner podem utilizar o processamento de GPUs localizadas em outros computadores sem nenhum mecanismo de processamento distribuído, como o RPC ou MPI, somente utilizando a biblioteca OpenCL fornecida pelo protótipo.

A arquitetura modular do *middleware* DistributedCL permite uma série de otimizações no protótipo, principalmente para a execução assíncrona de chamadas da API OpenCL, que resultam em um bom desempenho nas avaliações com as ferramentas de *benchmark* SHOC, LuxMark e BFGMiner. Além dessas, com a ferramenta CLBench, construída especialmente para esta avaliação, é possível testar GPUs remotas em conjunto, com diferentes cargas de dados, avaliando não só a execução, mas também o transporte de dados através da rede.

O desempenho do protótipo do *middleware* DistributedCL é muito bom na maioria dos casos, tendo, em algumas avaliações, resultados próximos do ideal. No entanto, em alguns casos a eficiência das GPUs é prejudicada, sendo possível encontrar uma relação entre a queda do desempenho e o aumento do volume de dados utilizados. Esta relação entre o desempenho e o volume de dados mostra que é boa a estratégia empregada na arquitetura do *middleware* DistributedCL de reduzir as transferências de rede sempre que possível, pois esse transporte de dados é o maior fator limitante da solução.

Em comparação com propostas similares, rCUDA e Mosix VirtualCL, os resultados nos testes de desempenho apontam a superioridade do protótipo do *middleware* DistributedCL em relação a estas plataformas. A comparação entre o rCUDA e o protótipo mostra que este possui menor *overhead* em praticamente todos os testes, salvo o Stencil2D onde o rCUDA possui melhor desempenho. O protótipo tem um desempenho muito superior ao Mosix VirtualCL em todos os testes, sendo 165 vezes mais rápido no teste S3D do SHOC, 19 vezes mais rápido no LuxMark e 5 vezes mais rápido no teste com o CLBench.

6.1 Trabalhos futuros

Como atividade imediata, é importante realizar avaliações de desempenho em escala maior, com um conjunto maior de GPUs explorando a escalabilidade horizontal do protótipo do *middleware* DistributedCL em casos de dezenas ou centenas de dispositivos.

Também seria interessante aprofundar os estudos relacionados com os aspectos de

comunicação. Um estudo pode ser feito para aumentar a vazão da rede conservando-se a estrutura de rede atual, através do uso de múltiplas interfaces de rede com o *Linux Ethernet Bonding*, ou o uso de *Jumbo Frames*. Além disso, uma avaliação de desempenho pode ser feita com o protótipo do *middleware* DistributedCL com o suporte de redes de alto desempenho, como o *Infiniband* (INFINIBAND TRADE ASSOCIATION, 2013), com o código atual baseado em TCP/IP e também com a construção de uma nova camada de rede utilizando a API específica dessa tecnologia.

Como as avaliações identificaram a rede como principal fator limitante, outras funcionalidades também podem ser construídas no protótipo para reduzir o uso da rede, como, por exemplo, o uso de *multicast* para enviar as mensagens relativas à API de contexto, que necessitam ser transmitidas a todos os servidores conectados. Outra ideia para reduzir o uso da rede é alterar a arquitetura para permitir que o servidor do *middleware* DistributedCL seja capaz de ler seus dados através da rede via sistema de arquivos. Assim na transmissão de dados entre cliente e servidor estaria somente o caminho de rede onde estão os dados e não o conjunto total de dados. Logo a transmissão de um grande volume de dados não seria feita de um ponto central, que não se tornaria fator limitante neste caso.

Outra alteração no protótipo sobre redução do uso da rede pode ser feita com o uso de algoritmos de compactação de dados nos buffers de memória e imagens, que normalmente possuem o maior volume de dados para transmissão, em ambas as direções. Os algoritmos podem ser portados para OpenCL e executados diretamente na GPU.

Um trabalho pode ser feito para criar a gerência dos servidores do *middleware* DistributedCL, com a alocação de dispositivos a cada aplicação de acordo com a utilização de cada GPU. Uma alternativa deste trabalho poderia apresentar todos os dispositivos como um único dispositivo OpenCL, fazendo que a aplicação não necessite separar os seus dados entre as GPUs e comandar execuções de forma separada. Porém, neste caso deverá existir uma preocupação com a consistência da memória, que deve ser compartilhada entre os dispositivos, que pode limitar a solução a casos com alta independência entre o processamento de cada *thread* na GPU.

A arquitetura do Mosix VirtualCL possui o *Broker* para realizar as conexões entre o cliente e os servidores. Com este componente é possível uma melhor gerência de conexões no cliente, permitindo a conexão prévia com cada servidor, que é utilizada por um conjunto de aplicações no cliente. Inspirado nesta ideia, pode ser feita uma alteração na arquitetura do *middleware* DistributedCL para incluir um elemento como esse que permitiria uma melhor gerência das conexões, criando previamente as conexões com cada servidor.

O *middleware* DistributedCL não possui suporte a tolerância de falhas, podendo ser este também um tema de trabalhos futuros. Questões de segurança, como autenticação, autorização e controle de acesso aos dados entre clientes diferentes, também são fonte de trabalhos futuros.

Além dos pontos mencionados até aqui, durante este trabalho foram identificadas outras questões que poderiam ser aprofundadas em trabalhos futuros. Na Subseção 4.2.2, por exemplo, a descrição dos parâmetros mostra que é possível para um servidor acessar outros servidores do protótipo, porém esta possibilidade não é explorada neste estudo. Uma avaliação pode ser feita neste caso, identificando se esta configuração traria benefício.

A Subseção 5.3.1 mostra que o *benchmark* LuxMark reduz drasticamente o desempenho a partir de 4 GPUs remotas e com a rede sobrecarregada. Um estudo pode ser feito analisando o código do LuxMark para identificar o motivo desta queda no desempenho.

A Subseção 5.5.1 inicia uma discussão sobre o comportamento da GPU com o teste CLBench, mostrando uma possível relação entre o tamanho máximo do *work-group* de uma GPU com o tamanho do vetor de dados onde há uma estabilização do desempenho. Uma verificação mais detalhada pode ser feita para validar esta relação.

Aplicações podem utilizar o *middleware* DistributedCL para acessar múltiplas GPUs e utilizar seu processamento. As aplicações podem ter grande ganho nesta arquitetura, principalmente para aquelas onde existe um pequeno conjunto de dados a serem tratados e um grande processamento nas GPUs. Mesmo as aplicações com mais dados podem utilizar este *middleware*, mas para conseguir um bom ganho no desempenho é necessário alterar o código de forma que torne paralela a execução do programa na CPU, a transmissão dos dados via rede e o processamento nas GPUs remotas. Assim, seria importante validar a proposta em trabalhos futuros como suporte à execução de aplicações de domínios diversos.

REFERÊNCIAS

ALTERA. OpenCL for Altera FPGAs: Accelerating Performance and Design Productivity, 2012. Disponivel em: <http://www.altera.com/products/software/opencl/opencl-index.html>. Acesso em: 29 dez. 2012.

ASTRA TEAM, V. Fastra, GPU SuperPC, Maio 2008. Disponivel em: <http://fastra.ua.ac.be>. Acesso em: Maio 2011.

ASTRA TEAM, V. Fastra II, abr. 2009. Disponivel em: <http://fastra2.ua.ac.be>. Acesso em: Dezembro 2011.

BARAK, A. E. B.-N. A Package for OpenCL Based Heterogeneous Computing on Clusters with Many GPU Devices, 2010.

BARAK, A.; SHILOH, A. **MOSIX Cluster Operating System**, 1999. Disponivel em: <http://www.mosix.org>. Acesso em: 07 abr. 2013.

DANALIS, A. et al. **The Scalable Heterogeneous Computing (SHOC) benchmark suite**. Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units. New York, NY, USA: ACM. 2010. p. 63--74.

DAWES, B.; ABRAHAMS, D.; RIVERA, R. Boost C++ Libraries, 2004. Disponivel em: <http://www.boost.org/>.

DUATO, J. et al. **rCUDA:** Reducing the number of GPU-based accelerators in high performance clusters. High Performance Computing and Simulation (HPCS), 2010 International Conference on. [S.l.]: [s.n.]. 2010a. p. 224-231.

DUATO, J. et al. **An Efficient Implementation of GPU Virtualization in High Performance Clusters**. Proceedings of the 2009 international conference on Parallel processing. [S.l.]: Springer Berlin Heidelberg. 2010b. p. 385--394.

FAN, Z. E. Q. GPU Cluster for High Performance Computing, 2004.

FLYNN, M. Some Computer Organizations and Their Effectiveness. **Computers, IEEE Transactions on**, v. C-21, n. 9, p. 948-960, 1972.

FRIEDEMANN A. RÖßLER, T. W.; ERTL, T. Distributed video generation on a GPU-cluster for the web-based analysis of medical image data, 2007.

GAMMA, E. et al. **Design Patterns:** Elements of Reusable Object-Oriented Software. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 1995.

INFINIBAND TRADE ASSOCIATION. Home page. **Infiniband Trade Association**, 2013. Disponivel em: <http://www.infinibandta.org/>. Acesso em: 9 junho 2013.

INTEL. Intel® MPI Library 4.1. **Intel® Developer Zone**, 2013. Disponivel em: <http://software.intel.com/en-us/intel-mpi-library>. Acesso em: 02 jan. 2013.

KHRONOS GROUP. **The OpenCL Specification - Version 1.1**. Khronos OpenCL Working Group. [S.l.], p. 385. 2010.

KHRONOS GROUP. **The OpenCL Extension Specification**. Khronos OpenCL Working Group. Beaverton, p. 113. 2011. (OpenCL versão 1.2).

KHRONOS GROUP. OpenCL Conformant Products. **Khronos Groups**, 2012. Disponivel em: <http://www.khronos.org/conformance/adopters/conformant-products#opencl>. Acesso em: 29 dez. 2012.

LUKE-JR. BFGMiner Announce, 2012. Disponivel em: <https://bitcointalk.org/index.php?topic=78192.0>. Acesso em: 20 jan. 2013.

LUXRENDER PROJECT. LuxMark, 2012. Disponivel em: <http://www.luxrender.net/wiki/LuxMark>. Acesso em: 16 Março 2013.

LUXRENDER PROJECT. LuxRender, 2013. Disponivel em: <http://www.luxrender.net/en_GB/index>. Acesso em: 16 Março 2013.

MCCALPIN, J. D. Memory Bandwidth and Machine Balance in Current High Performance Computers. **IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter**, p. 19--25, dec 1995.

NVIDIA CORPORATION. **NVIDIA CUDA C Programming Guide - Version 3.2**. [S.l.]: [s.n.], 2010a.

NVIDIA CORPORATION. **OpenCL Best Practices Guide**. [S.l.]: [s.n.], 2010b.

NVIDIA CORPORATION. **OpenCL Programming Guide for the CUDA Architecture - Version 3.2**. [S.l.]: [s.n.], 2010c. 6 p.

OPENMPI. OpenMPI, 2013. Disponivel em: <http://www.open-mpi.org/>. Acesso em: 02 jan. 2013.

PANETTA, J. et al. Accelerating Kirchhoff Migration by CPU and GPU Cooperation, 2009.

PENNYCOOK, S. J. E. H. Performance Analysis of a Hybrid MPI/CUDA Implementation of the NAS-LU Benchmark, 2010.

SVOBODOVA, L. Implementing OSI systems. **Selected Areas in Communications, IEEE Journal on**, v. 7, n. 7, p. 1115-1130, 1989.

TANENBAUM, A. S.; STEEN, M. **Distributed Systems:** Principles and Paradigms. 2ª. ed. [S.l.]: Prentice-Hall, Inc., 2006.

THE BITCOIN FOUNDATION. Bitcoin, 2012. Disponivel em: <http://bitcoin.org/en/>. Acesso em: 20 jan. 2013.

THOMPSON, C. J. E. H. Using modern graphics architectures for general-purpose computing: a framework and analysis, p. 306-317, 2002. ISSN 0-7695-1859-1.

TUPINAMBÁ, A. DistributedCL. **SourceForge**, jan. 2011. Disponivel em: <http://sf.net/projects/distributedcl>.

TUPINAMBÁ, A.; SZTAJNBERG, A. **DistributedCL:** A Framework for Transparent Distributed GPU Processing Using the OpenCL API. Computer Systems (WSCAD-SSC), 2012 13th Symposium on. [S.l.]: [s.n.]. 2012. p. 187-193.