# DistributedCL: a framework for transparent distributed GPU processing using the OpenCL API

André Tupinambá
Programa de Engenharia Eletrônica – PEL
Universidade do Estado do Rio de Janeiro – UERJ
Rio de Janeiro, Brasil
e-mail: andrelrt@gmail.com

Alexandre Sztajnberg
Programa de Engenharia Eletrônica – PEL
Instituto de Matemática e Estatistica – IME
Departamento de Informática e Ciência da Computação
Universidade do Estado do Rio de Janeiro – UERJ
Rio de Janeiro, Brasil
e-mail: alexszt@ime.uerj.br

***Abstract*—This paper presents the DistributedCL, a framework that provides the applications developed using the OpenCL interface location-transparent GPU processing. The application can explore distributed processing with no modification. The architecture of the framework and the programming model are presented, and the possible performance bottlenecks are discussed.**

***Keywords-OpenCL; GPGPU; framework; distributed systems***

## I. INTRODUÇÃO

Com o intuito de realizar pesquisas com reconstrução tomográfica, o grupo ASTRA montou com sucesso um computador com 8 GPUs, que chegou a capacidade de processamento equivalente ao do cluster de processamento usado anteriormente [1]. Para utilizar um algoritmo mais complexo de reconstrução, o grupo criou um segundo equipamento, desta vez com 13 GPUs [2]. Porém, o funcionamento correto só foi possível após alterações na BIOS e no *kernel* do Linux.

Ainda que tais problemas tenham sido resolvidos, existe um limite físico de quantas placas podem ser colocadas em um único computador. Por exemplo, a placa mãe utilizada pelo projeto possui espaço para no máximo sete placas de vídeo, cada uma contendo no máximo 2 GPUs. Torna-se então necessária outra abordagem para utilizar o poder de processamento de várias GPUs.

A solução mais imediata é distribuir o programa através de uma rede de computadores, cada um com o seu conjunto de GPUs. Para realizar esta distribuição, podem-se utilizar mecanismos e *frameworks* já estabelecidos. Porém esta abordagem eleva a complexidade do programa, pois é necessário conciliar a distribuição do problema através do framework escolhido e o desenvolvimento em GPU.

Uma alternativa é utilizar um *framework* de distribuição com a mesma interface da OpenCL. Assim, uma aplicação preparada para utilizar a OpenCL pode executar de forma distribuída, sem necessidade de alteração do seu código.

Neste artigo apresentamos a proposta de um *framework* de processamento distribuído de GPUs com a interface de programação da OpenCL, que chamamos DistributedCL, com as características discutidas acima.

O restante deste artigo está assim organizado: a Seção II descreve o modelo de programação de GPUs e trabalhos relacionados; a Seção III analisa a API OpenCL e questões de implementação do framework de processamento distribuído em GPU, com a interface de programação da OpenCL; a Seção IV analisa os possíveis impactos negativos ao desempenho das aplicações e alternativas de contorno; a Seção V apresenta o framework DistributedCL e como ele implementa as alternativas para melhora do desempenho e o modelo de programação; a Seção VI apresenta uma avaliação de desempenho do framework; e a Seção VII apresenta observações finais.

## II. CONCEITOS E TRABALHOS RELACIONADOS

Há alguns anos, poucas placas de vídeo eram capazes de algum processamento. Hoje as GPUs possuem um poder de processamento da ordem de TeraFlops, superando os processadores de uso geral, que estão na faixa de 100 GigaFlops [3].

Originalmente, este poder não estava disponível para processamento de uso geral, as GPUs evoluíram para outro propósito: a geração de gráficos 3D para aplicações de realidade virtual, simulação e jogos de computador. Para efetuar o processamento gráfico com eficiência, sua arquitetura interna foi desenvolvida para realizar cálculos vetoriais e matriciais, de ponto flutuante com alto grau de paralelismo.

Porém, cálculos matemáticos são usados também em outras áreas de conhecimento. A GPU passou então ser utilizada para cálculos genéricos, utilizando as APIs de geração de gráfico, como em [4].

A utilização de uma API específica para este processamento veio principalmente com a fabricante Nvidia e sua plataforma CUDA. Esta plataforma apresenta um modelo bem definido de programação e arquitetura de hardware, expondo uma API específica para processamento em GPUs [5]. Posteriormente os outros fabricantes, como ATI e IBM, também disponibilizaram produtos equivalentes.

Uma dessas iniciativas foi o modelo genérico de acesso a GPUs da Apple, posteriormente padronizado pelo Khronos Group, a OpenCL [6]. A OpenCL define um modelo de dispositivo genérico (podendo ser GPU, CPU ou um acelerador, como o IBM Cell/B.E.), uma API de acesso e a linguagem de programação própria.

### A. Processamento distribuído em GPUs

Com a necessidade de distribuir o processamento realizado em GPU, vários sistemas foram desenvolvidos utilizando mecanismos de distribuição de programas já

consolidados. Uma alternativa recorrente é utilizar os *frameworks* de distribuição mais conhecidos, como o MPI, por exempl, em [7], [8], [9] e [10].

Porém, ao utilizar esta abordagem é necessário combinar dois modelos diferentes de distribuição em um mesmo programa: um específico relacionado à GPU, que necessita da divisão do problema em pequenos blocos de execução independentes; e outro relacionado ao MPI, que executa o programa em um grupo de computadores interligados usando troca de mensagens.

O uso das duas tecnologias em conjunto aumenta a complexidade do programa como um todo. A programação da GPU e a MPI, possuem modelos distintos de tratamento de dados, sincronização e execução que precisam ser conciliados.

## III. Processamento distribuído via OpenCL

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.

O modelo de processamento de GPUs é comparável ao processamento distribuído sem memória compartilhada. Cada GPU é tratada isoladamente como uma unidade de processamento, sua memória não é acessível à CPU ou outras GPUs e a comunicação entre eles é realizada através de troca de mensagens, que passam através do barramento do computador. Isso é um indicativo que este modelo poderia ser viável para distribuir também o processamento entre computadores.

A 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 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 OpenCL, cada GPU trabalha de forma isolada das demais e sua comunicação com a aplicação é realizada somente através de troca de mensagens, existe a oportunidade para utilizar a API OpenCL como interface de um *framework* de processamento distribuído de GPUs, como feito pelo Mosix Virtual OpenCL [11]. Este *framework* oferece suporte à comunicação entre aplicação e GPU, através de troca de mensagens através da rede, ao invés do barramento local.

### A. Características da API OpenCL

A interface de programação da OpenCL pode ser classificada em três categorias: de plataforma, de comunicação e de contexto.

A API de plataforma provê as aplicações um meio para descobrir os dispositivos disponíveis e suas características, que na OpenCL podem ser uma GPU, CPU ou um chip acelerador.

A API de comunicação provê às aplicações a criação das filas de comando, exclusiva de cada dispositivo; por transferir dados de objetos; e comandar 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 pelas filas de comando de forma assíncrona e, dependendo do dispositivo, 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 para todos os dados usados pela API. São tratados como objetos: os buffers de memória; as imagens 2D e 3D; os *samplers*; e os programas e *kernels*. Um detalhe interessante é que para a criação de programas, o código fonte é 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. Além disso, os buffers de dados precisam ser enviados previamente ao dispositivo, e para a execução somente seu ponteiro é passado como parâmetro. Isto permite que o conjunto de dados seja enviado ao dispositivo, mesmo que ele não seja tratado de uma só vez.

### B. Acesso remoto à API OpenCL

Um modelo no estilo 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. Sendo que as características físicas do equipamento não se alteram, os valores retornados por ela, de uma forma geral, são constantes. Assim, é possível ler todas as características em um só momento e criar um *cache* com essas informações, evitando transmissões de chamadas via rede. Existem algumas exceções a esta regra, mas a maioria dos dados do dispositivo pode ser tratada desta forma.

A API de comunicação é responsável pela transmissão de dados através do barramento. Para um modelo distribuído, ela deve ser escrita para criar filas de comando remotas, simulando as filas de comando da OpenCL. Como as filas são todas assíncronas, essa API pode armazenar os comandos e aguardar um comando de sincronização, `clFlush` por exemplo, para envio de todos os comando ao *stub* via rede. Especificamente, 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 [12]. Podemos assumir então que as aplicações terão este comportamento, assim os dados poderiam ser enviados pela rede o quanto antes, pois seu envio pode ser realizado de forma assíncrona com a preparação de comandos para execução dos *kernels*.

A API de contexto permite a criação e manipulação de objetos. Normalmente várias chamadas dessa API são executadas em sequência, sendo os objetos somente

utilizados depois de todos os passos. Essa API poderia aguardar um conjunto de comandos, conhecidamente usados em sequência, para transmissão em conjunto pela rede, minimizando o tráfego.

### C. Diversos dispositivos remotos

Apesar de funcionar corretamente, um modelo estilo RPC para o *framework* obrigaria a aplicação gerenciar várias instâncias da biblioteca OpenCL para realizar processamento em vários computadores, algo que complicaria a implementação. 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 simples biblioteca para a aplicação.

A OpenCL já prevê o uso de múltiplos dispositivos, logo o *framework* para processamento distribuído via OpenCL deverá ser capaz de apresentar todos os dispositivos disponíveis através de sua interface.

As filas de comando são exclusivas de um dispositivo, logo a API de comunicação não necessita de tratamento. O único tratamento necessário é o de se associar, no momento da criação, o dispositivo escolhido à conexão de rede correta.

Já para a API de contexto temos necessidade de criar contextos e objetos compostos, que serão retornados à aplicação, para gerenciar os contextos e objetos de cada conexão remota. Em cada chamada deve ser identificada qual conexão remota deve ser chamada e selecionar o contexto e objeto correspondente.

## IV. Impactos no desempenho

A medição de transmissão entre uma placa GeForce 8800 Ultra [13] e o computador, na melhor das condições, obteve uma taxa de 3.182MB/s. Para a transferência de 1.000*MB*, obtém-se um tempo total de 0,314*s* (1).

Porém o tempo de transferência através de uma rede é significativamente maior. Mesmo considerando a velocidade nominal da Ethernet 1*Gb* temos um tempo total de 8*s* (2).

$$1000MB \,/\, 3182MB/s = 0{,}314s \quad (1)$$

$$8000Mb \,/\, 1000Mb/s = 8s \quad (2)$$

É preciso então encontrar um modo de diminuir o impacto no desempenho desta transmissão, tirando vantagem de algumas características da API OpenCL.

### A. Minimizando o tráfego de dados

Minimizar o tráfego de dados na rede é o primeiro passo para diminuir o impacto no desempenho. É preciso encontrar um meio para tornar a passagem de parâmetros através da rede mais eficiente.

A API de plataforma é a mais simples, composta por apenas 4 funções, 2 para enumerar plataformas e dispositivos; e 2 para retornar as características desses. Como estes dados, na maior parte, são constantes, o melhor meio é transmitir todas as informações de uma só vez e criar um *cache* com estes dados. Assim, após os dados carregados, todas as chamadas serão respondidas localmente.

Exceto pela transmissão de imagens e buffers, as funções da API possuem um pequeno conjunto de dados a serem transmitidos. Analisando a API OpenCL, cada função recebe um ou dois objetos do sistema como parâmetro. A API OpenCL define os identificadores de objetos como ponteiros, que são opacos para aplicação, ou seja, não são manipulados. Estes ponteiros possuem o tamanho de 32 ou 64 bits de acordo com a arquitetura do computador utilizado, porém, para a transferência de rede, este identificador deverá ser padronizado. É possível, então, utilizar um identificador de tamanho menor, como por exemplo, 16 bits, reduzindo o volume de dados transmitidos. Isto reduz o número de objetos possíveis para 65535, que é um número inicialmente viável.

### B. Fila de comando

Toda a transmissão entre aplicativo e dispositivo é feita pelas filas de comando. Como descrito anteriormente, as filas de comando são sempre assíncronas, podendo ter execução fora de ordem. Segundo o guia de programação da OpenCL da AMD [14], os comandos são mantidos na memória do computador e são enviados aos grupos ao dispositivo para execução.

Aproveitando essas características, o *framework* de distribuição pode enviar os comandos de forma assíncrona ao programa e também realizar um armazenamento temporário dos comandos, para minimizar as transferências de rede.

## V. *Framework* DistributedCL

O DistributedCL [15] é um *framework* de processamento distribuído em GPUs com a interface da OpenCL que implementa as melhorias propostas na Seção 4.

Para ser compatível com processadores de várias arquiteturas, vários sistemas operacionais e ainda possuir um bom desempenho, o *framework* DistributedCL 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 questões como o tratamento de threads e comunicação entre processos podem ser mais bem tratados com classes específicas. Para isso, o *framework* DistributedCL utiliza as bibliotecas *boost* [16], que já tratam estas questões.

Outro ponto de atenção é a arquitetura do processador e sistema operacional. Para a questão de arquitetura do processador, o código do framework DistributedCL não utilizou os tipos típicos `int` ou `long`, dando preferência os novos tipos definidos pelo C99 que indicam exatamente o número de bit, como o `int32_t` e `uint64_t`. O framework DistributedCL foi testado em sistemas Windows e Linux nas versões de 32 e 64 bits e é distribuído como software livre no site *sourceforge.net*.

A arquitetura do *framework* DistributedCL é composta por camadas com seus papéis bem definidos. Na Figura 1 temos a estrutura e a comunicação entre as camadas.

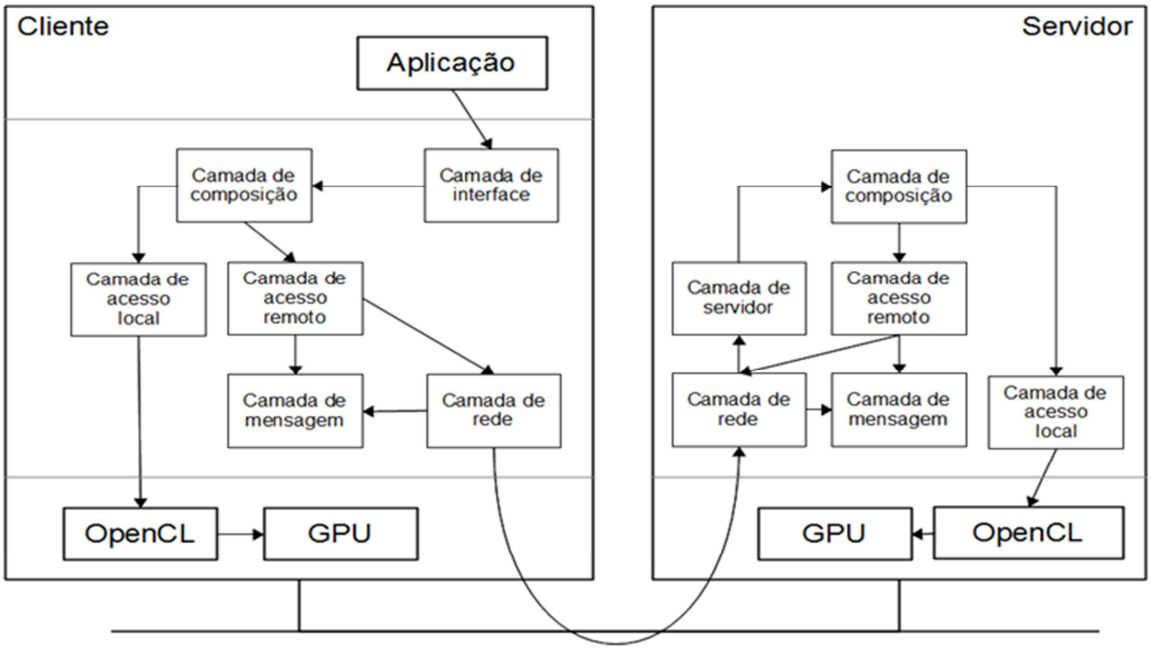


Figure 1. Arquitetura do framework DistributedCL

Estas camadas possuem uma similaridade com as camadas do NFS [17]. 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 DistributedCL também não requer do programador uma mudança no programa.

Diferente do NFS, entretanto, o *framework* não utiliza RPC, principalmente pelas chamadas assíncronas e armazenamento temporário de comandos, onde várias chamadas da API OpenCL são enviadas ao servidor em uma única mensagem da camada de rede.

### A. *Camada de interface.*

A camada de interface é responsável por prover às aplicações a API OpenCL e traduzir as chamadas para os objetos da camada de composição. Ela também é responsável pela contagem de referência dos objetos da OpenCL.

Esta camada é construída como uma biblioteca dinâmica e provê as funções definidas no padrão da OpenCL para acesso das aplicações. Todas as funções da API OpenCL estão implementadas nesta camada e fazem simplesmente a tradução dos parâmetros passados para as chamadas necessárias nas classes da camada de composição para execução. Estas funções também verificam alguns parâmetros passados de acordo com a especificação da OpenCL e as características do *framework* DistributedCL.

Esta camada também é responsável por suportar a extensão ICD definida pelo Khronos Group [18]. Como definido por esta extensão, todos os objetos OpenCL devem ser endereços de memória e devem possuir como o primeiro membro um ponteiro para uma tabela onde estão todas as funções da OpenCL suportadas pela biblioteca.

### B. *Camada de composição*

A camada de composição é responsável por criar a abstração de uma única plataforma OpenCL para as camadas de interface e servidor. Para isso, ela possui todas as funcionalidades da OpenCL e repassa as chamadas para o acesso local e acesso remoto. Esta camada também administra as instâncias das classes de acesso local e acesso remoto, seus dispositivos, contextos, filas de comando e demais objetos.

Uma atribuição desta camada é carregar as instâncias das classes de acesso local e acesso remoto. Para isso, esta camada provê às camadas superiores o serviço de adicionar uma biblioteca local ou conectar a um servidor remoto. Estas chamadas executam estes serviços criando novas instâncias das classes de acesso local e acesso remoto, respectivamente, e disponibilizando seus recursos para as camadas superiores.

A principal tarefa desta camada é direcionar as chamadas vindas das camadas de interface e servidor para as respectivas camadas de acesso local e acesso remoto. Esta escolha pode ser feita de duas maneiras: (i) repassando a chamada para todas as instâncias carregadas ou (ii) selecionando qual instância deverá ter a chamada repassada, sendo que esta escolha depende exclusivamente do tipo da mesma.

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 instância a chamada deverá ser repassada.

### C. *Camada de acesso local*

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

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.

### D. Camada de acesso remoto

A camada de acesso remoto é responsável por acessar a camada de servidor em outro computador através da rede. A camada de acesso remoto possui a mesma interface das camadas de composição e de acesso local e provê as funcionalidades da OpenCL através de suas classes. A camada de acesso remoto recebe as requisições da camada de composição, cria objetos da camada de mensagens e os enviam através da camada de rede.

Também é responsabilidade desta camada a decisão se o envio da mensagem será síncrono ou assíncrono, como descrito na Seção 4.2. Esta decisão é feita pelo tipo da mensagem (normalmente as funções `clEnqueue*` da API OpenCL podem ser assíncronas) e pelos parâmetros passados.

A principal funcionalidade da camada de acesso remoto é a criação dos objetos de mensagem e a chamada da camada de rede para transporte das mensagens para a camada de servidor em outro computador. Para isso, todos os objetos desta camada possuem um identificador de 16 bits que é utilizado nas mensagens. As classes da camada de mensagem manipulam somente estes identificadores de 16 bits, não tendo nenhum acesso às classes de acesso remoto.

### E. Camada de mensagens

A camada de mensagens é responsável pelos dados passados pela rede e pela serialização (*marshalling*) destes dados. Também é responsabilidade desta camada a criação e tratamento de pacotes de mensagens.

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 classe de 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 classe `dcl_message<msgGetDeviceInfo>` equivalente, trata todas as informações em uma única chamada.

### F. 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 é capaz de utilizar diferentes protocolos de transporte confiável. Também é responsabilidade da camada de rede o estabelecimento e manutenção da sessão.

A camada de rede é separada em duas partes, o cliente e o servidor, sendo ambas preparadas para utilizar qualquer protocolo de transporte confiável. Para isso, ambas são construídas como classes *templates* que recebem como parâmetro a classe de tratamento do protocolo de rede. No caso do *framework* DistributedCL somente foi utilizado o TCP.

O servidor desta camada 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. Sempre que o envio de uma mensagem é solicitado, todas as mensagens anteriormente enfileiradas são enviadas.

### G. Camada de servidor

A camada de servidor é responsável por tratar as mensagens recebidas pela camada de rede e executá-las através de chamadas aos métodos das classes da camada de composição.

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* [19], recebendo a mensagem no construtor e tendo somente o método `execute()` que trata a mensagem e salva o resultado da execução de volta no próprio objeto de mensagem.

### H. Modelo de Programação e Configuração

Como discutido anteriormente o DistributedCL não introduz um novo modelo para programação distribuída de GPUs. O modelo convencional é usado normalmente. As funções das camadas de Interface e Composição oferecem a transparência necessária em tempo de execução, juntamente com a configuração de máquinas e GPUs, definidas em tempo de configuração, mapeando os recursos físicos para um domínio de referências lógico.

O trecho de código apresenta uma chamada que aciona as GPUs disponíveis. Na linha 1 descobre-se quais são as GPUs disponíveis, criando então um contexto de trabalho com elas na linha 2. Nas linhas 3 e 4 são criadas filas de comando para cada GPU. As linhas 5, 6 e 7 criam um *kernel* para ser executado em cada GPU pelas linhas 8 e 9. A linha 10 aguarda o fim de execução de todos os *kernels.*

```
1: clGetDeviceIDs(plat, CL_DEVICE_TYPE_GPU,
                  cnt, devs, NULL);
2: ctx = clCreateContext(0, devcnt, devs,
                  NULL, NULL, &err);
3: for(int i = 0; i < cnt; ++i)
4:    queue[i] = clCreateCommandQueue(ctx,
                        devs[i], 0, &err);
5: prg = clCreateProgramWithSource(ctx, 1,
        (const char**)&src, &srclen, &err);
6: clBuildProgram(prg, 0, NULL, NULL,
                   NULL, NULL);
7: kernel = clCreateKernel(prg, "test",
                           &err);
8: for( int i = 0; i < cnt; ++i )
9:    clEnqueueNDRangeKernel( queue[i],
               kernel, 1, 0, global, local,
               0, NULL, &evnt[i] );
10: clWaitForEvents( devcnt, evnt );
```

Este exemplo de funcionamento da OpenCL é padrão da biblioteca para múltiplas GPUs. Na abordagem do DistributedCL, a configuração pode ser definida associando GPUs locais, ou GPUs remotas localizadas em

máquinas diferentes à lista de GPUs disponíveis. Na implementação do protótipo avaliado na Seção VI é disponibilizado um arquivo de configuração que pode ser redefinido para cada aplicação.

## VI. AVALIAÇÃO DE DESEMPENHO

Para avaliar o desempenho do DistributedCL, utilizou-se o teste S3D do benchmark SHOC [20] específica para testes com bibliotecas OpenCL e CUDA. Nenhuma modificação à ferramenta SHOC foi realizada, sendo a distribuição do processamento feita de forma transparente com uma nova configuração administrativa, da mesma maneira que um administrador de rede faria com o NFS, exportando e montando diretórios.

Cada maquina possui a seguinte configuração:

- placa mãe Intel® DX58SO;
- processador Intel® Core™ i7-940;
- 8GB DDR3 de memória principal;
- interface de vídeo NVIDIA GeForce GTX 480 com 1GB GDDR5; e
- duas interfaces de rede Gigabit Ethernet.

Em cada computador está instalado o Linux Ubuntu 12.04 long-term support 64bits, com os seus pacotes nas versões mais atuais, e os *drivers* para o vídeo da NVIDIA versão 295.53. 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*.

O teste S3D do SHOC implementa um algoritmo de simulação de combustão, considerado no benchmark como sendo uma boa aproximação de carga normal de uma aplicação utilizando GPUs. No SHOC este teste utiliza ponto flutuante de precisão simples (S3D-SP) e de precisão dupla (S3D-DP). O resultado é mostrado em GFLOPS.

Para a comparação, a execução do teste S3D do SHOC foi feita, em quatro configurações (Tabela I). Nas configurações *opencl*, *dcl_local* e *dcl_loopback*, apenas o teste local, em um coputador foi utilizado. Na configuração *dcl_remote* foram utilizados dois computadores: um local e outro remoto.

TABLE I. CONFIGURAÇÕES DO TESTE S3D DO SHOC

| Configuração | Característica |
|---|---|
| opencl | Sem passar pelo framework. |
| dcl_local | Usa a camada de acesso local do framework. |
| dcl_lookback | Usa a camada de acesso remoto, com o servidor na mesma máquina, loopback. |
| dcl_remote | Usa a camada de acesso remoto, a outro computador. |

O resultado do teste está na Figura 2. O gráfico está com um intervalo de confiança de 95%.

Os testes com a configuração *opencl*, *dcl_local* e *dcl_loopback* mostraram ter o mesmo desempenho, não tendo diferenças expressivas. Destaque apenas para a configuração *dcl_loopback* na execução com S3D-SP, que apresenta uma média menor, com o intervalo de confiança maior, porém comparável aos outros dois, podendo-se então considerar o resultado como equivalente.

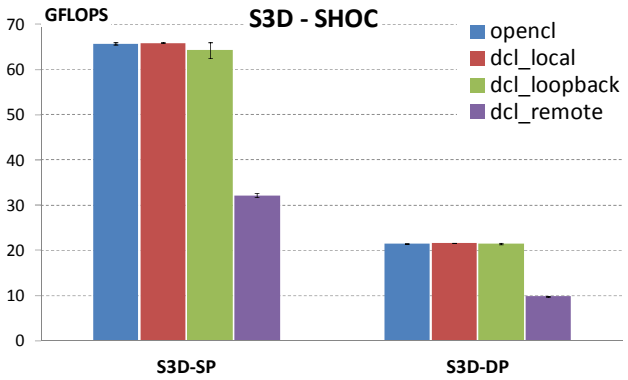


Figure 2. Teste de desempenho com SHOC

Por observação, podemos assumir que o *framework* DistributedCL apresenta pouco overhead para a aplicação, mesmo utilizando TCP/IP na mesma máquina. Além disso, a execução correta em todos os quatro testes confirma a transparência ao programador, que utiliza o modelo de programação convencional da API do OpenCL (sem recompilação de código).

A configuração *dcl_remote* mostra um desempenho cerca de 50% menor do que as outras configurações, tanto no S3D-SP, quanto no S3D-DP. Essa diferença de desempenho deve ser creditada ao overhead da passagem de dados através da rede, como era previsto na Seção IV.

Apesar da queda do desempenho, a utilização de múltiplas GPUs em rede aumenta o poder de processamento de uma aplicação, mesmo tendo o *overhead* da transmissão de dados em rede. Entretanto para aplicações pequenas, com poucos cálculos, este *overhead* pode ser significativo no tempo total de execução. Está sendo construída uma análise de desempenho, com diferentes configurações e topologias de interligação de máquinas, mostrando a partir de que ponto a carga de processamento torna a utilização do framework DistributedCL mais eficiente do que a utilização somente da GPU local.

## VII. CONCLUSÃO

Apesar de ser possível utilizar um mecanismos robustos como o RPC para acessar remotamente a API OpenCL para processamento distribuído em GPU, a mudança de estilo de programação pode impor uma carga de trabalho desnecessária. O programador OpenCL já assume a responsabilidade de explorar o paralelismo das aplicações e distribuir a carga de processamento por várias GPUs. Neste caso, ele ainda teria que adicionar às suas responsabilidades a definição da IDL e a programação dos módulos cliente e servidor RPC, por exemplo, para usar GPUs remotas. Além disso, não seria possível otimizar a comunicação.

A abordagem do *framework* DistributedCL é tornar transparente ao programador o uso de GPUs em qualquer configuração local ou remota. Assim, o modelo de programação é o mesmo adotado para o uso da OpenCL.

Um estudo do comportamento da biblioteca OpenCL permitiu identificar vários pontos onde é possível explorar uma combinação de *cache* de dados e o assincronismo na comunicação para reduzir o tráfego na rede.

*O framework* DistributedCL está em estágio avançado de desenvolvimento. Outras possibilidades de redução no

tráfego estão sendo investigadas, como por exemplo, a compressão de dados.

Uma outra possibilidade atualmente sendo investigada para a redução do tráfego, é a criação de uma extensão da API OpenCL para incluir a semântica de arquivos, onde cada servidor faz a leitura direta dos arquivos, com o cliente passando somente o caminho através da rede. Com esta semântica é possível explorar ambientes de *cluster* com *storage* dedicado ou ambientes com sistemas de arquivos distribuídos como as Grades Computacionais.

## AGRADECIMENTOS

Os autores gostariam de agradecer ao projeto Cooperação entre as Pós-Graduações de Computação Científica LNCC-UERJ / Faperj pela disponibilidade do cluster de GPUs no LabIME/IME/UERJ.

## REFERÊNCIAS

[1] ASTRA Team, VisionLab, Fastra, GPU SuperPC, http://fastra.ua.ac.be, visitado em 04/2010.

[2] ASTRA Team, VisionLab, Fastra II, http://fastra2.ua.ac.be, visitado em 04/2010.

[3] NVIDIA Corporation, OpenCL Programming Guide for the CUDA Architecture - Version 3.2, 2010.

[4] Thompson, C. J., Hahn, S. e Oskin, M., Using modern graphics architectures for general-purpose computing: a framework and analysis, 35th ACM/IEEE Intl. symposium on Microarchitecture (MICRO 35). pp. 306-317, 2002.

[5] NVIDIA Corporation, NVIDIA CUDA C Programming Guide - Version 3.2, 2010.

[6] Khronos Group, The OpenCL Specification - Version 1.1, 2010.

[7] Friedemann A. Rößler, Torsten Wolff, Sabine Iserhardt-Bauer, et all, Distributed video generation on a GPU-cluster for the web-based analysis of medical image data, Medical Imaging 2007: Visualization and Image-Guided Procedures. SPIE, Volume 6509, 2007.

[8] Fan, Z.; Qiu, F.; Kaufman, A. et all, GPU Cluster for High Performance Computing, ACM/IEEE Conference on Supercomputing (SC '04), 2004.

[9] Panetta, J., Teixeira, T., de Souza Filho, P.R.P., et all, Accelerating Kirchhoff Migration by CPU and GPU Cooperation, SBAC-PAD '09, pp. 26-32, 2009.

[10] Pennycook, S.J., Hammond, S.D., et all, Performance Analysis of a Hybrid MPI/CUDA Implementation of the NAS-LU Benchmark, SIGMETRICS Perform. Eval. Rev. 38, 4, pp. 23-29, 2011.

[11] Barak, A., Ben-Nun, T., Levy, E. e Shiloh, A., A Package for OpenCL Based Heterogeneous Computing on Clusters with Many GPU Devices, IEEE International Conference on Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010

[12] NVIDIA Corporation, OpenCL Best Practices Guide, 2010

[13] Schaa, D. e Kaeli, D., Exploring the Multiple-GPU Design Space, IEEE IPDPS '09, pp. 1-12, 2009

[14] AMD Corporation, ATI Stream SDK OpenCL Programming Guide, 2010

[15] Tupinambá, A., DistributedCL, http://sourceforge.net/ projects/distributedcl/, visitado em 08/2012

[16] Boost C++ Libraries, http://www.boost.org, visitado em 08/2012

[17] Tanenbaum, A.S., e Steen, M., Distributed Systems: Principles and Paradigms (2nd Edition), Prentice-Hall, Inc., 2006

[18] Khronos Group, The OpenCL Extension Specification. Khronos OpenCL – Version 1.2, p. 113. 2011.

[19] Gamma, E., et all, Design patterns: elements of reusable object-oriented software, Addison-Wesley Longman Publishing Co., Inc., 1995

[20] Danalis, A., et all, The Scalable Heterogeneous Computing (SHOC) benchmark suite, 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU '10). ACM, pp. 63-74, 2010.