# Programação em OpenCL: Uma introdução prática

**César L. B. Silveira[1], Luiz G. da Silveira Jr.[2], Gerson Geraldo H. Cavalheiro[3]**

[1]V3D Labs
São Leopoldo, RS, Brasil

[2]Universidade do Vale do Rio dos Sinos
São Leopoldo, RS, Brasil

[3]Universidade Federal de Pelotas
Pelotas, RS, Brasil

{cesar,gonzaga}@v3d.com.br, gerson.cavalheiro@ufpel.edu.br

***Abstract.*** *OpenCL is an open standard for high-performance computing in heterogeneous computational environments composed of multicore CPUs and many-core GPUs. OpenCL makes it possible to write multi-platform code for such devices, without the need to use vendor-specific languages and tools. Focused on parallelism, programming in OpenCL is based on writing functions that are executed in multiple simultaneous instances. This minicourse aims to introduce key concepts and explore the architecture defined by the OpenCL standard.*

***Resumo.*** *OpenCL é um padrão aberto para programação de alto desempenho em ambientes computacionais heterogêneos equipados com CPUs multicore e GPUs many-core. OpenCL torna possível a escrita de código multi-plataforma para tais dispositivos, sem a necessidade do uso de linguagens e ferramentas específicas de fabricante. Com foco no paralelismo, a programação em OpenCL baseia-se na escrita de funções que são executadas em múltiplas instâncias simultâneas. Este minicurso visa introduzir conceitos-chave e explorar a arquitetura definida no padrão OpenCL.*

## 1. Introdução

O processamento paralelo é hoje uma realidade em praticamente todos os segmentos que fazem uso de sistemas computacionais. Este fato deve-se aos avanços tecnológicos que permitiram baratear o custo das opções de hardware paralelo, como os processadores *multicore* e as placas gráficas disponibilizando até centenas de processadores (*stream processors*). Como consequência, mesmo usuários domésticos possuem configurações dotadas de múltiplas unidades de processamento destinadas aos mais prosaicos usos, como navegação na Internet, processamento de texto e jogos. O problema que se coloca é capacitar recursos humanos a utilizar de forma efetiva todo este potencial de processamento disponível para computação de alto desempenho.

O emprego de processadores *multicore* para o processamento paralelo não é, na verdade, uma novidade no mercado do desenvolvimento de software. Ressalvas feitas,

contudo, à possibilidade de tirar benefícios deste hardware específico explorando suas características próprias [Cavalheiro and dos Santos 2007]. Por outro lado uso tradicional das placas gráficas, como seu nome já indica, esteve durante muito tempo associado às aplicações de computação gráfica e/ou processamento de imagens. Nos últimos anos, este tipo de arquitetura tem sido considerada para implementações de aplicações científicas em um escopo mais genérico, uma vez que seus processadores, chamados de GPU (*Graphics Processing Unit* – Unidade de Processamento Gráfico), dispõem de grande capacidade de processamento. O uso de placas gráficas, ou simplesmente, de GPUs, justifica-se em função do desempenho obtido pelo investimento realizado [Genovese et al. 2009], o qual é altamente favorável ao usuário final. Como resultado, diversas aplicações com alto grau de demanda computacional, como dinâmica molecular [Amos G. Anderson 2007] e métodos de elementos finitos [Göddeke et al. 2007], entre outras, têm recebido soluções implementadas sobre GPUs. Os jogos de computadores têm se beneficiado tanto do poder de processamento gráfico das GPUs quanto do potencial para computação genérica destes dispositivos, para a realização de simulações de fenômenos físicos, detecção de colisões, simulação de fluidos, entre outros. O uso da GPU para processamento de propósito geral possibilidade ganhos significativos de desempenho, permitindo o desenvolvimento de jogos e aplicações mais realistas. Além disso, o poder de processamento paralelo das GPUs pode também ser empregado no processamento de áudio e na execução de algoritmos inteligência artificial. Atualmente o mercado está sendo suprido por ferramentas para programação neste tipo de arquitetura, como a plataforma CUDA [NVIDIA Corporation 2010], disponível para várias linguagens de programação, e OpenCL [Apple Inc. 2009, Khronos Group 2010a].

A presença maciça de configurações de computadores dotados de processadores *multicore* e placas gráficas dotadas de múltiplas unidades de processamento indica a necessidade de uma convergência dos esforços de desenvolvimento de software. Tais configurações são ditas **heterogêneas**, dada a natureza dos recursos de processamento. Tradicionalmente, cada tipo de recurso tem seu poder explorado por meio de técnicas e tecnologias específicas de cada um. A capacidade de processamento paralelo oferecida por processadores *multicore* pode ser explorada pelo uso de *multithreading*, habilitado por tecnologias como POSIX Threads, Cilk, Anahy, OpenMP, entre outros. Por sua vez, o poder de processamento oferecido pelas GPUs vem sendo explorado através de *toolkits* específicos de fabricante, como NVIDIA CUDA e ATI Streaming SDK. No contexto do processamento gráfico e renderização 3D em tempo-real, emprega-se uma API gráfica como OpenGL [Shreiner and Group 2009] ou Direct3D, parte do DirectX SDK da Microsoft[1].

Assim, o desenvolvimento de soluções em plataformas computacionais heterogêneas apresenta-se com custo elevado para o desenvolvedor, que deve possuir domínio de diversos paradigmas e ferramentas para extrair o poder computacional oferecido por estas plataformas. Neste contexto, surge OpenCL (*Open Computing Language*), criada pela Apple [Apple Inc. 2009] e padronizada pelo Khronos Group [Khronos Group 2010a]. OpenCL é a primeira plataforma aberta e livre de *royalties* para computação de alto desempenho em sistemas heterogêneos compostos por CPUs, GPUs, e outros processadores paralelos. OpenCL oferece aos desenvolvedores um ambiente uniforme de programação

[1] `http://www.microsoft.com/games/en-US/aboutGFW/pages/directx.aspx`

paralela para escrever códigos portáveis para estes sistemas heterogêneos, a fim de tirarem proveito do poder de processamento dos dispositivos gráficos e processadores e usá-los para diversas aplicações em múltiplas plataformas computacionais. O objetivo do padrão OpenCL é unificar em um único paradigma e conjunto de ferramentas o desenvolvimento de soluções de computação paralela para dispositivos de naturezas distintas. Para viabilizar o emprego de OpenCL no desenvolvimento de jogos é necessário que se promova a interoperação entre OpenCL e OpenGL. Assim, além dos conceitos de OpenCL, será mostrado de forma prática como informações processadas de forma paralela por duas diferentes APIs são compartilhadas, ou seja, como resultados de cálculos realizados pela OpenCL podem afetar o processamento gráfico realizado pela OpenGL.

Atualmente na sua versão 1.1 [Khronos Group 2010b], a especificação OpenCL é realizada em três partes: uma linguagem, uma camada de plataforma e um *runtime*. A especificação da linguagem descreve a sintaxe e a API para escrita de código em OpenCL, que executa nos aceleradores suportados: CPUs *multicore*, GPUs *many-core* e processadores OpenCL dedicados. A linguagem é baseada na especificação C99 da linguagem C [ISO 2005]. A camada de plataforma fornece ao desenvolvedor acesso às rotinas que buscam o número e os tipos de dispositivos no sistema. Assim, o desenvolvedor pode escolher e inicializar os dispositivos adequados para o processamento. O *runtime* permite ao desenvolvedor enfileirar comandos para execução nos dispositivos, sendo também é responsável por gerenciar os recursos de memória e computação disponíveis.

Na prática, aplicações OpenCL são estruturadas em uma série de camadas, como mostra a Figura 1. *Kernels* correspondem às entidades que são escritas pelo desenvolvedor na linguagem OpenCL C. A aplicação faz uso da API C do padrão para comunicar-se com a camada de plataforma (*Platform Layer*), enviando comandos ao *runtime*, que gerencia diversos aspectos da execução.

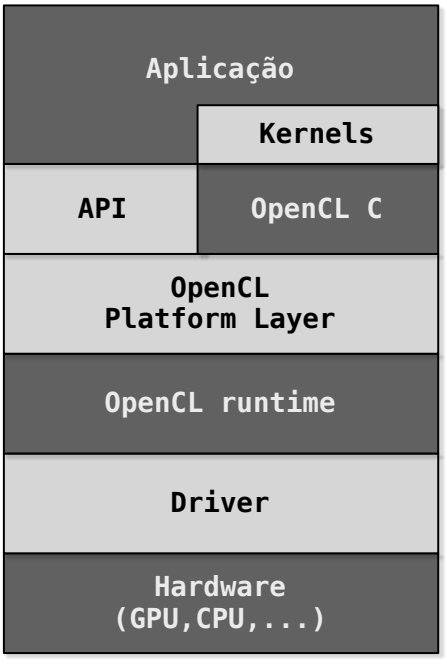


**Figura 1. Arquitetura em camadas de uma aplicação OpenCL.**

### 1.1. Hello World OpenCL

A fim de evitar que o leitor tenha que ser introduzido a muitos conceitos teóricos antes de ter uma ideia prática de como ocorre a programação em OpenCL, será apresentado aqui um exemplo simples de código. O trecho de código a seguir corresponde a um *kernel* que calcula a diferença entre os elementos de dois *arrays* e os armazena em um terceiro:

```
__kernel void ArrayDiff(
    __global const int* a,
    __global const int* b,
    __global int* c)
{
    int id = get_global_id(0);
    c[id] = a[id] - b[id];
}
```

Por ora, os detalhes a respeito de palavras-chave da linguagem, como `__kernel` e `__global` serão deixados de lado, pois serão introduzidos mais adiante. O que é importante observar no código é que não há um laço para iterar sobre os *arrays*. Em OpenCL, o código escrito geralmente foca-se na computação de uma unidade do resultado desejado. O *runtime* fica responsável por criar tantas instâncias do *kernel* quantas forem necessárias para o processamento de todo o conjunto de dados, no momento da execução. Cada instância recebe um conjunto de identificadores que permitem determinar a porção dos dados pela qual é responsável, habilitando o processamento paralelo dos dados.

A título de comparação, o código na sequência realiza a mesma tarefa, porém de forma sequencial:

```
void ArrayDiff(const int* a, const int* b, int* c, int n)
{
    for (int i = 0; i < n; ++i)
    {
        c[i] = a[i] - b[i];
    }
}
```

É interessante observar que não foi necessário informar ao *kernel* OpenCL o tamanho do conjunto de dados, uma vez que a manipulação desta informação é responsabilidade do *runtime*.

O uso de OpenCL é, portanto, recomendado para aplicações que realizam operações computacionalmente custosas, porém paralelizáveis por permitirem o cálculo independente de diferentes porções do resultado.

### 1.2. Organização do texto

O restante do texto está organizado como segue. A Seção 2 introduz os conceitos da arquitetura OpenCL. As Seções 3 e 4 discutem alguns aspectos da linguagem OpenCL C e da sua API, respectivamente. A Seção 5 ilustra a utilização de OpenCL por meio de exemplos práticos. Finalmente, a Seção 6 apresenta algumas considerações finais.

## 2. Arquitetura OpenCL

O padrão OpenCL propõe uma arquitetura caracterizada por uma abstração de baixo nível do *hardware*. Dispositivos que suportam o padrão devem obedecer a semântica descrita para esta arquitetura, mapeando suas características físicas a esta abstração.

A arquitetura OpenCL é descrita por quatro modelos, bem como por uma série de conceitos associados a estes. Cada modelo descreve um aspecto da arquitetura e as relações entre os seus conceitos. Estes modelos devem ser conhecidos pelo desenvolvedor para uma exploração efetiva das facilidades oferecidas pelo padrão, bem como dos recursos oferecidos por um determinado dispositivo que o implementa.

## 2.1. Modelo de plataforma

O modelo de plataforma descreve as entidades presentes em um ambiente computacional OpenCL. Um ambiente computacional OpenCL é integrado por um **hospedeiro** (*host*), que agrega um ou mais **dispositivos** (*devices*). Cada dispositivo possui uma ou mais **unidades de computação** (*compute units*), sendo estas compostas de um ou mais **elementos de processamento** (*processing elements*). O hospedeiro é responsável pela descoberta e inicialização dos dispositivos, bem como pela transferência de dados e tarefas para execução nestes. A Figura 2 apresenta um diagrama do modelo de plataforma.

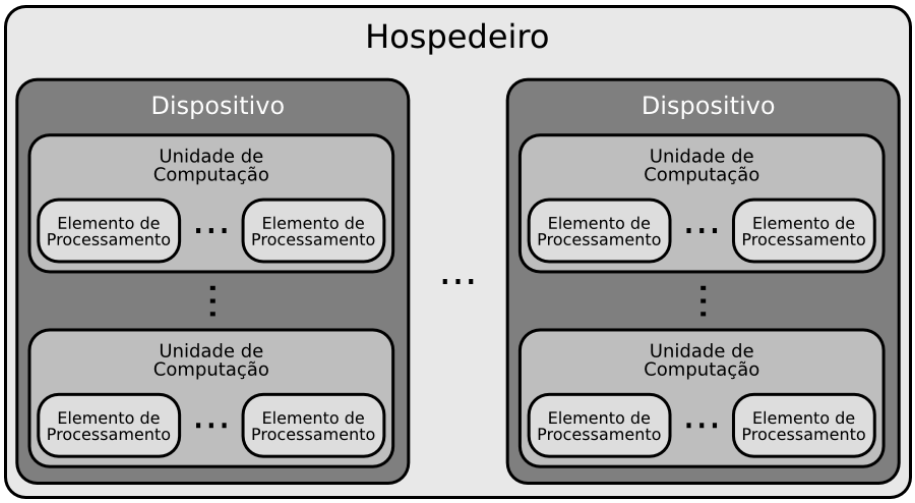


**Figura 2. Diagrama do modelo de plataforma OpenCL.**

## 2.2. Modelo de execução

O modelo de execução descreve a instanciação de *kernels* e a identificação das instâncias. Em OpenCL, um *kernel* é executado em um espaço de índices de 1, 2 ou 3 dimensões, denominado **NDRange** (*N-Dimensional Range*). Cada instância do *kernel* é denominada **item de trabalho** (*work-item*), sendo este identificado por uma tupla de índices, havendo um índice para cada dimensão do espaço de índices. Estes índices são os **identificadores globais** do item de trabalho.

Itens de trabalho são organizados em **grupos de trabalho** (*work-groups*). Cada grupo de trabalho também é identificado por uma tupla de índices, com um índice para cada dimensão do espaço. Dentro de um grupo de trabalho, um item de trabalho recebe ainda outra tupla de índices, os quais constituem os seus **identificadores locais** no grupo de trabalho.

A Figura 3 ilustra um NDRange de duas dimensões, dividido em quatro grupos de trabalho. Cada grupo de trabalho contém quatro itens de trabalho. Observando-se os diferentes índices existentes, pode-se constatar que um item de trabalho pode ser identificado individualmente de duas maneiras:

1. Por meio de seus identificadores globais.
2. Por meio da combinação de seus identificadores locais e dos identificadores do seu grupo de trabalho.

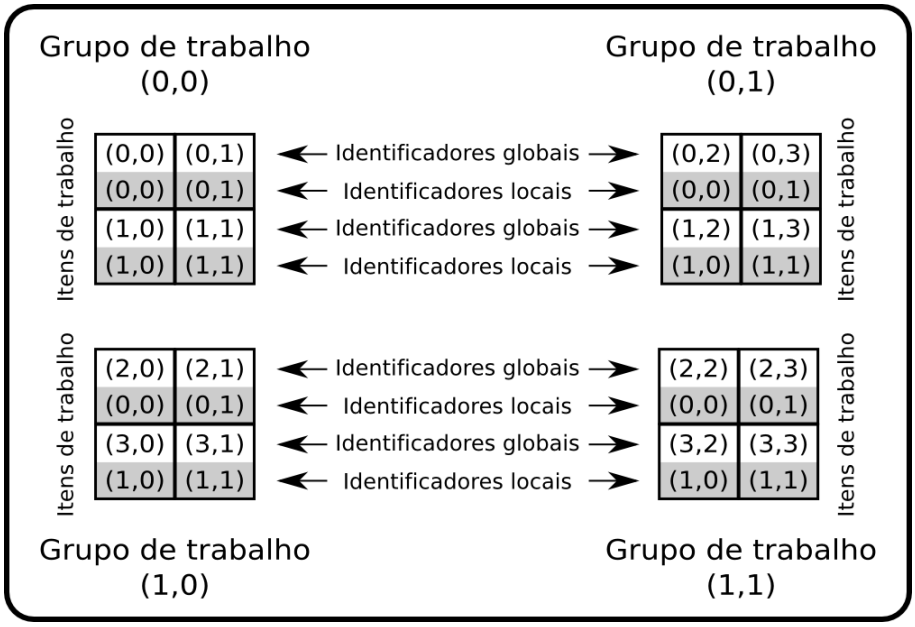


**Figura 3. Exemplo de espaço de índices de duas dimensões.**

Os identificadores de um item de trabalho são, em geral, empregados para indexar estruturas que armazenam os dados de entrada e saída do *kernel*. O espaço de índices é frequentemente dimensionado de em função do tamanho dos conjuntos de dados a serem processados. Assim, por meio de seus identificadores, cada item de trabalho pode ser designado responsável por um ponto ou uma parte específica do resultado.

A execução de *kernels* em uma aplicação OpenCL só é possível após a definição de um **contexto** (*context*). Um contexto engloba um conjunto de dispositivos e *kernels*, além de outras estruturas necessárias para a operação da aplicação, como filas de comandos, objetos de programa e objetos de memória.

Para serem executados, *kernels* são submetidos a **filas de comandos** (*command queues*) associadas aos dispositivos em uso. Cada dispositivo possui uma fila de comandos associada a si. Além de comandos para a execução de *kernels*, existem ainda comandos para a leitura e escrita de dados na memória dos dispositivos e comandos de sincronização, que suspendem a execução no hospedeiro até que um determinado conjunto de comandos, ou todos os comandos, da fila terminem de executar. Os comandos em uma fila de execução são sempre iniciados na ordem em que foram enviados, porém não há garantias quanto à ordenação do seu término. Em vista disso, por vezes é necessário que o hospedeiro realize algum tipo de sincronização, a fim de garantir a consistência da informação e da execução da aplicação.

Comandos de leitura e escrita na memória de um dispositivo empregam **objetos de memória** (*memory objects*) para realizar esta comunicação. Objetos de memória possuem associados a si um tamanho, além de um conjunto de parâmetros que definem se a região de memória associada ao objeto é, por exemplo, somente leitura, ou se encontra

mapeada em uma região de memória do hospedeiro. Existem duas categorias de objetos de memória: *buffers* e imagens. *Buffers* são equivalentes a *arrays* comumente encontrados nas linguagens de programação, sendo seus elementos acessíveis por índices ou ponteiros. Imagens são objetos especiais, em geral alocados em memória dedicada ao armazenamento de texturas, e seus elementos são acessados por meio de objetos especiais denominados *samplers*. Imagens não serão tratadas neste tutorial, dada a complexidade significativa da sua manipulação.

*Kernels* são gerados a partir de **objetos de programa** (*program objects*). Um objeto de programa encapsula o código-fonte de um ou mais *kernels*, sendo estes identificados no código-fonte por meio da palavra-chave `__kernel`. Um objeto de programa também encapsula uma ou mais representações binárias do código, de acordo com o número de dispositivos presentes no contexto em que o objeto de programa se encontra.

### 2.3. Modelo de programação

O modelo de programação descreve as abordagens possíveis para a escrita e execução de código OpenCL. *Kernels* podem ser executados de dois modos distintos:

- **Paralelismo de dados** (*Data parallel*): são instanciados múltiplos itens de trabalho para a execução do *kernel*. Este é o modo descrito até o momento, e consiste no modo principal de uso de OpenCL.
- **Paralelismo de tarefas** (*task parallel*): um único item de trabalho é instanciado para a execução do *kernel*. Isto permite a execução de múltiplos *kernels* diferentes sobre um mesmo conjunto de dados, ou sobre conjuntos de dados distintos.

### 2.4. Modelo de memória

O modelo de memória descreve os níveis de acesso à memória dos dispositivos e o seu compartilhamento entre os itens e grupos de trabalho. A memória de um dispositivo OpenCL é dividida em quatro categorias:

- **Memória global** (*global memory*): compartilhada por todos os itens de trabalho, sendo permitido o acesso de escrita e leitura;
- **Memória local** (*local memory*): compartilhada apenas entre os itens de trabalho de um mesmo grupo de trabalho, sendo permitido o acesso para escrita e leitura;
- **Memória privada** (*private memory*): restrita a cada item de trabalho, para escrita e leitura;
- **Memória constante** (*constant memory*): compartilhada por todos os itens de trabalho, porém o acesso é somente-leitura.

A Figura 4 ilustra esta categorização da memória, bem como a associação entre as categorias e as entidades do modelo de plataforma.

#### 2.4.1. Consistência de memória

O estado visível da memória a um item de trabalho pode não ser o mesmo para outros itens de trabalho durante a execução de um *kernel*. Porém, o padrão OpenCL ordena que as seguintes garantias sejam feitas quanto à consistência do acesso à memória:

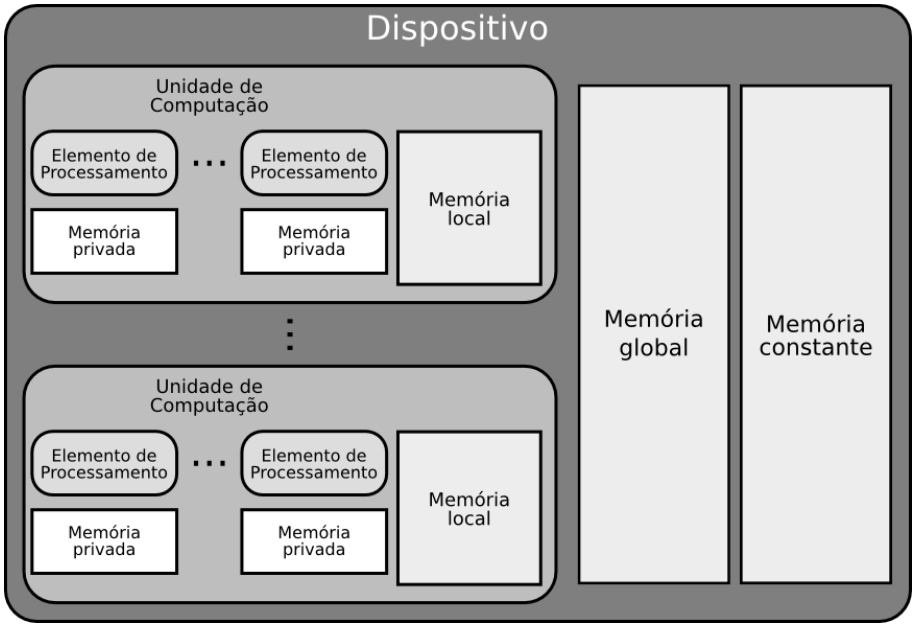


**Figura 4. Diagrama do modelo de memória OpenCL.**

1. Para um único item de trabalho, há consistência de leitura e escrita. Se um item de trabalho escrever em uma região de memória e, a seguir, ler a mesma região, o valor lido será aquele que foi escrito.
2. As memórias global e local são consistentes entre itens de trabalho de um mesmo grupo de trabalho em uma barreira (ver Seção 4).

## 3. Linguagem OpenCL C

A linguagem utilizada para a escrita *kernels* OpenCL, chamada OpenCL C, é baseada na especificação C99 da linguagem C, com algumas extensões e um conjunto de restrições. As seções a seguir detalham os aspectos mais relevantes da linguagem OpenCL C.

### 3.1. Tipos de dados

A linguagem OpenCL C oferece tipos de dados divididos em duas categorias: tipos escalares e tipos vetoriais. *Tipos escalares* permitem o armazenamento de informações numéricas com diferentes níveis de precisão e em diversos intervalos. *Tipos vetoriais* permitem o armazenamento de múltiplos dados escalares em uma única estrutura. Também tipos definidos através de `struct` ou `union`, além de *arrays*, funções e ponteiros, são suportados.

A Tabela 1 apresenta os principais tipos escalares da linguagem OpenCL C. Tipos vetoriais são nomeados de acordo com os tipos escalares, sendo sua forma geral `tipo`*`n`*. `tipo` corresponde a um dos tipos listados na Tabela 1, enquanto *`n`* especifica o número de componentes do vetor, o qual pode ser 2, 4, 8 ou 16. Por exemplo, um vetor com 4 componentes do tipo `int` possui tipo `int4`. Tipos vetoriais com componentes sem sinal sempre são declarados seguindo a forma abreviada do tipo escalar, isto é, com o prefixo `u` no início do nome do tipo, ao invés do modificador `unsigned`. Não são suportados vetores de `bool`, `half`, `size_t` e `void`, nem de tipos definidos pelo usuário.

| Tipo | Descrição |
|---|---|
| `bool` | Booleano; assume os valores `true` ou `false`. |
| `uchar` / `char` | Inteiro de 8 bits sem/com sinal. |
| `ushort` / `short` | Inteiro de 16 bits sem/com sinal. |
| `uint` / `int` | Inteiro de 32 bits sem/com sinal. |
| `ulong` / `long` | Inteiro de 64 bits sem/com sinal. |
| `float` | Ponto-flutuante de 32 bits. |
| `half` | Ponto-flutuante de 16 bits. |
| `size_t` | Equivalente a `uint` ou `ulong`, dependendo da implementação. |
| `void` | Tipo vazio. |

**Tabela 1. Tipos de dados escalares da linguagem OpenCL C.**

As componentes de uma variável de tipo vetorial são acessadas por meio do operador `.` (ponto). As componentes são identificadas pelos caracteres `x`, `y`, `z` e `w`, no caso de vetores de até 4 componentes, ou por índices numéricos de 0 a F (notação hexadecimal), prefixados pelo caractere `s`. Mais de uma componente pode ser informada em um único acesso, em qualquer ordem, ou mesmo de forma repetida, para formar outro vetor, contanto que sejam respeitadas as quantidades de componentes permitidas. Os exemplos a seguir ilustram alguns acessos a vetores:

```
float4 vec1;
float4 vec2;
float16 vec3;

// Acesso à componente x de vec1
vec1.x = 1.0f;

// Acesso simultâneo às componentes x e z de vec2
vec2.xz = (float2)(2.0f, 3.0f);

// Acesso com componentes permutadas
vec1.zw = vec2.yx;

// Acesso com componentes repetidas
vec2.yz = vec1.ww;

// Acesso às componentes 0, 5, 10 e 15 de vec3
vec3.s05af = vec1;
```

### 3.2. Qualificadores

A linguagem OpenCL C oferece um único qualificador de função, `__kernel`, cujo propósito é indicar que a função qualificada é um *kernel*. Dentro de um código OpenCL, uma chamada a uma função definida com o qualificador `__kernel` comporta-se de forma similar a qualquer outra chamada. Deve-se atentar, no entanto, a chamadas a funções `__kernel` que tenham no seu código variáveis declaradas com o qualificador `__local` (ver texto a seguir), pois o comportamento é definido pela implementação.

São definidos quatro qualificadores de espaço de endereçamento:

- `__global` ou `global`
- `__local` ou `local`
- `__constant` ou `constant`
- `__private` ou `private`

Estes qualificadores indicam o tipo de região de memória onde uma variável ou *buffer* está alocado, de acordo com o modelo de memória descrito na Seção 2.4. As seguintes regras se aplicam ao uso de qualificadores de espaço de endereçamento:

1. Variáveis declaradas sem um qualificador de espaço de endereçamento são automaticamente tratadas como `__private`.
2. Em um *kernel*, argumentos declarados como ponteiros devem ser `__global`, `__local` ou `__constant`. O uso de um destes qualificadores é obrigatório para tais argumentos.
3. Um ponteiro de um espaço de endereçamento só pode ser atribuído a outro ponteiro do mesmo espaço de endereçamento. Por exemplo, tentar atribuir o valor de um ponteiro `__local` a outro declarado com o qualificador `__global` constitui uma operação ilegal.
4. Variáveis cujo escopo seja o programa inteiro devem ser declaradas com o qualificador `__constant`.

A seguir estão ilustradas algumas declarações de variáveis com os qualificadores de espaço de endereçamento apresentados:

```
// Array de inteiros alocado na memória global
__global int* arr;

// Vetor com 4 componentes ponto-flutuante alocado
// na memória local
__local float4 vec;

// Inteiro de 64 bits alocado na memória privada
long n;
```

### 3.3. Operadores

A linguagem OpenCL C oferece um conjunto de operadores que trabalham tanto com tipos escalares quanto vetoriais, permitindo a construção de expressões aritméticas, lógicas e relacionais com tais tipos. A Tabela 2 apresenta os principais operadores da linguagem, divididos de acordo com a sua categoria.

A seguir são listadas algumas observações importantes sobre o uso dos operadores da linguagem OpenCL C:

1. Uma operação entre um vetor e um escalar converte o escalar para o tipo do vetor[2] e aplica a operação sobre cada componente do vetor. Por exemplo, uma soma entre um vetor e um escalar resultará em um vetor cujas componentes foram acrescidas da quantidade escalar. Da mesma forma, os operadores unários retornam resultados vetoriais onde a operação foi aplicada individualmente sobre cada componente do vetor.

[2]O escalar deve possuir o mesmo número de bits ou menos que o tipo das componentes do vetor; caso contrário, a tentativa de conversão para um tipo menor resultará em um erro de compilação.

| Categoria | Operadores |
|---|---|
| Operadores aritméticos binários | +, −, ⋆, /, % |
| Operadores aritméticos unários | +, −, ++, − |
| Operadores relacionais | >, >=, <, <=, ==, != |
| Operadores lógicos binários | &&, \|\| |
| Operadores lógicos unários | ! |
| Operadores bit-a-bit | &, \|, ^, , <<, >> |
| Operadores de atribuição | =, +=, −=, ⋆=, /=, %=, &=, \|=, ^=, <<=, >>= |

**Tabela 2. Principais operadores da linguagem OpenCL C.**

2. Operações entre vetores requerem que estes sejam do mesmo tamanho, isto é, contenham o mesmo número de componentes. Caso contrário, ocorrerá um erro de compilação. Tais operações são aplicadas componente-a-componente sobre o vetor, por isso a necessidade de estes possuírem o mesmo número de componentes.
3. Operações lógicas e relacionais com pelo menos um operando vetorial retornam um vetor do mesmo tipo deste, porém sempre com sinal. Cada componente pode conter o valor 0 para um resultado falso e -1 (todos os bits configurados em 1) para um resultado verdadeiro.
4. Os operadores aritméticos `%`, `++`, `−−`, operadores lógicos e operadores bit-a-bit não suportam tipos ponto-flutuante.

### 3.4. Restrições

A linguagem OpenCL C estende a linguagem da especificação C99 com alguns tipos novos e outros elementos, mas também acrescenta uma série de restrições. A lista a seguir enumera algumas das principais destas restrições:

1. Ponteiros para função não são permitidos.
2. Argumentos para *kernels* não podem ser ponteiros para ponteiros.
3. Funções e macros com número variável de argumentos não são suportados.
4. Os qualificadores `extern`, `static` e `auto` não são suportados.
5. Não há suporte a recursão.
6. A escrita em ponteiros ou *arrays* com tipos de tamanho inferior a 32 bits (inclusive os os tipos vetoriais `char2` e `uchar2`) não é permitida[3].
7. Os elementos de uma `struct` ou `union` devem pertencer ao mesmo espaço de endereçamento, de acordo com os qualificadores discutidos na Seção 3.1.

## 4. API de suporte

Neste seção é descrita a interface de programação OpenCL. Inicialmente, são apresentados os serviços associados ao desenvolvimento de funções *kernel*. Na sequência, serviços para realizar a interação do hospedeiro com o *runtime* OpenCL.

Está fora do escopo deste material realizar uma listagem exaustiva de toda a API OpenCL. Caso esteja interessado em ter acesso a todas as funções disponíveis, o leitor deve referir-se ao documento da especificação OpenCL 1.1 [Khronos Group 2010b].

[3]Esta restrição se aplica somente à versão 1.0 do padrão OpenCL.

## 4.1. API para *kernels*

Além da linguagem OpenCL C, o padrão OpenCL define uma API para uso em conjunto com esta linguagem, provendo diversas funções que podem ser usadas pelos *kernels* durante a computação das suas tarefas. Estas funções permitem a a identificação de itens e grupos de trabalho, a realização de operações matemáticas diversas (livrando o programador da tarefa de implementá-las), sincronização, e ainda outras funcionalidades.

Aqui serão apresentadas apenas as funções de identificação e sincronização, uma vez que estas constituem as funcionalidades cruciais para o desenvolvimento de *kernels* para quaisquer objetivos.

### 4.1.1. Identificação

Funções de identificação permitem que cada item de trabalho adquira conhecimento sobre a sua localização no espaço de índices sobre o qual o *kernel* é executado.

**get_global_id**

```
size_t get_global_id(uint dimindx)
```

Retorna o índice global do item de trabalho no espaço de índices, na dimensão identificada por `dimindx`.

**get_local_id**

```
size_t get_local_id(uint dimindx)
```

Retorna o índice local do item de trabalho no seu grupo de trabalho, na dimensão identificada por `dimindx`.

**get_group_id**

```
size_t get_group_id(uint dimindx)
```

Retorna o índice do grupo de trabalho ao qual o item de trabalho pertence, na dimensão identificada por `dimindx`.

**get_global_size**

```
size_t get_global_size(uint dimindx)
```

Retorna o número total de itens de trabalho na dimensão identificada por `dimindx`.

**get_local_size**

```
size_t get_local_size(uint dimindx)
```

Retorna o número de itens de trabalho em um grupo de trabalho na dimensão identificada por `dimindx`.

**get_num_groups**

```
size_t get_num_groups(uint dimindx)
```

Retorna o número de grupos de trabalho na dimensão identificada por `dimindx`.

**get_work_dim**

```
uint get_work_dim()
```

Retorna o número de dimensões do espaço de índices sobre o qual o *kernel* está sendo executado.

### 4.1.2. Sincronização

A API de sincronização permite a especificação de pontos de sincronização que garantem a consistência de memória, devendo ser atingidos por todos os itens de trabalho em um grupo de trabalho antes que a execução prossiga.

**barrier**

```
void barrier(cl_mem_fence_flags flags)
```

Cria uma barreira de sincronização. A execução só continuará depos que todos os itens de trabalho pertencentes a um mesmo grupo de trabalho alcancem a barreira. Uma situação de *deadlock* pode ocorrer quando, por erro de programação, um ou mais itens de trabalho de um mesmo grupo de trabalho falham em atingir a barreira, pois a execução dos itens de trabalho que a atingiram jamais prosseguira.

Uma barreira permite a criação de uma *memory fence*, que garante que todas as escritas na memória anteriores à barreira terão sido realmente efetuadas. O parâmetro `flags` permite controlar a *memory fence*. Seu valor pode ser `CLK_LOCAL_MEM_FENCE`, garantindo a consistência da memória local, `CLK_GLOBAL_MEM_FENCE`, garantindo a consistência da memória global, ou a combinação de ambos os valores através do operador `|` (*or* bit-a-bit).

## 4.2. API de interface com o *runtime*

Além da API disponível para o desenvolvimento de *kernels*, o padrão OpenCL define uma interface com o *runtime* para o hospedeiro. A interface de programação OpenCL consiste em uma API para a linguagem C, formada por um conjunto de funções, tipos de dados e constantes. Neste seção, são abordados os elementos mais comuns desta API, habilitando o leitor para o desenvolvimento de uma ampla gama de soluções em OpenCL. Esta fora do escopo deste tutorial realizar uma revisão exaustiva de toda a API OpenCL. Para uma descrição completa, o leitor deve consultar o documento de especificação do padrão.

As seções a seguir apresentam em detalhes algumas funções da API OpenCL para o hospedeiro. As funções estão agrupadas de acordo com os objetos OpenCL aos quais se referem. A Seção 5.1 expõe um exemplo completo de código de aplicação OpenCL, fazendo uso das funções aqui apresentadas.

### 4.2.1. Identificação da plataforma

**clGetPlatformIDs**

```
cl_int clGetPlatformIDs(cl_uint num_entries,
                        cl_platform_id* platforms,
                        cl_uint* num_platforms)
```

Em uma aplicação OpenCL, o primeiro passo na execução consiste em identificar a plataforma sobre a qual a esta será sendo executada. A função `clGetPlatformIDs` é utilizada para a descoberta de plataformas OpenCL no hospedeiro.

**Argumentos**:

1. `num_entries`: número de plataformas desejadas.

2. `platforms`: local onde os identificadores das plataformas encontradas devem ser escritos.
3. `num_platforms`: número de plataformas encontradas.

**Retorno**:

Código de erro. `CL_SUCCESS` caso a operação tenha sido bem sucedida. Ver o documento de especificação do padrão para demais códigos de erro.

### 4.2.2. Descoberta de dispositivos

#### clGetDeviceIDs

```
cl_int clGetDeviceIDs(cl_platform_id platform,
                      cl_device_type device_type,
                      cl_uint num_entries,
                      cl_device_id* devices,
                      cl_uint* num_devices)
```

A função `clGetDeviceIDs` possibilita a obtenção de um ou mais identificadores para dispositivos do hospedeiro que suportem o padrão OpenCL.

**Argumentos**:

1. `platform`: identificador de plataforma.
2. `device_type`: tipo de dispositivo. Os seguintes valores são válidos:
   - `CL_DEVICE_TYPE_CPU`: processador do hospedeiro.
   - `CL_DEVICE_TYPE_GPU`: dispositivo gráfico.
   - `CL_DEVICE_TYPE_ACCELERATOR`: processador OpenCL dedicado.
   - `CL_DEVICE_TYPE_DEFAULT`: dispositivo OpenCL padrão do sistema.
   - `CL_DEVICE_TYPE_ALL`: todos os dispositivos OpenCL do sistema.
3. `num_entries`: número de dispositivos desejados.
4. `devices`: *array* onde serão armazenados os identificadores dos dispositivos encontrados.
5. `num_devices`: número de dispositivos encontrados que correspondem aos tipos indicados em `device_type`. Caso seja `NULL`, o argumento será ignorado.

**Retorno**:

Código de erro. `CL_SUCCESS` caso a operação tenha sido bem sucedida. Ver o documento de especificação do padrão para demais códigos de erro.

### 4.2.3. Criação de contextos

#### clCreateContext

```
cl_context clCreateContext(
    const cl_context_properties* properties,
    cl_uint num_devices,
    const cl_device_id* devices,
    void (CL_CALLBACK *pfn_notify)(const char *errinfo,
                                   const void *private_info,
```

```
                                        size_t cb,
                                        void *user_data),
    void* user_data,
    cl_int* errcode_ret)
```

Para tornar possível o acesso a dispositivos OpenCL, é necessário que o hospedeiro primeiro inicialize e configure um contexto. A função `clCreateContext` cria um contexto OpenCL.

**Argumentos**:

1. `properties`: lista de propriedades para o contexto. A Seção 5.2 apresenta um exemplo de código de interoperação entre OpenCL e OpenGL, onde algumas destas propriedades são configuradas. Ver especificação para mais detalhes.
2. `num_devices`: número de dispositivos para o contexto.
3. `devices`: lista de identificadores de devices. Deve possuir tantos identificadores quantos indicados em `num_devices`.
4. `pfn_notify`: ponteiro para uma função de notificação chamada se houver um erro no contexto. Consultar a especificação OpenCL para mais detalhes a respeito desta função de notificação.
5. `user_data`: ponteiro para dados arbitrários passados para a função de notificações.
6. `errcode_ret`: local para armazentamento do código de erro da chamada. Pode ser `NULL`.

**Retorno**:

Contexto OpenCL. O contexto é válido apenas se o valor `CL_SUCCESS` estiver presente no local apontado por `errcode_ret` após a chamada.

### 4.2.4. Criação de filas de comandos

#### clCreateCommandQueue

```
cl_command_queue clCreateCommandQueue(
    cl_context context,
    cl_device_id device,
    cl_command_queue_properties properties,
    cl_int* errcode_ret)
```

A função `clCreateCommandQueue` cria uma fila de comandos para um dispositivo específico.

**Argumentos**:

1. `context`: contexto OpenCL.
2. `device`: identificador do dispositivo que será associado à fila.
3. `properties`: propriedades da fila de comandos. Ver especificação para mais detalhes.
4. `errcode_ret`: local para armazentamento do código de erro da chamada. Pode ser `NULL`.

**Retorno**:

Fila de comandos. O objeto retornado é válido apenas se o valor `CL_SUCCESS` estiver presente no local apontado por `errcode_ret` após a chamada.

### 4.2.5. Compilação de *kernels*

São necessário três passos para a compilação de um ou mais *kernels* em uma aplicação OpenCL:

1. Criação de um objeto de programa.
2. Compilação do programa para um ou mais dispositivos.
3. Criação de um ou mais *kernels* a partir do programa compilado.

**clCreateProgramWithSource**

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

A função `clCreateProgramWithSource` cria um objeto de programa a partir do código-fonte de um *kernel*. O código-fonte deve estar armazenado em um *array* de *strings*.

**Argumentos**:

1. `context`: contexto OpenCL.
2. `count`: número de *strings* no *array*.
3. `strings`: *array* de *strings* do código-fonte.
4. `lengths`: *array* contendo os tamanhos das *strings* do *array*. Pode ser `NULL` caso as *strings* sejam terminadas em `\0`.
5. `errcode_ret`: local para armazentamento do código de erro da chamada. Pode ser `NULL`.

**Retorno**:

Objeto de programa. O objeto retornado é válido apenas se o valor `CL_SUCCESS` estiver presente no local apontado por `errcode_ret` após a chamada.

**clBuildProgram**

```
cl_int clBuildProgram(
    cl_program program,
    cl_uint num_devices,
    const cl_device_id* device_list,
    const char* options,
    void (CL_CALLBACK *pfn_notify)(cl_program program,
                                   void *user_data),
    void* user_data)
```

A função `clBuildProgram` compila o código-fonte de um objeto de programa para um ou mais dispositivos do contexto.

**Argumentos**:

1. `program`: objeto de programa.
2. `num_devices`: número de dispositivos para o qual o programa deve ser compilado. O valor 0 causa a compilação para todos os dispositivos presentes no contexto.

3. `device_list`: lista de dispositivos para os quais o programa deve ser compilado. NULL indica que o programa deve ser compilado para todos os dispositivos presentes no contexto.
4. `options`: *string* de opções para o compilador OpenCL. O padrão define um conjunto de opções que devem ser suportadas por todos os compiladores.
5. `pfn_notify`: ponteiro para função de notificação chamada após a compilação.
6. `user_data`: ponteiro para dados arbitrários passados para a função de notificação.

Algumas opções de compilação definidas pelo padrão OpenCL são detalhadas a seguir:

- `-D`*símbolo*, `-D`*símbolo*`=`*valor*: opção para o pré-processador. Define um símbolo, opcionalmente com um valor correspondente.
- `-cl-opt-disable`: desabilita otimizações.
- `-cl-mad-enable`: utiliza uma única instrução para operações do tipo $a * b + c$, com perda de precisão.

O leitor deve consultar o padrão OpenCL para uma listagem de todas as opções definidas para o compilador. É interessante ressaltar que o compilador OpenCL é sempre fornecido por meio de um *toolkit* de fabricante, sendo chamado somente pelo *runtime*, nunca pelo próprio desenvolvedor.

**Retorno**:

Código de erro. `CL_SUCCESS` caso a operação tenha sido bem sucedida. Ver o documento de especificação do padrão para demais códigos de erro.

**clCreateKernel**

```
cl_kernel clCreateKernel(cl_program program,
                         const char* kernel_name,
                         cl_int* errcode_ret)
```

A função `clCreateKernel` cria um *kernel* a partir de um objeto de programa que passou pela etapa de compilação.

**Argumentos**:

1. `program`: objeto de programa.
2. `kernel_name`: nome de função `__kernel` definida no código-fonte.
3. `errcode_ret`: local para armazentamento do código de erro da chamada. Pode ser `NULL`.

**Retorno**:

*Kernel*. O objeto retornado é válido apenas se o valor `CL_SUCCESS` estiver presente no local apontado por `errcode_ret` após a chamada.

### 4.2.6. Manipulação de *buffers* OpenCL

A comunicação com a memória global de dispositivos OpenCL é realizada por meio de objetos de memória. Um dos tipos de objetos de memória é o *buffer*, que corresponde a uma região contígua de memória, com tamanho fixo.

**clCreateBuffer**

```
cl_mem clCreateBuffer(cl_context context,
                      cl_memflags flags,
                      size_t size,
                      void* host_ptr,
                      cl_int* errcode_ret)
```

A função `clCreateBuffer` permite a criação de *buffers* em um contexto. Estes *buffers* habilitam a transferência de dados de e para a memória global dos dispositivos do mesmo contexto.

**Argumentos**:

1. `context`: contexto OpenCL.
2. `flags`: *flags* de especificação de informações sobre a alocação e o uso da região de memória associada ao *buffer*. A *flag* `CL_MEM_READ_ONLY` cria um *buffer* somente-leitura. A *flag* `CL_MEM_READ_WRITE` especifica a criação de um *buffer* para operações de leitura e escrita de dados. Ver especificação para demais *flags* permitidas.
3. `size`: tamanho, em bytes, do *buffer* a ser alocado.
4. `host_ptr`: ponteiro para dados de inicialização do *buffer*.
5. `errcode_ret`: local para armazentamento do código de erro da chamada. Pode ser `NULL`.

**Retorno**:

Objeto de memória. O objeto retornado é válido apenas se o valor `CL_SUCCESS` estiver presente no local apontado por `errcode_ret` após a chamada.

**clEnqueueReadBuffer**

```
cl_int clEnqueueReadBuffer(cl_queue queue,
                           cl_mem buffer,
                           cl_bool blocking_read,
                           size_t offset,
                           size_t cb,
                           const void* ptr,
                           cl_uint events_in_wait_list,
                           const cl_event* event_wait_list,
                           cl_event* event)
```

A função `clEnqueueReadBuffer` insere em uma fila de comandos queue um comando para leitura de dados do dispositivo para a memória do hospedeiro. A leitura ocorre por meio de um objeto de memória.

**Argumentos**:

1. `queue`: fila de comandos.
2. `buffer`: objeto de memória do tipo *buffer*.
3. `blocking_read`: caso `CL_TRUE`, a chamada é bloqueante e o hospedeiro suspende a execução até que os dados tenham sido completamente transferidos do dispositivo. Caso `CL_FALSE`, a chamada é não-bloqueante e a execução prossegue assim que o comando é enfileirado.

4. `offset`: *offset* a partir do qual os dados devem ser transferidos.
5. `cb`: comprimento, em bytes, dos dados a serem transferidos.
6. `ptr`: ponteiro para a região de memória do host onde os dados transferidos devem ser escritos.
7. `events_in_wait_list`: número de eventos na lista de eventos que devem ser aguardados antes do início da transferência dos dados.
8. `event_wait_list`: lista de eventos que devem ser aguardados antes do início da transferência dos dados.
9. `event`: local para retorno do **objeto de evento** (*event object*) para o comando. Objetos de evento permitem o aguardo do término de comandos inseridos em uma fila de comandos. Este objeto é útil para a garantia da consistência dos dados quando a escrita é realizada de forma não-bloqueante.

**Retorno**:

Código de erro. `CL_SUCCESS` caso a operação tenha sido bem sucedida. Ver o documento de especificação do padrão para demais códigos de erro.

**clEnqueueWriteBuffer**

```
cl_int clEnqueueWriteBuffer(cl_queue queue,
                            cl_mem buffer,
                            cl_bool blocking_write,
                            size_t offset,
                            size_t cb,
                            void* ptr,
                            cl_uint events_in_wait_list,
                            const cl_event* event_wait_list,
                            cl_event* event)
```

A função `clEnqueueWriteBuffer` insere em uma fila de comandos queue um comando para escrita de dados do hospedeiro na memória do dispositivo associado à fila. A escrita ocorre por meio de um objeto de memória.

**Argumentos**:

1. `queue`: fila de comandos.
2. `buffer`: objeto de memória do tipo *buffer*.
3. `blocking_write`: caso `CL_TRUE`, a chamada é bloqueante e o hospedeiro suspende a execução até que os dados tenham sido completamente transferidos para o dispositivo. Caso `CL_FALSE`, a chamada é não-bloqueante e a execução prossegue assim que o comando é enfileirado.
4. `offset`: *offset* a partir do qual os dados devem ser transferidos.
5. `cb`: comprimento, em bytes, dos dados a serem transferidos.
6. `ptr`: ponteiro para a região de memória do host onde os dados a serem transferidos estão localizados.
7. `events_in_wait_list`: número de eventos na lista de eventos que devem ser aguardados antes do início da transferência dos dados.
8. `event_wait_list`: lista de eventos que devem ser aguardados antes do início da transferência dos dados.
9. `event`: local para retorno do objeto de evento para o comando.

**Retorno**:

Código de erro. `CL_SUCCESS` caso a operação tenha sido bem sucedida. Ver o documento de especificação do padrão para demais códigos de erro.

### 4.2.7. Execução de *kernels*

**clSetKernelArg**

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

Antes do envio do *kernel* para execução, é necessário que seu argumentos sejam configurados com os valores adequados. A configuração dos argumentos se dá pela informação das regiões de memória do hospedeiro onde seus valores se encontram. A função `clSetKernelArg` é utilizada para configurar os argumentos para um *kernel*.

**Argumentos:**

1. `kernel`: *kernel* cujo argumento deve ser configurado.
2. `arg_index`: posição do argumento, de acordo com a ordem em que os argumentos foram definidos no código-fonte, iniciando em 0.
3. `arg_size`: comprimento dos dados do argumento.
4. `arg_value`: ponteiro para dados do argumento.

**Retorno:**

Código de erro. `CL_SUCCESS` caso a operação tenha sido bem sucedida. Ver o documento de especificação do padrão para demais códigos de erro.

**clEnqueueNDRangeKernel**

```
cl_int clEnqueueNDRangeKernel(cl_command_queue command_queue,
                              cl_kernel kernel,
                              cl_uint work_dim,
                              const size_t* global_work_offset,
                              const size_t* global_work_size,
                              const size_t* local_work_size,
                              cl_uint events_in_wait_list,
                              const cl_event* event_wait_list,
                              cl_event* event)
```

Após a configuração dos argumentos, o *kernel* deve ser enviado para execução no dispositivo. É importante observar que a chamada que enfileira o comando de execução do *kernel* é sempre não-bloqueante. A sincronização, isto é, a espera pelo término da execução do *kernel*, deve ser realizada pelo hospedeiro de forma explícita. A função `clEnqueueNDRangeKernel` insere um comando para execução de um *kernel* em uma fila de comandos de um dispositivo. Seus argumentos permitem a configuração de fatores como o número de dimensões do espaço de índices, bem como os tamanhos destas dimensões.

**Argumentos:**

1. `command_queue`: fila de comandos.
2. `kernel`: *kernel* a ser executado.
3. `work_dim`: número de dimensões do espaço de índices. São permitidos os valores 1, 2 e 3.
4. `global_work_offset`: *array* de deslocamentos para valores dos índices em cada dimensão. Por exemplo, um deslocamento de 10 na dimensão 0 fará com que os índices naquela dimensão iniciem em 10 [4].
5. `global_work_size`: *array* contendo os tamanhos para cada dimensão do espaço de índices.
6. `local_work_size`: *array* de tamanhos dos grupos de trabalho em cada dimensão. Caso `NULL`, o *runtime* determina os tamanhos automaticamente. Caso os valores sejam informados explicitamente, é mandatório que a divisão dos tamanhos das dimensões do espaço de índices por estes valores seja inteira.
7. `events_in_wait_list`: número de eventos na lista de eventos que devem ser aguardados antes do início da execução do *kernel*
8. `event_wait_list`: lista de eventos que devem ser aguardados antes do início da execução do *kernel*.
9. `event`: local para retorno do objeto de evento para o comando.

**Retorno:**

Código de erro. `CL_SUCCESS` caso a operação tenha sido bem sucedida. Ver o documento de especificação do padrão para demais códigos de erro.

### 4.2.8. Sincronização

#### clFinish

```
cl_int clFinish(cl_command_queue command_queue)
```

A função `clFinish` bloqueia a execução no hospedeiro até que todos os comandos na fila informada tenham sido completados.

### 4.2.9. Liberação de recursos

Existe uma função de liberação para cada tipo de objeto OpenCL:

- Contextos:
  `clReleaseContext(cl_context context)`
- Filas de comandos:
  `clReleaseCommandQueue(cl_command_queue command_queue)`
- Objetos de programa:
  `clReleaseProgram(cl_program program)`
- *Kernels*:
  `clReleaseKernel(cl_kernel kernel)`
- Objetos de memória:
  `clReleaseMemObject(cl_mem buffer)`

[4] Apesar de presente na especificação OpenCL 1.0, este argumento tem efeito apenas em dispositivos com suporte ao padrão OpenCL a partir da versão 1.1.

### 4.2.10. Integração com OpenGL

O padrão OpenCL define algumas funções para interoperação com OpenGL, nos casos em que o dispositivo empregado para a computação consiste em uma GPU.

**clCreateFromGLBuffer**

```
cl_mem clCreateFromGLBuffer(cl_context context,
                            cl_mem_flags flags,
                            GLuint bufobj,
                            cl_int* errcode_ret)
```

A função `clCreateFromGLBuffer` cria um *buffer* OpenCL a partir de um *buffer* OpenGL (por exemplo, um *Vertex Buffer Object*) previamente alocado. Isto permite o compartilhamento de *buffers* entre OpenGL e OpenCL.

**Argumentos:**

1. `context`: contexto OpenCL.
2. `flags`: *flags* de criação do *buffer*. Ver Seção 4.2.6.
3. `bufobj`: identificador do *buffer* OpenGL, conforme retornado pela função `glGenBuffers`.
4. `errcode_ret`: local para armazentamento do código de erro da chamada. Pode ser `NULL`.

**Retorno**:

Objeto de memória. O objeto retornado é válido apenas se o valor `CL_SUCCESS` estiver presente no local apontado por `errcode_ret` após a chamada.

**clEnqueueAcquireGLObjects**

```
cl_int clEnqueueAcquireGLObjects(cl_command_queue command_queue,
                                 cl_uint num_objects.
                                 const cl_mem* mem_objects,
                                 cl_uint num_events_in_wait_list,
                                 const cl_event* event_wait_list,
                                 cl_event* event)
```

Antes de operar sobre *buffers* criados a partir de objetos OpenGL, é necessário que os mesmos sejam "adquiridos" (*acquired*) pelo contexto OpenCL. A função `clEnqueueAcquireGLObj` realiza a aquisição de um objeto OpenGL para o contexto associado com a fila de comandos passada como argumento para a função.

**Argumentos:**

1. `command_queue`: fila de comandos para um dispositivo.
2. `num_objects`: número de objetos a serem adquiridos.
3. `mem_objects`: *array* de objetos de memória associados aos objetos OpenGL a serem adquiridos. Deve conter `num_objects` objetos.
4. `events_in_wait_list`: número de eventos na lista de eventos que devem ser aguardados antes do início da execução do *kernel*
5. `event_wait_list`: lista de eventos que devem ser aguardados antes do início da execução do *kernel*.
6. `event`: local para retorno do objeto de evento para o comando.

**Retorno:**

Código de erro. `CL_SUCCESS` caso a operação tenha sido bem sucedida. Ver o documento de especificação do padrão para demais códigos de erro.

**clEnqueueReleaseGLObjects**

```
cl_int clEnqueueReleaseGLObjects(cl_command_queue command_queue,
                                 cl_uint num_objects.
                                 const cl_mem* mem_objects,
                                 cl_uint num_events_in_wait_list,
                                 const cl_event* event_wait_list,
                                 cl_event* event)
```

A função `clEnqueueReleaseGLObjects` libera objetos OpenGL adquiridos por um contexto OpenCL.

**Argumentos:**

1. `command_queue`: fila de comandos para um dispositivo.
2. `num_objects`: número de objetos a serem liberados.
3. `mem_objects`: *array* de objetos de memória associados aos objetos OpenGL a serem liberados. Deve conter `num_objects` objetos.
4. `events_in_wait_list`: número de eventos na lista de eventos que devem ser aguardados antes do início da execução do *kernel*
5. `event_wait_list`: lista de eventos que devem ser aguardados antes do início da execução do *kernel*.
6. `event`: local para retorno do objeto de evento para o comando.

**Retorno:**

Código de erro. `CL_SUCCESS` caso a operação tenha sido bem sucedida. Ver o documento de especificação do padrão para demais códigos de erro.

## 5. Exemplos

Nesta seção, são apresentados dois exemplos de utilização de OpenCL. O primeiro consiste em um código completo para uma aplicação simples, que pode ser transcrito pelo leitor para ter sua primeira experiência na programação OpenCL. No segundo exemplo, apresenta-se o código completo para uma aplicação OpenCL que realiza interoperação com OpenGL, manipulando os vértices de uma malha tridimensional.

### 5.1. Aplicação OpenCL simples

O trecho de código a seguir corresponde a uma aplicação OpenCL simples, porém completa. A aplicação utiliza uma GPU para resolver o problema da diferença entre os elementos de dois *arrays*. Os *arrays* de entrada são gerados aleatoriamente. Ao final da execução, os resultados são impressos na saída padrão. Ressalta-se que este é o código executado pelo hospedeiro, sendo o código do *kernel* para a solução do problema embutido no primeiro, na forma de uma *string*.

É de especial interesse observar que o mesmo *kernel* poderia ser executado em outro dispositivo OpenCL de qualquer natureza sem a necessidade de alterar o seu código. Para isto, seria necessário apenas que o hospedeiro requisitasse outro tipo de dispositivo.

Para fins de clareza e brevidade, a verificação de erros nas operações foi omitida. No entanto, alerta-se que é uma boa prática verificar os códigos de erro retornados por todas as chamadas.

```
#include <stdio.h>
#include <stdlib.h>

#ifdef __APPLE__
#include <OpenCL/opencl.h>
#else
#include <CL/opencl.h>
#endif

#define ARRAY_LENGTH 1000

int main(int argc, char** argv)
{
    /* Variáveis para armazenamento de referências a
       objetos OpenCL */
    cl_platform_id platformId;
    cl_device_id deviceId;
    cl_context context;
    cl_command_queue queue;
    cl_program program;
    cl_kernel kernel;
    cl_mem bufA;
    cl_mem bufB;
    cl_mem bufC;

    /* Variáveis diversas da aplicação */
    int* hostA;
    int* hostB;
    int* hostC;
    size_t globalSize[1] = { ARRAY_LENGTH };
    int i;

    /* Código-fonte do kernel */
    const char* source =
    "__kernel void ArrayDiff( \
        __global const int* a, \
        __global const int* b, \
        __global int* c) \
    { \
        int id = get_global_id(0); \
        c[id] = a[id] - b[id]; \
    }";

    /* Obtenção de identificadores de plataforma
       e dispositivo. Será solicitada uma GPU. */
    clGetPlatformIDs(1, &platformId, NULL);
    clGetDeviceIDs(platformId, CL_DEVICE_TYPE_GPU,
        1, &deviceId, NULL);

    /* Criação do contexto */
    context = clCreateContext(0, 1, &deviceId,
        NULL, NULL, NULL);
```

```
/* Criação da fila de comandos para o
   dispositivo encontrado */
queue = clCreateCommandQueue(context, deviceId,
    0, NULL);

/* Criação do objeto de programa a partir do
   código-fonte armazenado na string source */
program = clCreateProgramWithSource(context, 1, &source,
    NULL, NULL);

/* Compilação do programa para todos os
    dispositivos do contexto */
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

/* Obtenção de um kernel a partir do
    programa compilado */
kernel = clCreateKernel(program, "ArrayDiff", NULL);

/* Alocação e inicialização dos arrays no hospedeiro */
hostA = (int*) malloc(ARRAY_LENGTH * sizeof(int));
hostB = (int*) malloc(ARRAY_LENGTH * sizeof(int));
hostC = (int*) malloc(ARRAY_LENGTH * sizeof(int));

for (i = 0; i < ARRAY_LENGTH; ++i)
{
    hostA[i] = rand() % 101 - 50;
    hostB[i] = rand() % 101 - 50;
}

/* Criação dos objetos de memória para comunicação com
   a memória global do dispositivo encontrado */
bufA = clCreateBuffer(context, CL_MEM_READ_ONLY,
    ARRAY_LENGTH * sizeof(int), NULL, NULL);

bufB = clCreateBuffer(context, CL_MEM_READ_ONLY,
    ARRAY_LENGTH * sizeof(int), NULL, NULL);

bufC = clCreateBuffer(context, CL_MEM_READ_WRITE,
    ARRAY_LENGTH * sizeof(int), NULL, NULL);

/* Transferência dos arrays de entrada para a memória
   do dispositivo */
clEnqueueWriteBuffer(queue, bufA, CL_TRUE, 0,
    ARRAY_LENGTH * sizeof(int), hostA, 0,
    NULL, NULL);

clEnqueueWriteBuffer(queue, bufB, CL_TRUE,  0,
    ARRAY_LENGTH * sizeof(int), hostB, 0,
    NULL, NULL);

/* Configuração dos argumentos do kernel */
clSetKernelArg(kernel, 0, sizeof(cl_mem), &bufA);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &bufB);
clSetKernelArg(kernel, 2, sizeof(cl_mem), &bufC);
```

```
    /* Envio do kernel para execução no dispositivo */
    clEnqueueNDRangeKernel(queue, kernel, 1, NULL,
        globalSize, NULL, 0, NULL, NULL);

    /* Sincronização (bloqueia hospedeiro até término da
       execução do kernel */
    clFinish(queue);

    /* Transferência dos resultados da computação para a
       memória do hospedeiro */
    clEnqueueReadBuffer(queue, bufC, CL_TRUE, 0,
        ARRAY_LENGTH * sizeof(int), hostC, 0,
        NULL, NULL);

    /* Impressão dos resultados na saída padrão */
    for (i = 0; i < ARRAY_LENGTH; ++i)
        printf("%d - %d = %d\n", hostA[i], hostB[i], hostC[i]);

    /* Liberação de recursos e encerramento da aplicação */
    clReleaseMemObject(bufA);
    clReleaseMemObject(bufB);
    clReleaseMemObject(bufC);
    clReleaseKernel(kernel);
    clReleaseProgram(program);
    clReleaseCommandQueue(queue);
    clReleaseContext(context);

    free(hostA);
    free(hostB);
    free(hostC);

    return 0;
}
```

### 5.1.1. Compilação e execução

O uso efetivo de OpenCL varia de plataforma para plataforma, uma vez que cada fabricante provê suas próprias ferramentas e *toolkits*. Os elementos definidos no padrão são os mesmos, mas aspectos como a compilação e ligação de aplicações OpenCL variam de acordo com a plataforma em uso.

Sendo assim, não cabe a este tutorial realizar uma listagem exaustiva de todas as possibilidades para a compilação do exemplo apresentado, dada a existência de suporte para múltiplos sistemas operacionais, bem como múltiplos dispositivos.

A seguir é apresentados o comando para compilação do exemplo de código desta seção em um ambiente Linux, considerando o uso de uma placa gráfica NVIDIA com suporte ao padrão OpenCL. Antes da compilação, é necessário realizar a instalação dos *drivers* de desenvolvimento da NVIDIA, bem como do CUDA *toolkit*, que inclui os cabeçalhos e bibliotecas necessários para o uso de OpenCL [5]. Assume-se que o *toolkit* tenha sido instalado no ser local padrão.

[5]Em dispositivos NVIDIA, o código OpenCL é traduzido para instruções da arquitetura CUDA.

```
gcc -I/usr/local/cuda/include ArrayDiff.c -o ArrayDiff -lOpenCL
```

Este comando realiza a compilação e ligação da aplicação com o *runtime* OpenCL. Como pode ser observado, assume-se que o código tenha sido armazenado em um arquivo chamado `ArrayDiff.c`, porém este nome não é obrigatório, nem mesmo o nome dado ao executável (`ArrayDiff`).

Após a compilação, a execução se dá como a de qualquer outro executável:

```
./ArrayDiff
```

### 5.2. Interoperação com OpenGL

O exemplo de código a seguir ilustra a interoperação entre OpenCL e OpenGL. No exemplo, um *Vertex Buffer Object* (VBO) OpenGL é utilizado para armazenar as posições dos vértices de uma malha tridimensional. Este VBO é, então, compartilhado com o contexto OpenCL. O *kernel* OpenCL da aplicação é responsável por calcular as posições dos vértices da malha ao longo do tempo, animando-a. A biblioteca GLUT é utilizada para visualização da malha.

```
#ifdef __APPLE__
#include <OpenGL/OpenGL.h>
#include <GLUT/glut.h>
#include <OpenCL/opencl.h>
#else
#include <GL/gl.h>
#include <GL/glut.h>
#include <CL/opencl.h>
#ifndef _WIN32
#include <GL/glx.h>
#endif
#endif

/* Dimensões da janela e da malha */
const unsigned int windowWidth = 512;
const unsigned int windowHeight = 512;
const unsigned int meshWidth = 256;
const unsigned int meshHeight = 256;

/* Objetos OpenGL */
GLuint vbo;
int window = 0;

/* Objetos OpenCL */
cl_platform_id platformId;
cl_context context;
cl_device_id deviceId ;
cl_command_queue queue;
cl_kernel kernel;
cl_mem vboCL;
cl_program program;

/* Código-fonte do kernel */
const char* kernelSource = " \
__kernel void sine_wave( \n  \
   __global float4* pos, \n  \
```

```
   unsigned int width, \n    \
   unsigned int height, \n   \
   float time) \n            \
{ \n                                          \
   unsigned int x = get_global_id(0); \n      \
   unsigned int y = get_global_id(1); \n      \
\n                                            \
   float u = x / (float) width; \n            \
   float v = y / (float) height; \n           \
   u = u*2.0f - 1.0f; \n                      \
   v = v*2.0f - 1.0f; \n                      \
\n                                            \
   float freq = 4.0f; \n                      \
   float w =
      sin(u * freq + time) * cos(v * freq + time) * 0.5f; \n \
\n                                            \
   pos[y * width + x] = (float4)(u, w, v, 1.0f); \n         \
}";

/* Tempo de simulação (determina posição dos vértices
   na malha) */
float anim = 0.0;

void initGL(int argc, char** argv);
void initCL();
void displayGL(void);
void keyboard(unsigned char key, int x, int y);
void cleanup();

int main(int argc, char** argv)
{
   /* Inicialização GLUT */
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
   glutInitWindowPosition(
      glutGet(GLUT_SCREEN_WIDTH)/2 - windowWidth/2,
      glutGet(GLUT_SCREEN_HEIGHT)/2 - windowHeight/2);
   glutInitWindowSize(windowWidth, windowHeight);
   window = glutCreateWindow("OpenCL/GL Interop (VBO)");
   glutDisplayFunc(displayGL);
   glutKeyboardFunc(keyboard);

   /* Inicialização OpenGL e OpenCL*/
   initGL(argc, argv);
   initCL();

   glutMainLoop();

   cleanup();

   return 0;
}

void initGL(int argc, char** argv)
{
   glClearColor(0.0, 0.0, 0.0, 1.0);
```

```
   glDisable(GL_DEPTH_TEST);
   glViewport(0, 0, windowWidth, windowHeight);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(
      60.0, (GLfloat) windowWidth / (GLfloat) windowHeight,
      0.1, 10.0);
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   glTranslatef(0.0, 0.25, -3.0);
   glRotatef(45.0, 1.0, 0.0, 0.0);
   glRotatef(45.0, 0.0, 1.0, 0.0);

   glewInit();

   /* Criação do VBO */
   glGenBuffers(1, &vbo);
   glBindBuffer(GL_ARRAY_BUFFER, vbo);
   glBufferData(
      GL_ARRAY_BUFFER,
      meshWidth * meshHeight * 4 * sizeof(float),
      0, GL_DYNAMIC_DRAW);
}

void initCL()
{
   /* Obtenção de identificadores de plataforma e
      dispositivo (GPU) */
   clGetPlatformIDs(1, &platformId, NULL);
   clGetDeviceIDs(
      platformId, CL_DEVICE_TYPE_GPU, 1, &deviceId, NULL);

   /* Criação do contexto com propriedades para o
      compartilhamento com OpenGL */
#ifdef _WIN32
   cl_context_properties props[] =
      {
         CL_GL_CONTEXT_KHR,
         (cl_context_properties) wglGetCurrentContext(),
         CL_WGL_HDC_KHR,
         (cl_context_properties) wglGetCurrentDC(),
         CL_CONTEXT_PLATFORM,
         (cl_context_properties) platformId,
         0
          };
#else
   cl_context_properties props[] =
      {
         CL_GL_CONTEXT_KHR,
         (cl_context_properties)glXGetCurrentContext(),
         CL_GLX_DISPLAY_KHR,
         (cl_context_properties)glXGetCurrentDisplay(),
         CL_CONTEXT_PLATFORM,
         (cl_context_properties)platformId,
         0
```

```
      };
#endif
   context = clCreateContext(
      props, 1, &deviceId, NULL, NULL, NULL);

   /* Criação da fila de comandos do dispositivo */
   queue = clCreateCommandQueue(context, deviceId, 0, NULL);

   /* Criação de compilação do programa e do kernel */
   program = clCreateProgramWithSource(
      context, 1, (const char **) &kernelSource, NULL, NULL);
   clBuildProgram(
      program, 0, NULL, "-cl-fast-relaxed-math", NULL, NULL);
   kernel = clCreateKernel(program, "sine_wave", NULL);

   /* Criação do objeto de memória para o VBO OpenGL */
   vboCL = clCreateFromGLBuffer(
      context, CL_MEM_WRITE_ONLY, vbo, NULL);

   /* Configuração dos argumentos fixos do kernel */
   clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *) &vboCL);
   clSetKernelArg(kernel, 1, sizeof(unsigned int), &meshWidth);
   clSetKernelArg(kernel, 2, sizeof(unsigned int), &meshHeight);
}

void displayGL()
{
   /* Dimensões do espaço índices */
   const size_t globalSize[2] = { meshWidth, meshHeight };

   /* Incremento do tempo de simulação */
   anim += 0.001f;

   /* Aguarda término das operações OpenGL */
   glFinish();

   /* Aquisição do VBO para uso com OpenCL */
   clEnqueueAcquireGLObjects(queue, 1, &vboCL, 0,0,0);

   /* Atualização do tempo de simulação no kernel */
   clSetKernelArg(kernel, 3, sizeof(float), &anim);

   /* Execução do kernel */
   clEnqueueNDRangeKernel(
      queue, kernel, 2, NULL, globalSize, NULL, 0,0,0 );

   /* Liberação do VBO de volta para OpenGL */
   clEnqueueReleaseGLObjects(queue, 1, &vboCL, 0,0,0);

   /* Sincronização */
   clFinish(queue);

   /* Renderização da malha */
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glBindBuffer(GL_ARRAY_BUFFER, vbo);
   glVertexPointer(4, GL_FLOAT, 0, 0);
```

```
    glEnableClientState(GL_VERTEX_ARRAY);
    glColor3f(1.0, 0.0, 0.0);
    glDrawArrays(GL_POINTS, 0, meshWidth * meshHeight);
    glDisableClientState(GL_VERTEX_ARRAY);
    glutSwapBuffers();
    glutPostRedisplay();
}

void keyboard(unsigned char key, int x, int y)
{
    if (key == 'q' || key == 'Q')
            cleanup();
}

void cleanup()
{
    clReleaseKernel(kernel);
    clReleaseProgram(program);
    clReleaseCommandQueue(queue);

    glBindBuffer(1, vbo);
    glDeleteBuffers(1, &vbo);
    clReleaseMemObject(vboCL);

    clReleaseContext(context);
    glutDestroyWindow(window);

    exit(0);
}
```

Este exemplo ilustra uma vantagem do uso de OpenCL em aplicações gráficas. Cálculos que seriam, a princípio, realizados pela CPU, podem ser delegados para a GPU. Isto aumenta o paralelismo das operações e, consequentemente, o desempenho da aplicação. Evita-se também a necessidade de transferência de dados entre a CPU e o dispositivo, o que gera ainda mais ganhos de desempenho. O *kernel* OpenCL opera diretamente sobre a região de memória que será, posteriormente, utilizada pela GPU na leitura de dados para a renderização da malha.

### 5.2.1. Compilação e execução

A compilação deste segundo exemplo prático requer a ligação com bibliotecas adicionais, para uso dos serviços OpenGL. É necessário, também, que o ambiente de desenvolvimento esteja configurado adequadamente com as bibliotecas GLUT e GLEW.

O comando a seguir pode ser utilizado para compilar o exemplo prático de interoperação com OpenGL, em um ambiente Linux, assumindo que o código tenha sido armazenado em um arquivo chamado `clgl.c` e que o CUDA Toolkit tenha sido instalado no local padrão:

```
gcc -I/usr/local/cuda/include -lOpenCL -lGL -lglut -lGLEW  clgl.c
-o clgl
```

Após a compilação, a execução é realizada como no primeiro exemplo:

```
./clgl
```

## 6. Considerações finais

OpenCL possibilita a utilização do poder de processamento de CPUs e GPUs para programação de alto desempenho em ambientes computacionais heterogêneos. Apesar de seu surgimento recente, o padrão tem se mostrado uma alternativa viável às linguagens e ferramentas específicas de fabricante. Por ser aberto e livre de *royalties*, sua crescente adoção vem sendo observada ao longo dos últimos meses. As principais fabricantes de GPUs, NVIDIA e AMD/ATI, já oferecem amplo suporte. A AMD/ATI adotou OpenCL como ferramenta oficial para desenvolvimento em ambientes heterogêneos equipados com processadores AMD e placas gráficas ATI Radeon. Na mídia especializada, já circulam notícias sobre a implementação de suporte a OpenCL por parte de fabricantes de chips para dispositivos móveis, como *smartphones*.

Este material introduziu os conceitos-chave do padrão OpenCL, explorarando a arquitetura definida no mesmo e apresentando exemplos práticos. Mostrou a interoperação entre OpenGL e OpenCL, o que viabiliza a utilização da GPU não somente para a renderização no desenvolvimento de jogos de computadores, mas também para implementação de algoritmos de física, inteligência artificla, áudio, entre outros. Espera-se que o leitor possa fazer uso da da base apresentada neste material para a resolução seus problemas que demandam alto grau de paralelismo de forma mais rápida e eficiente.

O padrão OpenCL encontra-se atualmente na sua versão 1.1. O material aqui apresentado aplica-se, sem restrições, à versão anterior, 1.0, dado o suporte mais amplo a esta. No entanto, algumas obvervações foram feitas em pontos onde havia alguma diferença relevante entre as duas versões. Em termos gerais, a versão 1.1 aumenta a flexibilidade das operações suportadas pela versão 1.0, introduzindo também novos tipos de dados.

Para manter um canal de comunicação entre os leitores e os autores, a V3D disponibiliza este material, código-fonte e dicas de programação OpenCL no endereço `http://labs.v3d.com.br`.

## Referências


Amos G. Anderson, William A. Goddard III, P. S. (2007). Quantum monte carlo on graphical processing units. *Comp. Phys. Comm.*, 177(3):298–306.

Apple Inc. (2009). OpenCL Programming Guide for Mac OS X. `http://developer.apple.com/mac/library/documentation/Performance/Conceptual/OpenCL_MacProgGuide/Introduction/Introduction.html`.

Cavalheiro, G. G. H. and dos Santos, R. R. (2007). *Multiprogramação leve em arquiteturas multi-core*, pages 327–379. PUC-Rio, Rio de Janeiro.

Genovese, L., Ospici, M., Deutsch, T., Méhaut, J.-F., Neelov, A., and Goedecker, S. (2009). Density functional theory calculation on many-cores hybrid central processing unit-graphic processing unit architectures. *J Chem Phys*, 131(3):034103.

Göddeke, D., Strzodka, R., Mohd-Yusof, J., McCormick, P., Buijssen, S. H., Grajewski, M., and Turek, S. (2007). Exploring weak scalability for fem calculations on a gpu-enhanced cluster. *Parallel Computing*, 33(10–11):685–699.

ISO (2005). ISO/IEC 9899:1999 - C language specification. http://www.open-std.org/JTC1/SC22/wg14/www/docs/n1124.pdf.

Khronos Group (2010a). OpenCL - The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/.

Khronos Group (2010b). The OpenCL Specification. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf.

NVIDIA Corporation (2010). NVIDIA CUDA C Programming Guide. http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf.

Shreiner, D. and Group, T. K. O. A. W. (2009). *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 3.0 and 3.1 (7th Edition)*. Addison-Wesley Professional.