Programação em GPU utilizando OpenCL

André Tupinambá

andrelrt@gmail.com

Abstract. *The OpenCL is an open standard, maintained by the Khronos Group, for writing programs that execute across heterogeneous platforms and it has been adopted by major GPU and CPU vendors. OpenCL includes a language, an API, libraries and development support environment. The programming language is based on C99 with some additions to support the programming and memory models. This paper presents the OpenCL framework with a case.*

Resumo. *O OpenCL é um padrão aberto, definido pelo Khronos Group, para programação em dispositivo genérico. Hoje ele é suportado pelos principais fornecedores de GPUs e CPUs e espera-se que outros processadores tenham suporte em breve. O framework OpenCL é composto por uma linguagem, uma API, bibliotecas e um sistema de suporte para o desenvolvimento. A linguagem é baseada no padrão C99 com algumas extensões para suportar os modelos de memória e execução do OpenCL. Este artigo apresenta a plataforma OpenCL de programação para GPU, com um estudo de caso.*

1. Introdução

Até a popularização das placas aceleradoras 3D, pouco antes do ano 2000, poucas interfaces de vídeo eram capazes de algum processamento gráfico. Hoje as GPUs possuem um poder de processamento genérico na ordem de TeraFlops, superando os processadores de uso geral, que estão na ordem de centenas de GigaFlops [1].

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

No entanto, esses cálculos matemáticos são usados em muitas outras áreas de conhecimento, assim, surgiram pesquisas para o uso desse processamento em cálculos genéricos. Os primeiros trabalhos nessa área utilizaram as APIs existentes para geração de gráficos, como em [2], que já permitiam um bom desempenho, mesmo com alguma perda de precisão devida à falta de uma API específica.

A utilização de uma plataforma específica para processamento em GPUs surgiu principalmente com a empresa NVIDIA e sua plataforma CUDA. Esta plataforma possui uma descrição detalhada da arquitetura da GPU, os modelos de memória e uma linguagem de programação própria [1]. Posteriormente outras empresas, como ATI e IBM, também disponibilizaram plataformas equivalentes.

Uma dessas iniciativas foi a plataforma genérica de acesso a GPUs da Apple, posteriormente padronizada pelo Khronos Group, o OpenCL [3]. O OpenCL define um modelo processamento em dispositivo genérico (podendo ser GPU, CPU ou um

acelerador, como um FPGA ou o IBM Cell/B.E.) que também possui arquitetura específica, modelos de memória, API de acesso e linguagem de programação própria.

Este artigo apresenta a plataforma OpenCL de programação em GPU está assim organizado. No Capítulo 2 está a apresentação da plataforma OpenCL; no Capítulo 3 está um estudo de caso utilizando OpenCL; e no Capítulo 4 as considerações finais.

2. OpenCL

O OpenCL é um padrão aberto, definido pelo Khronos Group, para programação em dispositivo genérico. Hoje ele é suportado pelos principais fornecedores de GPUs (NVIDIA, AMD e, recentemente, Intel) e CPUs (Intel, AMD e IBM); e a tendência é que outros processadores tenham suporte em breve, pois já existem chips para celulares homologados, como o CPU ARMv7 com Mali-T604 GPU [4], e outros chips, como o FPGA da empresa Altera [5], em desenvolvimento.

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

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

2.1. Modelo de plataforma

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

O modelo de plataforma descreve uma arquitetura composta de um computador (*host*) conectado em um ou mais dispositivos OpenCL (*compute device*). Cada dispositivo é dividido em uma ou mais unidades de computação (*compute units*), sendo que estes também são divididos em elementos de processamento (*processing elements*). Na Figura 1 temos a ilustração do manual do OpenCL [3] mostrando as relações entre os componentes.

A execução é realizada pelas unidades de computação e elementos de processamento do dispositivo. Nas GPUs esses dois papéis são distintos e bem definidos. As unidades de computação são grupos de elementos de processamento (os elementos são chamados de *CUDA Cores* na plataforma NVIDIA e *Stream Processors* na plataforma AMD) que mantém o mesmo ponto de execução, normalmente utilizando diferentes dados, em uma estrutura SIMD [6]. Nas CPUs esses dois papéis se confundem, sendo normalmente sinônimos. As aplicações OpenCL sempre rodam no computador host e enviam comandos para os dispositivos realizarem operações e cálculos específicos. Neste aspecto, o host e o dispositivo se assemelham ao modelo de processamento distribuído com memória independente, pois toda a comunicação é realizada por troca de mensagens, sendo que somente o host pode iniciar uma comunicação.

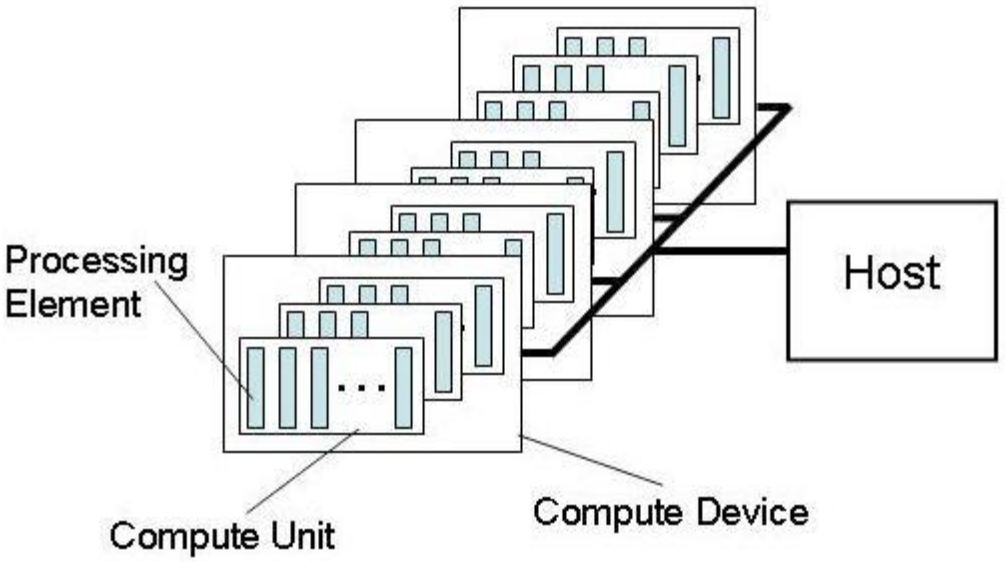

Figura 1. Modelo de plataforma do OpenCL

2.2. Modelo de execução

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

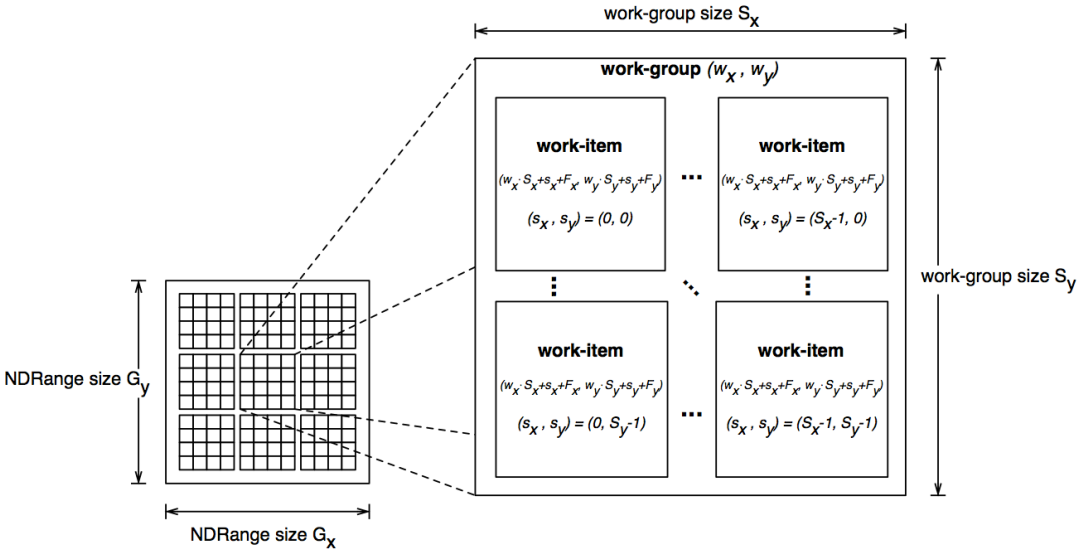

Figura 2. Modelo de execução do OpenCL

Para realizar a execução nos dispositivos, o *host* envia para os dispositivos os comandos para execução de códigos, conhecidos como *kernels*. Cada dispositivo inicia então um conjunto de *threads* referenciando o *kernel* submetido. Cada *thread* do *kernel*

será executado em um elemento de processamento diferente do dispositivo, em paralelo, recebendo os mesmos parâmetros de entrada. O OpenCL denomina esta *thread* do *kernel* como *work-item*.

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

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

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

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

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

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

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

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

2.3. Modelo de memória

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

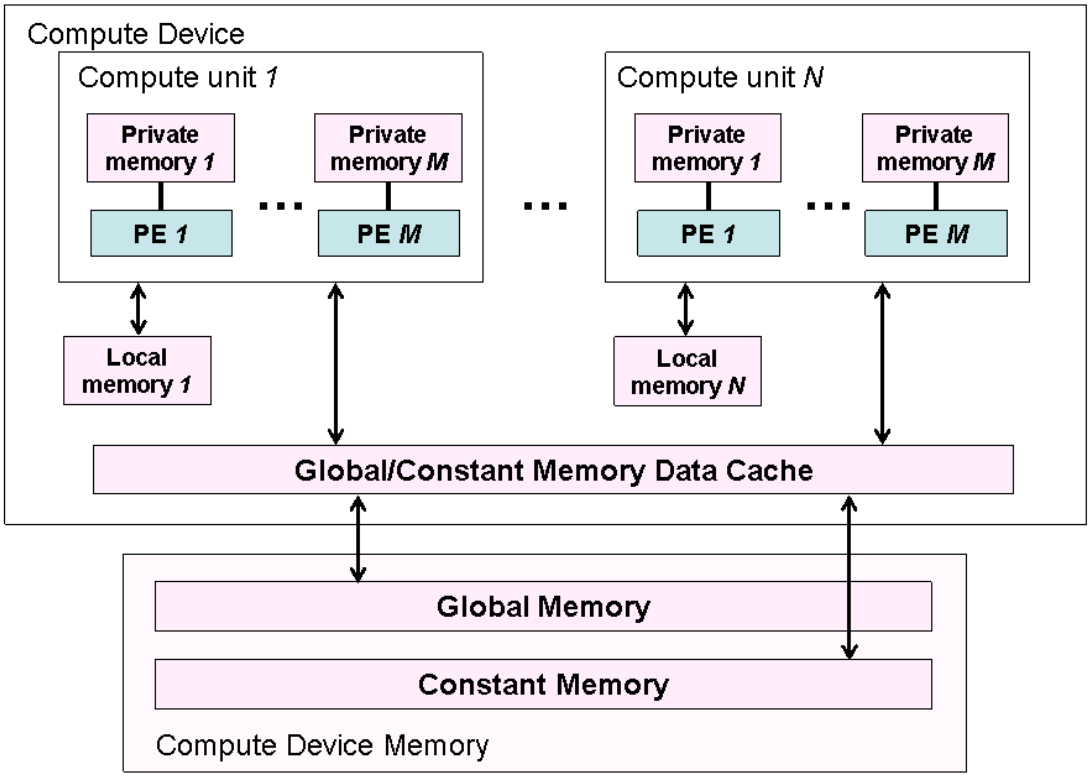

Figura 5. Modelo de memória do OpenCL

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

A memória local é utilizada pelos *work-items* de uma única unidade de computação, no entanto o host não possui acesso à mesma. Nas GPUs, normalmente, esta memória é menor (normalmente de 16k nas placas NVIDIA) e mais rápida do que as memórias globais e de constantes. Como se trata de um recurso escasso e de alta velocidade, a maximização do uso da memória local é um importante fator para a definição do tamanho dos *work-groups*.

A memória privada é utilizada exclusivamente por cada *work-item*, sendo também considerada rápida e inacessível à aplicação no host.

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

2.4. Modelo de programação

O OpenCL possui uma linguagem própria, baseada no C99, e dois modelos de programação, com dados paralelos, mais comum, e com tarefas paralelas. Um detalhe interessante é que a compilação do código OpenCL é feito em tempo de execução da aplicação no *host*, ou seja, 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.

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

Dos tipos de dados da linguagem C99, a linguagem OpenCL não suporta os tipos `long long` e `long double`, porém fornece mais 19 tipos de dados, escalares e vetoriais. Na Tabela1 estão os novos tipos de dados suportados pelo OpenCL.

Tabela 1. Novos tipos da linguagem OpenCL

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

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

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

Existem algumas limitações na linguagem. Os *kernels* só podem receber ponteiros `__global`, `__constant` ou `__local`, não podem receber ponteiros para ponteiros, não podem receber um `event_t` e só podem retornar `void`. Não são suportados: recursão, *bitfields*, ponteiros para funções, vetores sem tamanho definido,

macros e funções com número indefinido de parâmetros (com reticências) e as palavras reservadas `extern`, `static`, `auto` e `register`.

Tabela 2. Novos qualificadores da linguagem OpenCL

Qualificador	Descrição
`__global` ou `global`	Indica que o objeto de memória (buffer ou imagem) está armazenado na memória global do dispositivo.
`__local` ou `local`	Indica que a variável está armazenada na memória local do dispositivo e é visível por todos os *work-items* de um mesmo *work-group*.
`__constant` ou `constant`	Indica que a variável está armazenada na memória de constantes do dispositivo.
`__private` ou `private`	Indica que a variável está armazenada na memória privada do dispositivo e só é visível pelo próprio *work-item*.
`__kernel` ou `kernel`	Indica que a função pode se transformar em um *kernel* e ser executada pelo host através das funções `clEnqueueNDRangeKernel` ou `clEnqueueTask`.
`__read_only` ou `read_only`	Indica que o objeto imagem foi passado como somente leitura. Não é possível, pelo padrão, ler e escrever no mesmo objeto de imagem.
`__write_only` ou `write_only`	Indica que o objeto imagem foi passado como somente escrita. Não é possível, pelo padrão, ler e escrever no mesmo objeto de imagem.

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

Na linha 1, o qualificador `__kernel` indica que a função `vecmul` pode ser chamada do *host*, e que ela retorna `void`, que é o único valor possível. Na linha 2 e 3 estão os parâmetros `vectorA`, `vectorB` e `result` que são ponteiros, para o tipo `double`, que estão na memória global do dispositivo. Na linha 5 é lido o *global ID* deste *work-item*, que é usado como índice para a leitura dos dados nas linhas 6 e 7. Na linha 8 é feita a multiplicação dos dois valores lidos e o resultado é salvo.

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

Figura 6. Exemplo de um código OpenCL com dados paralelos

Na Figura 7 está um exemplo de um *kernel*, com tarefas paralelas, na linguagem OpenCL. O modelo de programação com tarefas paralelas permite que códigos diferentes sejam executados em *work-items* diferentes e operem em quaisquer dados.

Para acionar este modo, deve-se utilizar a chamada `clEnqueueTask` da API OpenCL para criar somente um *work-item*. Este modelo não é muito utilizado mesmo com dispositivos CPU, pelo ganho pelo uso de paralelismo nos ambientes multiprocessados e as instruções SIMD, comum nas CPUs atuais.

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

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

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

3. Aplicações utilizando OpenCL

Uma aplicação típica utilizando OpenCL possui quatro passos principais. O primeiro passo é o levantamento dos dispositivos disponíveis, a criação do contexto de execução, que serve como referência de todos os buffers, programas e *kernels* que serão utilizados pela aplicação, e a criação da fila de comando de cada dispositivo.

O segundo passo é o envio de dados para o dispositivo. Normalmente a memória é exclusiva do dispositivo e inacessível à CPU, como é no caso da GPU, então é necessário que os dados que serão manipulados sejam inicialmente copiados para o dispositivo, através da API OpenCL.

O terceiro passo é a execução do *kernel* no dispositivo. São passados os parâmetros através de chamadas à API OpenCL e comandada a execução do *kernel*, que é executado de forma assíncrona em relação à aplicação no *host*. Ou seja, a execução inicia no dispositivo enquanto a aplicação está livre para realizar outras operações.

O quarto e último passo é a leitura dos resultados gerados pela execução no dispositivo. Como discutido no segundo passo, é necessário chamar a API OpenCL para copiar os dados do dispositivo de volta para a memória do *host* para verificar os resultados.

3.1. Multiplicação de vetor

Na Figura 6, apresentada anteriormente, está um exemplo simples de uma multiplicação entre dois vetores utilizando OpenCL. No entanto, para executar este código é necessário construir uma aplicação para enviar os dados para o dispositivo, acionar a execução e ler os resultados. A API OpenCL é composta por funções em C, porém existe um *wrapper* C++, que é utilizado neste exemplo.

O primeiro passo é ler as características do dispositivo, criar o contexto de execução e a fila de comando, isso está apresentado na Figura 8. Este exemplo utiliza

somente um dispositivo, mas o conceito pode ser estendido para o uso de multidispositivos.

Na linha 2 o programa lê as plataformas OpenCL disponíveis no ambiente. Para simplificar o exemplo, a verificação dos resultados, como o da linha 3, foi omitida, no entanto ela deve ser feita em todos os casos em que a variável `err` foi retornada. A linha 6 seleciona somente a primeira plataforma para ler todas as GPUs existentes. Na linha 10 seleciona somente um dispositivo para a criação de um contexto de execução na linha 12. Este contexto é utilizado para criar a fila de comando do dispositivo na linha 13.

```
1:  std::vector<cl::Platform> platforms;
2:  cl_int err = cl::Platform::get( &platforms );
3:  if( err != CL_SUCCESS ) return;
4:
5:  std::vector<cl::Device> devs;
6:  cl::Platform platform = *platforms.begin();
7:  platform.getDevices( CL_DEVICE_TYPE_GPU, &devs );
8:  if( devs.empty() ) return;
9:
10: devs.resize(1);
11: cl_context_properties cps[3] =
        { CL_CONTEXT_PLATFORM,
          (cl_context_properties)(*platforms.begin())(),
          0 };
12: cl::Context context( devs, cps, 0, 0, &err );
13: cl::CommandQueue queue( context, devs[0], 0, &err );
```

Figura 8. Criação do contexto de execução

O segundo passo é a passagem de dados do *host* para o dispositivo e para isso é necessário primeiro criar os objetos de dados no dispositivo, isso está demonstrado na Figura 9. Nas linhas 1 e 3 os dois vetores, `vectorA` e `vectorB`, são criados no dispositivo e seus valores são enviados do host através das chamadas das linhas 5 e 7, respectivamente. Os vetores contêm números do tipo `cl_double` de tamanho `size` e inicializados com as áreas em memória passadas por `bufferA` e `bufferB`.

```
1:  cl::Buffer vectorA( context, CL_MEM_READ_ONLY,
                        sizeof(cl double) * size, NULL, &err);
2:
3:  cl::Buffer vectorB( context, CL_MEM_READ_ONLY,
                        sizeof(cl_double) * size, NULL, &err);
4:
5:  err = queue.enqueueWriteBuffer( vectorA, CL_FALSE, 0,
        sizeof(cl_double) * size, bufferA, NULL, NULL );
6:
7:  err = queue.enqueueWriteBuffer( vectorB, CL_FALSE, 0,
        sizeof(cl_double) * size, bufferB, NULL, NULL );
```

Figura 9. Criação dos buffers e transferência de dados

O terceiro passo é a execução do *kernel* no dispositivo. Para isso é necessário compilar o código OpenCL, retornar o *kernel*, passar os parâmetros e comandar a execução, este processo está na Figura 10. No exemplo, a função possui três parâmetros, dois buffers de entrada e um de retorno, assim, também é necessário criar este buffer de retorno para poder ser passado seu ponteiro como parâmetro, feito na linha 1. Na linha 3 a variável `std::string source`, que contém o fonte da Figura 6, é passado para uma estrutura para a criação de um *program* na linha 4. Na linha 5 o fonte é compilado e é

pego o *kernel* (de alguma forma similar ao ponteiro) da função `vecmul` na linha 6. Estes passos só precisam ser feitos uma única vez, a partir desde ponto, o *kernel* está pronto para ser utilizado, quantas vezes forem necessárias.

Nas linhas 8 a 10 são passados os parâmetros do *kernel* para ser executado na linha 11. Note que na linha 11 são iniciadas `size` *threads* de execução no dispositivo, devido ao parâmetro global passado.

```
1:  cl::Buffer result( context, CL_MEM_WRITE_ONLY,
                       sizeof(cl_double) * size, NULL, &err);
2:
3:  cl::Program::Sources sources( 1,
        std::make_pair( source.c_str(), source.length() ) );
4:  cl::Program program( context, sources );
5:  err = program.build( devs );
6:  cl::Kernel kernel( program, "vecmul", &err );
7:
8:  kernel.setArg( 0, vectorA );
9:  kernel.setArg( 1, vectorB );
10: kernel.setArg( 2, result );
11: queue.enqueueNDRangeKernel( kernel, cl::NullRange, // offset
                                cl::NDRange( size ),   // global
                                cl::NullRange, 0, 0 ); // local
```

Figura 10. Execução do kernel

Após a execução do *kernel*, o último passo é a leitura dos resultados, demonstrado na Figura 11. Vale lembrar que, como a execução é assíncrona ao processamento do *host*, é necessário criar um ponto de sincronismo. Isso pode ser feito de duas formas, uma implicitamente obrigando a leitura dos dados serem síncronos, ou explicitamente, através de uma chamada à API OpenCL. Na linha 1 é solicitada a leitura dos dados do buffer `result` e gravação na memória `bufRet`. Esta leitura é realizada de forma assíncrona, então é necessário solicitar ao OpenCL que termine todas as tarefas pendentes na fila de comando através da linha 2.

```
1: queue.enqueueReadBuffer( result, CL_FALSE, 0,
       sizeof(cl_double) * size, bufRet, NULL, NULL );
2: queue.finish();
```

Figura 11. Leitura dos dados do dispositivo

4. Conclusão

O OpenCL oferece uma estrutura padrão de acesso a diversos dispositivos existentes, hoje ele é suportado por produtos das empresas AMD, NVIDIA, Intel, IBM, Altera, Samsung, ARM, entre outras. Com ele é possível ao desenvolvedor a criação de programas que podem ser executados em diversos dispositivos, sem que seja necessário prévio conhecimento da arquitetura do mesmo.

Existem alternativas ao OpenCL para programação de GPUs. O mais conhecido é o CUDA, exclusivo para GPUs da NVIDIA. No entanto, existem outras propostas, como os OpenACC e OpenHMPP baseados em diretivas `#pragma`, similar ao OpemMP; e os DirectCompute e C++AMP da Microsoft.

Referências

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

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

3. KHRONOS GROUP. **The OpenCL Specification - Version 1.1**. [S.l.]: [s.n.], 2010.

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

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

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