OpenCL Tutorials – 1 – Hello World

Olá a todos,
este é meu primeiro tutorial sobre OpenCL, e espero que seja útil.
Não falarei aqui sobre OpenCL em si, já que há por aí suficientes introduções a OpenCL. Começarei o tutorial falando um pouco sobre o modelo de execução de OpenCL, mas será algo breve. Teremos tempo mais tarde para isso.
Neste tutorial, o que eu gostaria de providenciar é um *quickstart* para que fique mais tranquilo começar a usar e a fazer testes em OpenCL.

Os códigos deste tutorial encontram-se aqui:

- main.cpp
- vector_add.cl

Um programa OpenCL se divide em duas partes: a execução do *host*, que é a execução na CPU, como já a conhecemos. Nesta etapa apenas são adicionadas algumas instruções para preparar a execução do *device*.
A outra parte, a parte do *device*, refere-se à computação que ocorrerá, no nosso caso, na GPU. Para esta deveremos nos acostumar com alguns conceitos básicos. O primeiro deles é o de *kernel*. Um *kernel*, como já foi citado no blog, é uma função que executa no device. Uma característica importante do modelo é que temos uma execução SIMT (Single Instruction Multiple Thread), ou seja, o mesmo kernel é executado simultaneamente com dados diferentes.
Aceita essa idéia, passamos agora para alguns conceitos auxiliares:

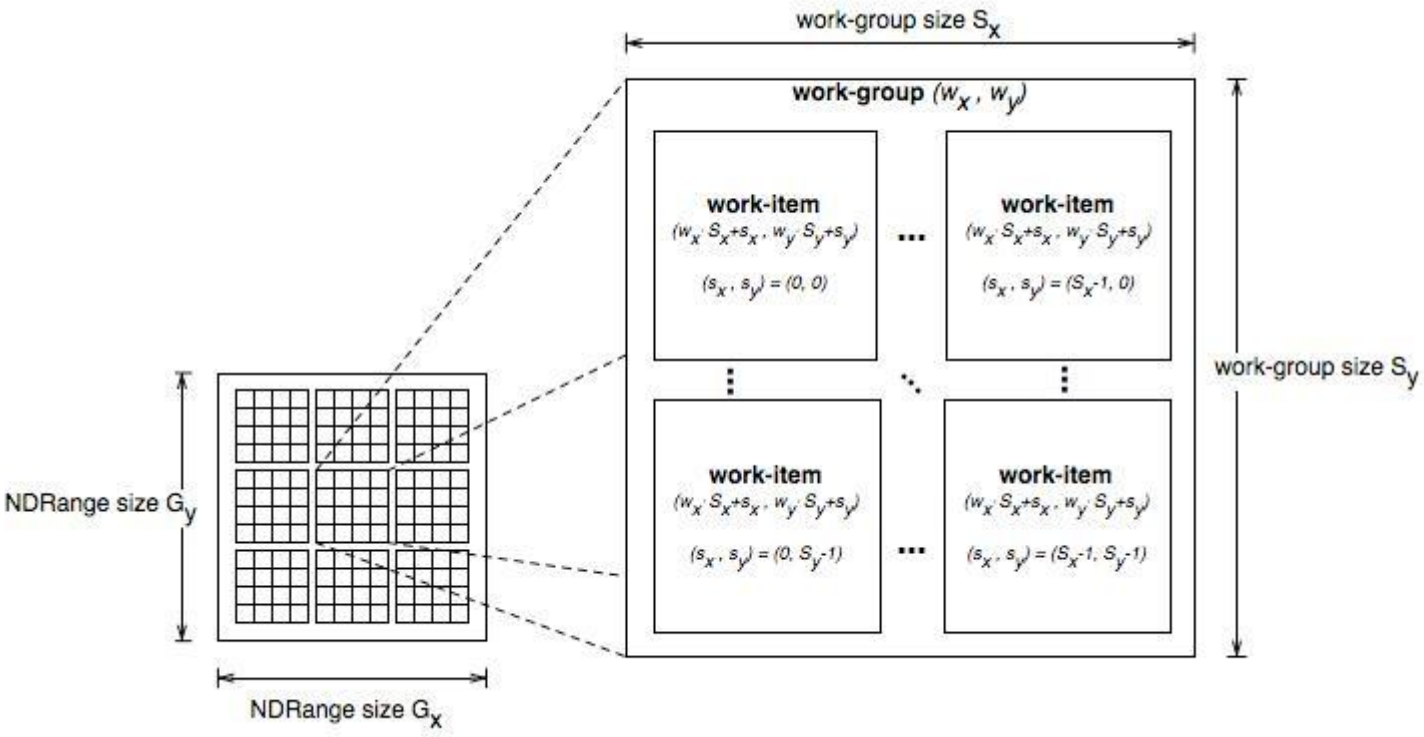
- **work-item**: sempre que um kernel é lançado, cada instancia em execução desse kernel torna-se um work-item. Work-items são identificados por um índice que pode ser acessado de dentro do kernel;

Na arquitetura NVIDIA, cada work-item é mapeado para uma CUDA thread;

- **work-group**: é como os work-items são agrupados. É importante ressaltar que esse agrupamento é muito mais importante do que pode parecer a primeira vista, dado que todos os work-items pertencentes a um work-group são executados na mesma *Compute Unit* (que são mapeados, na arquitetura NVIDIA, para Scalar Processors (SPs)), o que lhes permite interagir entre si de forma muito mais eficiente. Work-groups são mapeados, em CUDA, para *thread blocks*;

Work-groups possuem, assim como os work-items, IDs únicos que podem ser acessados de dentro do kernel;

- **NDRange**: um NDRange é um grid N-dimensional (com N = 1, 2 ou 3) de work-groups. Eles também são acessados por um ID;



Este é um exemplo de NDRange (retirado da especificação liberada pelo Khronos Group).

OK. Acho que é o suficiente por hoje de conceitos. Mais tarde teremos a oportunidade de estudar mais a fundo essas características da linguagem e da arquitetura. Por hora é tudo com o qual devemos nos preocupar.
Agora gostaria de desenvolver aqui um primeiro exemplo prático. Trata-se de um programa muito simples, que computa a soma de dois vetores.
Este exemplo é fortemente baseado no exemplo análogo da NVIDIA, que vem junto com o SDK do OpenCl.

Kernel

Certo, vamos começar com a parte fácil da história: programando o kernel.
Suponhamos que estivéssemos interessados em fazer uma função comum para calcular a soma entre dois vetores. Ela seria mais ou menos assim:

```
void vector_add_golden (const float * const srcA,
                                    const float * const srcB,
                                    float * const golden) {
  for (int i = 0; i < iNumElements; i++)
          golden[i] = srcA[i] + srcB[i];
}
```

O que temos aí é um laço que itera por todos os elementos do vetor e soma seus valores, colocando o resultado no terceiro vetor.
Se quiséssemos fazer isso em OpenCL, escreveríamos uma função que soma **um** elemento do vetor, cujo índice seria o índice da instancia do kernel em execução. Complicado? Vamos ver um exemplo:

```
__kernel void vector_add (__global const float * const v1,
                  __global const float * const v2,
                  __global float * const res,
                  __global const int n) {
  int idx = get_global_id(0);

  if (idx < n)
```

```
            res[idx] = v1[idx] + v2[idx];
    }
```

Percebam que utilizamos a palavra reservada **__kernel__** para especificar que trata-se de um kernel.
Outro detalhe está em

```
    int idx = get_global_id(0);
```

essa instrução coloca em 'idx' o ID da instancia do kernel em execução.

Em seguida o que fazemos é o seguinte: O kernel se pergunta: "Meu ID está dentro dos limites do vetor?". Caso afirmativo, ele computa a soma **desse** elemento, e caso negativo nada é feito.

Alguns detalhes adicionais:

Esse kernel deve estar obrigatóriamente em um arquivo com extensão ".cl", que deve ser separado dos arquivos ".cpp" ou ".c" que contiverem código a ser compilado.
Nos arquivos ".cl" deve constar **apenas** código OpenCL, visto que o código OpenCL é compilado em tempo de execução e deve ser 'invocado' de algum lugar.

Inicializando...

Ok. Falta agora apenas a preparação para o lançamento do kernel. Mas não sejamos tão apressados. OpenCL é uma linguagem de baixo nível, portanto nos cabe o trabalho de fazer grande parte das coisas que, por exemplo, CUDA faz por nós.
Mas vamos por partes. Precisaremos, inicialmente, dos headers necessários. Isso é facil:

```
#include <oclUtils.h>
```

Este é um arquivo disponibilizado juntamente com os exemplos da NVIDIA, e que nos dá algumas funções interessantes, mas que não é necessário.
Ele inclui sozinho o arquivo *CL/cl.h*, que é, esse sim, o verdadeiro necessário.

Faremos também algumas declarações adicionais:

```
#include <iostream>

#define NUM 11444777

using namespace std;

/* Estes são, em ordem, o nome do arquivo em que se encontra nosso kernel,
 * o número de elementos da computação e um inteiro que será utilizado para
 * verificação de erros;
 */
const char* cSourceFile = "vector_add.cl";
cl_int iNumElements = NUM;
cl_int error = 0;
```

Ok, mais uma tarefa fácil agora: alocar a memória do host, inicializá-la e computar no host para futura verificação:

```
            /* Aqui são alocadas as variáveis que residem na CPU. São elas os dois
vetores a serem somados,
         * e os dois vetores com os resultados da computação no host e no device;
         */
        cl_float * const srcA = (cl_float*) malloc(sizeof(cl_float)*iNumElements);
        cl_float * const srcB = (cl_float*) malloc(sizeof(cl_float)*iNumElements);
        cl_float * const golden = (cl_float*) malloc(sizeof(cl_float)*iNumElements);
        cl_float * const res = (cl_float*) malloc(sizeof(cl_float)*iNumElements);

        /* Fazemos a devida inicialização dos vetores a serem somados.
         */
        for (int i = 0; i < iNumElements; ++i) {
                srcA[i] = srcB[i] = i;
```

```
        }

        /* Aqui fazemos o cálculo da soma na CPU, para fins de verificação.
         */
        vector_add_golden(srcA, srcB, golden);
```

Contexto e Command-queue

Muito bem, a esta altura já estamos com tudo pronto para criar nosso contexto e nossa command-queue.
Um contexto engloba todas as variaveis e os registradores, ou seja, o estado de uma computação. A command-queue, por outro lado, é a fila de comandos que devem ser efetuados no programa.
Para criar o contexto, utilizaremos a seguinte função:

```
cl_context clCreateContextFromType5 (cl_context_properties *properties,
                                     cl_device_type device_type,
                                     void (*pfn_notify)(const char *errinfo,
                                     const void *private_info, size_t cb,
                                     void *user_data),
                                     void *user_data,
                                     cl_int *errcode_ret)
```

Onde:

- *properties* é um vetor de propriedades co contexto, que por hora será passado como NULL;
- *cl*device*type* especifica o tipo do device. A tabela completa encontra-se na especificação, utilizaremos CL*DEVICE*TYPE_GPU, pois estamos criando o contexto na GPU;
- As proximas duas informações tampouco serão importantes por hora, e serão passadas como NULL;
- *errcode*ret_ é o inteiro, que declaramos lá encima, que conterá o código de um eventual erro;

Se tudo der certo, será retornado CL_SUCCESS.

Nosso código fica então:

```
        cl_context context = clCreateContextFromType (NULL, CL_DEVICE_TYPE_GPU, NULL,
NULL, &error);
        if (error != CL_SUCCESS) {
                cout << "Erro criando contexto" << endl;
                exit (error);
        }
```

Para criar a command-queue, precisaremos primeiramente da lista de *devices* de nosso sistema. Para fazer isto, utilizaremos a função *clGetContextInfo*, como pode ser visto abaixo:

```
        // Estes são, respectivamente, o vetor de devices a ser preenchido e o tamanho
em memória desse vetor;
        cl_device_id *devices;
        size_t devices_size;
        // Chamamos a função uma primeira vez, para conseguir o tamanho em memória que
deve ser alocado a devices;
        error = clGetContextInfo (context, CL_CONTEXT_DEVICES, 0, NULL, &devices_size);
        // Alocamos o espaco necessario
        devices = (cl_device_id*) malloc (devices_size);
        // E chamamos novamente a funcao para efetivamente colocar em 'devices' a
informacao de todos os devices (no meu caso apenas uma GPU)
        error |= clGetContextInfo (context, CL_CONTEXT_DEVICES, devices_size, devices,
0);
        if (error != CL_SUCCESS) {
                cout << "Erro pegando informacoes do contexto;" << endl;
                exit (error);
        }
```

Muito bem, já estamos prontos para a criação da command_queue:

```
        cl_command_queue queue = clCreateCommandQueue (context, *devices, 0, &error);
```

```
        if (error != CL_SUCCESS) {
                cout << "Erro criando command_queue" << endl;
                exit (error);
        }
```

Alocando memória no device

Ok, já criamos as estruturas necessárias para gerenciar a execução OpenCL. Vamos agora alocar os vetores para serem somados na GPU, e em seguida invocar o kernel.
Em OpenCl, temos que alocar um buffer, que é um espaço de memória a ser alocado em um contexto.
A alocação tem a seguinte assinatura:

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

- As flags podem ser:
  - CL_MEM_READ_WRITE
  - CL_MEM_WRITE_ONLY
  - CL_MEM_READ_ONLY
  - CL_MEM_USE_HOST_PTR
  - CL_MEM_ALLOC_HOST_PTR
  - CL_MEM_COPY_HOST_PTR

E devem ser combinadas usando |

Nosso código fica da seguinte forma:

```
        cl_int error2 = 0, error3 = 0;
        cl_mem srcA_d = clCreateBuffer (context, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, sizeof(cl_float) * iNumElements,
                (void*)srcA, &error);
        cl_mem srcB_d = clCreateBuffer (context, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, sizeof(cl_float) * iNumElements,
                (void*)srcB, &error2);
        cl_mem res_d = clCreateBuffer (context, CL_MEM_WRITE_ONLY, sizeof(cl_float) *
iNumElements, NULL, &error3);
        if ((error | error2 | error3) != CL_SUCCESS) {
                cout << "Erro alocando memoria no host" << endl;
                exit (error|error2|error3);
        }
```

"Criando" um programa e um kernel

Ok, criamos os Buffers, agora precisamos "criar" um programa e compilá-lo. Na verdade o que precisamos é informar quais são nossos códigos OpenCl para que possam ser compilados.
Faremos isso usando:

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

para criar o programa, e:

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

para compila-lo;

Segue o código fonte desta parte:

```
        // Estas três linhas apenas conseguem o endereço do kernel
        size_t kernelLength;
              const char* cPathAndName = shrFindFilePath(cSourceFile, argv[0]);
              const char* cSourceCL = oclLoadProgSource(cPathAndName, "",
&kernelLength);
        cl_program program = clCreateProgramWithSource (context, 1, (const
char**)&cSourceCL, &kernelLength, &error);
        if (error != CL_SUCCESS) {
                cout << "Erro criando o programa" << endl;
                exit (error);
        }
        error = clBuildProgram (program, 0, NULL, NULL, NULL, NULL);
        if (error != CL_SUCCESS) {
                cout << "Erro buildando o programa" << endl;
                exit (error);
        }
```

A seguir precisamos agora criar um 'kernel object', que deve ser associado ao programa que acabamos de criar. Isso pode ser feito utilizando:

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

Ficando o código da seguinte maneira:

```
        cl_kernel vector_add_kernel = clCreateKernel (program, "vector_add", &error);
        if (error != CL_SUCCESS) {
                cout << "Erro criando o kernel" << endl;
                exit (error);
        }
```

Invocando o Kernel

Ok, já criamos um contexto, descobrimos os devices, criamos um command_queue, alocamos a memória, criamos um programa e associamos os kernels.
Basta agora chamar os kernels para que esteja tudo concluído. Para fazer isso, devemos primeiramente empilhar os argumentos. Isso pode ser feito usando:

```
         cl_int clSetKernelArg (cl_kernel kernel,
                                cl_uint arg_index,
                                size_t arg_size,
                                const void *arg_value)
      Onde:
      * _kernel_ é o kernel ao qual deve ser empilhado o parametro;
      *  {"arg index"}  é o índice do parâmetro a ser empilhado (primeiro parametro,
segundo parametro...);
      * _{"arg_size"}_ é o tamanho do parametro;
      * _{"*arg_value"}_ é o valor do argumento;
```

Nosso código de empilhamento de parametros fica da seguinte forma:

```
        error = clSetKernelArg (vector_add_kernel, 0, sizeof(cl_mem), (const
void*)&srcA_d);
        error |= clSetKernelArg (vector_add_kernel, 1, sizeof(cl_mem), (const
void*)&srcB_d);
        error |= clSetKernelArg (vector_add_kernel, 2, sizeof(cl_mem), (const
void*)&res_d);
        error |= clSetKernelArg (vector_add_kernel, 3, sizeof(cl_int), (const
void*)&iNumElements);
        if (error != CL_SUCCESS) {
                cout << "Error empilhando os parametros" << endl;
                exit (error);
        }
```

Agora chegou o tão esperado momento de chamar o kernel. Para tanto utilizaremos a seguinte função:

```
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 num_events_in_wait_list,
                               const cl_event *event_wait_list,
                               cl_event *event)
```

Onde:

- *work_dim* é o número de dimensões dos work-items (1,2 ou 3);
- **global_work_offset* deve ser mantido NULL (pelo menos nesta versão do driver opencl da NVIDIA;
- **global_work_size* é um vetor que descreve as dimensões dos work-items;
- **local_work_size* é o número de work-items por work-group (em uma analogia a CUDA, é o número de threads per block;
- O retorno é o código de erro;
- os outros parametros não importam por hora;

Aqui declaramos o numero de work-items por work-group e o número de work-items total, e em seguida chamamos o kernel;
shrRoundUp arredonda o numero de elementos para o maior multiplo do numero de work-items por work-group;

```
size_t work_groups = 256;
      size_t work_items = shrRoundUp((int)work_groups, iNumElements);

error = clEnqueueNDRangeKernel (queue, vector_add_kernel, 1, NULL, &work_items,
&work_groups, 0, NULL, NULL);
if (error != CL_SUCCESS) {
        cout << "Erro empilhando os kernels" << endl;
        exit (error);
}
```

Recuperando os valores e fazendo o check:

Chamamos o kernel! Não foi tão difícil.
Agora temos que conseguir de volta o resultado. Para isso, devemos enfileirar uma requisição de leitura de buffer. Isso pode ser feito da seguinte maneira:

```
cl_int clEnqueueReadBuffer (cl_command_queue command_queue,
                            cl_mem buffer,
                            cl_bool blocking_read,
                            size_t offset,
                            size_t cb,
                            void *ptr,
                            cl_uint num_events_in_wait_list,
                            const cl_event *event_wait_list,
                            cl_event *event)
```

onde:

- *buffer* é o buffer alocado no device que queremos ler;
- *blocking_read* é um booleano que indica se a leitura será bloqueante ou não. Isso não é importante agora;
- *cb* é o tamanho dessa memória;
- **ptr* é um ponteiro para um espaço de memória (que ja deve estar com *cb* bytes alocados) de destino, no HOST;

- O retorno é o código de erro;

Nosso código fica então da seguinte maneira:

```
        error = clEnqueueReadBuffer (queue, res_d, CL_TRUE, 0,
sizeof(cl_float)*iNumElements, (void *)res, 0, NULL, NULL);
        if (error != CL_SUCCESS) {
                cout << "Erro lendo o buffer" << endl;
                exit (error);
        }
```

Faremos agora o check com a computação no host:

```
        for (int i = 0; i < iNumElements; ++i) {
                if (golden[i] != res[i]) {
                        cout << "Nao fecha a computacao" << endl;
                        exit (1);
                }
        }
        cout << "Passou no TESTE!!!" << endl;
```

E pronto, nosso programa está concluído!
Pode parecer que é esforço demasiado, mas a questão é que essas etapas apenas serão desenvolvidas uma vez. Todos os códigos utilizarão basicamente a mesma estrutura, e não acredito que leve muito tempo para alguem fazer algum *wrapper*.

Limpeza

Feito o trabalho, se quisermos ser programadores responsáveis, devemos limpar a memória que utilizamos. Toda inicialização com *clCreate* deve ser desalocada usando *clRelease*
Assim temos, para nosso exemplo:

```
        clReleaseKernel(vector_add_kernel);
        clReleaseProgram(program);
        clReleaseCommandQueue(queue);
        clReleaseContext(context);
        clReleaseMemObject(srcA_d);
        clReleaseMemObject(srcB_d);
        clReleaseMemObject(res_d);
```

Chegamos assim ao fim deste tutorial.
Muito obrigado pela atenção. Quaisquer dúvidas podem me contatar.
Abraços

Leonardo Chatain