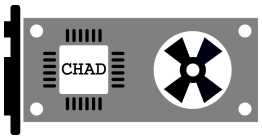

LAB 7 – OpenCL em GPU

Computação Heterogénea de Alto Desempenho (2019/2020)

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

O **OpenCL** é um standard aberto para programar uma colecção de CPUs, GPUs e outros dispositivos organizados numa única plataforma. É mais do que uma linguagem. O OpenCL é uma framework para programação paralela e inclui uma linguagem, API, bibliotecas e um sistema runtime para suportar desenvolvimento de software.

1. Analise o código seguinte que representa uma operação de multiplicação e o resultado é guardado num vector de inteiros (este exemplo está disponível como material de apoio). Neste programa identifica-se:

1.1. Função que executa na GPU (OpenCL kernel): Funções que executam na GPU são especificadas com a cláusula **`__kernel`**.

```
-----------------------------------------------------------------
// Device code
__kernel void device_func_name(__global int * device_buffer, int N)
{
      // Work-item identifier (1 dimensional problem)
      int index=get_global_id(0);
      // CODE
      if(index<N)
            device_buffer[index]=index*2;
}
-----------------------------------------------------------------
```

1.1.1. A **kernel OpenCL** que executa na GPU deve ter um índice associado que identifica o work-item no contexto do programa. O index space suportado em OpenCL é chamado de NDRange.

Uma instância da kernel executa em cada ponto do index space. Esta instância da kernel é chamada **work-item** e é identificada pelo seu ponto no index space, que é o global ID para o work-item.

Cada work-item executa o mesmo código mas o caminho de execução e os dados sobre o qual opera podem variar para cada work-item.

Work-items (threads em CUDA) são organizados em **work-groups** (blocks em CUDA). Os work-groups proporcionam uma decomposição mais granular do index space. Cada work-group tem um único work-gourp ID com a mesma dimensão que o index space usado para os work-items. Os work-items possuem um único local ID no work-group. Assim cada work-item pode ser identificado pelo seu global ID ou pela combinação do local ID com o work-group ID. Os work-items num work-group executam concorrentemente nos processing elements de uma única compute unit.

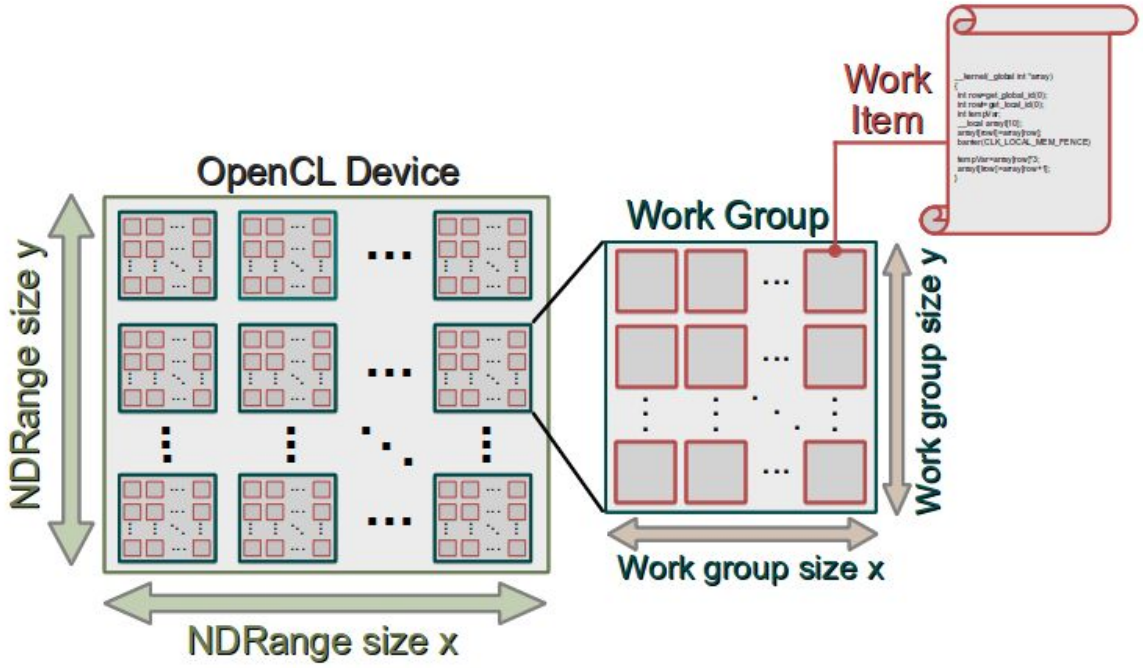

Figure 1: Exemplo de um NDRande index space (2D) que mostra os work-items agrupados em work-groups.

1.1.2. Descrição do algoritmo: No exemplo usado está definida a inicialização de um vetor que multiplica o índice do vetor por 2.

```
----------------------------------------------------------------
      ...
      // CODE
      if(index<N)
            device_buffer[index]=index*2;
      ...
----------------------------------------------------------------
```

O OpenCL possui um modelo de memória comparável ao CUDA. Work-items a executar a kernel tem acesso a 4 regiões de memória distintas:

- **Global Memory**. Esta região de memória permite leituras/escritas a todos os work-items em todos os work-groups. Os work-items podem ler/escrever para qualquer elemento desta região de memória. Leituras e escritas podem ficar em cache dependendo das capacidades do dispositivo.
- **Constant Memory**: Uma região em global memory que permanece constante durante a execução da kernel. O host aloca e inicializa objectos de memória que são colocados nesta região.

- **Local Memory (em CUDA, shared memory)**: Região de memória partilhada pelo work-group. Esta memória pode ser usada para alocar variáveis que vão ser partilhadas pelos work-items no work-group.
- **Private Memory (em CUDA, registers)**: Região de memória privada e somente acessível ao work-item. Variáveis definidas nesta região de memória por um work-item não são visíveis a outro work-item.

Neste exempo, `device_buffer` é definido na memória global (`global int * device_buffer`).

1.2. Código que executa no CPU (host code): Para além do código que qualquer programa requer, temos também as configurações necessárias para executar o programa na GPU, nomeadamente a definição de todos os objectos OpenCL, a alocação/transferência de dados na GPU e lançamento das funções que vão executar nela.

1.2.1. Obter as plataformas OpenCL disponíveis na máquina:

```
----------------------------------------------------------------------
cl_platform_id platform;
cl_error=clGetPlatformIDs(1, &platform, NULL);
----------------------------------------------------------------------
```

1.2.2. Identificar os dispositivos associados com a plataforma OpenCL:

```
----------------------------------------------------------------------
cl_device_id device_id;
cl_error=clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device_id,
NULL);
----------------------------------------------------------------------
```

1.2.3. Criar o contexto OpenCL associado com o dispositivo:

```
----------------------------------------------------------------------
cl_context context;
context=clCreateContext(NULL, 1, &device_id, NULL, NULL, &cl_error);
----------------------------------------------------------------------
```

1.2.4. Criar a command queue associada com o dispositivo e com o contexto OpenCL:

```
----------------------------------------------------------------------
cl_command_queue command_queue;
command_queue=clCreateCommandQueue(context,       device_id,       0,
&cl_error);
----------------------------------------------------------------------
```

1.2.5. Criar os objectos OpenCL program e respectivas kernels:

```
----------------------------------------------------------------------
cl_program program;
program=clCreateProgramWithSource(context,1,(const
char**)&kernel_string, (const size_t *)&file_size, &cl_error);
cl_error=clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
cl_kernel kernel;
```

```
kernel=clCreateKernel(program, "device_func_name", &cl_error);
```

1.2.6. Alocação de memória na GPU: A função `clCreateBuffer` aloca memória no dispositivo. Na chamada desta função o aluno deve indicar uma FLAG (`CL_MEM_WRITE_ONLY`) que especifica o tipo alocação e utilização do buffer.

```
Cl_mem        device_buffer=clCreateBuffer(context,CL_MEM_WRITE_ONLY,
size_bytes, NULL, &cl_error);
```

1.2.7. Associar os argumentos à kernel OpenCL:

```
cl_error=clSetKernelArg(kernel, 0, sizeof(cl_mem), &device_buffer);
cl_error=clSetKernelArg(kernel, 1, sizeof(int), &N);
```

1.2.8. Lançamento das kernels OpenCL: Aqui o utilizador deve definir o NDRange. Para tal, deve indicar:

1. A command queue onde a kernel vai executar;
2. A kernel;
3. A dimensão do NDRange (1, 2 ou 3);
4. O global work size (dimensão total do problema);
5. O local work size (em CUDA, thread per block);

```
unsigned int work_dim=1;
size_t local_work_size=256;
size_t global_work_size=ceil(N/local_work_size)*local_work_size;
cl_error=clEnqueueNDRangeKernel(command_queue,   kernel,   work_dim,
NULL, &global_work_size, &local_work_size, 0, NULL, NULL);
```

1.2.9. Transferência dos dados da GPU para o host: Depois do programa executar é pertinente recolher o resultado do dispositivo. Para tal, é necessário usar a função `clEnqueueReadBuffer` para transferir os dados da GPU para o host.

```
cl_error=clEnqueueReadBuffer(command_queue,  device_buffer,  CL_TRUE,
0, vec_size_bytes, host_buffer, 0, NULL, NULL);
```

1.2.10. Libertar os recursos requeridos pelo OpenCL:

```
clReleaseMemObject(device_buffer);
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(command_queue);
```

```
clReleaseContext(context);
-------------------------------------------------------------------
```

2. Tendo em conta a análise do ponto anterior, escreva um novo programa, em OpenCL, que implemente a soma de vetores: $v_1 + v_2 = v_3$.
2.1. Implemente uma versão do programa onde os vectores v_1 e v_2 são inicializados no host e transferidos para a GPU.
2.2. Implemente uma versão do programa onde os v_1 e v_2 são inicializados na GPU.

3. O OpenCL permite expressar problemas em três dimensões (x, y, z). Implemente um programa em OpenCL para somar duas matrizes e guardar o resultado numa terceira. Tenha em atenção que deve expressar as duas dimensões deste problema na CUDA kernel (use `get_global_id(1)`).

NOTE 2: O aluno deve consultar a especificação do OpenCL disponibilizada pelo Khronos Group (https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf**).**

NOTA 2: O aluno deve consultar o guia de programação disponibilizado pela NVIDIA para mais informações (https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html).