

Programação Paralela em OpenCL

César L. B. Silveira
cesar@v3d.com.br

19 de novembro de 2010

Introdução

- Open Computing Language (OpenCL)
 - Padrão aberto, livre de *royalties*, para programação paralela em ambientes heterogêneos
 - Ambientes heterogêneos
 - CPUs *multicore*, GPUs many-core
 - *Framework*
 - Arquitetura
 - Linguagem
 - API

Introdução

- Neste tutorial...
 - Conceitos da arquitetura OpenCL
 - Breve visão da linguagem OpenCL C
 - APIs fundamentais para o desenvolvimento
 - Exemplo completo de aplicação
 - Integração com OpenGL

Hello World

- Código sequencial

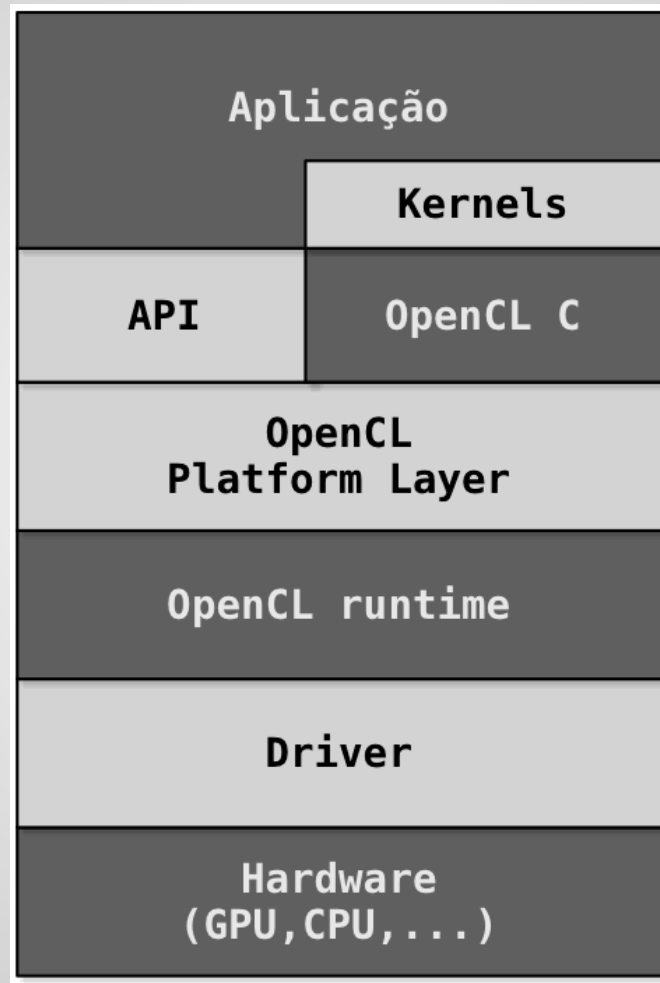
```
void ArrayDiff(const int* a,
               const int* b,
               int* c,
               int n)
{
    for (int i = 0; i < n; ++i)
    {
        c[i] = a[i] - b[i];
    }
}
```

Hello World

- Código OpenCL (*kernel*)

```
__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];  
}
```

Aplicações OpenCL



Camadas de uma aplicação OpenCL

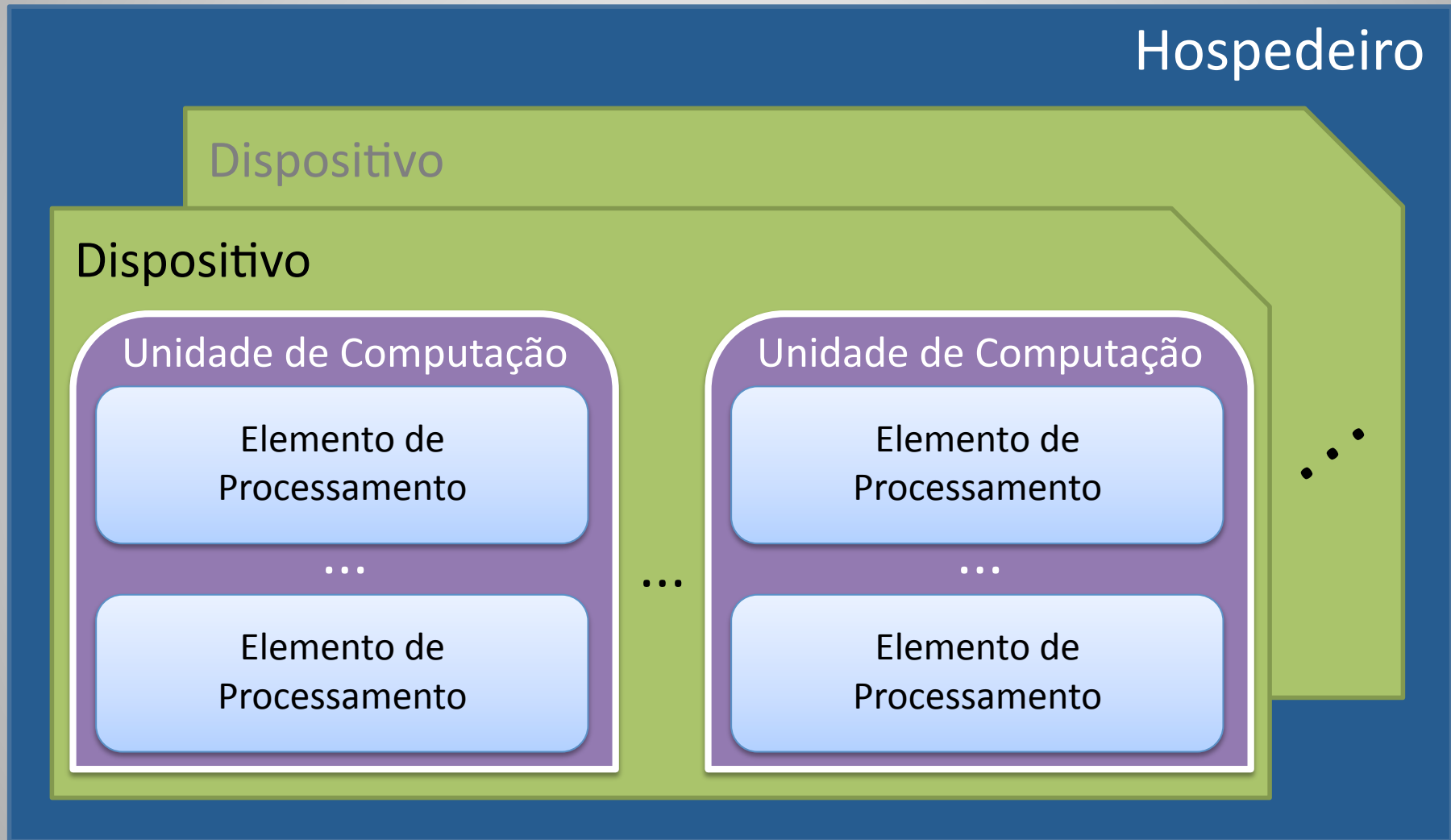
Arquitetura OpenCL

- Arquitetura abstrata de baixo nível
- Implementações mapeiam para entidades físicas
- Quatro modelos
 - Plataforma
 - Execução
 - Programação
 - Memória

Arquitetura OpenCL

- Modelo de plataforma
 - Descreve entidades do ambiente OpenCL

Arquitetura OpenCL



Modelo de Plataforma

Arquitetura OpenCL

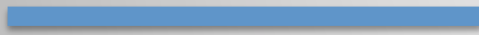
- Exemplo: GPU NVIDIA
 - GPU: dispositivo
 - *Streaming Multiprocessor*: unidade de computação
 - *Processor/Core*: elemento de processamento

Arquitetura OpenCL

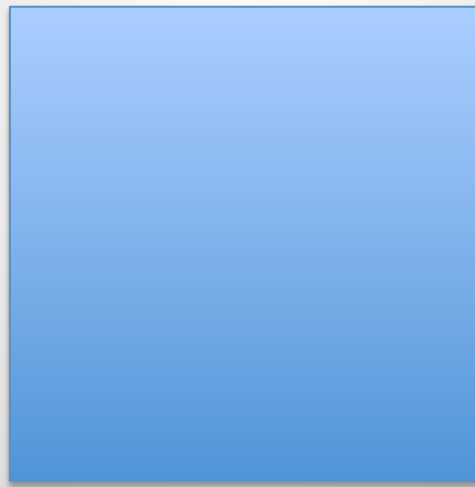
- Modelo de execução
 - Descreve instanciação e identificação de *kernels*

Arquitetura OpenCL

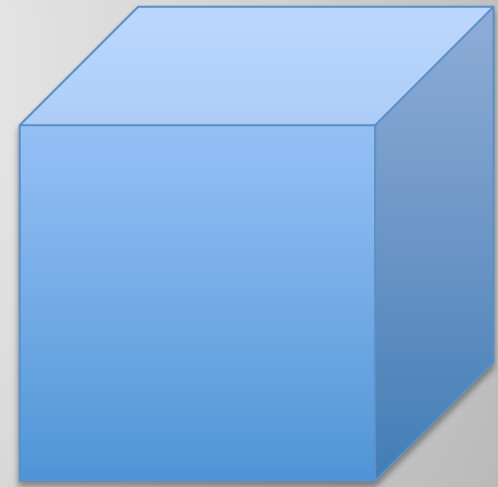
- Modelo de execução
 - Espaços de índices



1D

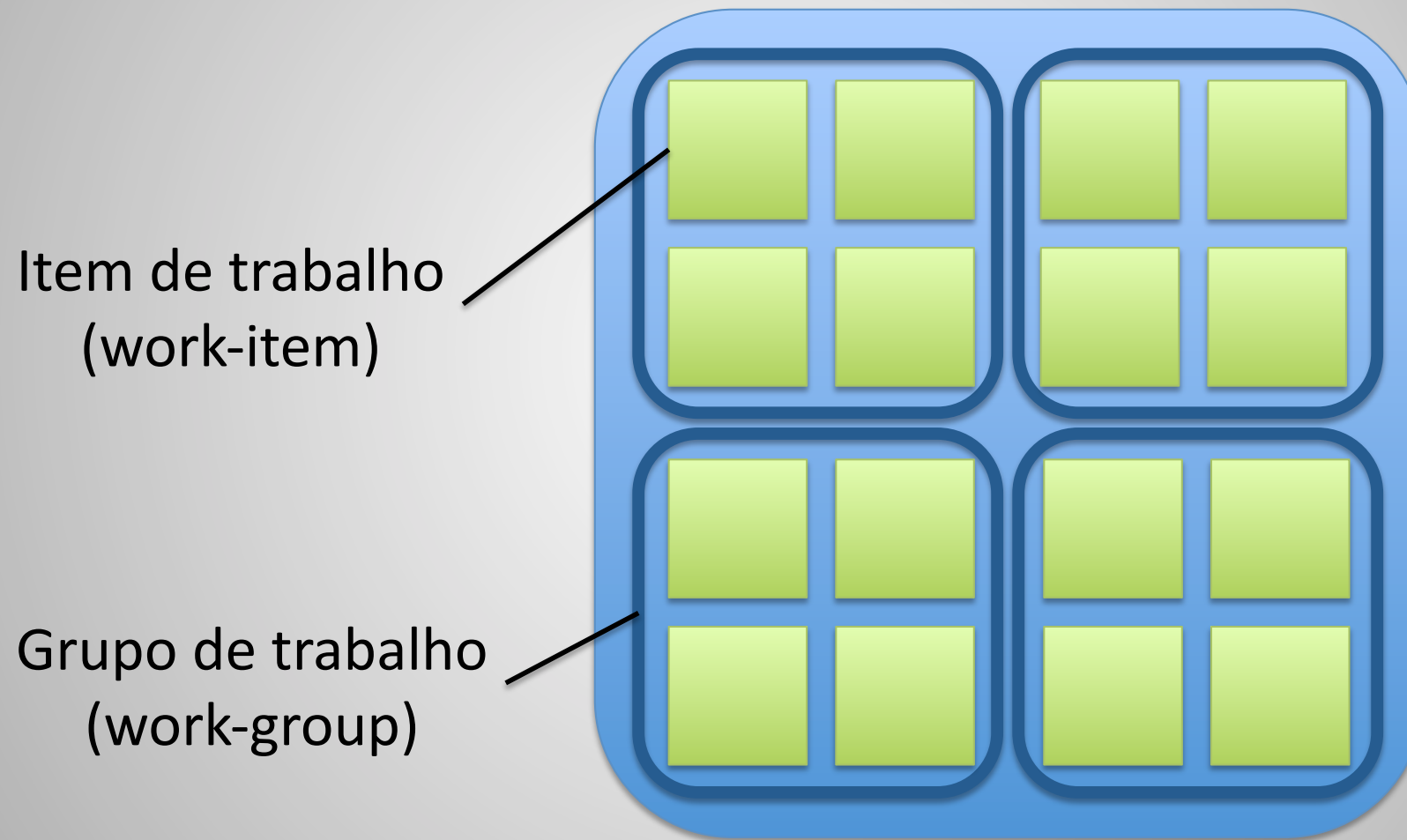


2D



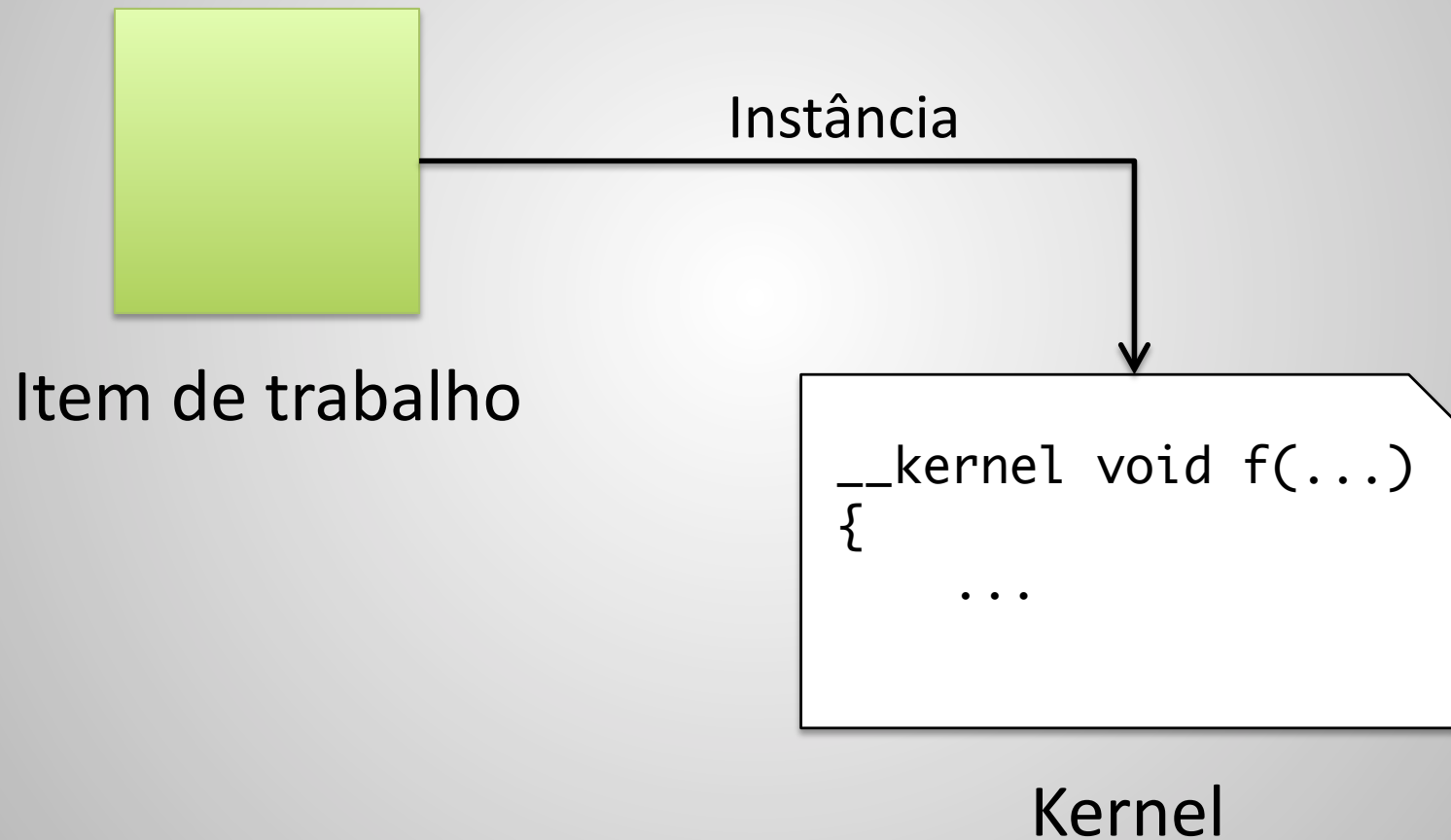
3D

Arquitetura OpenCL

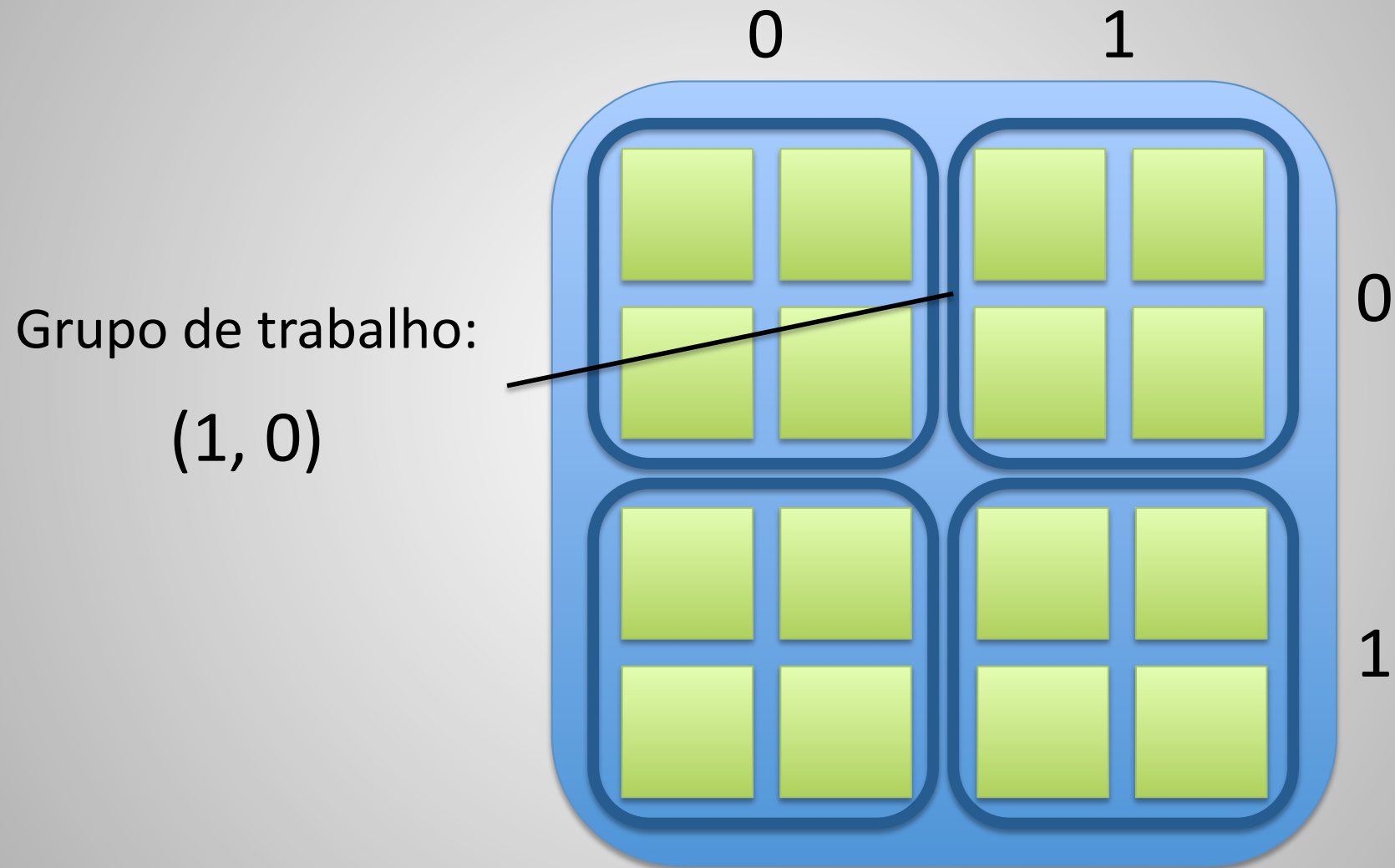


Modelo de execução: itens e grupos de trabalho

Arquitetura OpenCL

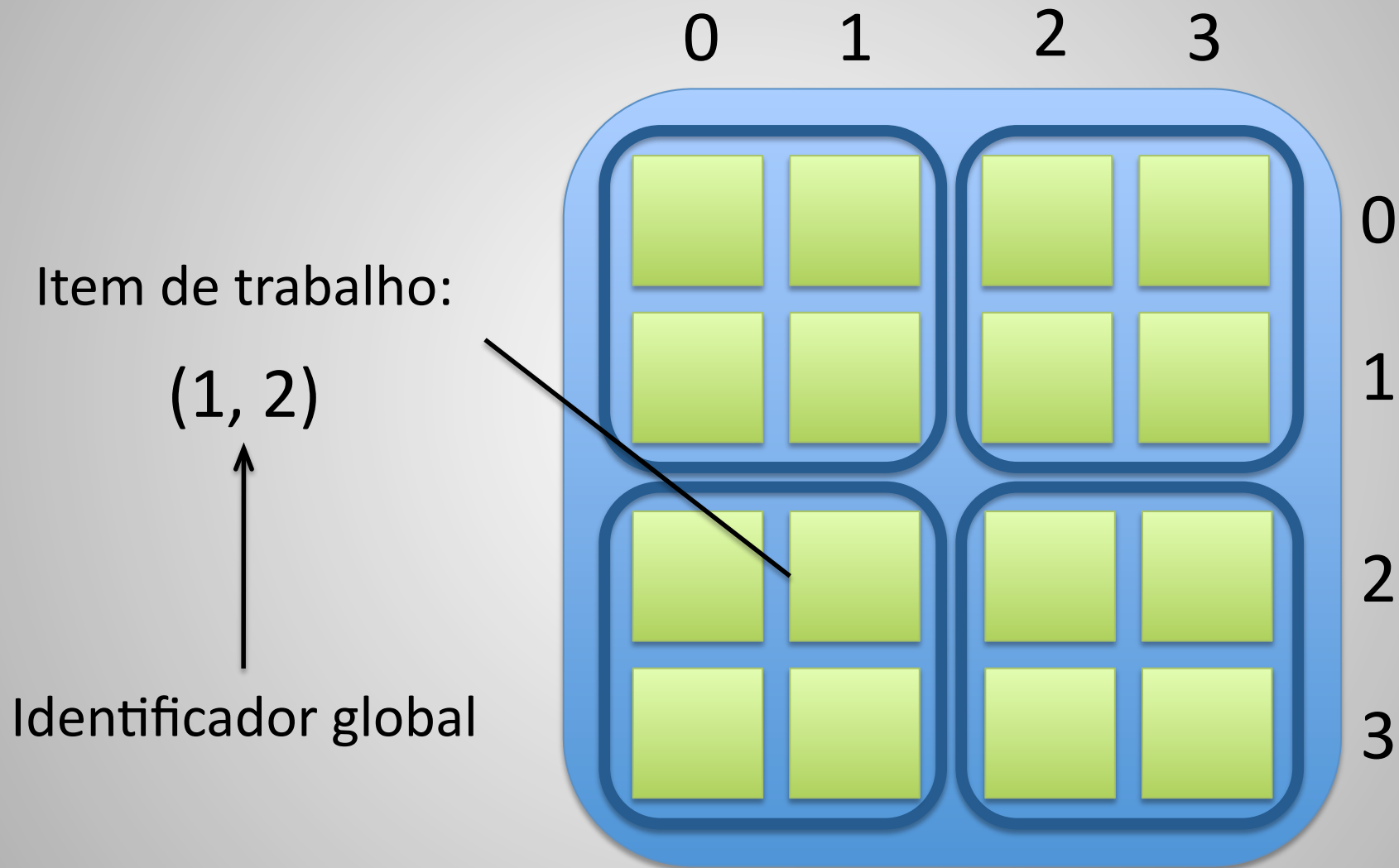


Arquitetura OpenCL



Modelo de execução: identificadores

Arquitetura OpenCL



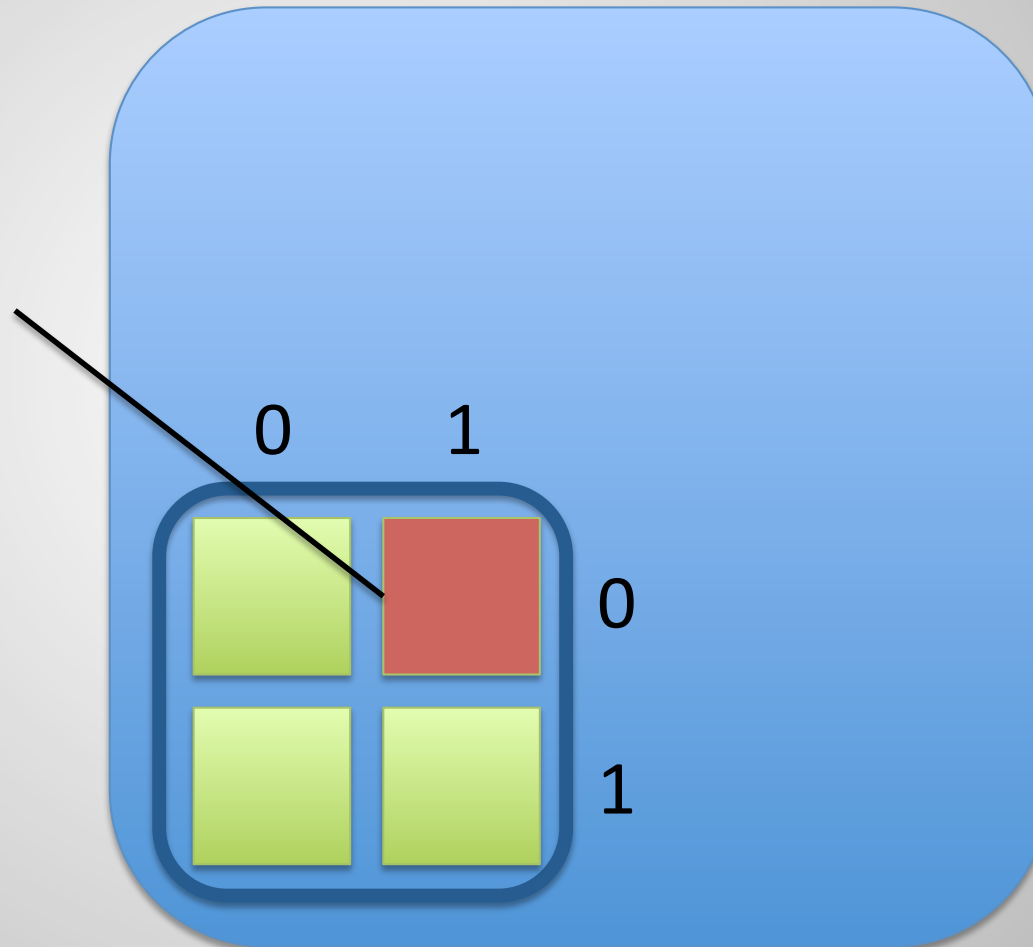
Modelo de execução: identificadores

Arquitetura OpenCL

Item de trabalho:

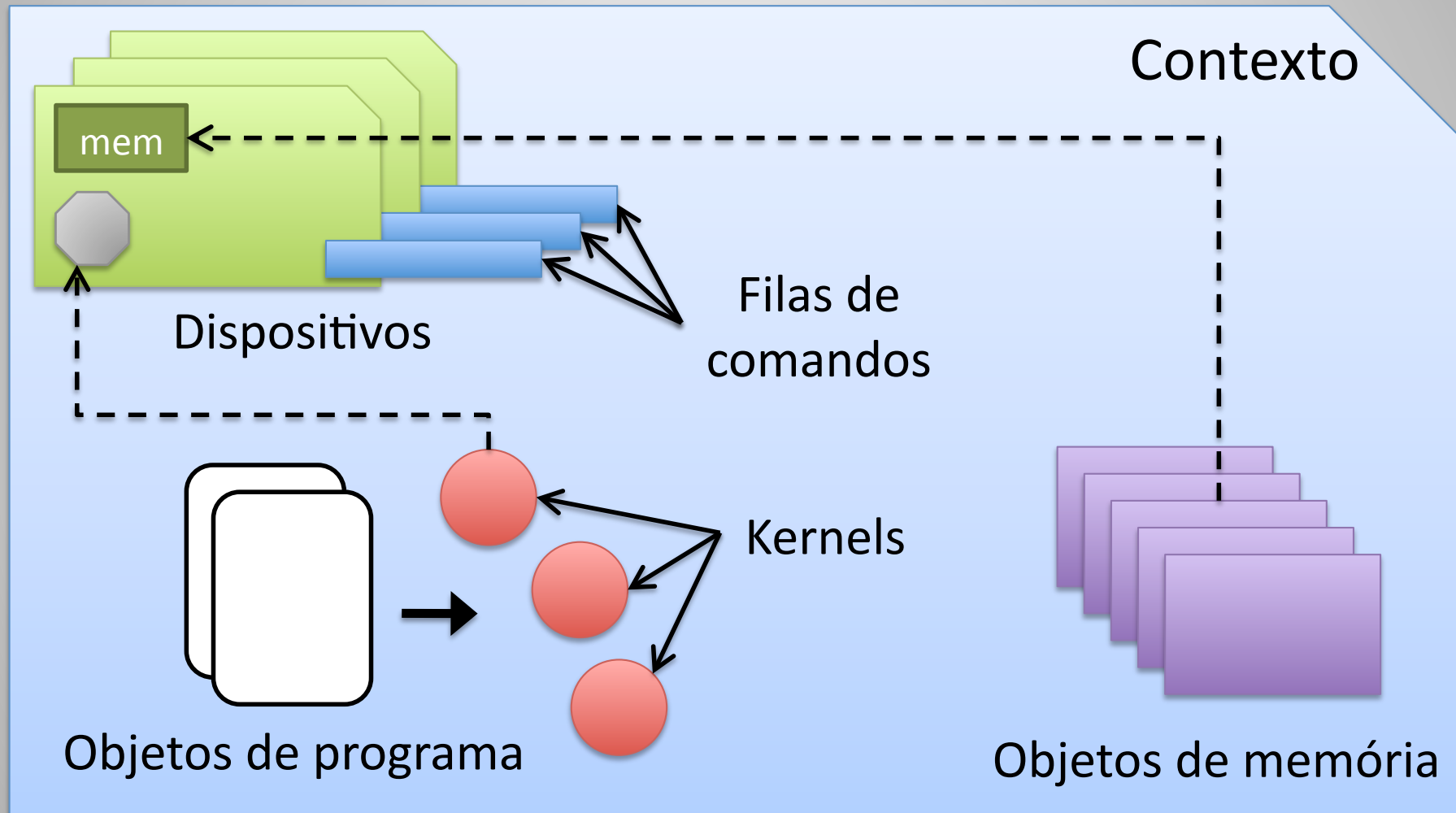
$(1, 0)$

Identificador local



Modelo de execução: identificadores

Arquitetura OpenCL



Modelo de execução: objetos

Arquitetura OpenCL

- Modelo de execução: objetos
 - Toda comunicação com um dispositivo é feita através da sua fila de comandos
 - Objetos de memória
 - *Buffers*: acesso direto via ponteiros
 - *Images*: acesso especial via samplers

Arquitetura OpenCL

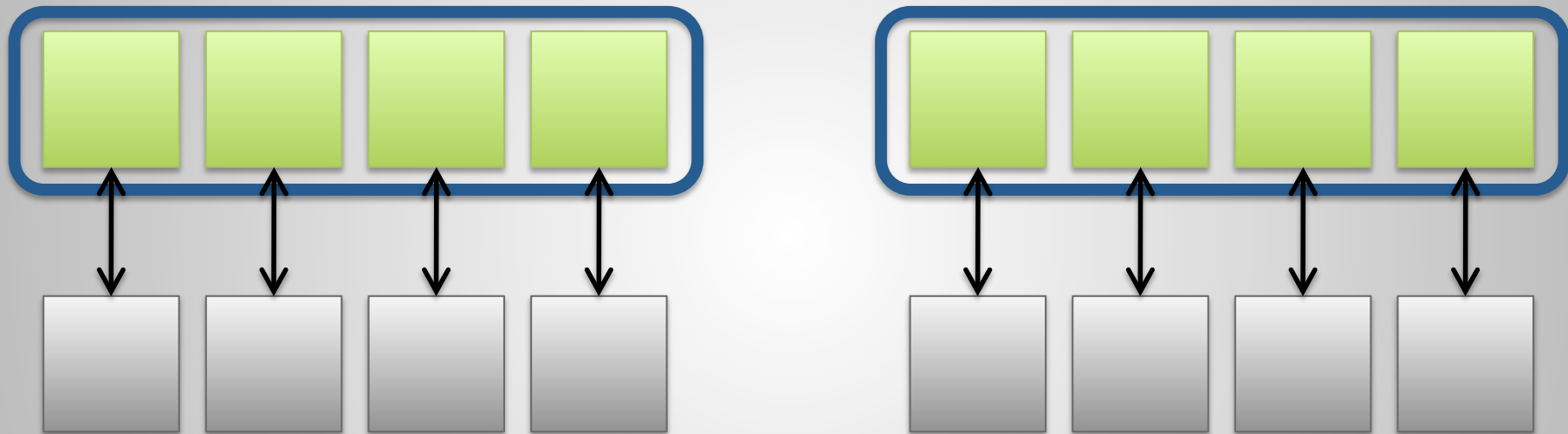
- Modelo de programação
 - **Paralelismo de dados** (*data parallel*): múltiplos itens de trabalho para um *kernel*
 - **Paralelismo de tarefas** (*task parallel*): um item de trabalho para um *kernel*

Arquitetura OpenCL

- Modelo de memória
 - Descreve uma hierarquia de memória e o compartilhamento de cada nível entre itens e grupos de trabalho

Arquitetura OpenCL

Memória privada

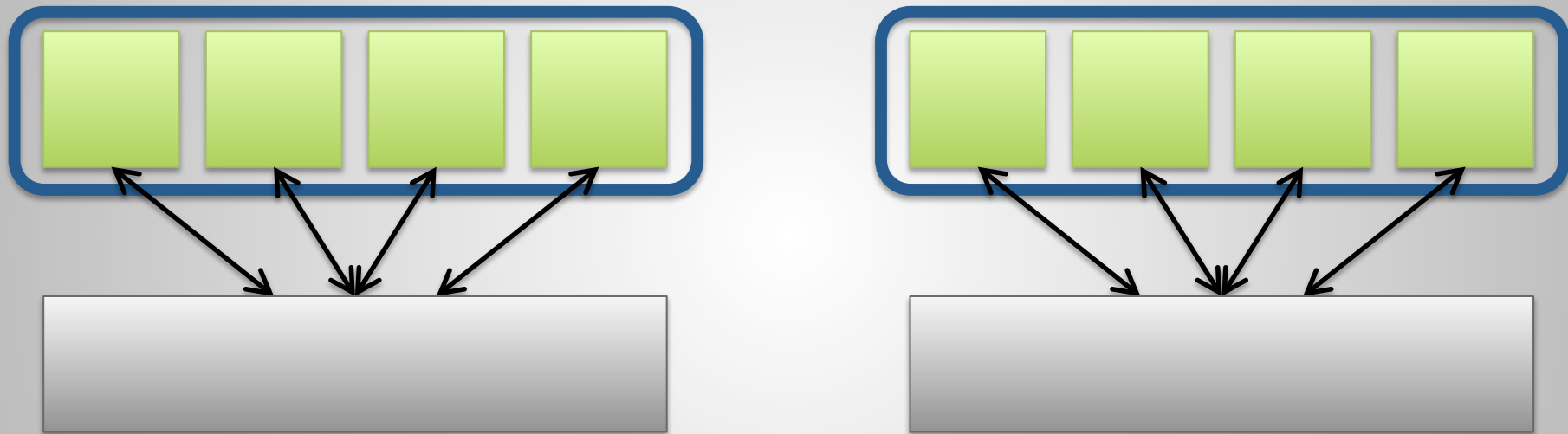


- Exclusiva de cada item de trabalho
- Leitura e Escrita

Modelo de memória

Arquitetura OpenCL

Memória local

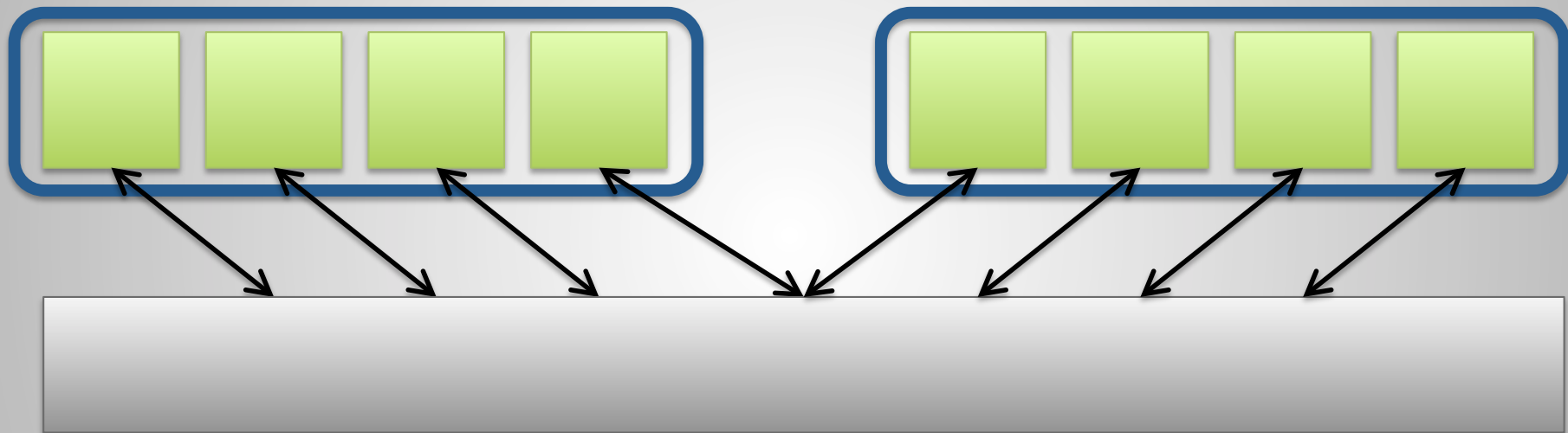


- Compartilhada pelo grupo de trabalho
- Leitura e Escrita

Modelo de memória

Arquitetura OpenCL

Memória global

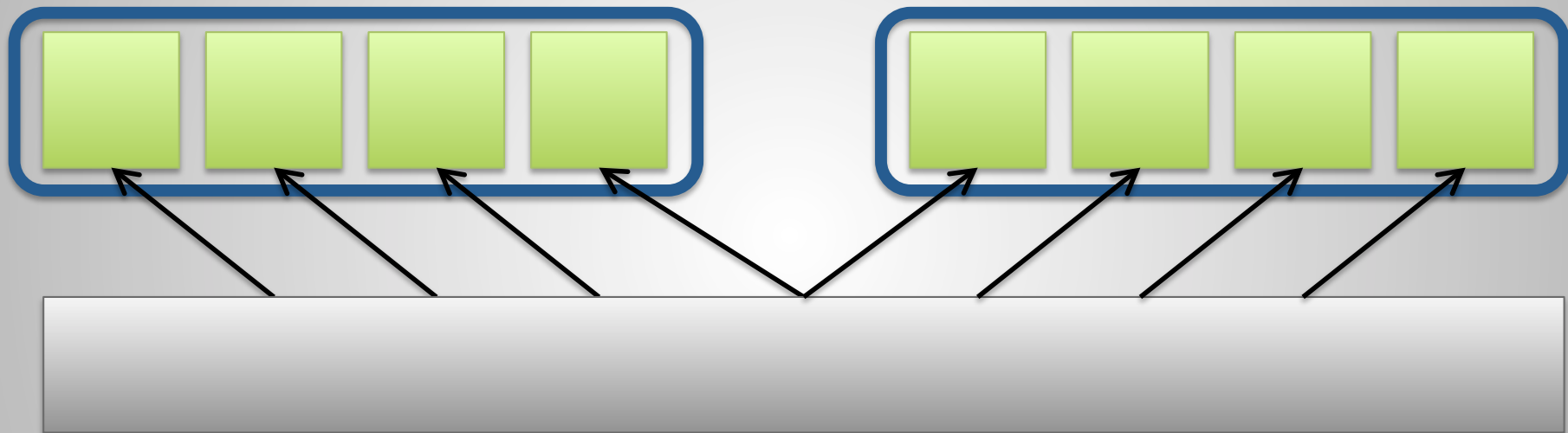


- Compartilhada por todos os itens de trabalho
- Leitura e Escrita

Modelo de memória

Arquitetura OpenCL

Memória constante



- Compartilhada por todos os itens de trabalho
- Somente leitura

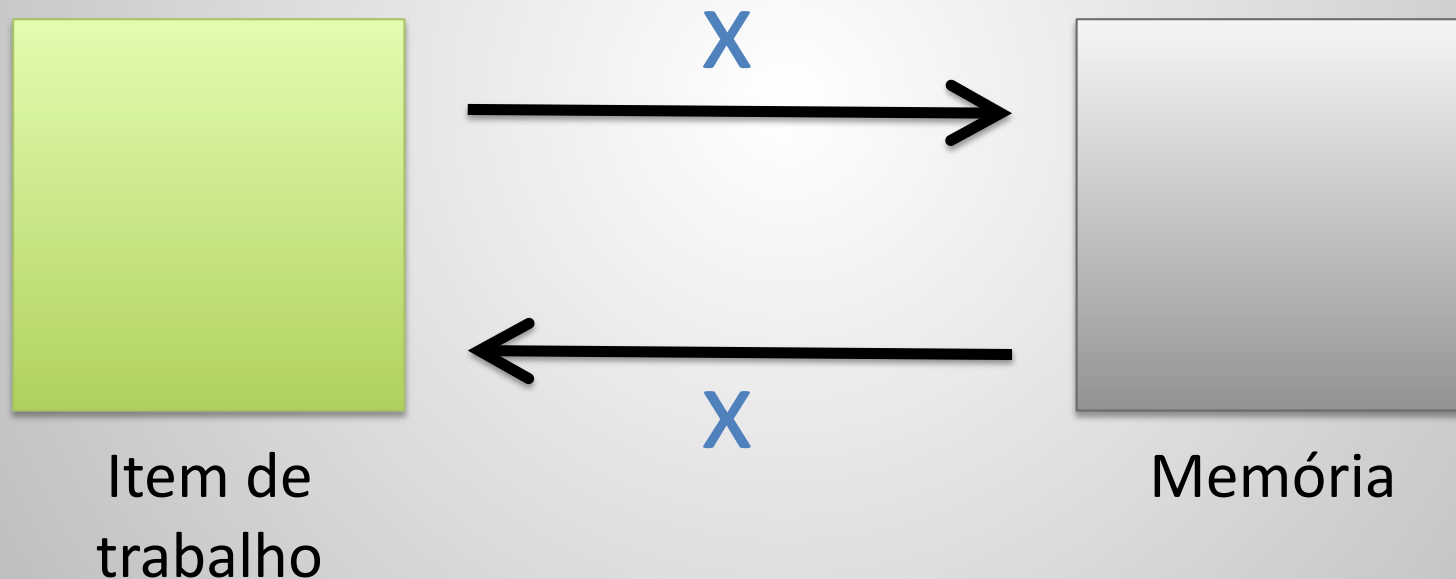
Modelo de memória

Arquitetura OpenCL

- Modelo de memória: consistência
 - Um item de trabalho lê corretamente o que outros escrevem?

Arquitetura OpenCL

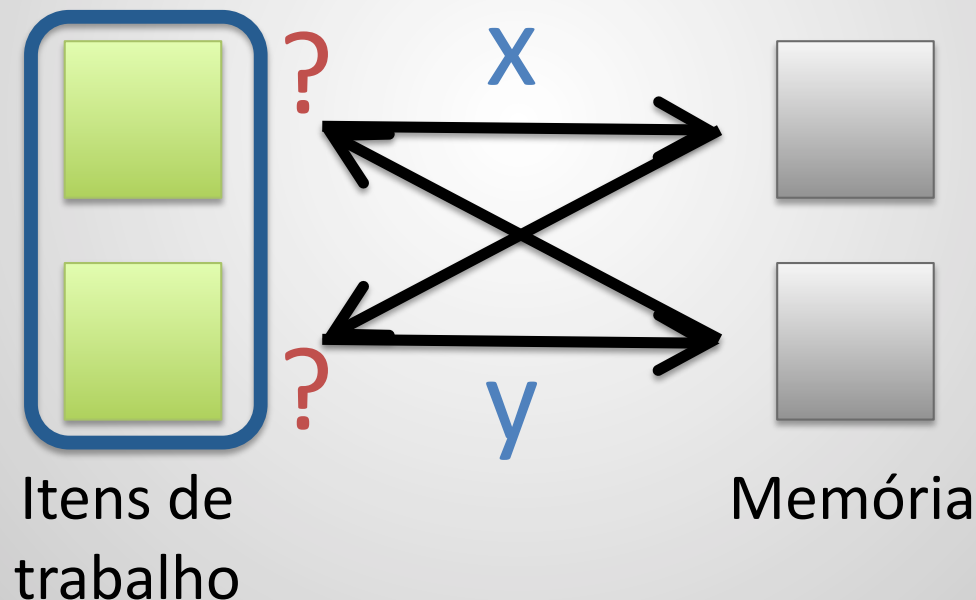
- Memória é consistente para um item de trabalho



Modelo de memória: consistência

Arquitetura OpenCL

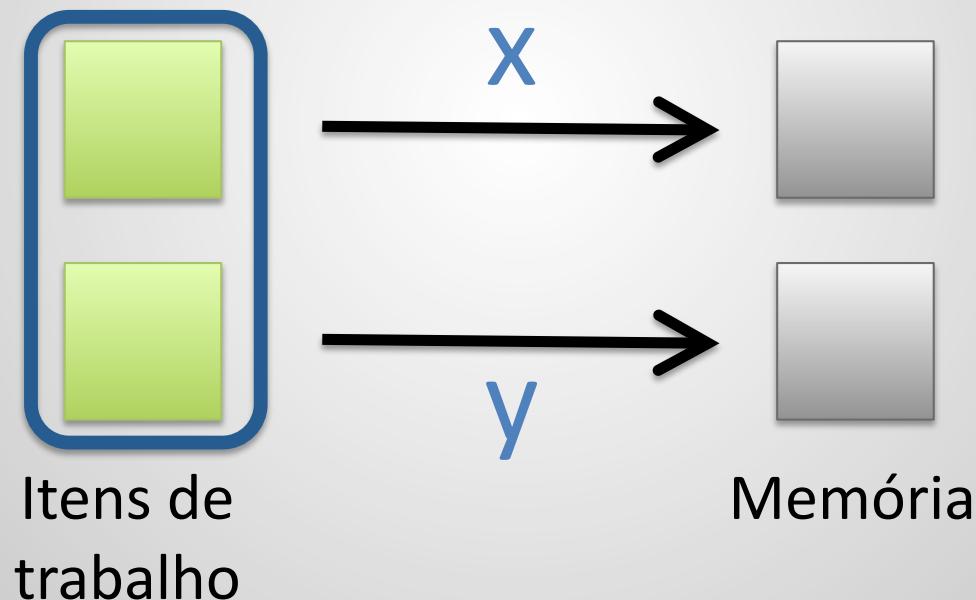
- Memória é consistente para um grupo de trabalho após uma barreira (*barrier*)



Modelo de memória: consistência

Arquitetura OpenCL

- Memória é consistente para um grupo de trabalho **após uma barreira (*barrier*)**



Modelo de memória: consistência

Arquitetura OpenCL

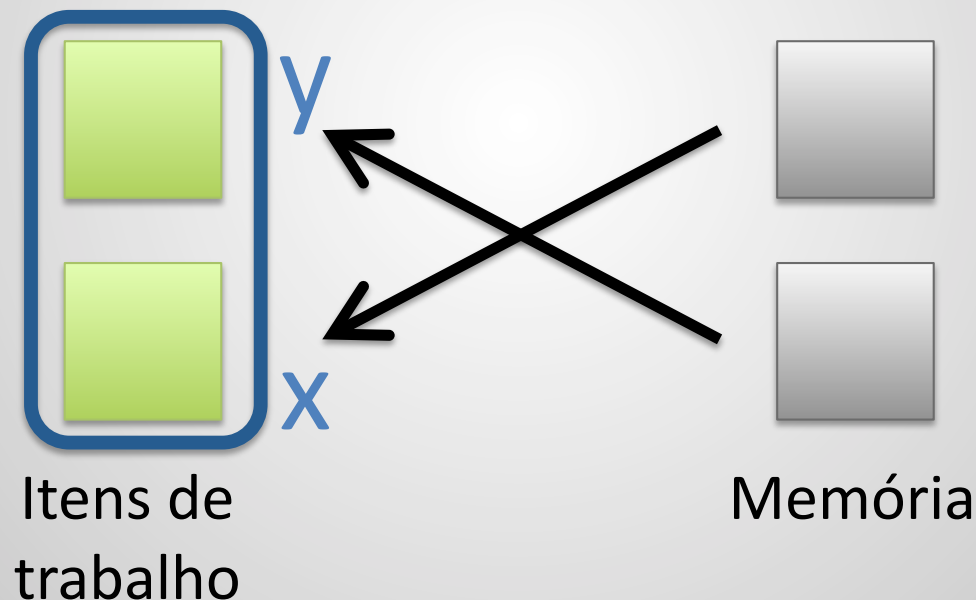
- Memória é consistente para um grupo de trabalho após uma barreira (*barrier*)



Modelo de memória: consistência

Arquitetura OpenCL

- Memória é consistente para um grupo de trabalho após uma barreira (*barrier*)



Modelo de memória: consistência

Arquitetura OpenCL

- Sincronização
 - Barreira: execução só prossegue após todos os itens de trabalho de um mesmo grupo de trabalho a terem atingido
 - **Não há sincronização entre grupos de trabalho**

Linguagem OpenCL C

- Utilizada para a escrita de *kernels*
- Baseada no padrão C99
- Acrescenta extensões e restrições

Linguagem OpenCL C

- Extensão: tipos vetoriais
 - Notação: `tipo[# de componentes]`
 - 1, 2, 4, 8 ou 16 componentes
 - Exemplos:

`float4`

`int8`

`short2`

`uchar16`

Linguagem OpenCL C

- Operações com tipos vetoriais
 - Entre vetores de mesmo número de componentes
 - Entre vetores e escalares

```
float4 v = (float4)(1.0, 2.0, 3.0, 4.0);  
float4 u = (float4)(1.0, 1.0, 1.0, 1.0);  
float4 v2 = v * 2;  
float4 t = v + u;
```

Linguagem OpenCL C

- Definição de *kernels*
 - Qualificador `__kernel`
 - Todo *kernel* deve ser `void`

```
__kernel void f(...)  
{  
    ...  
}
```

Linguagem OpenCL C

- Qualificadores de espaço de endereçamento
 - Definem nível da memória apontada por um ponteiro
 - `__global`, `__local`, `__private` e `__const`
 - *Default*: `__private`

Linguagem OpenCL C

- Restrições
 - Ponteiros para função não são suportados
 - Funções e macros com número variável de argumentos não são suportados
 - Qualificadores `extern`, `static` e `auto` não são suportados
 - Não há suporte a recursão

API de suporte: *kernels*

- Funções de identificação
 - Informações sobre espaço de índices, item e grupo de trabalho

```
get_global_id(uint dimindx)  
get_local_id(uint dimindx)  
get_group_id(uint dimindx)  
get_global_size(uint dimindx)  
get_local_size(uint dimindx)  
get_num_groups()  
get_work_dim()
```

API de suporte: *kernels*

- Funções de sincronização

`barrier(cl_mem_fence_flags flags)`

Flags:

`CLK_LOCAL_MEM_FENCE`

`CLK_GLOBAL_MEM_FENCE`

API de suporte: hospedeiro

- Ilustração através de um exemplo prático
- Aplicação completa
 - *Kernel* para subtração dos elementos de dois *arrays*

Exemplo prático

- Exemplo de código executado no hospedeiro
 - Ilustra as principais etapas para o desenvolvimento de soluções OpenCL
 - Multi-plataforma

Exemplo prático

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

Exemplo prático

```
cl_platform_id platformId;  
cl_device_id deviceId;
```

```
clGetPlatformIDs(1, &platformId, NULL);
```

```
clGetDeviceIDs(  
    platformId,  
    CL_DEVICE_TYPE_GPU, 1, &deviceId,  
    NULL);
```

Exemplo prático

```
cl_context context =  
    clCreateContext(  
        NULL, 1, &deviceId,  
        NULL, NULL, NULL);
```

```
cl_command_queue queue =  
    clCreateCommandQueue(  
        context, deviceId,  
        NULL, NULL);
```

Exemplo prático

```
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]; \  
    }";
```

Exemplo prático

```
cl_program program =  
    clCreateProgramWithSource(  
        context,  
        1, &source,  
        NULL, NULL);
```

Exemplo prático

```
clBuildProgram(  
    program,  
    0, NULL, NULL, NULL, NULL);
```

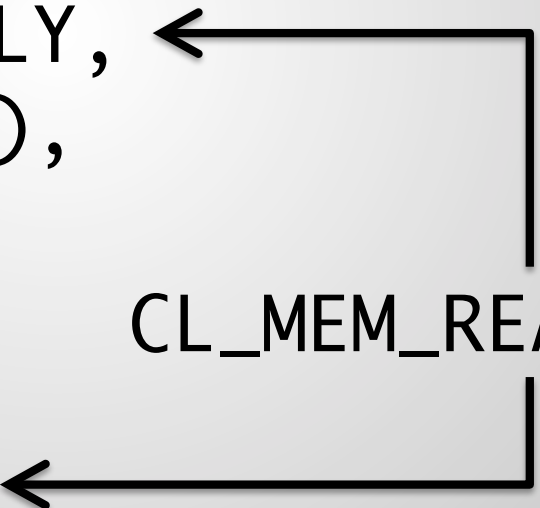
```
cl_kernel kernel =  
    clCreateKernel(  
        program, "ArrayDiff",  
        NULL);
```


Exemplo prático

```
#define N ...
```

```
cl_mem bufA = clCreateBuffer(  
    context,  
    CL_MEM_READ_ONLY, ←  
    N * sizeof(int),  
    NULL, NULL);
```

```
cl_mem bufB = ...;  
cl_mem bufC = ...; ← CL_MEM_READ_WRITE
```



Exemplo prático

```
int* hostA;
```

```
clEnqueueWriteBuffer(  
    queue,  
    bufA,  
    CL_TRUE,  
    0,  
    N * sizeof(int),  
    hostA,  
    0, NULL, NULL);
```

Exemplo prático

```
int* hostB;
```

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

Exemplo prático

```
clSetKernelArg(  
    kernel, 0,  
    sizeof(cl_mem), &bufA);  
clSetKernelArg(  
    kernel, 1,  
    sizeof(cl_mem), &bufB);  
clSetKernelArg(  
    kernel, 2,  
    sizeof(cl_mem), &bufC);
```

Exemplo prático

```
const size_t globalSize[] =  
    { N };
```

```
clEnqueueNDRangeKernel(  
    queue, kernel,  
    1, NULL, globalSize, NULL,  
    0, NULL, NULL);
```

```
clFinish(queue);
```

Exemplo prático

```
int* hostC;
```

```
clEnqueueReadBuffer(  
    queue,  
    bufC,  
    CL_TRUE,  
    0,  
    N * sizeof(int),  
    hostC,  
    0, NULL, NULL);
```

Exemplo prático

```
clReleaseMemObject(bufC);  
clReleaseMemObject(bufB);  
clReleaseMemObject(bufA);  
clReleaseKernel(kernel);  
clReleaseProgram(program);  
clReleaseCommandQueue(queue);  
clReleaseContext(context);
```

Interoperação com OpenGL

- Em GPUs, é possível compartilhar estruturas entre OpenCL e OpenGL
- Exemplo prático: malha tridimensional
 - Vértices posicionados via OpenCL
 - Exibição via OpenGL
 - Compartilhamento de Vertex Buffer Object (VBO)
 - OpenCL: *array* de float4

Interoperação com OpenGL

- Demonstração

Interoperação com OpenGL

```
__kernel void sine_wave(  
    __global float4* pos,  
    unsigned int width,  
    unsigned int height,  
    float time)  
{  
    unsigned int x = get_global_id(0);  
    unsigned int y = get_global_id(1);
```

Kernel

Interoperação com OpenGL

```
float u = x / (float) width;
```

```
float v = y / (float) height;
```

```
u = u*2.0f - 1.0f;
```

```
v = v*2.0f - 1.0f;
```

```
float freq = 4.0f;
```

```
float w = sin(u * freq + time) *  
          cos(v * freq + time) * 0.5f;
```

```
pos[y * width + x] =  
    (float4)(u, w, v, 1.0f);
```

```
}
```

```
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  
};
```

Criação do contexto

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

Interoperação com OpenGL

```
#define BUFFER_SIZE ...
```

```
GLuint vbo;
```

Criação do VBO

```
glGenBuffers(1, &vbo);  
glBindBuffer(GL_ARRAY_BUFFER, vbo);  
glBufferData(  
    GL_ARRAY_BUFFER, BUFFER_SIZE, 0,  
    GL_DYNAMIC_DRAW);
```

Interoperação com OpenGL

```
cl_mem vbo_cl;
```

```
vbo_cl = clCreateFromGLBuffer(  
    context,  
    CL_MEM_WRITE_ONLY,  
    vbo,  
    NULL);
```

Criação do objeto de
memória a partir do
VBO

Interoperação com OpenGL

```
clEnqueueAcquireGLObjects(  
    queue,  
    1,  
    &vbo_cl,  
    NULL,  
    NULL,  
    NULL);
```

Aquisição do VBO

Interoperação com OpenGL

```
size_t globalSize[] =  
    { MESH_WIDTH, MESH_HEIGHT };
```

```
clEnqueueNDRangeKernel(  
    queue, kernel, 2,  
    NULL, globalSize, NULL,  
    0, 0, 0);
```

```
clFinish(queue);
```

Execução do *kernel*

Interoperação com OpenGL

```
clEnqueueReleaseGLObjects(  
    queue,  
    1,  
    &vbo_cl,  
    NULL,  
    NULL,  
    NULL);
```

Liberação do VBO

Considerações finais

- SDKs
 - NVIDIA CUDA SDK
 - ATI Streaming SDK
 - Intel OpenCL SDK (*alpha*)
- Aplicações em jogos: IA, física, simulações diversas

Considerações finais

- Material disponível em

<http://labs.v3d.com.br/>

Dúvidas?