

Introdução ao OpenCL

Amanda Sabatini Dufek

amandasd@lncc.br

Agosto/2018

Justificativa

Por que GPU?

Justificativa

Por que GPU?

Por que GPU de um único fabricante?

Justificativa

Por que GPU?

Por que GPU de um único fabricante?

Por que apenas GPU?

Justificativa

Por que GPU?

Por que GPU de um único fabricante?

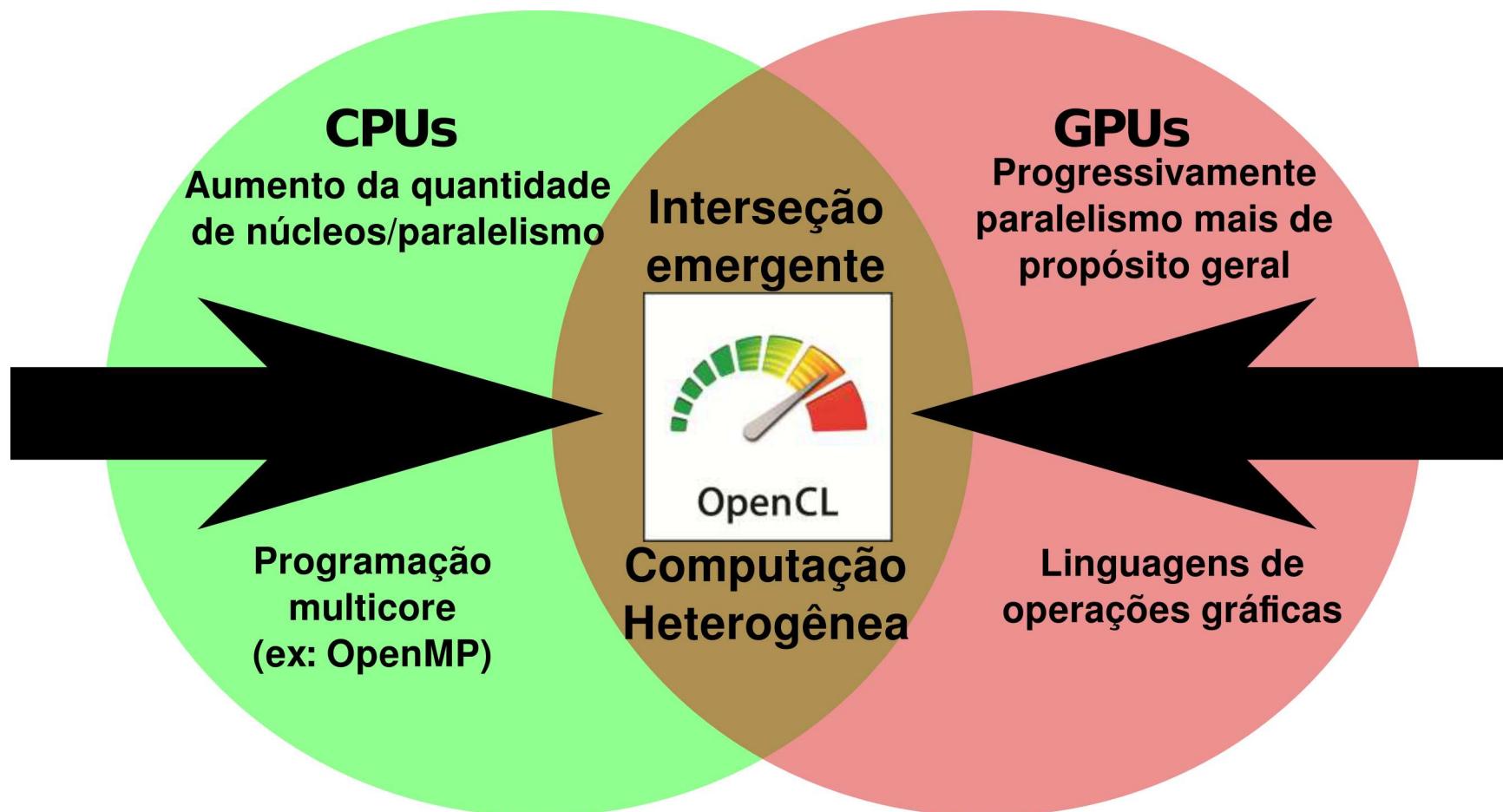
Por que apenas GPU?

Tendência:

- CPUs convencionais + aceleradores (sistema heterogêneo)

OpenCL | Open Computing Language

“Padrão aberto para a programação paralela de sistemas heterogêneos”

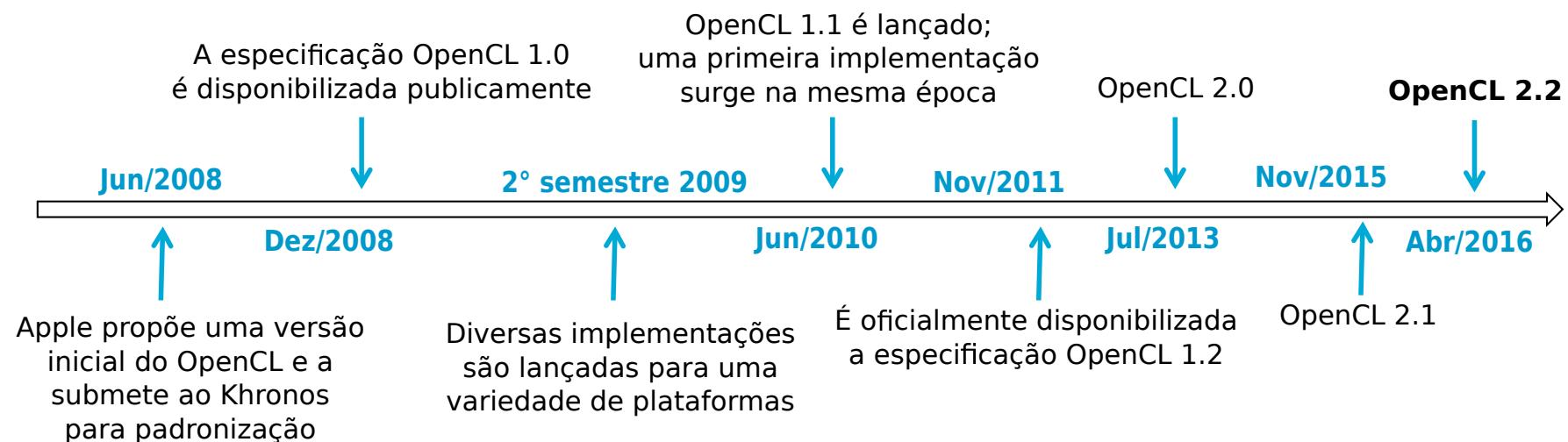


OpenCL | Open Computing Language

- Provê interface *homogênea* para a exploração da computação paralela *heterogênea*
 - abstração do hardware
 - CPUs (AMD, ARM, IBM, Intel), GPUs (AMD, Nvidia, Intel, ARM, Imagination), APU, MIC, FPGAs, Epiphany, DSPs
- Código *portável* entre *arquiteturas e gerações*
- Padrão *aberto*
 - especificação mantida por vários membros
 - gerenciada pelo grupo *Khronos*
- Especificação baseada nas linguagens C e C++

História

- ~2003: GPUs começam a adquirir características de propósito geral: a era da programabilidade
- 2003–2008: Cenário GP-GPU fragmentado, com várias soluções proprietárias e míopes
- 2008: Apple enxerga a oportunidade, intervém e desenvolve uma interface padronizada para computação GP-GPU em diferentes plataformas de hardware



História



Suporte da indústria em 2008

História



© Copyright Khronos Group, 2010 - Page 2

Suporte da indústria em 2010

OpenCL × CUDA

São tecnologias com alta interseção:

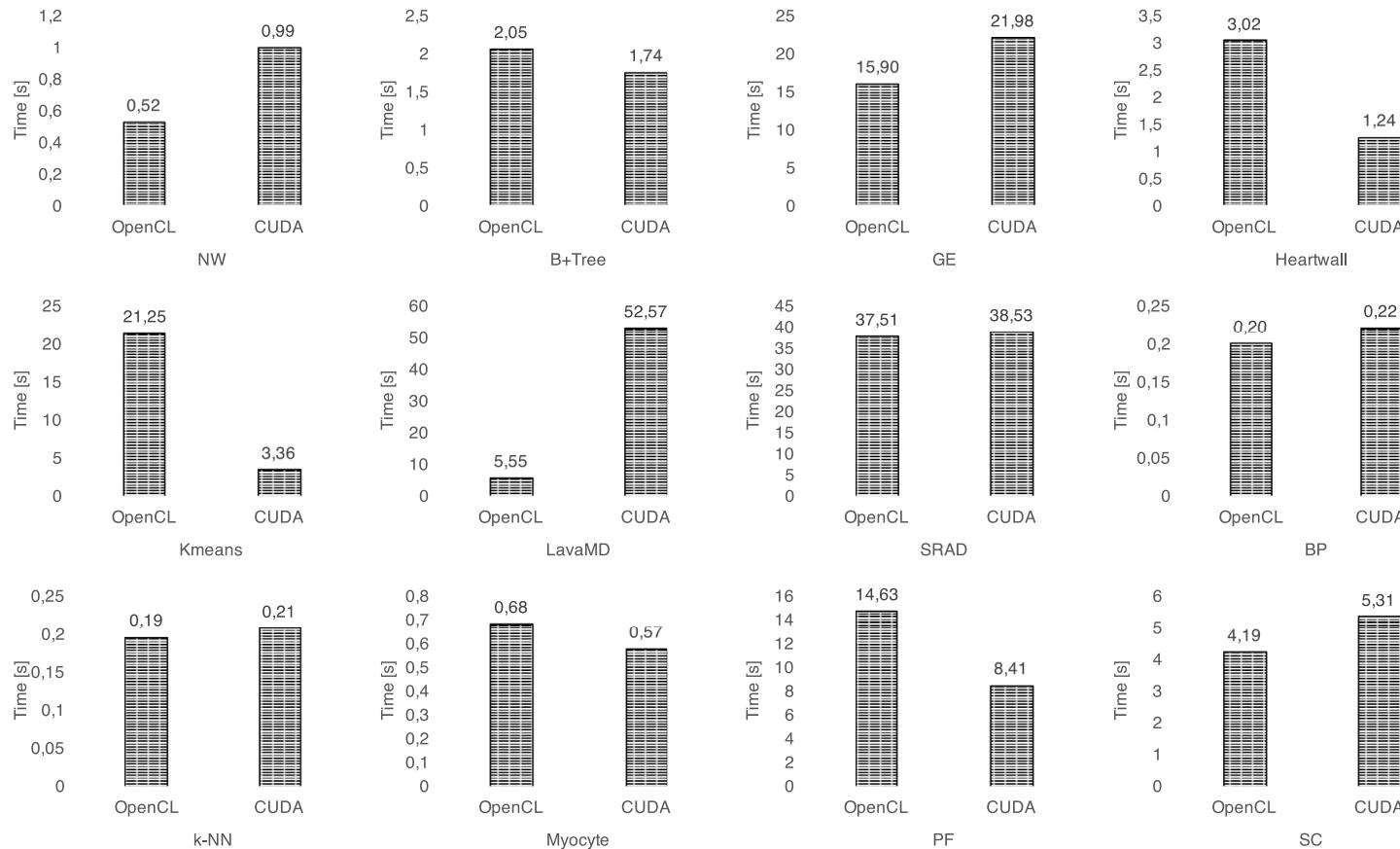
- Propósito parecidos
 - OpenCL foi influenciado por CUDA: ponto inicial
- Nível comparável de complexidade:
 - funcionalidades no que tange às GPUs
 - nível da linguagem
 - custo de engenharia de software
- Comparativamente mesmo desempenho

OpenCL × CUDA

Porém o CUDA:

- É uma tecnologia proprietária da Nvidia
- Não visa a computação heterogênea
- Desenvolvida especificamente para as GPUs Nvidia

OpenCL × CUDA



(a) Time

OpenCL × OpenMP

OpenMP:

- Paralelismo tradicionalmente focado em CPU
- Mais alto nível:
 - programação mais simples, porém limitada/menos flexível
 - ganho de desempenho usualmente sub-ótimo

OpenCL × MPI

São tecnologias ortogonais:

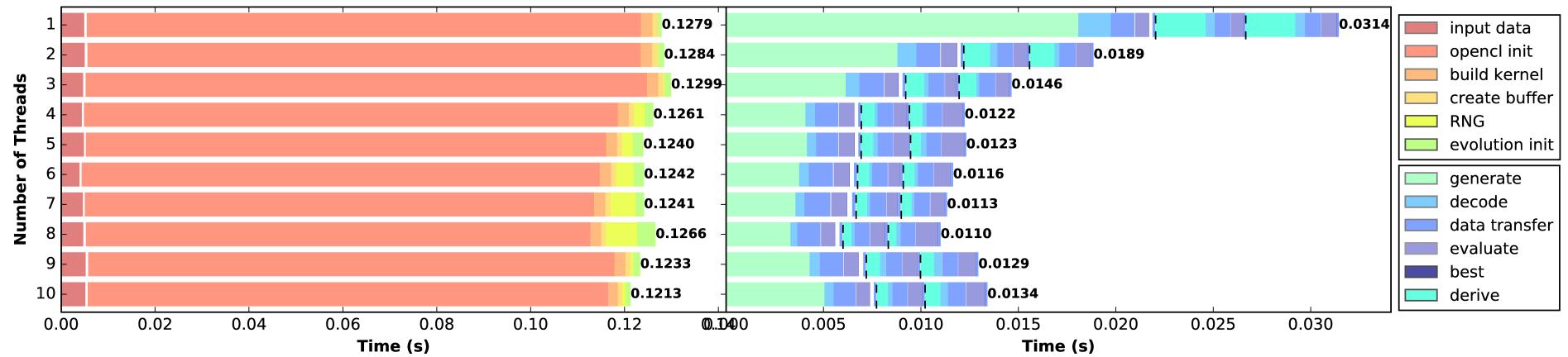
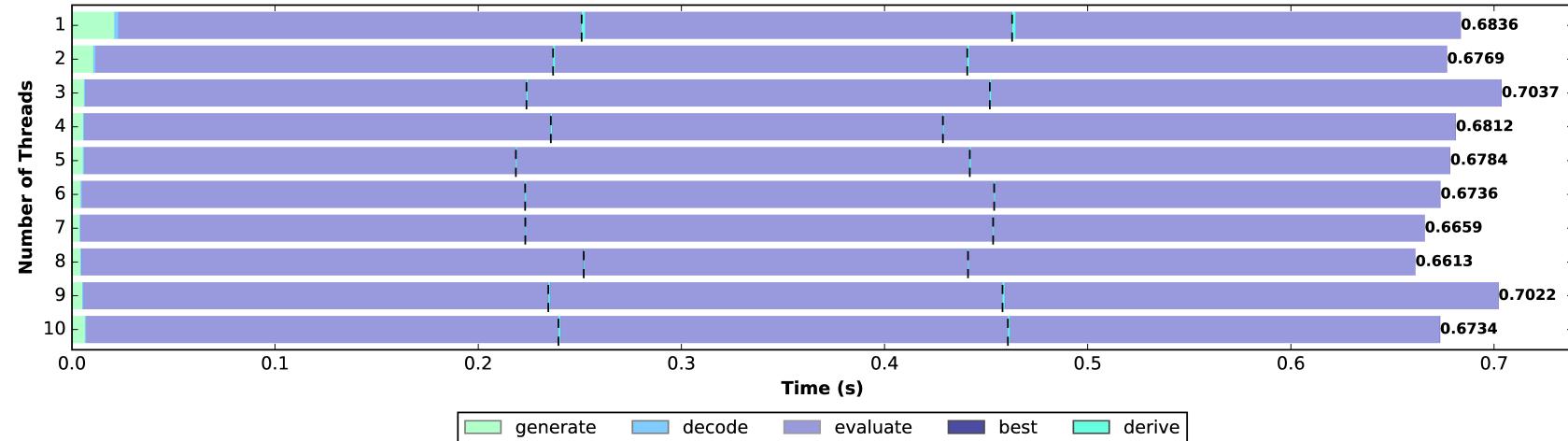
- OpenCL: paralelismo *local*
 - usualmente memória compartilhada
- MPI: paralelismo *distribuído*
 - memória distribuída
- Podem ser combinadas: paralelismo em dois níveis

Possível cenário futuro

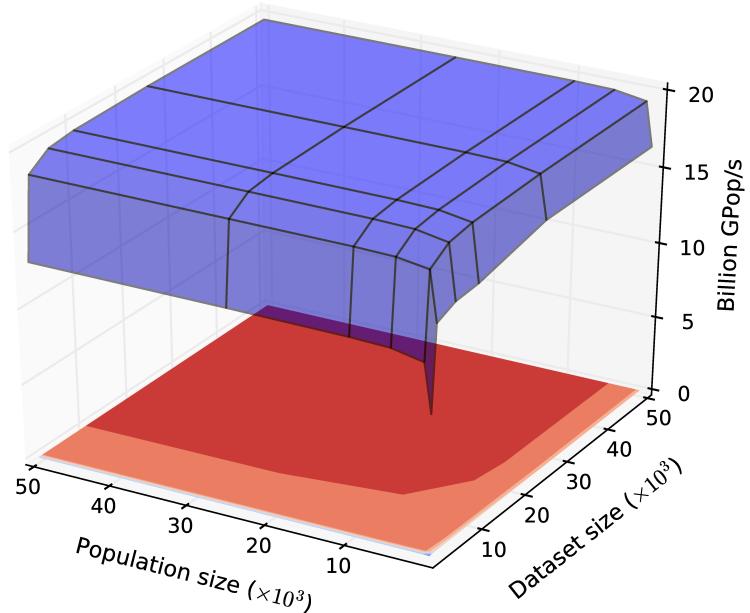
Convergência para as abordagens:

- **MPI**
 - ◆ paralelismo distribuído
- **OpenMP/OpenACC**
 - ◆ paralelismo incremental/fácil
- **OpenCL**
 - ◆ paralelismo massivo heterogêneo

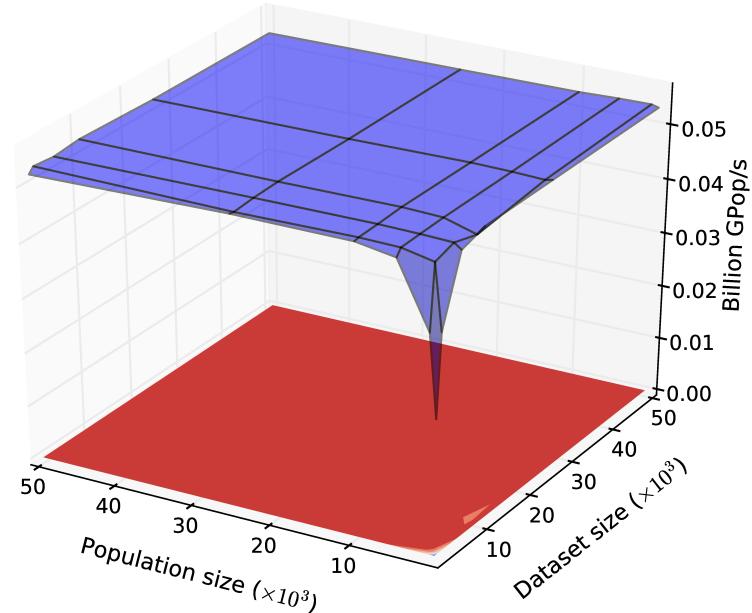
Aplicação



Aplicação



(a) GPU/PDP



(b) CPU sequential mode

Código do Kernel e Hospedeiro

Existem duas hierarquias de códigos no OpenCL:

- O *kernel*:
 - tarefa executada paralelamente em um dispositivo computacional

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

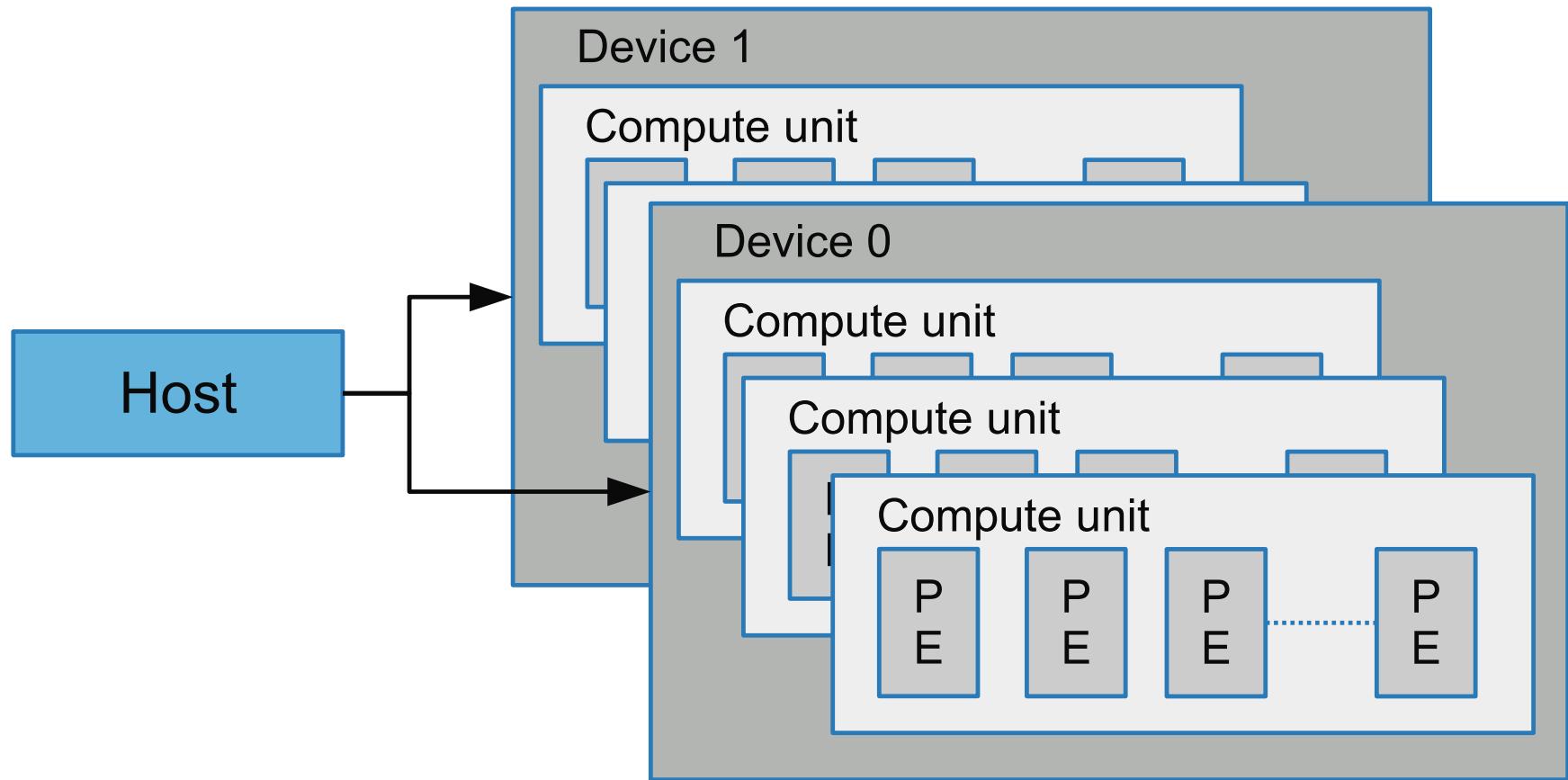
- O código *hospedeiro*:
 - coordena os recursos e ações do OpenCL

Arquitetura do OpenCL

O OpenCL pode ser conceitualmente visto sob quatro ângulos:

- Modelo de *plataforma*
- Modelo de *execução*
- Modelo de *memória*
- Modelo de *programação*

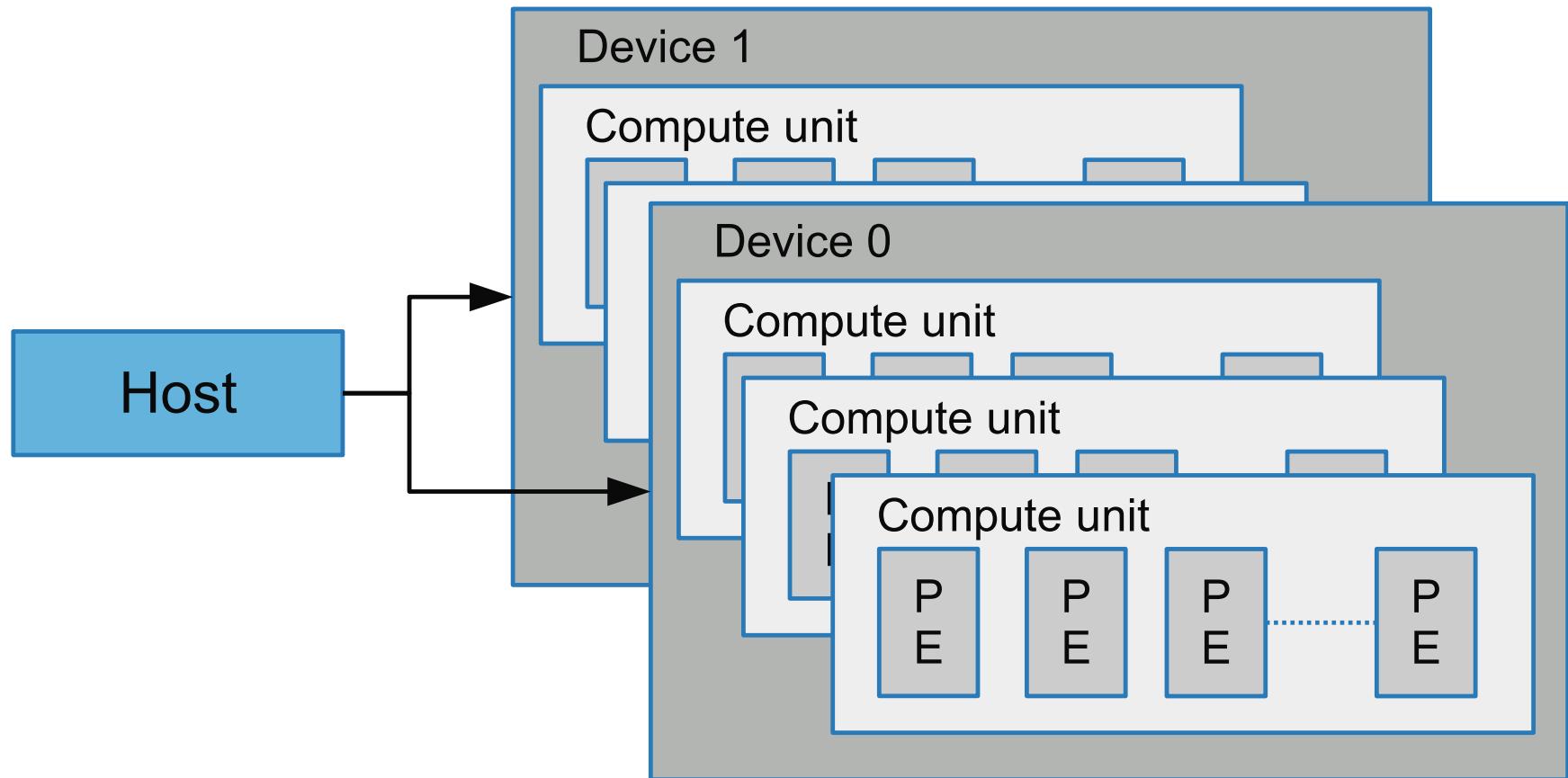
Modelo de Plataforma



Modelo de Execução

- **Item de trabalho (work-item):**
 - Cada item de trabalho é executado por um elemento de processamento.
 - Um elemento de processamento pode executar inúmeros itens de trabalho.
 - O número total de itens de trabalho é chamado de *global size*.

Modelo de Plataforma

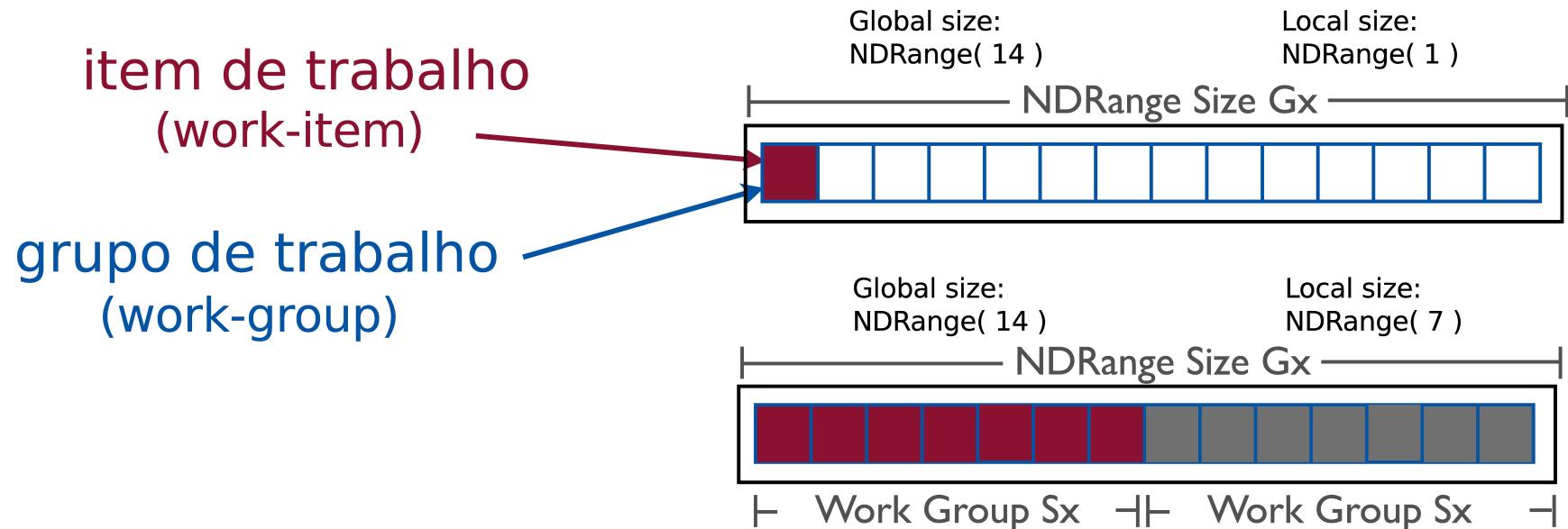


Modelo de Execução

■ Grupo de trabalho (*work-group*):

- Os itens de trabalho podem ser divididos em grupos de trabalho.
- Os itens de trabalho de um mesmo grupo são executados em uma mesma unidade computacional.
- O número de itens de trabalho de um mesmo grupo é chamado de *local size*.
- Itens de trabalho de um mesmo grupo podem se *comunicar eficientemente* e *sincronizar*.
- Diferentes grupos de trabalho são executados *independente*mente.

Modelo de Execução

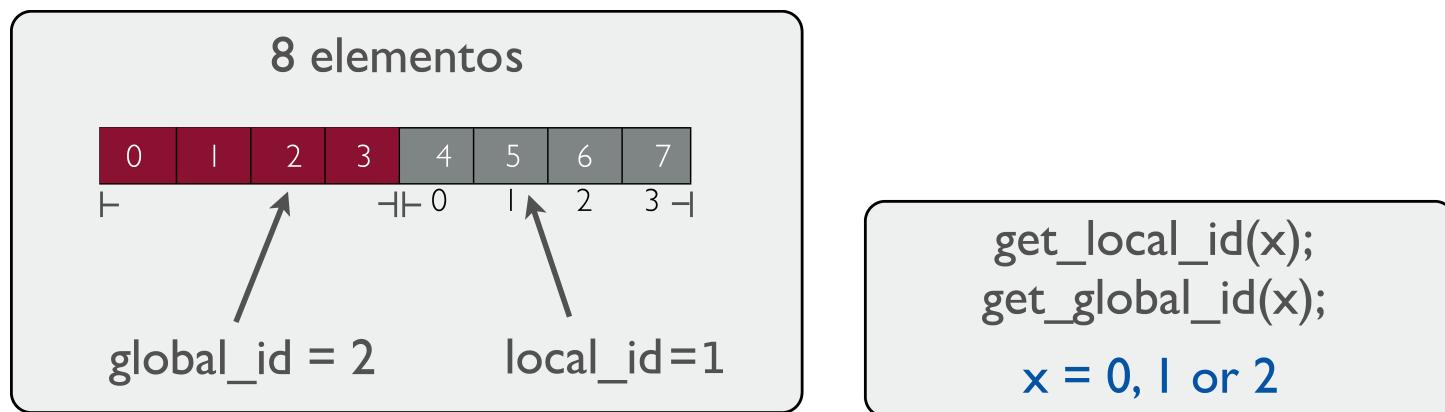


NDRange Size = Global Size
(Tamanho global)

Work Group Size = Local Size
(Tamanho do grupo de trabalho) (Tamanho local)

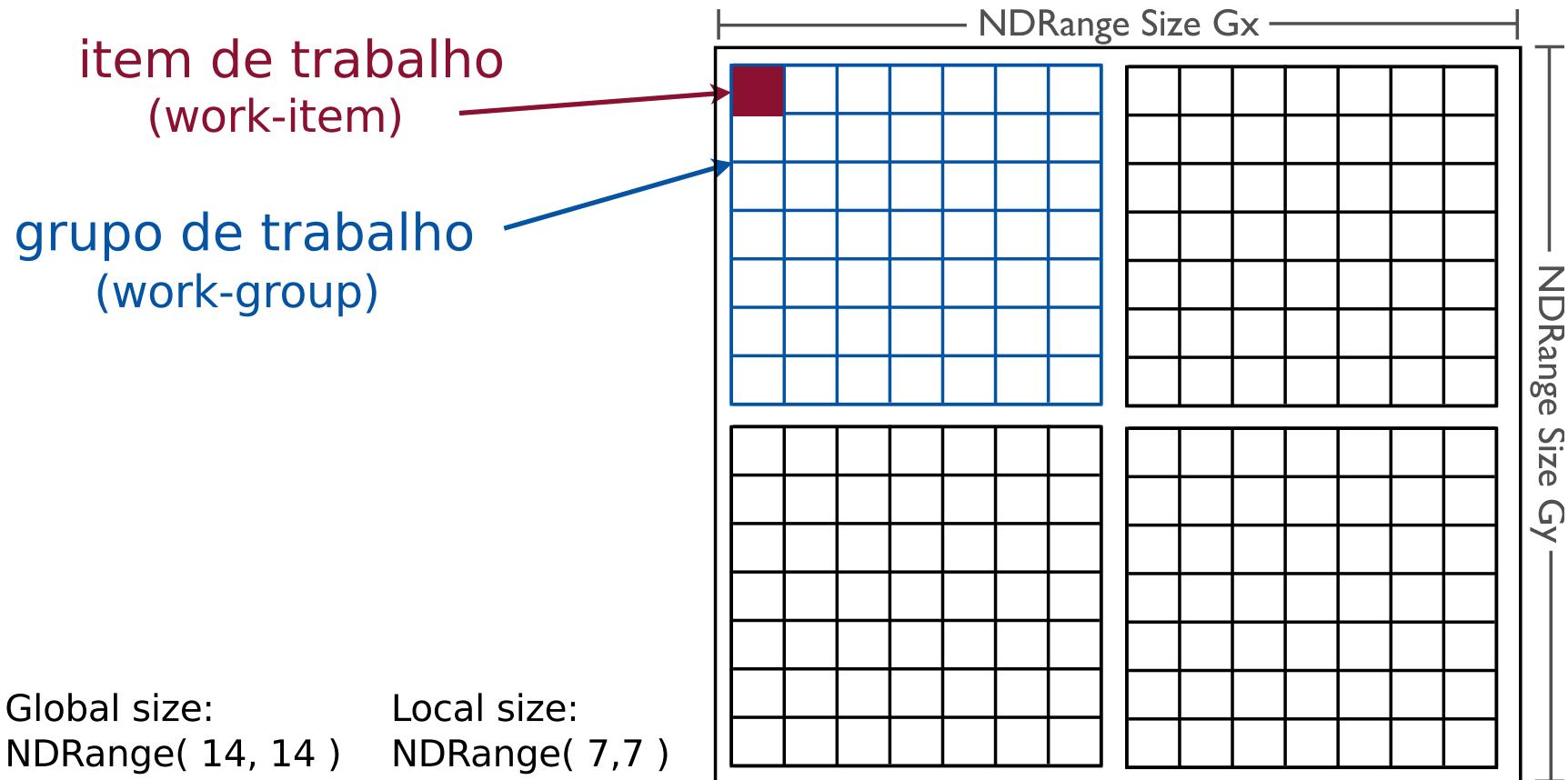
Domínio de índices unidimensional

Modelo de Execução



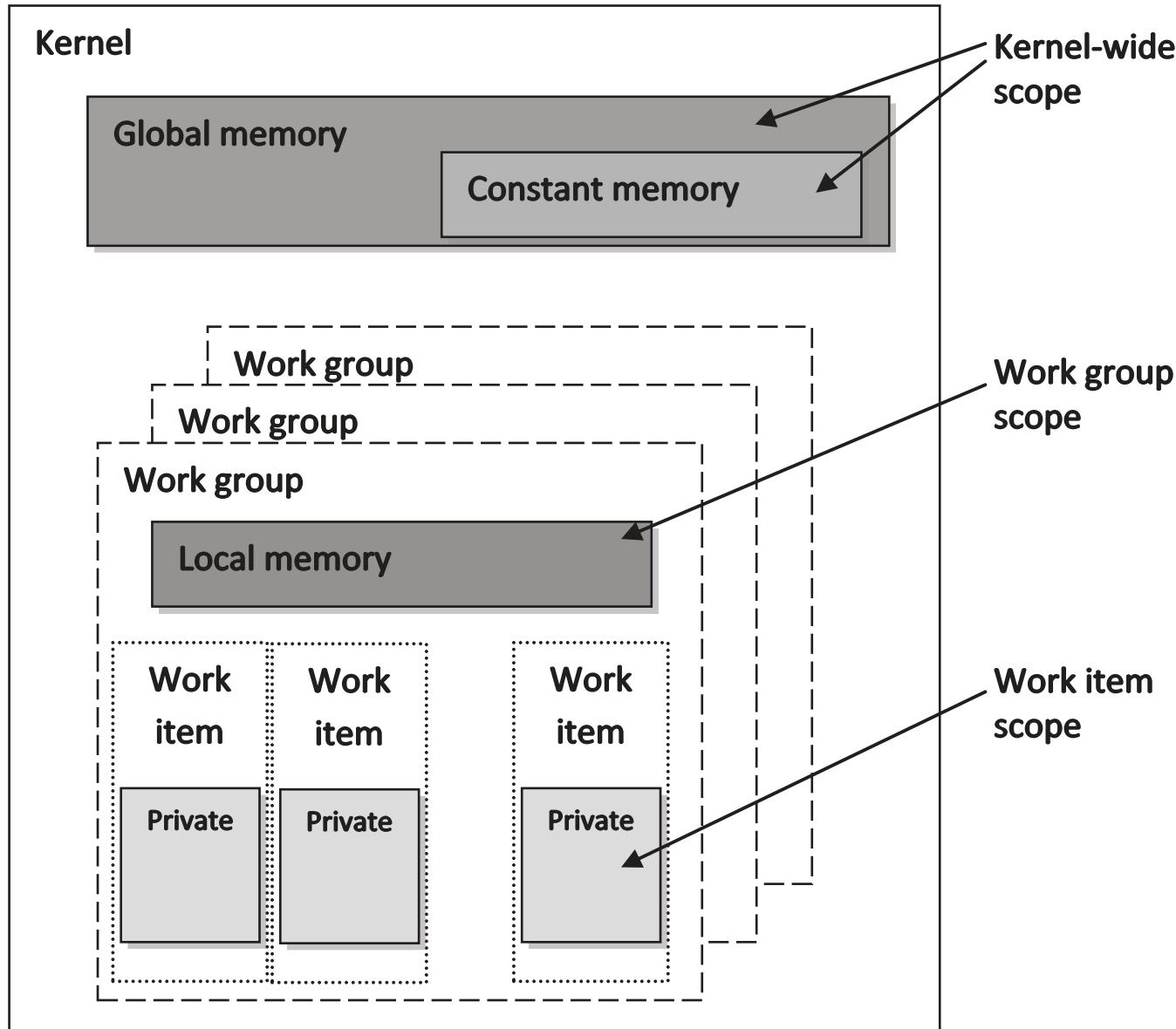
Identificadores

Modelo de Execução

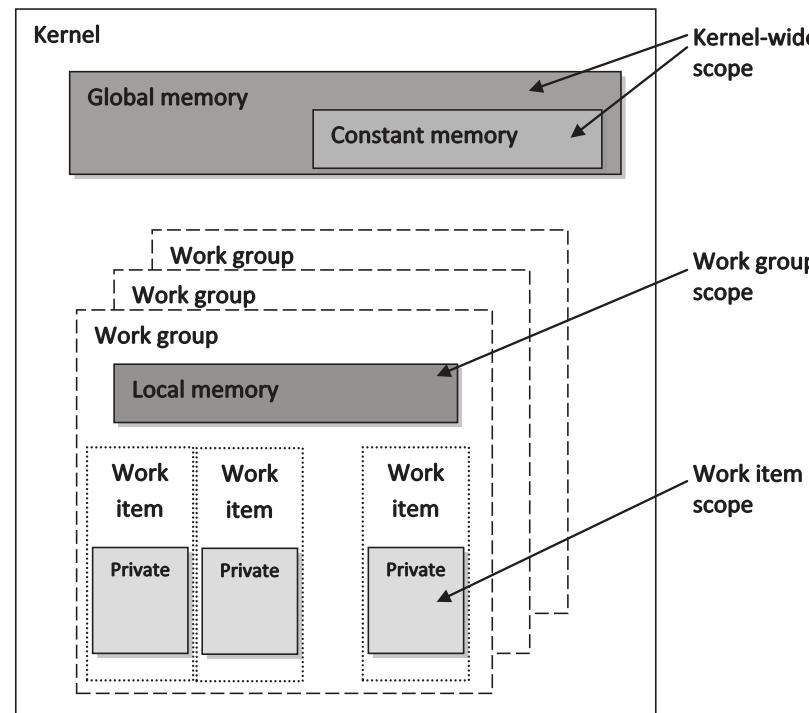


Domínio de índices bidimensional

Modelo de Memória



Modelo de Memória



- *global*: acessível por todos itens de trabalho
- *constant*: acesso global, mas somente leitura
- *local*: somente acessível pelos itens dentro de um mesmo grupo de trabalho
- *private*: somente acessível pelo item de trabalho

Modelo de Memória

- Memória **privada** (*private*):
consistência *garantida*
- Memória **constante** (*constant*):
consistência *garantida*
(não há modificação de conteúdo)
- Memória **local** e **global**:
consistência *relaxada* entre itens de trabalho
requer sincronismo explícito

Modelo de Memória

- *global*: não permitida
- *constant*: a definição deve acompanhar a declaração
- *local*: não pode ser definida na declaração
- *private*: é o escopo padrão

```
kernel void f()
{
    __constant float c = 3.1415; // constante
    __local int loc[16];        // local
    int i;                      // privada
    ...
}
```

Modelo de Plataforma

1. Descobrindo a plataforma e dispositivos

```
vector<cl::Platform> plataformas;  
vector<cl::Device> dispositivos;  
  
cl::Platform::get( &plataformas );// plataformas  
plataformas[0].getDevices( CL_DEVICE_TYPE_ALL, &dispositivos );// dispositivos
```

Plataformas e Dispositivos

| <i>Plataforma</i> | <i>Versão</i> | <i>OpenCL</i> | <i>Dispositivos</i> |
|-------------------|---------------|----------------------|----------------------|
| Intel OpenCL | 16.1.1 | 1.2 (CPU), 2.1 (GPU) | CPUs, Intel GPUs |
| Intel OpenCL | 14.2 | 1.2 | Xeon Phi (KNC) |
| Nvidia OpenCL | CUDA 8 | 1.2 (exp. 2.0) | Nvidia GPU |
| AMD APP | 3.0 | 2.0 (GPU), 1.2 (CPU) | GPU, CPUs |
| PoCL | 0.14 | 2.0 | CPUs, Xeon Phi (KNL) |
| Beignet | 1.3 | 2.0 | GPUs Intel |

- Suporte parcial do OpenCL 2.0 pela Nvidia
- Portabilidade do OpenCL válida para a versão 1.2 (lançada em Nov/2011)
- Intel não suporta mais o Xeon Phi

Plataformas e Dispositivos

| | |
|----------------------------|--|
| Number of platforms: | 1 |
| Platform Profile: | FULL_PROFILE |
| Platform Version: | OpenCL 2.0 AMD-APP (1642.5) |
| Platform Name: | AMD Accelerated Parallel Processing |
| Platform Vendor: | Advanced Micro Devices, Inc. |
| Platform Extensions: | cl_khr_icd cl_amd_event_callback cl_amd_offline_devices |
| Platform Name: | AMD Accelerated Parallel Processing |
| Number of devices: | 2 |
| Device Type: | CL_DEVICE_TYPE_GPU |
| Vendor ID: | 1002h |
| Board name: | AMD Radeon R9 200 Series |
| Device Topology: | PCI[B#1, D#0, F#0] |
| Max compute units: | 40 |
| Max work group size: | 256 |
| Native vector width int: | 1 |
| Max clock frequency: | 1000Mhz |
| Max memory allocation: | 2505572352 |
| Image support: | Yes |
| Max image 3D width: | 2048 |
| Cache line size: | 64 |
| Global memory size: | 3901751296 |
| Platform ID: | 0x7f54fb22cf0 |
| Name: | Hawaii |
| Vendor: | Advanced Micro Devices, Inc. |
| Device OpenCL C version: | OpenCL C 2.0 |
| Driver version: | 1642.5 (VM) |
| Profile: | FULL_PROFILE |
| Version: | OpenCL 2.0 AMD-APP (1642.5) |
| Extensions: | cl_khr_fp64 cl_amd_fp64 cl_khr_global_int32_base_atomics cl_khr_global_int32_extended_atomics cl_khr_local_int32_base_atomics |
| Device Type: | CL_DEVICE_TYPE_CPU |
| Vendor ID: | 1002h |
| Board name: | |
| Max compute units: | 8 |
| Max work items dimensions: | 3 |
| Max work items[0]: | 1024 |
| Max work items[1]: | 1024 |
| Name: | AMD FX(tm)-8120 Eight-Core Processor |
| Vendor: | AuthenticAMD |
| Device OpenCL C version: | OpenCL C 1.2 |
| Driver version: | 1642.5 (sse2,avx,fma4) |
| Profile: | FULL_PROFILE |

Modelo de Execução

1. Descobrindo a plataforma e dispositivos
2. **Criando o contexto de execução**

```
cl::Context contexto( dispositivos );
```

O código hospedeiro gerencia a execução dos kernels através do contexto de execução.

Modelo de Execução

1. Descobrindo a plataforma e dispositivos
2. Criando o contexto de execução
3. **Criando a fila de comandos para um dispositivo**

```
c1::CommandQueue fila( contexto, dispositivos[0] );
```

O código hospedeiro interage com os dispositivos através das filas de comandos. Uma fila de comando é criada para cada dispositivo.

Exemplo Ilustrativo

Calcular a raiz quadrada de cada elemento de um vetor:

| | 0 | 1 | 2 | 3 | 4 | 5 | ... | n-1 |
|----------|---|---|---|---|---|---|-----|-----|
| float* X | 0 | 1 | 2 | 3 | 4 | 5 | ... | n-1 |



| float* Y | $\sqrt{0}$ | $\sqrt{1}$ | $\sqrt{2}$ | $\sqrt{3}$ | $\sqrt{4}$ | $\sqrt{5}$ | $\sqrt{...}$ | $\sqrt{n-1}$ |
|----------|------------|------------|------------|------------|------------|------------|--------------|--------------|
| | 0 | 1 | 2 | 3 | 4 | 5 | ... | n-1 |

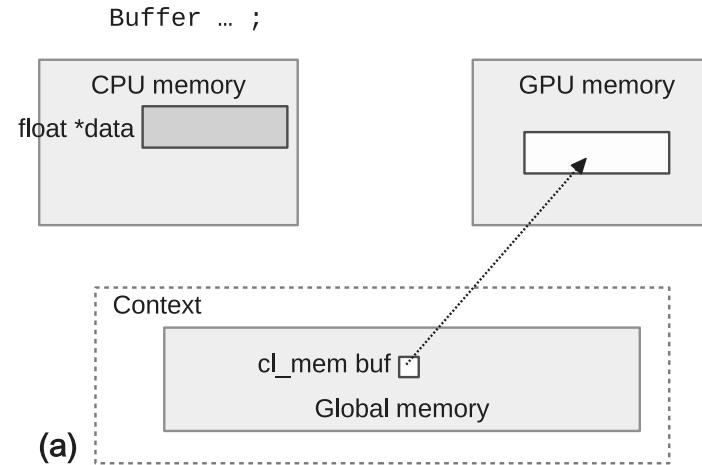
Modelo de Memória

1. Descobrindo a plataforma e dispositivos
2. Criando o contexto de execução
3. Criando a fila de comandos para um dispositivo
4. **Preparação da memória (leitura e escrita)**

```
cl::Buffer bufferX( contexto, CL_MEM_READ_ONLY, n*sizeof(float) );
cl::Buffer bufferY( contexto, CL_MEM_WRITE_ONLY, n*sizeof(float) );
```

Buffers são criados para lidar com os objetos de memória.

Modelo de Memória



Criação de um buffer não inicializado na memória do dispositivo.

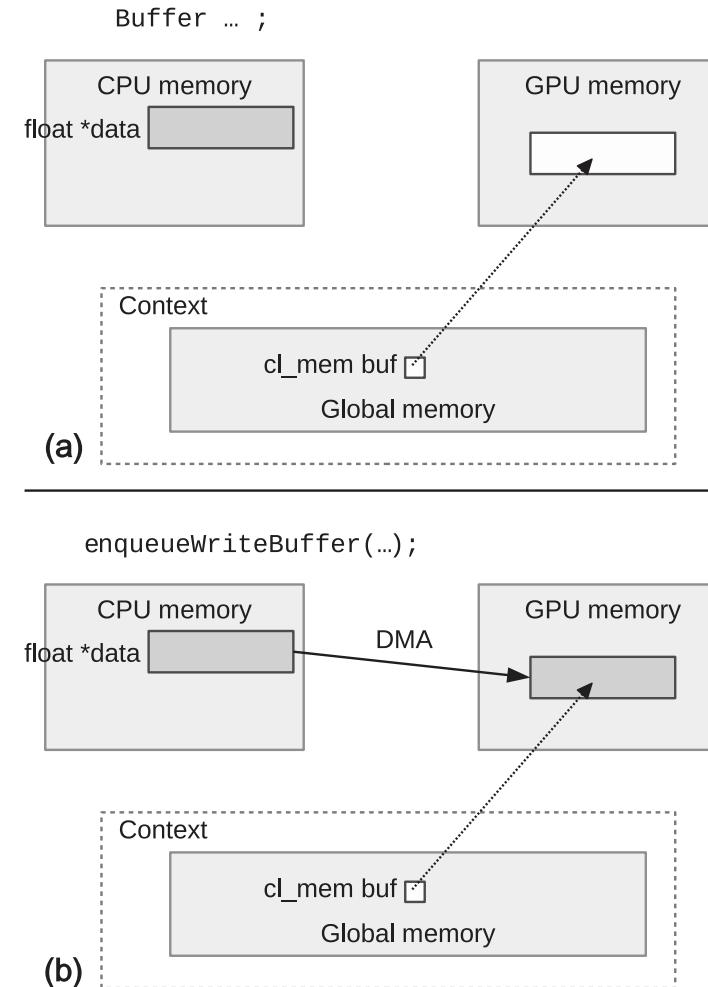
Modelo de Memória

1. Descobrindo a plataforma e dispositivos
2. Criando o contexto de execução
3. Criando a fila de comandos para um dispositivo
4. Preparação da memória (leitura e escrita)
5. **Transferência de dados para o dispositivo**

```
float* X = new float[n];
for(int i = 0; i < n; ++i ) X[i] = i;

fila.enqueueWriteBuffer( bufferX, CL_TRUE, 0, n*sizeof(float), X );
```

Modelo de Memória



Transferência explícita dos dados do host para o dispositivo.

Exemplo Ilustrativo

Calcular a raiz quadrada de cada elemento de um vetor:

| | 0 | 1 | 2 | 3 | 4 | 5 | ... | n-1 |
|----------|---|---|---|---|---|---|-----|-----|
| float* X | 0 | 1 | 2 | 3 | 4 | 5 | ... | n-1 |



| float* Y | $\sqrt{0}$ | $\sqrt{1}$ | $\sqrt{2}$ | $\sqrt{3}$ | $\sqrt{4}$ | $\sqrt{5}$ | $\sqrt{...}$ | $\sqrt{n-1}$ |
|----------|------------|------------|------------|------------|------------|------------|--------------|--------------|
| | 0 | 1 | 2 | 3 | 4 | 5 | ... | n-1 |

Kernel OpenCL

Solução sequencial:

```
void raiz ( const float * x, float * y, int n )
{
    for ( int i = 0; i < n; ++i )
    {
        y[i] = sqrt( x[i] );
    }
}
```

Solução paralela via OpenCL (*kernel*):

```
const char* kernel_str =
"__kernel void raiz( __global const float * x, __global float * y ) "
"{
    int i = get_global_id(0);
    y[i] = sqrt( x[i] );
}";
```

Kernel OpenCL

- Escrito em uma linguagem de programação conhecida como *OpenCL C/C++*
 - derivada da especificação C11/C++14
 - modificações para comportar arquiteturas heterogêneas

Kernel OpenCL

Exclusões:

- Recursividade
- Ponteiros para funções (e funções virtuais)
- Vetores (*arrays*) de tamanho variável
- Ponteiros para ponteiros como argumentos
- Exceções C++ (*throw, catch*)

Kernel OpenCL

Extensões

- Qualificadores de espaço de memória
 - global, constant, local, private;* ou
 - global, --constant, --local, --private*
- Biblioteca nativa de funções e constantes:
lógicas, aritméticas, relacionais, trigonométricas,
atômicas, etc.
- Tipos vetoriais
 - Notação: tipo $<n>$, com $n = 1, 2, 4, 8, 16$
 - Ex: int4, float8, short2, uchar16

Kernel OpenCL

Extensões (cont.)

- Operações vetoriais
 - entre vetores com mesmo número de componentes
 - entre vetores e escalares

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

Funções de identificação

- Item/grupo de trabalho:

`get_global_id(dim)`

`get_local_id(dim)`

`get_group_id(dim)`

- Domínio de índices:

`get_work_dim()`

`get_global_size(dim)`

`get_local_size(dim)`

`get_num_groups(dim)`

`get_global_offset(dim)`

Modelo de Programação

1. Descobrindo a plataforma e dispositivos
2. Criando o contexto de execução
3. Criando a fila de comandos para um dispositivo
4. Preparação da memória (leitura e escrita)
5. Transferência de dados para o dispositivo
6. **Criando e compilando o programa**

```
cl::Program::Sources fonte(1,make_pair( kernel_str,strlen( kernel_str ) ) );  
cl::Program programa( contexto, fonte );  
  
programa.build();
```

Kernel OpenCL

Solução paralela via OpenCL (*kernel*):

```
const char* kernel_str =  
"__kernel void raiz( __global const float * x, __global float * y ) "  
"{ "  
"    int i = get_global_id(0); "  
"    y[i] = sqrt( x[i] ); "  
"} ";
```

Modelo de Programação

1. Descobrindo a plataforma e dispositivos
2. Criando o contexto de execução
3. Criando a fila de comandos para um dispositivo
4. Preparação da memória (leitura e escrita)
5. Transferência de dados para o dispositivo
6. Criando e compilando o programa
7. **Extraindo o kernel do programa**

```
cl::Kernel kernel( programa, "raiz");
```

Modelo de Programação

1. Descobrindo a plataforma e dispositivos
2. Criando o contexto de execução
3. Criando a fila de comandos para um dispositivo
4. Preparação da memória (leitura e escrita)
5. Transferência de dados para o dispositivo
6. Criando e compilando o programa
7. Extrair o *kernel* do programa
8. **Execução do kernel**

```
kernel.setArg(0, bufferX );  
kernel.setArg(1, bufferY );  
fila.enqueueNDRangeKernel( kernel, cl::NDRange(),  
                           cl::NDRange( n ), cl::NDRange() );  
fila.finish();
```

Kernel OpenCL

Solução paralela via OpenCL (*kernel*):

```
const char* kernel_str =  
"__kernel void raiz( __global const float * x, __global float * y ) "  
"{ "  
"    int i = get_global_id(0); "  
"    y[i] = sqrt( x[i] ); "  
"} ";
```

Modelo de Memória

Declaração:

```
kernel void f( __global const float * glc,  
              __global int * gl,  
              __constant float * cnt,  
              __local uint * loc,  
              float s )  
{ ... }
```

Sintaxe de definição:

```
setArg( índice, objeto );  
setArg( índice, tamanho, ponteiro );
```

Definição:

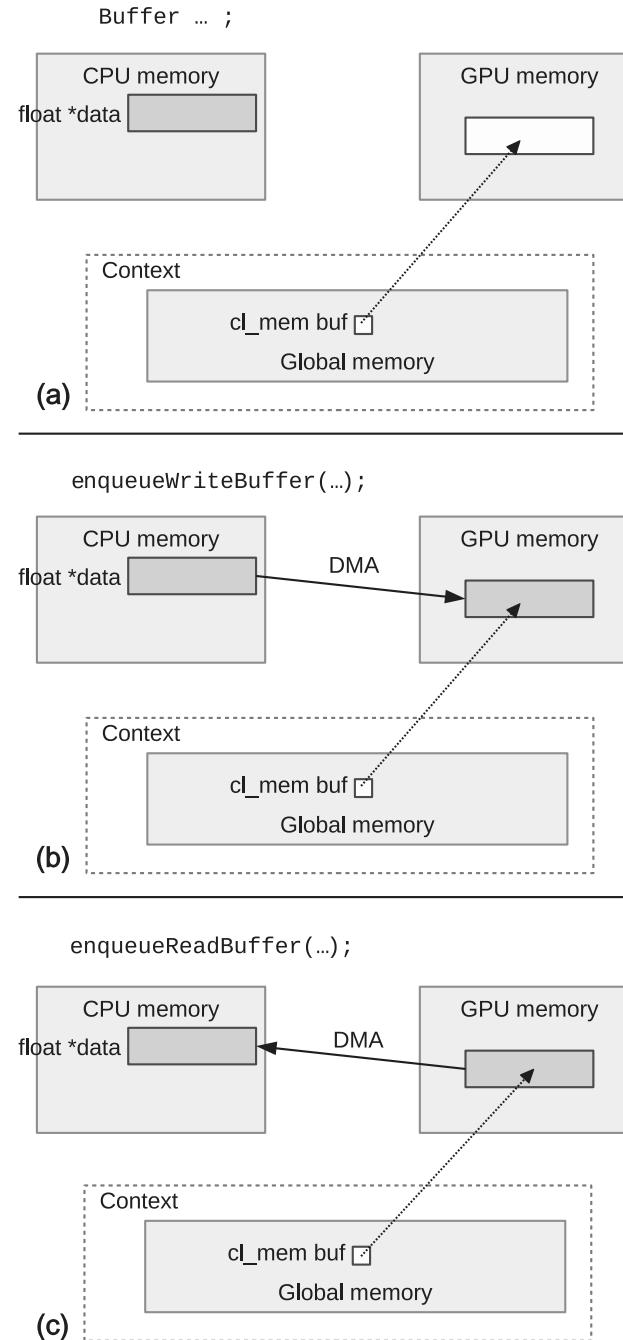
```
setArg( 0, bufferX );  
setArg( 1, bufferY );  
setArg( 2, bufferZ );  
setArg( 3, sizeof( uint ) * num_elementos, NULL );  
setArg( 4, (float) 3.1415 );
```

Modelo de Memória

1. Descobrindo a plataforma e dispositivos
2. Criando o contexto de execução
3. Criando a fila de comandos para um dispositivo
4. Preparação da memória (leitura e escrita)
5. Transferência de dados para o dispositivo
6. Criando e compilando o programa
7. Extraindo o *kernel* do programa
8. Execução do *kernel*
9. **Transferência dos resultados para o hospedeiro**

```
float* Y = new float[n];  
  
fila.enqueueReadBuffer( bufferY, CL_TRUE, 0, n*sizeof(float), Y );
```

Modelo de Memória



```

#define __CL_ENABLE_EXCEPTIONS
#include <cl.hpp>
#include <iostream>
#include <vector>
#include <utility>
#include <cstdlib>
using namespace std;
const char* kernel_str =
"__kernel void "
"raiz( __global const float * x, __global float * y ) "
"{ "
"    int i = get_global_id(0); "
"    y[i] = sqrt( x[i] ); "
"} ";
int main(int argc,char* argv[] )
{
    const int n =atoi( argv[1] );
    float* X = new float[n];
    for(int i = 0; i < n; ++i ) X[i] = i;
    float* Y = new float[n];

    vector<cl::Platform> plataformas;
    vector<cl::Device> dispositivos;
    cl::Platform::get( &plataformas );
    plataformas[0].getDevices( CL_DEVICE_TYPE_ALL, &dispositivos );
    cl::Context contexto( dispositivos );
    cl::CommandQueue fila( contexto, dispositivos[0] );
    cl::Program::Sources fonte(1,make_pair( kernel_str,strlen( kernel_str ) ) );
    cl::Program programa( contexto, fonte );
    programa.build();
    cl::Kernel kernel( programa,"raiz");
    cl::Buffer bufferX( contexto, CL_MEM_READ_ONLY, n*sizeof(float) );
    cl::Buffer bufferY( contexto, CL_MEM_WRITE_ONLY, n*sizeof(float) );
    fila.enqueueWriteBuffer( bufferX, CL_TRUE,0, n*sizeof(float), X );
    kernel.setArg(0, bufferX );
    kernel.setArg(1, bufferY );
    fila.enqueueNDRangeKernel( kernel, cl::NDRange(), cl::NDRange( n ), cl::NDRange() );
    fila.finish();
    fila.enqueueReadBuffer( bufferY, CL_TRUE,0, n*sizeof(float), Y );
    for(int i = 0; i < n; ++i ) cout <<'['<< Y[i] <<']'; cout << endl;
    delete[] X, Y;
    return 0;
}

```

Modelo de Memória

1. Descobrindo a plataforma e dispositivos
2. Criando o contexto de execução
3. Criando a fila de comandos para um dispositivo
4. Preparação da memória (leitura e escrita)
5. Transferência de dados para o dispositivo
6. Criando e compilando o programa
7. Extrair o *kernel* do programa
8. Execução do *kernel*
9. Transferência dos resultados para o hospedeiro

Eventos

```
vector<cl::Platform> plataformas;
vector<cl::Device> cpu_dispositivos, gpu_dispositivos;

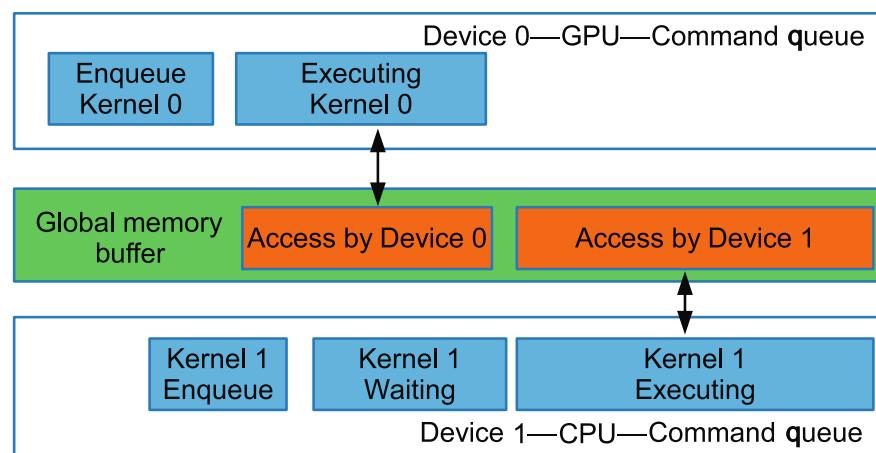
cl::Platform::get( &plataformas );

plataformas[0].getDevices( CL_DEVICE_TYPE_CPU, &cpu_dispositivos );
plataformas[1].getDevices( CL_DEVICE_TYPE_GPU, &gpu_dispositivos );

cl::Context cpu_contexto( cpu_dispositivos );
cl::Context gpu_contexto( gpu_dispositivos );

cl::CommandQueue cpu_fila( cpu_contexto, cpu_dispositivos[0] );
cl::CommandQueue gpu_fila( gpu_contexto, gpu_dispositivos[0] );

cl::Event event_gpu, event_cpu;
gpu_fila.enqueueNDRangeKernel( kernel_gpu, cl::NDRange(),
                               cl::NDRange( global ), cl::NDRange( local ), NULL, &event_gpu );
cpu_fila.enqueueNDRangeKernel( kernel_cpu, cl::NDRange(),
                               cl::NDRange( global ), cl::NDRange( local ), &event_gpu, &event_cpu );
```



Eventos

```
vector<cl::Platform> plataformas;
vector<cl::Device> cpu_dispositivos, gpu_dispositivos;

cl::Platform::get( &plataformas );

plataformas[0].getDevices( CL_DEVICE_TYPE_CPU, &cpu_dispositivos );
plataformas[1].getDevices( CL_DEVICE_TYPE_GPU, &gpu_dispositivos );

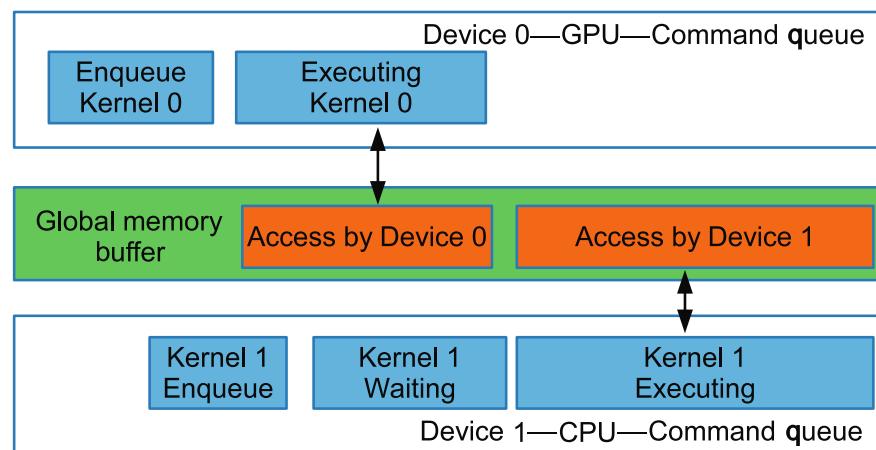
cl::Context cpu_contexto( cpu_dispositivos );
cl::Context gpu_contexto( gpu_dispositivos );

cl::CommandQueue cpu_fila( cpu_contexto, cpu_dispositivos[0] );
cl::CommandQueue gpu_fila( gpu_contexto, gpu_dispositivos[0] );

cl::Event event_gpu, event_cpu;
gpu_fila.enqueueNDRangeKernel( kernel_gpu, cl::NDRange(),
                               cl::NDRange( global ), cl::NDRange( local ), NULL, &event_gpu );

cl::Event::waitForEvents( vector<cl::Event> e(1, event_gpu ) );

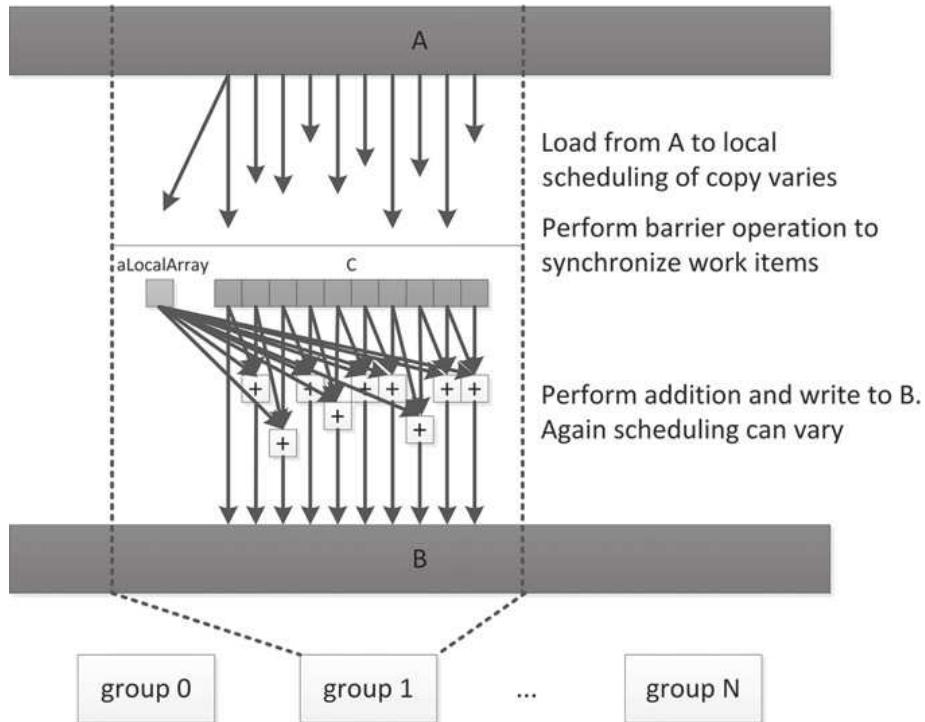
cpu_fila.enqueueNDRangeKernel( kernel_cpu, cl::NDRange(),
                               cl::NDRange( global ), cl::NDRange( local ), NULL, &event_cpu );
```



Sincronismo

1. Não há sincronia global.
2. Não é possível sincronizar fora de um grupo de trabalho.
3. Apenas itens de trabalho de um mesmo grupo podem sincronizar entre si.

Sincronismo

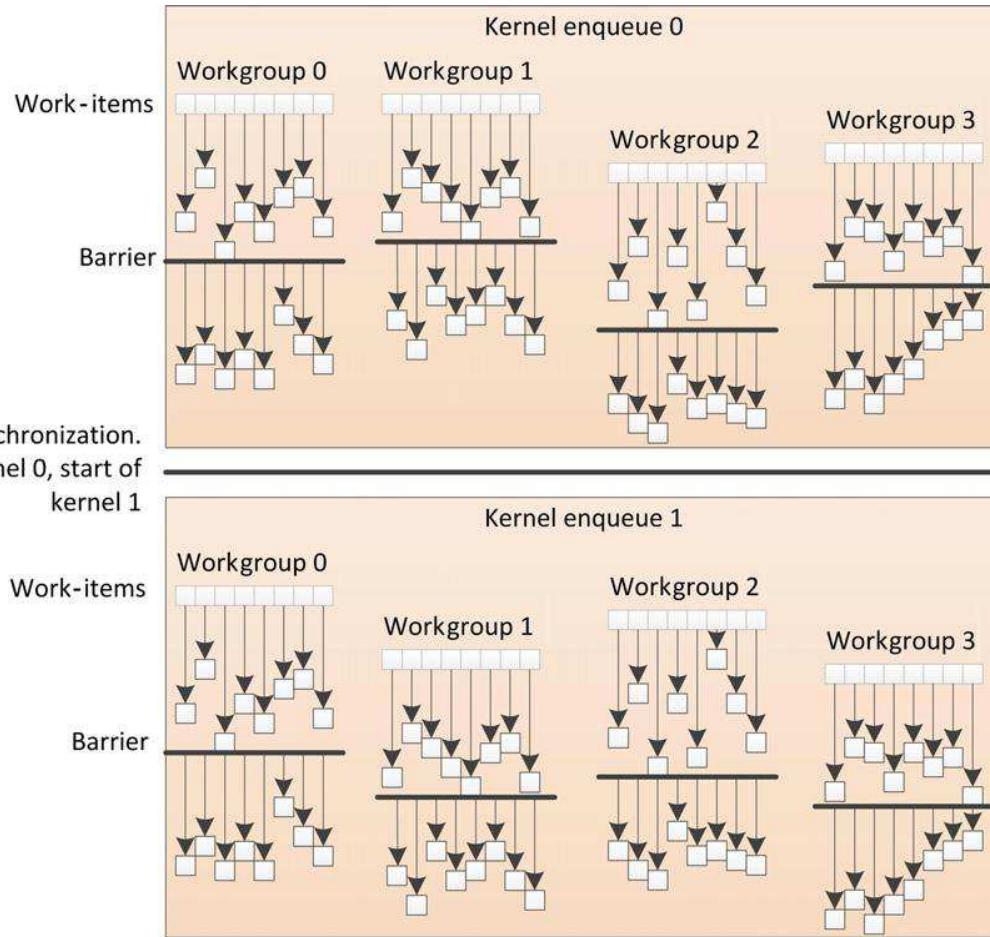


```
__kernel void localAccess(
    __global float* A,
    __global float* B,
    __local float* C )
{
    __local float aLocalArray[1];
    if( get_local_id(0) == 0 ) {
        aLocalArray[0] = A[0];
    }
    C[get_local_id(0)] = A[get_global_id(0)];
    barrier(CLK_LOCAL_MEM_FENCE);

    float neighborSum = C[get_local_id(0)] + aLocalArray[0];

    if( get_local_id(0) > 0 )
        neighborSum = neighborSum + C[get_local_id(0)-1];
    B[get_global_id(0)] = neighborSum;
}
```

Sincronismo



Global synchronization.
End of kernel 0, start of
kernel 1

```

__kernel void example(
    __global const float *a,
    __global float *b,
    __local float *localbuf)
{
    localbuf[get_local_id(0)] = a[get_global_id(0)];
    barrier( CLK_LOCAL_MEM_FENCE );
    b[get_global_id(0)] = localbuf[get_local_id(0)] +
        localbuf[(get_local_id(0) + 1) %
        get_local_size(0)];
}

cl::Buffer input( contexto, CL_MEM_READ_ONLY, 32*sizeof(float) );
cl::Buffer intermediate( contexto, CL_MEM_READ_WRITE, 32*sizeof(float) );
cl::Buffer output( contexto, CL_MEM_WRITE_ONLY, 32*sizeof(float) );
fila.enqueueWriteBuffer( input, CL_TRUE, 0, 32*sizeof(float), hostInput );
kernel.setArg( 0, input );
kernel.setArg( 1, intermediate );
kernel.setArg( 2, 8*sizeof(float), NULL );
fila.enqueueNDRangeKernel( kernel, cl::NDRange(),
    cl::NDRange( 32 ), cl::NDRange( 8 ) );

kernel.setArg( 0, intermediate );
kernel.setArg( 1, output );
kernel.setArg( 2, 8*sizeof(float), NULL );
fila.enqueueNDRangeKernel( kernel, cl::NDRange(),
    cl::NDRange( 32 ), cl::NDRange( 8 ) );
fila.enqueueReadBuffer( output, CL_TRUE, 0, 32*sizeof(float), hostOutput );

```

Sincronismo

```
kernel void f()
{
    int i = get_global_id(0);
    __local int x[10];
    x[i] = i;

    if( i > 0 )
        int y = x[i-1];
}
```

Exemplo de acesso inconsistente

Sincronismo

```
kernel void f()
{
    int i = get_global_id(0);
    __local int x[10];
    x[i] = i;

    barrier( CLK_LOCAL_MEM_FENCE );

    if( i > 0 )
        int y = x[i-1];
}
```

Acesso consistente após ponto de sincronia

Sincronismo

```
kernel void deadlock( global float * x )
{
    int i = get_global_id(0);

    if( i == 0 )
        barrier( CLK_LOCAL_MEM_FENCE );
    else
        x[i] = i;
}
```

Execução parada indefinidamente

Eventos

```
cl::CommandQueue fila( contexto, dispositivo, CL_QUEUE_PROFILING_ENABLE );

cl::Event e_tempo;

fila.enqueueNDRangeKernel( kernel, cl::NDRange(), cl::NDRange( elementos ),
                           cl::NDRange() , NULL, &e_tempo );

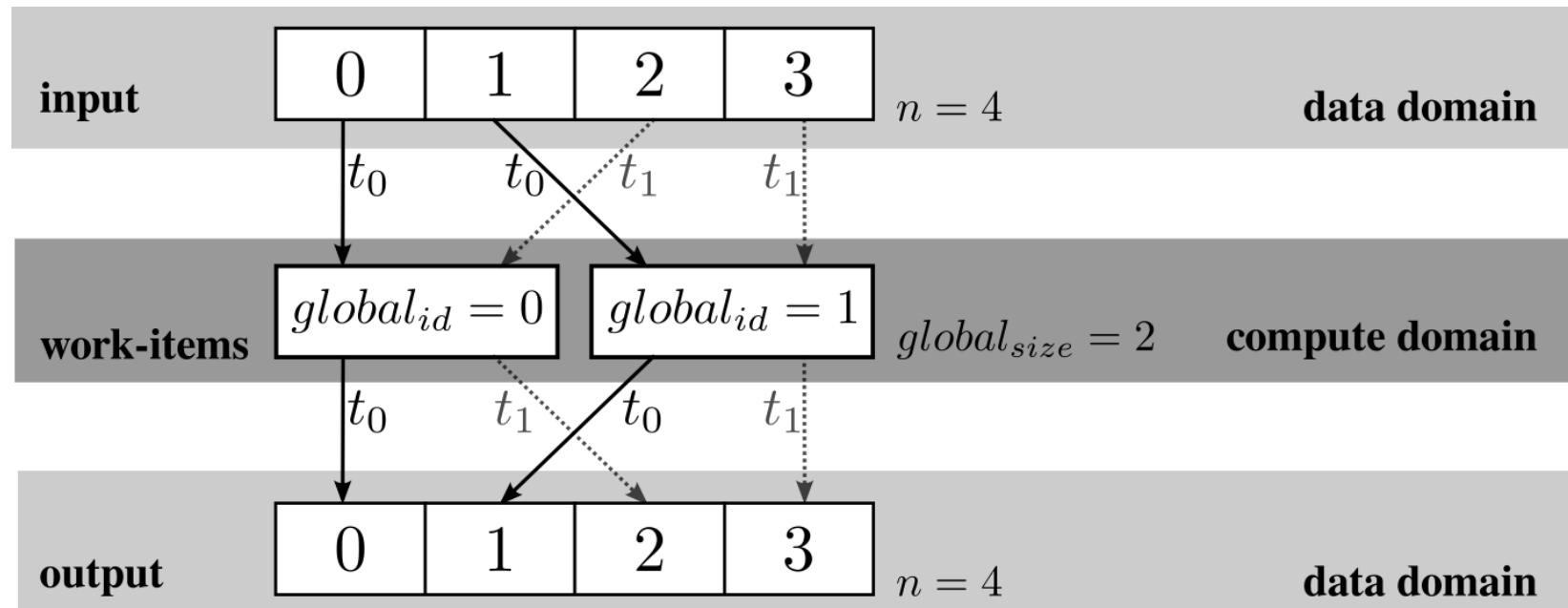
fila.finish();

cl_ulong inicio, fim;
e_tempo.getProfilingInfo( CL_PROFILING_COMMAND_START, &inicio );
e_tempo.getProfilingInfo( CL_PROFILING_COMMAND_END, &fim );

double tempo_execucao_s = (fim - inicio)/1.0E9;
```

Itens e Grupos de Trabalho

- O domínio de índices é o mecanismo que conecta o *domínio de dados* ao *domínio de cômputo*



Pseudo-kernel:

```
for  $t \leftarrow 0$  to  $n/global\_size - 1$  do
     $i \leftarrow t \times global\_size + global\_id;$ 
     $output[i] \leftarrow \sqrt{input[i]};$ 
end
```

Funções de Identificação

- Item/grupo de trabalho:

`get_global_id(dim)`

`get_local_id(dim)`

`get_group_id(dim)`

- Domínio de índices:

`get_work_dim()`

`get_global_size(dim)`

`get_local_size(dim)`

`get_num_groups(dim)`

`get_global_offset(dim)`

Consulta de Propriedades

Plataformas:

- Nome da plataforma:

```
plataforma.getInfo<CL_PLATFORM_NAME>();
```

Dispositivos:

- Tipo do dispositivo:

```
dispositivo.getInfo<CL_DEVICE_TYPE>();
```

- Nome do dispositivo:

```
dispositivo.getInfo<CL_DEVICE_NAME>();
```

- Número de unidades computacionais:

```
dispositivo.getInfo<CL_DEVICE_MAX_COMPUTE_UNITS>();
```

Consulta de Propriedades

Memória dos dispositivos:

- Memória *global* alocável (`_global`):

```
dispositivo.getInfo<CL_DEVICE_MAX_MEM_ALLOC_SIZE>();
```

- Memória *local* alocável (`_local`):

```
dispositivo.getInfo<CL_DEVICE_LOCAL_MEM_SIZE>();
```

- Memória *constante* alocável (`_constant`):

```
dispositivo.getInfo<CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE>();
```

Dimensões máximas:

- Tamanho máximo local:

```
dispositivo.getInfo<CL_DEVICE_MAX_WORK_GROUP_SIZE>();
```

- Tamanho máximo em cada dimensão:

```
dispositivo.getInfo<CL_DEVICE_MAX_WORK_ITEM_SIZES>() [dim];
```

Compilação e Execução

No GNU/Linux:

- Compilação:

```
g++ -o <out> <c++ source> -I<OpenCL-include-dir> -L<OpenCL-libdir> -lOpenCL
```

```
g++ -o ex ex.cc -I/usr/include/CL -lOpenCL  
g++ -o ex ex.cc -I. -lOpenCL
```

- Execução:

```
./ex  
[0] [1] [1.41421] [1.73205] [2] [2.23607] [2.44949] [2.64575] [2.82843] [3]
```

Referências

- **Heterogeneous Computing with OpenCL**

B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, D. Schaa

- **OpenCL Programming Guide**

A. Munshi, B. Gaster, T. G. Mattson, J. Fung, D. Ginsburg

- **OpenCL in Action**

Matthew Scarpino

- **The OpenCL Programming Book**

R. Tsuchiyama, T. Nakamura, T. Iizuka, A. Asahara, J. Son, S. Miki

- **OpenCL Specification**

<http://www.khronos.org/opencl/>

Obrigada!