

# Programação em OpenCL

## Uma introdução prática

INE 5645 Programação Paralela e Distribuída

Prof. João Bosco M. Sobral

# *Programação em OpenCL:* *Uma introdução prática*

- César L. B. Silveira<sub>1</sub>, Luiz G. da Silveira Jr.<sub>2</sub>, Gerson Geraldo H. Cavaleiro<sub>3</sub>

<sub>1</sub> V3D Labs, São Leopoldo, RS, Brasil

<sub>2</sub> Universidade do Vale do Rio dos Sinos, São Leopoldo, RS, Brasil

<sub>3</sub> Universidade Federal de Pelotas, Pelotas, RS, Brasil

{cesar,gonzaga}@v3d.com.br, gerson.cavaleiro@ufpel.edu.br

# *Introdução*

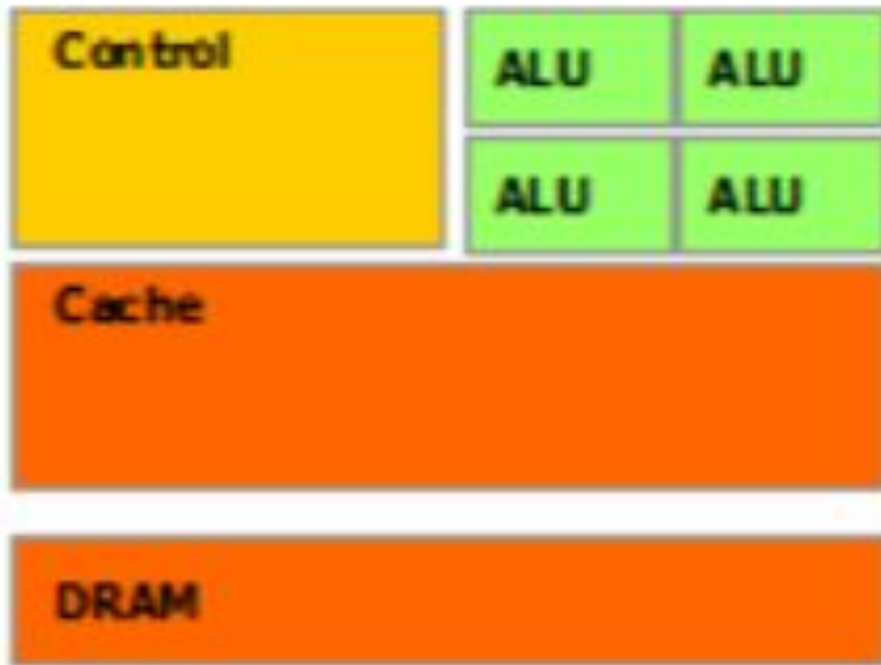
# *Plataformas Heterogêneas*

- Computadores dotados de processadores *multicore* e placas gráficas dotadas de múltiplas unidades de processamento *manycore*.
- Tais plataformas são ditas **heterogêneas**, dada a natureza dos recursos de processamento.
- Indica a necessidade de uma convergência dos esforços de desenvolvimento de software.
- Permitem um processador com CPUs executar tarefas gerais e a GPU executar tarefas de paralelismo de dados.

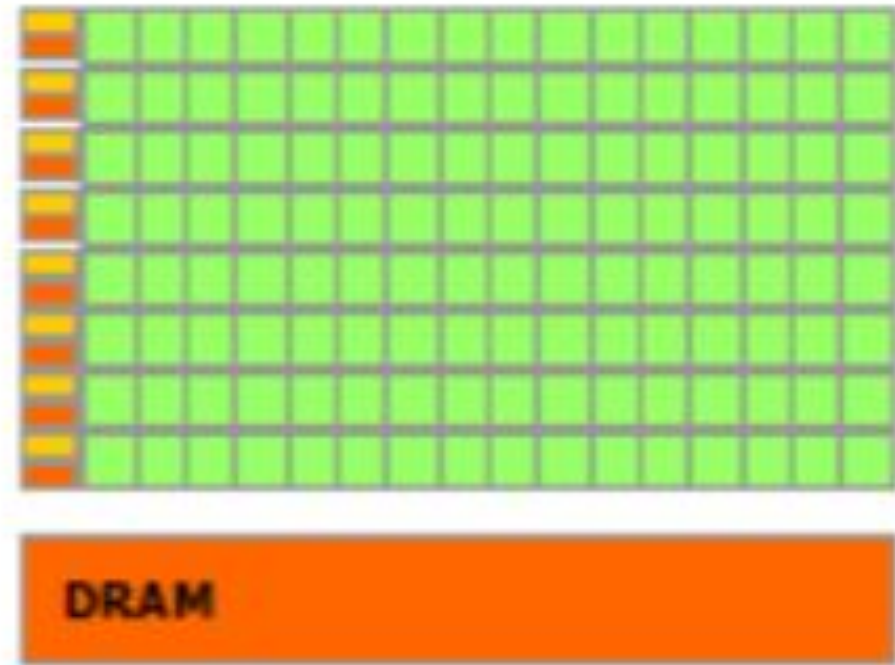
# *GPU – Graphics Processing Unit*

- O uso tradicional das **placas gráficas**, esteve durante muito tempo associado às aplicações de computação gráfica e/ou processamento de imagens.
- Nos últimos anos, **arquiteturas de placas gráficas** tem sido consideradas para implementações de **aplicações científicas em um escopo mais genérico**, uma vez que seus processadores, chamados de **GPU (Graphics Processing Unit)**, dispõem de grande capacidade de processamento.

Uma comparação entre a arquitetura de uma CPU com apenas quatro unidades lógica e aritmética (ULAs) e uma GPU com 128 ULAs.



**CPU**

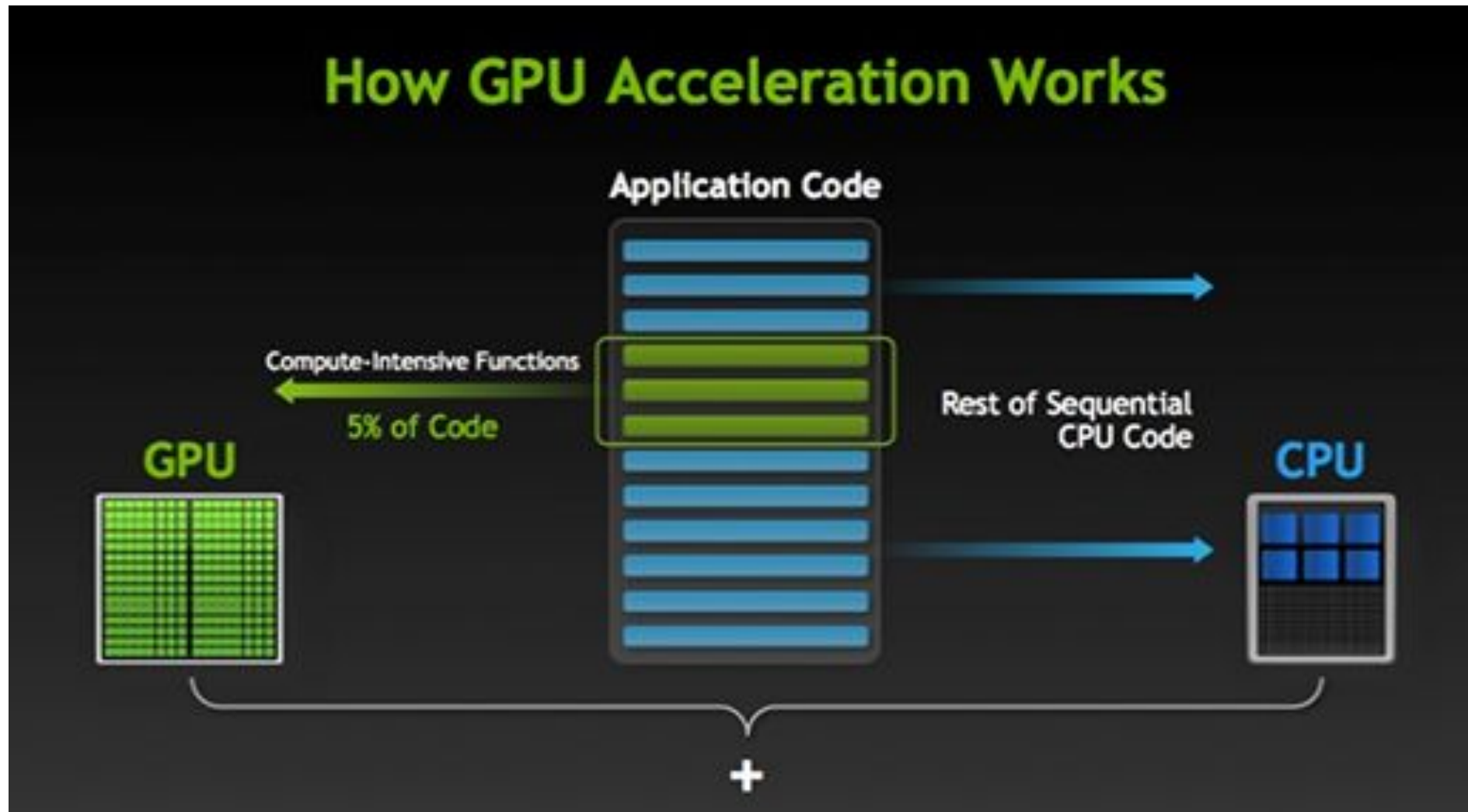


**GPU**

# *GPU – Graphics Processing Unit*

- A GPU surgiu para "aliviar" o processador principal do computador (CPUs) da pesada tarefa de gerar imagens.
- Por isso, é capaz de lidar com um grande volume de cálculos matemáticos e geométricos, condição para o processamento de imagens 3D, utilizadas em jogos, exames médicos computadorizados, entre outros.

Computação acelerada por placas de vídeo é o uso de uma unidade de processamento gráfico (GPU) juntamente com uma CPU para acelerar aplicações complexas ...





## *Uso de GPUs*

- GPU é um processador projetado para **processamento gráfico**, porém sua **arquitetura paralela** contendo **centenas de núcleos**, a torna adequada para processar dados em paralelo.
- O uso de GPUs, justifica-se em função do **desempenho** obtido pelo **investimento realizado**, o qual é altamente favorável ao usuário final.

# Aplicações de GPUs

- **Dinâmica Molecular** (um método de simulação computacional que estuda o movimentos físico dos átomos e moléculas das quais se conhecem o potencial de interação entre estas partículas e as equações que regem o seu movimento)
- **Jogos** (Programas de entretenimento — um jogo *virtual*) (Gaming)
- **Realidade Virtual** (Uma tecnologia de interface avançada entre um usuário e um sistema operacional, com o objetivo de recriar ao máximo a sensação de **realidade** para um indivíduo, levando-o a adotar essa interação como uma de suas realidades temporais)
- **Processamento gráfico de texto, áudio, imagem e vídeo** (multimídia)  
Como visualizar fotos, reproduzir e editar vídeos, organizar músicas, obter informações de endereço, aplicativos de escritório e interagir com o sistema operacional.
- **Algoritmos de inteligência artificial.**
- **Processamento gráfico e renderização 3D em tempo-real.**  
Renderizar é o ato de compilar e obter o produto final de um processamento digital. Ou seja, toda aquela sequência de imagens que você montou na sua linha do tempo precisa ser condensada em um vídeo.
- **Simulações de sistemas complexos**  
Projetos de automóveis, aeronaves, navios, engenharia em geral.

# *Poder de processamento GPUs*

- O poder de processamento oferecido pelas GPUs, vem sendo explorado através de *toolkits* específicos de fabricante, como **NVIDIA CUDA**, **AMD ATI Streaming SDK**, **Intel SDK for OpenCL**, ...
- No contexto das *APIs* (*Application Programming Interface*) direcionadas às aplicações gráficas, emprega-se, atualmente, a **API gráfica OpenGL** ou **Direct3D**, parte do **DirectX SDK** da Microsoft (conjunto de APIs para aplicações de áudio e vídeo).

# *O que é OpenCL - Open Computing Language*

- OpenCL é um padrão aberto para *programação de alto desempenho* em *ambientes computacionais heterogêneos* equipados com processador *multicore* (CPUs) e GPUs *manycore*.
- Com foco no paralelismo, a programação em OpenCL baseia-se na escrita de funções que são executadas em múltiplas instâncias simultâneas.

# *OpenCL – Open Computing Language*

- **Multiplataforma** - disponível em várias classes de hardware e sistemas operacionais.
- **Código portátil** entre arquiteturas (Nvidia, AMD, Intel, Apple, IBM).
- *Paralelismo de dados* (“SIMD”) e *paralelismo de tarefas* (“MIMD”).
- Especificação baseada nas linguagens C e C++.
- Define requisitos para **operações em ponto flutuante** (números reais)
- **Integração** com outras tecnologias (*OpenCL + OpenGL*).

# *Tipos de arquiteturas paralelas SIMD e MIMD*

- SIMD (**Single Instruction Multiple Data**)
- Significa que todas as unidades paralelas compartilham a mesma instrução, mas a realizam em diferentes elementos de dados.

Multiple workers, all doing same thing

Single coordinator



# *SIMD - Single Instruction Multiple Data*

- A idéia é que podemos adicionar os arrays  $A=[1,2,3,4]$  e  $B=[5,6,7,8]$  para obter o array  $S=[6, 8, 10, 12]$ .
- Para isso, tem que haver quatro unidades aritméticas no trabalho, mas todos podem compartilhar a mesma instrução (aqui, "*add*").





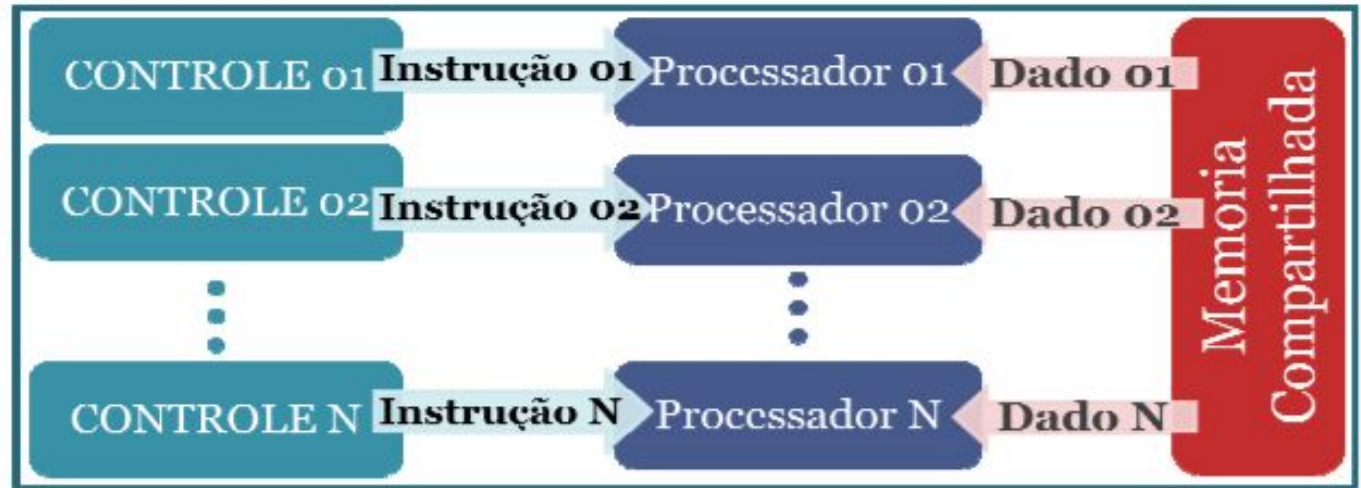
# *Tipos de arquiteturas paralelas SIMD e MIMD*

- MIMD (**M**ultiple **I**nstruction **M**ultiple **D**ata)
- Significa que as unidades paralelas têm instruções distintas, então cada uma delas pode fazer algo diferente em um dado momento.



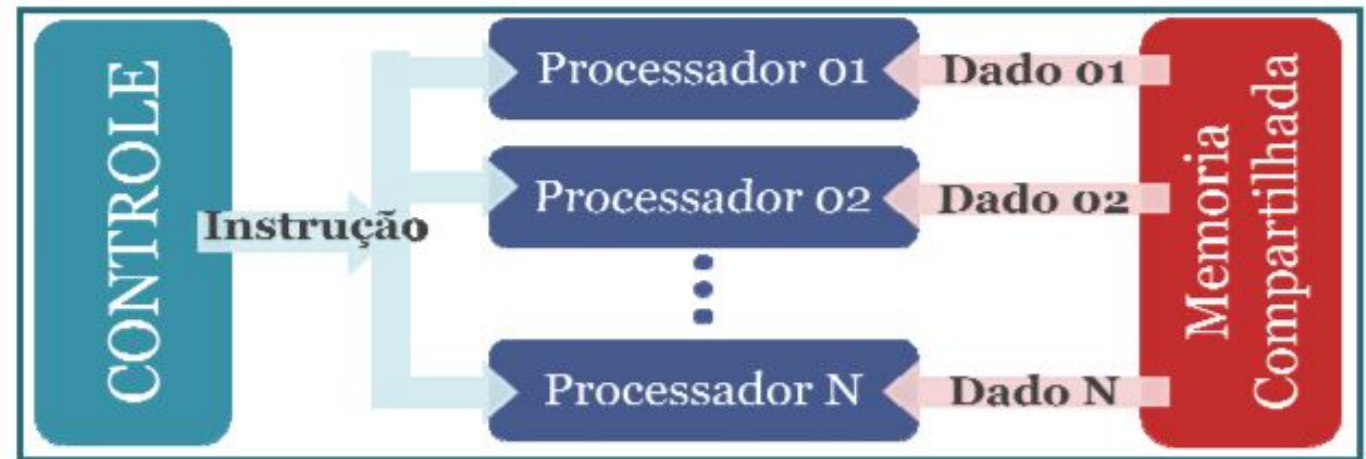
Workers with same objective,  
doing completely different things

# Tipos de Paralelismo



(a)

(a) MIMD



(b)

(b) SIMD

# Paralelismo de dados

- A mesma operação é executada simultaneamente (isto é, em paralelo) aos elementos em uma coleção de origem ou uma matriz.
- Em operações paralelas de dados, a coleção de origem é particionada para que vários threads possam funcionar simultaneamente em diferentes segmentos.

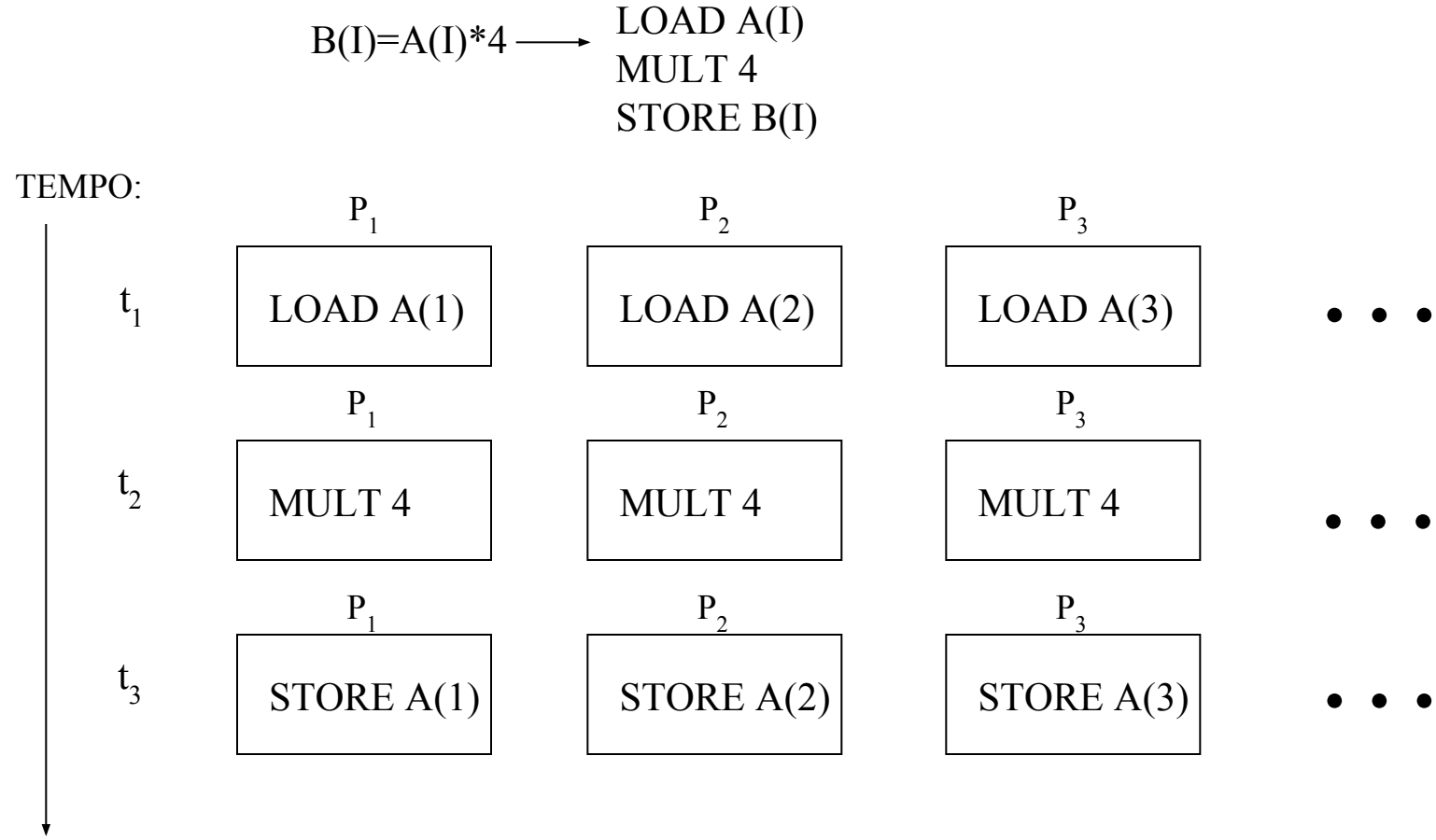
# Paralelismo de dados

- Uma técnica de programação que divide uma grande quantidade de dados em partes menores que podem ser operadas em paralelo.

## Single Instruction, Multiple Data (SIMD)

- **Síncrono**
  - Todas as unidades devem receber a mesma instrução no mesmo instante de tempo de modo que todas possam executá-la de forma simultânea
- **Determinismo**
  - Em cada instante de tempo só existe uma instrução sendo executada por várias unidades, então se o programa é executado com os mesmos dados de entrada, utilizando o mesmo número de unidades de execução, o resultado será o mesmo
- **Apropriado para aplicações que utilizam paralelismo orientado a instrução**

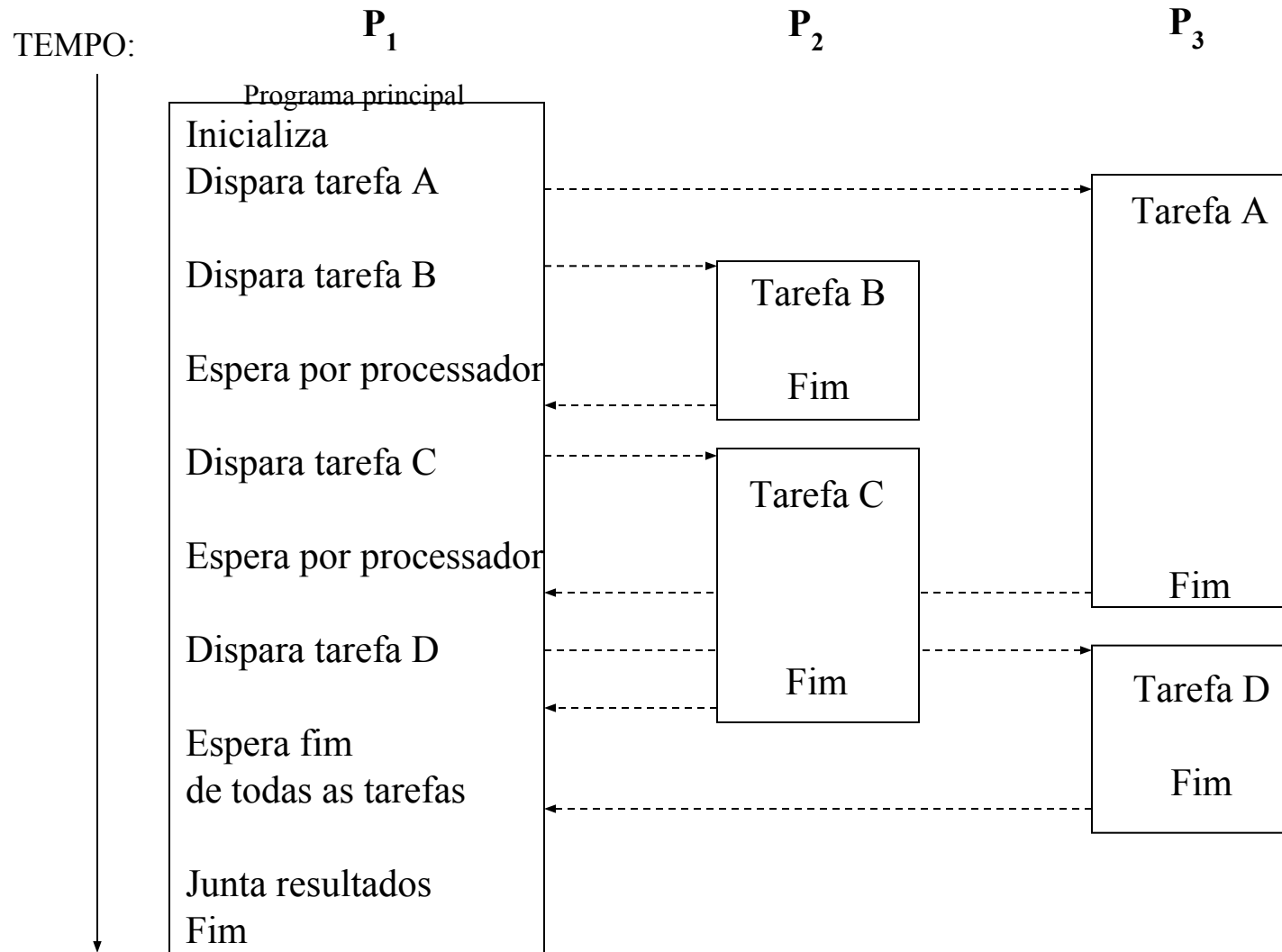
# Single Instruction, Multiple Data (SIMD)



# Terminologia CUDA e OpenCL

CUDA	OpenCL
<i>Streamming Multiprocessor</i> (SM)	Compute Unit (CU)
<i>Streamming Processor</i> (SP)	Processing Element (PE)
<i>host</i>	<i>host</i>
<i>device</i>	<i>device</i>
<i>kernel</i>	<i>kernel</i>
<i>thread</i>	<i>workitem</i>
<i>bloco</i>	<i>workgroup</i>
<i>grid</i>	<i>NDRange</i>

# Multiple Instruction, Multiple Data (MIMD)





# *Ambientes OpenCL Disponíveis*

- FOXC (Fixstars OpenCL Cross Compiler)
- NVIDIA OpenCL
- AMD (ATI) OpenCL
- Apple OpenCL
- IBM OpenCL

# Objetivo

- OpenCL torna possível a escrita de código multi-plataforma para tais dispositivos (CPU + GPU), sem a necessidade do uso de linguagens e ferramentas específicas de fabricante.
- Introduzir conceitos-chave e explorar a arquitetura definida no padrão OpenCL.
- Capacitar recursos humanos a utilizar de forma efetiva todo este potencial de processamento disponível para computação de alto desempenho.

# OpenCL

- A capacidade de processamento paralelo oferecida por processadores *multicore* pode ser explorada pelo uso de *multithreading*, habilitado por tecnologias como POSIX Threads, OpenMP, entre outras.
- O desenvolvimento de soluções em *plataformas computacionais heterogêneas* (CPU + GPU) apresenta-se com custo elevado para o desenvolvedor, que deve possuir domínio de diversos paradigmas e ferramentas para extrair o poder computacional oferecido por estas plataformas.

# OpenCL

- Neste contexto, surge OpenCL, criada pela Apple [Apple Inc. 2009] e padronizada pelo Khronos Group [Khronos Group 2010a].

- Apple Inc. (2009). *OpenCL Programming Guide for Mac OS X*.

[http://developer.apple.com/mac/library/documentation/Performance/Conceptual/OpenCL\\_MacProgGuide/Introduction/Introduction.html](http://developer.apple.com/mac/library/documentation/Performance/Conceptual/OpenCL_MacProgGuide/Introduction/Introduction.html).

- Khronos Group (2010a). *OpenCL - The open standard for parallel programming of heterogeneous systems*. <http://www.khronos.org/opencl/>.

# OpenCL

- OpenCL é uma plataforma aberta e livre de royalties para **computação de alto desempenho** em **sistemas heterogêneos** compostos por **CPU** e **GPUs**.
- OpenCL oferece aos desenvolvedores um ambiente de programação paralela para escrever **códigos portáveis para estes sistemas heterogêneos**.

## Código Sequencial em C

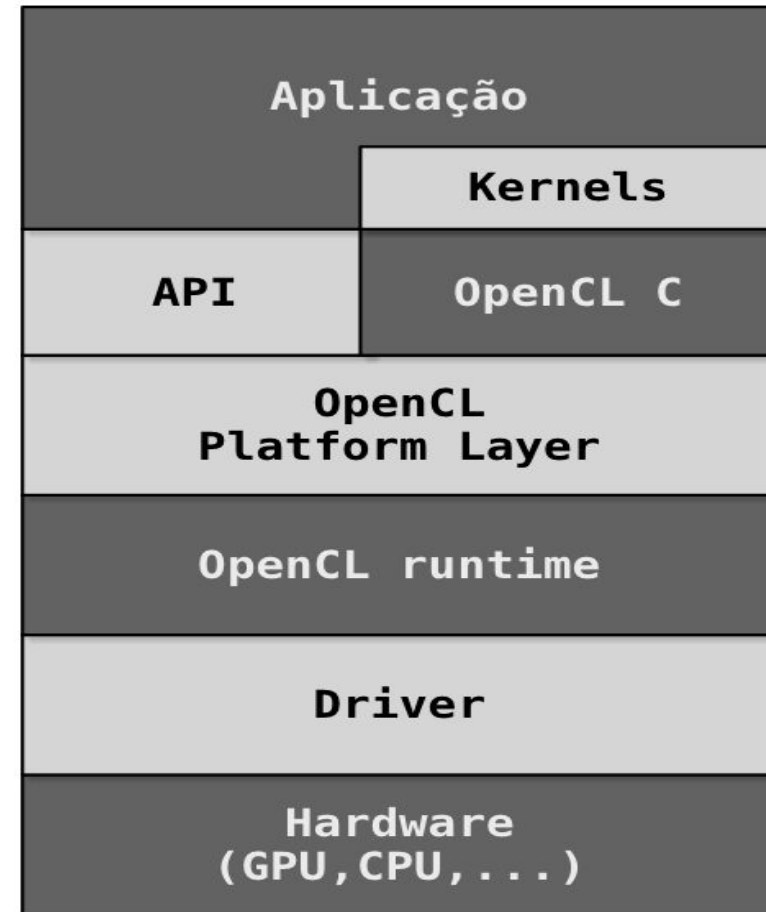
```
void ArrayDiff(const int* a,
const int* b, int* c, int n)
{
    for (int i = 0; i < n; ++i)
    {
        c[i] = a[i] - b[i];
    }
}
```

## Código em OpenCL

```
__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];  
}
```

# Arquitetura em Camadas - OpenCL

- Aplicações OpenCL são estruturadas em camadas, como mostra a Figura 1.
- OpenCL é um *framework* que inclui uma linguagem OpenCL C, uma API C, bibliotecas e um *runtime system*.





# *Importante observar no código OpenCL*

- Não há um loop em kernels, para iterar sobre os arrays.
- O código escrito geralmente focaliza na computação de uma unidade do resultado desejado.
- O *OpenCL runtime* fica responsável por criar tantas instâncias de kernel, quantas forem necessárias para o processamento de todo o conjunto de dados, no momento da execução.

# *Importante observar no código OpenCL*

- *Não é necessário informar ao kernel OpenCL o tamanho do conjunto de dados, uma vez que a manipulação desta informação é responsabilidade do OpenCL runtime.*
- O *OpenCL runtime* permite ao desenvolvedor *enfileirar comandos para execução nos dispositivos (GPUs)*, sendo também responsável por *gerenciar os recursos de memória* e *computação* disponíveis.

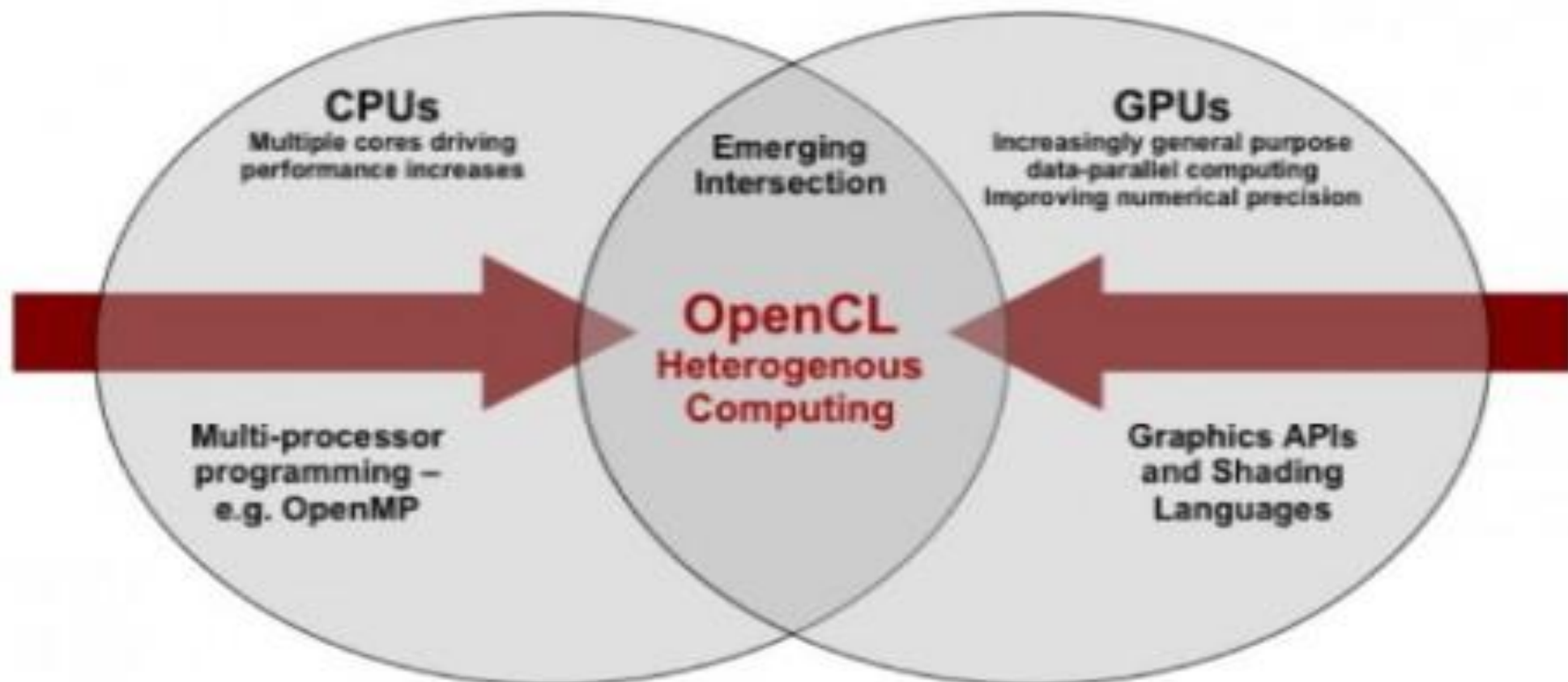
# *Kernels e a API C*

- A *aplicação* faz uso da *API C* para comunicar-se com a camada *OpenCL Platform Layer*, enviando comandos ao *OpenCL runtime*, que gerencia diversos aspectos da execução.
- *Kernels* correspondem às entidades que são escritas pelo desenvolvedor na linguagem *OpenCL C*.

# *Uso de OpenCL*

- O uso de **OpenCL** é, portanto, recomendado para aplicações que realizam operações computacionalmente custosas, porém paralelizáveis por permitirem o cálculo independente de diferentes porções do resultado.

# Processor Parallelism



## OpenCL – Open Computing Language

Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors

# *Arquitetura OpenCL*

- A arquitetura OpenCL é **descrita por quatro modelos**, bem como por uma série de conceitos associados a estes.
- Arquitetura abstrata de baixo nível.
- Implementações mapeiam para entidades físicas.
- Quatro modelos:
  - Plataforma
  - Execução
  - Programação
  - Memória

# Modelo de Plataforma

- Descreve as **entidades** presentes em um ambiente computacional OpenCL.
- Um ambiente computacional OpenCL é integrado por **um hospedeiro (host), que agrega um ou mais dispositivos (devices)**.
- Cada dispositivo possui uma ou mais unidades de computação (**compute units**), sendo estas compostas de um ou mais elementos de processamento (**processing elements**).
- O hospedeiro é responsável pela descoberta e inicialização dos dispositivos, bem como pela transferência de dados e tarefas para execução nestes.
- A Figura 2 apresenta um diagrama do **modelo de plataforma e de memória**.

# Modelo OpenCL

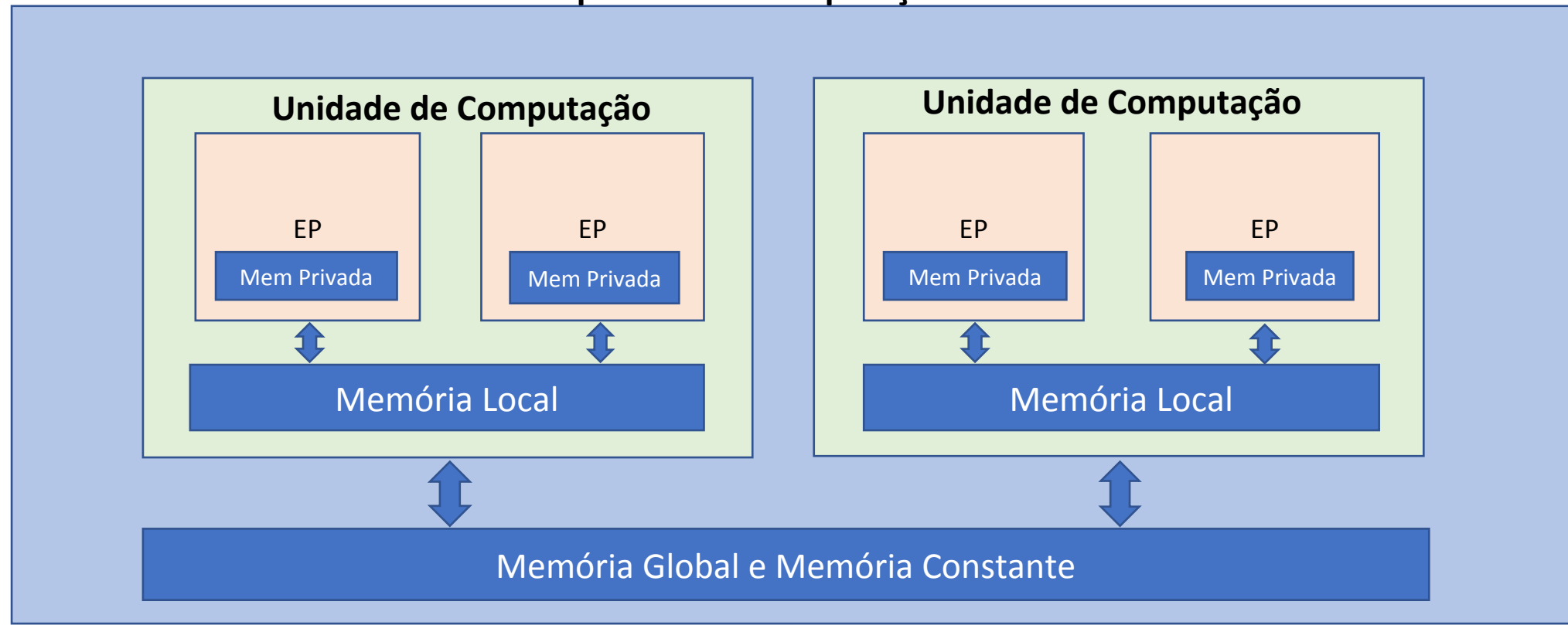
- Seguindo o estilo da especificação OpenCL, é possível descrever as principais ideias inerentes ao OpenCL usando uma hierarquia de modelos:
  - Modelo da Plataforma
  - Modelo de Execução
  - Modelo de Memória



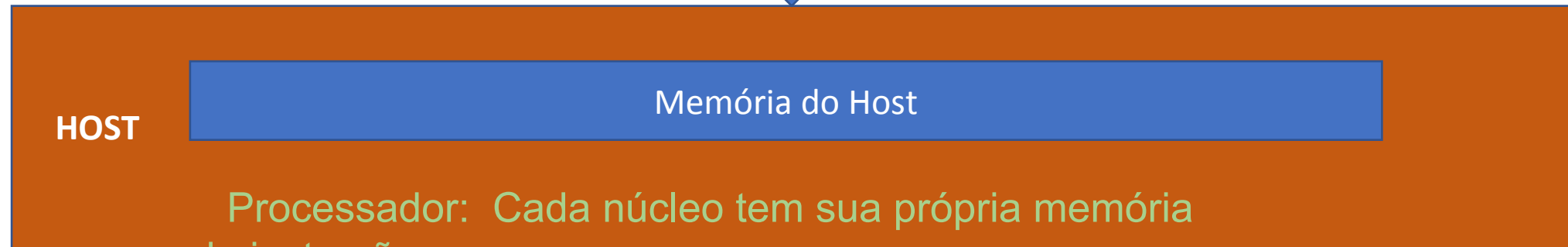
# *Modelo de Plataforma*

- Uma plataforma OpenCL é frequentemente descrita como uma combinação de dois elementos computacionais, o **host** e um ou mais dispositivos.
- O elemento **host** é responsável por administrar a execução da aplicação, enquanto os dispositivos realizam a tarefa propriamente dita.
- Um **dispositivo** OpenCL é composto de um ou mais unidades de computação (UC) e cada UC é composto por, um ou mais, elemento de processamento (EP).

## Dispositivo de Computação - GPU

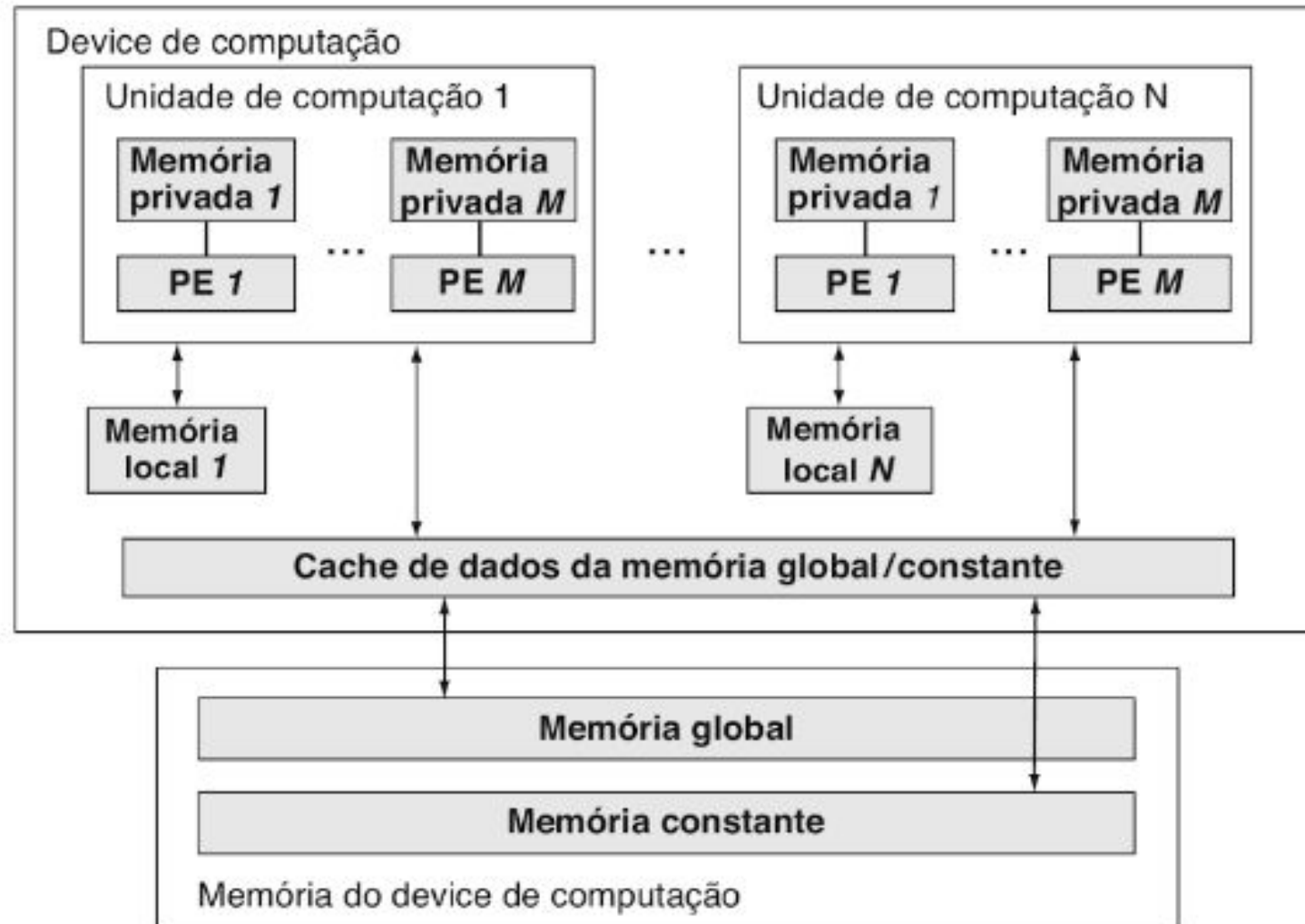


## Modelo de Plataforma OpenCL

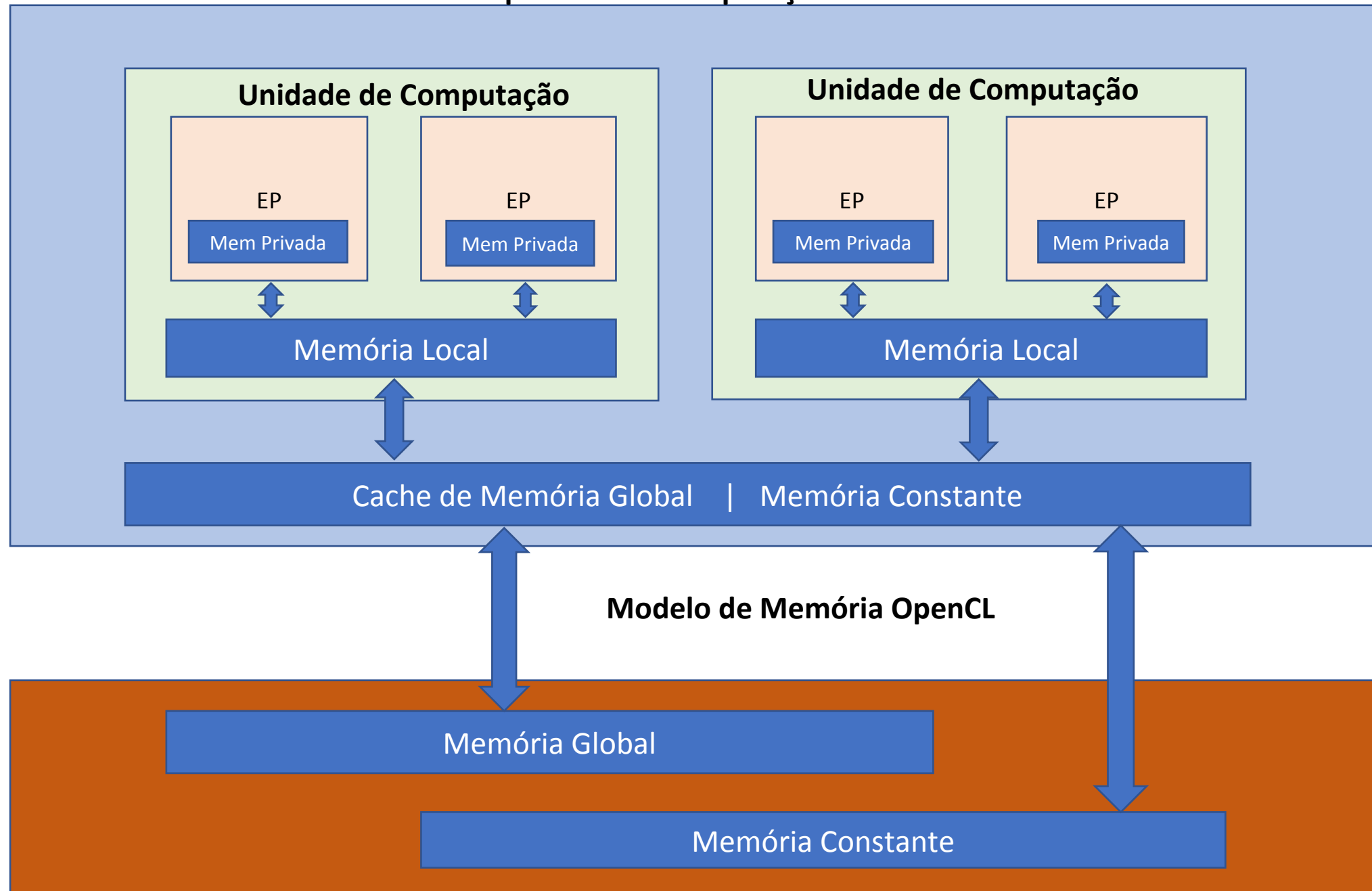


Processador: Cada núcleo tem sua própria memória de instruções.

# Arquitetura conceitual de um device OpenCL, o host não aparece



## Dispositivo de Computação - GPU



## *O dispositivo (device) OpenCL*

- Deve conter várias **Unidades de Computação**, que é composta por vários **Elementos de Processamento**. Como exemplo, a GPU descrita em termos OpenCL é a seguinte.

- Dispositivo OpenCL - GPU
- Unidade de Computação - Vários processadores
- Elementos de processamento – Núcleos de processadores

OpenCL dispositivo será chamado de um dispositivo (*device*) a partir deste ponto em frente.

# Modelo de programação

- O modelo de programação descreve as abordagens possíveis para a execução de código OpenCL. O modelo de programação em OpenCL é semelhante ao utilizado em CUDA.
- *Kernels* podem ser executados de dois modos distintos:
  - **Paralelismo de dados** (Data Parallel): são instanciados múltiplos *work-items* para a execução do *kernel*. Este é o modo descrito até o momento, e consiste no modo principal de uso de OpenCL.
  - **Paralelismo de tarefas** (Task Parallel): um único *work-item* é instanciado para a execução do *kernel*. Isto permite a execução de múltiplos *kernels* diferentes sobre *um mesmo conjunto de dados*, ou sobre *conjuntos de dados distintos*.

# Modelo de Execução

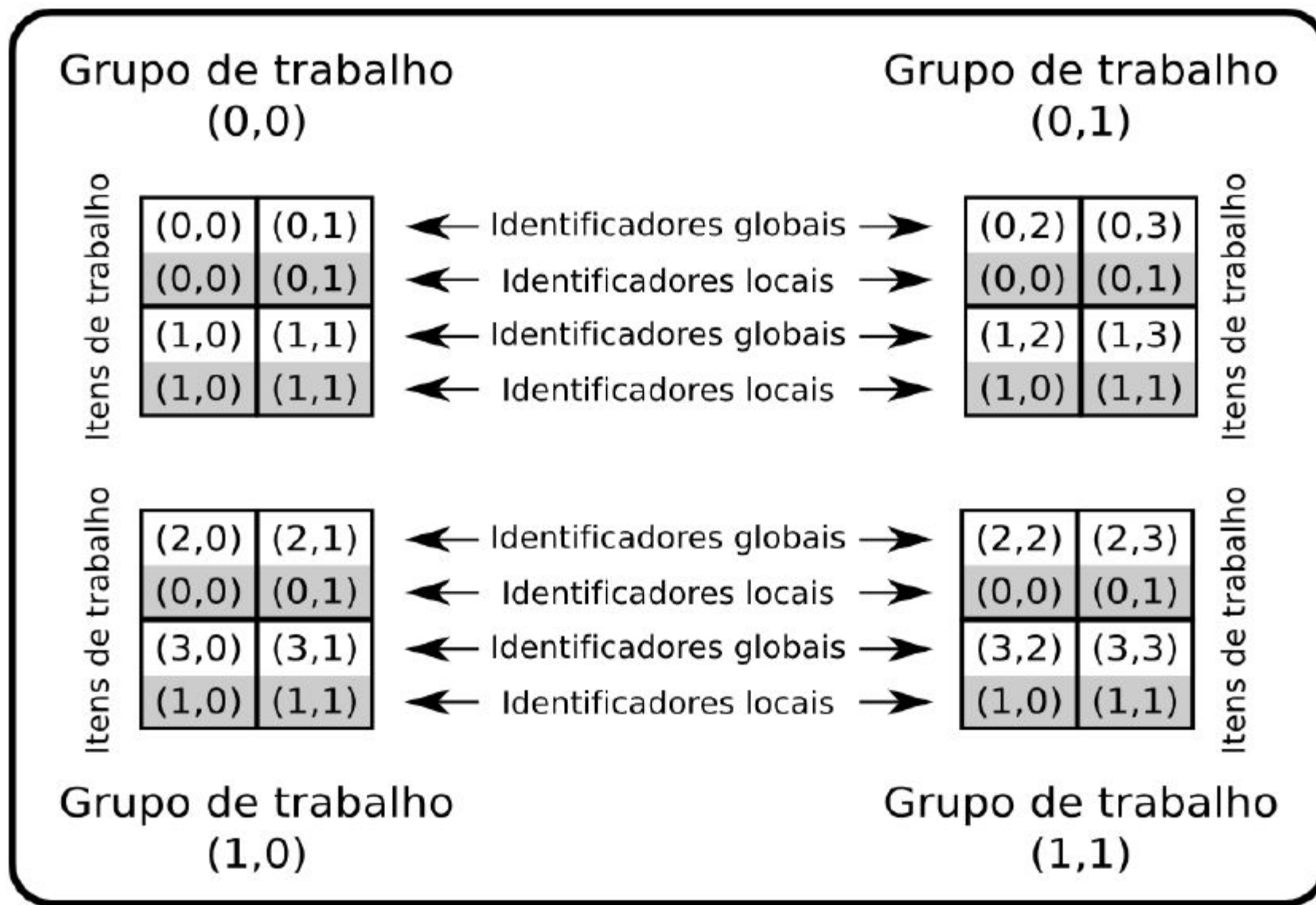
- O modelo de execução do OpenCL é definido em termos de duas unidades de execução distintas: *programa host* e *funções kernel*.
- O programa *host* usa a *API OpenCL* para criar e administrar uma estrutura chamada *OpenCL context*.
- As *funções kernel* são a *parte paralela* de uma aplicação OpenCL. É o código que é executado em um dispositivo (GPU).
- Um *kernel* em execução é referenciado como um *work-item* (*thread*) e um conjunto de *work-items* como um *work-group* (*bloco de threads*).

- O modelo de execução descreve a **instanciação de kernels** e a identificação das instâncias.
- Em OpenCL, um kernel é executado em um espaço de índices de 1, 2 ou 3 dimensões.
- **Cada instância do kernel é denominada item de trabalho (work-item)**, materializado por uma thread, sendo este identificado por uma *tupla de índices*, havendo um índice para cada dimensão do espaço de índices. Estes índices são os **identificadores globais do item de trabalho**.



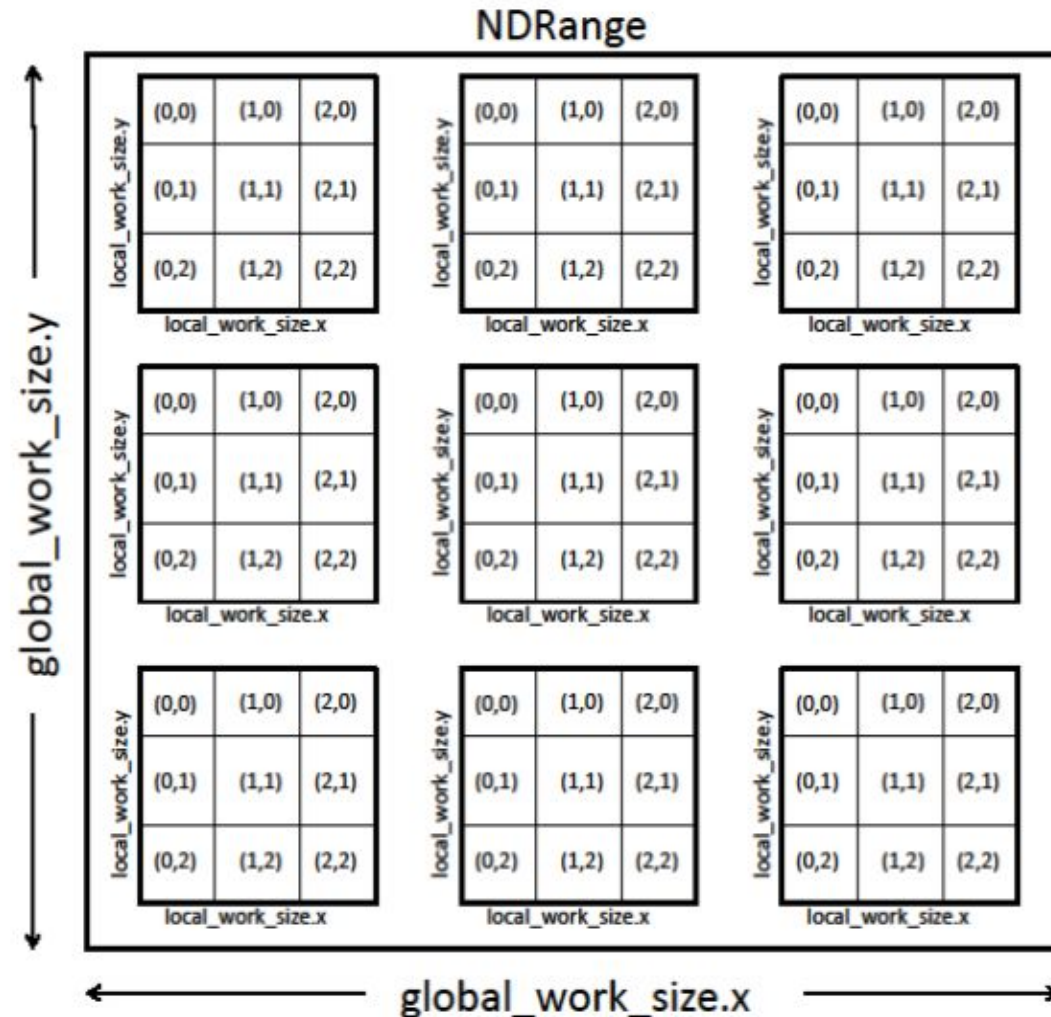
## Um NDRange de duas dimensões

- A Figura 3 ilustram NDRange de duas dimensões, dividido em quatro *work-groups*.
- Cada *work-group* contém quatro *work-items*.
- Observando-se os diferentes índices existentes, pode constar que um *work-item* pode ser identificado individualmente de duas maneiras:
  1. Por meio de seus identificadores globais.
  2. Por meio da combinação de seus identificadores locais e dos identificadores do seu *work-group*.



**Figura 3. Exemplo de espaço de índices de duas dimensões.**

# Organização de Work-groups (blocos) em um NDRange (grid)



# *Identificando work-items*

- Os identificadores de um *work-item* são, em geral, empregados para indexar estruturas que armazenam os dados de entrada e saída de um *kernel*.
- O *espaço de índices* é frequentemente dimensionado de em função do tamanho dos conjuntos de dados a serem processados.
- Assim, por meio de seus identificadores, cada *work-item* pode ser designado responsável por um ponto ou uma parte específica do resultado.

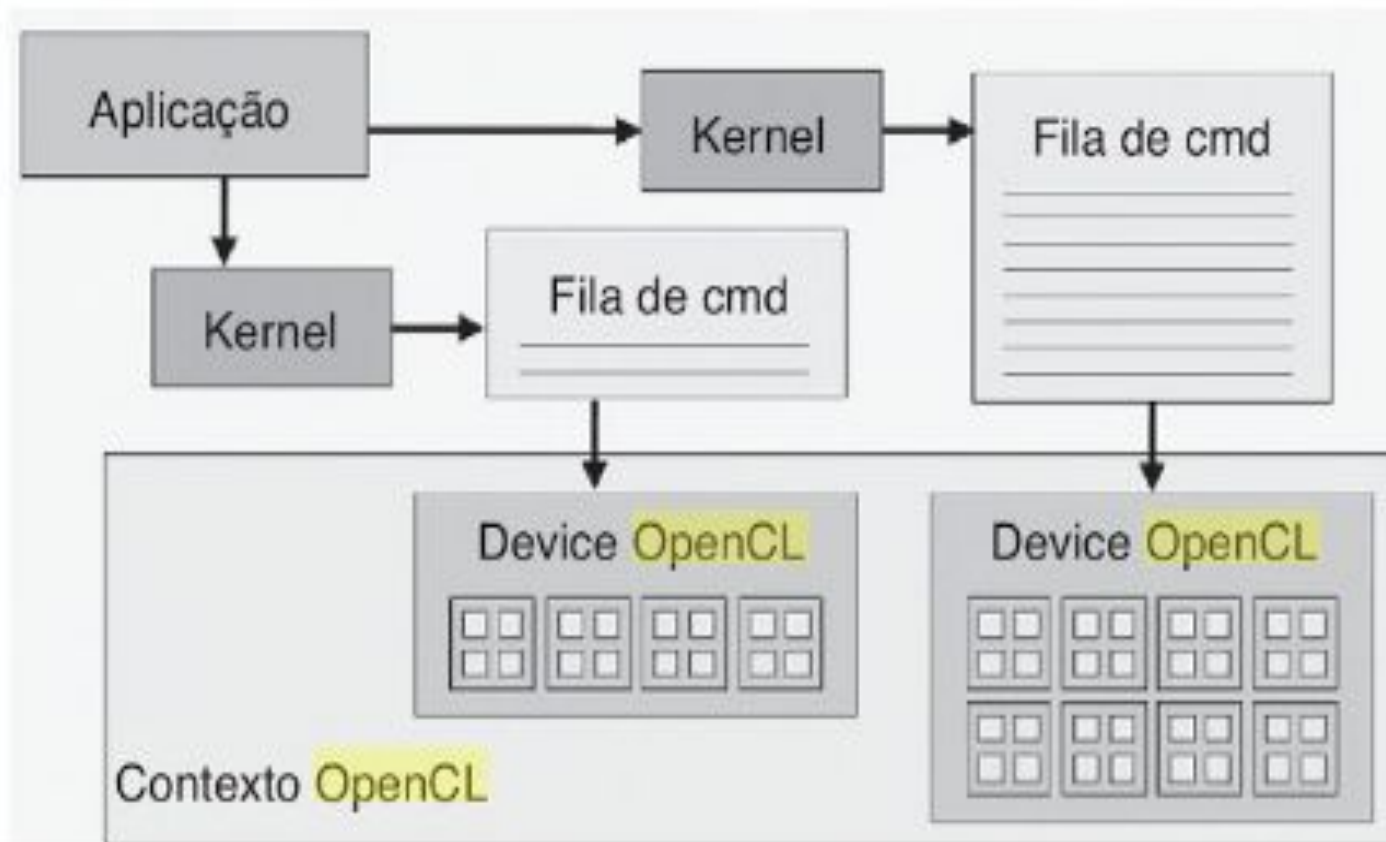
# *Kernels e Contexto*

- A execução de *kernels* em uma aplicação OpenCL só é possível após a definição de um *contexto* (*context*).
- Um contexto engloba um *conjunto de dispositivos* e *kernels*, além de outras estruturas necessárias para a operação da aplicação, como *filas de comandos*, *objetos de programa* e *objetos de memória*.

## *Kernels e filas de comandos*

- Para serem executados, *kernels* são submetidos a *filas de comandos* (*command queues*) associadas aos dispositivos em uso.
- Cada dispositivo (GPU) possui *uma* fila de comandos associada a ela.

# Contexto OpenCL para Device GPUs



# Kernels

- O modelo de execução descreve a **instanciação de kernels** e a identificação das instâncias.
- Em OpenCL, um **kernel** é executado em um espaço de índices de 1, 2 ou 3 dimensões, denominado **NDRange (N-Dimensional Range)**.
- **Cada instância do kernel é denominada *work-item***, sendo este identificado por uma *tupla de índices*, havendo um índice para cada dimensão do espaço de índices.
- Estes índices são os **identificadores globais do *work-item***.



# Modelo de Memória OpenCL

1. **Memória Global** - Memória que pode ser lida de todos os *work-items*. É fisicamente a memória principal do dispositivo (GPU).

2. **Memória Constante** - Também é memória que pode ser lida de todos os *work-items*.

É fisicamente a memória principal do dispositivo, mas pode ser usada de forma mais eficiente do que a memória global, se as unidades de computação contiverem hardware para suportar *cache de memória constante*.

# Memória Constante

- A memória constante não é especificamente projetada para cada tipo de dados somente leitura.
- Em vez disso, para dados em que cada elemento é acessado simultaneamente por todos os [work-items](#).
- Valores do programa que nunca mudam (por exemplo, uma variável de dados contendo o valor de pi) também cai nesta categoria.
- A memória constante é modelada como parte da memória global.
- Os objetos de memória que são transferidos para a memória global podem ser especificados como constantes.
- Os dados são mapeados para memória constante usando a palavra-chave `__constant`

# Modelo de Memória OpenCL

3. **Memória local** - Memória que pode ser lida por *work-items* dentro de um *work-group*. É fisicamente a memória compartilhada por cada das *unidades de computação (UC)*.

4. **Memória privada** - Memória que só pode ser usada dentro de cada *work-item*. É fisicamente os *registradores usados por cada elemento de processamento (EP)*.

# Modelo de Memória OpenCL

- Os 4 tipos de memória mencionados existem todas no *dispositivo*.
- O lado do *host* tem sua própria memória também.
- O *host*, por outro lado, é capaz de ler e escrever para a memória global, constante e do host.
- No entanto, somente o *kernel* pode acessar a memória no próprio *dispositivo*.

# Modelo de Memória para OpenCL

- O programa *host* e os *kernels* podem ler e escrever na memória global.
- A *memória Local* é disponível para todos os *work-items* em um determinado *work-group*.
- A *memória Constante* é disponível somente para leitura pelos *work-items*.
- Cada *work-item* tem sua própria *memória privada*, que não é visível para outros *work-items*.

## *Consistência de memória*

- O estado visível da memória a um item de trabalho pode não ser o mesmo para outros itens de trabalho durante a execução de um kernel.
- Porém, o padrão OpenCL ordena que as seguintes garantias sejam feitas quanto à consistência do acesso à memória:

# Consistência de memória

1. Para um único *work-item*, há consistência de leitura e escrita. Se um *work-item* escrever em uma região de memória e, a seguir, ler a mesma região, o valor lido será aquele que foi escrito.
2. As *memórias global e local* são consistentes entre *work-items* de um mesmo work-group em uma barreira de sincronização.

# Work-Items e Work-Groups

- *Work-items* são organizados em *work-groups*.
- Cada *work-group* também é identificado por uma *tupla de índices*, sendo um índice para cada dimensão do espaço.
- Dentro de um *work-group*, um *work-item* recebe ainda outra *tupla de índices*, os quais constituem os seus *identificadores locais* no *work-group*.
- Memória Local é modelada como sendo compartilhada por um *work-group*.

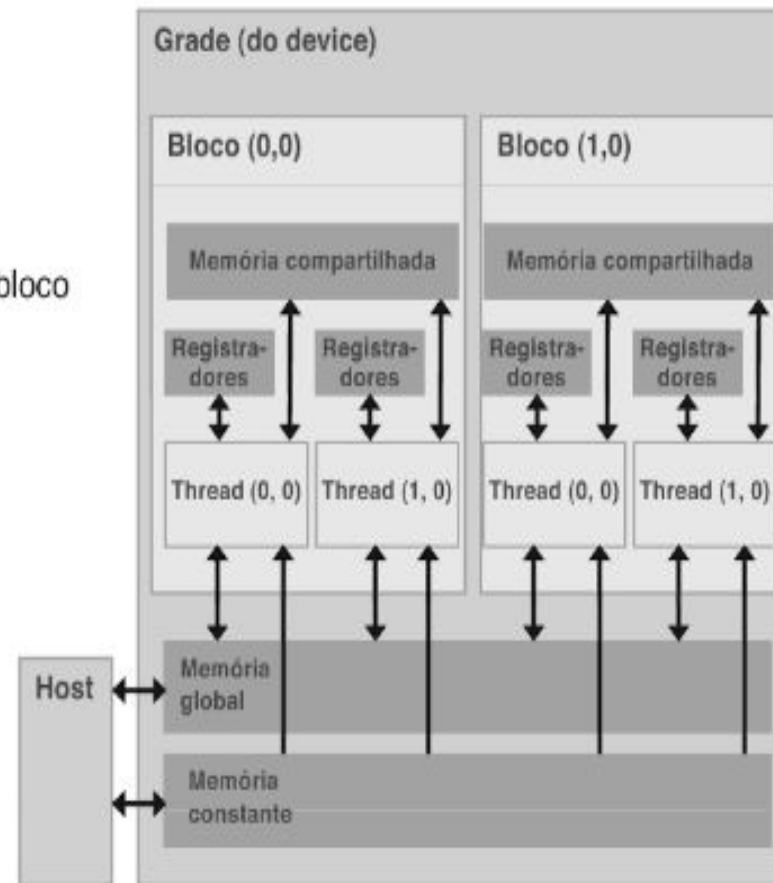


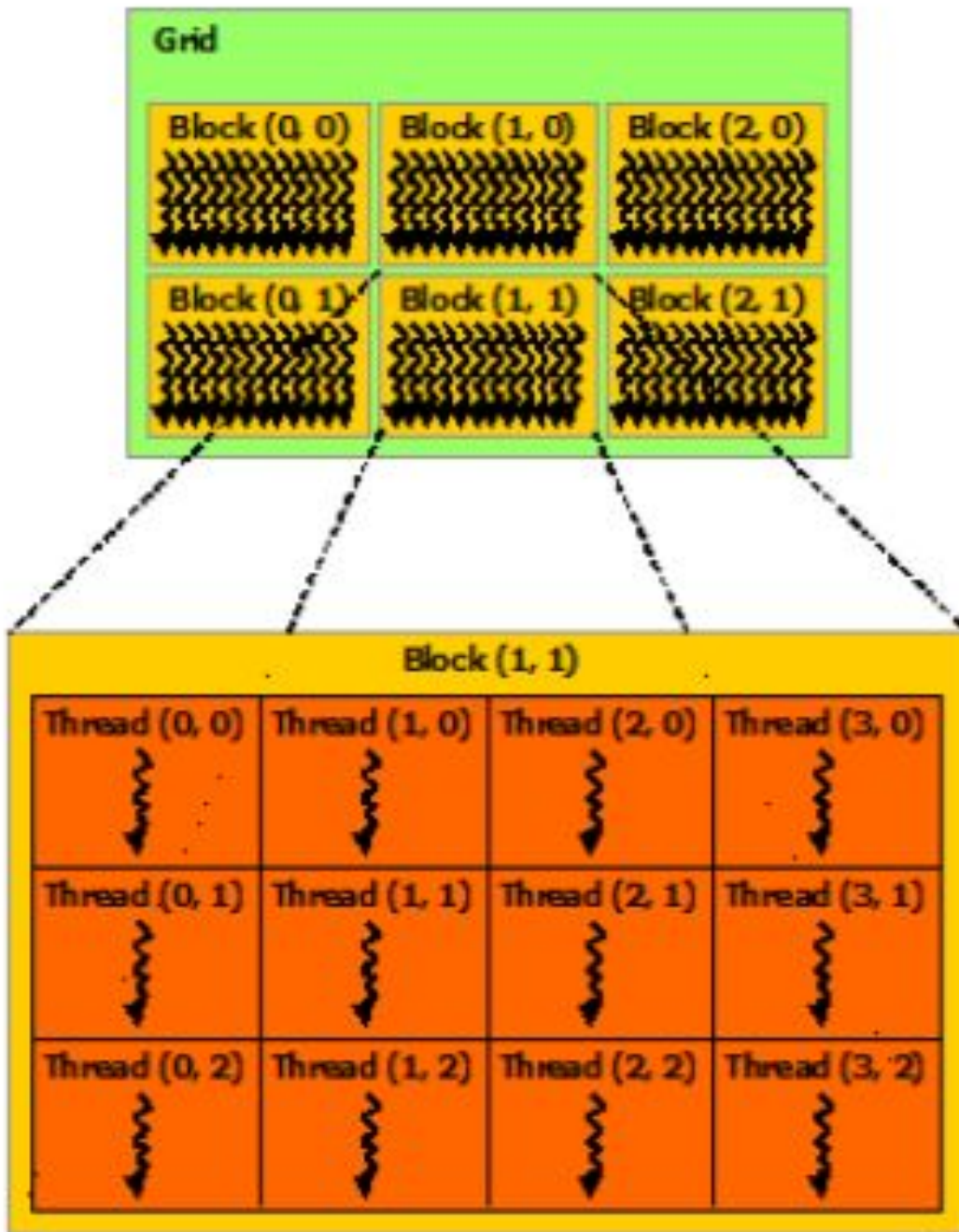
# *Aplicações em OpenCL: Passos*

1. Levantamento dos dispositivos disponíveis , a *criação do contexto de execução, programas e kernels* utilizados pela aplicação, e a *criação da fila de comandos do dispositivo*.
2. *Envio de dados* para o *dispositivo*.
3. A *execução do kernel no dispositivo*.
4. A *leitura dos resultados gerados* pela execução no dispositivo .

# Modelo de Memória CUDA

- O código do device pode:
  - L/E registradores por thread
  - L/E memória local por thread
  - L/E memória compartilhada por bloco
  - L/E memória global por grade
  - Apenas ler memória constante por grade
- O código do host pode:
  - Transferir dados de/para memórias global e constante por grade





Grid e Blocos de threads em CUDA

# *The OpenCL Programming Book*

<https://www.fixstars.com/en/openccl/book/OpenCLProgrammingBook/calling-the-kernel/>

## **4.3 Calling the kernel**

- *Data Parallelism and Task Parallelism*

# | Data Parallelism and Task Parallelism

- O código paralelizável é implementado com:  
"Paralelismo de dados" ou "Paralelismo de Tarefa".
- No OpenCL, a diferença entre os dois é, **se o mesmo kernel ou kernels diferentes** são executados em paralelo.

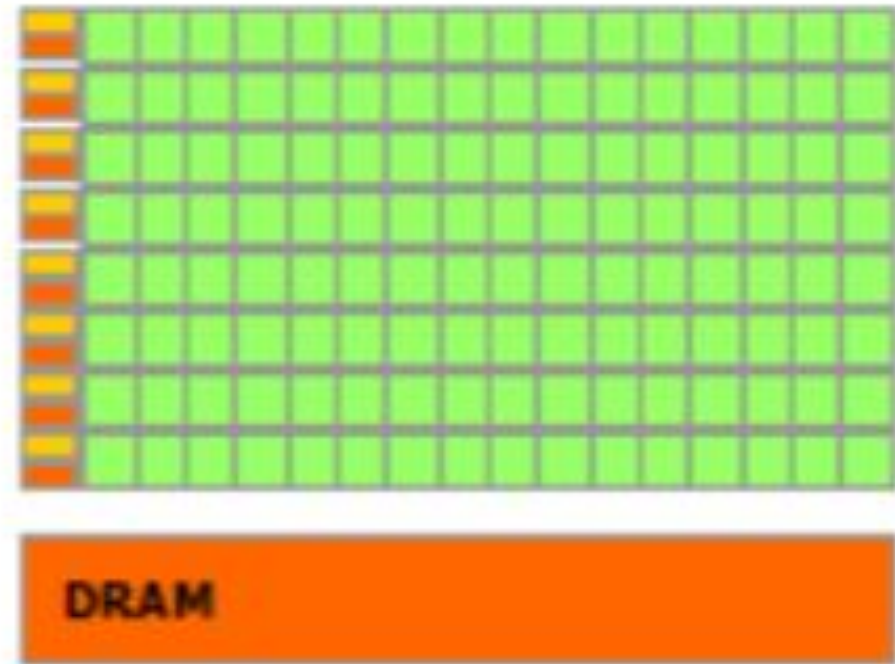
# | Data Parallelism and Task Parallelism

- Atualmente, a maioria das GPUs contém centenas de processadores.

Uma comparação entre a arquitetura de uma CPU com apenas quatro unidades lógica e aritmética (ULAs) e uma GPU com 128 ULAs.



**CPU**



**GPU**

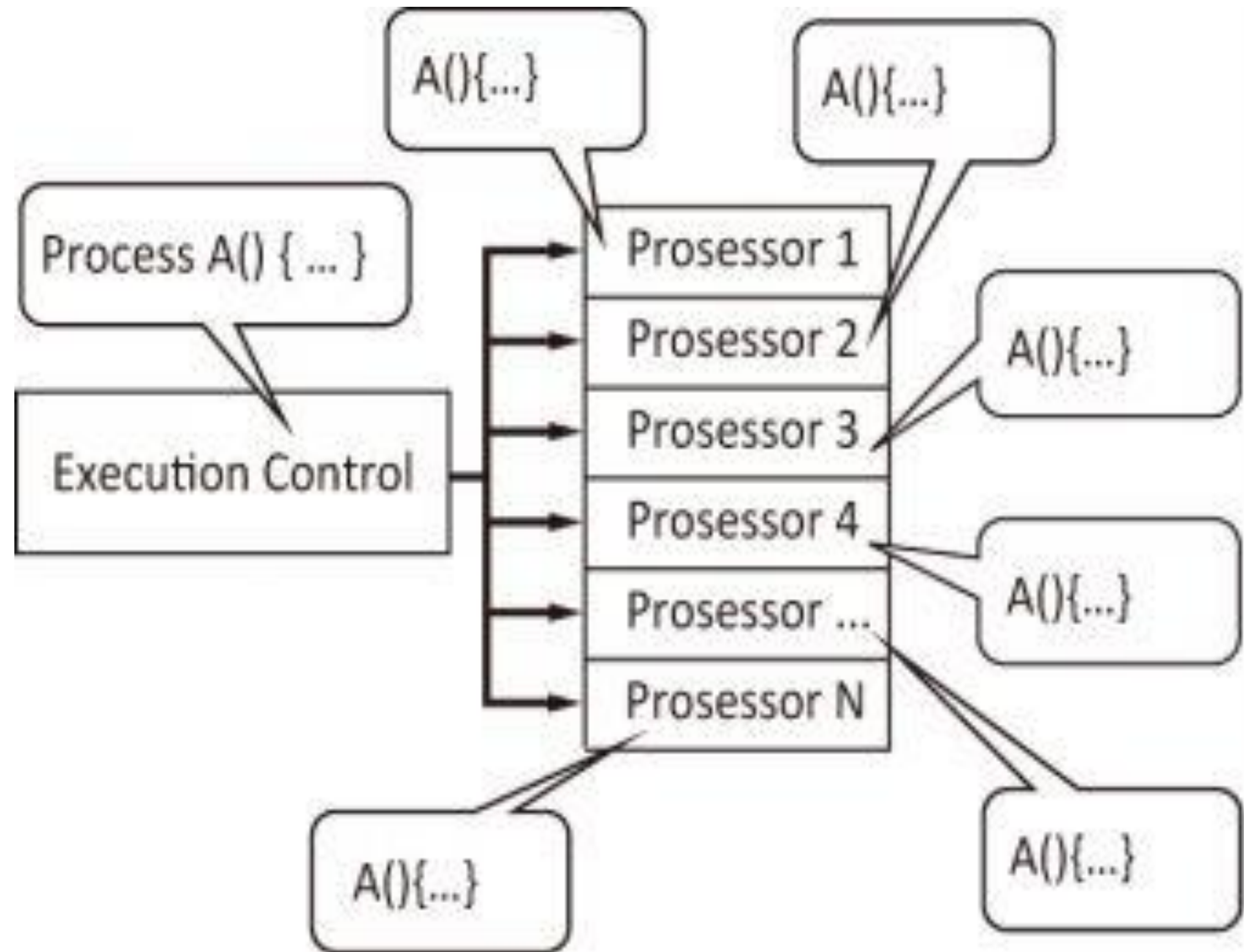
# *Hardware da GPU*

- Em hardware, como a busca de instruções e os contadores de programas, são compartilhados entre os processadores.
- Por esta razão, as GPUs funcionam muito bem para paralelismo de dados (códigos iguais) e são incapazes de executar tarefas de códigos diferentes em paralelo, ao mesmo tempo.
- Ver Figura 4.2 e Figura 4.3



Figure 4.2:  
**Efficient** use of  
the GPU

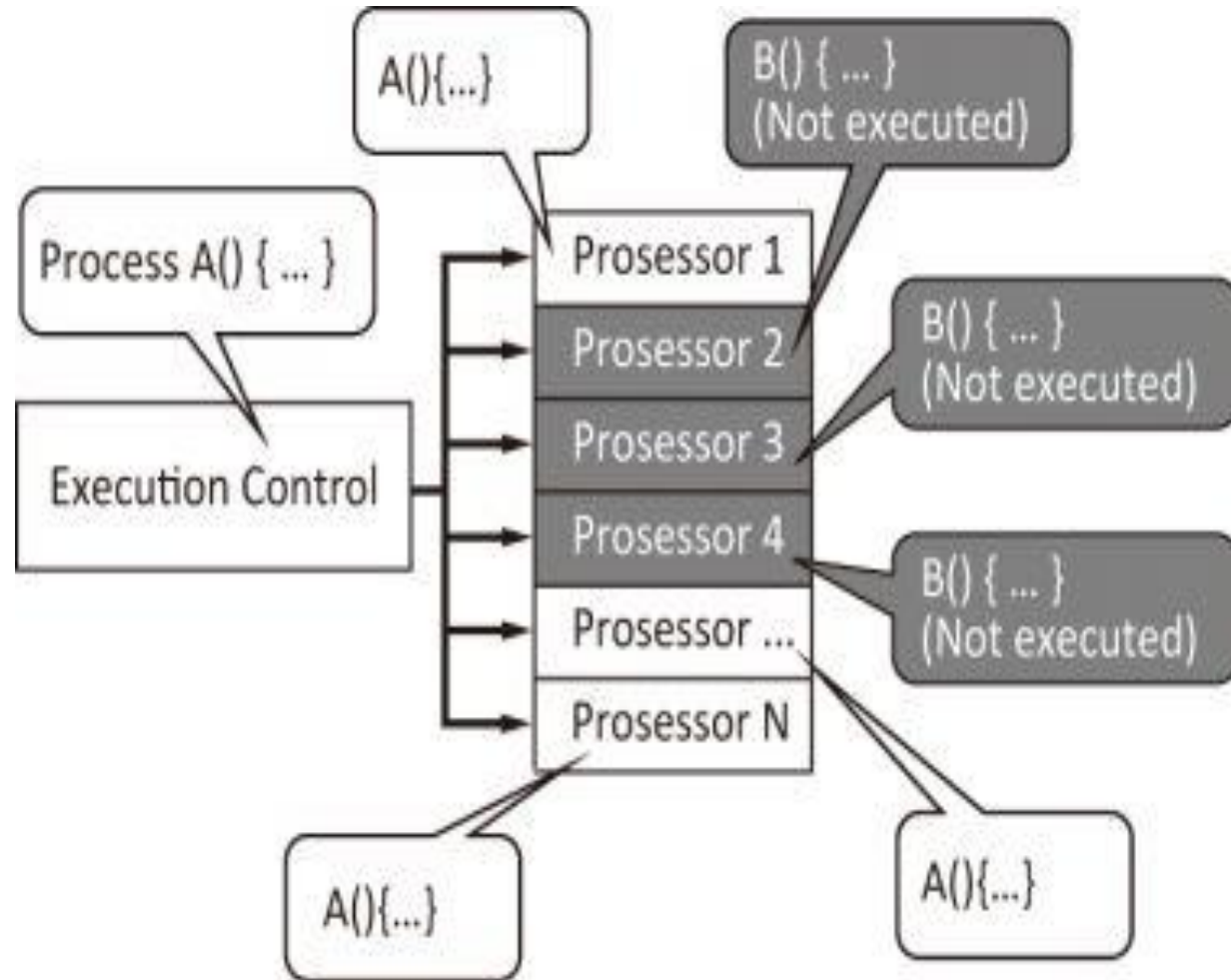
Paralelismo de  
Datos



## *Processadores executam a mesma tarefa*

- Conforme mostrado na Figura 4.2, **quando vários processadores executam a mesma tarefa**, o número de tarefas, igual ao número de processadores, pode ser executado ao mesmo tempo, de uma vez.

Figure 4.3:  
**Inefficient use**  
**of the GPU**  
**Paralelismo de**  
**Dados**



# GPU - Tarefas estão programadas para serem executadas em paralelo

- A Figura 4.3 mostra o caso em que tarefas A e B estão programadas para serem executadas em paralelo na GPU.
- Como processadores só podem processar o mesmo conjunto de instruções nos núcleos (códigos iguais), os processadores agendados para processar a Tarefa B (códigos diferentes de A) devem estar no modo inativo até que a Tarefa A esteja concluída.

## *Cuidar ...*

- Ao se desenvolver uma aplicação, *o tipo de paralelismo e o hardware precisam ser considerados*, e usar a API apropriada.

## | *Data Parallelism and Task Parallelism*

- Para **tarefas paralelas de dados** adequadas para um dispositivo como o GPU, o OpenCL fornece uma API para executar um mesmo kernel em vários processadores.

## | *Data Parallelism and Task Parallelism*

- Esta seção usará a operação aritmética vectorizada para explicar o método básico de implementações para comandos paralelos de dados e comandos paralelos de tarefas.
- O código de exemplo fornecido destina-se a ilustrar os conceitos de paralelização.

## Exemplo

- O código de exemplo executa as operações aritméticas básicas, que são adição, subtração, multiplicação e divisão, entre valores de flutuação (ponto flutuante).
- A visão geral é mostrada na Figura 4.4.



*Figure 4.4:*  
*Basic arithmetic operations between floats*

		A				B			
		1	2	3	4	1	2	3	4
Input	1								
	2								
	3								
	4								
-----									
		C							
		1	2	3	4				
Output	1	$A(1,1) + B(1,1)$	$A(1,2) - B(1,2)$	$A(1,3) \times B(1,3)$	$A(1,4) \div B(1,4)$				
	2	$A(2,1) + B(2,1)$	$A(2,2) - B(2,2)$	$A(2,3) \times B(2,3)$	$A(2,4) \div B(2,4)$				
	3	$A(3,1) + B(3,1)$	$A(3,2) - B(3,2)$	$A(3,3) \times B(3,3)$	$A(3,4) \div B(3,4)$				
	4	$A(4,1) + B(4,1)$	$A(4,2) - B(4,2)$	$A(4,3) \times B(4,3)$	$A(4,4) \div B(4,4)$				

## Na Figura 4.4

- Como mostra a Figura 4.4, os dados de entrada consistem em 2 conjuntos de matrizes 4x4, A e B. Os dados de saída são uma matriz 4x4, C.
- Veja primeiro a [implementação paralela de dados](#) (Lista 4.8, Lista 4.9).
- Este programa trata [cada linha de dados como um \*work-group\*](#) para executar a computação.

# *List 4.8 e List 4.9 – Implementações Kernel e Host*

- **List 4.8**

Data parallel model - **kernel** dataParallel.cl

- **List 4.9**

Data parallel model - **host** dataParallel.c

- **Exemplo – Modelo Paralelo de Dados** *Código Kernel e Código Host*

## *Versão paralela da tarefa*

- Em seguida, você pode ver a versão paralela da tarefa para o mesmo problema (Lista 4.10, Lista 4.11).
- Neste exemplo, *as tarefas são agrupadas de acordo com o tipo de operação aritmética* que está sendo executada.
- **Exemplo – Modelo Paralelo de Tarefas**  
*Código Kernel e Código Host*

## Comentando ... Modelo Paralelo de Dados

- Como você pode ver, os códigos-fonte são muito semelhantes.
- As únicas diferenças estão nos próprios *kernels*, e na *maneira de executar esses kernels*.
- No **modelo de dados em paralelo**, as 4 operações aritméticas são agrupadas como **um conjunto de comandos em um *kernel***.
- Enquanto que no **modelo paralelo de tarefas**, 4 kernels diferentes são implementados para cada tipo de operação aritmética.

# *Modelo Paralelo de Dados*

## *x Modelo Paralelo de Tarefas*

- Pode parecer que, uma vez o **modelo paralelo de tarefas** requer mais código, ele também deve executar mais operações.
- No entanto, **independentemente do modelo que seja utilizado** para este problema, **o número de operações efetuadas pelo dispositivo é realmente o mesmo.**
- Apesar desse fato, **o desempenho pode variar**, escolhendo um ou outro modelo, de modo que o **modelo de paralelização deve ser considerado com sabedoria na fase de planejamento do aplicativo.**

# Modelo de Execução

*Work-Items*

## List 4.8:

### Data parallel model - kernel *dataParallel.cl*

```
1.  __kernel void dataParallel(__global float* A, __global float* B,  
                               __global float* C)  
2.  {  
3.      int base = 4*get_global_id(0);  
4.      C[base+0] = A[base+0] + B[base+0];  
5.      C[base+1] = A[base+1] - B[base+1];  
6.      C[base+2] = A[base+2] * B[base+2];  
7.      C[base+3] = A[base+3] / B[base+3];  
8.  }
```



# *Passeando no código-fonte do Modelo Paralelo de dados*

1. `__kernel void dataParallel( __global float * A, __global float * B,  
__global float * C )`

- Quando o paralelismo de dados é enfileirada, os *work-items* são criados.
- Cada um desses *work-items* executa o mesmo kernel em paralelo.

# *Obtendo ID de item de trabalho*

3. `int base = 4*get_global_id(0);`

- A instrução OpenCL, `get_global_id (0)` obtém o *ID do work-item global*, que é usado para decidir quais os dados a processar, de modo que cada *work-item* possa processar diferentes conjuntos de dados em paralelo.

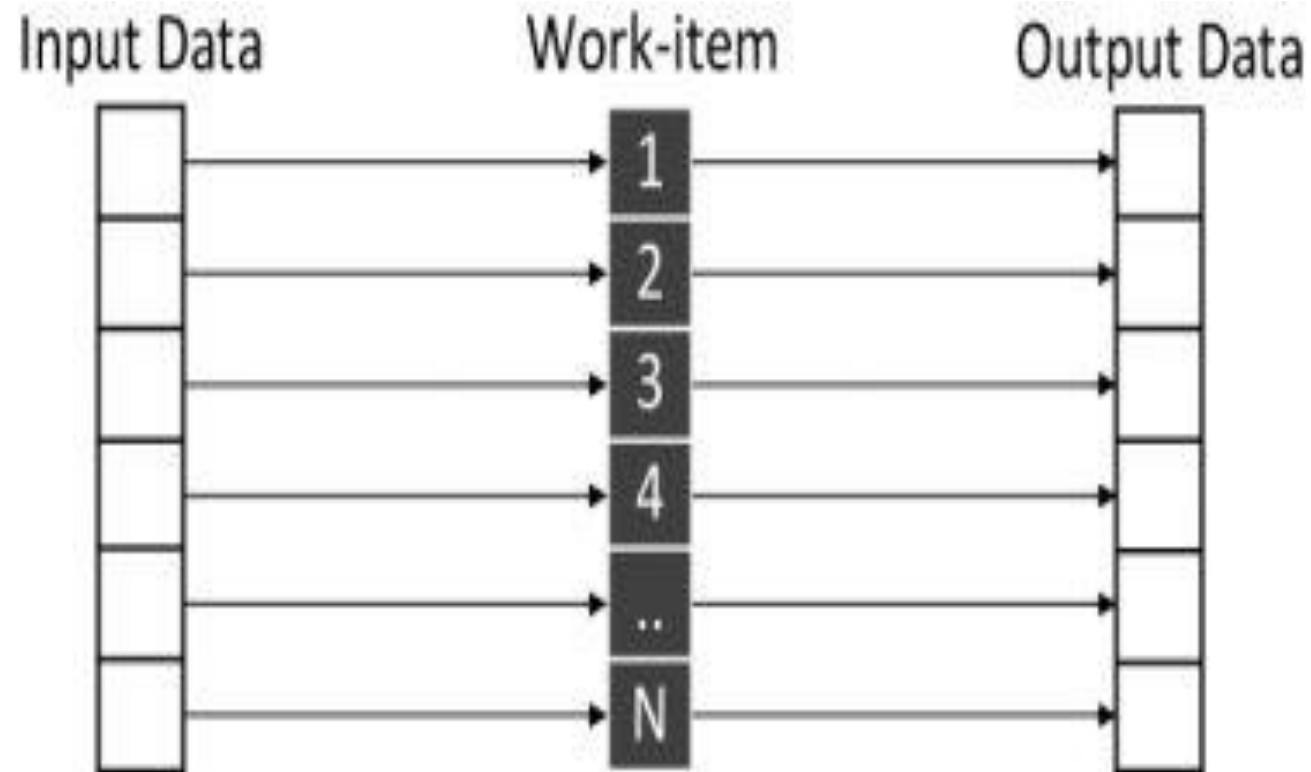
# *Etapas do processamento paralelo de dados*

Em geral, o **processamento paralelo de dados** é feito usando as seguintes etapas.

1. Get *work-item ID*
2. Processa o subconjunto de dados correspondendo ao *work-item ID*.

Um diagrama de bloco do processo é mostrado em Figura 4.5.

*Figure 4.5:*  
*Block diagram of the **data-parallel model** in relation to **work-items***



## Exemplo – Figura 4.5

- Neste exemplo, o *work-item global* é multiplicado por 4 e armazenado na variável "base".
- Esse valor é usado para decidir qual elemento da matriz A e B é processado.
  1.  $C[\text{base}+0] = A[\text{base}+0] + B[\text{base}+0];$
  2.  $C[\text{base}+1] = A[\text{base}+1] - B[\text{base}+1];$
  3.  $C[\text{base}+2] = A[\text{base}+2] * B[\text{base}+2];$
  4.  $C[\text{base}+3] = A[\text{base}+3] / B[\text{base}+3];$

## *Exemplo – Figura 4.5*

- Como cada *work-item* tem IDs diferentes, a variável "*base*" também tem um valor diferente para cada *work-item*, o que impede que os *work-items* processem os mesmos dados.
- Desta forma, *grande quantidade de dados* podem ser processados simultaneamente.

## *Exemplo – Figura 4.5*

- Discutimos que muitos *work-items* são criados, mas não abordamos como decidir o número de *work-items* a serem criados.
- Isso é feito no segmento de código a seguir do *código do host*.

## *Exemplo – Figura 4.5*

1. `size_t global_item_size = 4;`
2. `size_t local_item_size = 1;`
- 3.
4. `/* Execute OpenCL kernel as data parallel */`
5. `ret = clEnqueueNDRangeKernel(command_queue, kernel, 1,`  
`NULL,`
6. `&global_item_size, &local_item_size, 0, NULL, NULL);`



## Exemplo – Figura 4.5

- O `clEnqueueNDRangeKernel ()` é um comando OpenCL API usado para enfileirar tarefas paralelas de dados.
- Os argumentos 5 e 6 determinam o tamanho do *work-item*. Nesse caso, o `global_item_size` é definido como 4 e o `local_item_size` é definido como 1.
- As etapas gerais são resumidas da seguinte maneira.
  1. Criar itens de trabalho no host
  2. Processar dados correspondentes a ID do item de trabalho global no kernel

# *The source code for the task parallel model*

- Neste modelo, diferentes kernels podem ser executados em paralelo.
- Note que diferentes kernels são implementados para cada uma das 4 operações aritméticas.

```
1.  /* Execute OpenCL kernel as task parallel */  
2.  for (i=0; i < 4; i++) {  
3.      ret = clEnqueueTask(command_queue, kernel[i], 0, NULL, NULL);  
4.  }
```

- O segmento de código acima **enfileira os 4 kernels**.

# *OpenCL - Processo paralelo de tarefas*

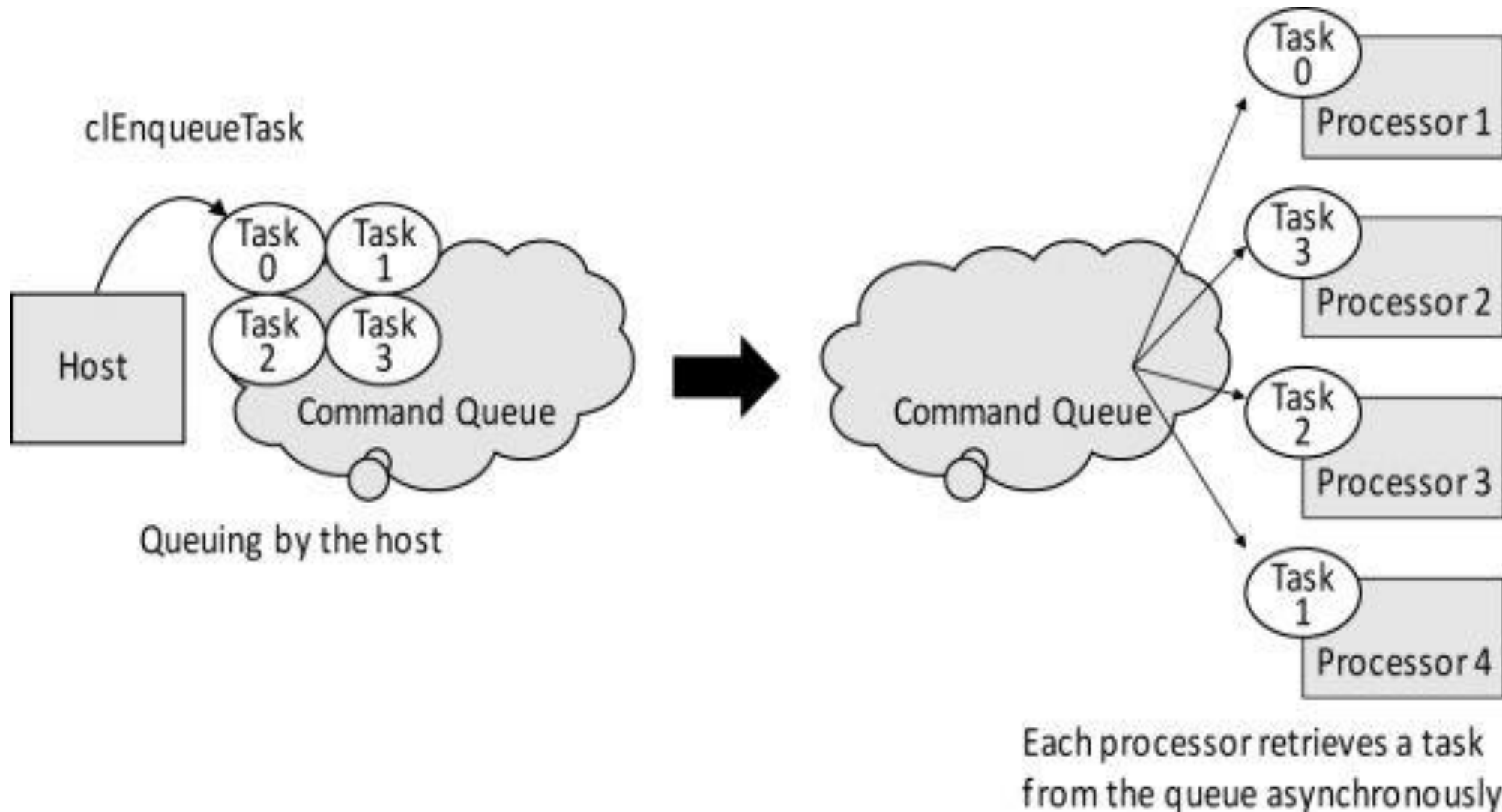
- Em OpenCL, para executar um **processo paralelo de tarefas**, o **modo fora de ordem** deve ser ativado quando a fila de comandos é criada.
- Usando esse modo, **a tarefa enfileirada não aguarda até que a tarefa anterior seja concluída**, se houver **unidades de computação inativas disponíveis** que possam estar executando essa tarefa.

# *Criando fila de comandos e execução paralela*

```
1.  /* Create command queue */  
2.  command_queue = clCreateCommandQueue(context,  
device_id,  
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &ret);
```

- O diagrama de blocos da natureza das filas de comandos e execução paralela são mostrados na Figura 4.6.

*Figure 4.6:  
Command queues and parallel execution*



## *Figure 4.6: Command queues and parallel execution*

- O `clEnqueueTask ()` é usado como um exemplo na Figura 4.6, mas um processamento paralelo similar poderia ocorrer para outras combinações de *enqueue-funções*, como `clEnqueueNDRangeKernel ()`, `clEnqueueReadBuffer ()` e `clEnqueueWriteBuffer ()`.

## ***Figure 4.6: Command queues and parallel execution***

- Por exemplo, uma vez que o PCI Express suporta transferências simultâneas de memória bidirecional, enfileirar os comandos

`clEnqueueReadBuffer ()` e `clEnqueueWriteBuffer ()`

podem executar comandos de leitura e gravação simultaneamente, visto que os comandos estão sendo executados por diferentes processadores.

## ***Figure 4.6:***

### ***Command queues and parallel execution***

- No diagrama acima da Figura 4.6, podemos esperar que as 4 tarefas sejam executadas em paralelo, uma vez que elas estão sendo enfileiradas em uma fila de comandos que tem o parâmetro *out-of-execution* habilitado.



# | Modelo de Execução

*Work*

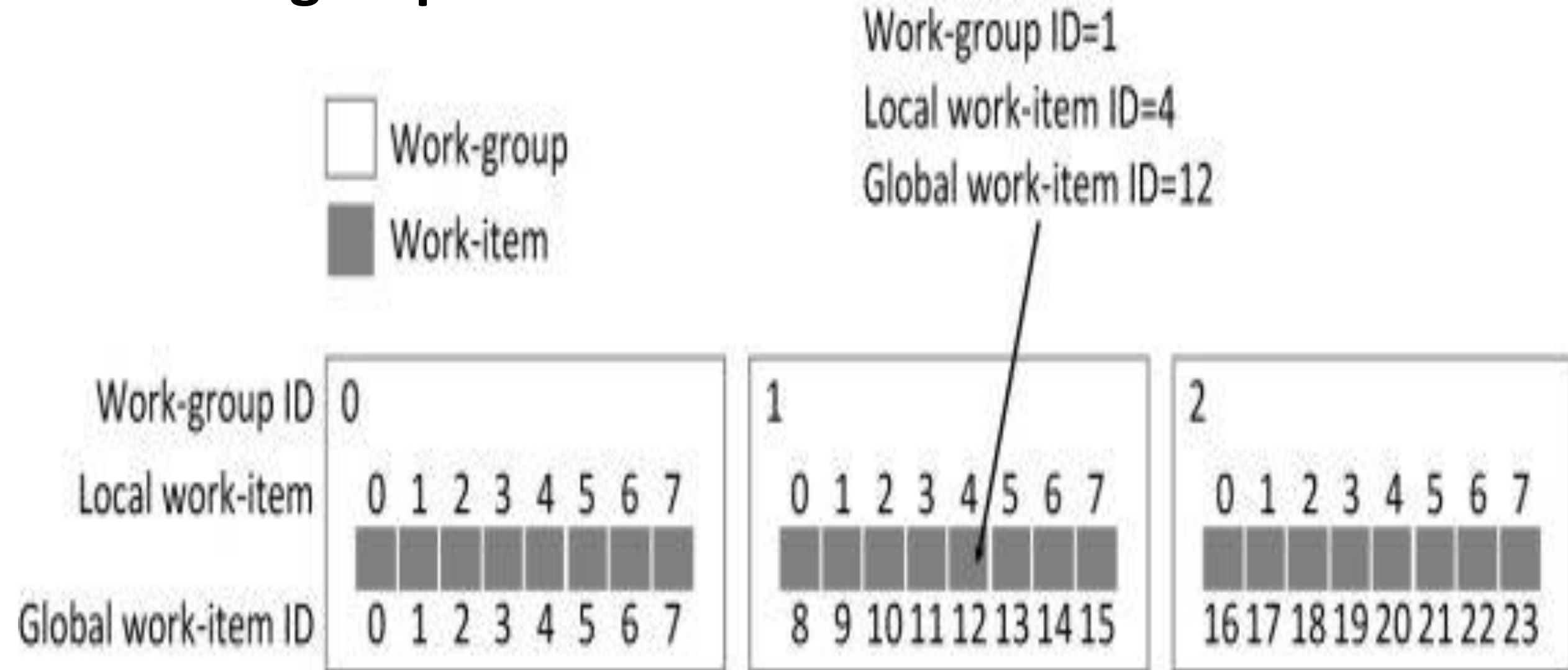
*Group*

# Work-group

- *Work-items* (thread) são organizados em *work-groups* (bloco de threads).
- Cada *work-group* também é identificado por uma tupla de índices, com um índice para cada dimensão do espaço.
- Dentro de um *work-group*, um *work-item* recebe ainda outra tupla de índices, os quais constituem os seus identificadores locais no grupo de trabalho.

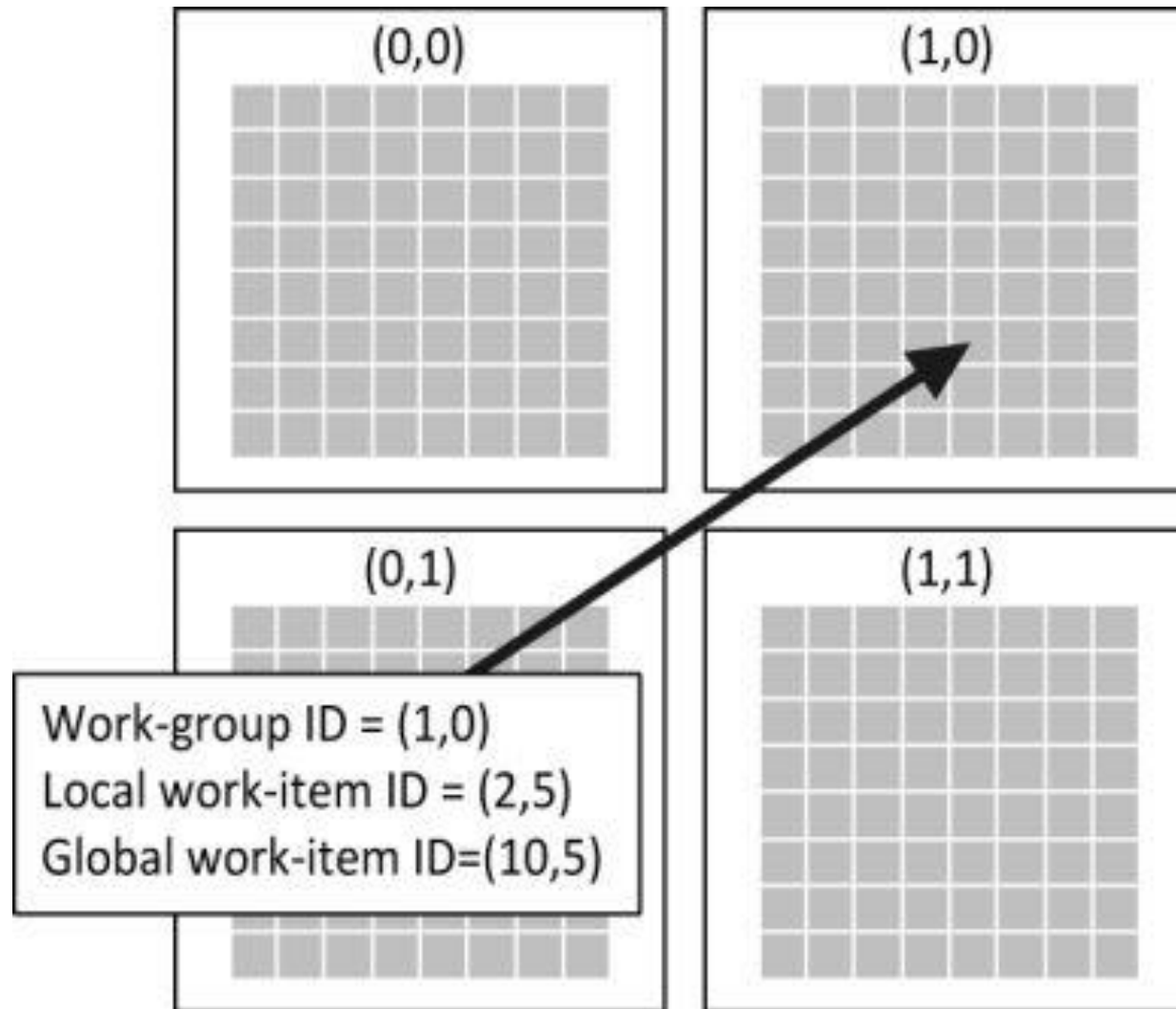
**Figure 4.7**

**Work-group ID and Work-item ID**



**Figure 4.8**

**Work-group and work-item defined in 2-D**



# Table 4.1

## Functions used to retrieve the ID's

Function	Retrieved value
get_group_id	Work-group ID
get_global_id	Global work-item ID
get_local_id	Local work-item ID

## *The ID's of the work-item in Figure 4.8*

Call	Retrieved ID
get_group_id(0)	1
get_group_id(1)	0
get_global_id(0)	10
get_global_id(1)	5
get_local_id(0)	2
get_local_id(1)	5

**Table 3.1: Include header location (as of March 2010)**

OpenCL implementation	Include Header
AMD	CL/cl.h
Apple	OpenCL/ocl.h
FOXC	CL/cl.h
NVIDIA	CL/cl.h

The sample code defines the following macro so that the header is correctly included in any environment.

**List 3.1: Include Header Location (As of March, 2010)**

```
1.  #ifdef __APPLE__
2.  #include <OpenCL/ocl.h>
3.  #else
4.  #include <cl.h>
5.  #endif
```

# Objetos de Memória (Memory Objects)

- Buffers
- Objetos de memória possuem associados a si um tamanho, além de um conjunto de parâmetros que definem se a região de memória associada ao objeto é, por exemplo, somente leitura, ou se encontra mapeada em uma região de memória do hospedeiro.



# *Objetos de programa (program objects)*

- *Kernels* são gerados a partir de **objetos de programa** (program objects).
- Um **objeto de programa** encapsula o *código-fonte de um kernel*, sendo este identificado no código-fonte por meio da palavra-chave `__kernel`.

# Comandos OpenCL

**Algumas funções da API OpenCL para  
o hospedeiro (host)**

As funções estão agrupadas de acordo com os objetos OpenCL aos quais se referem.

## Comandos de cabeçalho

```
#include <stdio.h>
#include <stdlib.h>

#ifdef __APPLE__
#include <OpenCL/opencl.h>
#else
#include <CL/opencl.h>
#endif
#define ARRAY_LENGTH 1000
```

# Programa principal

```
int main(int argc, char** argv)
{
    . . . . .

    . . . . .

    . . . . .

    return 0
}
```

## Variáveis para armazenamento de referências a objetos OpenCL

```
cl_platform_id platformId;  
cl_device_id deviceId;  
cl_context context;  
cl_command_queue queue;  
cl_program program;  
cl_kernel kernel;  
cl_mem bufA;  
cl_mem bufB;  
cl_mem bufC;
```

## ***/\* Variáveis diversas da aplicação \*/***

```
/* ponteiros para inteiros */
```

```
int* hostA;
```

```
int* hostB;
```

```
int* hostC;
```

*/\* The dimensions of the NDRange are specified as an N-element array of type `size_t`, where N represents the number of dimensions used to describe the *work-item* being created. \*/*

*/\* In the vector addition example, our data will be one-dimensional and assuming that there are 1024 elements, the size can be specified as a one-, two-, or three-dimensional vector. The host code to specify an ND Range for 1024 elements is as follows:*

```
size_t indexSpaceSize[3] = {1024, 1, 1}; */
```

```
size_t globalSize[1] = { ARRAY_LENGTH };
```

```
int i;
```

## ***/\* Código-fonte do kernel \*/***

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

# Obtenção de identificador de plataforma

## Argumentos:

```
cl_int  clGetPlatformIDs (cl_uint  num_entries,  
                          cl_platform_id* platforms,  
                          cl_uint*  num_platforms)
```

1. **num\_entries**: número de plataformas desejadas.
2. **platforms**: local onde os identificadores das plataformas encontradas devem ser escritos.
3. **num\_platforms**: número de plataformas encontradas.



## Obtenção de identificador de plataforma

```
/* Será solicitada uma GPU. */
```

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

**Caso seja NULL, o argumento será ignorado.**

# Descoberta de dispositivos

## Obtenção de identificador de dispositivo GPU

```
cl_int  clGetDeviceIDs (cl_platform_id platform,  
                        cl_device_type device_type,  
                        cl_uint num_entries,  
                        cl_device_id* devices,  
                        cl_uint* num_devices)
```

1. **platform**: identificador de plataforma.
2. **device\_type**: tipo de dispositivo. Os seguintes valores são válidos:
  - CL\_DEVICE\_TYPE\_CPU: processador do hospedeiro.
  - CL\_DEVICE\_TYPE\_GPU: dispositivo gráfico.
  - CL\_DEVICE\_TYPE\_ACCELERATOR: processador OpenCL dedicado.
  - CL\_DEVICE\_TYPE\_DEFAULT: dispositivo OpenCL padrão do sistema.
  - CL\_DEVICE\_TYPE\_ALL: todos os dispositivos OpenCL do sistema.
3. **num\_entries**: número de dispositivos desejados.
4. **devices**: array onde serão armazenados os identificadores dos dispositivos encontrados.
5. **num\_devices**: número de dispositivos encontrados que correspondem aos tipos indicados em **device\_type**.

Caso seja NULL, o argumento será ignorado.

## Descoberta de dispositivos

### Obtenção de identificador de dispositivo GPU

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

O segundo argumento pode ser **CL\_DEVICE\_TYPE\_CPU**, caso só se tenha uma CPU.

O quinto argumento é NULL, portanto, ignorado.

# Criação de Contexto

```
cl_context clCreateContext(  
    const cl_context_properties* properties,  
    cl_uint num_devices,  
    const cl_device_id* devices,  
    NULL, NULL, NULL )
```

## Criação de Contexto

- **properties**: lista de propriedades para o contexto.
- **num\_devices**: número de dispositivos para o contexto.
- **devices**: lista de identificadores de devices. Deve possuir tantos identificadores quantos indicados em `num_devices`.

## Criação de Contexto

- Para tornar possível o acesso a dispositivos OpenCL, é necessário que o hospedeiro primeiro inicialize e configure um contexto.

```
context = clCreateContext(0, 1, &deviceId,  
                          NULL, NULL, NULL) ;
```

- Caso seja NULL, o argumento será ignorado.

## Criação da fila de comandos para o dispositivo encontrado

```
cl_command_queue clCreateCommandQueue(  
    cl_context context,  
    cl_device_id device,  
    cl_command_queue_properties properties,  
    cl_int* errcode_ret)
```

- A função `clCreateCommandQueue()` cria uma fila de comandos para um dispositivo específico.



# Criação da fila de comandos para o dispositivo encontrado

1. **context**: contexto OpenCL.
2. **device**: identificador do dispositivo que será associado à fila.
3. **properties**: propriedades da fila de comandos.
4. **errcode\_ret**: local para armazenamento do código de erro da chamada.  
Pode ser NULL.

## Criação da fila de comandos para o dispositivo encontrado

```
queue = clCreateCommandQueue (  
    context,  
    deviceId,  
    0,  
    NULL) ;
```

# Compilação de kernels

- São necessário três passos para a compilação de um ou mais kernels em uma aplicação OpenCL:
  1. Criação de um objeto de programa.
  2. Compilação do programa para um ou mais dispositivos.
  3. Criação de um ou mais kernels a partir do programa compilado.

## Criação de um objeto de programa a partir do código-fonte armazenado na string

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

## Criação de um objeto de programa a partir do código-fonte armazenado na string

**context:** contexto OpenCL.

**count:** número de *strings* no *array*.

**strings:** *array* de *strings* do código-fonte.

**lengths:** *array* contendo os tamanhos das *strings* do *array*.

Pode ser NULL caso as *strings* sejam terminadas em \0.

**errcode\_ret:** local para armazenamento do código de erro da chamada.

Pode ser NULL.





# Compilação do programa para todos os dispositivos do contexto

1. **program**: objeto de programa.
2. **num\_devices**: número de dispositivos para o qual o programa deve ser compilado. O valor 0 causa a compilação para todos os dispositivos presentes no contexto.
3. **device\_list**: lista de dispositivos para os quais o programa deve ser compilado. **NULL indica que o programa deve ser compilado para todos os dispositivos presentes no contexto.**



## Compilação do programa para todos os dispositivos do contexto

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

# Obtenção de um kernel a partir do programa compilado

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

# Obtenção de um kernel a partir do programa compilado

1. **program**: objeto de programa.
2. **kernel\_name**: nome de função **\_\_kernel** definida no código-fonte.
3. **errcode\_ret**: local para armazenamento do código de erro da chamada.  
Pode ser **NULL**.

## Obtenção de um kernel a partir do programa compilado

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

## Alocação dos arrays no host

```
hostA = (int*) malloc (ARRAY_LENGTH * sizeof(int)) ;
```

```
hostB = (int*) malloc (ARRAY_LENGTH * sizeof(int)) ;
```

```
hostC = (int*) malloc (ARRAY_LENGTH * sizeof(int)) ;
```

## Inicialização dos arrays no host

```
for (i = 0; i < ARRAY_LENGTH; ++i)
{
    hostA[i] = rand() % 101 - 50;
    hostB[i] = rand() % 101 - 50;
}
```

# Manipulação de buffers OpenCL

- A comunicação do **host** com a **memória global** de dispositivos OpenCL é realizada por meio de **objetos de memória**.
- Um dos tipos de objetos de memória é o **buffer**, que corresponde a uma região contígua de memória, com tamanho fixo.

# Manipulação de buffers OpenCL

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



## Manipulação de buffers OpenGL

1. **context**: contexto OpenGL.
2. **flags**: especificação de informações sobre a alocação e o uso da região de memória associada ao buffer.  
A flag `CL_MEM_READ_ONLY` cria um buffer somente-leitura.  
A flag `CL_MEM_READ_WRITE` especifica a criação de um buffer para operações de leitura e escrita de dados. Ver especificação para demais flags permitidas.
3. **size**: tamanho, em bytes, do buffer a ser alocado.
4. **host\_ptr**: ponteiro para dados de inicialização do buffer.
5. **errcode\_ret**: local para armazenamento do código de erro da chamada.  
Pode ser **NULL**.

## Criação dos objetos de memória para comunicação com a memória global do dispositivo encontrado

```
bufA = clCreateBuffer(context, CL_MEM_READ_ONLY,  
ARRAY_LENGTH * sizeof(int), NULL, NULL);
```

```
bufB = clCreateBuffer(context, CL_MEM_READ_ONLY,  
ARRAY_LENGTH * sizeof(int), NULL, NULL);
```

```
bufC = clCreateBuffer(context, CL_MEM_READ_WRITE,  
ARRAY_LENGTH * sizeof(int), NULL, NULL);
```

## Transferência dos arrays de entrada para a memória do dispositivo

```
cl_int  clEnqueueWriteBuffer( cl_queue queue,  
                              cl_mem  buffer,  
                              cl_bool  blocking_write,  
                              size_t  offset,  
                              size_t  cb,  
                              void*   ptr,  
                              cl_uint  events_in_wait_list,  
                              NULL,  
                              NULL )
```

## Transferência dos arrays de entrada para a memória do dispositivo

1. **queue**: fila de comandos.
2. **buffer**: objeto de memória do tipo *buffer*.
3. **blocking\_write**: caso CL\_TRUE, a chamada é bloqueante e o hospedeiro suspende a execução até que os dados tenham sido completamente transferidos para o dispositivo. Caso CL\_FALSE, a chamada é não-bloqueante e a execução prossegue assim que o comando é enfileirado.
4. **offset**: *offset* a partir do qual os dados devem ser transferidos.
5. **cb**: comprimento, em bytes, dos dados a serem transferidos.
6. **ptr**: ponteiro para a região de memória do host onde os dados a serem transferidos estão localizados.
7. **events\_in\_wait\_list**: número de eventos na lista de eventos que devem ser aguardados antes do início da transferência dos dados.
8. **NULL**
9. **NULL**

## Transferência dos arrays de entrada para a memória do dispositivo

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

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

## Configuração dos argumentos do kernel

```
cl_int clSetKernelArg(cl_kernel kernel,  
                      cl_uint arg_index,  
                      size_t arg_size,  
                      const void* arg_value)
```

## Configuração dos argumentos do kernel

1. **kernel**: kernel cujo argumento deve ser configurado.
2. **arg\_index**: posição do argumento, de acordo com a ordem em que os argumentos foram definidos no código-fonte, iniciando em 0.
3. **arg\_size**: comprimento dos dados do argumento.
4. **arg\_value**: ponteiro para dados do argumento.

## Configuração dos argumentos do kernel

```
clSetKernelArg(kernel, 0, sizeof(cl_mem), &bufA);
```

```
clSetKernelArg(kernel, 1, sizeof(cl_mem), &bufB);
```

```
clSetKernelArg(kernel, 2, sizeof(cl_mem), &bufC);
```



## Envio do kernel para execução

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

## Envio do kernel para execução

1. **command\_queue**: fila de comandos.
2. **kernel**: kernel a ser executado.
3. **work\_dim**: número de dimensões do espaço de índices. São permitidos os valores 1, 2 e 3.
4. **global\_work\_offset**: array de deslocamentos para valores dos índices em cada dimensão. Por exemplo, um deslocamento de 10 na dimensão 0 fará com que os índices naquela dimensão iniciem em 10.
5. **global\_work\_size**: array contendo os tamanhos para cada dimensão do espaço de índices.
6. **local\_work\_size**: array de tamanhos dos grupos de trabalho em cada dimensão.

## Envio do kernel para execução

Caso NULL, o runtime determina os tamanhos automaticamente. Caso os valores sejam informados explicitamente, é mandatório que a divisão dos tamanhos das dimensões do espaço de índices por estes valores seja inteira.

7. **events\_in\_wait\_list**: número de eventos na lista de eventos que devem ser aguardados antes do início da execução do kernel.
8. **event\_wait\_list**: lista de eventos que devem ser aguardados antes do início da execução do kernel.
9. **event**: local para retorno do objeto de evento para o comando.

## Envio do kernel para execução no dispositivo

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

# Sincronização

(bloqueia host até término da execução do kernel)

```
cl_int clFinish(cl_command_queue command_queue)
```

- A função **clFinish()** bloqueia a execução no hospedeiro até que todos os comandos na fila informada tenham sido completados.

No programa: **clFinish(queue)**

## Transferência dos resultados da computação na GPU para a memória do host

```
cl_int  clEnqueueReadBuffer(cl_queue queue,  
                             cl_mem  buffer,  
                             cl_bool  blocking_read,  
                             size_t  offset,  
                             size_t  cb,  
                             const void* ptr,  
                             cl_uint  events_in_wait_list,  
                             const cl_event* event_wait_list,  
                             cl_event* event )
```

# Transferência dos resultados da computação na GPU para a memória do host

1. **queue**: fila de comandos.
2. **buffer**: objeto de memória do tipo buffer.
3. **blocking\_read**: caso CL\_TRUE, a chamada é bloqueante e o hospedeiro suspende a execução até que os dados tenham sido completamente transferidos do dispositivo. Caso CL\_FALSE, a chamada é não-bloqueante e a execução prossegue assim que o comando é enfileirado.
4. **offset**: *offset* (deslocamento) a partir do qual os dados devem ser transferidos.

# Transferência dos resultados da computação na GPU para a memória do host

5. **cb**: comprimento dos dados, em bytes, a serem transferidos.
6. **ptr**: ponteiro para a região de memória do host onde os dados transferidos devem ser escritos.
7. **events\_in\_wait\_list**: número de eventos na lista de eventos que devem ser aguardados antes do início da transferência dos dados.
8. **event\_wait\_list**: lista de eventos que devem ser aguardados antes do início da transferência dos dados.
9. **event**: local para retorno do objeto de evento (*event object*) para o comando.  
Objetos de evento permitem o aguardo do término de comandos inseridos em uma fila de comandos. Este objeto é útil para a garantia da consistência dos dados quando a escrita é realizada de forma não-bloqueante.



## Transferência dos resultados da computação na GPU para a memória do host

```
lEnqueueReadBuffer( queue,
                    bufC,
                    CL_TRUE,
                    0,
                    ARRAY_LENGTH * sizeof(int),
                    hostC,
                    0,
                    NULL,
                    NULL );
```

## Impressão dos resultados na saída padrão

```
for (i = 0; i < ARRAY_LENGTH; ++i)
    printf("%d - %d = %d\n", hostA[i], hostB[i],
           hostC[i]);
```

## Liberação de recursos e encerramento da aplicação

- Existe uma função de liberação para cada tipo de objeto OpenCL:
- Contextos: `clReleaseContext(cl_context context)`
- Filas de comandos: `clReleaseCommandQueue(cl_command_queue command_queue)`
- Objetos de programa: `clReleaseProgram(cl_program program)`
- Kernels: `clReleaseKernel(cl_kernel kernel)`
- Objetos de memória: `clReleaseMemObject(cl_mem buffer)`

## Liberação de recursos e encerramento da aplicação

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

## Compilação e Execução

```
> gcc -I/usr/local/cuda/include ArrayDiff.c  
    -o ArrayDiff -lOpenCL
```

Este comando realiza a **compilação** e **ligação** da aplicação com o *runtime OpenCL*.

Como pode ser observado, assume-se que o código tenha sido armazenado em um arquivo chamado **ArrayDiff.c**, porém este nome não é obrigatório, nem mesmo o nome dado ao executável ( **ArrayDiff** ).

Após a compilação, a execução se dá como a de qualquer outro executável em Linux: **> ./ArrayDiff**