Introdução ao OpenCL

Douglas Adriano Augusto
douglas@lncc.br

Semana Massivamente Paralela 2012 — LNCC

Conteúdo

Conteúdo

1. Introdução & contextualização
2. Arquitetura de dispositivos
3. A linguagem OpenCL

Justificativa

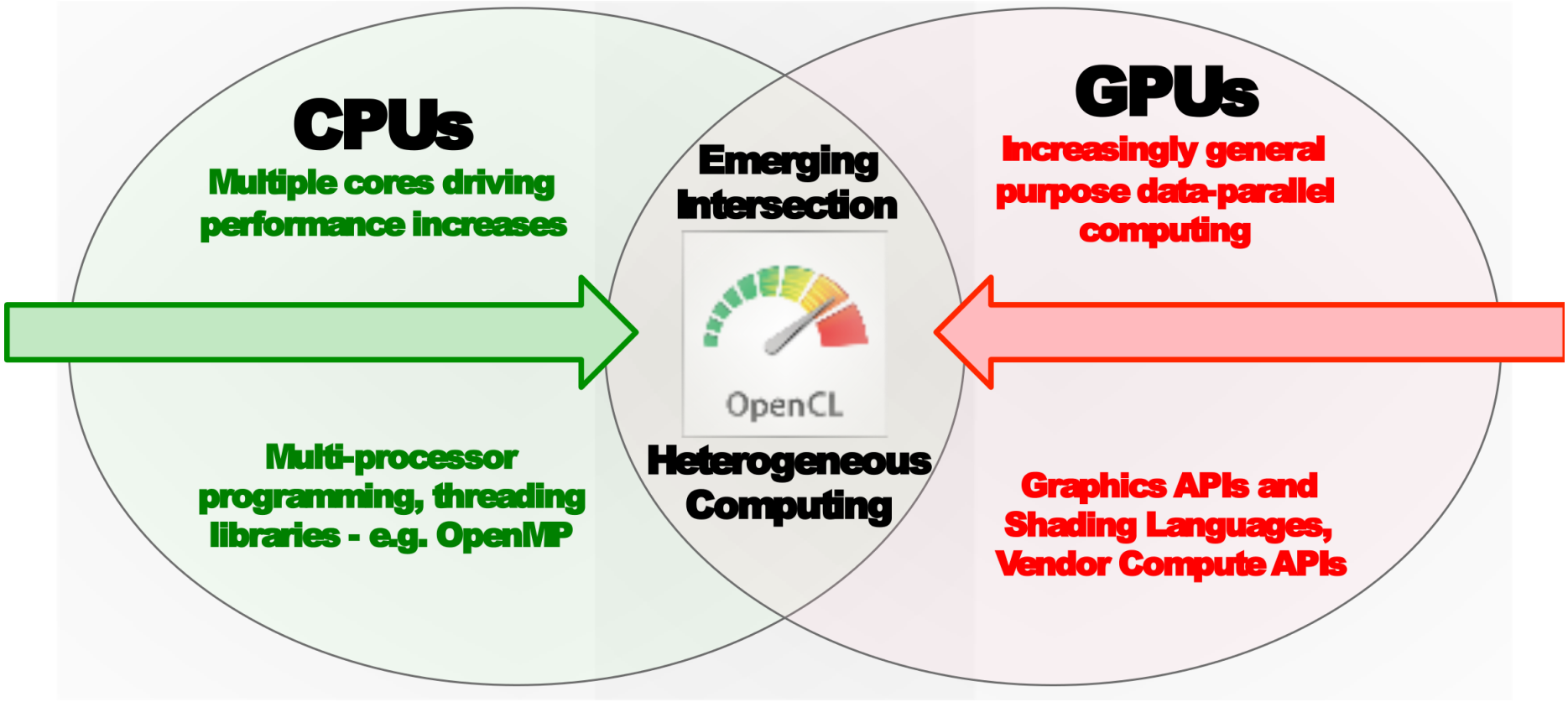
- Por que GPU?

Justificativa

- Por que **apenas** GPU?

OpenCL – Open Computing Language

"Padrão aberto para a programação paralela de sistemas heterogêneos"



OpenCL – Open Computing Language

Características:

- Provê interface *homogênea* para a exploração da computação paralela *heterogênea*
  - abstração do hardware
  - CPU's (AMD, ARM, IBM, Intel), GPU's (AMD, ARM, Intel, Nvidia), APU's, CBE, DSP's, FPGA's, MIC
- Padrão *aberto*
  - especificação mantida por vários membros
  - gerenciada pelo grupo *Khronos*
- Alto desempenho
  - possui diretivas de baixo nível para uso eficiente dos dispositivos
  - alto grau de flexibilidade

OpenCL – Open Computing Language

Características (cont.):

- Multi-plataforma
  - disponível em várias classes de hardware e sistemas operacionais
- Código *portável* entre *arquiteturas e gerações*
- Paralelismo de *dados* ("SIMD") e *tarefa* ("MIMD")
- Especificação baseada nas linguagens C e C++
- Define requisitos para operações em ponto flutuante:
  - resultados consistentes independente do dispositivo
- Integração com outras tecnologias (ex: OpenGL)

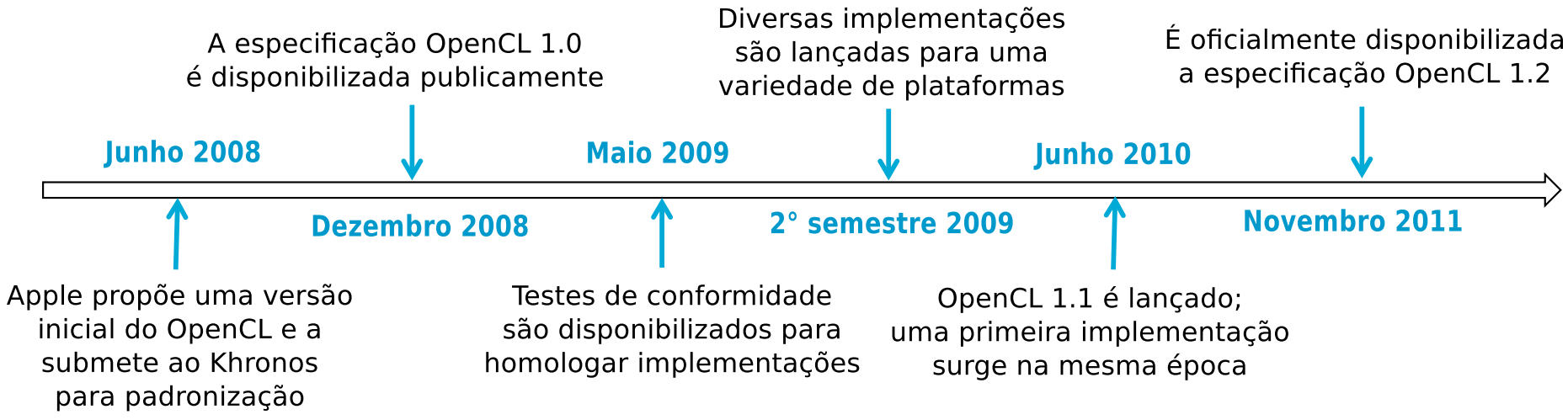
WebCL – Computação Paralela na Web

"**OpenCL acessível via navegador**"

- O objetivo é permitir que aplicações Web explorem todos os recursos computacionais disponíveis em um sistema heterogêneo
- É basicamente uma interface (em JavaScript) que acessa o OpenCL
- Pode ser integrado com o WebGL
- Definição da especificação ainda em desenvolvimento
  - várias demonstrações: editor de imagem/vídeo, simulações físicas, minerador bitcoin, etc.
- `http://www.khronos.org/webcl/`

História

- $\sim$**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



- **2012**: OpenCL 2.0?

História

Suporte da indústria em 2008

História

Suporte da indústria em 2010

Exemplo de Aplicações

Edição/manipulação de vídeo:

- Apple Final Cut Pro, Sony Vegas Pro, MotionDSP Ikena, Cyberlink PowerDirector, Magix Movie & Video Pro

Modelagem/Renderização 3D:

- Blender

Computação científica:

- Matlab: OpenCL Toolbox
- Folding@home

Ferramentas de "segurança":

- Pyrit, cRARk, etc.

Contextualização

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
- É mais maduro:
  - comunidade, bibliotecas, ferramentas, depuradores, ambiente de desenvolvimento, etc.
- Provê extensão para Fortran ("CUDA Fortran")

OpenCL $\times$ CUDA: Reflexão

Argumentos a favor do OpenCL:

- O rumo de uma tecnologia de tamanho impacto deveria ser de interesse geral, não apenas de *uma* companhia.

- Quem investiria em uma tecnologia controlada por um único fabricante (possivelmente concorrente)?

  - ex: a AMD não adotaria tecnologia de um rival, e ela hoje detém a GPU mais potente do mercado (pico teórico de 3788 GFLOP/s SP, 947 GFLOP/s DP)

OpenCL × CUDA: Reflexão

Argumentos a favor do OpenCL (cont.):

- Desenvolver compiladores para hardware de terceiros não é trivial:
  - poucos fabricantes publicam especificações completas da arquitetura
  - quando o fazem, um compilador desenvolvido independentemente virá provavelmente defasado

Uma "corrida armamentista" de hardware seria muito mais interessante (para todos) do que uma guerra de padrões.

OpenCL × OpenMP

OpenMP:

- Paralelismo apenas em CPU
- Não tira facilmente proveito das instruções SIMD dos processadores
- Mais alto nível:
  - programação mais simples, porém limitada/menos flexível
  - ganho de desempenho usualmente sub-ótimo
- Tem suporte para Fortran

OpenCL × OpenMP + CUDA

Computação heterogênea via OpenMP + CUDA:

- Dois códigos distintos:
  - não portável
- Não contempla outros fabricantes de GPUs senão Nvidia
  - AMD, Intel, ARM, etc.
- Não contempla outras arquiteturas além de CPU e GPU
  - APUs, DSPs, FPGA's, etc.

OpenCL × MPI

São tecnologias ortogonais:

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

OpenCL – Open Computing Language

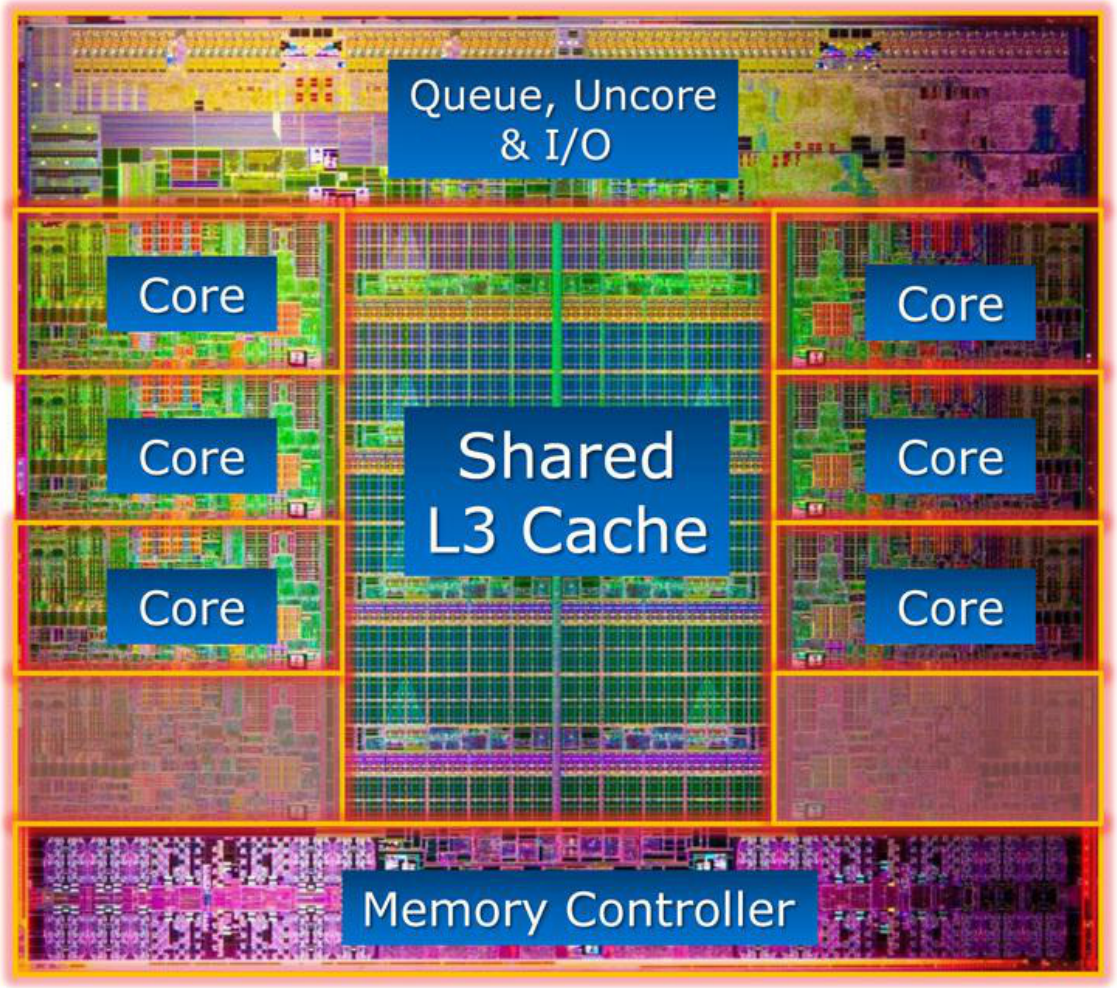
Considerações:

- Pode ser demasiadamente de *baixo nível*
  - problemático se não houver intimidade com C/C++
  - há alternativas, no entanto:
    PyOpenCL, *JOCL*, *Aparapi*, *Cloo*, etc.
- Ecossistema ainda não muito rico
  - comunidade, bibliotecas, ferramentas, depuradores, ambiente de desenvolvimento, etc.
- Implementações não tão maduras
  - grande margem para otimização
  - bugs
- Carência de implementações livres
  - mais grave no que tange às GPUs

Arquiteturas CPU & GPU

Arquitetura CPU

Intel Core i7-3690X: 160 GFLOP/s (DP)

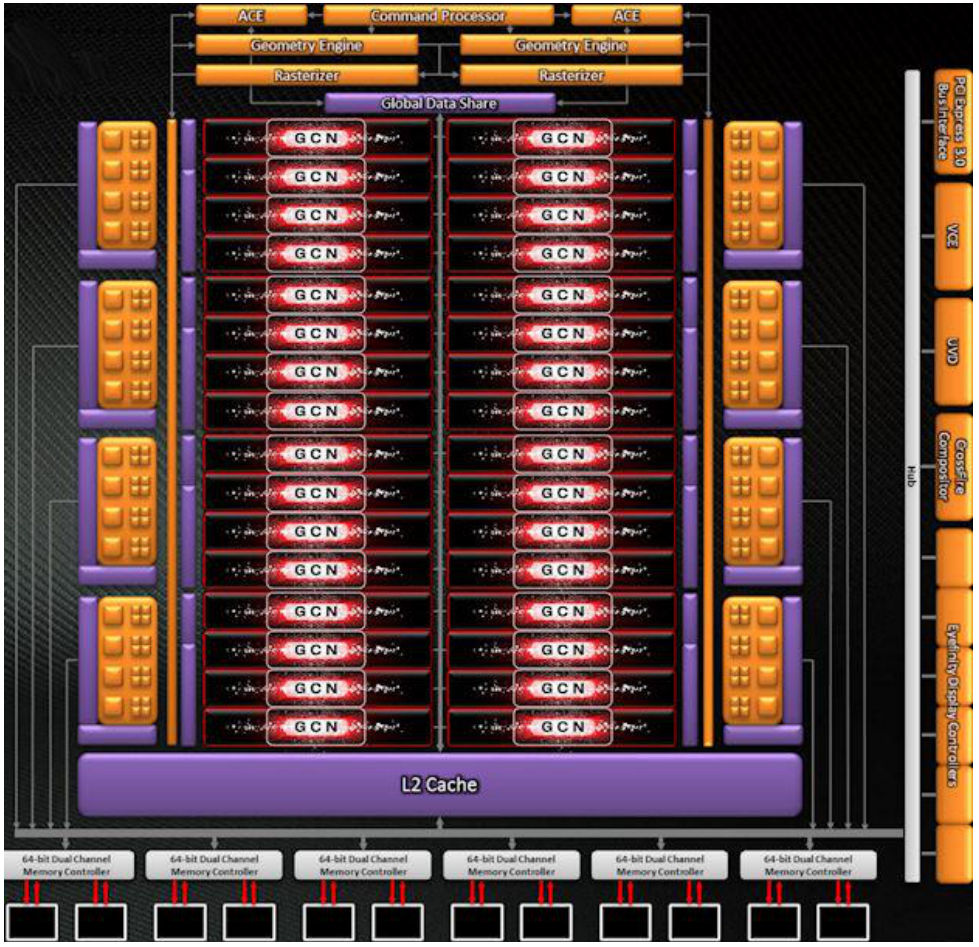


Processo	Dimensão	Transístores	Núcleos	Frequência	Consumo
32nm	435mm^2	2,27 Bilhões	6 físicos e 6 lógicos	3,33GHz	130W

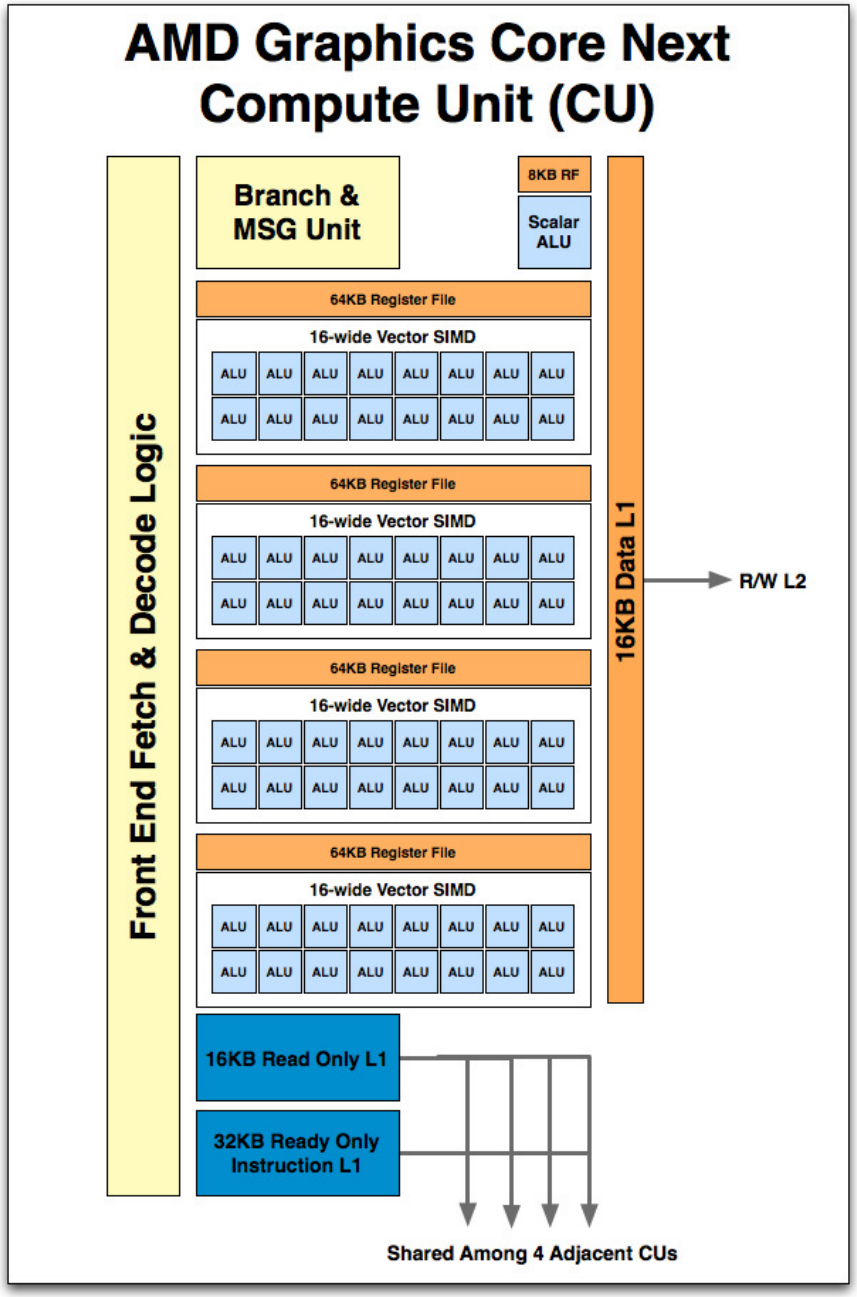
Arquitetura Graphics Core Next (GCN) – AMD

Arquitetura GCN – AMD

- AMD 7970 – *GCN*
- 32 unidades de computação
  "SIMD engines"
- 64 elementos de processamento por unidade
  "Stream cores"
  - cada um do tipo escalar
- $\sim$3,79 Teraflop/s em precisão simples
- $\sim$947 $(1/4)$ Gigaflop/s em precisão dupla
- Memória: 264GB/s
- Consumo: $\sim$250W
- 4,3 Bilhões de transístores



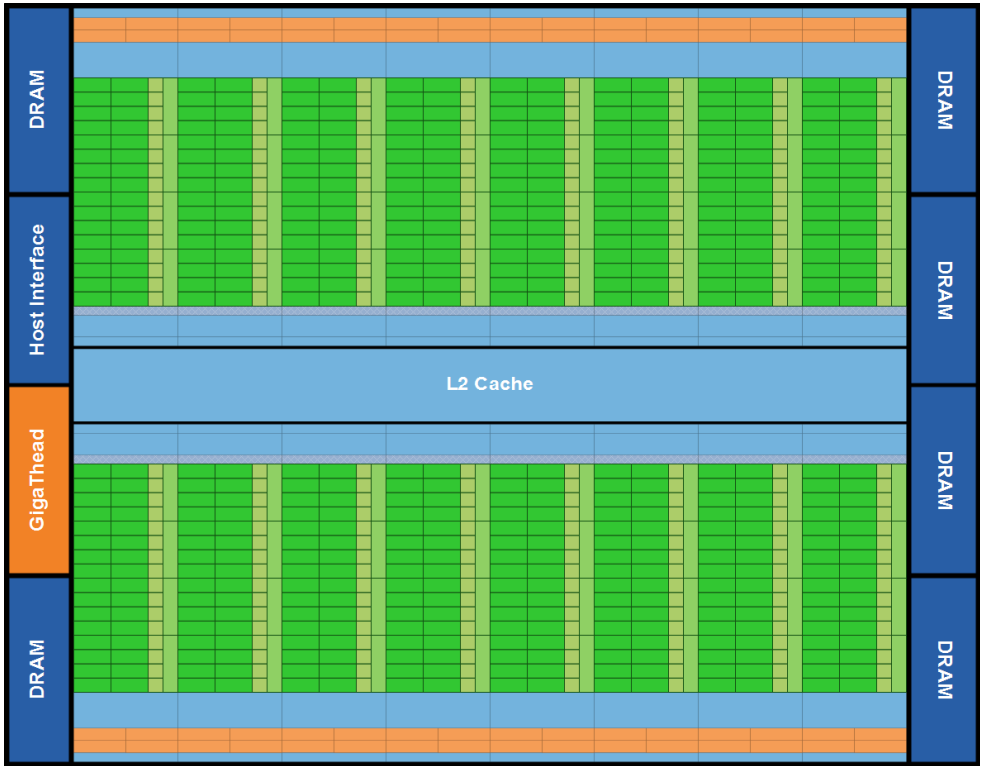
Unidade Computacional



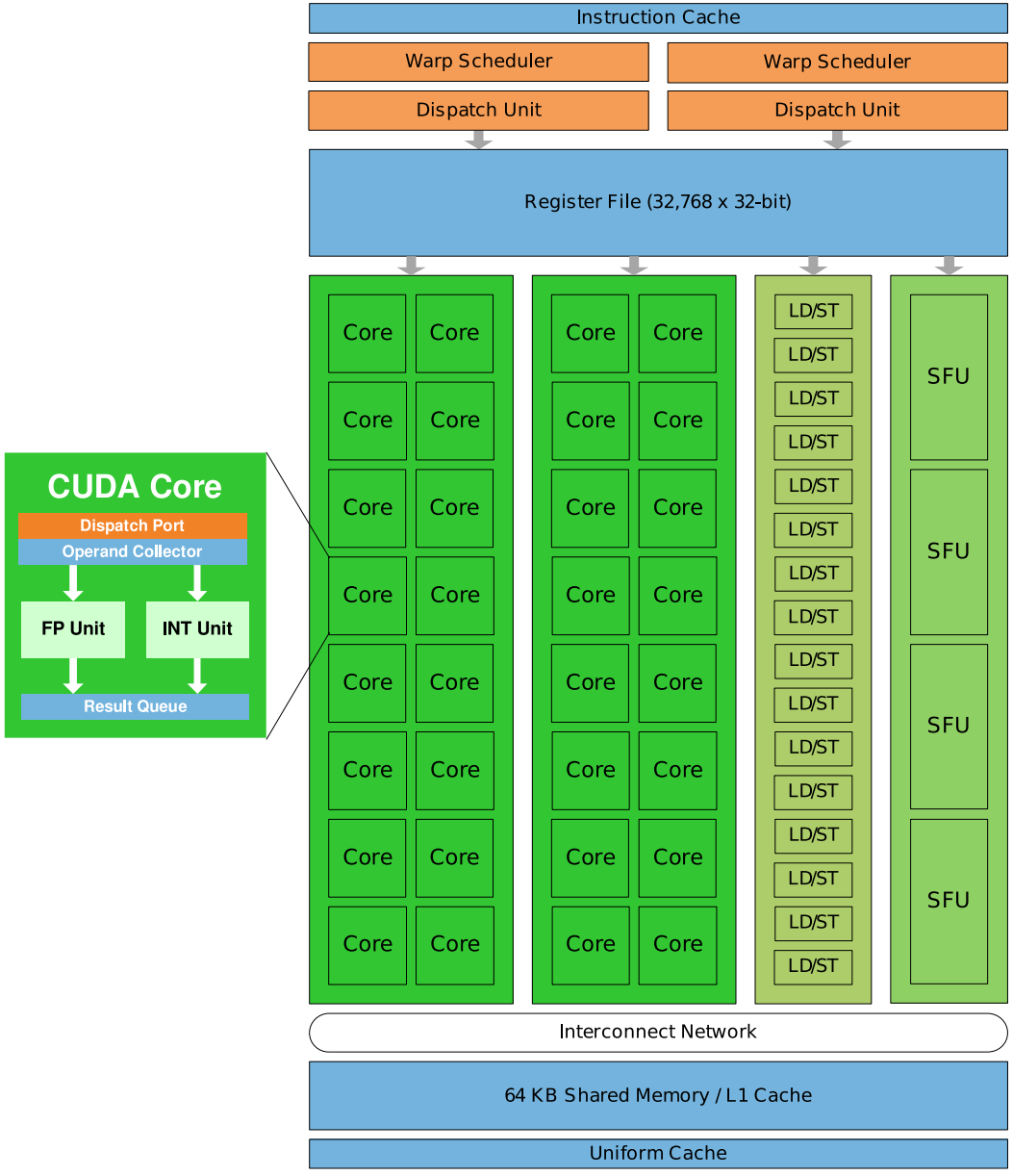
Arquitetura Fermi – Nvidia

Arquitetura Fermi

- Tesla C2070 – *Fermi*
- 14 unidades de computação
  "Streaming Multiprocessors"
- 32 elementos de processamento por unidade
  "CUDA cores"
  - cada um do tipo escalar
- ∼1,03 Teraflop/s em precisão simples
- ∼515 $(1/2)$ Gigaflop/s em precisão dupla
- Memória: 144GB/s
- Consumo: ∼240W
- 3 Bilhões de transístores



Unidade Computacional

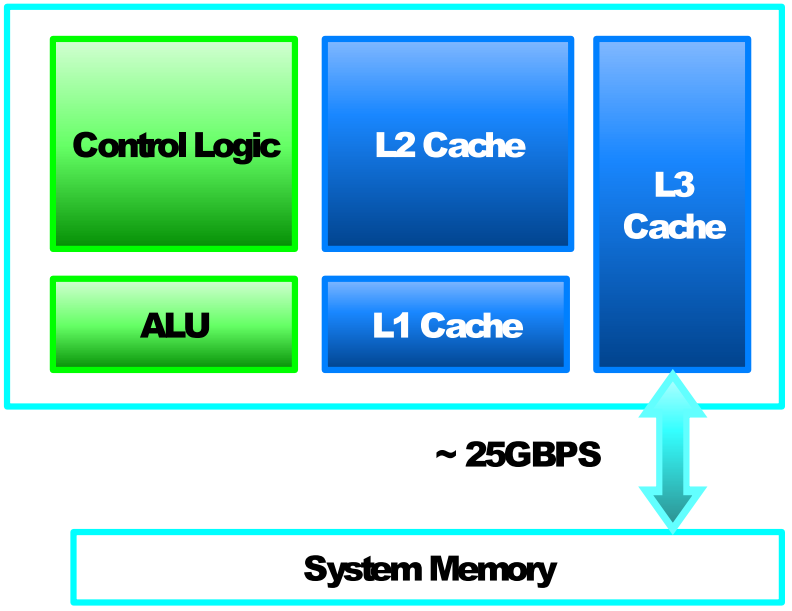


Fermi Streaming Multiprocessor (SM)

CPU versus GPU

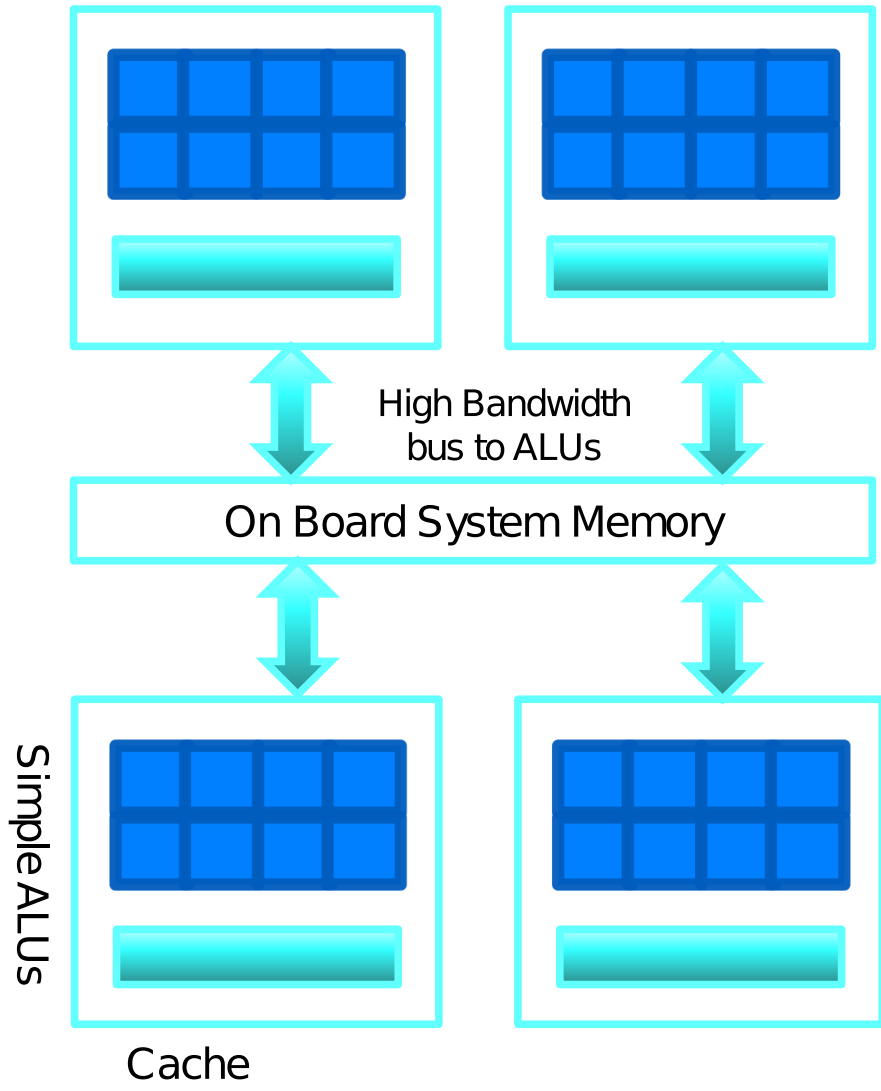
Arquitetura CPU Convencional

- Espaço dedicado à *unidade de controle* em vez de ALUs
- São otimizadas para minimizar a latência de um único *thread*
  - lida eficientemente com controle de fluxo
- Usa vários níveis de cache para encobrir latência
- Unidade de controle para reordenar execução, prover paralelismo de instruções e minimizar interrupções no pipeline



Arquitetura GPU Moderna

- Menos espaço dedicado à *unidade de controle* e *caches*
- Latência encoberta por alternância de *threads*
- Grande número de ALUs por unidade de computação
  - cada unidade contendo um pequeno cache
- Memória com grande largura de banda
  - ∼200 GB/Ps suprir várias ALUs simultaneamente



Diferenças Fundamentais: CPU x GPU

Arquitetura:

- **CPU**: MIMD (*Multiple Instruction Multiple Data*)
  - paralelismo de tarefas e dados
  - também possui paralelismo via instruções estendidas SIMD
  - mais flexível, propósito geral

- **GPU**: SIMD (*Single Instruction Multiple Data*)
  - paralelismo de dados
  - mais restrita (especializada), mas continuamente adquire características de propósito geral

Diferenças Fundamentais: CPU x GPU

Programação:

- **CPU**: facilmente programável
  - conceitualmente mais simples, foco essencialmente sequencial
  - maior disponibilidade e maturidade de linguagens e ferramentas de suporte (ex: depuração)

- **GPU**: programação menos direta
  - foco no paralelismo e escalabilidade
  - custo de engenharia de software (implementação, depuração, manutenção, etc.)
  - mais sensível ao projeto do algoritmo...
    ...ou, por outro lado, maior margem de otimização

Diferenças Fundamentais: CPU x GPU

Carga de trabalho:

- **CPU**: projetada para **reduzir a latência** na execução de uma tarefa:
  - baixa latência na execução de instruções e acesso à memória
  - uso intenso de memórias *cache* e outras tecnologias

- **GPU**: projetada para **aumentar a vazão** (*throughput*):

  "cada pixel pode demorar quanto tempo for...

  ...desde que sejam processados vários ao mesmo tempo"

Diferenças Fundamentais: CPU x GPU

Processamento: o poder bruto de processamento da **GPU** é significativamente maior:

- Grande número de unidades computacionais: centenas ou milhares
- Aproveitamento dos recursos (transístores) em processadores mais simples:
  - implementação minimalista (ou inexistente): *unidades de controle*, *memórias cache*, *execução fora-de-ordem*, *predição de desvios*, *execução especulativa*, etc.

Mas o desempenho da **GPU** cai consideravelmente em *cargas de trabalhos irregulares*, por exemplo, com muitos desvios.

Diferenças Fundamentais: CPU x GPU

Memória:

- **CPU**:
  - baixa latência de acesso à memória
  - alta frequência
  - goza de acesso direto à memória do sistema
  - em geral maior capacidade de armazenamento

- **GPU**:
  - maior largura de banda, mas grande latência
  - necessita de alta razão *cômputo/acesso à memória*: grande intensidade de operações aritméticas
  - seu uso normalmente requer transferência prévia de dados da memória do sistema para o dispositivo

Diferenças Fundamentais: CPU x GPU

Escalabilidade:

- **CPU**: menos escalável
  - os processadores (núcleos) são complexos
  - a arquitetura limita o número máximo viável de núcleos

- **GPU**: mais escalável
  - a expansão do número de processadores é trivial
  - pode-se facilmente adicionar ao computador novos dispositivos ou atualizar os existentes

Perfil Ótimo de Carga em GPU

Recomendações

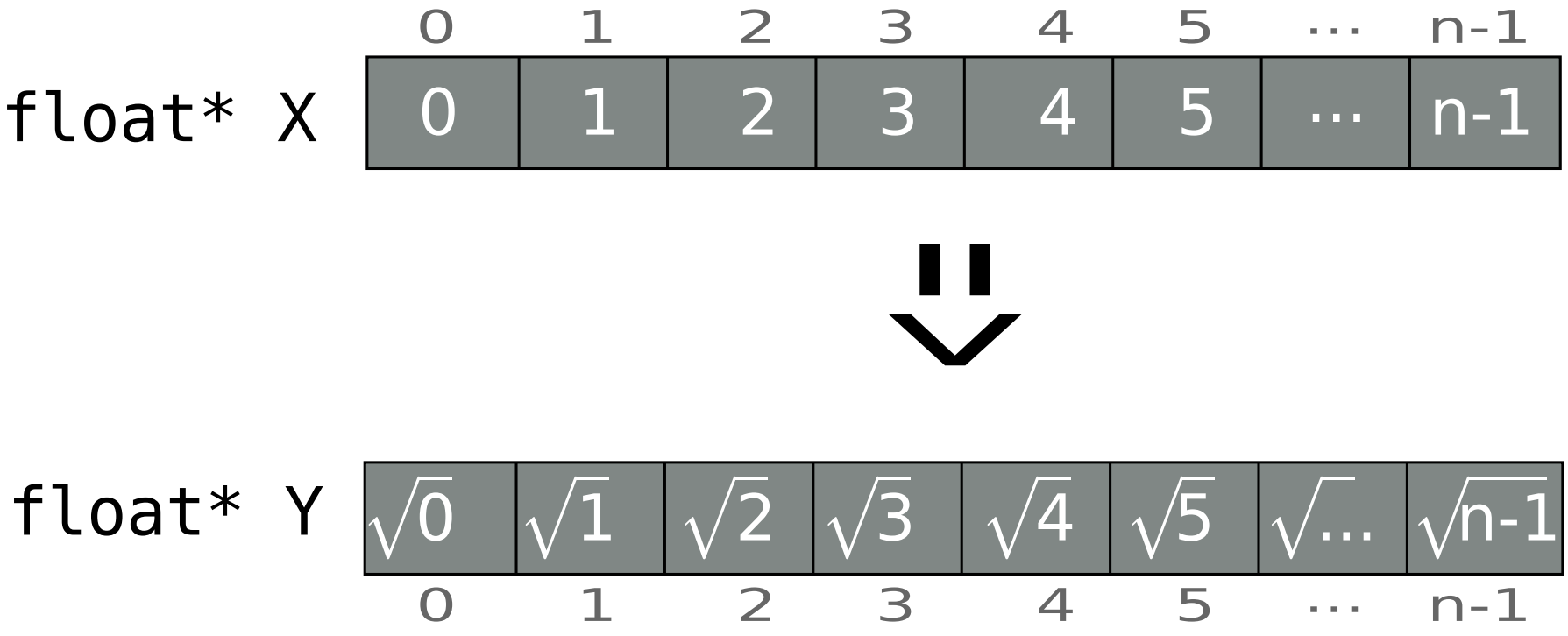
Válido para qualquer arquitetura de GPU moderna:

- Muitos (milhares) de *threads* independentes
  - uso de todas unidades de computação
  - admite alternância de *threads* para encobrir latência
- Minimiza desvios de fluxo (baixa ramificação)
  - evita o problema da divergência
- Possui alta intensidade aritmética
  - razão cômputo/acesso à memória é alta
  - evita gargalo de acesso à memória

Fundamentos do OpenCL

Problema Ilustrativo: $\sqrt{}$

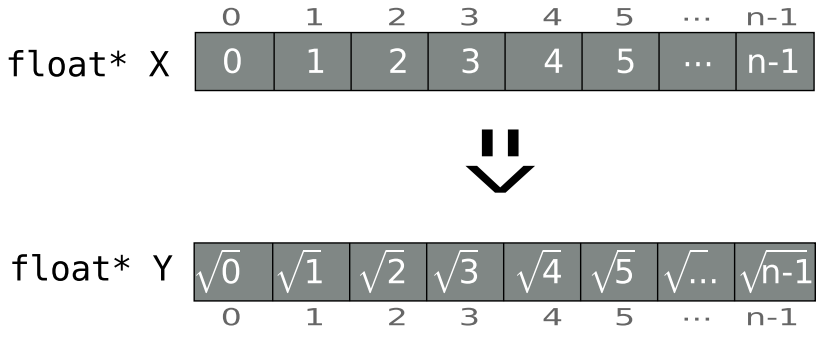
Calcular a raiz quadrada de cada elemento de um vetor:



Problema Ilustrativo: $\sqrt{}$

Solução sequencial:

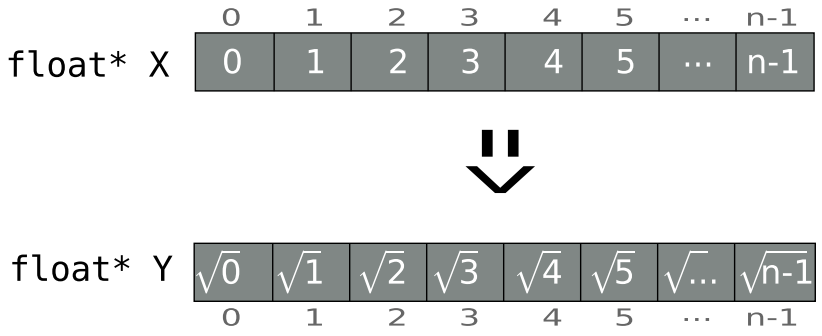
```
void raiz( const float * x, float * y, int n )
{
   for( int i = 0; i < n; ++i )
   {
      y[i] = sqrt( x[i] );
   }
}
```



Problema Ilustrativo: $\sqrt{}$

Solução paralela via OpenCL (*kernel*):

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



Problema Ilustrativo: $\sqrt{}$

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*):

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

Introdução

Código do Kernel e Hospedeiro

Existem duas hierarquias de códigos no OpenCL:

- O *kernel*:
  - tarefa executada paralelamente em um dispositivo computacional
  - implementado em C (baseado na especificação C99)

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

- O código *hospedeiro*:
  - coordena os recursos e ações do OpenCL
  - implementado em C ou C++

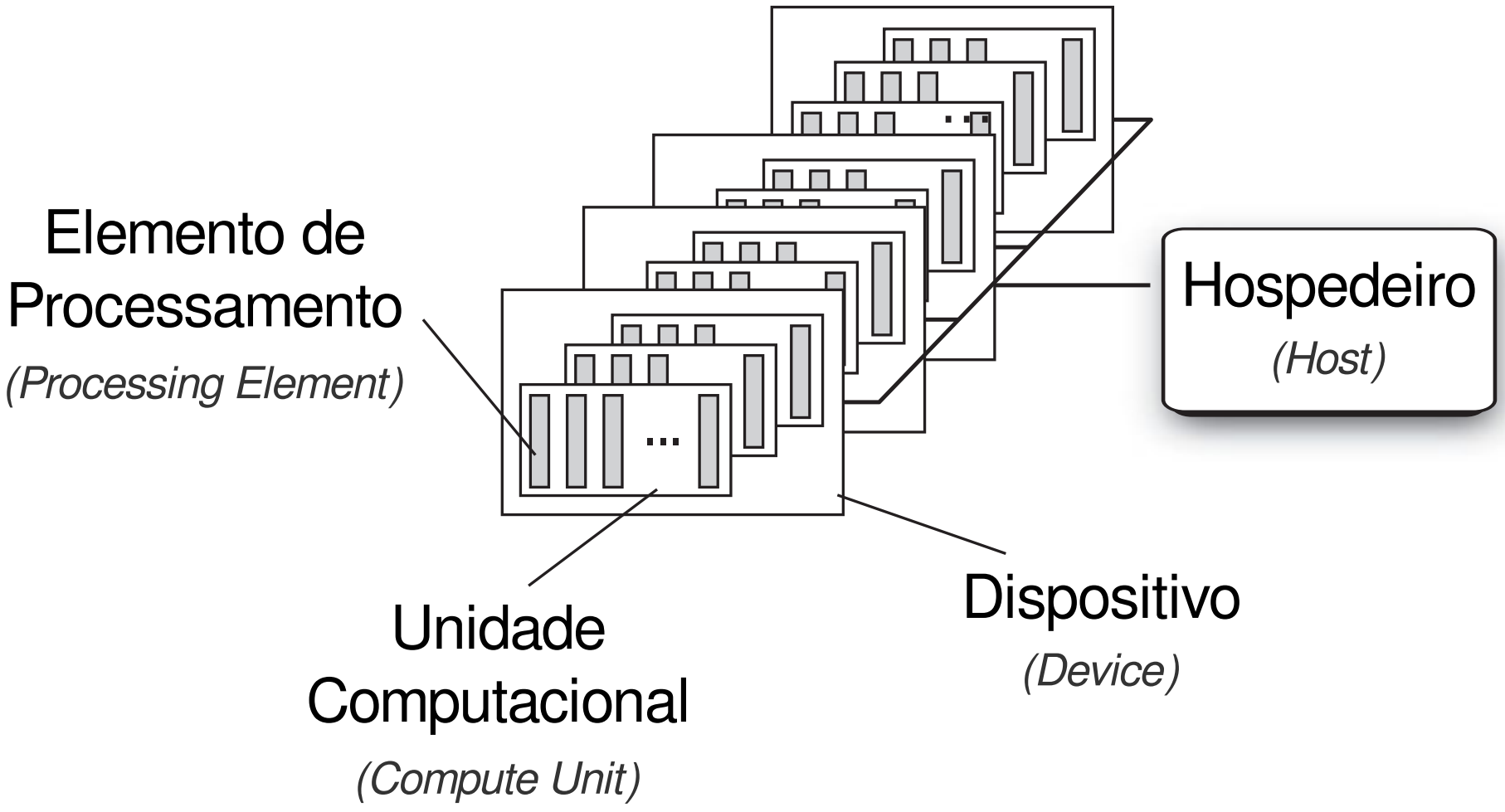
Arquitetura do OpenCL

Modelos

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

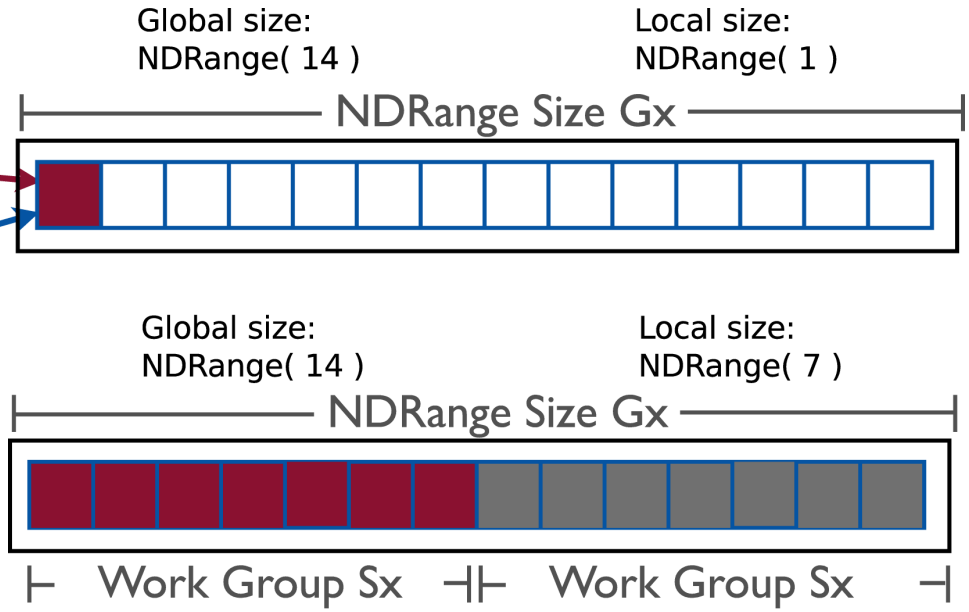
Baseia-se nos elementos:

- **Item de trabalho** (*work-item*):
  - uma *instância* do *kernel* em execução
  - *unidade de execução* concorrente do OpenCL
  - possui identificadores *local* e *global* dentro de um *espaço de índices*
- **Grupo de trabalho** (*work-group*):
  - uma coleção de itens de trabalho
  - itens de trabalho de um mesmo grupo podem se *comunicar eficientemente* e *sincronizar*

Modelo de Execução

item de trabalho
(work-item)

grupo de trabalho
(work-group)



NDRange Size = Global Size
(Tamanho global)

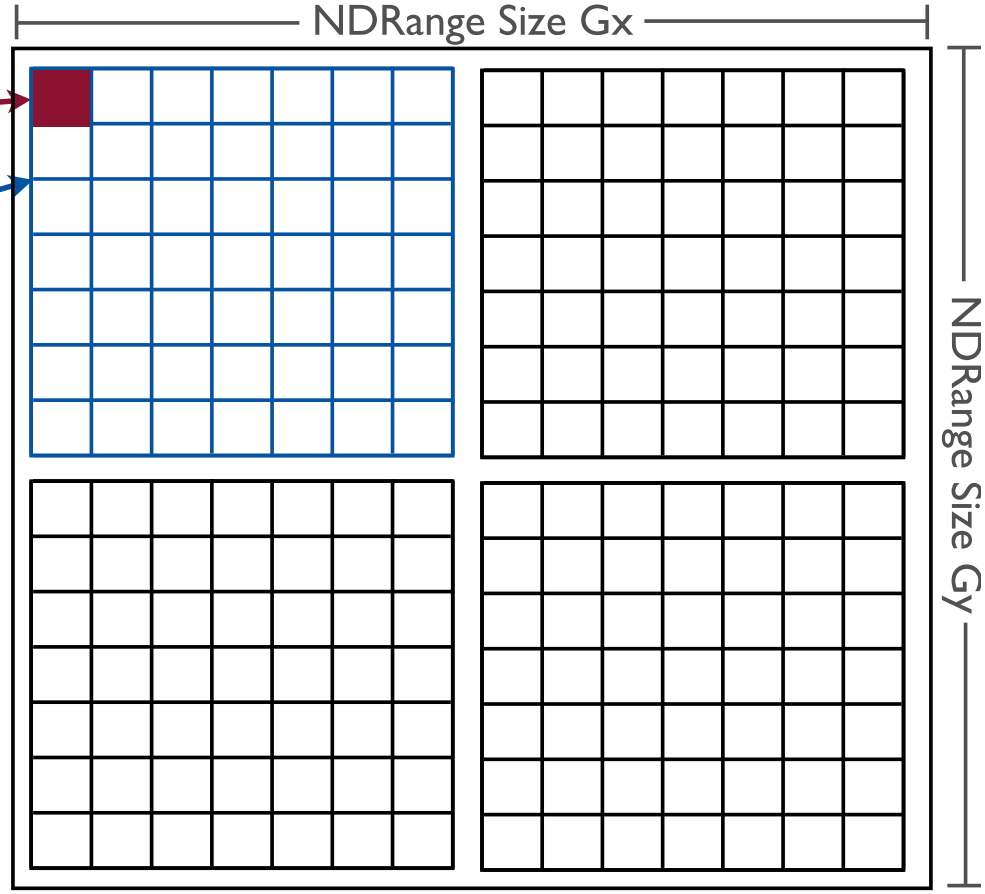
Work Group Size = Local Size
(Tamanho do grupo de trabalho) (Tamanho local)

Espaço de índices unidimensional

Modelo de Execução

item de trabalho
(work-item)

grupo de trabalho
(work-group)



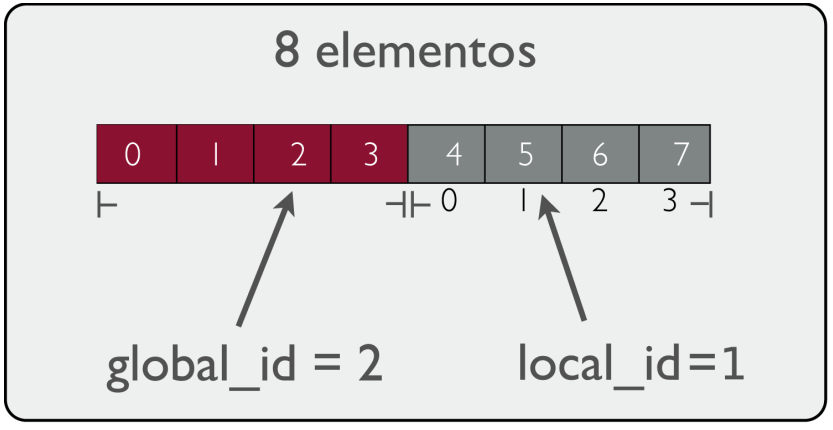
Global size:
NDRange(14, 14)

Local size:
NDRange(7,7)

Espaço de índices bidimensional

Modelo de Execução

- Cada item de trabalho está "ciente" sobre qual elemento do problema ele está trabalhando
- Cada item de trabalho (e grupo) pode ser identificado dentro do *kernel*



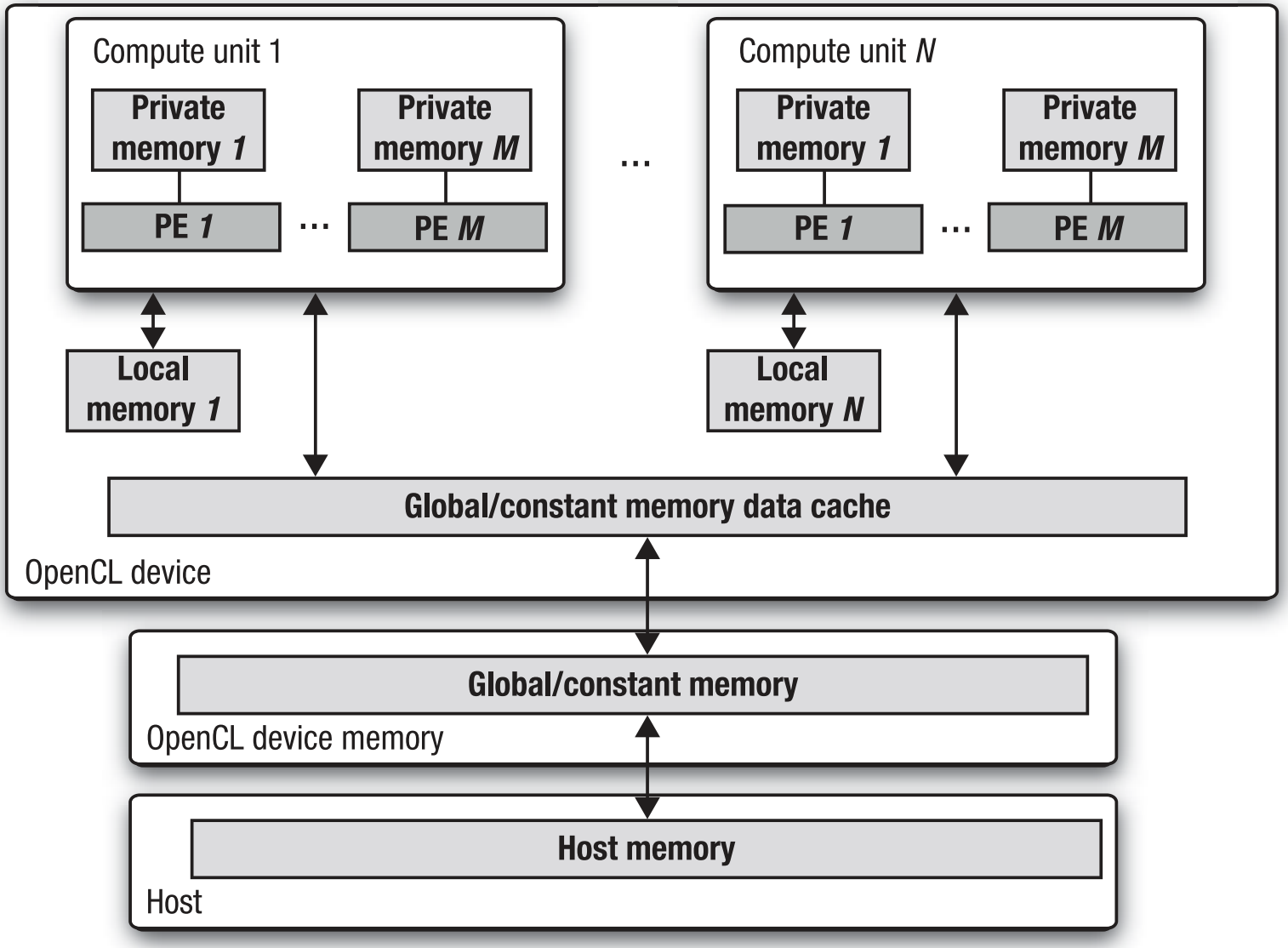

```
get_local_id(x);
get_global_id(x);
```

x = 0, 1 or 2

Identificadores

Modelo de Memória

`__private, __local, __constant, __global`



Modelo de Programação

- Paralelismo de dados
  - modelo mais natural ao OpenCL
  - hierárquico: inter e intra grupo de trabalho
  - instruções vetoriais SIMD
- Paralelismo de tarefas

Dinâmica do OpenCL

Passos de Execução

Uma modelagem típica consiste em:

1. **Inicialização**
2. **Preparação da memória** (leitura e escrita)
3. **Execução**

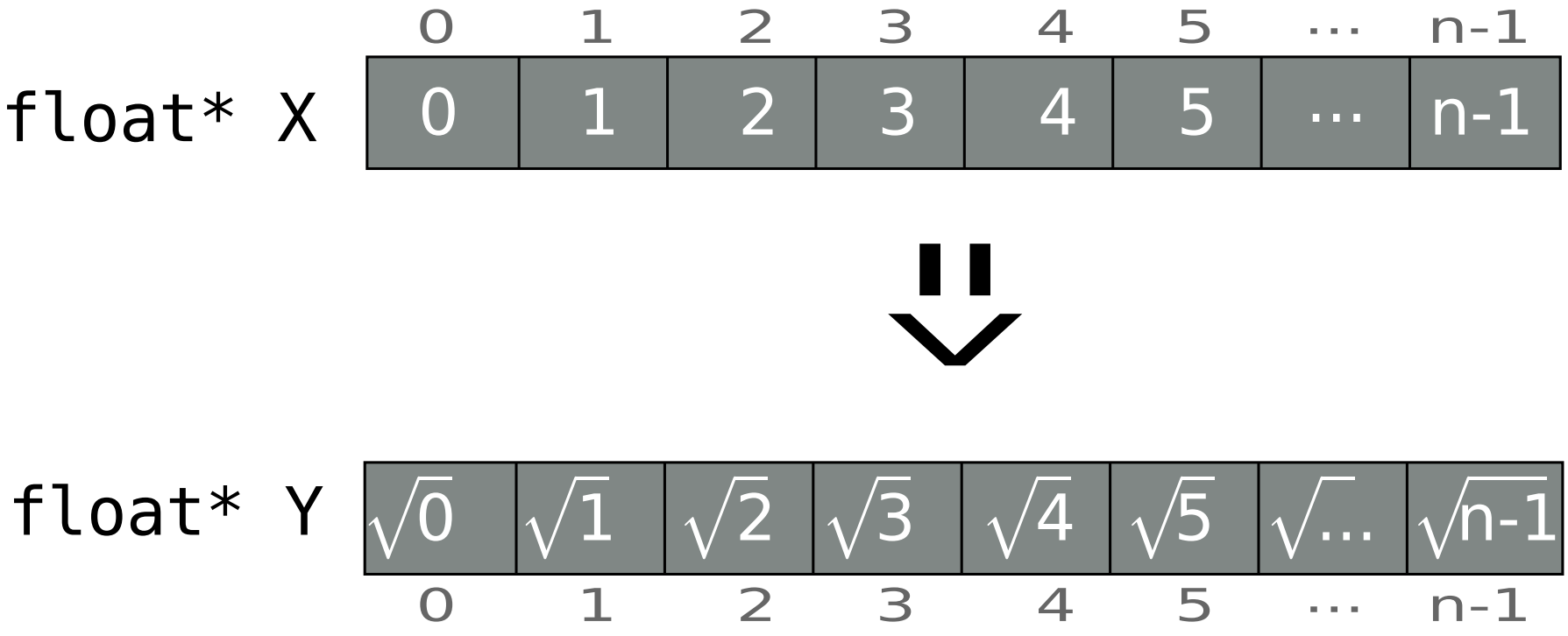
Passos de Execução

1. **Inicialização**
   - Descobrir e escolher as plataformas e dispositivos
   - Criar o contexto de execução
   - Criar a fila de comandos para um dispositivo
   - Carregar o programa, compilá-lo e gerar o *kernel*
2. **Preparação da memória** (leitura e escrita)
3. **Execução**
   - Transferência de dados para o dispositivo
   - Execução do *kernel*: definição dos argumentos e trabalho/particionamento
   - Espera pela finalização da execução do *kernel*
   - Transferência dos resultados para o hospedeiro

Problema Ilustrativo: $\sqrt{}$

Problema Ilustrativo: $\sqrt{}$

Calcular a raiz quadrada de cada elemento de um vetor:



Cabeçalho

```
// Habilita disparar exceções C++
#define __CL_ENABLE_EXCEPTIONS

// Cabeçalho OpenCL para C++
#include <cl.hpp>

#include <iostream>
#include <vector>
#include <utility>
#include <cstdlib>

using namespace std;
```

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] ); "
   "} ";
```

Entrada

```
int main( int argc, char* argv[] )
{
   // Dados de entrada: vetor de número reais
   const int elementos = atoi( argv[1] );

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

Inicialização

```
//    Descobrir e escolher as plataformas e dispositivos
vector<cl::Platform> plataformas;
vector<cl::Device> dispositivos;

cl::Platform::get( &plataformas ); // plataformas
plataformas[0].getDevices( CL_DEVICE_TYPE_ALL, &dispositivos ); // dispositivos

//    Criar o contexto
cl::Context contexto( dispositivos );

//    Criar a fila de comandos para um dispositivo
cl::CommandQueue fila( contexto, dispositivos[0] );

//    Carregar o programa, compilá-lo e gerar o kernel
cl::Program::Sources fonte( 1, make_pair( kernel_str, strlen( kernel_str ) ) );
cl::Program programa( contexto, fonte );

programa.build( vector<cl::Device>() );

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

Preparação da Memória

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

Execução

```
//    Transferência de dados para o dispositivo
fila.enqueueWriteBuffer( bufferX, CL_TRUE, 0, elementos * sizeof( float ), X );

//    Execução do kernel: definição dos argumentos e trabalho/particionamento
kernel.setArg( 0, bufferX );
kernel.setArg( 1, bufferY );
fila.enqueueNDRangeKernel( kernel, cl::NDRange(),
                           cl::NDRange( elementos ), cl::NDRange() );

//    Espera pela finalização da execução do kernel
fila.finish();

//    Transferência dos resultados para o hospedeiro
fila.enqueueReadBuffer( bufferY, CL_TRUE, 0, elementos * sizeof( float ), Y );
```

Finalização

```
    // Impressão do resultado
    for( int i = 0; i < elementos; ++i ) cout << '[' << Y[i] << ']'; cout << endl;

    // Limpeza
    delete[] X, Y;

    return 0;
}
```

```cpp
#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 elementos = atoi( argv[1] );
   float *X = new float[elementos];
   for( int i = 0; i < elementos; ++i ) X[i] = i;
   float *Y = new float[elementos];
   // --- Inicialização:
   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( vector<cl::Device>() );
   cl::Kernel kernel( programa, "raiz" );
   // --- Preparação da memória:
   cl::Buffer bufferX( contexto, CL_MEM_READ_ONLY, elementos * sizeof( float ) );
   cl::Buffer bufferY( contexto, CL_MEM_WRITE_ONLY, elementos * sizeof( float ) );
   // --- Execução:
   fila.enqueueWriteBuffer( bufferX, CL_TRUE, 0, elementos * sizeof( float ), X );
   kernel.setArg( 0, bufferX );
   kernel.setArg( 1, bufferY );
   fila.enqueueNDRangeKernel( kernel, cl::NDRange(), cl::NDRange( elementos ), cl::NDRange() );
   fila.finish();
   fila.enqueueReadBuffer( bufferY, CL_TRUE, 0, elementos * sizeof( float ), Y );
   for( int i = 0; i < elementos; ++i ) cout << '[' << Y[i] << ']'; cout << endl;
   delete[] X, Y;
   return 0;
}
```

Compilação e Execução

OpenCL é uma especificação; implementações (plataformas) são fornecidas independentemente:

- AMD
  - Suporte às GPUs AMD + CPUs AMD e Intel
- Intel
  - Suporte às CPUs Intel
- Nvidia
  - Suporte às GPUs Nvidia
- Outras: IBM, etc.

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
```

- Execução:

```
./ex <n>

./ex 10
[0][1][1.41421][1.73205][2][2.23607][2.44949][2.64575][2.82843][3]
```

O Kernel OpenCL

Kernel OpenCL

- Escrito em uma linguagem de programação conhecida como *OpenCL C*
  - derivada da especificação C99
  - modificações para comportar arquiteturas heterogêneas

Linguagem OpenCL C

Exclusões:

- Recursividade
- Apontadores para funções
- Vetores (*arrays*) de tamanho variável
- Apontadores para apontadores como argumentos
- Tipo real de dupla precisão (*double*) é opcional

Linguagem OpenCL C

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`

Linguagem OpenCL C

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;
```

Linguagem OpenCL C

Funções de identificação

- Item de trabalho:

  ```
  get_global_id(dim)
  get_local_id(dim)
  ```

- Grupo de trabalho:

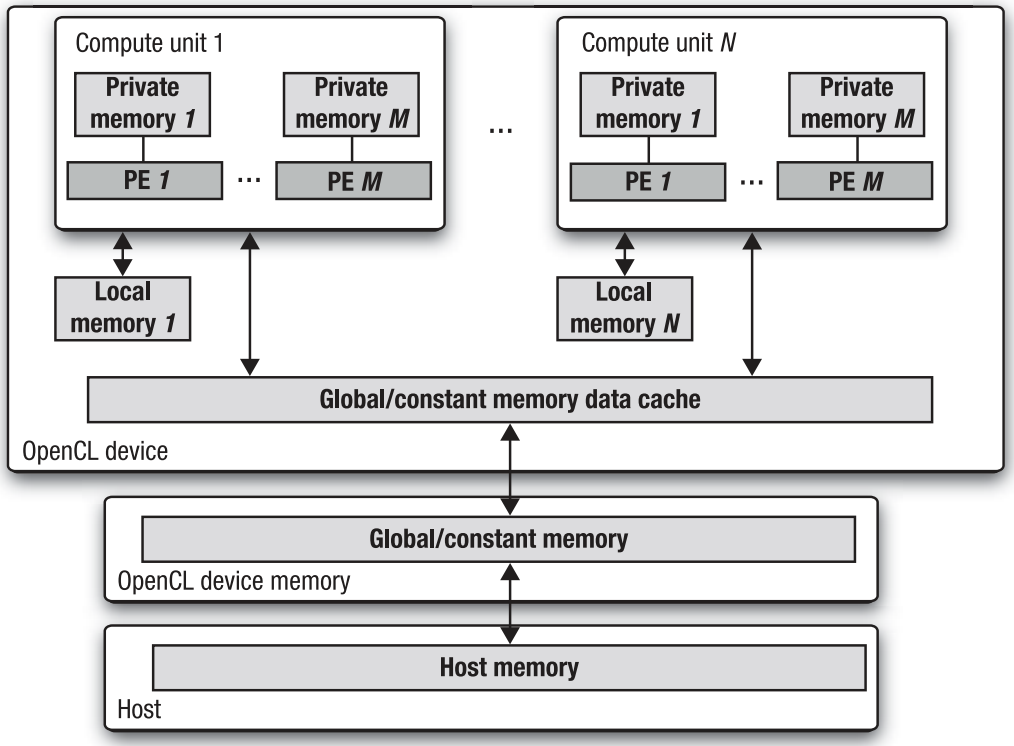
  ```
  get_group_id(dim)
  ```

- Espaço de índices:

  ```
  get_work_dim()
  get_global_size(dim)
  get_local_size(dim)
  get_num_groups(dim)
  get_global_offset(dim)
  ```

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

Escopo de Alocação/Acesso à Memória

Memória	**Hospedeiro**		**Kernel**	
	Alocação	*Acesso*	*Alocação*	*Acesso*
global	dinâmica	leitura/escrita	–	leitura/escrita
constant	dinâmica	leitura/escrita	estática	leitura
local	dinâmica	–	estática	leitura/escrita
private	–	–	estática	leitura/escrita

Declarações de Variáveis no Kernel

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

Declarações dos Argumentos do Kernel

Declaração:

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

Sintaxe de definição:

```
setArg( índice, objeto );
```

Definição:

```
setArg( 0, bufferX );
setArg( 1, bufferY );
setArg( 2, bufferZ );
setArg( 3, (float) 3.1415 );
```

Consistência de Memória e Sincronia

Introdução

Consistência de memória diz respeito à correta visibilidade, em tempo de execução, de conteúdo de memória entre os itens de trabalho:

- não basta conhecer onde o conteúdo será armazenado; é preciso garantir que um item de trabalho leia corretamente os valores escritos pelos demais

O OpenCL adota um modelo *relaxado* de consistência de memória:

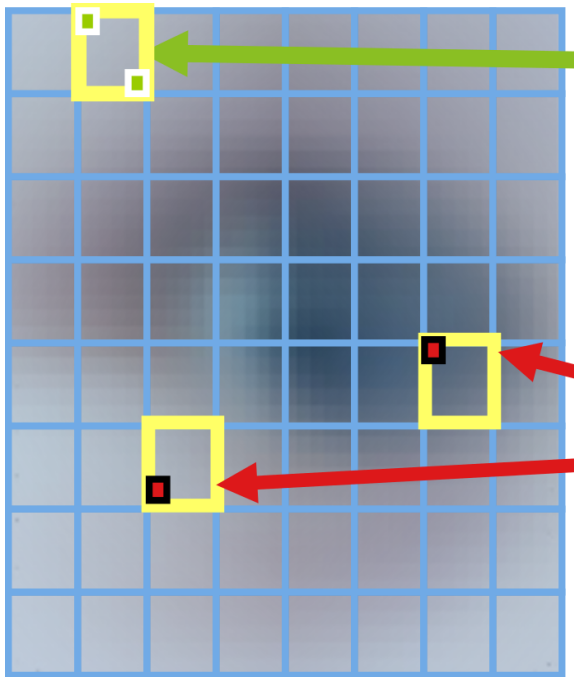
- dependendo do espaço de memória, a consistência só é obtida em pontos de sincronização

Modelo de Execução do OpenCL

- *Itens de trabalho* são executados nos *elementos de processamento*
- Cada *grupo de trabalho* é executado em uma *única unidade computacional*
  - diferentes grupos de trabalho são executados *independentemente*
  - na CPU uma unidade computacional é mapeada em um *núcleo*;
  - na GPU ela é mapeada em uma coleção de *elementos de processamento*

Modelo de Execução do OpenCL

- *"Não há"* sincronia *global*
- Apenas itens de trabalho de um mesmo grupo podem sincronizar entre si



Sincronização de itens de trabalho somente *dentro* de um grupo de trabalho

***Não* é possível sincronizar *fora* de um grupo de trabalho**

Modelo de Execução do OpenCL

Razões em favor da inexistência de sincronia global:

- Tácita, induzir um melhor particionamento do problema:
  - sincronia global implica em menor escalabilidade
- Suporte a dispositivos heterogêneos:
  - com sincronia global uma determinada arquitetura deveria ser capaz de gerenciar/executar *todos* os grupos de trabalho *concorrentemente*

Consistência por Escopo 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

Primitiva de Sincronia

Primitiva de Sincronia

Itens de trabalho de um mesmo grupo são sincronizados—e a consistência garantida—usando-se no *kernel* a primitiva:

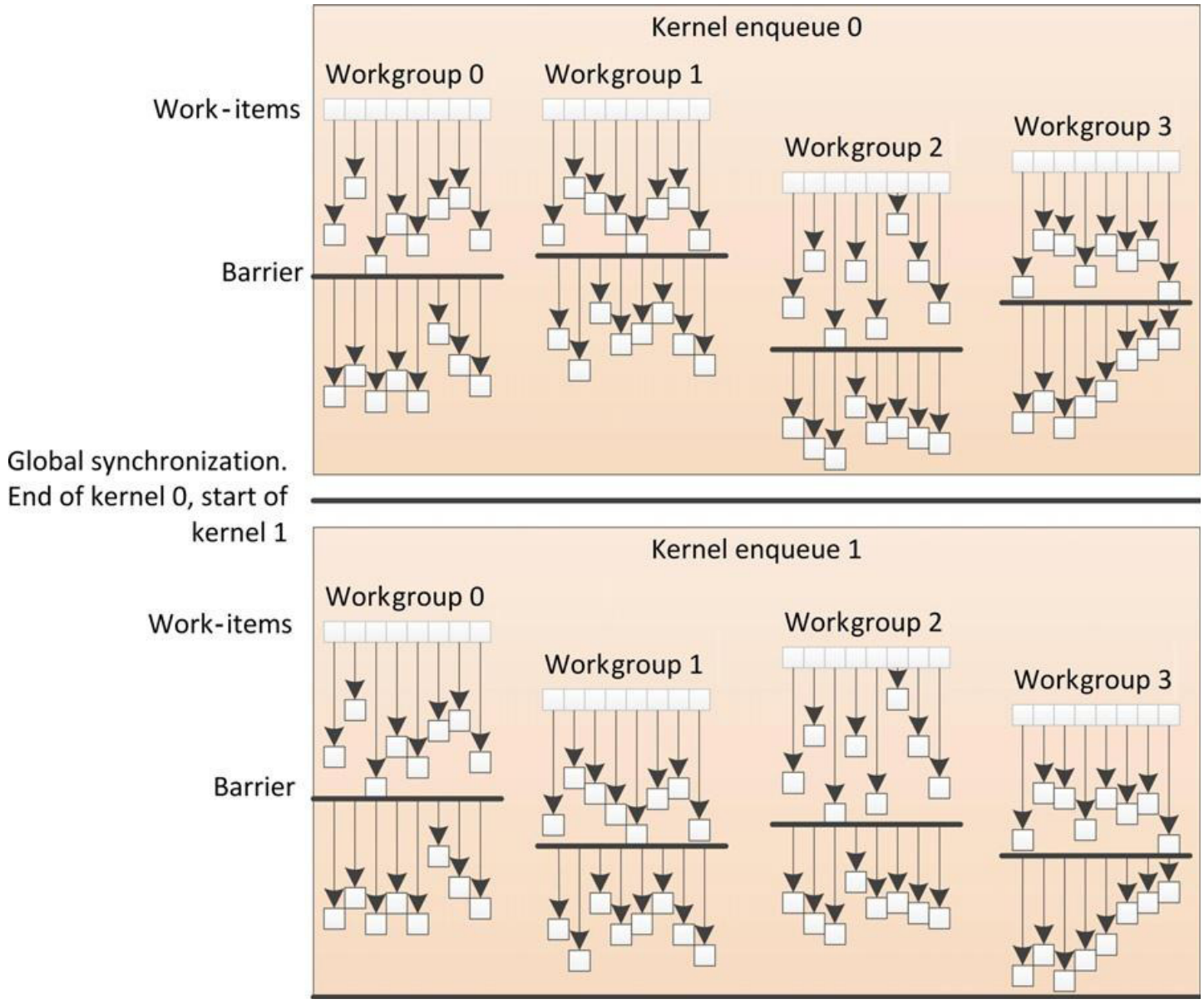
```
void barrier( <escopo> )
```

Onde *escopo* pode ser:

- `CLK_LOCAL_MEM_FENCE`: escopo *local*
- `CLK_GLOBAL_MEM_FENCE`: escopo *global*
- ou `CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE`

Todos os itens de trabalho de *um grupo* devem atingir este ponto do *kernel* para que a execução continue.

Primitiva de Sincronia



Exemplo Ilustrativo

Exemplo Ilustrativo

```
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

Exemplo Ilustrativo

```
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

Pontos de Sincronia

Considerações:

- Sincronia afeta negativamente o desempenho:

  itens de trabalho no ponto de sincronia aguardam ociosamente pelos demais

- Pontos de sincronia devem ser escolhidos com cautela:

  se um item de trabalho (de um grupo) não atinge o *barrier* a execução para indefinidamente:

```
kernel void deadlock( global float * x )
{
   int i = get_global_id(0);
   if( i == 0 )
      barrier( CLK_LOCAL_MEM_FENCE );
   else
      x[i] = i;
}
```

Computação Heterogênea: Modelagem de um Problema

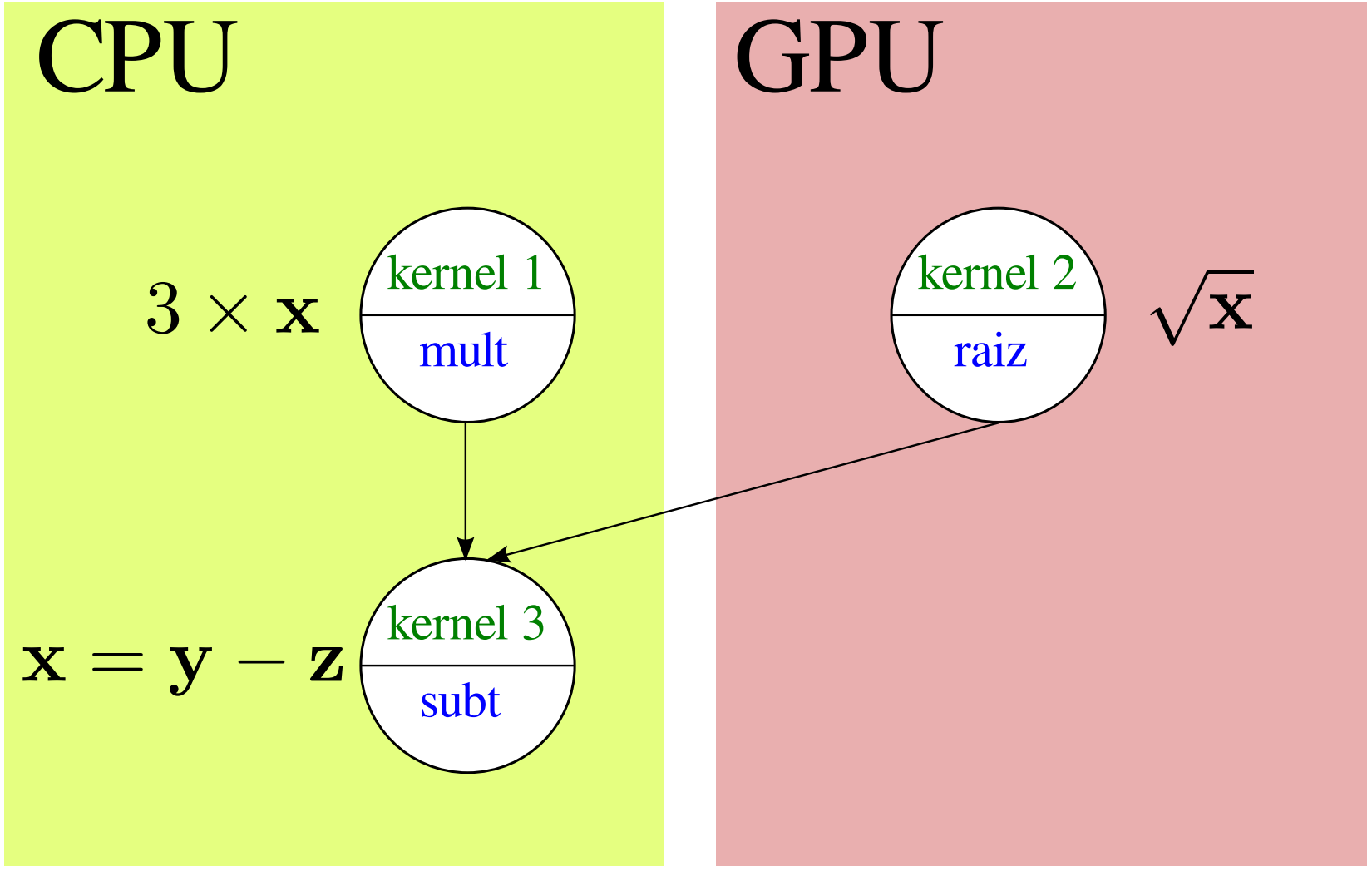
Problema

Computar:

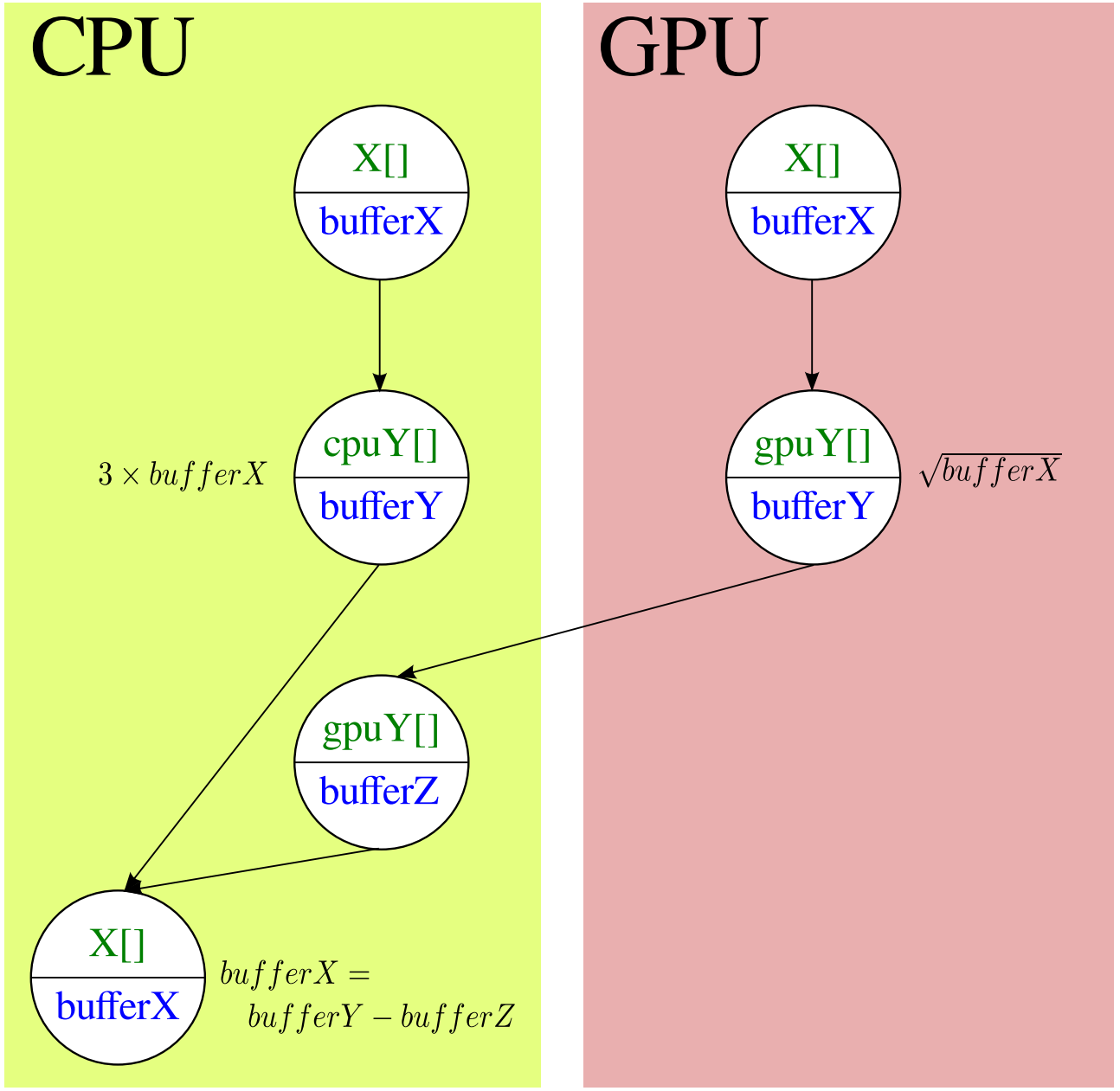
$$\mathbf{x} = 3\mathbf{x} - \sqrt{\mathbf{x}}$$

- Tempo 1:
  CPU computa paralelamente $\mathbf{y} = 3\mathbf{x}$
  GPU computa paralelamente $\mathbf{z} = \sqrt{\mathbf{x}}$
- Tempo 2:
  CPU computa paralelamente $\mathbf{x} = \mathbf{y} - \mathbf{z}$

Kernels e Dependências



Buffers e Dependências



Kernels

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

__kernel void
mult( __global const float * x, __global float * y, float s )
{
   int i = get_global_id(0);
   y[i] = s * x[i];
}

__kernel void
subt( __global float * x, __global const float * y, __global const float * z )
{
   int i = get_global_id(0);
   x[i] = y[i] - z[i];
}
```

Memórias no Hospedeiro

```
// Aloca as memórias para os vetores X, cpuY e gpuY, e faz cada elemento do
// vetor X ter o valor do seu próprio índice
float*X = new float[elementos];
float*cpuY = new float[elementos];
float*gpuY = new float[elementos];

for(int i =0; i < elementos; ++i ) X[i] = i;
```

Plataformas, Contextos e Filas

```
// Descobrir e escolher as plataformas e dispositivos
vector<cl::Platform> plataformas;
vector<cl::Device> cpu_dispositivos, gpu_dispositivos;

// Descobre as plataformas instaladas no hospedeiro
cl::Platform::get( &plataformas );
// Descobre os dispositivos CPU e GPU: para simplificar, vamos procurar
// apenas os dispositivos da primeira plataforma (plataformas[0])
plataformas[0].getDevices( CL_DEVICE_TYPE_CPU, &cpu_dispositivos );
plataformas[0].getDevices( CL_DEVICE_TYPE_GPU, &gpu_dispositivos );

// Criar os contextos
cl::Context cpu_contexto( cpu_dispositivos );
cl::Context gpu_contexto( gpu_dispositivos );

// Criar as filas de comandos para cada arquitetura (o primeiro dispositivo)
cl::CommandQueue cpu_fila( cpu_contexto, cpu_dispositivos[0] );
cl::CommandQueue gpu_fila( gpu_contexto, gpu_dispositivos[0] );
```

Programas e Kernels

```
// Carregar os programas, compilá-los e gerar os kernels
cl::Program::Sources fonte(1,make_pair( kernel_str,strlen( kernel_str ) ) );
cl::Program cpu_programa( cpu_contexto, fonte );
cl::Program gpu_programa( gpu_contexto, fonte );

// Compila para todos os dispositivos associados a '[cpu|gpu]_programa'
// através de '[cpu|gpu]_contexto': vector<cl::Device>() é um vetor nulo
cpu_programa.build( vector<cl::Device>() );
gpu_programa.build( vector<cl::Device>() );

// Cria os objetos que representarão cada um dos três kernels
cl::Kernel kernel_mult( cpu_programa,"mult");
cl::Kernel kernel_subt( cpu_programa,"subt");
cl::Kernel kernel_raiz( gpu_programa,"raiz");
```

Buffers Iniciais

```
// Preparação da memória dos dispositivos (leitura e escrita}
cl::Buffer cpu_bufferX( cpu_contexto, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
                        elementos *sizeof(float), X );
cl::Buffer gpu_bufferX( gpu_contexto, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                        elementos *sizeof(float), X );

cl::Buffer cpu_bufferY( cpu_contexto, CL_MEM_READ_WRITE, elementos *sizeof(float) );
cl::Buffer gpu_bufferY( gpu_contexto, CL_MEM_WRITE_ONLY, elementos *sizeof(float) );
```

Execução dos Kernels mult e raiz

```
// Execução dos kernels: definição dos argumentos e trabalho/particionamento
kernel_mult.setArg(0, cpu_bufferX );
kernel_mult.setArg(1, cpu_bufferY );
kernel_mult.setArg(2, float(3) );

kernel_raiz.setArg(0, gpu_bufferX );
kernel_raiz.setArg(1, gpu_bufferY );

// Paralelismo implícito: tamanho local é definido como "nulo"; a
// implementação é que vai decidir se divide em grupos e como dividi-los
cpu_fila.enqueueNDRangeKernel( kernel_mult, cl::NDRange(),
                               cl::NDRange( elementos ), cl::NDRange() );
gpu_fila.enqueueNDRangeKernel( kernel_raiz, cl::NDRange(),
                               cl::NDRange( elementos ), cl::NDRange() );

cpu_fila.flush();// força a execução dos comandos da fila
gpu_fila.flush();// força a execução dos comandos da fila
```

Coleta dos Resultados da GPU

```
// Transferência dos resultados da GPU para o hospedeiro (joga em gpuY)
// (comando bloqueante: CL_TRUE)
gpu_fila.enqueueReadBuffer( gpu_bufferY, CL_TRUE,0,
                            elementos *sizeof(float), gpuY );

// Criar um buffer na CPU com os resultados oriundos da GPU
cl::Buffer cpu_bufferZ( cpu_contexto, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                        elementos *sizeof(float), gpuY );
```

Coleta dos Resultados Finais

```
// Execução do kernel final: definição dos argumentos e trabalho/particionamento
kernel_subt.setArg(0, cpu_bufferX );
kernel_subt.setArg(1, cpu_bufferY );
kernel_subt.setArg(2, cpu_bufferZ );

cpu_fila.enqueueNDRangeKernel( kernel_subt, cl::NDRange(),
                               cl::NDRange( elementos ), cl::NDRange() );

// Transferência dos resultados da CPU para o hospedeiro (joga em X)
// (comando bloqueante: CL_TRUE)
cpu_fila.enqueueReadBuffer( cpu_bufferX, CL_TRUE,0, elementos *sizeof(float), X );
```

Impressão e Limpeza

```
// Impressão do resultado
for( int i =0; i < elementos; ++i ) cout <<'['<< X[i] <<']'; cout << endl;

// Limpeza (as variáveis específicas do OpenCL já são automaticamente destruídas)
delete[] X, cpuY, gpuY;
```

Consulta de Propriedades

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];
```

Referências

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 Specification**

  `http://www.khronos.org/opencl/`