OpenCL

FRANCESCO GAROFALO

PROFESSORESSA - ANGELAMARIA CARDONE

CORSO: METODI NUMERICI PER L'INFORMATICA

Index:

- Introduzione ad OpenCL
- Anatomia di OpenCL
- Architettura OpenCL
- Cuda vs OpenCL
- Esempi di Applicazioni in OpenCL

Index:

Introduzione a GPGPU

Le GPGPU, sigla di **general-purpose computing on graphics processing units** (letteralmente "calcolo a scopo generale su unità di elaborazione grafica"), si intende l'uso di un'unità di elaborazione grafica (GPU) per scopi diversi dal tradizionale utilizzo nella grafica computerizzata.

In sostanza, una pipeline GPGPU è una sorta di elaborazione parallela tra una o più GPU e CPU che analizza i dati come se fossero in un'immagine o in altra forma grafica. Mentre le GPU funzionano a frequenze più basse, in genere hanno molte volte il numero di core. Pertanto, le GPU possono elaborare molte più immagini e dati grafici al secondo rispetto a una CPU tradizionale.
La migrazione dei dati in forma grafica e quindi l'utilizzo della GPU per scansionarli e analizzarli può creare una grande accelerazione.

Cos'è OpenCL?

Open Computing Language (OpenCL) è un'API di elaborazione parallela open e royalty-free progettata per consentire a GPU e CPU di funzionare in parallelo.

Lo standard **OpenCL** 1.0 è stato rilasciato l'8 dicembre 2008 da The Khronos Group, un consorzio indipendente di standard.

OpenCL nasce dal problema dei programmatori di avere linguaggi di programmazione proprietari, limitando la loro capacità di scrivere applicazioni multipiattaforma indipendenti dal fornitore.

Implementazioni proprietarie come **CUDA** funzionante solo su hardware NVIDIA costringevano i programmatori a dover riprogrammare l'applicazione su altri sistemi hardware.

Cenni Storici

Lo standard è stato originariamente proposto da Apple, successivamente ratificato dalla stessa assieme alle principali aziende del settore (Intel, NVIDIA, AMD), e infine portato a compimento dal consorzio no-profit Khronos Group.
OpenCL è supportato su schede video AMD e NVIDIA e su processori ARM.
Per quanto riguarda i sistemi operativi, è certificato su OSX, Linux, Windows.

La prima versione finale di OpenCL (versione 1.0) è stata rilasciata in data 18 novembre 2008 e rilasciata ufficialmente l'8 dicembre 2008.
La versione attuale è la 3.0, ma la versione stabile attualmente utilizzata è la 2.2.

Vantaggi

1. **Accelerazione parallela nei processi paralleli;**
2. **Cross-vendor;**
3. **Semplicità di migrazione su altre piattaforme;**

Index:

Anatomia OpenCL

Il framework OpenCL si divide in 3 parti principali:

1. **Specifica del linguaggio**
2. **Livello Platform API**
3. **Runtime API**

Anatomia OpenCL

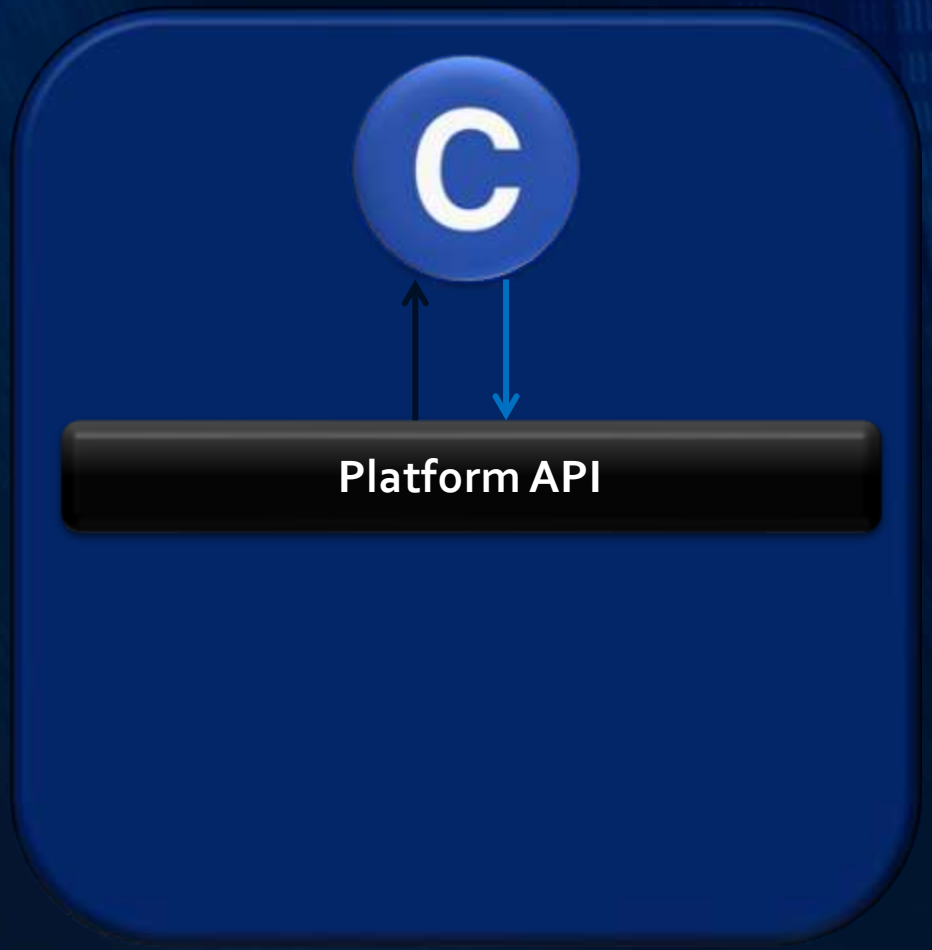
1. Specifica del linguaggio

Le specifiche del linguaggio descrivono la sintassi e l'interfaccia di programmazione per la scrittura di programmi kernel eseguiti sull'acceleratore supportato (GPU, CPU multi-core, o DSP).

I kernel possono essere precompilati o lo sviluppatore può consentire il runtime OpenCL per compilare il programma del kernel in fase di esecuzione.

Il linguaggio di programmazione usato per OpenCL è C.

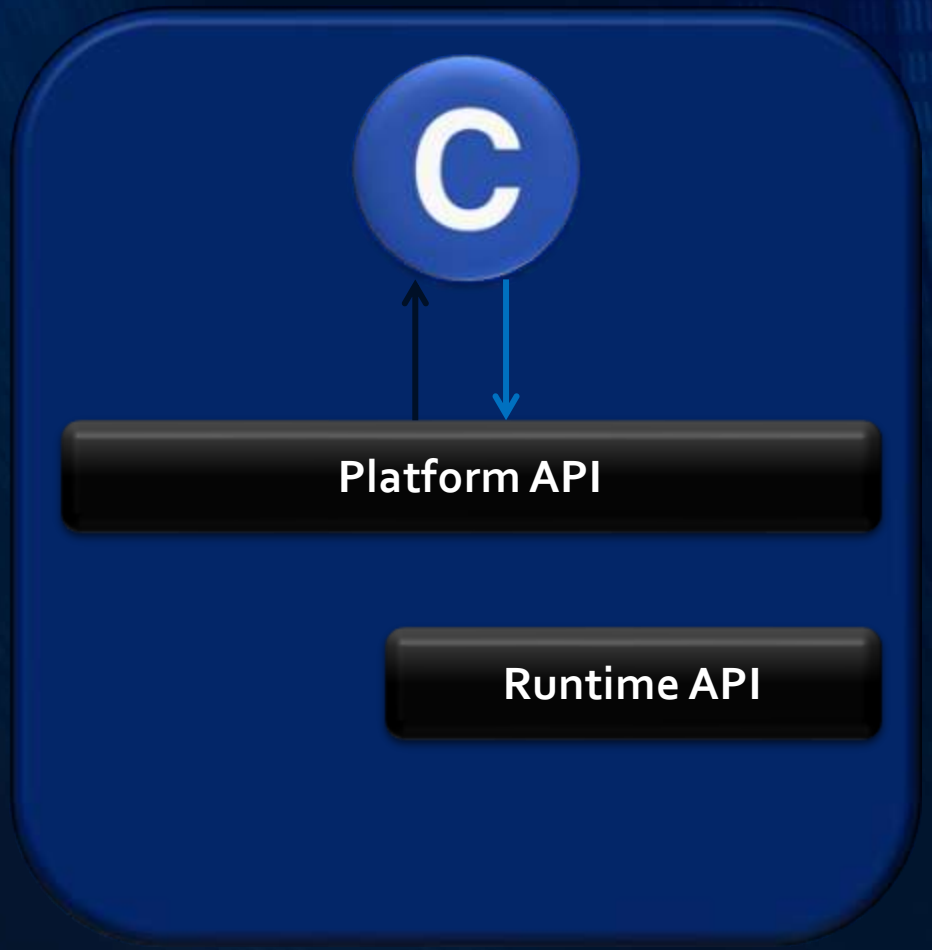
2. Platform API



L'API platform level offre allo sviluppatore l'accesso a routine di applicazioni software che possono interrogare il sistema per l'esistenza di dispositivi supportati da OpenCL.

Questo livello consente inoltre allo sviluppatore di utilizzare i concetti di contesto del dispositivo e le code di lavoro per selezionare e inizializzare i dispositivi OpenCL, inviare il lavoro ai dispositivi e consentire il trasferimento dei dati da e verso i dispositivi.

3. Runtime API



Il framework OpenCL utilizza i contesti per gestire uno o più dispositivi OpenCL.

L'API di runtime utilizza contesti per la gestione di oggetti come code di comandi, oggetti di memoria e oggetti del kernel, nonché per eseguire kernel su uno o più dispositivi specificati nel contesto.

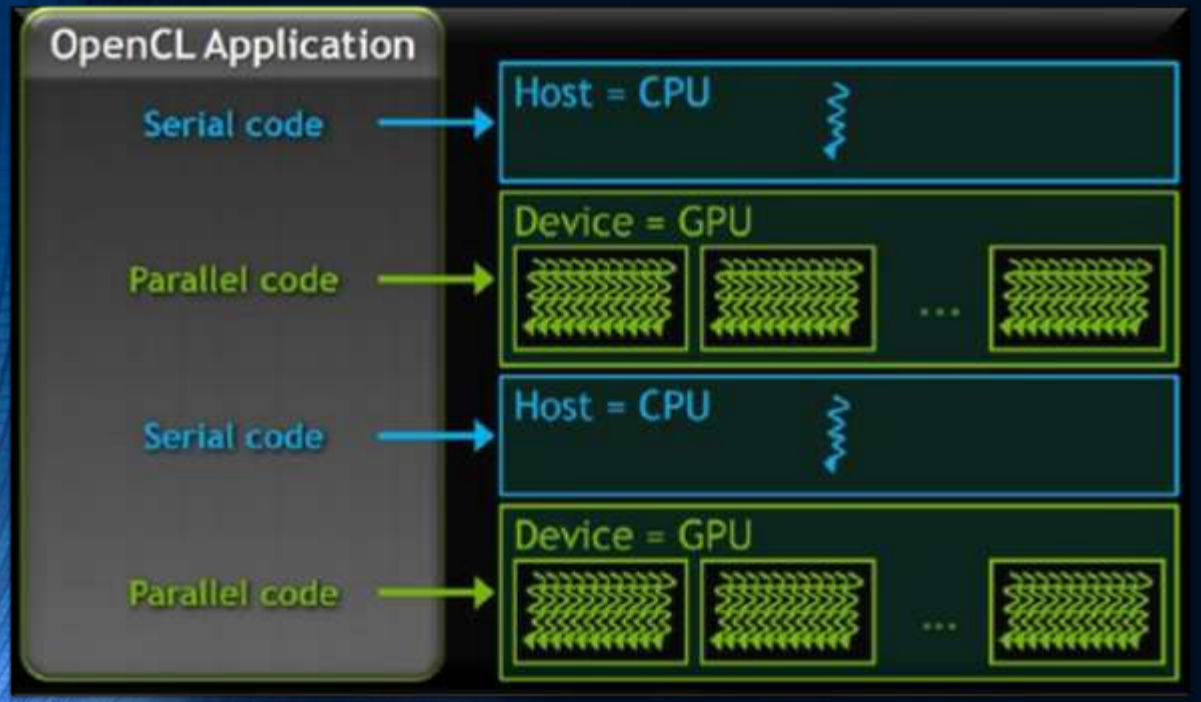
Index:

OpenCL – Architettura

Il modello OpenCL è simile al modello Cuda e può essere descritto grazie a quattro concetti chiave:

1. Platform Model – Alto livello, interazione host/device;
2. Execution Model – Programmi OpenCL eseguiti su host/device;
3. Memory Model – Differenti risorse di memoria sul device;
4. Programming Model – Tipi di carichi di lavoro paralleli;

Platform Model
Terminologia OpenCL



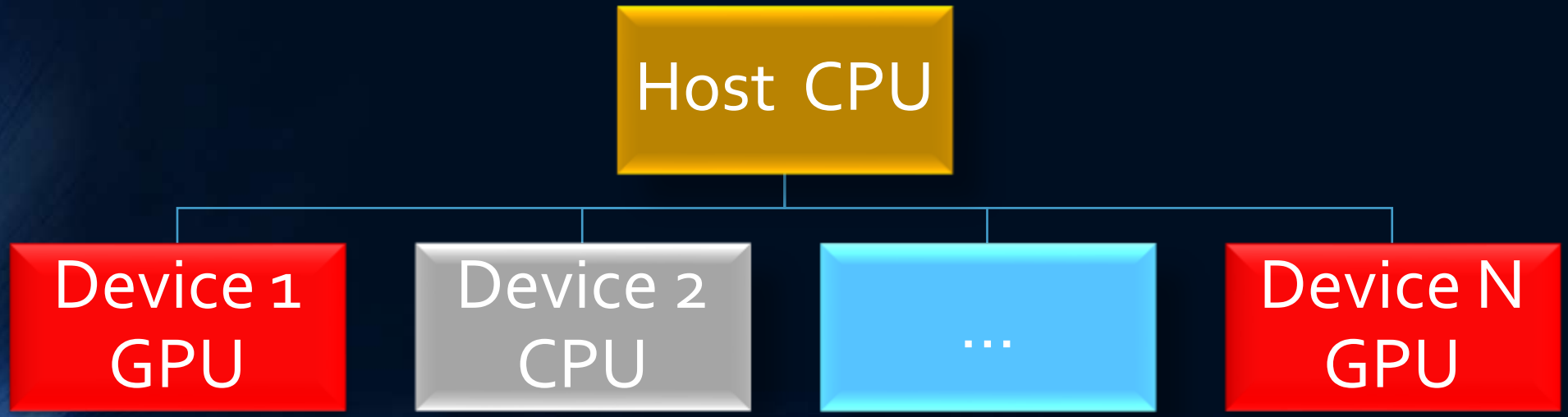
Una piattaforma OpenCL è composta da:

- **Work Item** sono le unità base di lavoro di un device OpenCL;
- i **Kernel**, il codice che vengono eseguiti sul work item;
- un programma **Host** che gestisce l'esecuzione di kernel;
- **I Device** OpenCL possono essere una GPU, DSP, CPU multicore.
- Le **Compute Unit** sono le parti del Device che possono avere 1 o più core (multi processori);
- **Context**: L'ambiente in cui vengono eseguiti i work item.
- I thread vengono raggruppati in blocchi chiamati **Work Group**;

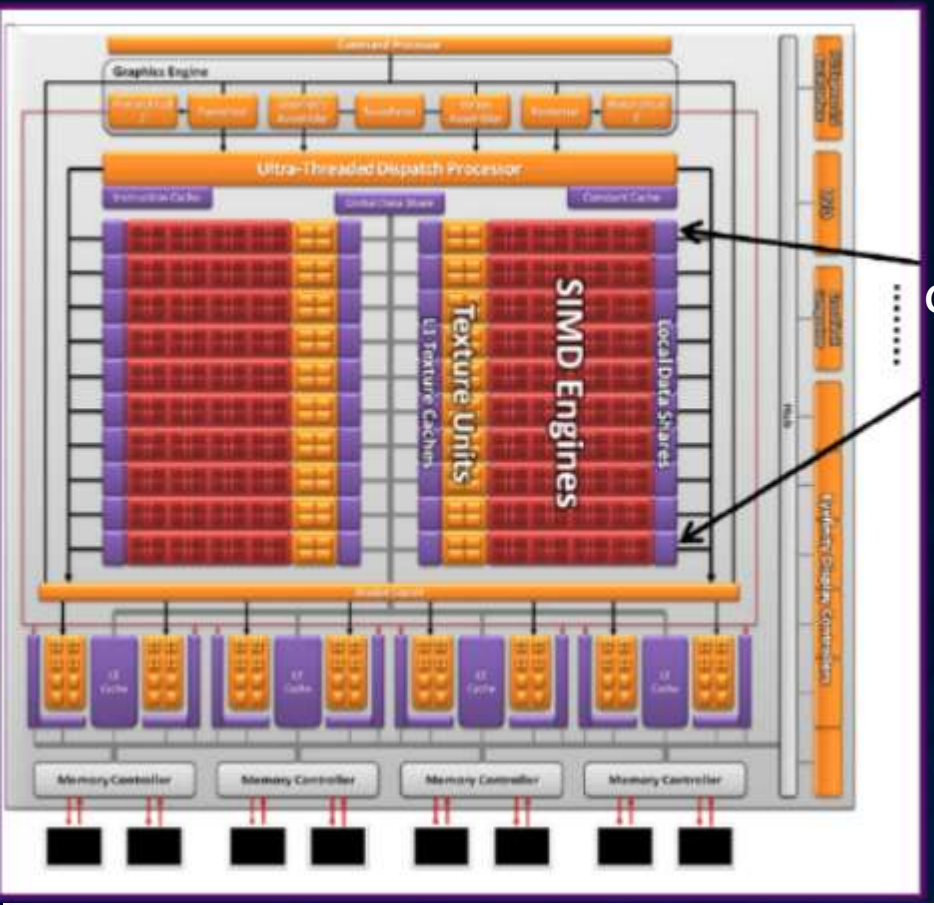
Platform Model
Struttura base di OpenCL

Gli elementi di elaborazione eseguono le istruzioni come SIMD o SPMD.

Le istruzioni SPMD vengono in genere eseguite su dispositivi generici come le CPU, mentre le istruzioni SIMD richiedono un processore vettoriale come una GPU o unità vettoriali in una CPU.



Platform Model – Esempio AMD

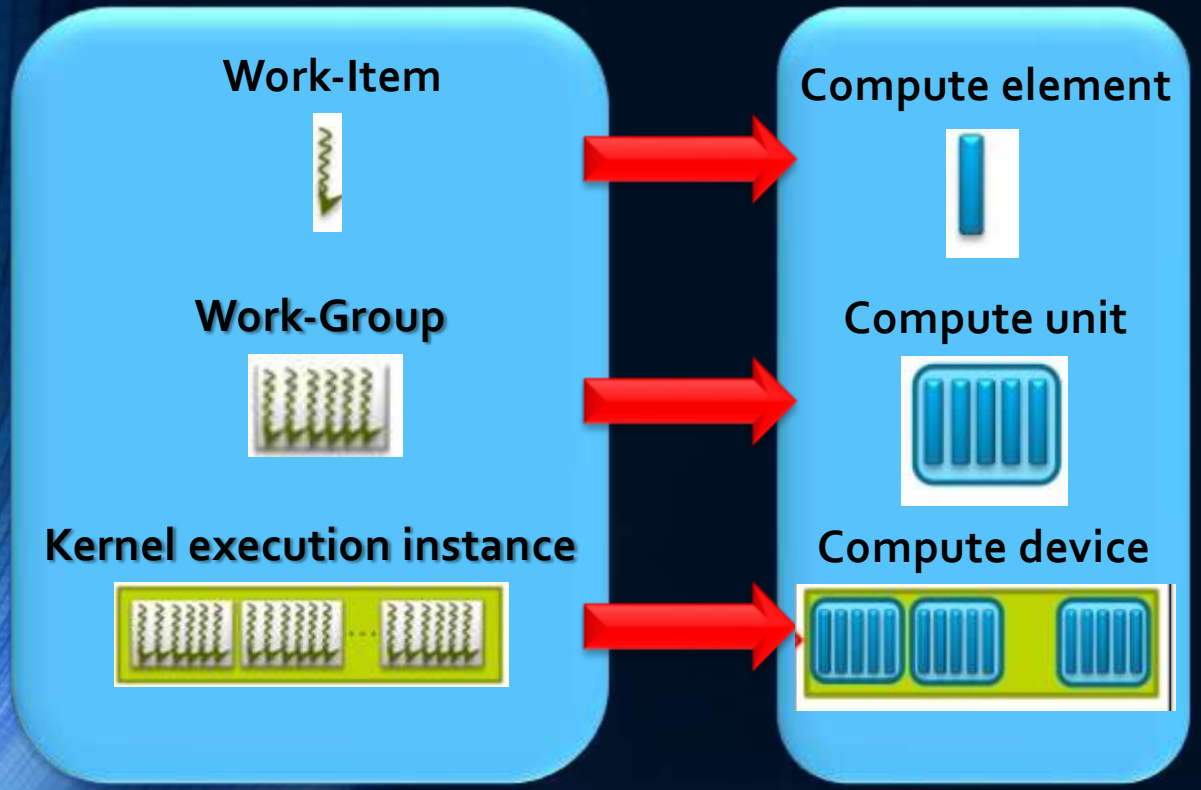


Compute Unit

La GPU ATI Radeon HD 5870 è composta da 20 unità SIMD, che si traducono in 20 unità di calcolo in OpenCL.

Ogni unità SIMD contiene 16 stream core e ogni stream core ospita cinque elementi di elaborazione. Pertanto, ogni unità di calcolo nell'ATI Radeon HD 5870 ha 80 (16 × 5) elementi di elaborazione.

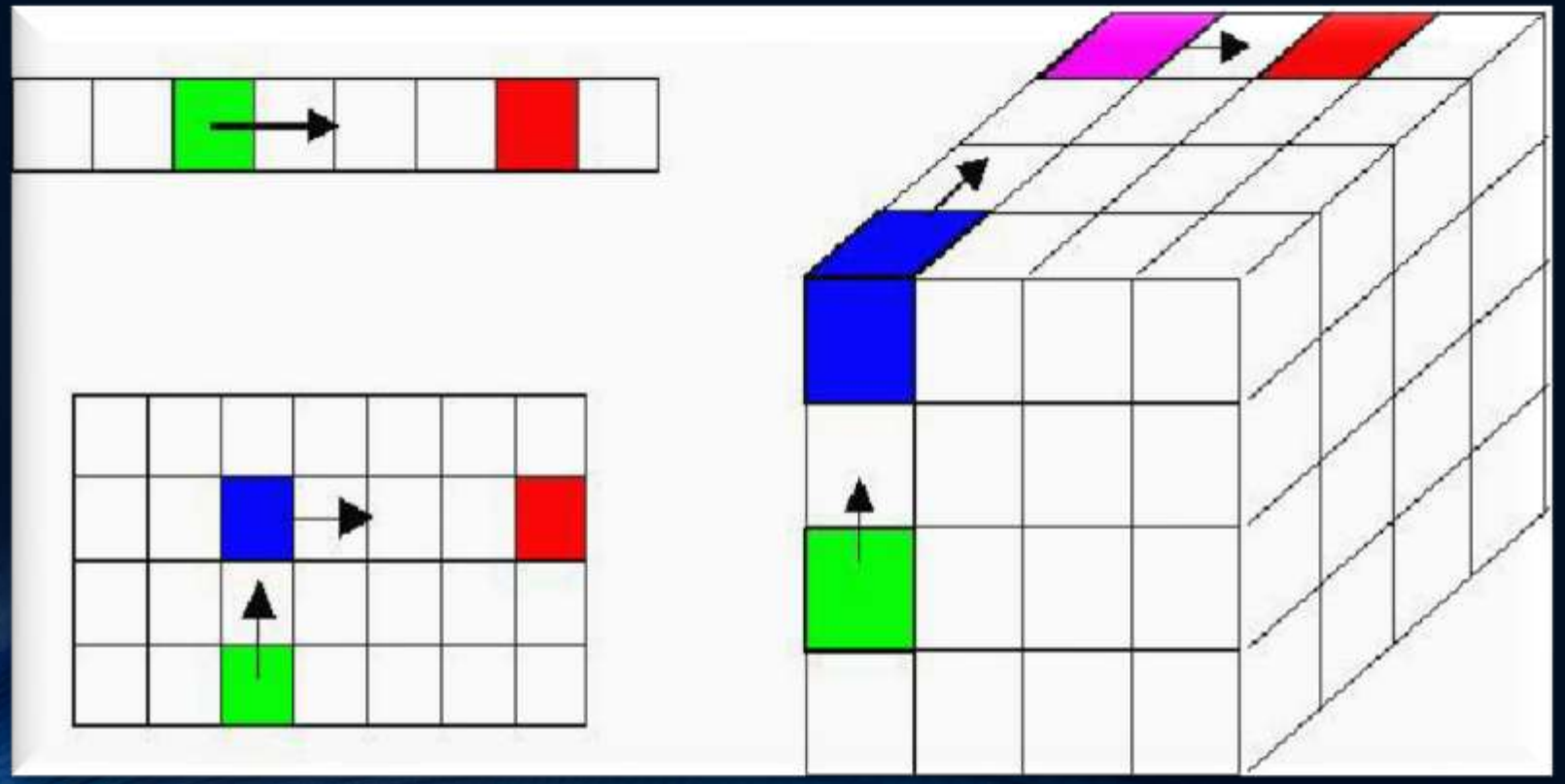
Execution Model
Platform Model vs Execution Model



- Ogni work-item è eseguito in un compute element;
- Ogni work-group è eseguito in un compute unit*;
- Ogni Kernel viene eseguito in un Compute Device;

Execution Model
Multidimensional

OpenCL sfrutta il calcolo parallelo su dispositivi di calcolo definendo il problema in uno spazio di indice N-dimensionale.
Lo spazio dell'indice N-dimensionale può essere N = 1, 2 o 3.



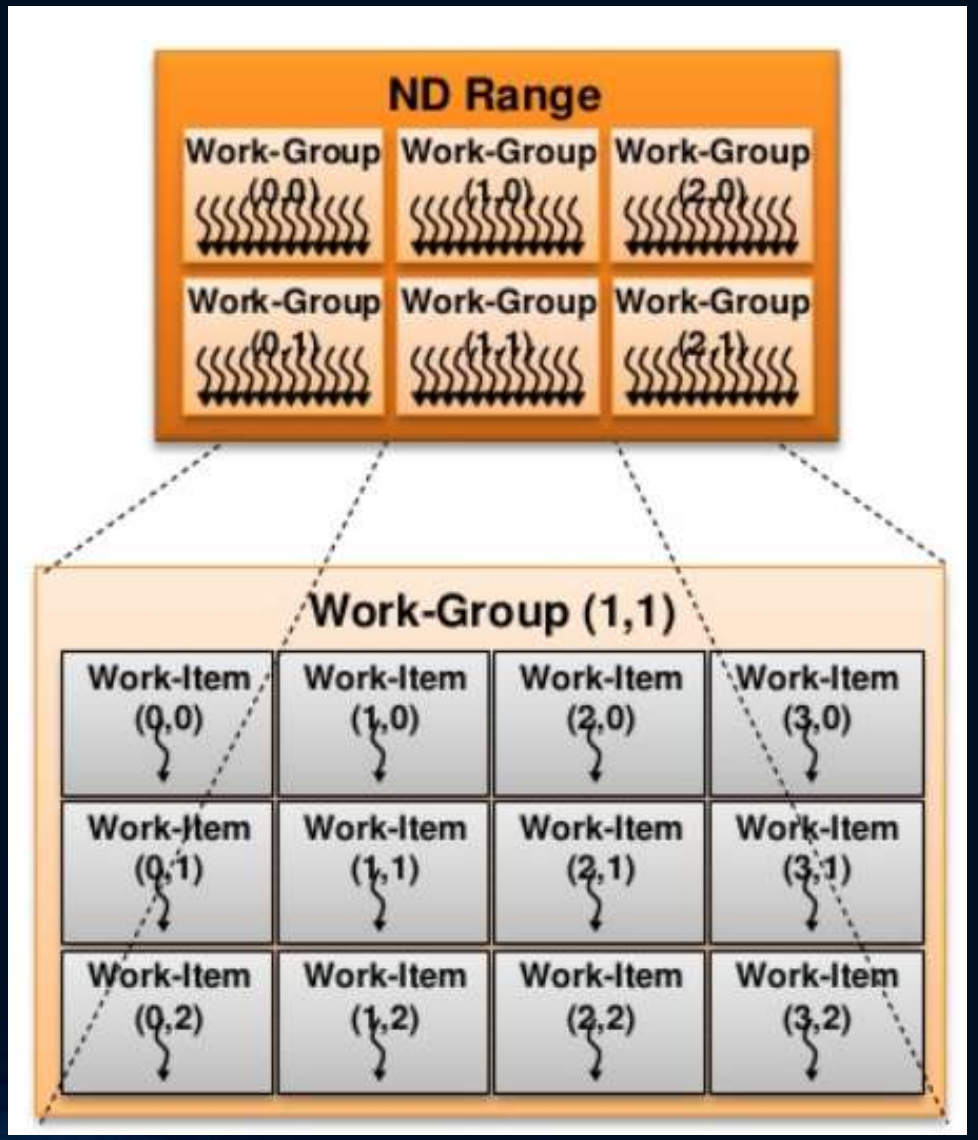
Execution Model

Modello di programmazione OpenCL

Quando viene lanciata una funzione del **kernel**, il suo codice viene eseguito da **work item**, (che corrispondono ai thread CUDA).

Un **index space** definisce gli elementi di lavoro e il modo in cui i dati vengono associati agli elementi di lavoro.

OpenCL raggruppa work items in work group, la cui dimensione è definita dal local index space.

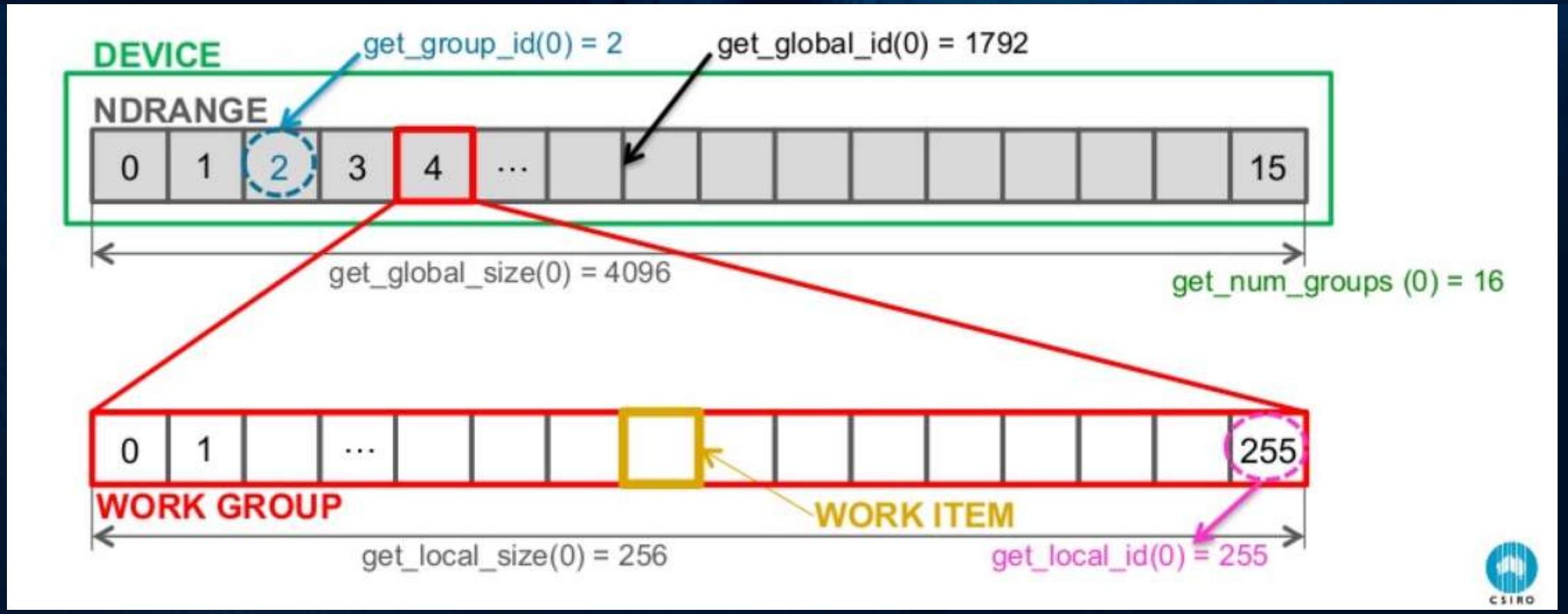


Execution Model

Esempio N=1

L'esempio mostra un vettore di N=1 dimensione, dove abbiamo 4096 work item suddivisi in 16 work group da 256 work item ciascuno.

Per individure un global_id = get_local_size()*get_group_id()+get_local_ID.get_local_size();



Execution Model

Trovare un global_id in OpenCL con N=1


```
__global__
void vecAddKernel (float *A,
                   float *B,
                   float * C,
                   int n)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < n)
    {
        C[index] = A[index] + B[index] ;
    }
}
```


```
_kernel void
addVector(_global const float *A,
          _global const float *B,
          _global float *C,
          int N)
{
    int index = get_global_id(0);

    if(index < N)
    C[index] = A[index]+B[index]
```

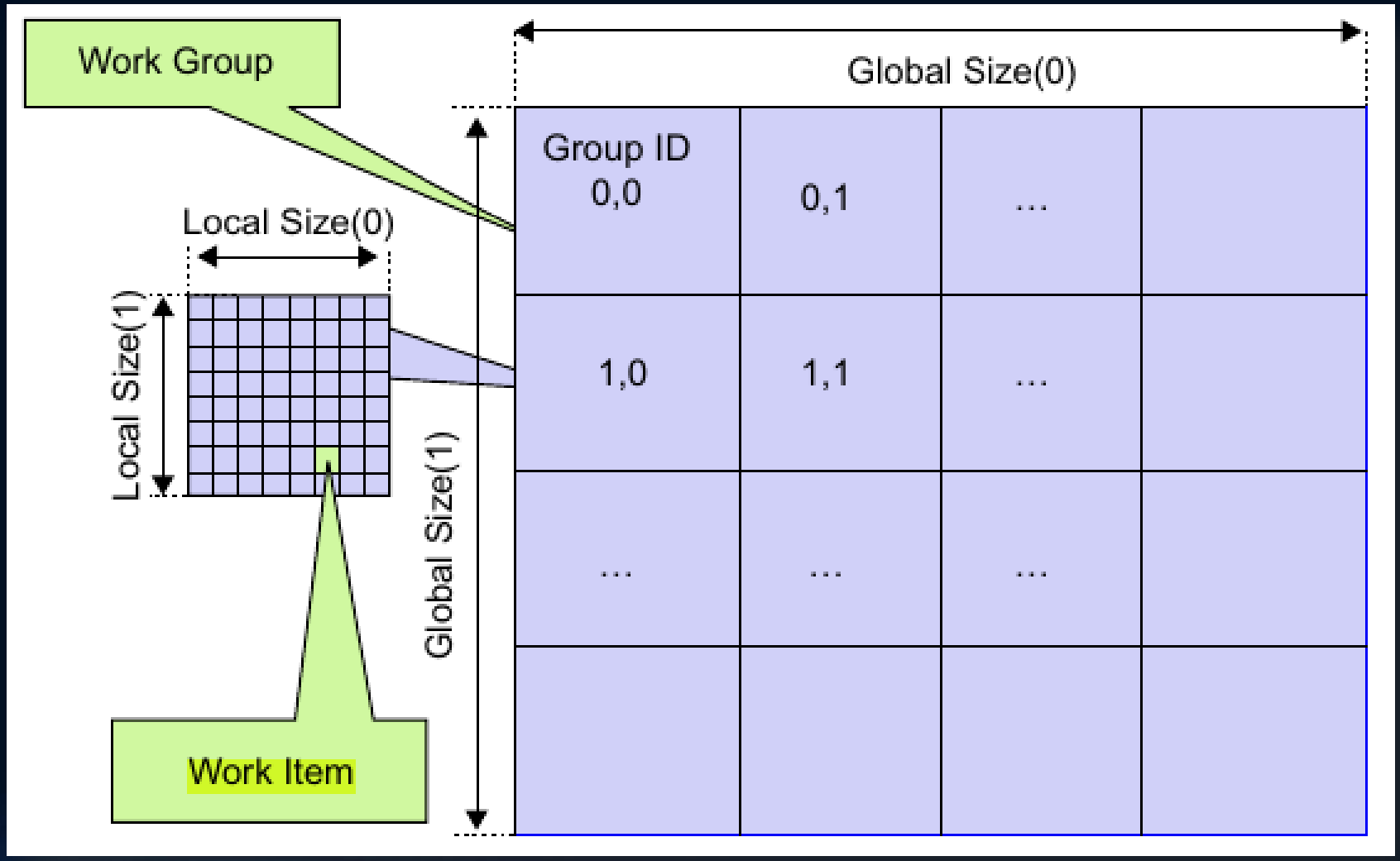
Execution Model

Modello di programmazione OpenCL N=2

In OpenCL i valori di indice globale univoci si ottengono con la funzione:

«get_global_id(0)» e «get_global_id(1)».

Per ottenere il valore di indice locale è necessario effettuare la chiamata e «get_local_id(0)» e «get_local_id(1)».

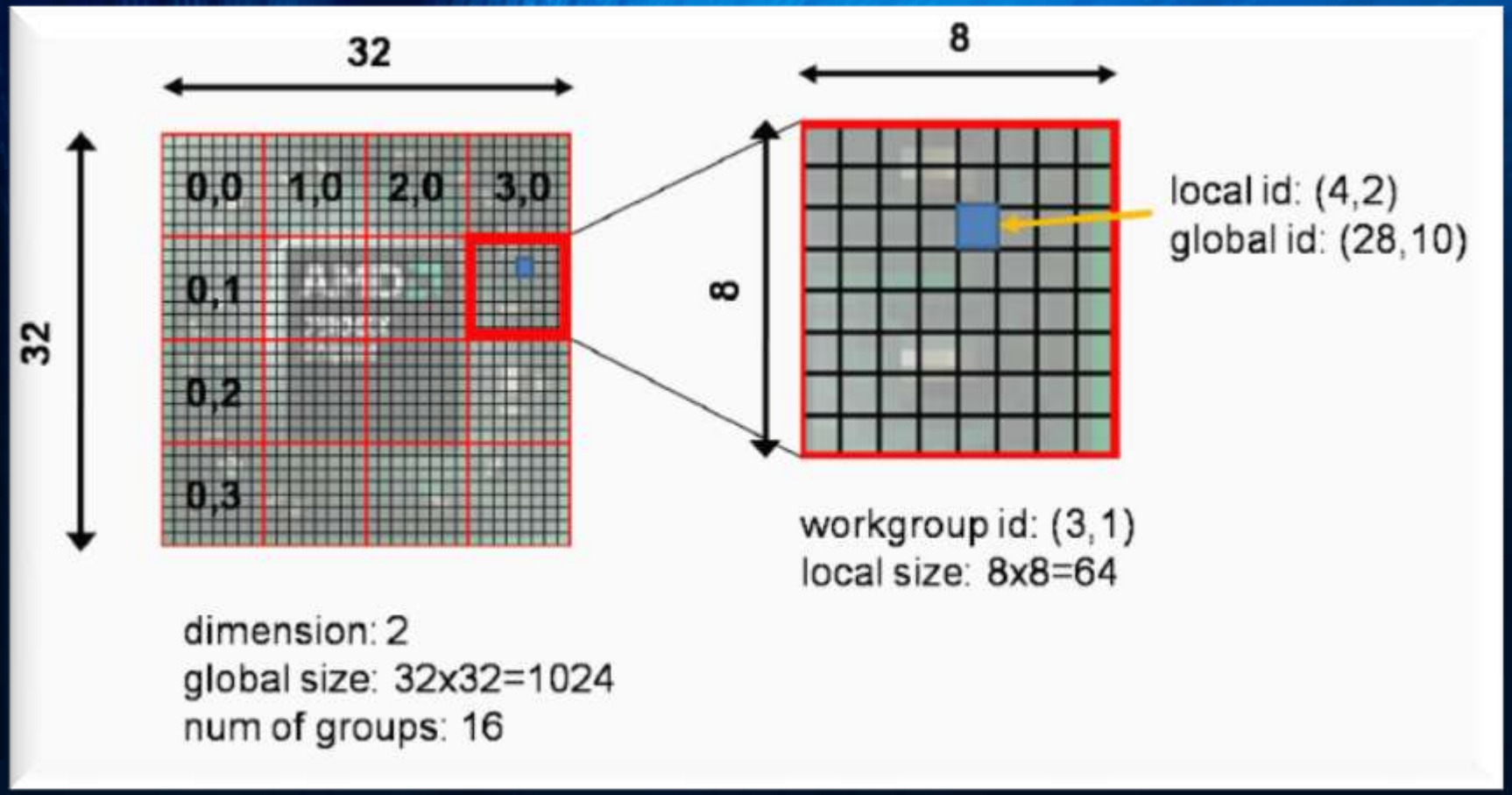


Execution Model
Esempio N=2

L'esempio mostra un'immagine bidimensionale con una dimensione di 1024 (32x32).

Il work group evidenziato ha un id (3,1).

Il work item interno al work group (3,1) ha un id locale (4,2).



Execution Model

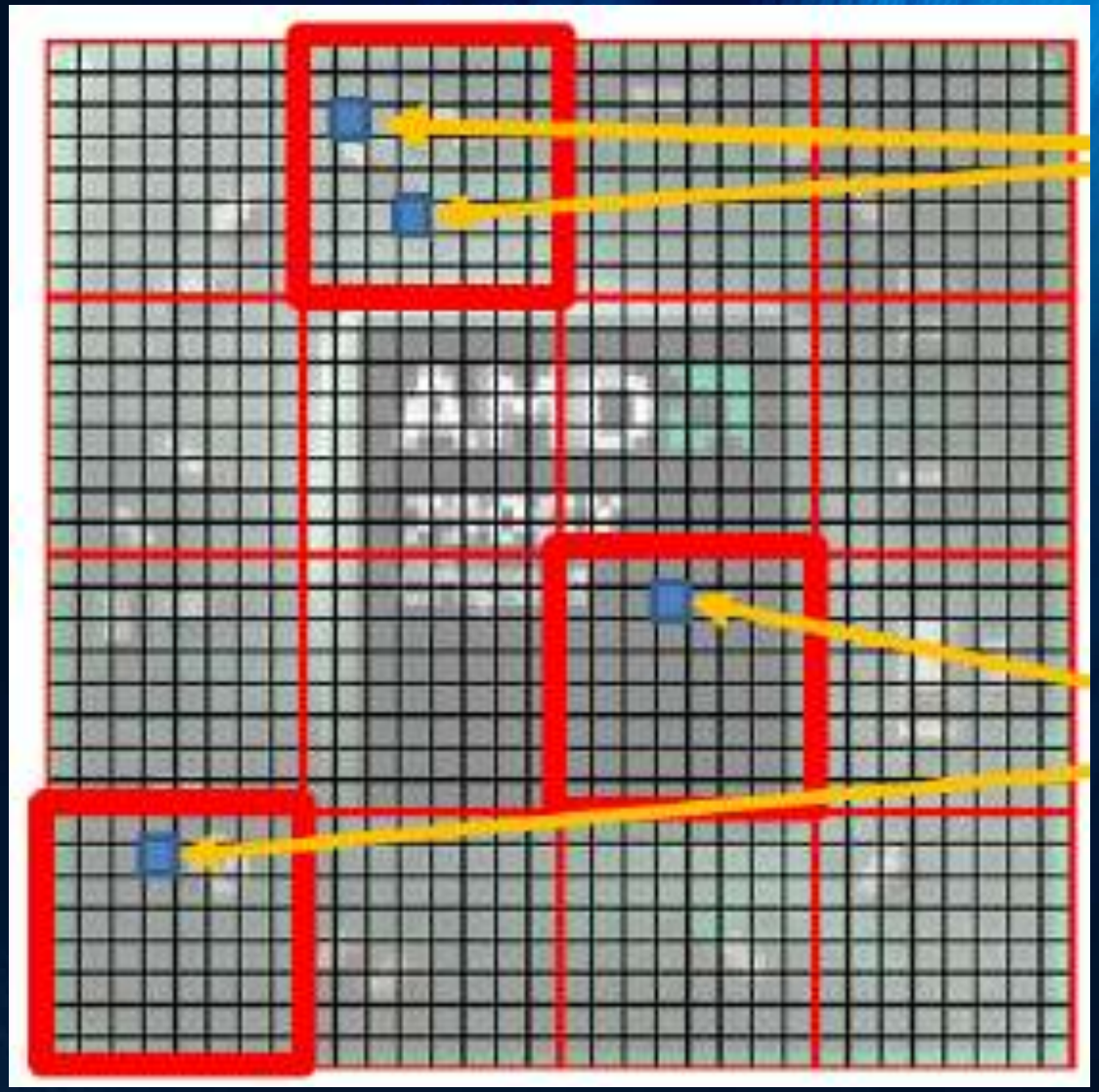
Trovare un global_id in OpenCL con N=2

In OpenCL per trovare il global_id di un work item è necessario conoscere:

- Global_id(0)
- Global_id(1)
- Global_size(0)
- Effettuare l'operazione: «global_size(0) * indice x + indice y

```
Device OpenCL
OpenCL:
__kernel voidclenergy(...) {
unsigned int xindex= get_global_id(0);
unsigned int yindex= get_global_id(1);
unsigned int outaddr= get_global_size(0) * UNROLLX
*yindex+xindex;
DEVICE CUDA
CUDA:
__global__ void cuenergy(...) {
Unsigned int xindex= blockIdx.x *blockDim.x +threadIdx.x;
unsigned int yindex= blockIdx.y *blockDim.y +threadIdx.y;
unsigned int outaddr= gridDim.x *blockDim.x *
UNROLLX*yindex+xindex
```

Execution Model
Sincronizzazione



La sincronizzazione è consentita tra work item appartenenti allo stesso work group

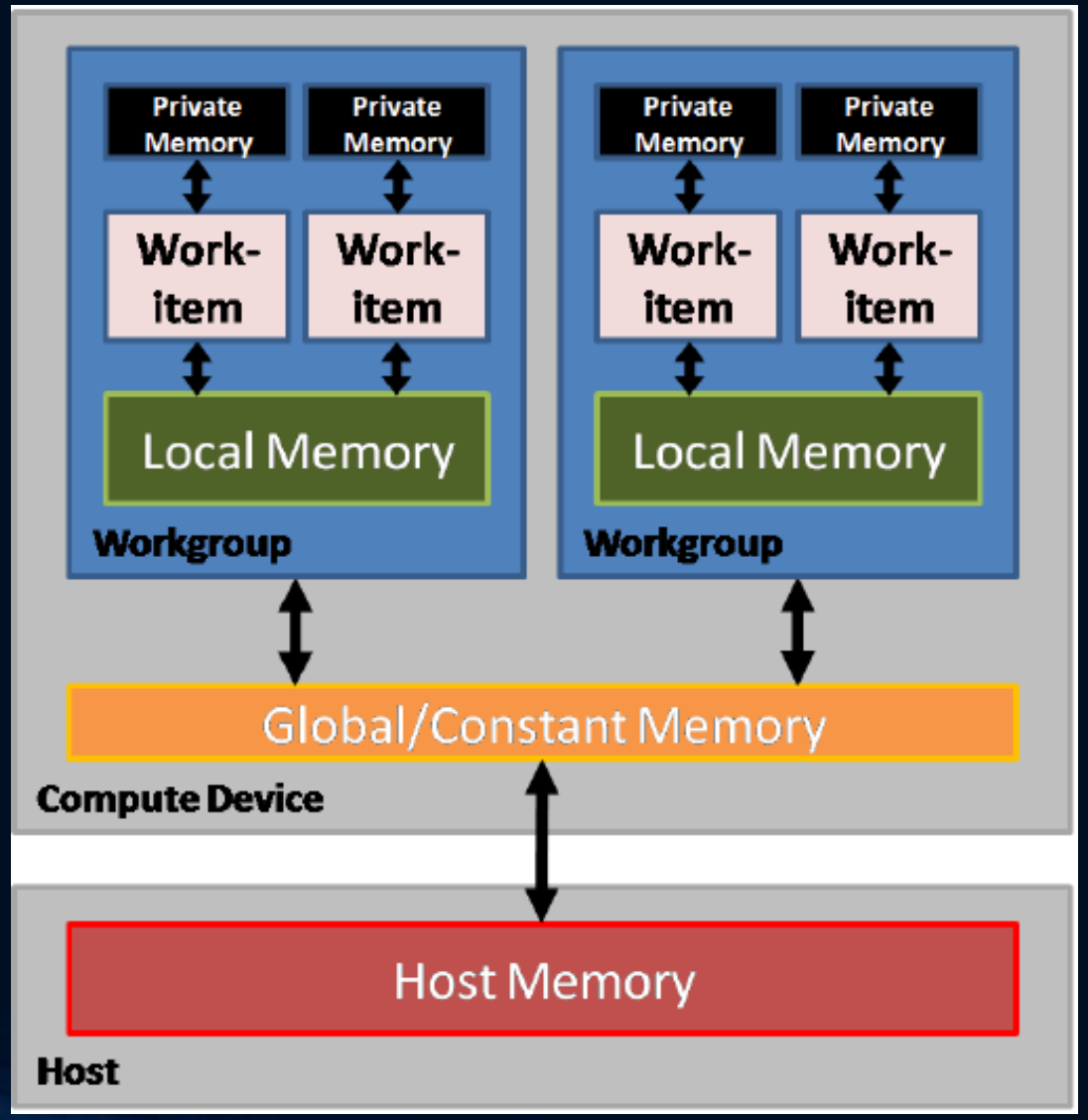
La sincronizzazione non è consentita in work item appartenenti a work group diversi.

Memory Model
Modello di programmazione OpenCL

Il modello di memoria OpenCL definisce quattro aree di memoria accessibili agli oggetti di lavoro durante l'esecuzione di un kernel.

Spazi di indirizzo:

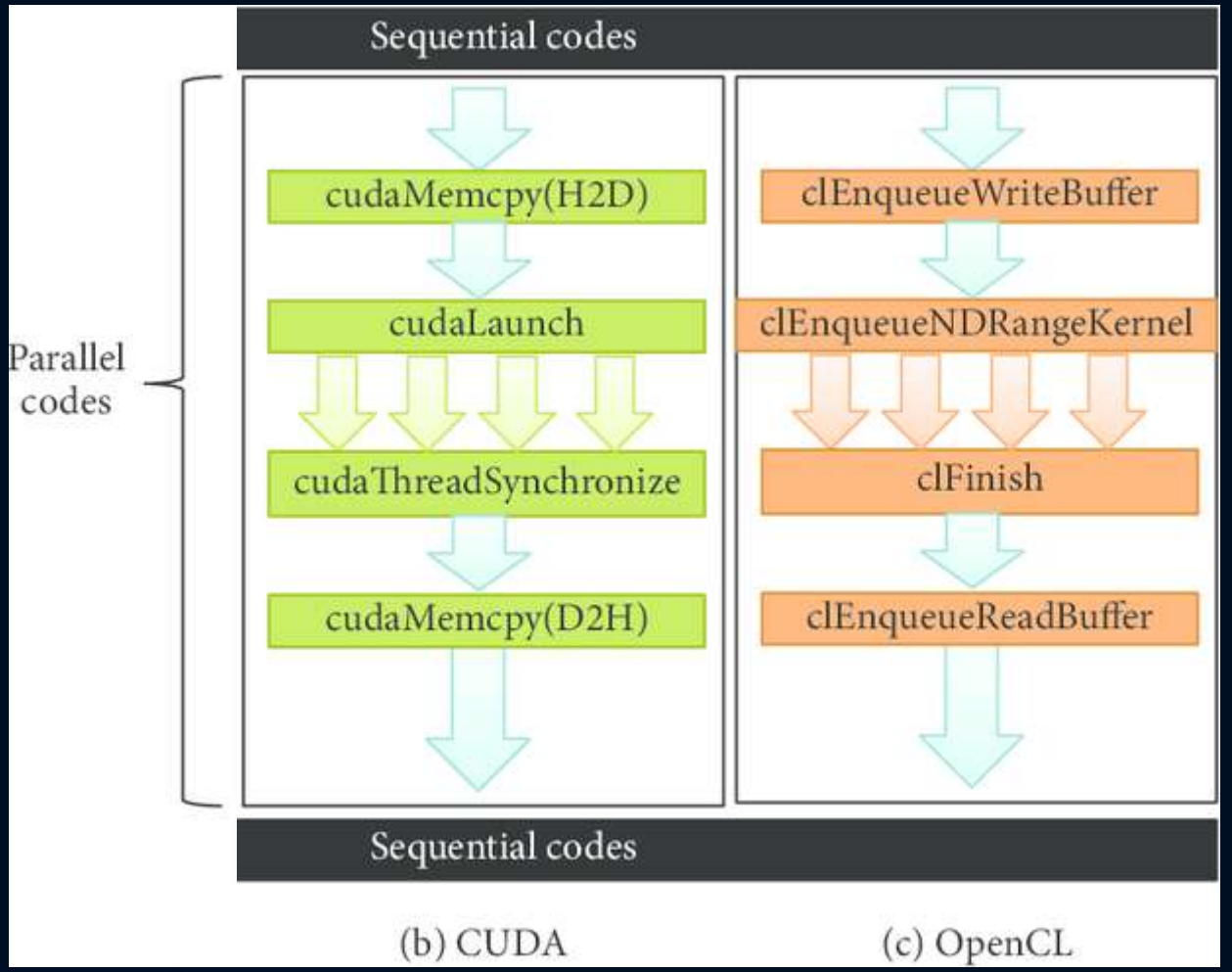
- **Privata**: accesso in lettura/scrittura per i work item;
- **Locale**: accesso in lettura/scrittura per l'intero work group;
- **Global/Costant**: memoria visibile a tutti i work-group;
- **Host**: accessibile solo dalla CPU.



Programming Model
Modello di programmazione OpenCL

L'host lancia i kernel, esegue il codice seriale e si occupa di:

- Gestione della memoria
- Scambio dei dati tra device e host
- Gestione degli errori



Index:

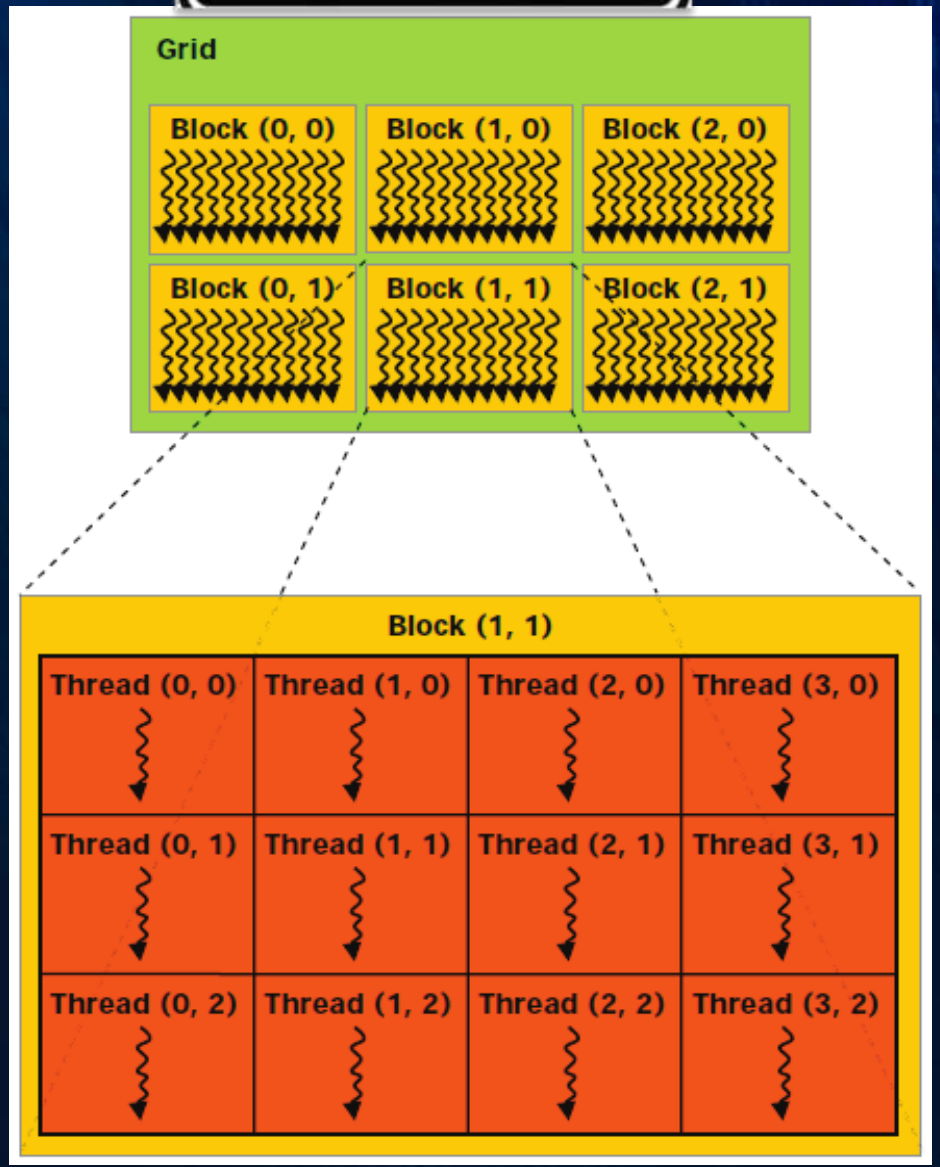
- Introduzione ad OpenCL
- Anatomia OpenCL
- Architettura OpenCL
- Cuda vs OpenCL
  - Terminologia
  - Struttura
  - Pro e Contro
- Esempi di Applicazioni OpenCL

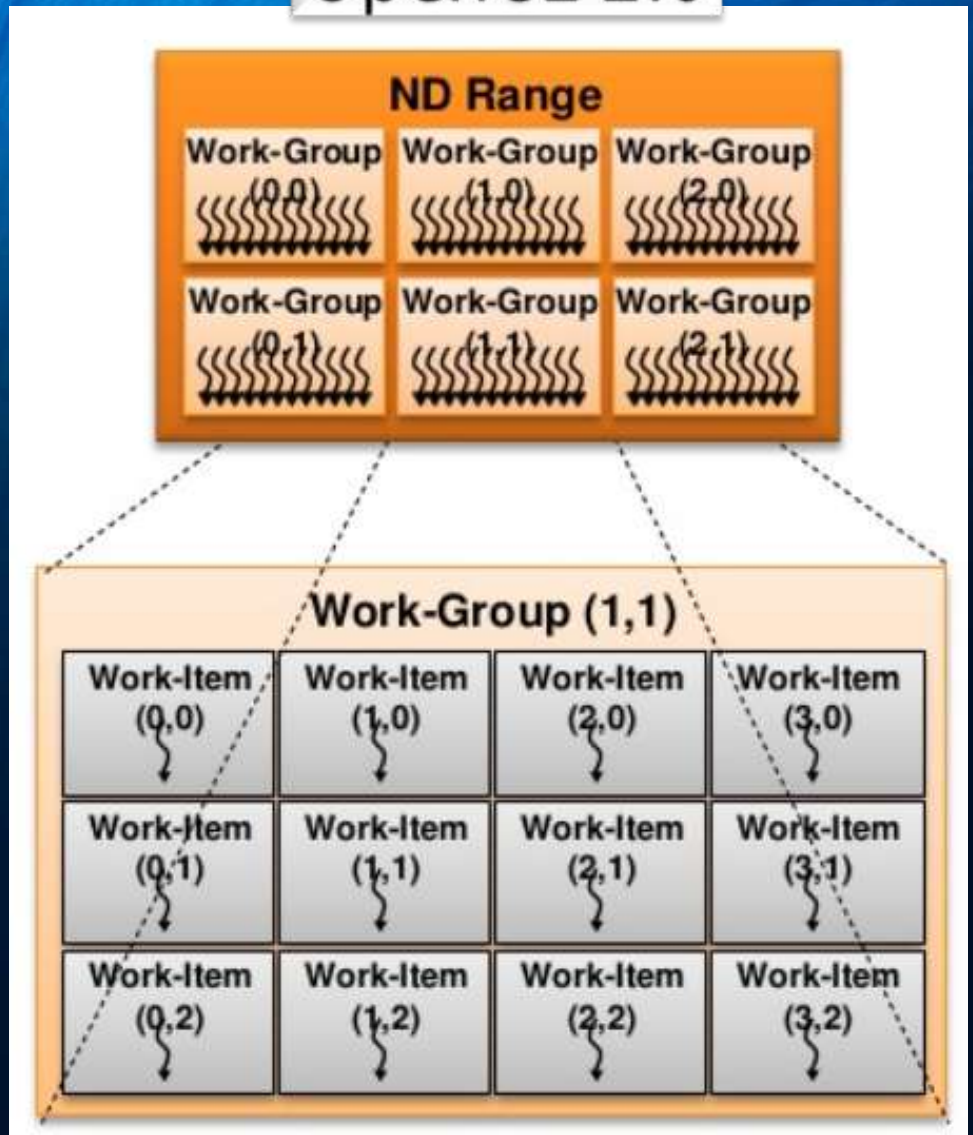
CUDA vs OpenCL
Terminologia

CUDA	OpenCL
thread	work-item
warp	wavefront
thread block	work-group
grid	computation domain
global memory	global memory
shared memory	local memory
local memory	private memory
streaming multiprocessor (SM)	compute unit
scalar core	processing element

Memory Type	Host Access	Device Access	CUDA Equivalent
Global memory	Dynamic allocation; read/write access	No allocation; read/write access by all work items in all work groups, large and slow, but may be cached in some devices	Global memory
Constant memory	Dynamic allocation; read/write access	Static allocation; read-only access by all work items	Constant memory
Local memory	Dynamic allocation; no access	Static allocation; shared read/write access by all work items in a work group	Shared memory
Private memory	No allocation; no access	Static allocation; read/write access by a single work item	Registers and local memory

CUDA vs OpenCL
Struttura





CUDA vs OpenCL
Pro e Contro

- Cuda è proprietario Nvidia;
- CUDA avvia i kernel definendo il numero di blocchi e il numero di thread per blocco;
- CUDA non è molto astratto e consente agli sviluppatori di ottimizzare in modo specifico per l'architettura Nvidia sottostante;
- Per CUDA, è possibile il controllo delle versioni dell'architettura e richiedere il numero di unità logiche in programmazione;
- CUDA è facile da entrare, difficile da uscire.
- È più difficile passare da CUDA a OpenCL

- OpenCL è cross-vendor;
- OpenCL avvia i kernel definendo il numero totale di thread e il numero di thread per blocco;
- OpenCL estrae gran parte dell'hardware dallo sviluppatore. Quindi lo sviluppatore scrive solo codice parallelizzato e sintonizza finemente i parametri dei parametri di lancio;
- OpenCL non sa quanti core contiene una GPU;
- Usando OpenCL, migrare su altre piattaforme è facile;
- OpenCL è difficile da inserire, difficile da uscire;
- È più semplice passare da OpenCL a CUDA;

Index:

- Introduzione ad OpenCL
- Anatomia OpenCL
- Architettura OpenCL
- Cuda vs OpenCL
- Esempi di Applicazioni OpenCL
  - Installare l'ambiente
  - Compilare ed Eseguire applicazioni
  - Programmare in OpenCL
  - HelloWorldCL
  - Funzioni
  - Review Ciclo di Vita

Installare l'ambiente OpenCL

Da terminale:

```
sudo apt update
sudo apt install ocl-icd-opencl-dev
```

Se non si ha un acceleratore grafico dedicato installare i seguenti pacchetti:

- sudo apt-get install beignet clinfo
- sudo apt-get install beignet-dev
- sudo apt-get install clinfo ocl-icd-opencl-dev opencl-headers

Per macchine che supportano versioni di OpenCL < 2.0 è necessario installare ulteriori pacchetti. *(Rif. Documentazione OpenCL)*

Scaricare il pacchetto: https://github.com/GPUOpen-LibrariesAndSDKs/OCL-SDK/releases

Eseguire ed Installare il file OCL_SDK_Light_AMD.exe
(funziona anche su dispositivi Intel)

Compilare ed Eseguire OpenCL file

Se non si vuole utilizzare nessun acceleratore grafico dedicato è sufficiente digitare:

- gcc -o outputfile inputfile.c -lOpenCL

Per l'esecuzione:

- ./outputfile

Flusso di un programma in OpenCL

1. Ottenere l'elenco di piattaforme disponibili
2. Seleziona dispositivo
    1. Crea contesto
    2. Crea coda comandi
    3. Crea oggetti di memoria
    4. Leggi il file del kernel
    5. Crea oggetto programma
        1. Compilare il kernel (runtime)
        2. Crea un oggetto kernel
        3. Imposta argomenti del kernel
        4. Esegui kernel (attività Enqueue) ← Qui viene chiamata la funzione kernel hello ()
    6. Leggi l'oggetto di memoria
    7. Free

HelloWorld in OpenCL
Codice Kernel

File: kernel hello.cl

```
1  __kernel void hello(__global char* string)
2  {
3  string[0] = 'H';
4  string[1] = 'e';
5  string[2] = 'l';
6  string[3] = 'l';
7  string[4] = 'o';
8  string[5] = ',';
9  string[6] = ' ';
10 string[7] = 'W';
11 string[8] = 'o';
12 string[9] = 'r';
13 string[10] = 'l';
14 string[11] = 'd';
15 string[12] = '!';
16 string[13] = '\0';
17 }
18
```

Il codice relativo al kernel va salvato in un file .cl.
La compilazione sarà eseguita dall'host a runtime.

HelloWorld in OpenCL
Codice Host - Inizializzazione

```c
#include <stdio.h>
#include <stdlib.h>
#include <CL/cl.h>

#define MEM_SIZE (128)
#define MAX_SOURCE_SIZE (0x100000)

int main()
{
/* Inizializzazione Elementi */
cl_platform_id platform_id = NULL;
cl_device_id device_id = NULL;
cl_context context = NULL;
cl_command_queue command_queue = NULL;
cl_mem memobj = NULL;
cl_program program = NULL;
cl_kernel kernel = NULL;
cl_uint ret_num_platforms; //Get delle piattaforme disponibili
cl_uint ret_num_devices; //Get dei device disponibili

cl_int ret;
```

Definizione degli elementi Host e Kernel

HelloWorld in OpenCL
Codice Host – Inizializzazione Host

```
/* Get Info Platform and Device */
ret = clGetPlatformIDs(1, &platform_id,
&ret_num_platforms); //L'host seleziona la
piattaforma da utilizzare;
ret = clGetDeviceIDs(platform_id,
CL_DEVICE_TYPE_DEFAULT, 1, &device_id,
&ret_num_devices); //L'host selezione il dveice da
utilizzare, CL__..._DEFAULT;

/* Crea il contex OpenCL */
context = clCreateContext(NULL, 1, &device_id,
NULL, NULL, &ret);

/* Crea la Coda dei Comandi */
command_queue = clCreateCommandQueue(context,
device_id, 0, &ret);

/* Crea il buffer di Memoria */
memobj = clCreateBuffer(context,
CL_MEM_READ_WRITE,MEM_SIZE * sizeof(char), NULL,
&ret);
```

Inizializzo la Piattaforma, il device, il contesto, la command queue ed il buffer di memoria

HelloWorld in OpenCL
Codice Host – Kernel

```
/* Carico il kernel dalla sorgente .cl */
program = clCreateProgramWithSource(context, 1,
(const char **)&source_str,
(const size_t *)&source_size, &ret);

/* Effettuo il build del kernel  */
ret = clBuildProgram(program, 1, &device_id, NULL,
NULL, NULL);

/* Creo il kernel OpenCL*/
kernel = clCreateKernel(program, "hello", &ret);

/* Setto i paramentri del Kernel */
ret = clSetKernelArg(kernel, 0, sizeof(cl_mem),
(void *)&memobj);

/* Eseguo il Kernel */
ret = clEnqueueTask(command_queue, kernel, 0,
NULL,NULL);

/* Copio i risultati dal buffer alla memoria host */
ret = clEnqueueReadBuffer(command_queue, memobj,
CL_TRUE, 0,
MEM_SIZE * sizeof(char),string, 0, NULL, NULL);
  printf("Stampo il risultato");

/* Stampa i risultati */
puts(string);
printf("Stampo il risultato");
```

1. Carico il kernel della sorgente
2. Effettuo il build del kernel
3. Creo il kernel
4. Setto i paramentri del kernel
5. Inserisco il kernel nella command queue
6. Copio i risultati dalla memoria all'host
7. Stampo i risultati

HelloWorld in OpenCL
Codice Host – Free

```
/* Libero la Memoria  */
ret = clFlush(command_queue);
ret = clFinish(command_queue);
ret = clReleaseKernel(kernel);
ret = clReleaseProgram(program);
ret = clReleaseMemObject(memobj);
ret = clReleaseCommandQueue(command_queue);
ret = clReleaseContext(context);

free(source_str);

return 0;
```

Il programma host termina liberando la memoria occupata.

Funzioni Codice
clGetPlatofrmIDs() – clGetDeviceIDs()

Dopo la fase di inizializzazione (Slide 37) è necessario configurare tutte le componenti HOST prima di caricare il kernel.

```
/* Get Info Platform and Device */
ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms); //L'host
seleziona la piattaforma da utilizzare;
ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id,
&ret_num_devices); //L'host selezione il dveice da utilizzare,
CL__..._DEFAULT;
```

La prima operazione è quella di ottenere un elenco delle piattaforme OpenCL disponibili.

clGetPlatofrmIDs()
Consente al programma host di rilevare i dispositivi OpenCL;
viene restituito come elenco al puntatore platform_id di tipo cl_platform_id.
Il primo argomento specifica quante piattaforme OpenCL trovare quale per la maggior parte è una. Il terzo argomento restituisce il numero di piattaforme OpenCL che è possibile utilizzare.

clGetDeviceIDs()
La funzione clGetDeviceIDs () seleziona il dispositivo da utilizzare.
Il primo argomento è la piattaforma che contiene il dispositivo desiderato. Il secondo argomento specifica il tipo di dispositivo. In questo caso, viene passato CL_DEVICE_TYPE_DEFAULT, che specifica qualsiasi impostazione predefinita.
Se il dispositivo desiderato è la GPU, allora deve essere CL_DEVICE_TYPE_GPU e, se si tratta di CPU, deve essere CL_DEVICE_TYPE_CPU.
Il terzo argomento specifica il numero di dispositivi da utilizzare.
Il 4 ° argomento restituisce l'handle al dispositivo selezionato. Il quinto argomento restituisce il numero di dispositivi che corrisponde al tipo di dispositivo specificato nel secondo argomento. Se il dispositivo specificato non esiste, allora ret_num_devices verrà impostato su 0.

Funzioni Codice
Context, Command Queue, Buffer

```
/* Crea il contex OpenCL */
context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);

/* Crea la Coda dei Comandi */
command_queue = clCreateCommandQueue(context, device_id, 0, &ret);

/* Crea il buffer di Memoria */
memobj = clCreateBuffer(context, CL_MEM_READ_WRITE,MEM_SIZE * sizeof(char),
NULL, &ret);
```

clCreateContext()
Crea il contesto.
Il secondo argomento specifica il numero di dispositivi da utilizzare. Il terzo argomento specifica l'elenco dei gestori di dispositivi. Il contesto viene utilizzato dal runtime OpenCL per la gestione degli oggetti.

clCreateCommandQueue()
Crea la coda dei comandi.
Il primo argomento specifica il contesto della coda comandi. Il secondo argomento specifica il dispositivo che eseguirà il comando nella coda. La funzione restituisce un handle alla coda comandi, che verrà utilizzata per la copia della memoria e l'esecuzione del kernel.

clCreateBuffer()
La funzione clCreateBuffer() alloca spazio sulla memoria del dispositivo.
È possibile accedere alla memoria allocata dal lato host utilizzando il puntatore memobj restituito. Il primo argomento specifica il contesto in cui l'oggetto di memoria entrerà a far parte. Il secondo argomento specifica una bandiera che descrive come verrà utilizzata. Il flag CL_MEM_READ_WRITE consente al kernel di leggere e scrivere nella memoria del dispositivo allocata. Il terzo argomento specifica il numero di byte da allocare. In questo esempio, questo spazio di archiviazione allocato ottiene la stringa di caratteri "Hello, World!".()

Funzioni Codice
Build e Creazione del Kernel

```
/* Effettuo il build del kernel  */
ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

/* Creo il kernel OpenCL*/
kernel = clCreateKernel(program, "hello", &ret);

/* Setto i paramentri del Kernel */
ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobj);

/* Eseguo il Kernel */
ret = clEnqueueTask(command_queue, kernel, 0, NULL,NULL);
```

clCreateProgramWithSource()
Creo l'oggetto Programma
Ho prima caricato in un buffer di char il file .cl
L'oggetto programma viene creato utilizzando la funzione clCreateProgramWithSource (). Il 3 ° argomento specifica il codice sorgente letto e il 4 ° argomento specifica la dimensione del codice sorgente in byte. Se l'oggetto del programma deve essere creato da un file binario, al suo posto viene utilizzato clCreateProgramWithBinary ().

clBuildProgram()
ClBuildProgram crea l'oggetto programma per creare un file binario. Il primo argomento è l'oggetto del programma da compilare. Il terzo argomento è il dispositivo di destinazione per il quale viene creato il binario. Il secondo argomento è il numero di dispositivi target. Il quarto argomento specifica la stringa dell'opzione del compilatore.
Si noti che questo passaggio non è necessario se il programma viene creato da binario utilizzando clCreateProgramWithBinary ().

Funzioni Codice
Settaggio ed Esecuzione del Kernel

```
/* Setto i paramentri del Kernel */
ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobj);

/* Eseguo il Kernel */
ret = clEnqueueTask(command_queue, kernel, 0, NULL,NULL);
```

clSetKernelArg()
Imposta argomenti del kernel
Una volta creato l'oggetto kernel, è necessario impostare gli argomenti per il kernel. In questo esempio, hello.cl si aspetta che un puntatore a un array di stringhe venga creato sul lato del dispositivo. Questo puntatore deve essere specificato sul lato host. In questo esempio, viene passato il puntatore all'oggetto di memoria allocato. In questo modo, la gestione della memoria può essere eseguita sul lato host.
La funzione clSetKernelArg () nella riga 67 imposta gli argomenti da passare nel kernel. Il primo argomento è l'oggetto kernel. Il secondo argomento seleziona quale argomento del kernel viene passato, che è o in questo esempio, il che significa che è impostato il o ° argomento per il kernel. Il quarto argomento è il puntatore all'argomento in cui deve essere passato, con il terzo argomento che specifica la dimensione di questo argomento in byte. In questo modo, è necessario chiamare clSetKernelArg () per ogni argomento del kernel.

clEnqueueTask()
Esegui kernel (attività Enqueue)
Questo lancia il kernel nella coda di comando, da eseguire sull'unità di calcolo sul dispositivo. Nota che questa funzione è asincrona, il che significa che getta semplicemente il kernel nella coda per essere eseguito sul dispositivo. Il codice che segue la funzione clEnqueueTask () dovrebbe tener conto di ciò.
Per attendere il completamento dell'esecuzione del kernel, il quinto argomento della funzione precedente deve essere impostato come oggetto evento.

Funzioni Codice
Read Buffer e Stampa dei Risulati

```
/* Copio i risultati dal buffer alla memoria host */
ret = clEnqueueReadBuffer(command_queue, memobj, CL_TRUE, 0,
MEM_SIZE * sizeof(char),string, 0, NULL, NULL);

/* Stampa i risultati */
printf("Stampo il risultato\n");
printf(string);
```

clEnqueueReadBuffer ()
Copia dati dalla memoria del lato dispositivo alla memoria del lato host. Per copiare i dati dalla memoria sul lato host nella memoria sul lato dispositivo, viene invece utilizzata la funzione clEnqueueWriteBuffer ().
L'istruzione di copia dei dati viene inserita nella coda comandi prima di essere elaborata. Il secondo argomento è il puntatore alla memoria sul dispositivo da copiare sul lato host, mentre il quinto argomento specifica la dimensione dei dati in byte. Il sesto argomento è il puntatore alla memoria sul lato host in cui vengono copiati i dati.Il terzo argomento specifica se il comando è sincrono o asincrono. "CL_TRUE" che viene passato rende la funzione sincrona, che impedisce all'host di eseguire il comando successivo fino al termine della copia dei dati. Se invece viene passato "CL_FALSE", la copia diventa asincrona, il che mette in coda l'attività ed esegue immediatamente l'istruzione successiva sul lato host.che mette in coda l'attività ed esegue immediatamente l'istruzione successiva sul lato host.che mette in coda l'attività ed esegue immediatamente l'istruzione successiva sul lato host.

Funzioni Codice
Read Buffer e Stampa dei Risulati

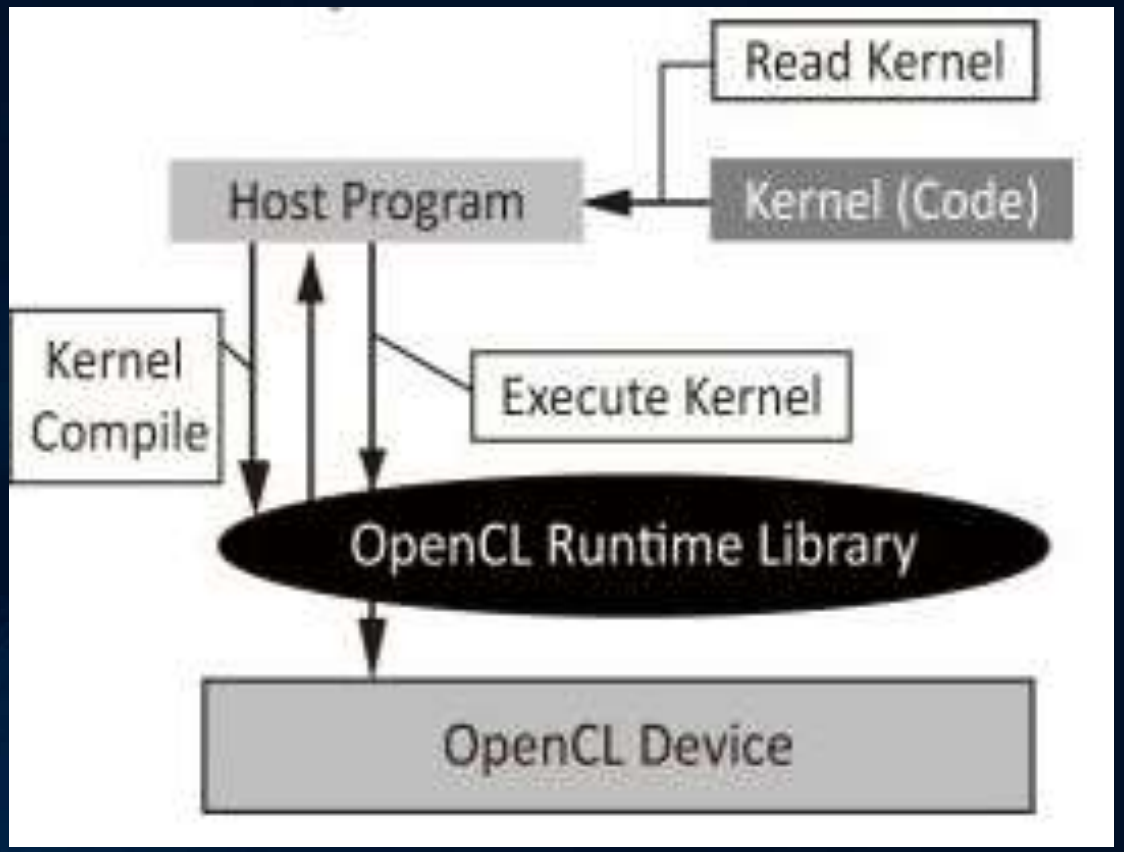
```
/* Copio i risultati dal buffer alla memoria host */
ret = clEnqueueReadBuffer(command_queue, memobj, CL_TRUE, 0,
MEM_SIZE * sizeof(char),string, 0, NULL, NULL);

/* Stampa i risultati */
printf("Stampo il risultato\n");
printf(string);
```

clEnqueueReadBuffer ()
Copia dati dalla memoria del lato dispositivo alla memoria del lato host. Per copiare i dati dalla memoria sul lato host nella memoria sul lato dispositivo, viene invece utilizzata la funzione clEnqueueWriteBuffer ().
L'istruzione di copia dei dati viene inserita nella coda comandi prima di essere elaborata. Il secondo argomento è il puntatore alla memoria sul dispositivo da copiare sul lato host, mentre il quinto argomento specifica la dimensione dei dati in byte. Il sesto argomento è il puntatore alla memoria sul lato host in cui vengono copiati i dati.Il terzo argomento specifica se il comando è sincrono o asincrono. "CL_TRUE" che viene passato rende la funzione sincrona, che impedisce all'host di eseguire il comando successivo fino al termine della copia dei dati. Se invece viene passato "CL_FALSE", la copia diventa asincrona, il che mette in coda l'attività ed esegue immediatamente l'istruzione successiva sul lato host.che mette in coda l'attività ed esegue immediatamente l'istruzione successiva sul lato host.che mette in coda l'attività ed esegue immediatamente l'istruzione successiva sul lato host.

Review Ciclo di vita OpenCL

Nelle applicazioni reali, il ciclo di vita principale è di solito una ripetizione dell'impostazione degli argomenti del kernel o della copia da host a dispositivo -> esecuzione del kernel -> ciclo di copia da dispositivo a host;



Domande?

Riferimenti

1. http://developer.amd.com/wordpress/media/2013/01/Introduction_to_OpenCL_Programming-Training_Guide-201005.pdf
2. Programming Massively Parallel Processors, A Hands-on Approach Second Edition - David B. Kirk and Wen-mei W. Hwu
3. https://www.slideshare.net/TomaszBednarz1/introduction-to-opencl-2010
4. GP-GPU e l'ambiente CUDA, MNI, Prof.ssa A.Cardone
5. https://streamhpc.com/blog/2011-06-24/install-opencl-on-debianubuntu-orderly/
6. https://www.khronos.org/opencl/
7. https://www.intel.com/content/www/us/en/programmable/products/design-software/embedded-software-developers/opencl/support.html
8. https://us.fixstars.com/products/opencl/book/OpenCLProgrammingBook/basic-program-flow/

Codice sorgente -> https://github.com/Francesco182g/Seminario-OpenCL

Grazie dell’attenzione