

PROGRAMMAZIONE FUNZIONALE E PARALLELA

By Edoardo

PROGRAMMAZIONE FUNZIONALE E PARALLELA

SCALA

val x:Int = 10

N.B. x non può essere riassegnato!

Penso omettere il tipo, secca lo identificherà da solo: val x=10

name_val.TAB TAB mostra le funzioni possibili su quella variabile

-funzioni

def f(x:tipo): tipo = x+x (onde in questo caso posso omettere il tipo Solo per l'output)

f: nome funzione

tipo1 = tipo input

tipo2 = tipo output

def f(x:Int) = {
 Val y = x+x
 }
Se il tipo di ritorno è Unit posso omettere l'uguale
Y → l'ultima funzione dà il valore di ritorno

def p(x:Int):Unit = println("mio x: "+x)
↓ un po' diverso da void: ritorna almeno "()"

: load nome_script → compilo ed esegue

: paste → modalità interattiva

- IF

if (condizione) ris1 else ris2

N.B. ris1 e ris2 possono essere anche funzioni

Se le due risposte hanno tipo diverso il risultato sarà di classe Any avendo la prima superposta in comune che però non può essere usata facilmente

In funzioni di tipo ricorsivo è necessario specificare il tipo di ritorno.

- Tuple

Val x = (Valore1, Valore2)

def minMax(x:Int, y:Int):(Int, Int) = if (x>y) (y, x) else (x, y)
minMax(34, 120) → (34, 120)

Val (a,b) = minMax(12, 5)

↳ a = 5

b = 12

Solo può instantiare oggetti (classi) di JAVA

Object Nome_Classe extends App {
}

oppure

Object Nome_Classe {
def main (arg: Array[String]) = {
}
}

LE FUNZIONI SONO VALORI

def doppio (x:Int) = 2*x
def stampaDoppio (x:Int) = println (2*x) *Troppo specifico*
def stampaTrasf (x:Int, f: $D \Rightarrow C$) = println (f(x))
Domino \Rightarrow codominio

Tipo Valore Funzione = $f: \text{dominio} \Rightarrow \text{codominio}$

es. def somma (x:Int, y:Int) = x+y
val f: (Int, Int) \Rightarrow Int = somma
 $\| f(10, 20)$ equivalente a somma (10, 20)

Funzioni Anonime (Lambda functions)

Espressioni che denotano funzioni

es. $(x: \text{Int}) \Rightarrow 2^x$

\Rightarrow StampaTrasf (10, $x: \text{Int} \Rightarrow 2^x$) \rightarrow $x \Rightarrow 2^x$ può diventare $2^x -$
Sì può omettere

def Specifica un metodo

Val $x: D \Rightarrow C$ = metodo $\rightarrow x$ è una funzione
dominio e codominio uguali a quelli del metodo
oppure

Val $x = f -$ con il trattino basso \rightarrow il "Cost" automatico

del $f[T] (x:T, y:T) = (x, y)$ \rightarrow Tipo generico

Programmazione Difensiva

es. $\text{def div } (x:\text{Int}, y:\text{Int}):\text{Int} = \{$
 $\text{require } (y \neq 0)$
 x/y
}

Se la condizione non è accettata si lancia un'eccezione

Passaggio Parametri

In scala se in una funzione uso delle funzioni come parametri vengono eseguite prima le funzioni come usate come parametri

es

- $p(x:\text{Int}) = \{\text{println}(x); x\}$
- $\text{myIf}(x:\text{int}, y:\text{int}, c:\text{boolean}) = \{\text{if}(c) a \text{ else } b\}$

$\rightarrow \text{myIf}(p(10), p(20), \text{true})$

↳ prima esegue $p(10)$, poi $p(20)$ e poi esegue myIf

$\text{def f}(x: \Rightarrow \text{Int}) = \{x+x\}$ $x: \Rightarrow \text{Int}$ passaggio per nome

così facendo la funzione verrà eseguita nel corpo di f e non quando dichiaro i parametri

es. $f(p(10))$

↳ stampa 10 2 volte (come è giusto che sia dato $x+x$ e non 3) e poi ritorna 20

Funzione di ordine superiore è una funzione che prende come parametri o restituisce funzioni.

Esempio: `def componi (f1: Int → Int, f2: Int → Int): Int → Int = {
 def aux (x: Int): Int = f1(f2(x))
 aux
}`

f₁(f₂(x)) → chiusura: una parola dell'ambiente (f₁ e f₂)

N.B. Posso omettere il tipo di ritorno di componi mettendo "—" dopo aux

Alternativa:

`def componi (f1: Int → Int, f2: Int → Int): Int → Int = {
 x ⇒ f1(f2(x)) ← f1(f2(_))
}`

LAZY VAL

`lazy val p = [expres.]`

Dichiarando lazy la funzione di p viene valutata (eseguita) solo quando p viene usata esplicitamente. Facendo così l'espressione potrà avere side effects che verranno applicati solo quando richiesti (una sola volta)

Sequenze Immutabili

- 1) Liste: colligate
- 2) Vector: indirizzata

LISTE

val $l = \text{List}(1, 2, 3)$

val $v = \text{Nil} \rightarrow$ lista vuota $\neq \text{NULL}$

Accedere a Nil non crea NullpointerException, ma non trova nulla.

È possibile fare liste di elementi di tipo differente che verranno identificati come "ANY" (oggetti)

l. head \rightarrow ritorna il primo elemento

l. tail \rightarrow ritorna il resto della lista (tutto tranne il primo)

l. length \rightarrow # elementi

l == lz \rightarrow uguaglianza profonda

l :: lz \rightarrow concatena le due liste

elem :: l \rightarrow aggiunge elem in testa

l :+ elem \rightarrow aggiunge elem in coda

l. reverse \rightarrow ritorna l'involuta

l. drop(n) \rightarrow elimina i primi n elementi e ritorna la nuova lista

l. splitAt(n) \rightarrow ritorna una coppia di liste spezzando l all' n-esimo elemento

l. isEmpty \rightarrow true se l è vuota, false altrimenti

l. filter(function booleana) \rightarrow restituisce una lista con gli elementi di l che soddisfano la funzione

l. map(function) \rightarrow restituisce una lista che ha come elementi il risultato della funzione applicata a ogni elemento di l.

es. l. map(_ * 2) \rightarrow l' ha come elementi gli elementi di l moltiplicati per 2

l. reduce(function): la funzione deve prendere come input due elementi

es. l. reduce(_ + _) \rightarrow a due a due somma gli elementi di l e ritorna il totale

l. forall(func.boolean) \rightarrow True se è True per ogni elemento.

l. exists(func.boolean) \rightarrow true se contiene almeno un elemento che soddisfa la funzione

l. foreach(func) \rightarrow applica la funzione a tutti gli elementi: usata molto per side effect. Al contrario di map non ritorna una nuova lista.

```

def filtraPosPari (l: List[int]): List[int] = {
    l.zipWithIndex.filter(p => p._2 % 2 == 0).map(p => p._1)
        ↓
        (x, i)
            _ - 1

```

zip aggiunge a ogni elemento il suo indice formando una coppia
 es. List(1, 2, 3).zipWithIndex => List((1, 0), (2, 1), (3, 2))

1. distinct toglie i duplicati dalla lista

1. sorted ordina gli elementi

1. groupby (funzione) restituisce una map di liste (dizionario (chiave; lista)). La funzione "trasforma" l'elemento nel criterio di raggruppamento es. groupby (prima lettera: s => s.charAt(0))

Lo map può essere accedita tramite chiavi

N.B. Crea sempre una nuova lista, mai side effect

TIPI GENERALI

```
def f[T](x:T, y:T):T = { ... }
```

```
def g[T <: Ordered[T]](...)
```

<: → lo classe T deve essere sottoclasse di Ordered (ordinabile) e T può essere convertito in modo implicito a Orderable

PATTERN MATCHING

PATTERN (letterale, costante)

```

x match {
    case "lunedì" => 1
    case "martedì" => 2
    ...
}
```

case _ -> -1 (catch all) N.B. Se non metto il "catch all" se x non "matcha" con nessun case genera un'eccezione a runtime

MATCHING SU VARIABILI

```

x match {
    case z: Int => "Int"
    case z: String => "String"
    case _ => "altro"
}
```

matching sul tipo di x
 (come get(class di JAVA))

x può essere un'espressione qualsiasi, il pattern no!

È possibile aggiungere una clausola if nel case es. case n if (n < 2) => ..

MATCHING STRUTTURALE

```
X match {  
  case Nil => "Vuota"  
  case h :: t => h + " è la testa" + t + " è il resto"  
    x.head      x.tail  
}  
}
```

```
(2b) match {  
  case (Nil, Nil) => ...  
  case (_, Nil) => ...  
  case (Nil, _) => ...  
  case (h1 :: t1, h2 :: t2) =>  
}
```

CLASSI

```
class Punto (x: Double, y: Double) { NB Costruttore compreso nella classe
```

```
def getX = x  
def getY = y  
def dist (p: Punto) = Math.sqrt ((x - p.getX)2 + (y - p.getY)2)  
}
```

```
val p = new Punto(10, 20)
```

NB. def this () specifica un nuovo costruttore

Mettendo Val nei parametri, scalo crea dei getter automatici con il nome del parametro

es. class Punto (Val x: Double, Val y: Double) {}

Val p = new Punto(10, 20)

p.x => 10

Lo classe è immutabile, ma è possibile mettere var invece di val: in questo caso verranno creati anche i settor.

Superclasse

```
class Punto3D(Val x: Double, Val y: Double, Val z: Double) extends Punto(x, y) {...}
```

L'override dei metodi bisogna esplorarlo: `override def toString = "(+x; +y; +z)"`

Apply

```
def apply(i:Int) = if (i==1) x else y
```

```
val p = new Punto(10,20)
```

```
println(p.apply(1))
```

printIn(p(1))

uguale: stampa x

Case class

```
case class Point(Val x: Double, Val y: Double)
```

- equals, hashCode e toString già fatti
- non serve new per istanziare l'oggetto → val p = Point(10,20)
- è possibile fare pattern matching sulle classi case

Companion Object

```
• class Punto(...)  
• object Punto { → contiene tutti i campi statici della classe  
    def apply(x: Double) = new Punto(x) già fatta nel caso in cui Punto fosse case class  
}
```

→ Punto(10) invocazione apply = new Punto(10)

CONVERSIONI IMPLICITE

Rendere possibile la conversione automatica di un oggetto in un altro.

```
val a: A = ...
```

```
val b: B = a
```

→ metodo implicito di conversione da tipo A a tipo B → $f: A \rightarrow B$

```
Implicit def nomef(a:A): B = new B(pot: di a)
```

Questo metodo deve essere nello stesso scope del punto in cui applico la conversione

es. a.metodo() con metodo non appartenente ad a ma ad B

In questo caso Scala non trovando il metodo in a, converte a in B e lo applica

Il nome del metodo deve essere univoco intutte le classi

Se modulerizzo i file, questo metodo andrà nella parte statica (object) della classe importando l'oggetto.

SINTACTIC SUGAR

$l_1.map(_ \times z) \Rightarrow \text{for } (i \leftarrow l_1) \text{ yield } z * i$
yield "mette da parte" ogni elemento i e lo moltiplica per z (in questo caso)

Ese. $l_1 \times l_2$

$l_1.map(x \Rightarrow l_2.map(y \Rightarrow (x,y))).flatten$ lista di coppia
 $\hookrightarrow \text{for } (x \leftarrow l_1; y \leftarrow l_2) \text{ yield } (x,y)$

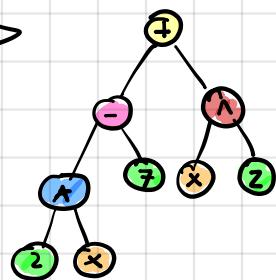
generatori

Potrei aggiungere filtri (if) e altre variabili:
(selezio lista numeri pari in $[0, n]$)

$(0 \text{ to } n).filter(_ \% 2 == 0)$
 $\hookrightarrow \text{for } \{ x \leftarrow 0 \text{ to } n \\ \text{if } (x \% 2 == 0) \} \text{ yield } x$

val $f = z^x - 7 + (x^z)$

\Rightarrow



sealed abstract class Exp {

```
def apply(x: Double): Double = {
    this match {
        case X() => x
        case Const(i) => i
        case Add(a, b) => a.apply(x) + b.apply(x) // a(x) + b(x)
    }
    def +(e: Exp) = Add(this, e)
}
```

}

case class X() extends Exp

case class Const(i: Int) extends Exp

case class Add(a: Exp, b: Exp) extends Exp

object Exp {

```
implicit def int2Exp(i: Int) = Const(i)
```

}

import Exp._

object Prova extends App {

```
val f = Add(X(), Const(1)) // x+1
```

```
val g = X() + 1 // Add(X(), 1)
```

```
val y = g(10.7) // 11.7
```

}

Classe Mutabile

es. MyArray(a: Array) { .. }

array(i) \rightarrow array.apply(i) = { .. }

array(i, k) = z \rightarrow array.update(i, k) = z

• def update(i:Int, k:Int) = a(i)=v

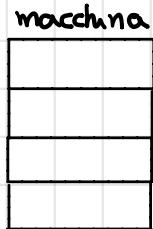
PROGRAMMAZIONE PARALLELA 23/11

Al momento la frequenza di clock è "limata" sui 3 GHz e si tende ad aumentare il numero di core. Per questo motivo si deve programmare parallelamente su più core.

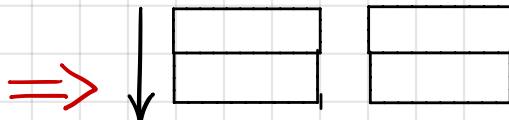
Hardware Parallello

(anche su singolo core)

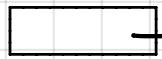
Messo istruzioni sequenziali



parallelista



core



ILP = Instruction Level Parallelism

└ pipelining (automatico)

 └ istruzioni vettoriali

 └ MMX

 └ AVX

Istruzioni Vettoriali:

con una singola istruzione vettoriale effettuate più istruzioni parallele su vettori

Data Center: macchine del tipo data center sono composte da più processori. Ognuno di essi ha dei registri ("memoria" più rapida locale), ha accesso a memoria cache L1, L2 e L3 con "ritardi" progressivi. Avendo più processori sarà possibile lavorare parallelamente e diminuire i tempi di attesa.

WORK = tempo totale impiegato dai threads

SPAN = Massimo dei tempi dei threads

Non sempre lavorare in parallelo aumenta le prestazioni: merge Sort per esempio utilizza un eccessivo numero di threads e questo porta alla saturazione dei core che sono un numero limitato (es. 4)

Modifica: inizio l'algoritmo in modo parallelo fino ad avere un numero di thread adeguato al numero di core e poi passare al modo sequenziale su singoli core

2 Algoritmi → **Divide et Impera**

Questa modalità di programmazione è chiamata **consenning della ricorsione**

$x : T$

$f(x) \rightarrow T$ se f è corretta.

Ma in caso di errore?

O usiamo un valore del dominio di T , ma in caso (per esempio) $T = \text{Int}$ così facendo sacrifichiamo numero oppure



Lo funzione restituisce sempre qualcosa, ma nel caso di errore l'oggetto di ritorno è vuoto ($\neq \text{null}$)

`val x: Option[T] = ...`

`println(x.getOrElse("errore"))` fallback

Solvore valore di ritorno di un thread figlio nel padre:

`def par[S,T](body1: => S)(body2: => T): (S,T) = {`

`var res: Option[T] = None`

`val r = new Runnable {`

`def run() {`

`res2 = Some(body2)`

`}`

`}`

`val t = new Thread(r)`

`t.start()`

`val res1 = body1`

`t.join()`

`res2 match {`

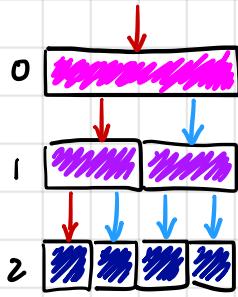
`case Some(x) => (res1, x)`

`case None => //System.exit(1) // non succede mai`

`//(res1, body2) // in caso di errore nel thread esegue in sequenziale`

`}`

CYBERCHALLENGE



Lo striscia rossa indica il thread principale mentre quella blu indica gli spawn dei nuovi thread figli.

Potendoci dietro un indice è possibile "dividere" lo spazio sotto in modo sequenziale per i livelli bassi (es 2) e parallelo per gli altri in cui abbiamo un'elasta quantità di dati rispetto agli strati più bassi.

```
def update (i:Int, j:Int, v:Int) = m(i)(j)=v
def apply (i:Int, j:Int) = m(i)(j)
```

ISTRUZIONI VETTORIALI

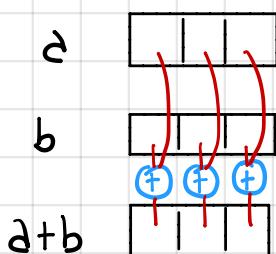
Operazioni:

$\begin{cases} \text{S1MD} \\ \text{M1MD} \\ \text{M1SD} \\ \text{S1SD} \end{cases}$ Single Instruction Multiple Data

Taxonomia di Flynn

es

SIMD:



Definizioni

Operazioni Vettoriali:

- x86 = istruzioni vettoriali dedicate
- C = **intrinsics** del compilatore

↳ come se fosse una chiamata a funzione, ma è mappata su istruzione macchina

Tipi di dato:

- x86 = registri
- C = tipi (struttura) dedicati

In C #include <immintrin.h> e gcc -mX dove X sarà un'estensione

TIPI DI DATO

Anche chiamati packed data types

intero	prc. singola	prc. doppia	Register nominativi	Estensione	bit	
--m64			8 MMX0- MMX1-	MMX	64	char[8] short[4] int[2]
--m128i	--m128	--m128d	16 MM0- MM16-	SSE	128	char[16] short[8] int[4]
--m256i	--m256	--m256d	16 MM0- MM16-	AVX	256	char[32] short[16] int[8]
--m512i	32 MM0- MM31-	AVX-512	512	

es. Void Somma (short A[8], short B[8], short C[8]) {

$$C[0] = A[0] + B[0];$$

$$C[1] = A[1] + B[1];$$

...

$$\text{short}[8] = 2 \cdot 8 = 16 \text{ byte} = 128 \text{ bit}$$

Void Somma (...) {

--m128i a,b,c  Struct Ospca

a = mm_load_si128 ((const --m128i*) A);  

b = ... B;

c = mm_add_epi16(a,b);   

mm_store_si128 ((--m128i*) C, c); 

}

main:

short A[8] = {1, 2, 3, ...}

short B[8] = ...

short C[8]

Somma(A,B,C)

con riferimento alla memoria: mm_load_si128 → necessita che l'indirizzo sorgente sia allineato a multipli di 128 bit (16 byte)

GPU

Utilizzate per il calcolo "massivo" ovvero che richiede un alto livello di parallelismo

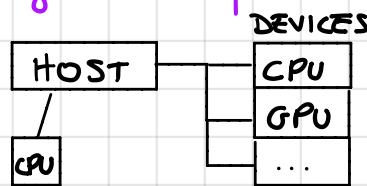
Heterogeneous Computing: macchine che utilizzano CPU e GPU per i calcoli

Framework Programmazione

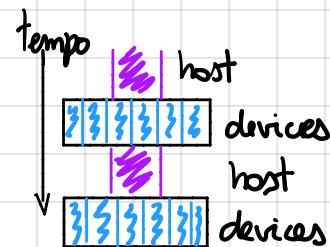
1) CUDA (Nvidia)

2) OpenCL (Kronos Group): ha lo caratteristica di poter creare le "task" da eseguire e lanciare sulla macchina e lo scheduling sarà automatico potendo così usare tutta la potenza di calcolo possibile (sia CPU che GPU)

Programma per architettura eterogenea (OpenCL)



Anatomia Esecuzione:



Il programma parte sull'host (calcolo sequenziale)

L'host aspetta che i devices terminino i loro threads

Host programmato in C e i devices in OpenCL e dialetto di C99 con qualche piccola restrizione

Modello Calcolo OpenCL

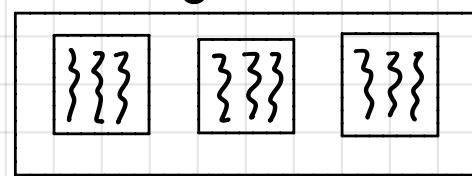
Work item (thread)

Core
item
(compute element)

Work group

Streaming Multiproc.
(compute unit)

NDRange

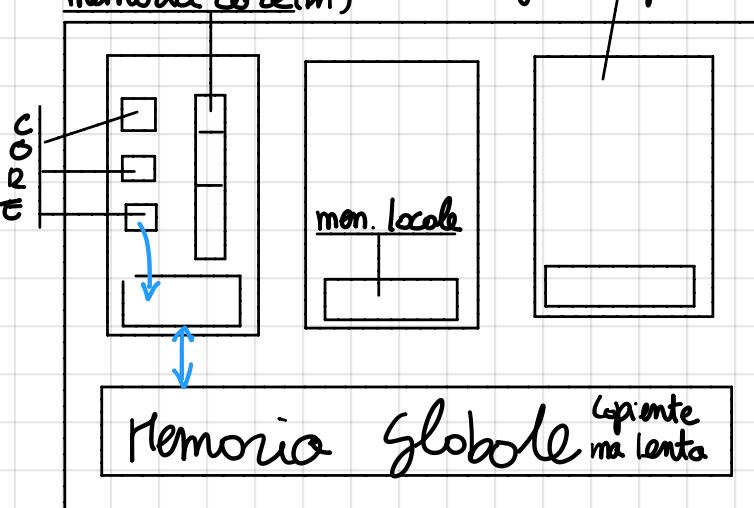


GPU (Compute Device)
Istante multiple di work group

Architettura GPU

memoria core(MT)

Streaming Multiprocessor



Work group ha dimensione massima e work item si mappano in parallelo sui singoli core

I vari work group vengono schedulati da OpenCL sugli Stream Multiprocessors disponibili

Accesso a Privata Locale Globale memoria

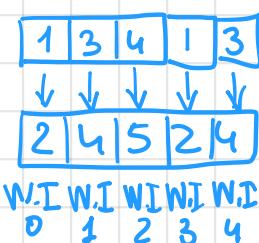
WORK ITEM	✓ W.I.	✓ W.G.	✓	
WORK GROUP		✓	✓	
ND RANGE			✓	

Privata	Locale	Globale	
Variabili locali, parametri fermi del thread	Buffer ad alta velocità di accesso accessibili da un W.G.	Tutti i dati da elaborare	

Come si programma un W.I.? Scrivendo una funzione OpenCL chiamata **kernel** ovvero la funzione di interesse (come run per i thread). Kernel viene istanziata su più W.I. in automatico

es. pippo.cl

```
-- kernel void miokernel (...) {
    int i = get_global_id(0);
    ↳ id thread
    v[i]++;
}
```



Ogni istanza di questa funzione incrementa un elemento di v

Come posso definire l'intervallo degli ID dei thread (W.I.)?

→ Host lancia la computazione con chiamata **enqueueNDRangeKernel** de fondo task in parallelo istanziando lo stesso Kernel su un intervallo di indici da 0 a n-1

NDRange (monodimensionale)



Se WG size = 4

E' possibile sincronizzare W.I solo se sono nello stesso WG.

N.B. global size (# W.I) è sempre multiplo della local size (dim. W.G.)

16/12

Pipeline Costruzione Programma OpenCL

- Prog. OpenCL =
 - Codice host (c)
 - Codice device (openCL C)

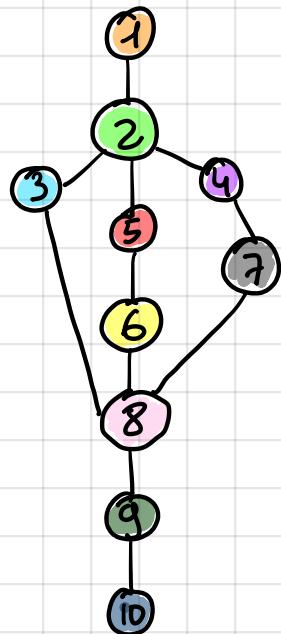
- 1) **Query Platform Calcolo** → una o più platform = host + lista device
↳ uno o più device per platform

cl-platform-id = tipo riferimento ad una platform

cl-device-id = tipo riferimento ad un device di una platform

- 2) **Ottene Contesto per device** → uno per device

cl-context



- 3) **Create coda comandi** : host $\xrightarrow{\text{QUEUE}}$ DEVICE
cl-command-queue

Le tasks possono essere :

- copia mem. host \leftrightarrow mem device
- do eseguire sul device

In più si dividono in blocchi e non

- 4) **Costruzione Progmma OpenCL** Costruire un programma (cl_program) + costruire kernel (cl-kernel)

Un programma è una collezione di uno o più kernel

In questa fase viene caricato e compilato il codice OpenCL C

- 5) Allocazione Memoria
sul device
- 6) Copia host → device
- 7) Passaggio Parametri
kernel
- 8) Lancio Kernel sul
device
- 9) Copia Risultato
Device → host
- 10) Rilascio Risorse

Per programmare in OpenCL le matrici sono espresse nella forma Row-major
in cui tutte le righe sono consecutive e non una sopra l'altra.
Il puntatore punta al primo elemento della prima riga.

$$m[i, j] = i * n + j \quad \text{con} \quad n = \text{lunghezza riga}$$

```
#define IDX(i, j) ((i) * (n) + (j))
```