GPU Computing

Lab1

Il mio primo programma CUDA

Hello world (a più voci!)

Hello World in C (ma compilare con nvcc)

1. Creare un file sorgente con estensione **.cu**
2. Compilare il sorgente usando il compilatore CUDA **nvcc**
3. Eseguire il programma (che contiene il codice kernel ed eseguibile)

Main

```
#include <stdio.h>


int main(void) {
   printf("Hello World from CPU!\n");
}
```

Salvare il codice nel file **hello.cu** e compilare con **nvcc** (la procedura è simile a **gcc** o altri compilatori – fornire parametri di compilazione ed elenco sorgenti) ed eseguire:

terminal

```
$ nvcc hello.cu -o hello
$ hello
Hello World from CPU!
```

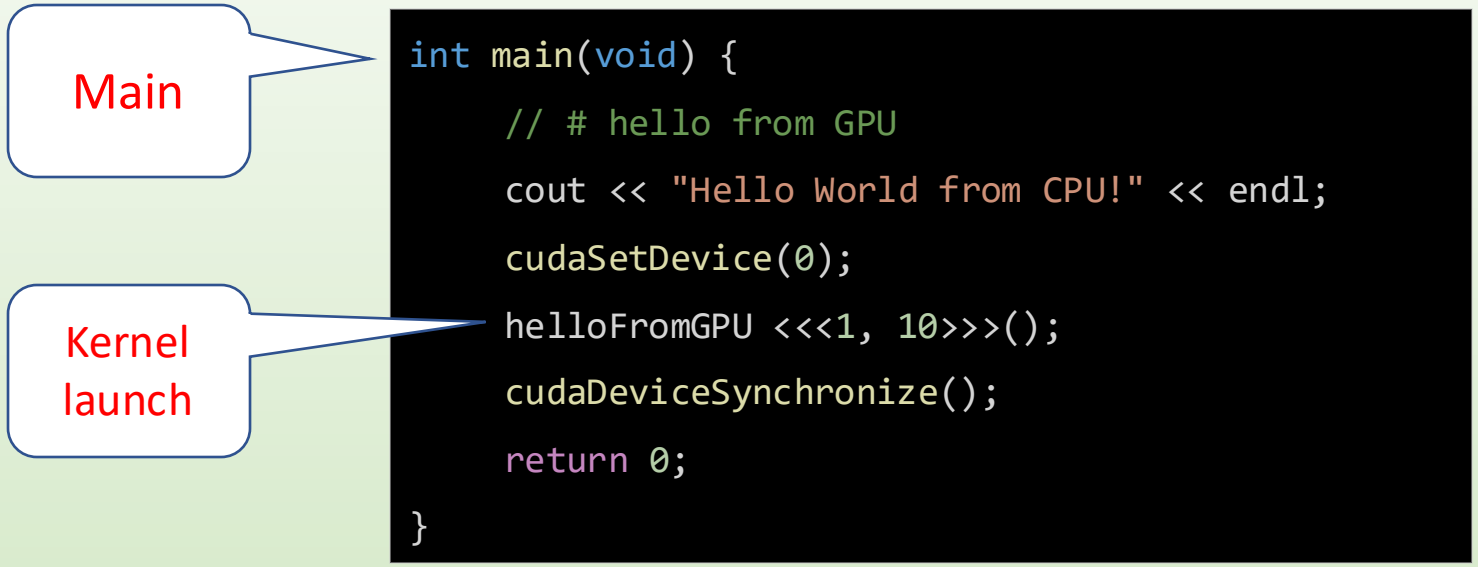
Hello world in CUDA

Hello world in CUDA

```
__global__ void helloFromGPU (void) {
    int tID = threadIdx.x;
    printf("Hello World from GPU (I'am thread %d)!\n", tID);
}
```

- ✓ Il **qualificatore** `__global__` dice al compilatore che la function (**kernel**) è chiamata dalla CPU ma eseguita in GPU
- ✓ Variabile **builtin** che denota **thread ID**: `threadIdx.x`
- ✓ Il **tipo** restituito è sempre `void`
- ✓ Accetta arbitraria sequenza di **argomenti**: qui `(void)`

Hello world in CUDA



- ✓ L'invocazione del **kernel**: `helloFromGPU <<<1, 10>>>();`
- ✓ Un **kernel** viene eseguito da un **array di thread** e tutti i thread eseguono lo **stesso codice**
- ✓ La **tripla parentesi angolare** `<<<1, 10>>>();` denota una **chiamata** dal **codice host** al **codice device** che ne attiva l'esecuzione… max `<<<1, 1024>>>();`
- ✓ I parametri all'interno della **triplice parentesi angolare** sono i **parametri** di **configurazione** che specificano quanti thread verranno eseguiti dal kernel
- ✓ In questo esempio, verranno eseguiti **10 GPU thread!**

La compilazione con nvcc

✓ **Specificare l'architettura GPU** (per generare codice ottimizzato per la tua GPU):

```
$ nvcc -arch=sm_75 hello.cu -o hello
$ ./hello
```

(Dove **sm_75** è per GPU **Turing T4**. Per Ampere usa **sm_80**, ecc.)

✓ **Verifica** dell'installazione di **nvcc**:

```
$ nvcc --version
```

```
nvcc: NVIDIA (R) Cuda compiler driver Copyright
(c) 2005-2024 NVIDIA Corporation Built on
Thu_Jun__6_02:18:23_PDT_2024 Cuda compilation
tools, release 12.5, V12.5.82 Build
cuda_12.5.r12.5/compiler.34385749_0
```

Running

✓ Il sorgente finale:

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

using namespace std;

__global__ void helloFromGPU (void) {
   int tID = threadIdx.x;
   printf("Hello World from GPU (I'am thread %d)!\n", tID);
}

int main(void) {
     // # hello from GPU
     cout << "Hello World from CPU!" << endl;
     cudaSetDevice(0);
     helloFromGPU <<<1, 10>>>();
     cudaDeviceSynchronize();
     return 0;
}
```

✓ Compila con **nvcc** ed esegui

```
$ nvcc .arch=sm_75 hello.cu -o hello
$ ./hello
Hello World from CPU!
Hello World from GPU (I'am thread 0)!
Hello World from GPU (I'am thread 1)!
Hello World from GPU (I'am thread 2)!
Hello World from GPU (I'am thread 3)!
Hello World from GPU (I'am thread 4)!
Hello World from GPU (I'am thread 5)!
Hello World from GPU (I'am thread 6)!
Hello World from GPU (I'am thread 7)!
Hello World from GPU (I'am thread 8)!
Hello World from GPU (I'am thread 9)!
```

✓ La funzione **`cudaDeviceReset()`** distrugge e ripulisce tutte le risorse associate al device corrente nel processo corrente (non indispensabile!... vedere successivamente)

Somma cumulata in CUDA

Uso di thread per operazioni su array

Somma cumulata di una sequenza

Dati una sequenza di numeri $a_1, a_2, \dots, a_n$ e un indice k, la somma cumulata è:

$$s_k = \sum_{i=1}^{k} a_i \qquad k = 1,2, \dots, n$$

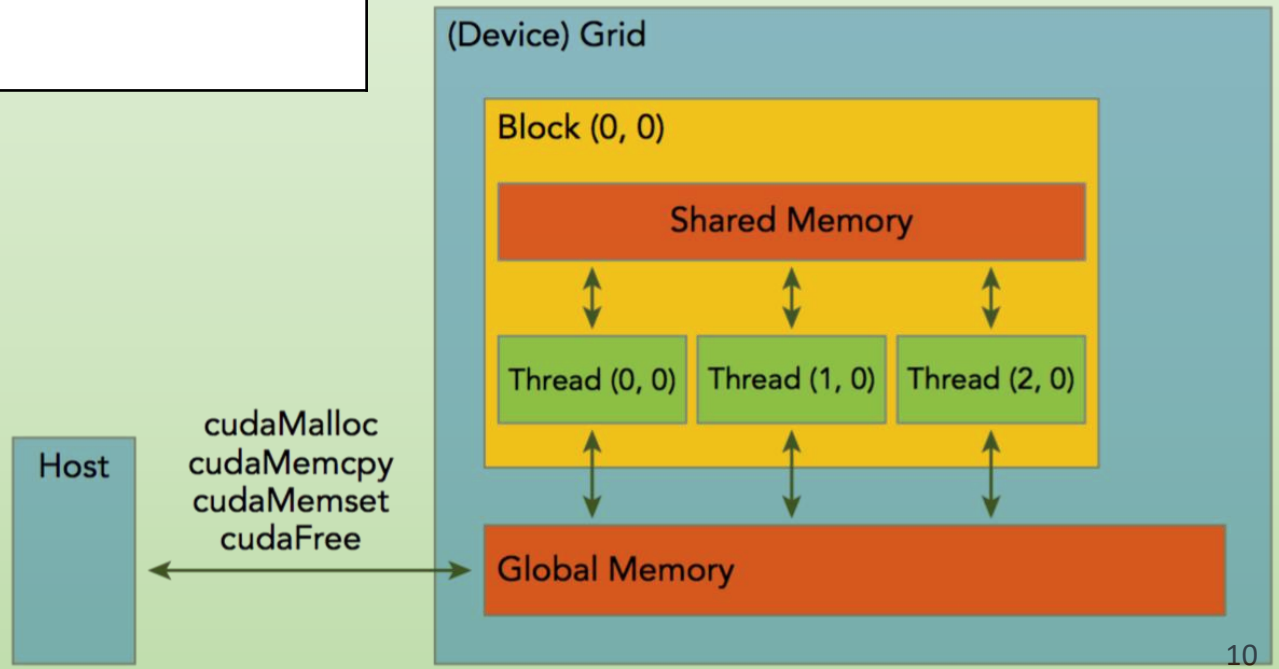
 # TODO

1. La **sequenza** di lunghezza 1000 è posta in un `array_host` di **FP32** allocata in memoria
2. **trasferimento** dell'`array_host` nell'array `array_dev` in **memoria device**
3. Progetto del kernel per la somma parziale
   - Ogni **thread** si occupa di calcolare una **entry** dell'array `array_dev`
4. **trasferimento** dell'`array_dev` nell'array `array_host` in **memoria device**

Gestione della Memoria (preview)

Funzioni C standard	Funzioni CUDA C
`malloc`	`cudaMalloc`
`memcpy`	`cudaMemcpy`
`memset`	`cudaMemset`
`free`	`cudaFree`

- ✓ When you use **cudaMemcpy** to copy data between the host and device, **implicit synchronization** at the host side is performed and the host application must wait for the data copy to complete



cudaMalloc e cudaMemcpy

Segnatura

```
cudaError_t cudaMalloc ( void** devPtr, size_t size )
```

- devPtr è un puntatore a un puntatore in Global memory del device

Segnatura

```
cudaError_t cudaMemcpy ( void* dst, const void* src, size_t count,
                         cudaMemcpyKind kind )
```

- Questa funzione esibisce un comportamento sincrono che blocca il programma host fino a che il trasferimento non viene completato
- Per capire se **destination** e **src** sono puntatori a memoria CPU o GPU si guarda la variabile **kind**

kind

```
cudaMemcpyHostToHost, cudaMemcpyHostToDevice
cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice
```

Segnatura

```
cudaError_t cudaFree ( void* devPtr)
```

- libera la global memory puntata da **devPtr**

Esempio: allocazione su host e device

Allocazione su host (CPU)

```
float *h_A;
nBytes = n * sizeof(float)
h_A = (float *) malloc(nBytes);
```

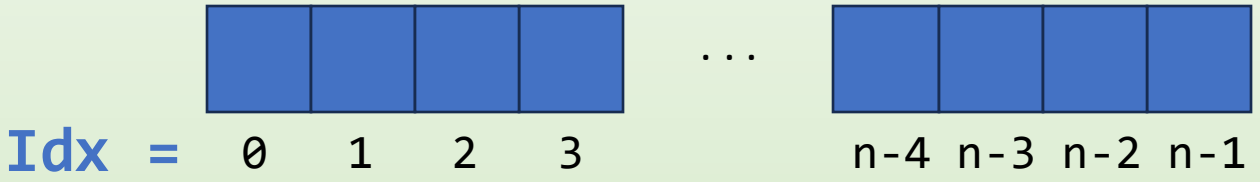
Allocazione su device (GPU)

```
float *d_A;
cudaMalloc((void **) &d_A, nBytes);
```

✓ **NOTA**: Notare l'uso degli identificatori specifici per i due dispositivi

Indicizzazione con thread

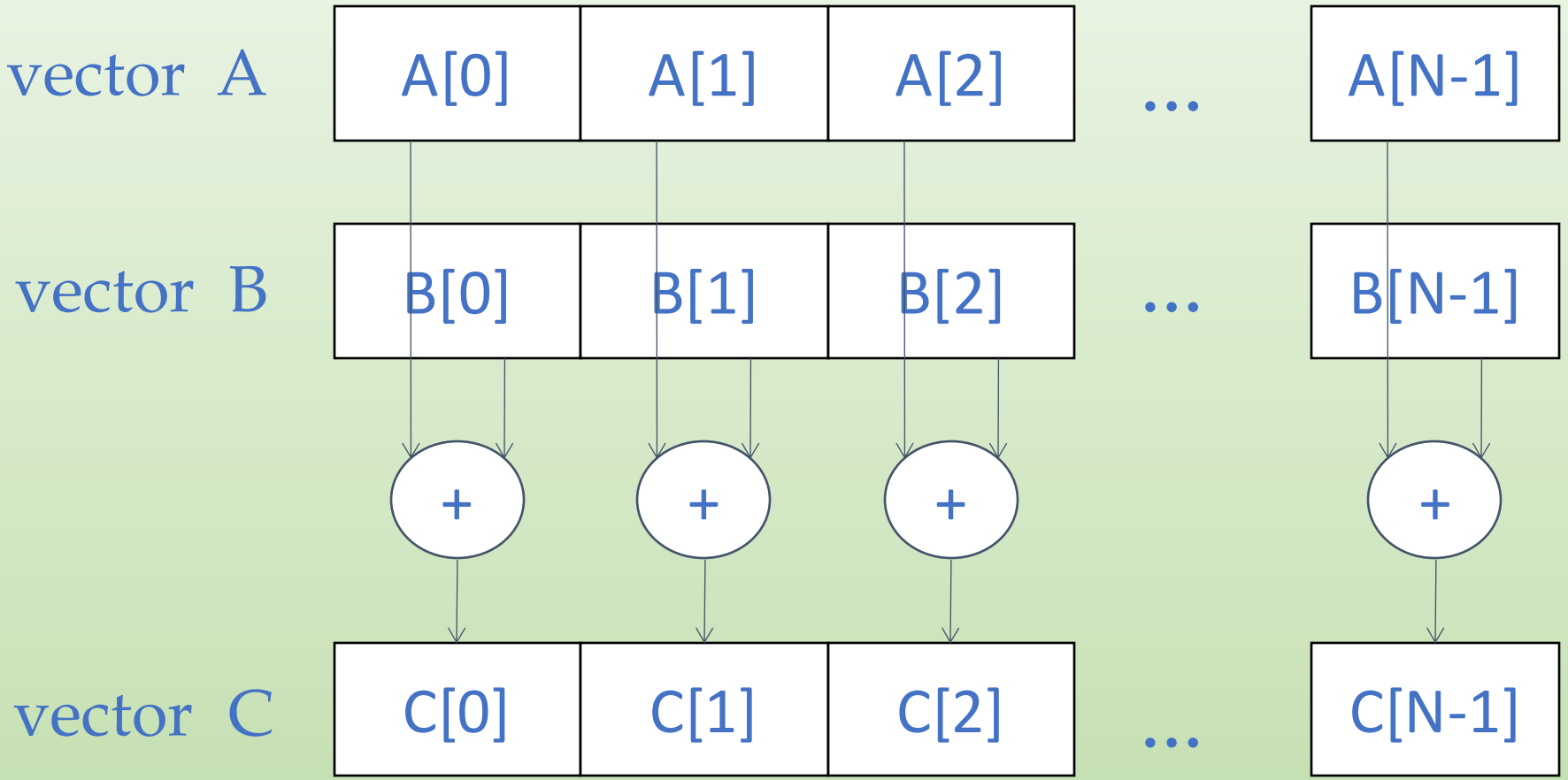
thread... uno per ogni entry del vettore



Indice del thread – var builtin

```
int idx = threadIdx.x;
```

Parallelismo nei dati: somma di vettori



Somma di vettori: kernel

Dime array
<= 1024!

Indice array = ID
del thread

Uso dell'indice
sul dato

```
#include <stdio.h>
#define N 1000 // vector size <= 1024

/* kernel: vector add */
__global__ void add_vect(int *a, int *b, int *c) {
    int idx = threadIdx.x;
        c[idx] = a[idx] + b[idx];
}
```

Somma di vettori: memoria

dev_a, dev_b, dev_c sono puntatori a puntatori della global memory del device

```
int main(void) {
int *a, *b, *c;
int *dev_a, *dev_b, *dev_c;
int nBytes = N * sizeof(int);

// malloc host memory
a = (int *) malloc(nBytes);
b = (int *) malloc(nBytes);
c = (int *) malloc(nBytes);

// malloc device memory
cudaMalloc((void**) &dev_a, nBytes);
cudaMalloc((void**) &dev_b, nBytes);
cudaMalloc((void**) &dev_c, nBytes);
```

```
// Free host memory
free(a);
free(b);
free(c);

// free the memory allocated on the GPU
cudaFree(dev_a);
cudaFree(dev_b);
cudaFree(dev_c);
```

Liberare memoria alla fine

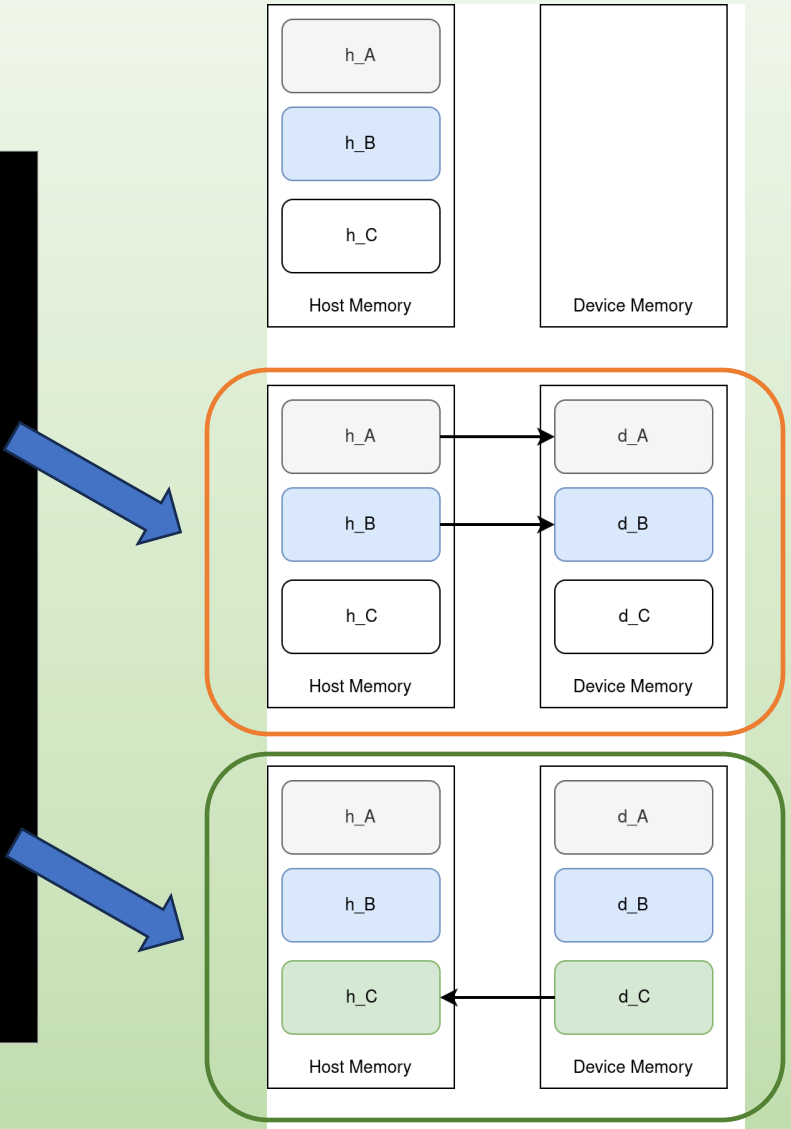
H-to-D, D-to-H copy e invocazione del kernel

```
// copy the arrays 'a' and 'b' to the GPU
cudaMemcpy(dev_a, a, nBytes, cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b, nBytes, cudaMemcpyHostToDevice);

add_vect<<<N>>>(dev_a, dev_b, dev_c);

// copy the array 'c' back from the GPU to the CPU
cudaMemcpy(c, dev_c, nBytes, cudaMemcpyDeviceToHost);

. . .
```



Schema algoritmico

kernel

main

```
// kernel cumsum
__global__ void ...

// Main function
int main(){

    1. allocate space for vectors in host memory

    2. put 1 in all entries of the vector A

    3. allocate space for vectors in device memory

    4. copy vectors A and B from host to device:

    5. launch the vector adding kernel

    6. wait for the kernel to finish execution

    7. copy from device memory

    8. print some results

   return 0;
}
```

Compilazione & esecuzione

Compilazione nella cella jupyter (escape char ‘!’)

```
!nvcc -arch=sm_75 src/misc/cumsum.cu -o cumsum

!./cumsum
```

Stampa del risultato (10 a casa nel vettore somma cumulata)

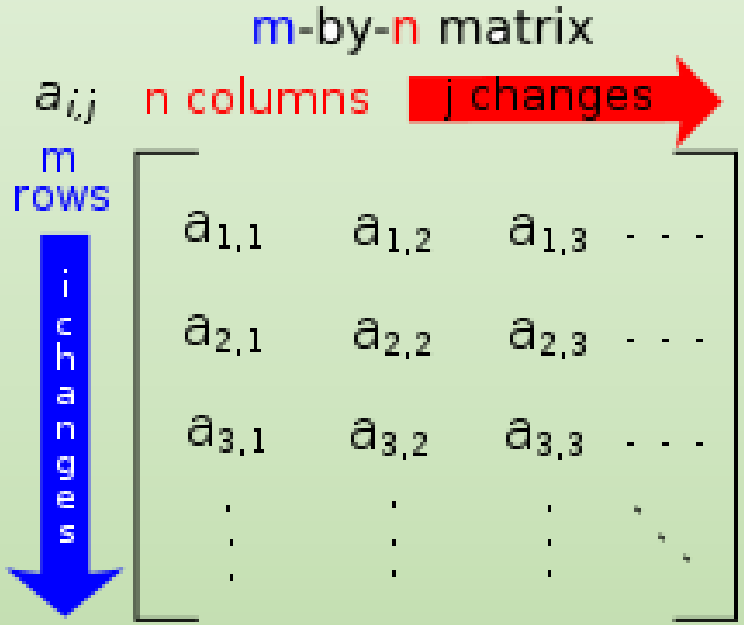
```
B[255] = 256.000000
B[578] = 579.000000
B[251] = 252.000000
B[547] = 548.000000
B[360] = 361.000000
B[126] = 127.000000
B[221] = 222.000000
B[625] = 626.000000
B[824] = 825.000000
B[398] = 399.000000
```

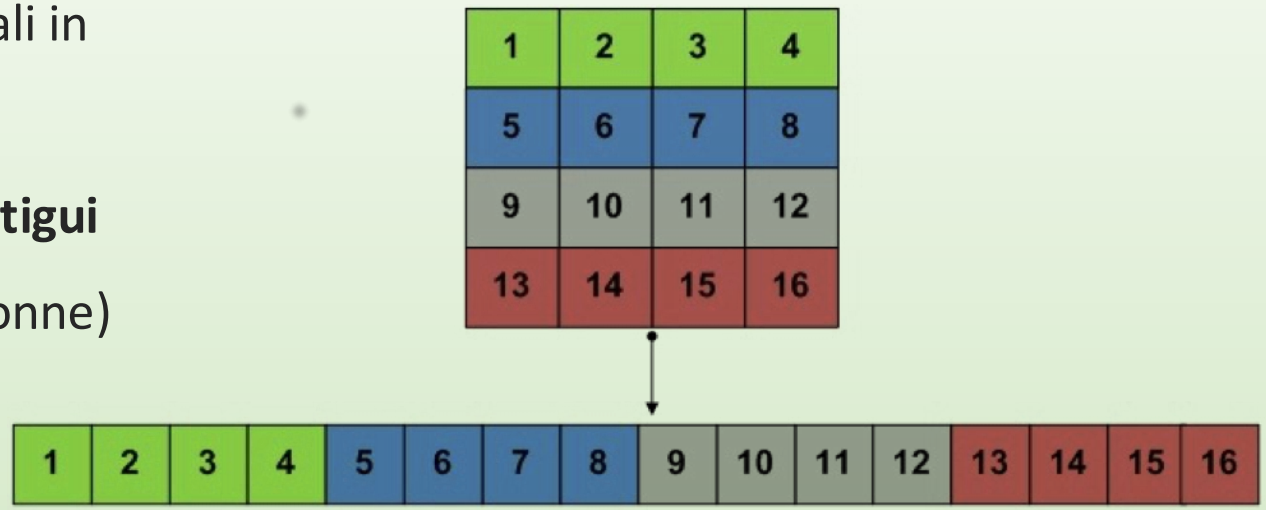
Array multidimensionali in C

Esercitazione su prodotti di matrici “linearizzate”

Allocazione dinamica della memoria

- ✓ C organizza i dati di array multidimensionali in **row-major order** (''linearizzati'')
- ✓ Elementi **consecutivi** delle **righe** sono **contigui**
- ✓ Esempio: matrice $m \times n$ (m righe e n colonne)



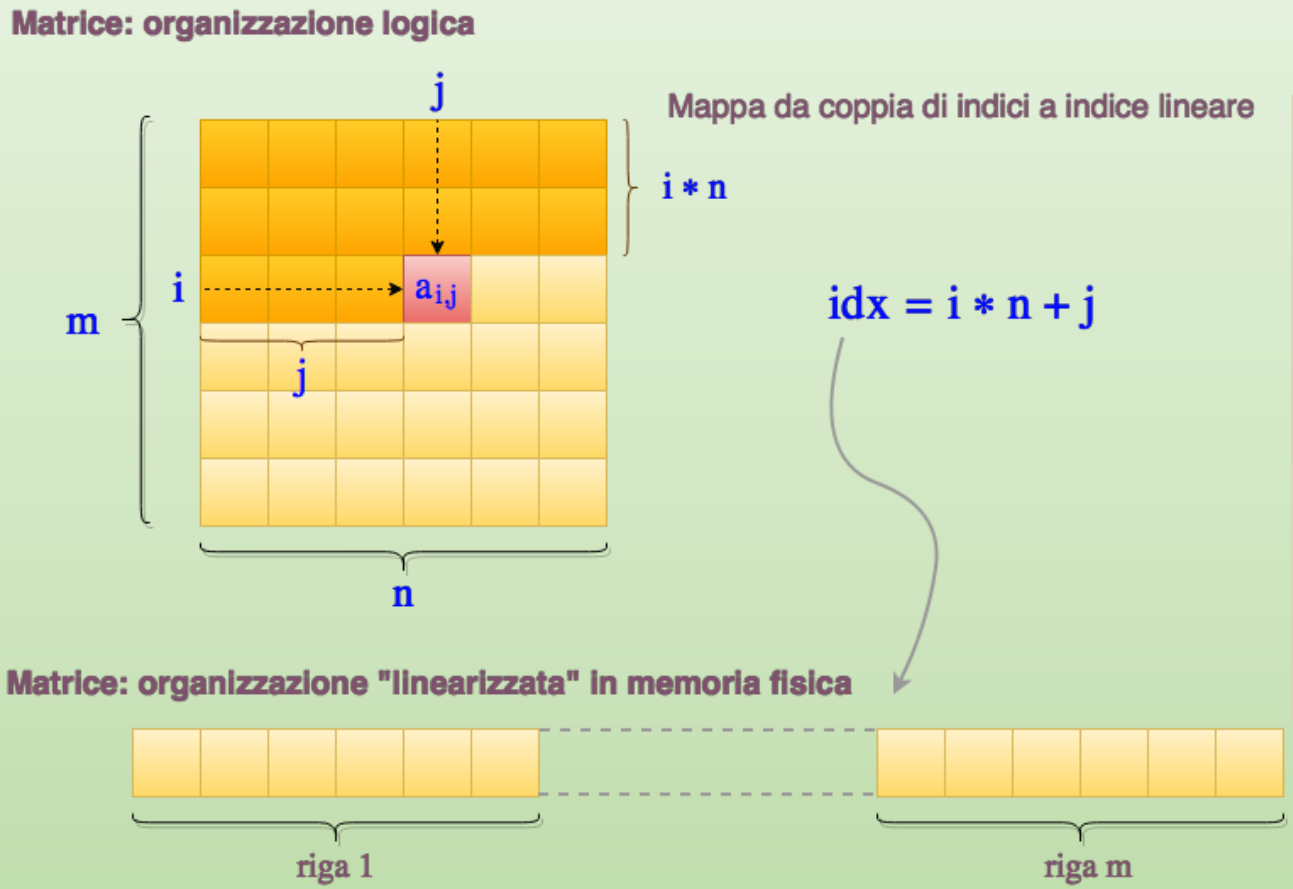



```
// azzeramento della matrice generata dinamicamente
A = (int *) malloc(m * n * sizeof(int));

for (int i = 0; i < m; i++) {
  for (int j = 0; j < n; j++) {
    // calcola idx (indice linearizzato)
    A[idx] = 0;
  }
}
```

Allocazione dinamica e indicizzazione

- ✓ **Allocazione fisica** di dati in memoria e **accesso logico** alle strutture dati in C
- ✓ Codice C per l'azzeramento di dati in una matrice di **m** righe e **n** colonne con **accesso** tramite **indice linearizzato**



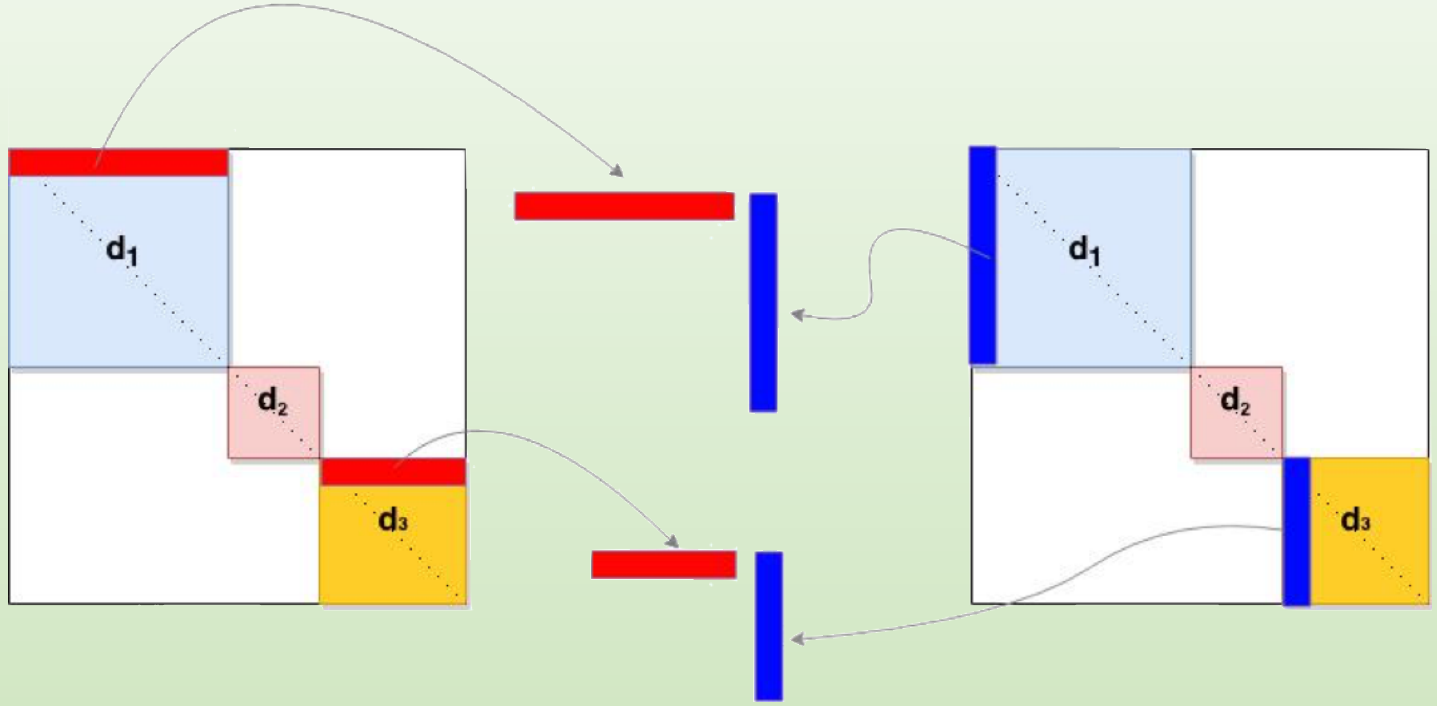

```
// azzeramento della matrice generata
// dinamicamente

A = (int *) malloc(n * m * sizeof(int));

for (int i = 0; i < m; i++) {
  for (int j = 0; j < n; j++) {
    idx = i * n + j;  // indice linearizzato
    A[idx] = func(i,j); // funz di i e j
  }
}
```

Esercizio: Prodotto di matrici diagonali a blocchi

- ✓ L'algoritmo naïve per una matrice quadrata esegue in tempo $\mathbf{O}(n^3)$
- ✓ L'algoritmo di Strassen, basato sul prodotto efficiente di matrici, esegue in tempo $\mathbf{O}(n^{2.8})$
- ✓ Algoritmi più efficienti arrivano a $\mathbf{O}(n^{2.37})$ sfruttando il prodotto di matrici $k \times k$
- ✓ Il tempo di esecuzione per il prodotti di matrici diagonali a blocchi di dim $k_1, k_2, \ldots, k_k$ è $\sum_{i=1}^{k} k_i^3$



Sviluppare un programma C che implementi in modo efficiente il prodotto di **matrici diagonali a blocchi** ammettendo di conoscere le dimensioni (contenute in un array) dei blocchi componenti la matrice

Risultato

```
volta[~]->mat_blk_prod

matrix A:
  4  4  4  4
  4  4  4  4
  4  4  4  4
  4  4  4  4
          2  2
          2  2
              1
                 3  3  3
                 3  3  3
                 3  3  3
```

```
matrix B:
  4  4  4  4
  4  4  4  4
  4  4  4  4
  4  4  4  4
          2  2
          2  2
              1
                 3  3  3
                 3  3  3
                 3  3  3
```

```
matrix C:
 64 64 64 64
 64 64 64 64
 64 64 64 64
 64 64 64 64
          8  8
          8  8
              1
                 27 27 27
                 27 27 27
                 27 27 27
```

Esercizio: prodotto di Kronecker

Date le matrici:

$$A \in \mathbf{R}^{n \times m}, \qquad B \in \mathbf{R}^{p \times q}$$

Calcolare:

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} A_{11}\mathbf{B} & A_{12}\mathbf{B} & \cdots & A_{1m}\mathbf{B} \\ A_{21}\mathbf{B} & A_{22}\mathbf{B} & \cdots & A_{2m}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1}\mathbf{B} & A_{n2}\mathbf{B} & \cdots & A_{nm}\mathbf{B} \end{bmatrix}$$

Sviluppare un programma C che implementi in modo efficiente il prodotto di Kronecker tra due **matrici diagonali a blocchi**