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)

Hello world in C ->

```
#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:

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

Hello world in CUDA

Hello world in CUDA ->

```
__global__ void helloFromGPU(void) {
    printf("Hello World from GPU!\n");
}
```

- ✓ Il qualificatore `__global__` dice al compilatore che la function (kernel) è chiamata dalla CPU ma eseguita in GPU
- ✓ L'invocazione del kernel : `helloFromGPU <<< 1, 10 >>>();`
- ✓ La **tripla parentesi angolare** `<<< * , * >>>` denota una **chiamata** dal **codice host** al **codice device** che ne attiva l'esecuzione
- ✓ Un **kernel** viene eseguito da un **array di thread** e tutti i thread eseguono lo **stesso codice**
- ✓ 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!**

Listato finale

```
#include <stdio.h>

__global__ void helloFromGPU (void) {
    printf("Hello World from GPU!\n");
}

int main(void) {
    // hello from GPU
    printf("Hello World from CPU!\n");
    helloFromGPU <<<1, 10>>>();
    cudaDeviceReset();
    return 0;
}
```

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

- ✓ La compilazione richiede lo switch **`-arch sm_20`** per generare un codice eseguibile su architettura **Fermi** capability **2.0**

```
$ nvcc -arch sm_20 hello.cu -o
hello
$ ./hello
Hello World from CPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
```

In sintesi

Struttura di un programma CUDA

1. Allocare la memoria GPU
2. Copiare i dati da memoria CPU a memoria GPU
3. Invocare il kernel CUDA
4. Copiare i dati da memoria GPU a memoria CPU
5. Ripulire le memorie GPU

Tool di sviluppo CUDA

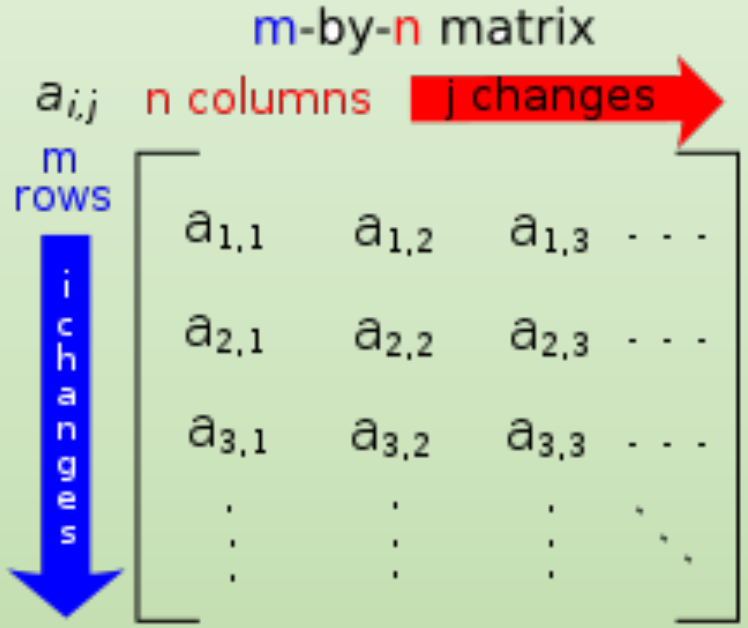
1. Ambiente integrato di sviluppo NVIDIA Nsight™
2. Debugger a linea di comando CUDA-GDB
3. Profiler grafico o a linea di comando per analisi di prestazioni
4. Analizzatore CUDA-MEMCHECK per la memoria
5. Tool per la gestione del device GPU

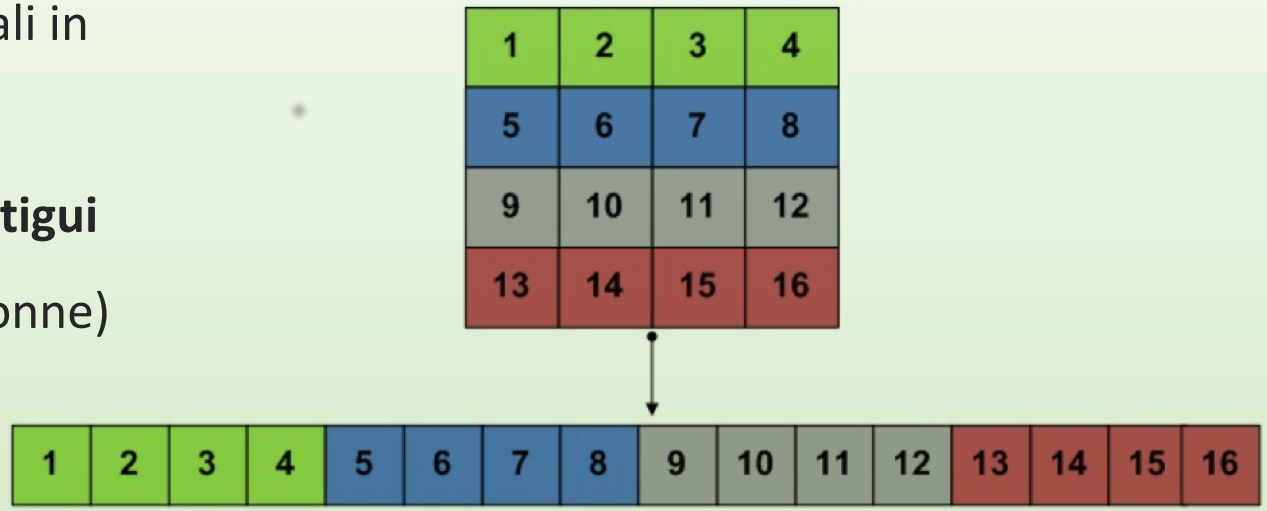
Array multidimensionali in C

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)



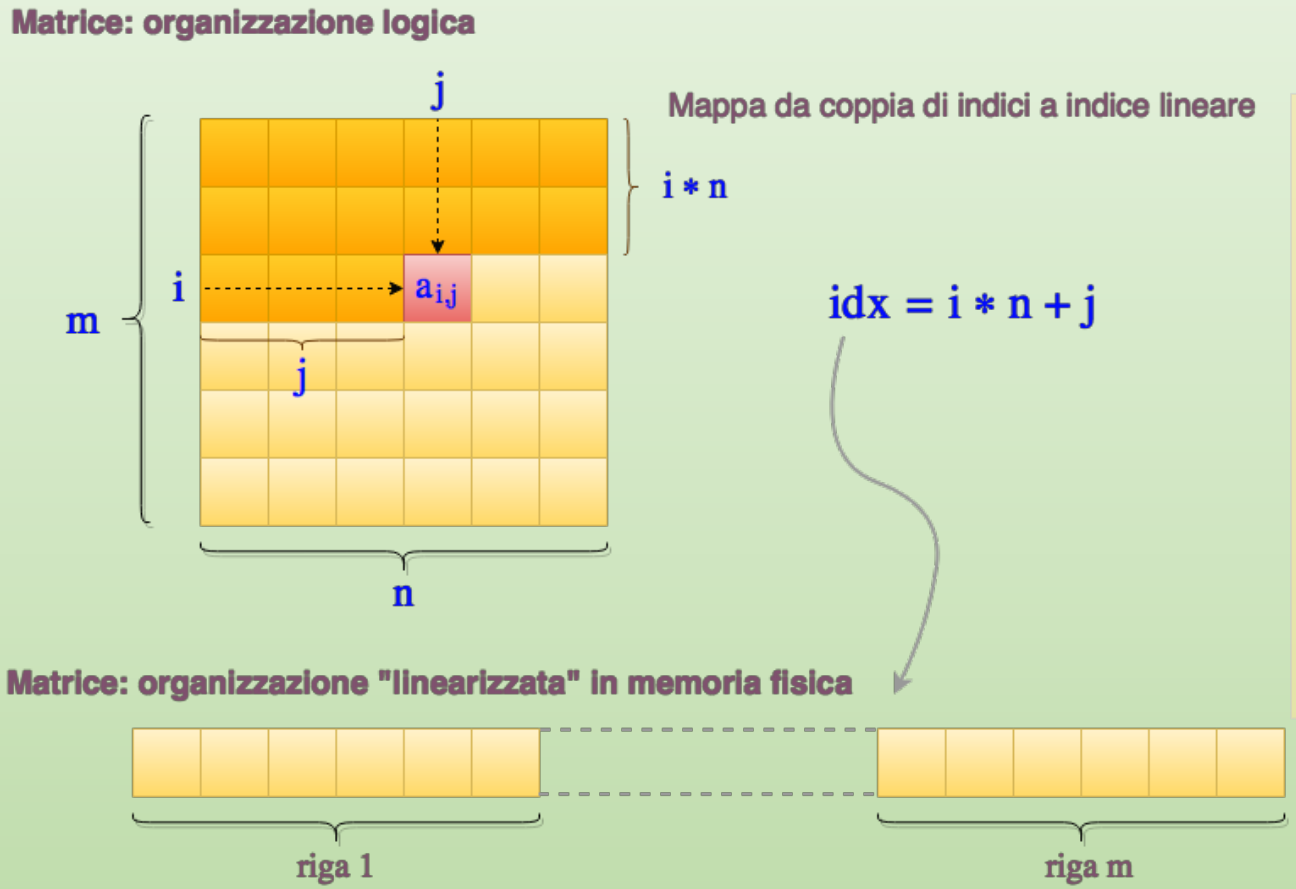



```
// azzeramento della matrice generata dinamicamente
A = (int *) malloc(m * n * sizeof(int));

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**



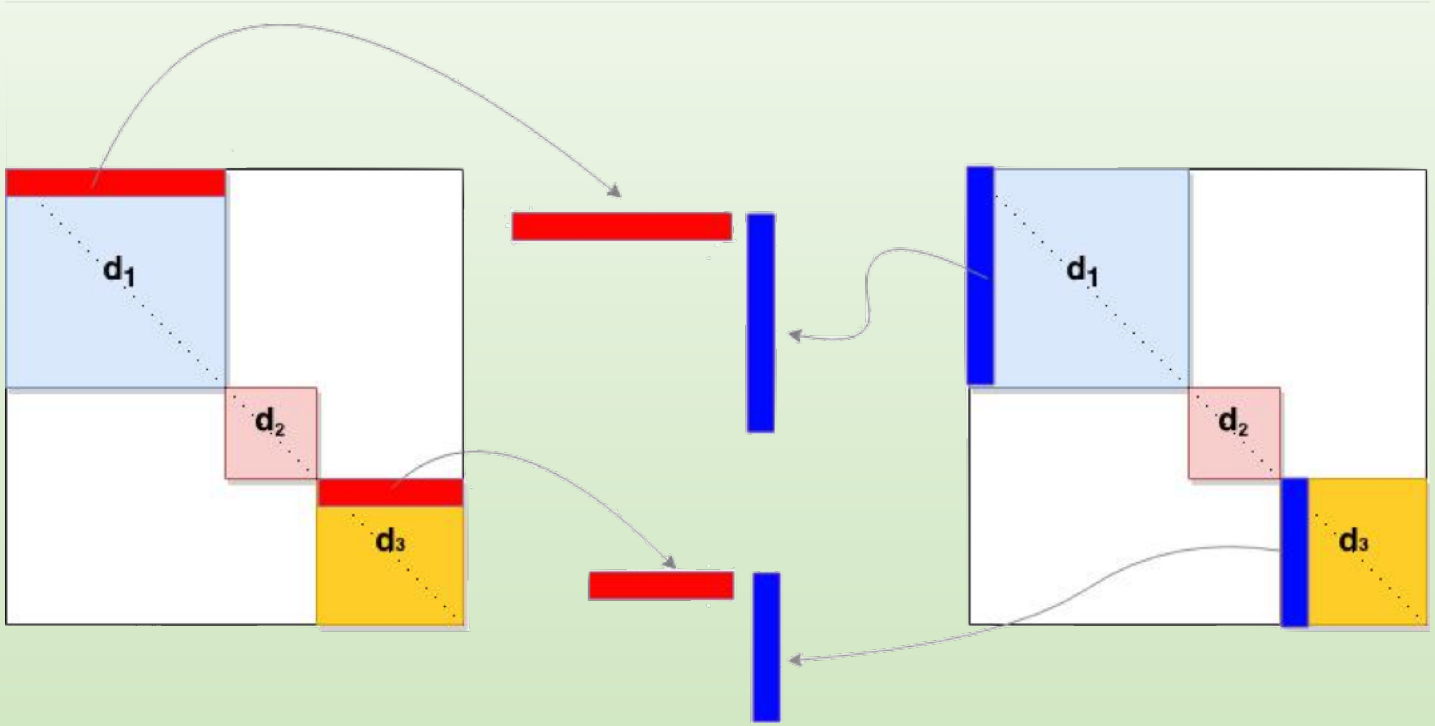

```
// azzeramento della matrice generata
// dinamicamente

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 $d_1, d_2, \ldots, d_k$ è $\sum_{i=1}^{k} d_i^3$



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

```
pascal[~]->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**