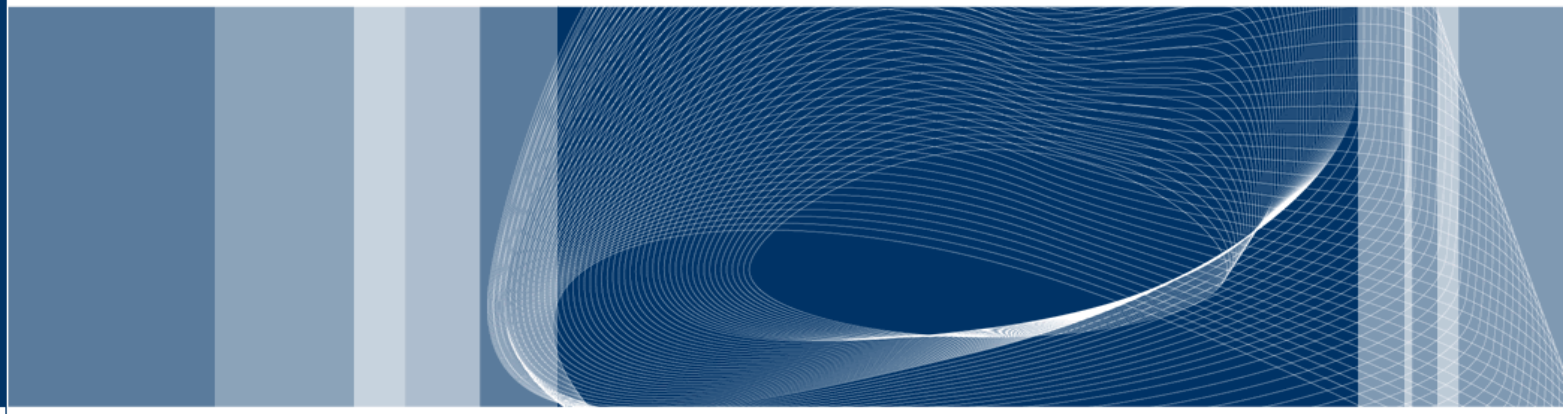
POLITECNICO DI MILANO



Esercizi su CUDA

Algoritmi e Calcolo Parallelo

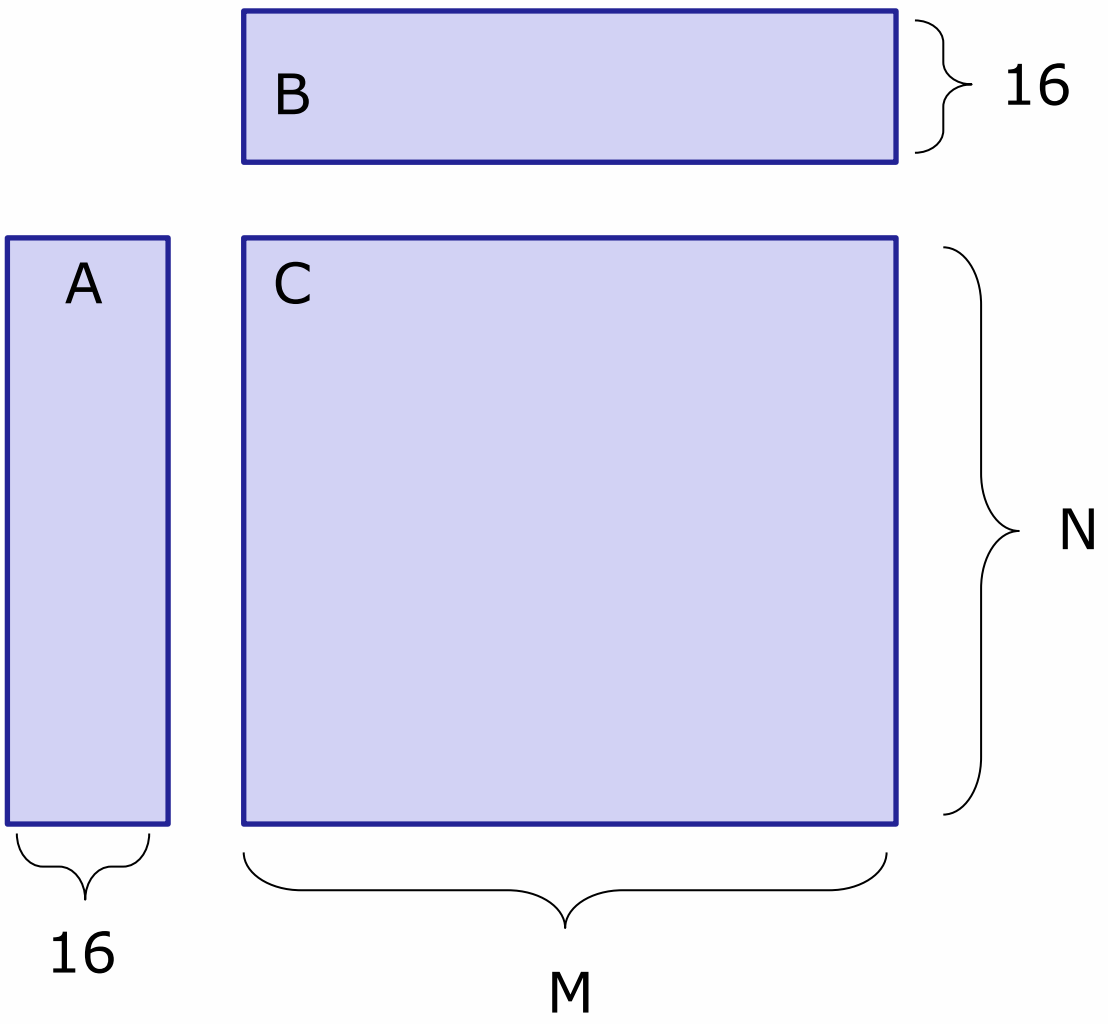
Daniele Loiacono

Esercizio 1

- Implementare in CUDA il prodotto fra due matrici

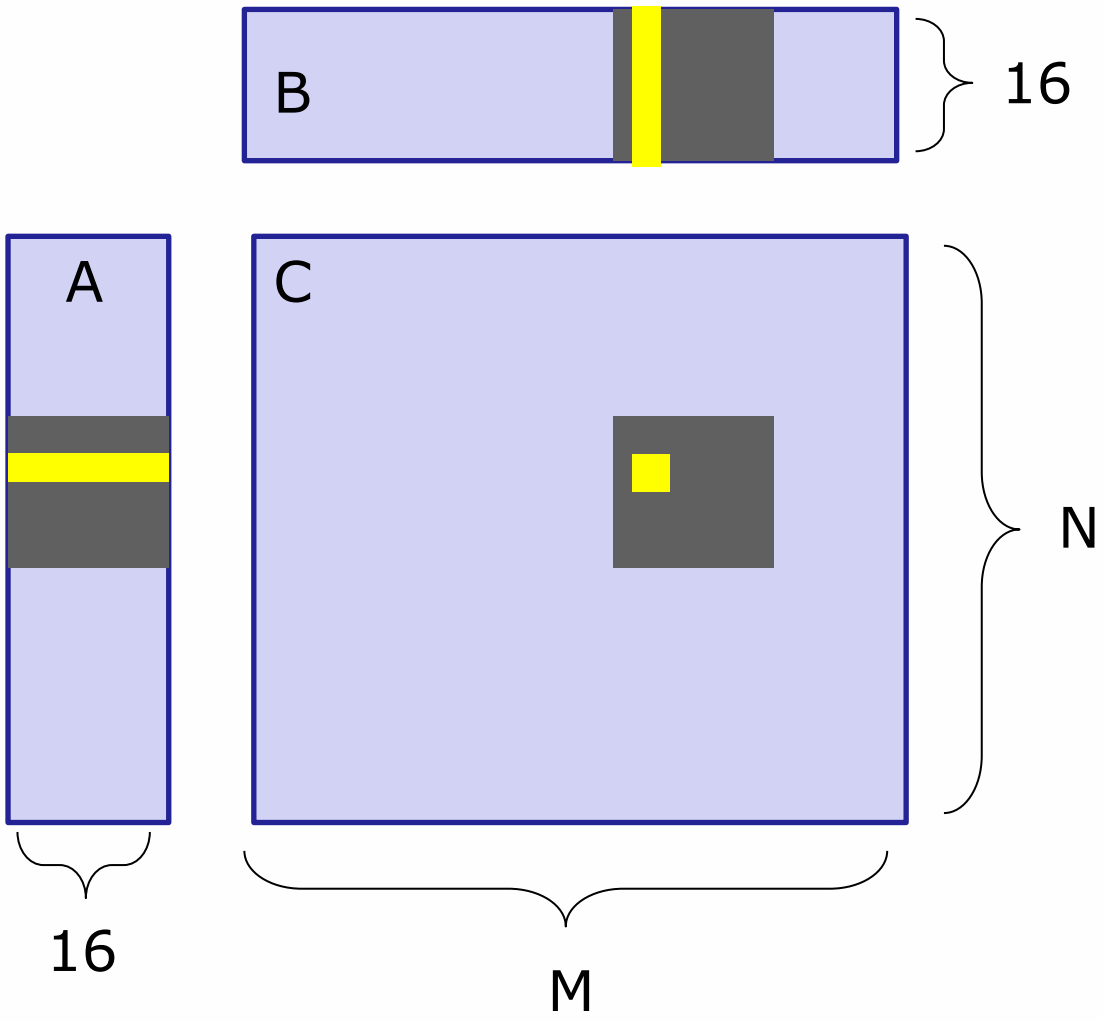
$$C = AB$$

  - A è una matrice Nx16
  - B è una matrice 16xM
  - M ed N sono multipli interi di 16



Soluzione

- Le matrici A e B vengono divise in tile 16x16
- Ogni tile viene assegnata ad un blocco di thread
- Ogni thread calcola un elemento della matrice C

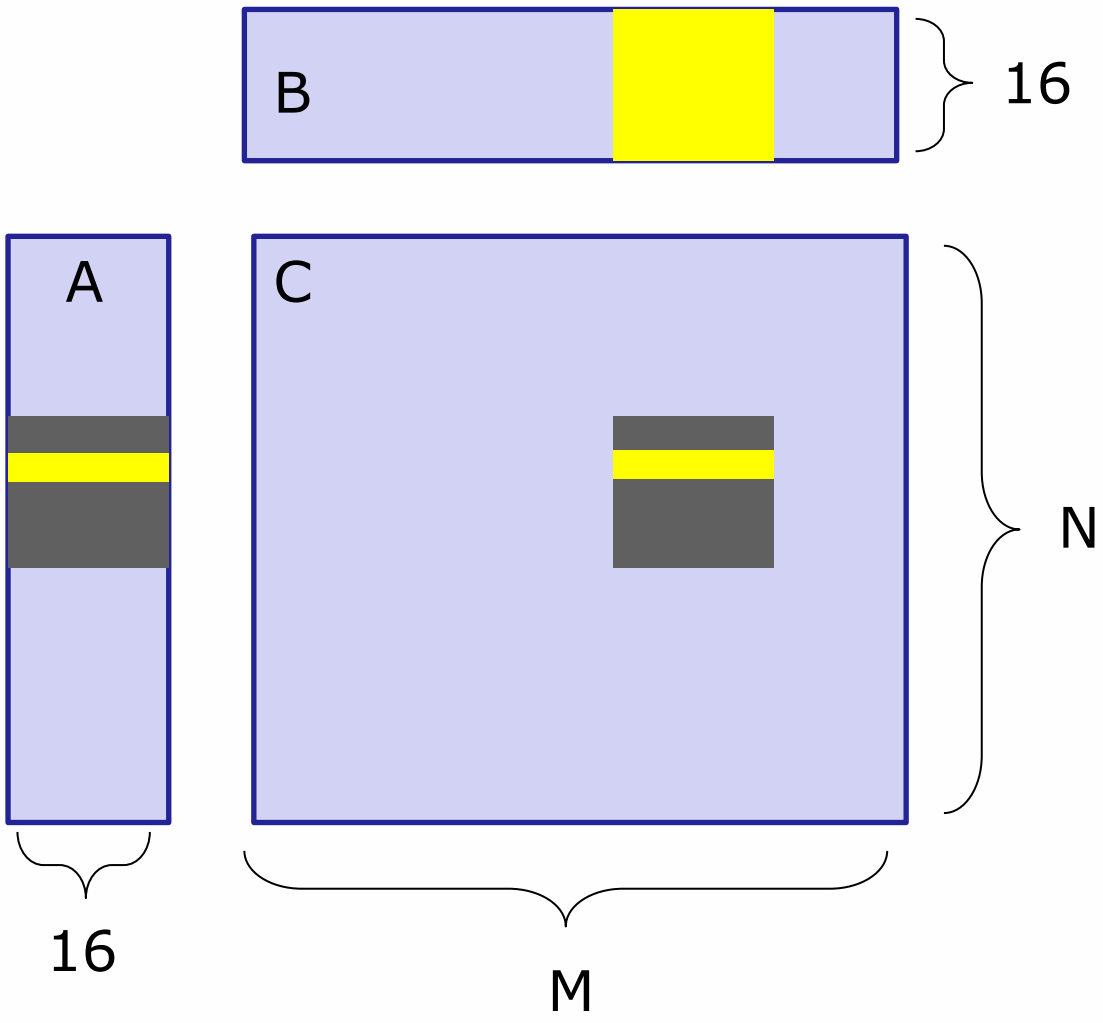


Soluzione: implementazione

```
__global__ void simpleMultiply(float *a, float* b,
                               float *c, int N){
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  float sum = 0.0f;
  for (int i = 0; i < TILE_DIM; i++) {
    sum += a[row*TILE_DIM+i] * b[i*N+col];
  }
  c[row*N+col] = sum;
}
```

Soluzione: ottimizzazione

- Un half-warp calcola una riga della matrice C
  - ogni thread legge la stessa riga di A
  - ad ogni iterazione, la lettura dello stesso elemento di A richiede
    - 16 transazione su schede < 1.2
    - 1 transazione su schede >= 1.2
  - la lettura di A avviene più volte inutilmente
- L'uso della memoria shared può migliorare le prestazioni



Soluzione: implementazione (2)

```
__global__ void coalescedMultiply(float *a, float*
                                  b, float *c, int N) {
  __shared__ float aTile[TILE_DIM][TILE_DIM];
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  float sum = 0.0f;
  aTile[threadIdx.y][threadIdx.x] =
                    a[row*TILE_DIM+threadIdx.x];
  for (int i = 0; i < TILE_DIM; i++) {
    sum += aTile[threadIdx.y][i]* b[i*N+col]; }
  c[row*N+col] = sum;
}
```

Soluzione: implementazione (3)

```
__global__ void sharedABMultiply(float *a, float* b,
                                 float *c, int N) {
  __shared__ float aTile[TILE_DIM][TILE_DIM],
                   bTile[TILE_DIM][TILE_DIM];
  int row = blockIdx.y * blockDim.y + threadIdx.y;
   int col = blockIdx.x * blockDim.x + threadIdx.x;
  float sum = 0.0f;
  aTile[threadIdx.y][threadIdx.x] =
                      a[row*TILE_DIM+threadIdx.x];
   bTile[threadIdx.y][threadIdx.x] =
                            b[threadIdx.y*N+col];
  __syncthreads();
  for (int i = 0; i < TILE_DIM; i++) {
     sum+=aTile[threadIdx.y][i]*bTile[i][threadIdx.x];
  }
  c[row*N+col] = sum;
}
```

Esercizio 2

- Implementare in cuda il prodotto scalare fra due array di float

$$c = a \cdot b = \sum_i a_i b_i$$

  - a e b sono array di uguali dimensioni e con N elementi (N costante definita opportunamente o letta dall'utente)

Problemi che richiedono Numeri Random

- Due possibili approcci
  - Generare i numeri random con la CPU e trasferirli nella memoria della GPU
  - Generare i numeri random tramite la GPU
- Per la generazione random su GPU è possibile usare la libreria **CURAND**
  - Manuale disponibile qui: http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CURAND_Library.pdf
  - Consente la generazione di numeri random sia dall'host sia dal device

CURAND Host API: overview

1. Creazione di un generatore con `curandCreateGenerator()`
2. Definizione del seed del generatore con `curandSetPseudoRandomGeneratorSeed()` e/o altre opzioni
3. Allocare memoria sul device (`cudaMalloc()`)
4. Generazione numeri random con `curandGenerate()` o altre funzioni
5. Esecuzione di uno o più kernel che richiedono i numeri random generati
6. Deallocazione risorse tramite `curandDestroyGenerator()`

CURAND Host API: funzioni generazione

- ```
  curandStatus_t curandGenerate(
        curandGenerator_t generator,
        unsigned int *outputPtr,
        size_t num)
  ```
- ```
  curandStatus_t CurandGenerateUniform(
        curandGenerator_t generator,
        float *outputPtr,
        size_t num)
  ```
- ```
  curandStatus_t curandGenerateNormal(
         curandGenerator_t generator,
         float *outputPtr,
         size_t n,
         float mean,
         float stddev)
  ```

- Analoghe funzioni per generare numeri double
  ```

CURAND Host API: esempio

```
#include <curand.h>
…
size_t n = 100;
curandGenerator_t gen;
float *devData, *hostData;

/* Allocate n floats on host */
hostData = (float *)calloc(n, sizeof(float));

/* Allocate n floats on device */
cudaMalloc((void **)&devData, n * sizeof(float));
```

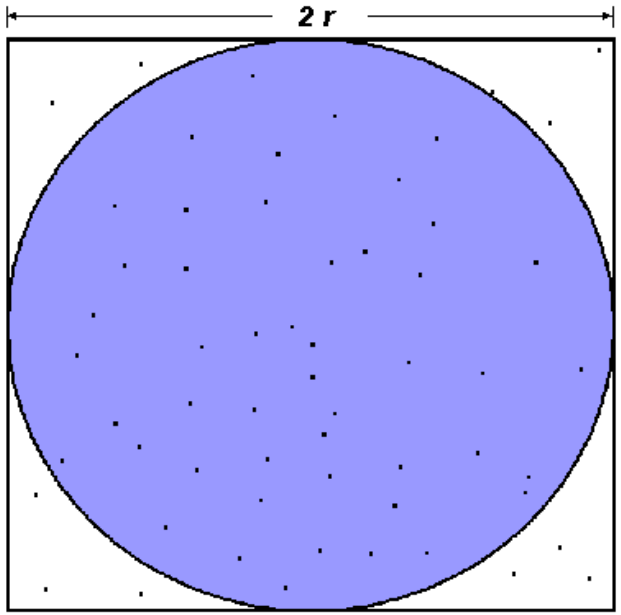
CURAND Host API: esempio (2)

```
…
/* Create pseudo-random number generator */
curandCreateGenerator(&gen,CURAND_RNG_PSEUDO_DEFAULT);
/* Set seed */
curandSetPseudoRandomGeneratorSeed(gen, 1234ULL);
/* Generate n floats on device */
curandGenerateUniform(gen, devData, n);
/* Use devData on device */
…
/* Cleanup */
curandDestroyGenerator(gen);
cudaFree(devData);
free(hostData);
…
```

Esercizio 3

- Implementare in CUDA una soluzione parallela del seguente algoritmo per approssimare PI

```
cin >> npoints; count = 0;
for(j=0, j<npoints; j++) {
    x = random();
    y = random();
    if (inCircle(x, y))
        count++;
}
PI = 4.0*count/npoints
```



  - A che livello parallelizzereste?
  - Pensare ad una implementazione "furba" di `inCircle(x, y)`
    - **Nota:** è possibile riformulare il problema