1

Cosa si intende per GPGPU ?

Calcolo General Purpose su GPU (Graphic Processing Unit) in applicazioni differenti dalla grafica 3D tradizionale

- Le GPU sono usate come acceleratori dei calcoli nelle applicazioni scientifiche

Algoritmi "Data parallel" sono i piu' adatti per le caratteristiche delle GPU

- Grandi array di dati, streaming
- Parallelismo a "grana fine" di tipo SIMD
- Low-latency per operazioni floating point

Applicazioni gia' sviluppate – vedi www.gpgpu.org

- videogames, image processing
- Modellazione di fenomeni fisici, ingegneria computazionale, algebra lineare, FFT, ordinamenti,…

2

Storia delle GPU

- Negli anni '80 furono introdotti chip specializzati per il rendering grafico per sollevare la CPU dalla gestione dell'output
- Prime GPU integrate nella scheda madre e con funzioni fisse (pipeline grafica)
- Anni '90 prime GPU con funzionalita' programmabili attraverso opportune librerie (DirectX, OpenGL)
- Le GPU diventano schede di espansione del bus di sistema
- Fine anni '90 primi tentativi di utilizzare le GPU come CPU
- Meta' anni 2000 sviluppo delle prime GPU appositamente prodotte per l'HPC senza il connettore video e con ambienti di sviluppo "general purpose"

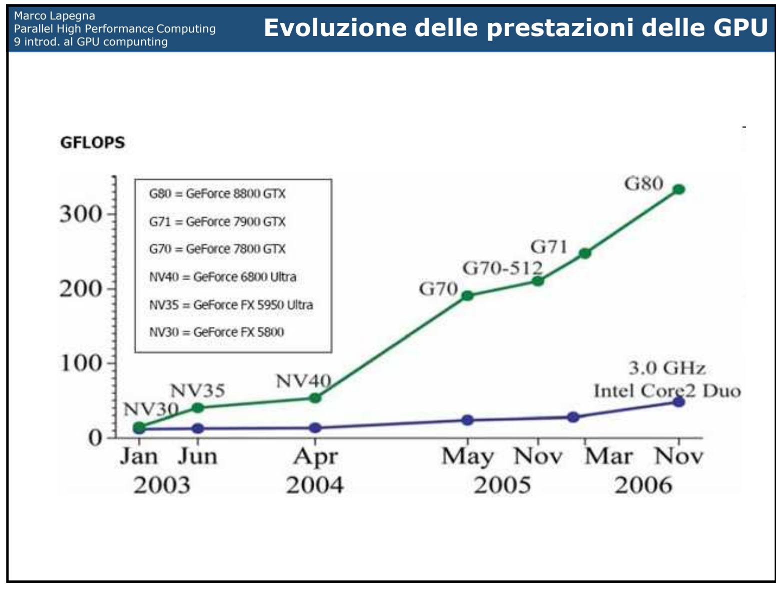
3

GPU di precedente generazione

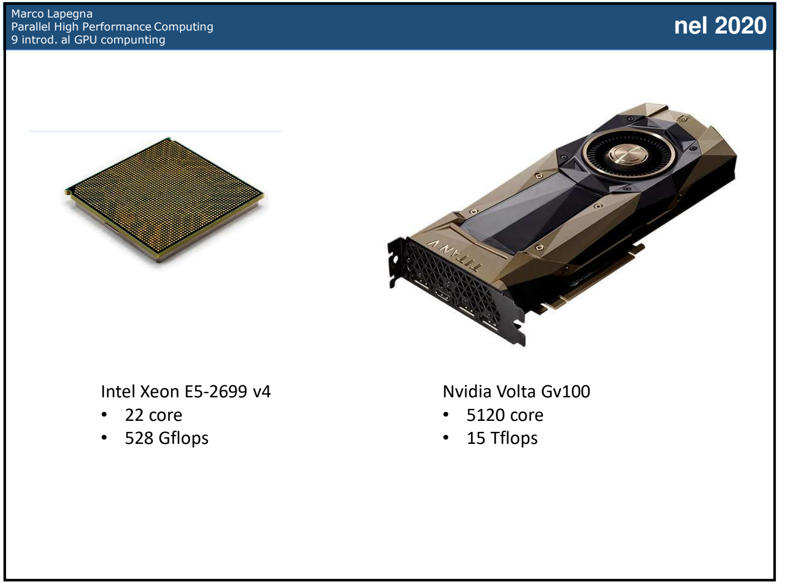
Le GPU di qualche anno fa presentavano alcuni problemi:

- Strumenti rivolti soprattutto ad operazioni di grafica
- necessita' di riscrivere le applicazioni scientifiche in termini di operazioni grafiche
- Assenza di adeguato supporto al floating point
- Indirizzamento in memoria limitato
- Scarso insieme del set di struzioni macchina
- Comunicazione limitata tra eventuali tasks paralleli

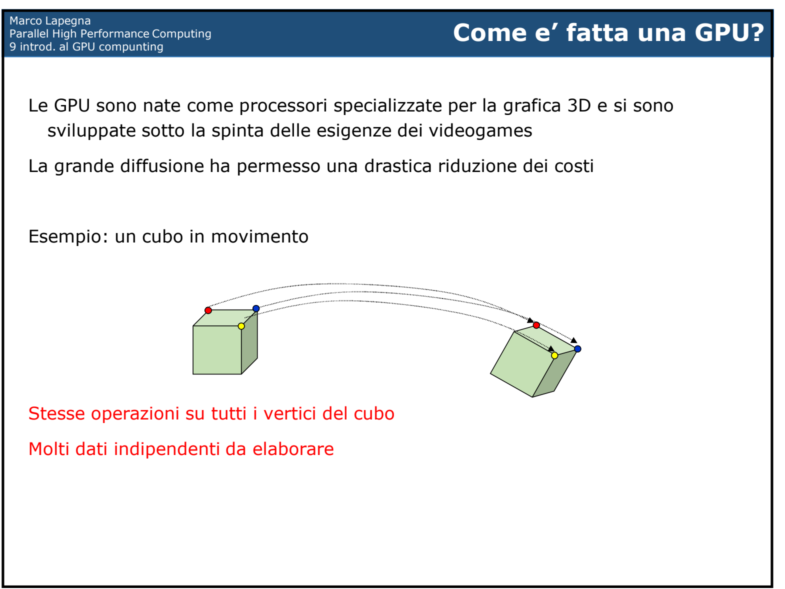
4



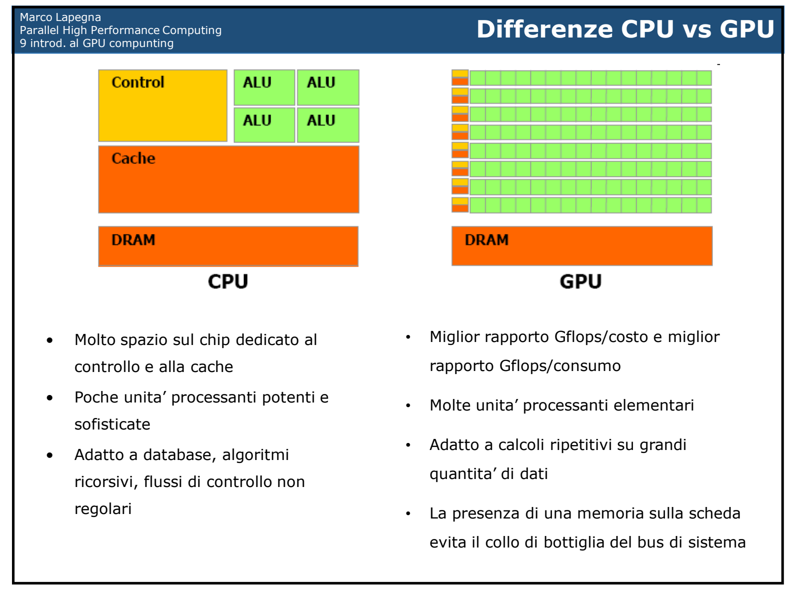
5



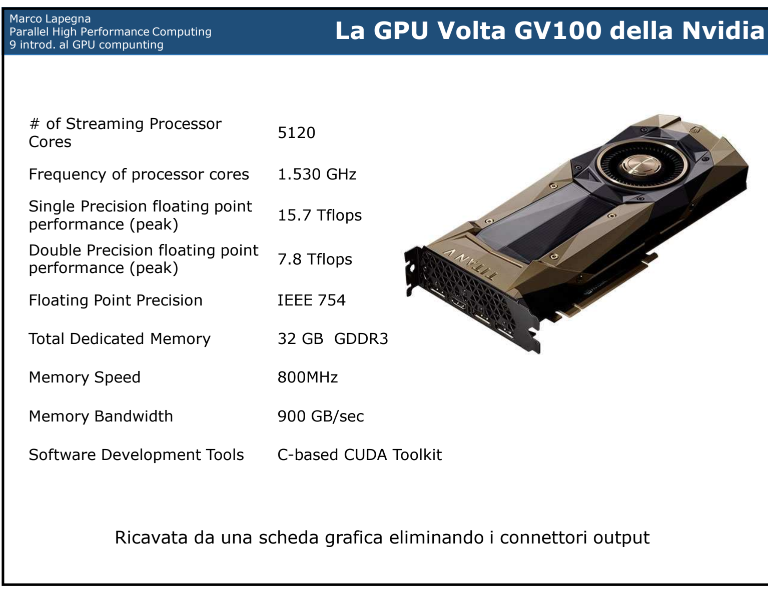
6



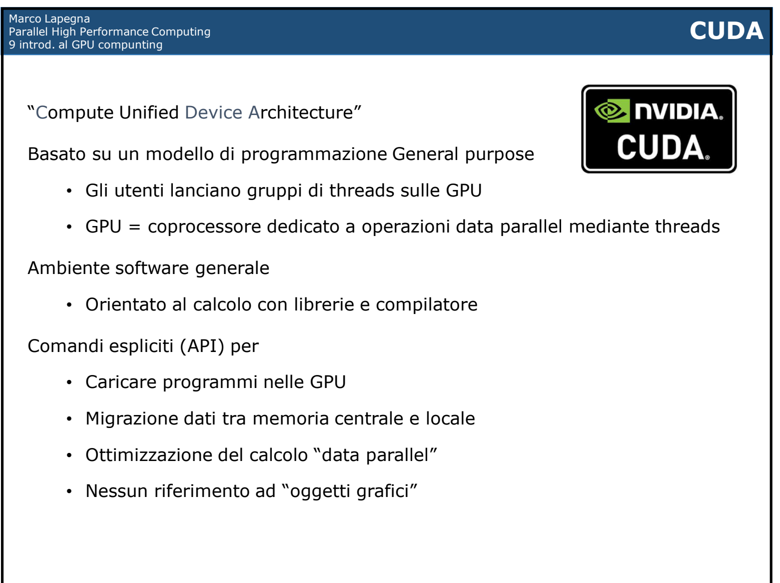
7



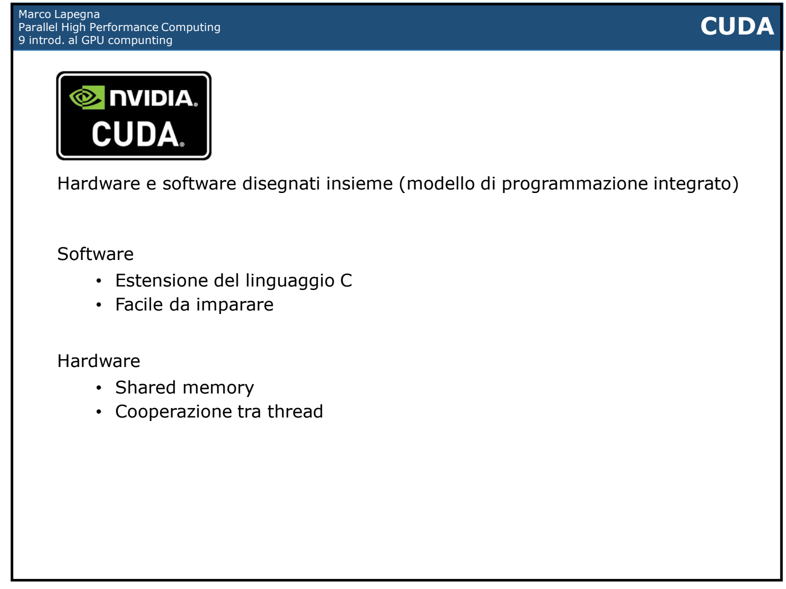
8



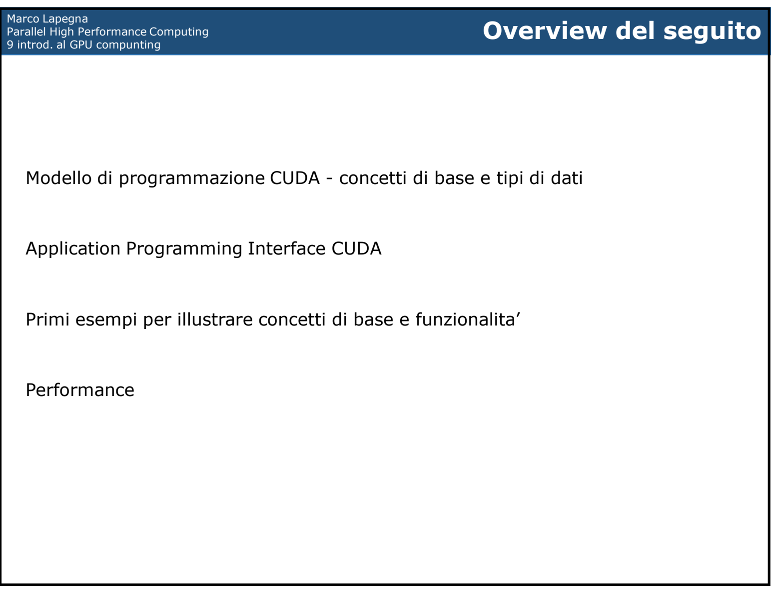
9



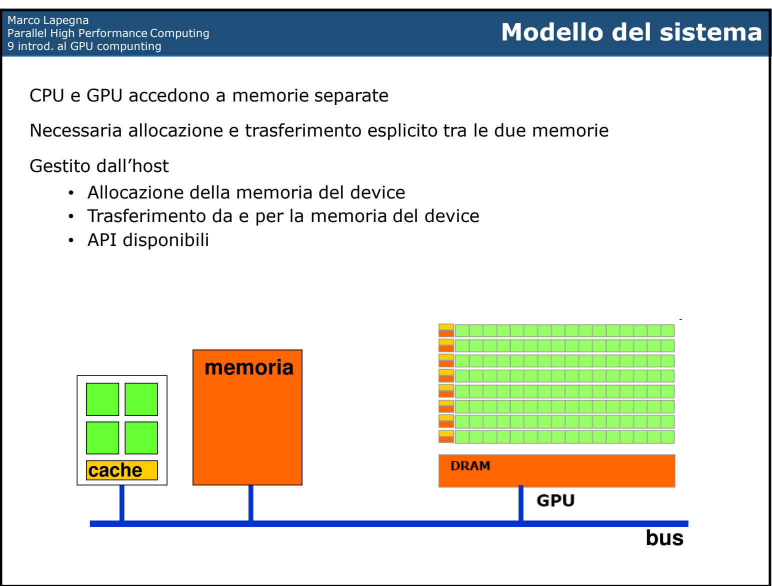
10



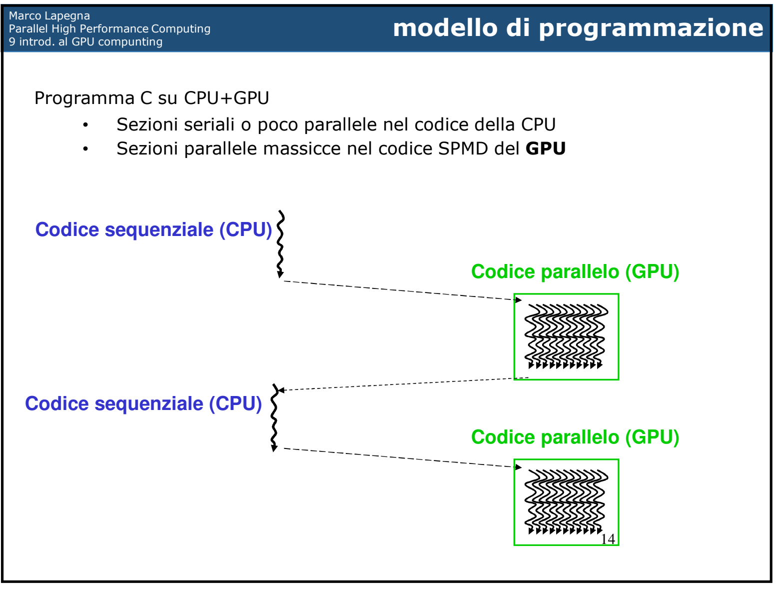
11



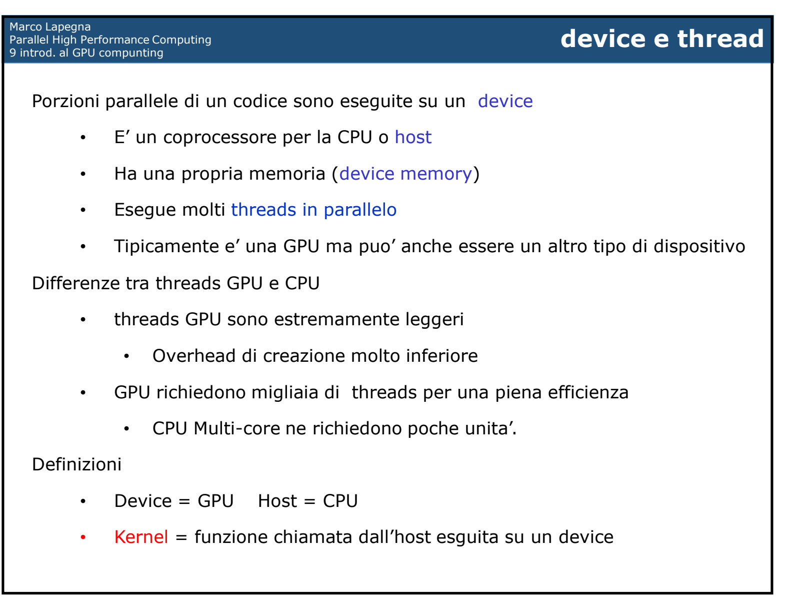
12



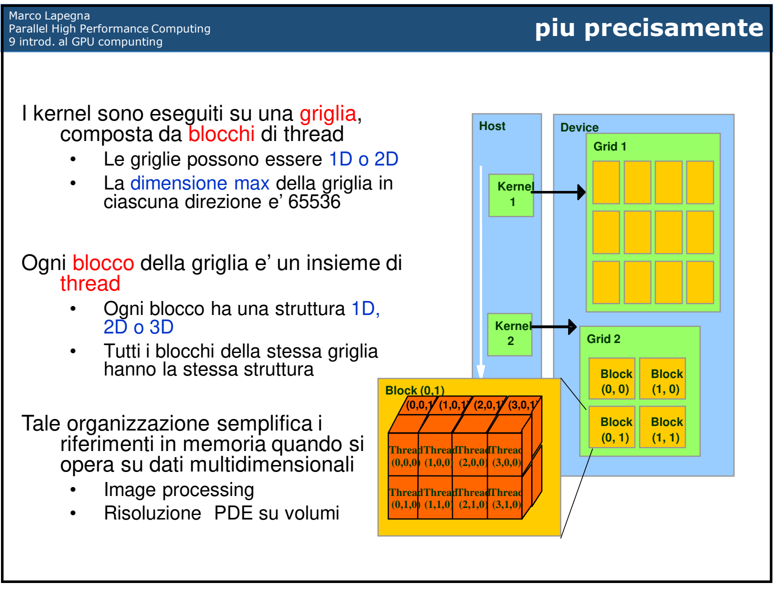
13



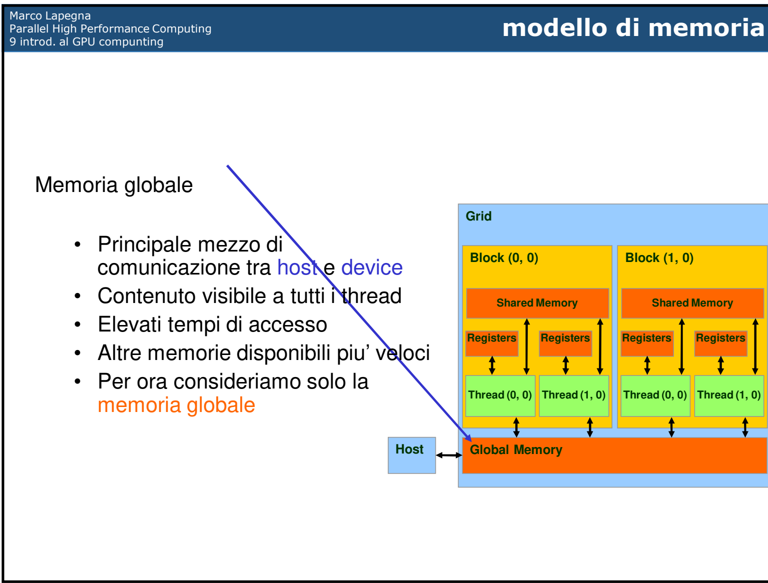
14



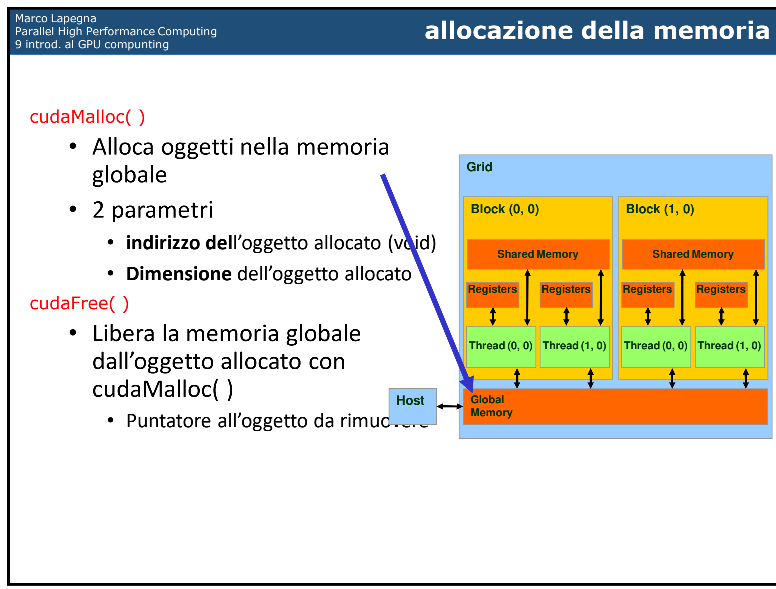
15



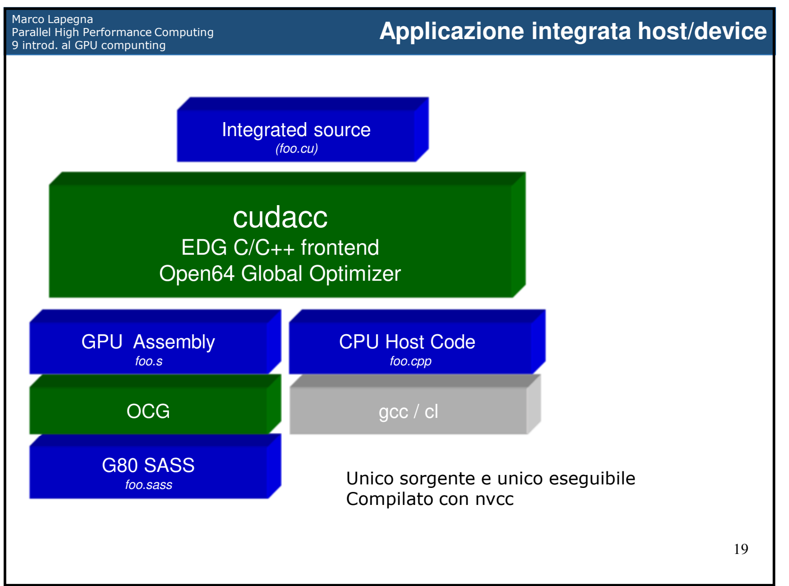
16



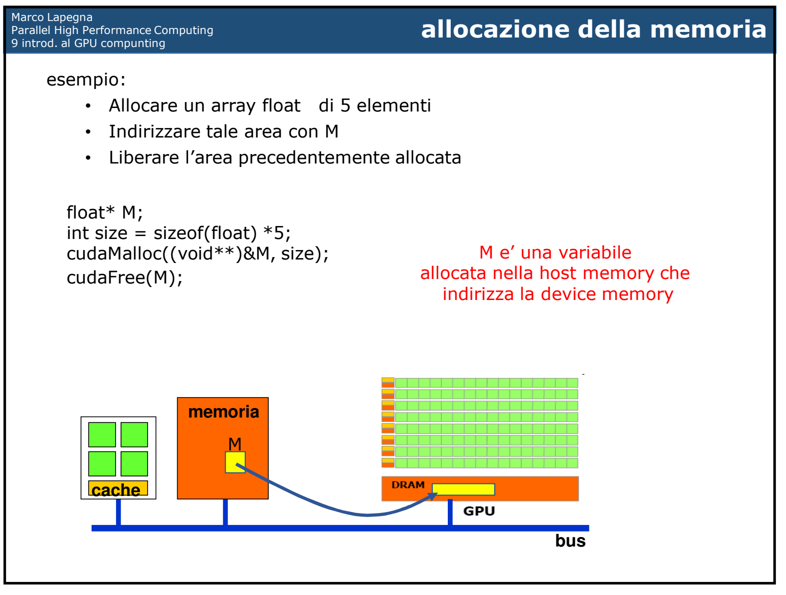
17



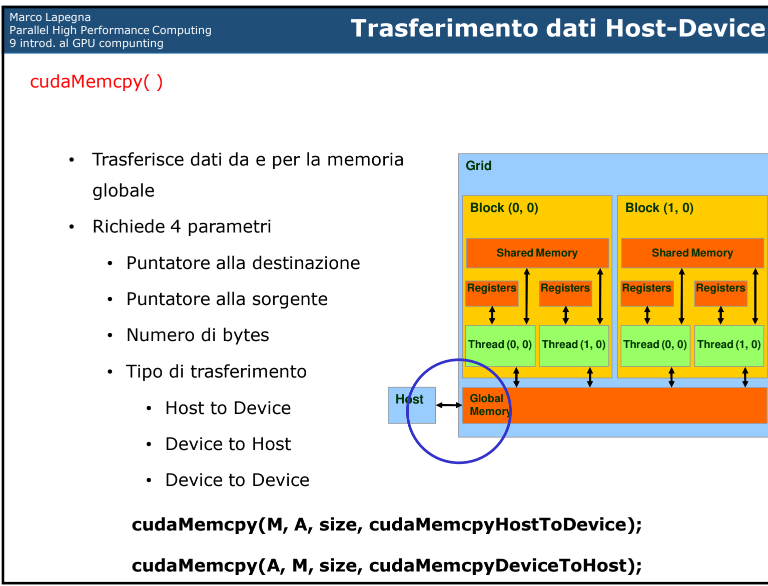
18



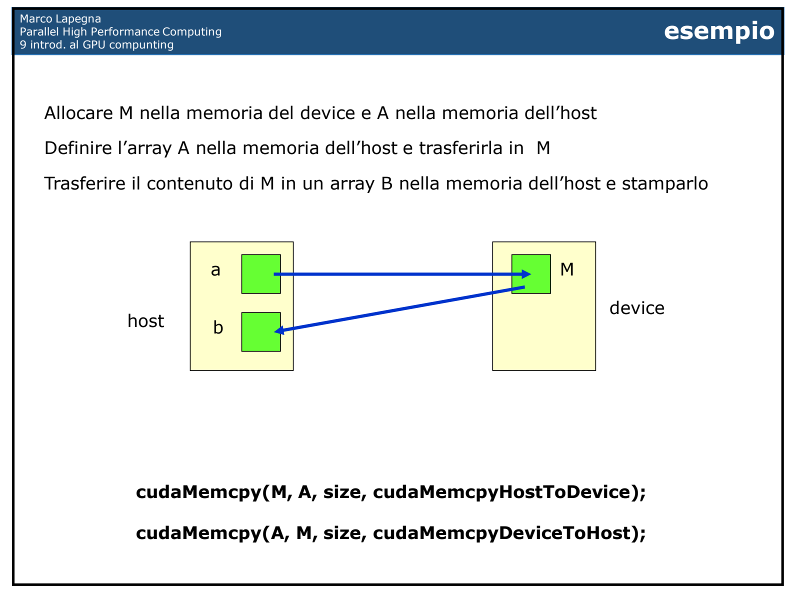
19



20



21



22

Codice integrato

```
#include<stdio.h>
#include<cuda.h>
main(){
float *a, *b, *M;
int size, i;

size=sizeof(float);
a=(float*)malloc(size*5);
b=(float*)malloc(size*5);
cudaMalloc((void**)&M, size*5);

for(i=0; i<5; i++) *(a+i)=i+1;
cudaMemcpy(M, a, size*5, cudaMemcpyHostToDevice);

cudaMemcpy(b, M, size*5, cudaMemcpyDeviceToHost);
for(i=0; i<5; i++) printf("b= %f\n", *(b+i));

cudaFree(M);

}
```

23

dichiarazione funzioni in CUDA

	Eseguita su:	Richiamabile da:
__device__ float DeviceFunc()	device	device
__global__ void KernelFunc()	device	host
__host__ float HostFunc()	host	host

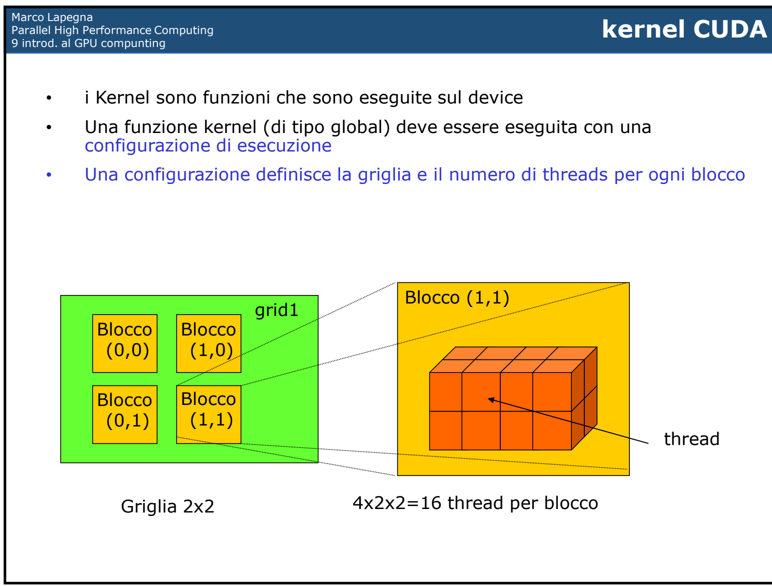
Le funzioni vanno dichiarate specificandone il tipo con opportune estensioni al linguaggio C

__global__ definisce la funzione del kernel
Deve essere di tipo **void**

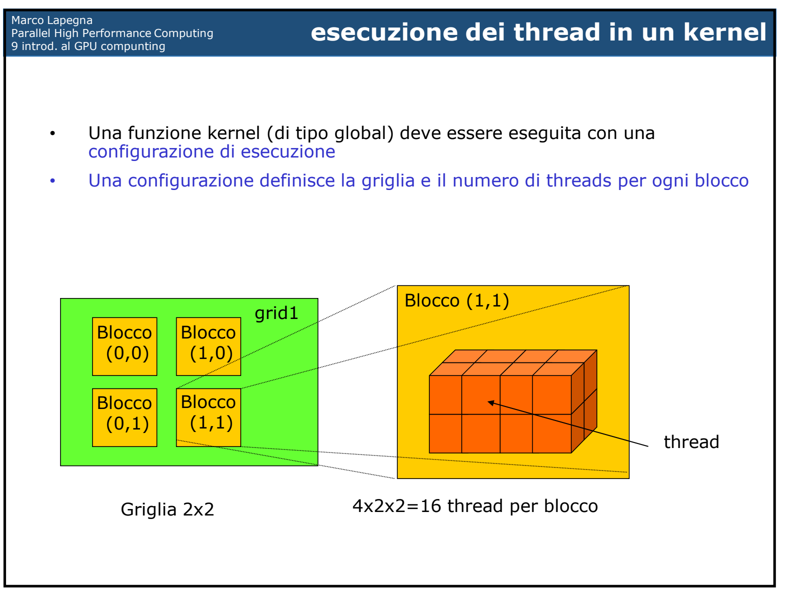
__device__ e **__host__** definiscono le funzioni che devono essere eseguite sul device e sull'host rispettivamente

- Per le funzioni eseguite sul device:
  - No ricorsione
  - No dichiarazione di variabili statiche
  - No numero variabile di argomenti

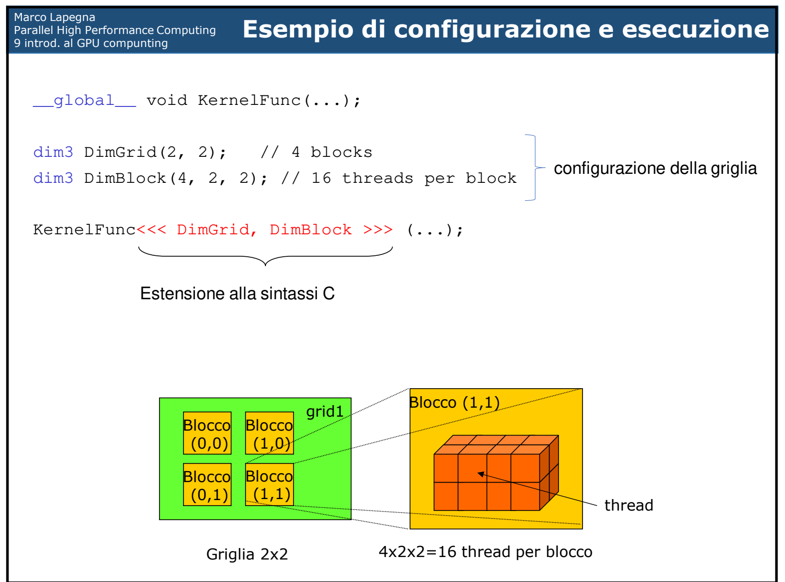
24



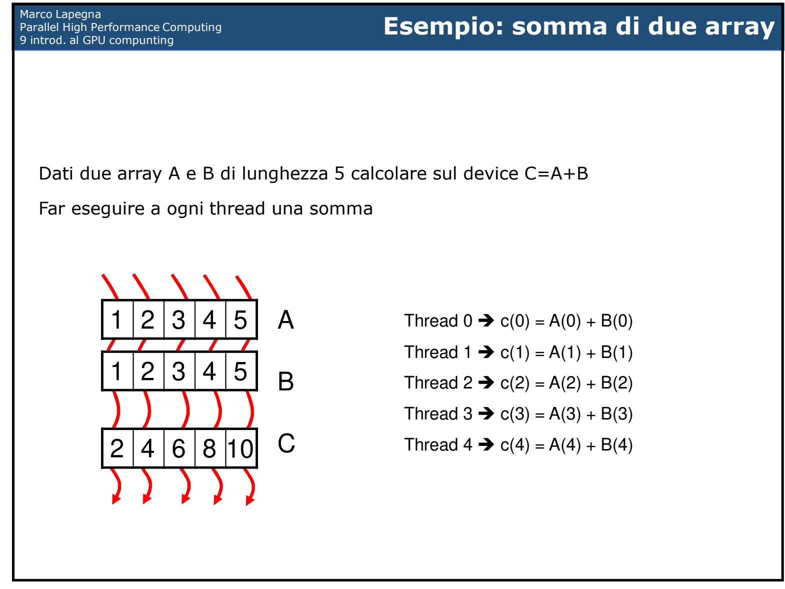
25



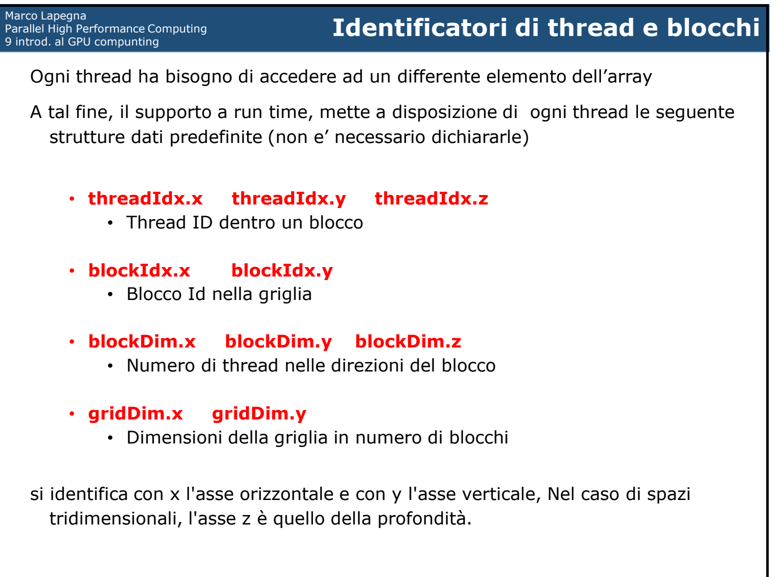
26



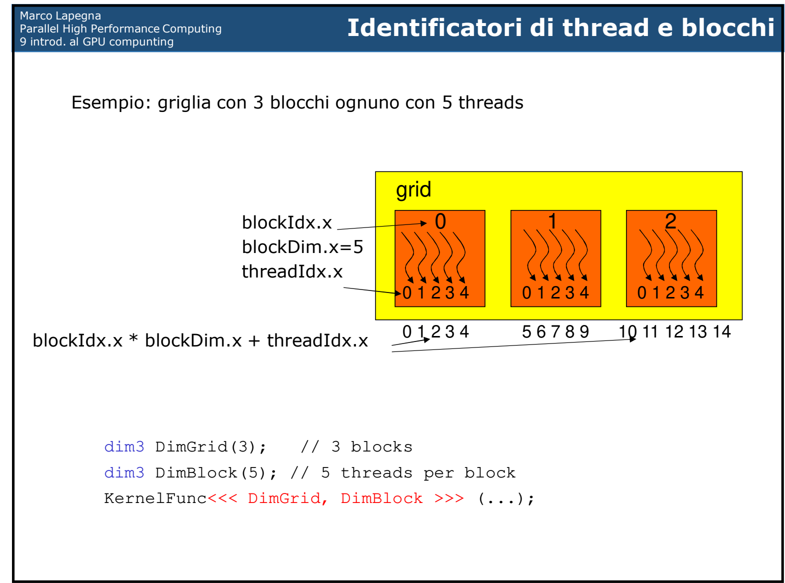
27



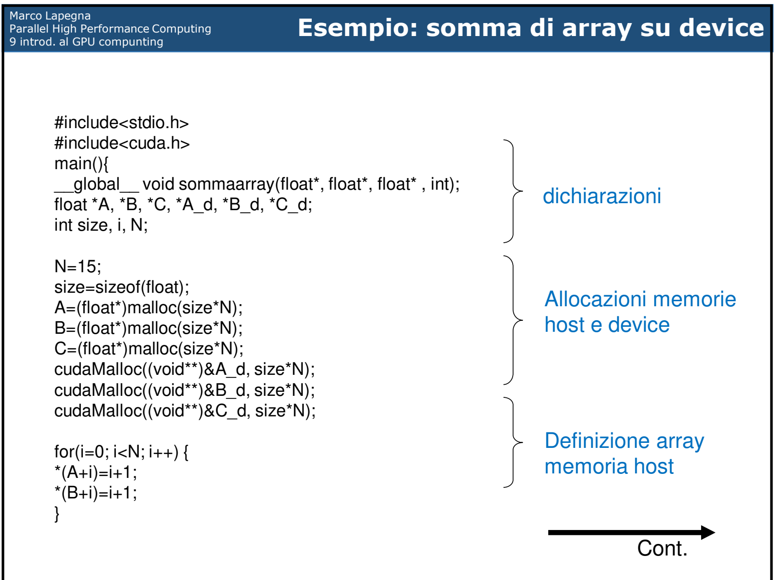
28



29



30



31

Marco Lapegna
Parallel High Performance Computing
9 introd. al GPU computing

Esempio: somma di array su device

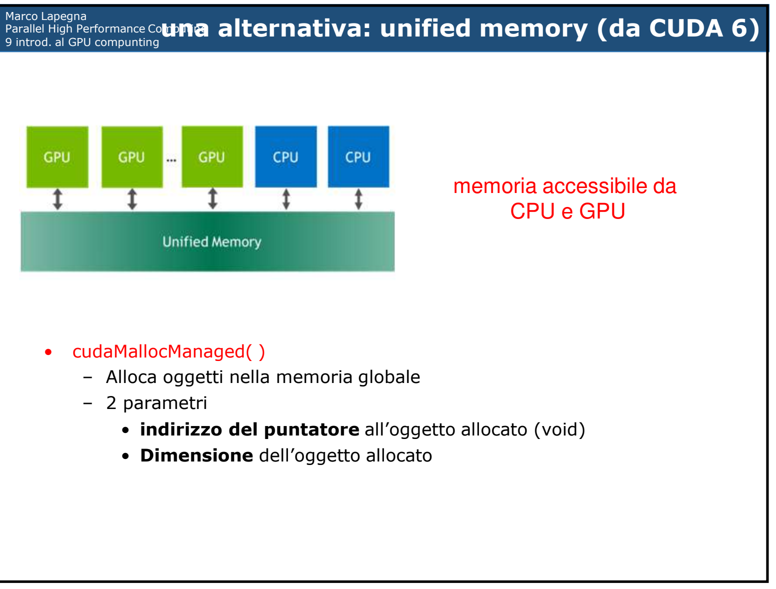
```
cudaMemcpy(A_d, A, size*N, cudaMemcpyHostToDevice);
cudaMemcpy(B_d, B, size*N, cudaMemcpyHostToDevice);
```
Copia in mem. device

```
dim3 DimGrid(3); dim3 DimBlock(5);
sommaarray<<<DimGrid,DimBlock>>>(A_d, B_d, C_d, N);
cudaDeviceSynchronize();
```
Definizione configuraz. Esecuzione thread

```
cudaMemcpy(C, C_d, size*N, cudaMemcpyDeviceToHost);
for(i=0; i<N; i++) printf("C= %f\n", *(C+i));
printf("---------\n");
cudaFree(A_d); cudaFree(B_d); cudaFree(C_d);

}
```
Copia in mem. host e deallocazione mem. device

```
__global__ void sommaarray(float *A, float *B, float *C, int N){
int  idglobal;
idglobal = blockIdx.x * blockDim.x + threadIdx.x ;
if (idglobal < N)
        *(C+idglobal) = *(A+idglobal) + *(B+idglobal);
}
```
Funzione del kernel

32



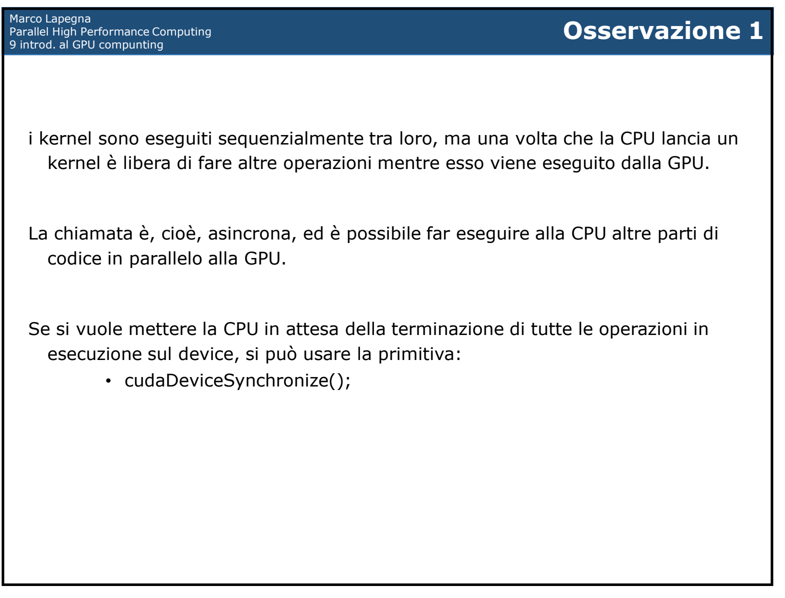
33

Esempio: somma di array su device

```
#include<stdio.h>
main(){

__global__ void sommaarray(float*, float*, float* );
float *A, *B, *C;
int size, i, N;

N=10;
cudaMallocManaged(&A, sizeof(float)*N);
cudaMallocManaged(&B, sizeof(float)*N);
cudaMallocManaged(&C, sizeof(float)*N);

for(i=0; i<N; i++) {
*(A+i)=i+1;  *(B+i)=i+1;
}

dim3 DimGrid(1,1);   dim3 DimBlock(N,1,1);
sommaarray<<<DimGrid,DimBlock>>>(A, B, C);
cudaDeviceSynchronize();

for(i=0; i<N; i++) printf("C= %f\n", *(C+i));
cudaFree(A); cudaFree(B); cudaFree(C);

}

__global__ void sommaarray(float *A,  float *B,  float *C){
*(C+threadIdx.x) = *(A+threadIdx.x) + *(B+threadIdx.x);
}
```

- dichiarazioni
- Allocazioni unified memory
- accesso dall'host
- esecuzione kernel
- accesso dal device

34



35

Osservazione 2

Ogni blocco puo' eseguire al piu' 1024 thread

- Es. DimBlock(1024,1,1) DimBlock(16,16,4) DimBlock (2,2,128) sono consentiti
- Es DimBlock(16,16,8) DimBlock(2,1024,1) non sono consentiti
- Nell'esempio precedente, se N > 1024 e' necessario utilizzare 2 o piu' blocchi

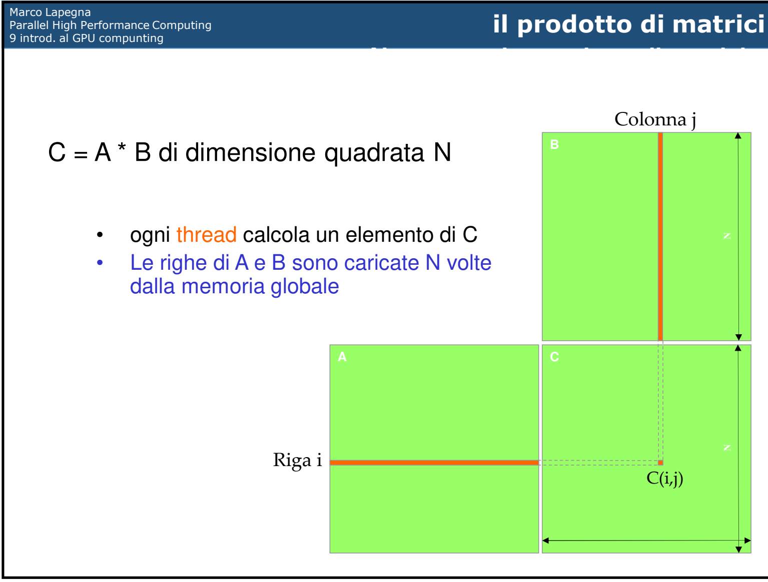
```
dim3 DimGrid(2,1); dim3 DimBlock(1024,1,1);
sommaarray<<<DimGrid,DimBlock>>>(A_d,B_d,C_d);
```

Host definisce 2 blocchi di 1024 thread

```
__global__ void sommaarray(float *A, float *B, float *C){
int idx;
idx = blockIdx.x*blockDim.x + threadIdx.x;
*(C+idx) = *(A+idx) + *(B+idx);
}
```

Codice device

36



37

Marco Lapegna
Parallel High Performance Computing
9 introd. al GPU compunting

memorizzazione delle matrici

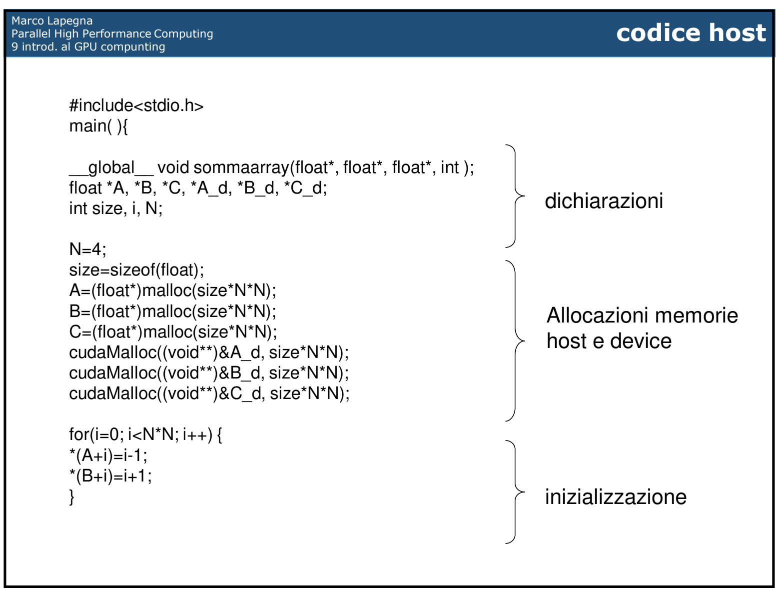
$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

Memorizzazione per righe

M

$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$ $M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$

38



39

Marco Lapegna
Parallel High Performance Computing
9 introd. al GPU compunting

continua

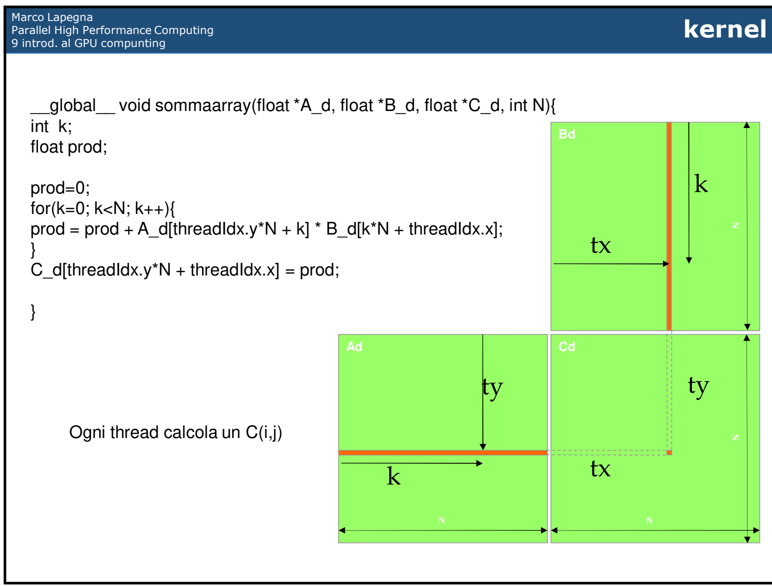
```
cudaMemcpy(A_d, A, size*N*N, cudaMemcpyHostToDevice);
cudaMemcpy(B_d, B, size*N*N, cudaMemcpyHostToDevice);

dim3 DimGrid(1,1); dim3 DimBlock(N,N,1);
sommaarray<<<DimGrid,DimBlock>>>(A_d, B_d, C_d, N);

cudaMemcpy(C, C_d, size*N*N, cudaMemcpyDeviceToHost);
for(i=0; i<N*N; i++) printf("C= %f\n", *(C+i));
printf("---------\n");

cudaFree(A_d); cudaFree(B_d); cudaFree(C_d);

}
```

Copia in mem. device

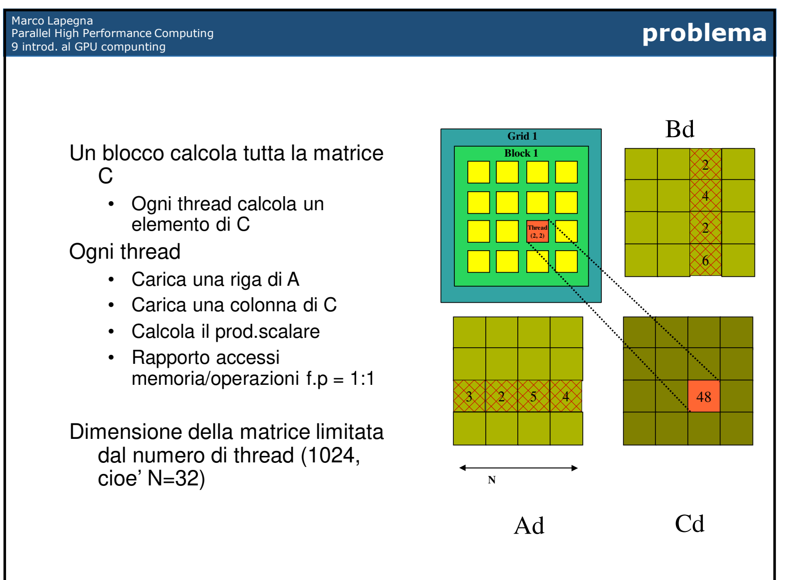
Definizione configuraz. Esecuzione thread

Copia in mem. host e deallocazione mem. device

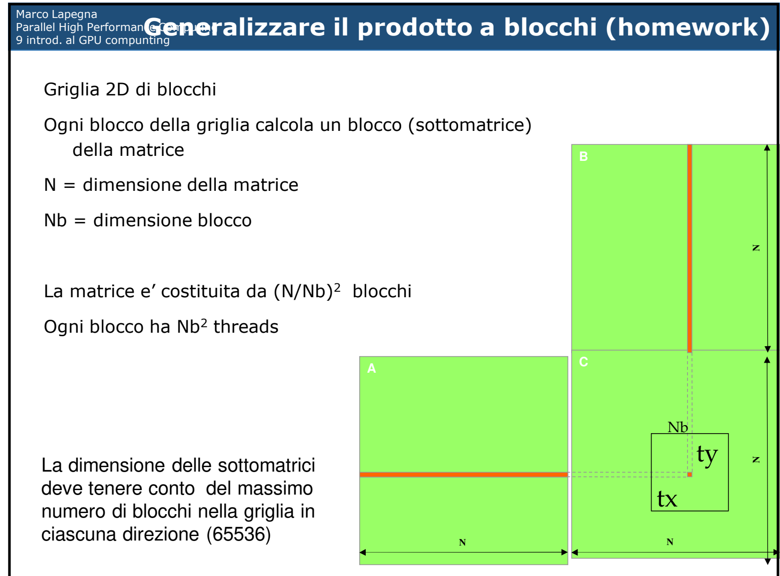
40



41



42



43