



**PARALLEL HIGH  
PERFORMANCE COMPUTING**  
CdS magistrale in informatica

**9 – introduzione al GPU computing**  
Dipartimento di Matematica e Applicazioni  
Universita' degli Studi di Napoli Federico II

[wpage.unina.it/lapegna](http://wpage.unina.it/lapegna)

1

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

## 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](http://www.gpgpu.org)



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

2

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

## 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

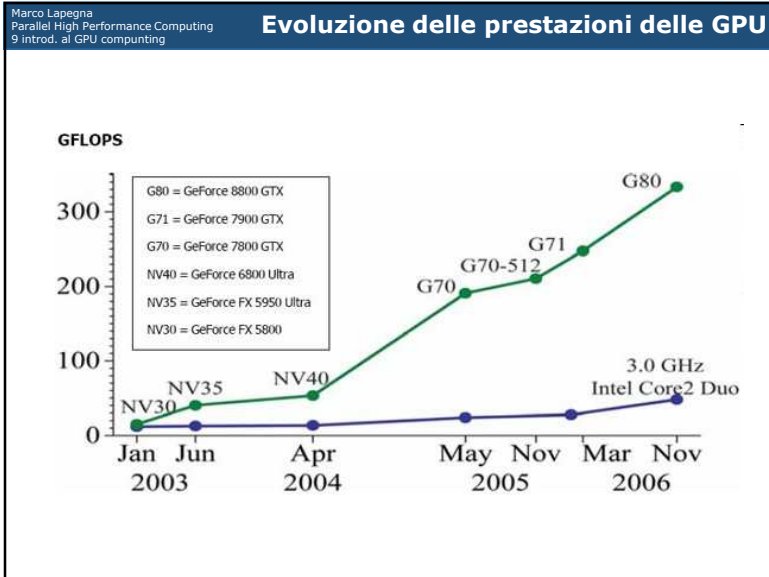
Marco Lapegna  
Parallel High Performance Computing  
9 introd. al GPU computing

## 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 istruzioni macchina
- Comunicazione limitata tra eventuali tasks paralleli

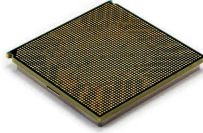
4



5


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

### nel 2020



**Intel Xeon E5-2699 v4**

- 22 core
- 528 Gflops



**Nvidia Volta Gv100**

- 5120 core
- 15 Tflops

6


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

### Come e' fatta una GPU?

Le GPU sono nate come processori specializzate per la grafica 3D e si sono sviluppate sotto la spinta delle esigenze dei videogames

La grande diffusione ha permesso una drastica riduzione dei costi

Esempio: un cubo in movimento



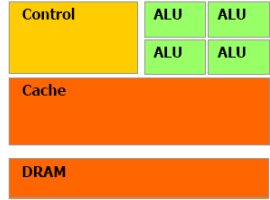
Stesse operazioni su tutti i vertici del cubo

Molti dati indipendenti da elaborare

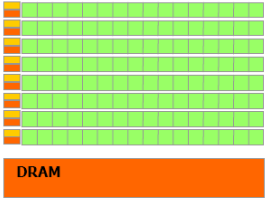
7

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

### Differenze CPU vs GPU



**CPU**



**GPU**

- Molto spazio sul chip dedicato al controllo e alla cache
- Poche unita' processanti potenti e sofisticate
- Adatto a database, algoritmi ricorsivi, flussi di controllo non regolari


- Miglior rapporto Gflops/costo e miglior rapporto Gflops/consumo
- Molte unita' processanti elementari e sofisticate
- Adatto a calcoli ripetitivi su grandi quantita' di dati
- La presenza di una memoria sulla scheda evita il collo di bottiglia del bus di sistema

8

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

## La GPU Volta GV100 della Nvidia

# of Streaming Processor Cores	5120
Frequency of processor cores	1.530 GHz
Single Precision floating point performance (peak)	15.7 Tflops
Double Precision floating point performance (peak)	7.8 Tflops
Floating Point Precision	IEEE 754
Total Dedicated Memory	32 GB GDDR3
Memory Speed	800MHz
Memory Bandwidth	900 GB/sec
Software Development Tools	C-based CUDA Toolkit




Ricavata da una scheda grafica eliminando i connettori output

9

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

## CUDA

"Compute Unified Device Architecture"



Basato su un modello di programmazione General purpose

- Gli utenti lanciano gruppi di threads sulle GPU
- GPU = coprocessore dedicato a operazioni data parallel mediante threads

Ambiente software generale

- Orientato al calcolo con librerie e compilatore


Comandi espliciti (API) per

- Caricare programmi nelle GPU
- Migrazione dati tra memoria centrale e locale
- Ottimizzazione del calcolo "data parallel"
- Nessun riferimento ad "oggetti grafici"

10

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

## CUDA



Hardware e software disegnati insieme (modello di programmazione integrato)

Software

- Estensione del linguaggio C
- Facile da imparare

Hardware

- Shared memory
- Cooperazione tra thread

11

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

## Overview del seguito

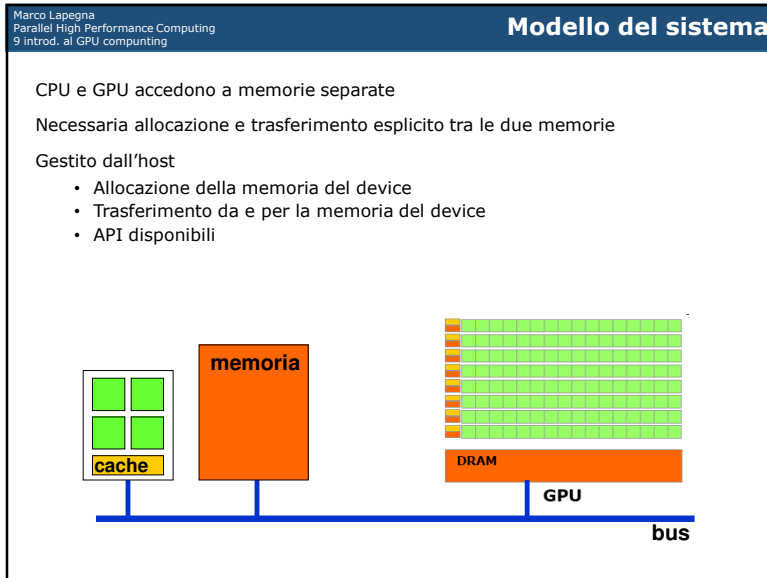
Modello di programmazione CUDA - concetti di base e tipi di dati

Application Programming Interface CUDA

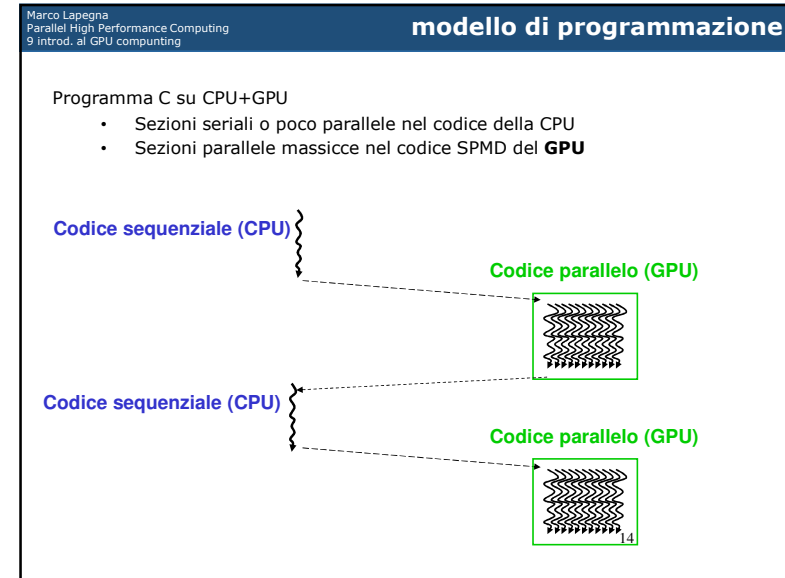
Primi esempi per illustrare concetti di base e funzionalita'

Performance

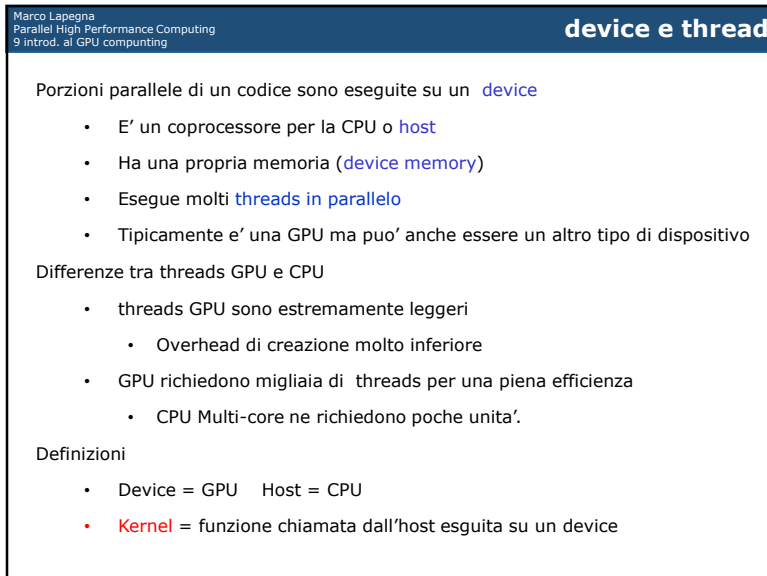
12



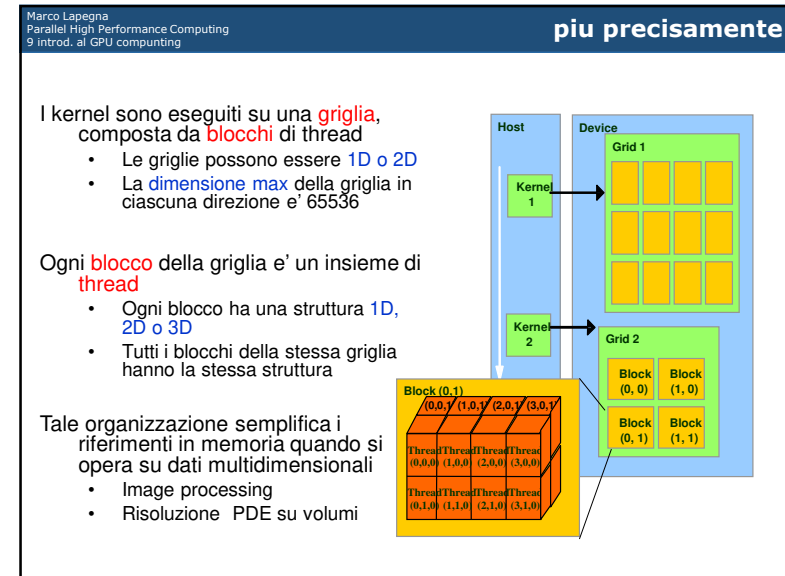
13



14



15



16

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

### modello di memoria

Memoria globale

- Principale mezzo di comunicazione tra **host** e **device**
- Contenuto visibile a tutti i thread
- Elevati tempi di accesso
- Altre memorie disponibili piu' veloci
- Per ora consideriamo solo la **memoria globale**

17

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

### allocazione della memoria

`cudaMalloc()`

- Alloca oggetti nella memoria globale
- 2 parametri
  - **indirizzo** dell'oggetto allocato (`void`)
  - **Dimensione** dell'oggetto allocato

`cudaFree()`

- Libera la memoria globale dall'oggetto allocato con `cudaMalloc()`
  - Puntatore all'oggetto da rimuovere

18

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

### Applicazione integrata host/device

Unico sorgente e unico eseguibile  
Compilato con nvcc

19

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

### allocazione della memoria

esempio:

- Allocare un array float di 5 elementi
- Indirizzare tale area con M
- Liberare l'area precedentemente allocata

```
float* M;
int size = sizeof(float) * 5;
cudaMalloc((void**)&M, size);
cudaFree(M);
```

M e' una variabile allocata nella host memory che indirizza la device memory

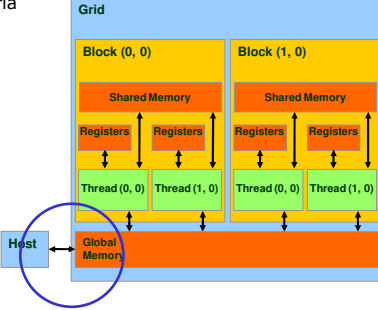
20

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

## Trasferimento dati Host-Device

`cudaMemcpy()`

- Trasferisce dati da e per la memoria globale
- Richiede 4 parametri
  - Puntatore alla destinazione
  - Puntatore alla sorgente
  - Numero di bytes
  - Tipo di trasferimento
    - Host to Device
    - Device to Host
    - Device to Device



The diagram illustrates the GPU architecture. At the top is the Host. Below it is the Global Memory. A Grid contains two Blocks: Block (0, 0) and Block (1, 0). Each Block contains Shared Memory, Registers, and Threads. The Host is connected to the Global Memory, which is connected to the Grid. A blue circle highlights the Host and Global Memory connection.

```

cudaMemcpy(M, A, size, cudaMemcpyHostToDevice);
cudaMemcpy(A, M, size, cudaMemcpyDeviceToHost);

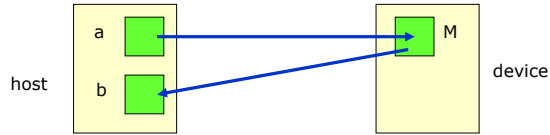
```

21

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

## esempio

Allocare M nella memoria del device e A nella memoria dell'host  
Definire l'array A nella memoria dell'host e trasferirla in M  
Trasferire il contenuto di M in un array B nella memoria dell'host e stamparlo



The diagram shows a host memory area with two green boxes labeled 'a' and 'b'. A device memory area has a green box labeled 'M'. Blue arrows indicate data transfer: one arrow from 'a' to 'M', and another from 'M' to 'b'.

```

cudaMemcpy(M, A, size, cudaMemcpyHostToDevice);
cudaMemcpy(A, M, size, cudaMemcpyDeviceToHost);

```

22

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

## 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

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

## dichiarazione funzioni in CUDA

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

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

### kernel CUDA

- i Kernel sono funzioni che sono eseguite sul device
- Una funzione kernel (di tipo global) deve essere eseguita con una **configurazione di esecuzione**
- Una configurazione definisce la griglia e il numero di threads per ogni blocco

Griglia 2x2      4x2x2=16 thread per blocco

25

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

### esecuzione dei thread in un kernel

- Una funzione kernel (di tipo global) deve essere eseguita con una **configurazione di esecuzione**
- Una configurazione definisce la griglia e il numero di threads per ogni blocco

Griglia 2x2      4x2x2=16 thread per blocco

26

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

### Esempio di configurazione e esecuzione

```

__global__ void KernelFunc(...);

dim3 DimGrid(2, 2); // 4 blocks
dim3 DimBlock(4, 2, 2); // 16 threads per block } configurazione della griglia

KernelFunc<<< DimGrid, DimBlock >>> (...);
    
```

Estensione alla sintassi C

Griglia 2x2      4x2x2=16 thread per blocco

27

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

### Esempio: somma di due array

Dati due array A e B di lunghezza 5 calcolare sul device  $C=A+B$   
Far eseguire a ogni thread una somma

1	2	3	4	5	A	Thread 0 → $c(0) = A(0) + B(0)$
1	2	3	4	5	B	Thread 1 → $c(1) = A(1) + B(1)$
2	4	6	8	10	C	Thread 2 → $c(2) = A(2) + B(2)$
						Thread 3 → $c(3) = A(3) + B(3)$
						Thread 4 → $c(4) = A(4) + B(4)$

28

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

## Identificatori di thread e blocchi

Ogni thread ha bisogno di accedere ad un differente elemento dell'array  
A tal fine, il supporto a run time, mette a disposizione di ogni thread le seguente strutture dati predefinite (non e' necessario dichiararle)

- **threadIdx.x threadIdx.y threadIdx.z**
  - Thread ID dentro un blocco
- **blockIdx.x blockIdx.y**
  - Blocco Id nella griglia
- **blockDim.x blockDim.y blockDim.z**
  - Numero di thread nelle direzioni del blocco
- **gridDim.x gridDim.y**
  - Dimensioni della griglia in numero di blocchi

si identifica con x l'asse orizzontale e con y l'asse verticale, Nel caso di spazi tridimensionali, l'asse z è quello della profondità.

29

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

## Identificatori di thread e blocchi

Esempio: griglia con 3 blocchi ognuno con 5 threads

blockIdx.x \* blockDim.x + threadIdx.x

```

dim3 DimGrid(3); // 3 blocks
dim3 DimBlock(5); // 5 threads per block
KernelFunc<<< DimGrid, DimBlock >>> (...);
    
```

30

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

## Esempio: somma di array su device

```

#include<stdio.h>
#include<cuda.h>
main(){
__global__ void sommaarray(float*, float*, float*, int);
float *A, *B, *C, *A_d, *B_d, *C_d;
int size, i, N;

N=15;
size=sizeof(float);
A=(float*)malloc(size*N);
B=(float*)malloc(size*N);
C=(float*)malloc(size*N);
cudaMalloc((void**)&A_d, size*N);
cudaMalloc((void**)&B_d, size*N);
cudaMalloc((void**)&C_d, size*N);

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

dichiarazioni

Allocazioni memorie host e device

Definizione array memoria host

Cont.

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);

dim3 DimGrid(3); dim3 DimBlock(5);
sommaarray<<<DimGrid,DimBlock>>>(A_d, B_d, C_d, N);
cudaDeviceSynchronize();

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);

}

__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);
}
    
```

Copia in mem. device

Definizione configuraz. Esecuzione thread

Copia in mem. host e deallocazione mem. device

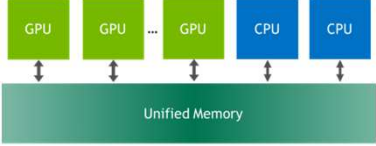
Funzione del kernel

32



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

## una alternativa: unified memory (da CUDA 6)



memoria accessibile da CPU e GPU

- `cudaMallocManaged( )`
  - Alloca oggetti nella memoria globale
  - 2 parametri
    - **indirizzo del puntatore** all'oggetto allocato (void)
    - **Dimensione** dell'oggetto allocato

33

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

## 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= %fn", *(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

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

## Osservazione 1

i kernel sono eseguiti sequenzialmente tra loro, ma una volta che la CPU lancia un kernel è libera di fare altre operazioni mentre esso viene eseguito dalla GPU.

La chiamata è, cioè, asincrona, ed è possibile far eseguire alla CPU altre parti di codice in parallelo alla GPU.

Se si vuole mettere la CPU in attesa della terminazione di tutte le operazioni in esecuzione sul device, si può usare la primitiva:

- `cudaDeviceSynchronize();`

35

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

## 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);

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

Host definisce 2 blocchi di 1024 thread

Codice device

36

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

### il prodotto di matrici

$C = A * B$  di dimensione quadrata  $N$

- ogni **thread** calcola un elemento di  $C$
- Le righe di  $A$  e  $B$  sono caricate  $N$  volte dalla memoria globale

37

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

### memorizzazione delle matrici

Memorizzazione per righe

38

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

### codice host

```

#include<stdio.h>
main() {

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

    N=4;
    size=sizeof(float);
    A=(float*)malloc(size*N*N);
    B=(float*)malloc(size*N*N);
    C=(float*)malloc(size*N*N);
    cudaMalloc((void**)&A_d, size*N*N);
    cudaMalloc((void**)&B_d, size*N*N);
    cudaMalloc((void**)&C_d, size*N*N);

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

dichiarazioni

Allocazioni memorie host e device

inizializzazione

39

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

### 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

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

### kernel

```

__global__ void sommaarray(float *A_d, float *B_d, float *C_d, int N){
int k;
float prod;

prod=0;
for(k=0; k<N; k++){
prod = prod + A_d[threadIdx.y*N + k] * B_d[k*N + threadIdx.x];
}
C_d[threadIdx.y*N + threadIdx.x] = prod;
}
    
```

Ogni thread calcola un C(i,j)

41

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

### problema

Un blocco calcola tutta la matrice C

- Ogni thread calcola un elemento di C

Ogni thread

- Carica una riga di A
- Carica una colonna di C
- Calcola il prod. scalare
- Rapporto accessi memoria/operazioni f.p = 1:1

Dimensione della matrice limitata dal numero di thread (1024, cioè N=32)

42

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

### Generalizzare il prodotto a blocchi (homework)

Griglia 2D di blocchi

Ogni blocco della griglia calcola un blocco (sottomatrice) della matrice

N = dimensione della matrice

Nb = dimensione blocco

La matrice e' costituita da  $(N/Nb)^2$  blocchi

Ogni blocco ha  $Nb^2$  threads

La dimensione delle sottomatrici deve tenere conto del massimo numero di blocchi nella griglia in ciascuna direzione (65536)

43