Il modello di programmazione di CUDA

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,…

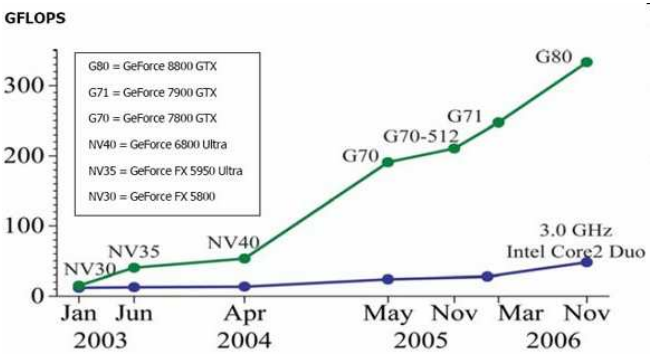
Storia delle GPU

- 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”

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

Evoluzione delle prestazioni delle GPU

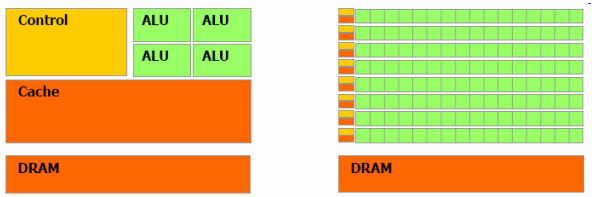


Come e' fatta una GPU?

- 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

Differenze CPU vs 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
- Adatto a calcoli ripetitivi su grandi quantita' di dati
- La presenza di una memoria sulla scheda evita il collo di pottiglia del bus di sistema

La GPU Tesla C1060 della Nvidia

# of Streaming Processor Cores	240
Frequency of processor cores	1.3 GHz
Single Precision floating point performance (peak)	933 Gflops
Double Precision floating point performance (peak)	78 Gflops
Floating Point Precision	IEEE 754
Total Dedicated Memory	4 GDDR3
Memory Speed	800MHz
Memory Bandwidth	102 GB/sec
Software Development Tools	C-based CUDA Toolkit

Ricavata da una scheda grafica eliminando i connettori output

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"

CUDA

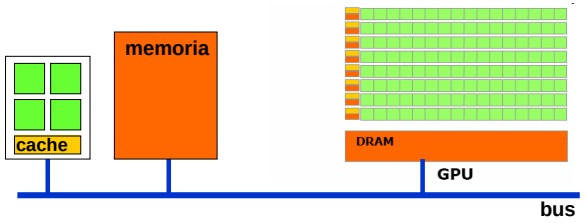
- Hardware e software disegnati insieme (modello di programmazione integrato)
- Software
  - Estensione del linguaggio C
  - Facile da imparare
- Hardware
  - Shared memory
  - Cooperazione tra thread

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

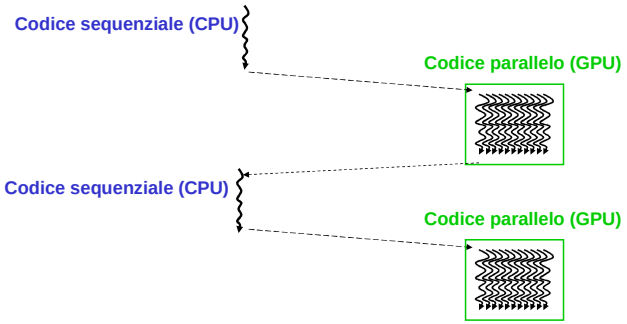
Modello del sistema

- CPU e GPU accedono a memorie separate
- Necessaria allocazione e trasferimento esplicito tra le due memorie
- Gestito dall'host
  - Allocazione della memoria del device
  - Trasferimento da e per la memoria del device
  - API disponibili



Modello di programmazione

- Programma C su CPU+GPU
  - Sezioni seriali o poco parallele nel codice della CPU
  - Sezioni parallele massicce nel codice SPMD del **GPU**

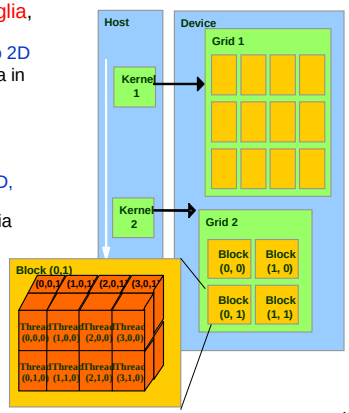


Devices e Threads CUDA

- Porzioni parallele di un codice sono eseguite su un device
  - E' un coprocessore per la CPU o host
  - Ha una propria memoria (device memory)
  - Esegue molti threads in parallelo
  - Tipicamente e' una GPU ma puo' anche essere un altro tipo di dispositivo
- Differenze tra threads GPU e CPU
  - threads GPU sono estremamente leggeri
    - Overhead di creazione molto inferiore
  - GPU richiedono migliaia di threads per una piena efficienza
    - CPU Multi-core ne richiedono poche unita'.
- Definizioni
  - Device = GPU Host = CPU
  - Kernel = funzione chiamata dall'host esguita su un device

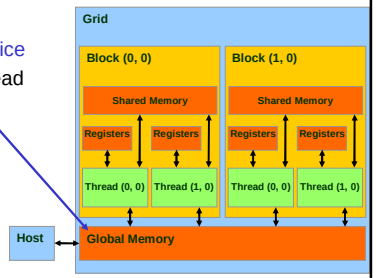
Piu' precisamente

- I kernel sono eseguiti su una griglia, composta da blocchi di thread
  - Le griglie possono essere 1D o 2D
  - La dimensione max della griglia in ciascuna direzione e' 65536
- Ogni blocco della griglia e' un insieme di thread
  - Ogni blocco ha una struttura 1D, 2D o 3D
  - Tutti i blocchi della stessa griglia hanno la stessa struttura
- Tale organizzazione semplifica i riferimenti in memoria quando si opera su dati multidimensionali
  - Image processing
  - Risoluzione PDE su volumi



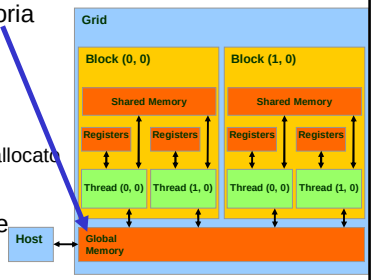
Modello della memoria

- 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

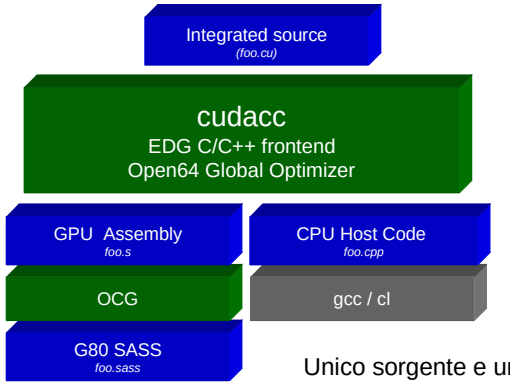


Allocazione della memoria CUDA

- cudaMalloc( )
  - Alloca oggetti nella memoria globale
  - 2 parametri
    - **indirizzo del puntatore** all'oggetto allocato (void)
    - **Dimensione** dell'oggetto allocato
- cudaFree( )
  - Libera la memoria globale dall'oggetto allocato con cudaMalloc( )
    - Puntatore all'oggetto da rimuovere



Applicazione integrata host/device



Unico sorgente e unico eseguibile
Compilato con nvcc

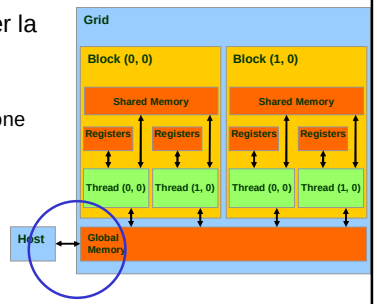
Allocazione della memoria CUDA

- 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);
```


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



CUDA Host-Device Data Transfer

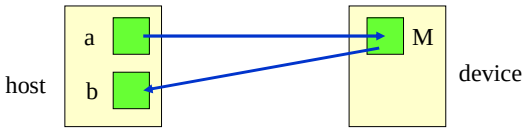
- esempio:
  - Trasferire un array float 5
  - A e' nella memoria host e M e' nella memoria device
  - cudaMemcpyHostToDevice e cudaMemcpyDeviceToHost sono costanti simboliche

cudaMemcpy(M, A, size, cudaMemcpyHostToDevice);

cudaMemcpy(A, M, size, cudaMemcpyDeviceToHost);

Mettiamo tutto insieme

- Allocare M nella memoria nella memoria del device
- Definire un array a nella memoria dell'host e trasferirla nella memoria del device
- Trasferire il contenuto della memoria del device in un array b nella memoria dell'host e stamparlo



Codice integrato

```
#include<stdio.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);

}
```

Dichiarazione delle funzioni 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 estensione al lingiaggio C**
- `__global__` definisce la funzione del kernel
  - Deve essere di tipo `void`

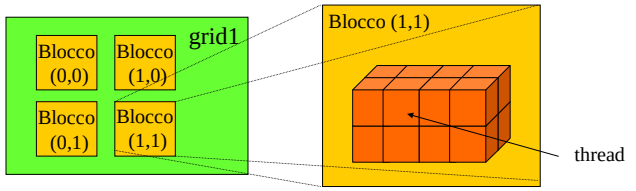
CUDA Function Declarations (cont.)

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

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

Esecuzione dei 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

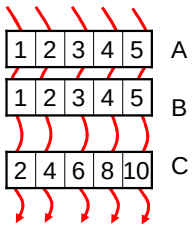
Esmpio di configurazione e esecuzione

```
__global__ void KernelFunc(...);

dim3 DimGrid(2, 2);   // 4 blocks
dim3 DimBlock(4, 2, 2); // 16 threads per block

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

Estensione alla sintassi C

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



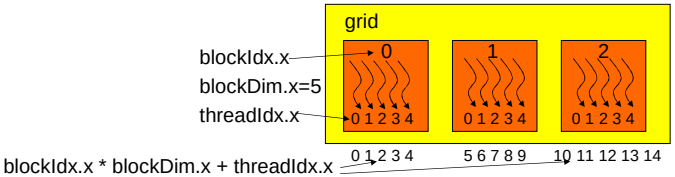
Thread 0 ➔ c(0)=A(0)+B(0)
Thread 1 ➔ c(1)=A(1)+B(1)
Thread 2 ➔ c(2)=A(2)+B(2)
Thread 3 ➔ c(3)=A(3)+B(3)
Thread 4 ➔ c(4)=A(4)+B(4)

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.

Identificatori di thread e blocchi

Esempio: griglia con 3 blocchi ognuno con 5 threads



Esempio: somma di array su device

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

N=10;
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.

Esempio: somma di array su device

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

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

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){
*(C+threadIdx.x) = *(A+threadIdx.x) + *(B+threadIdx.x);
}
```

Copia in mem. device

Definizione configuraz. Esecuzione thread

Copia in mem. host e deallocazione mem. device

Funzione del kernel

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:
  - cudaThreadSynchronize();

Osservazione 2

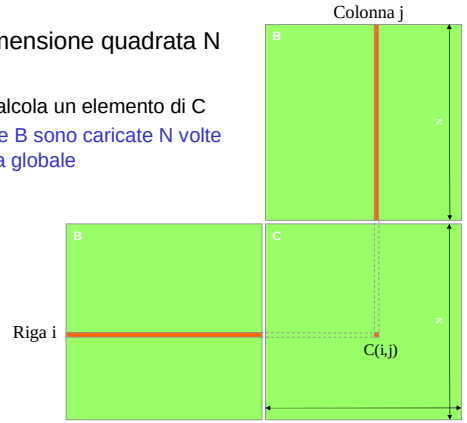
- Ogni blocco puo' eseguire al piu' 512 thread
  - Es. DimBlock(512,1,1) DimBlock(16,16,2) DimBlock (1,2,128) sono consentiti
  - Es DimBlock(8,8,8,) DimBlock(16,16,4) DimBlock(1,1024,1) non sono consentiti
  - Nell'esempio precedente N > 512 e' necessario utilizzare 2 o piu' blocchi

```
dim3 DimGrid(2,1); dim3 DimBlock(512,1,1);
sommaarray<<<DimGrid,DimBlock>>>(A_d,B_d,C_d);
```
Host definisce 2 blocchi di 512 thread

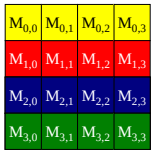
```
__global__ void sommaarray(float *A, float *B, float *C){
int idx;
idx = blockIdx.x*blockDim.x + threadIdx.x;
*(C+idx) = *(A+idx) * *(B+idx);
}
```
Codice device

Altro esempio: prodotto di matrici

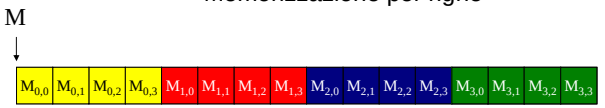
- C = A * B di dimensione quadrata N
  - ogni thread calcola un elemento di C
  - Le righe di A e B sono caricate N volte dalla memoria globale



Memorizzazione delle matrici in C



Memorizzazione per righe



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=3;
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

Codice host (cont.)

```
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

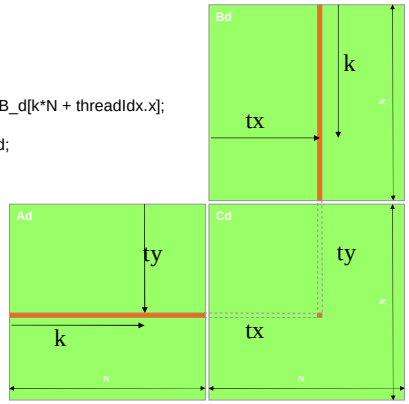
Definizione configuraz. Esecuzione thread

Copia in mem. host e deallocazione mem. device

Codice del kernel

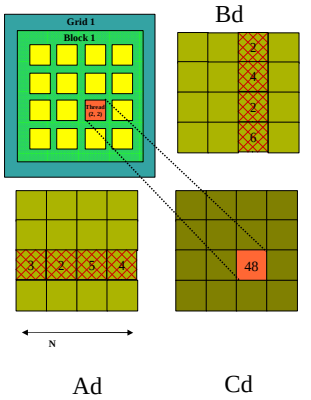
```
__global__ void sommaarray(float *A_d, float *B_d, float *C_d, int N){
int  k;
float prod;

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)



Utilizzato solo un blocco di thread

- 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 (512, cioe' N=16)



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)