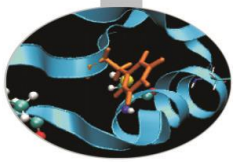
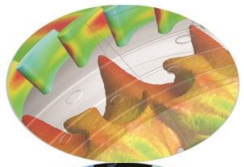
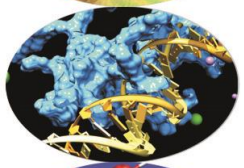
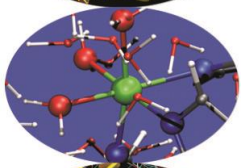
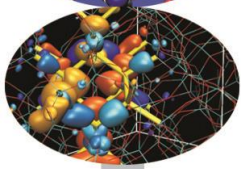
CINECA SCAI
SuperComputing Applications and Innovation



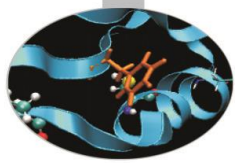
GPU (Graphics Processing Unit) Programming in CUDA

Giorno 1

CINECA

Agenda



I° giorno:

- Introduzione a CUDA
- Utilizzo del compilatore nvcc
- Laboratorio

--- pausa pranzo ---

- Introduzione all'hardware GPU
- Controllo errore
- Misura delle prestazioni
- Laboratorio

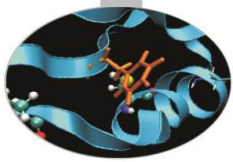
II° giorno:

- Le aree di memoria
- Tipologie di accesso alla memoria
- Laboratorio

--- pausa pranzo ---

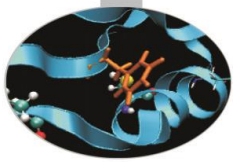
- Stream e Multi-GPU
- Utilizzo di librerie CUDA enabled
- Laboratorio

Acknowledgements



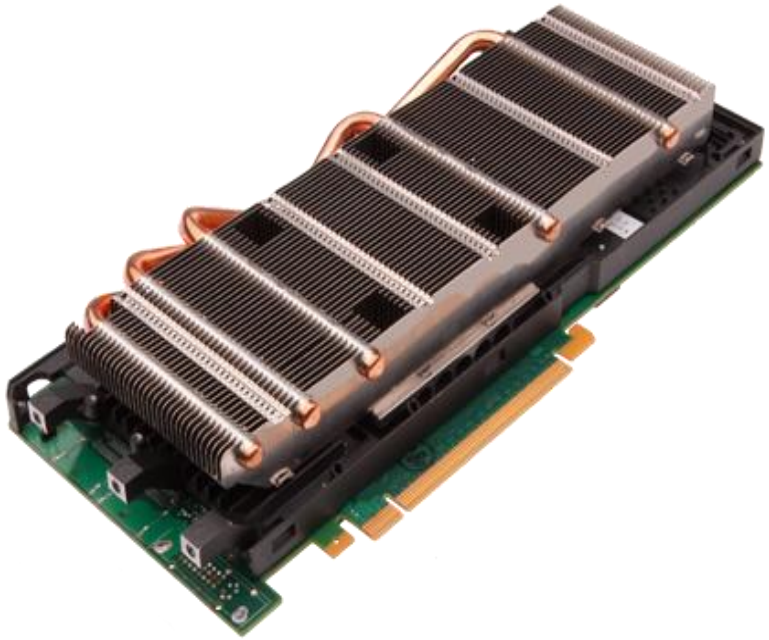
- Docenti di questa edizione:
  - Paride Dagna (p.dagna@cineca.it)
  - Alice Invernizzi (a.invernizzi@cineca.it)

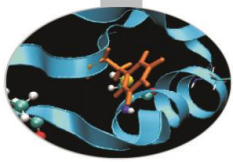
  - Gran parte del materiale e delle immagini sono tratte dalle guide di NVIDIA su CUDA, dall' SDK NVIDIA e dalle presentazioni di Paulius Micikevicius e Vasily Volkov a GTC2010



Cosa è una GPU

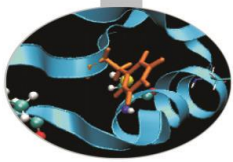
- **G**raphics **P**rocessing **U**nit
  - microprocessore altamente parallelo (*many-core*) dotato di memoria privata ad alta banda
- specializzate per le operazioni di rendering grafico 3D
- sviluppatesi in risposta alla crescente domanda di potenza grafica nel mercato dei videogiochi (grafica 3D ad alta definizione in tempo reale)





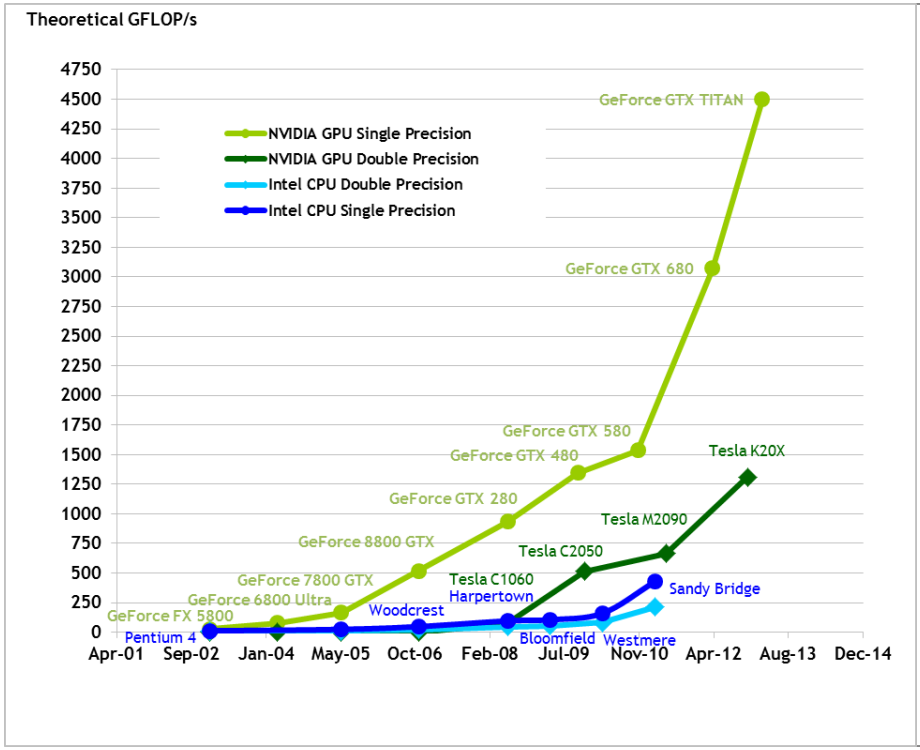
Storia del GPU Computing

- Evoluzione dei chip grafici
  - L'era delle pipeline grafiche a funzioni-fisse (inizi 1980-fine 1990)
  - L'era della grafica programmabile in tempo reale: GPU (2000-2003)
  - L'era dei processori grafici di elaborazione: GPGPU (2003-2006)
  - L'era del GPU Computing (2006)

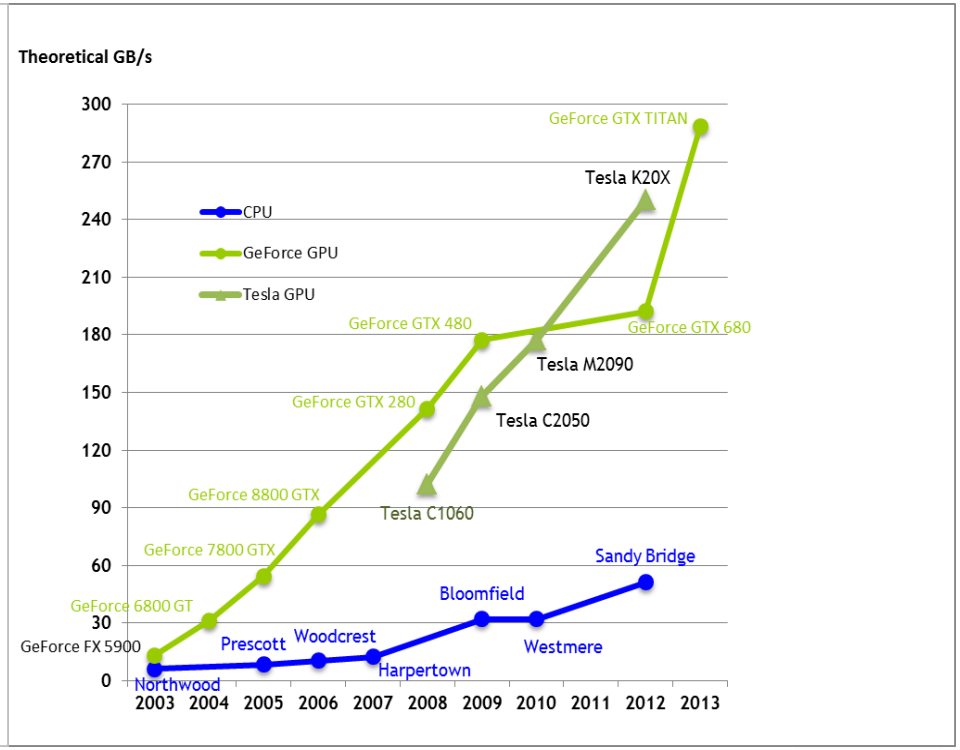


The concurrency revolution

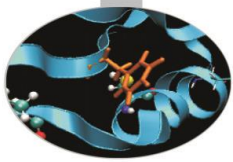
- Una nuova direzione di sviluppo per l'architettura dei microprocessori:
  - incrementare la potenza di calcolo complessiva tramite l'aumento del numero di unità di elaborazione piuttosto che della loro potenza



Numero di operazioni in virgola mobile al secondo per la CPU e la GPU

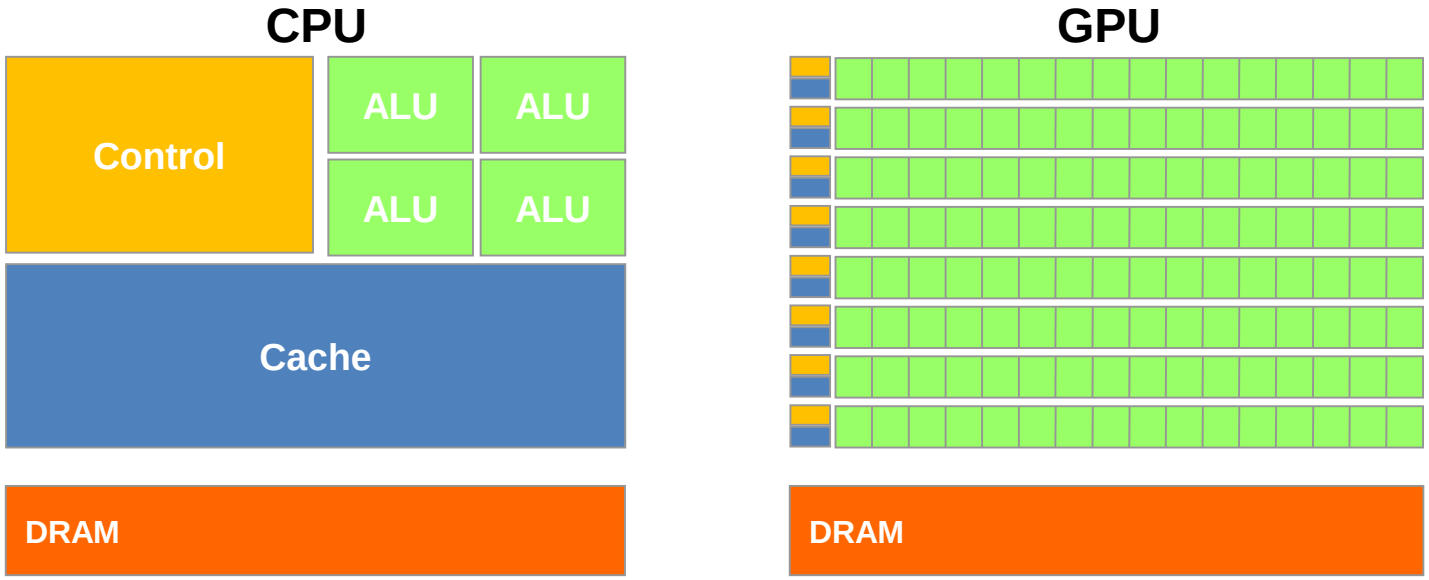


Larghezza di banda della memoria

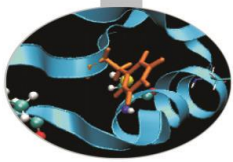


Architettura CPU vs GPU

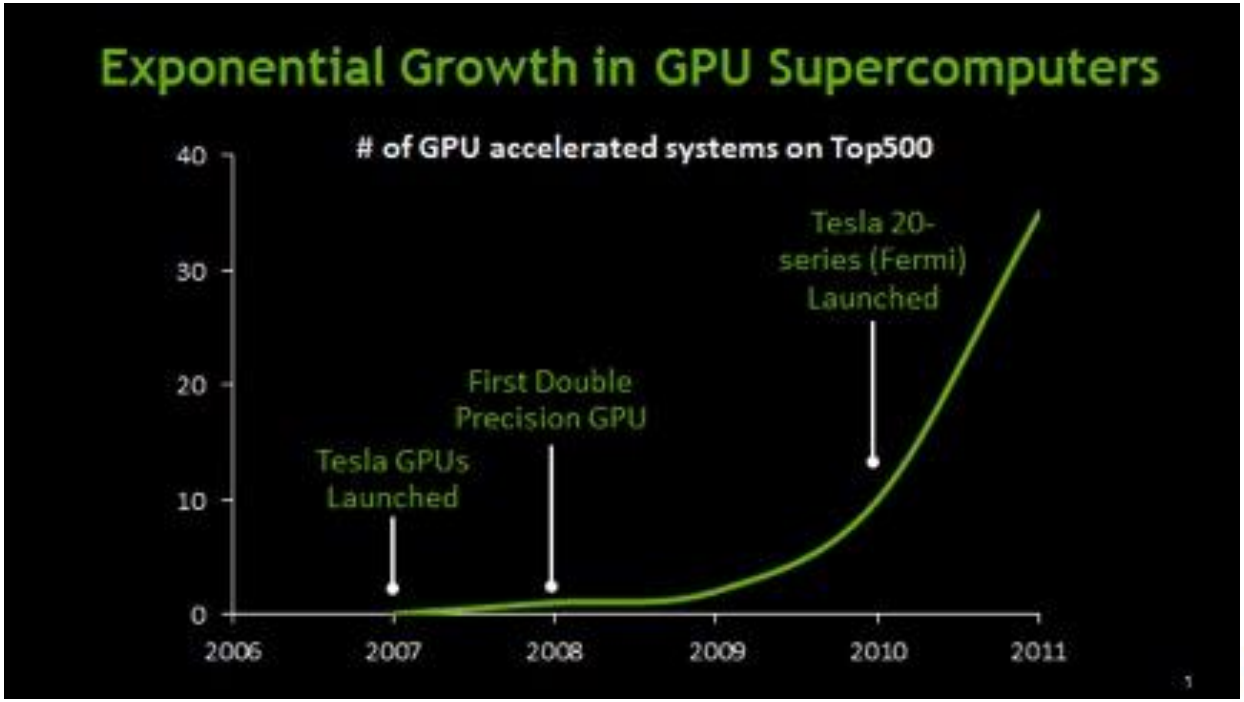
- Le GPU sono processori specializzati per problemi che possono essere classificati come *intense data-parallel computations*
  - lo stesso algoritmo è eseguito su molti elementi differenti in parallelo
  - controllo di flusso molto semplice (control unit ridotta)
  - parallelismo a granularità fine e alta intensità aritmetica che nasconde le latenze di load/store (cache ridotta)



"The GPU devotes more transistors to Data Processing"
(NVIDIA CUDA Programming Guide)



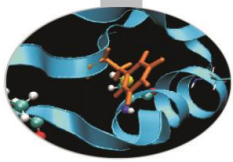
GPU Supercomputers



TOP500 SUPERCOMPUTER SITES

"Le GPU si sono evolute al punto che numerose applicazioni reali possono essere facilmente implementate ed eseguite su di esse in modo nettamente più rapido di quanto avviene sui tradizionali sistemi multi-core. **Le future architetture di computing saranno sistemi ibridi con GPU a core paralleli che lavoreranno in tandem con CPU multi-core.**"

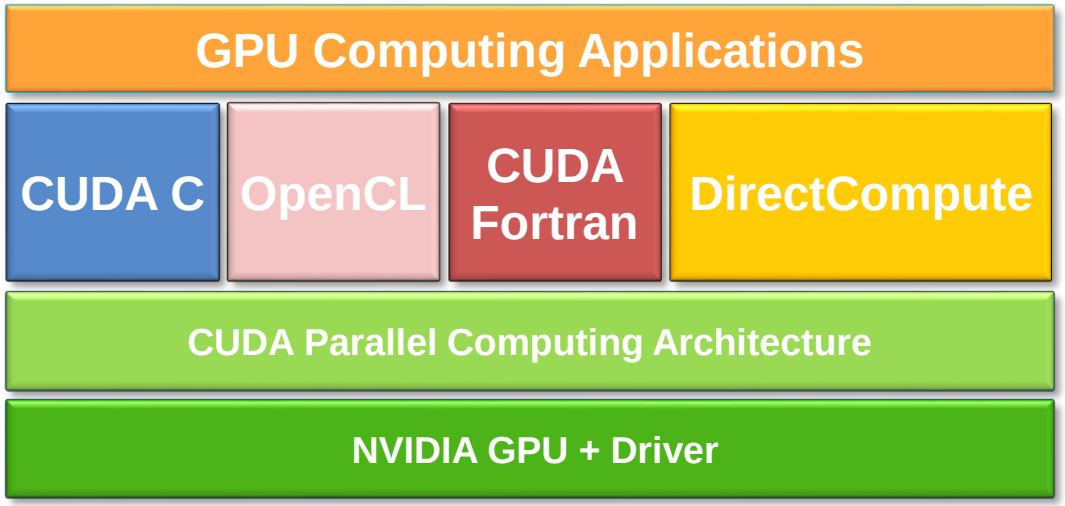
Prof. Jack Dongarra
Direttore dell'Innovative Computing Laboratory
University of Tennessee

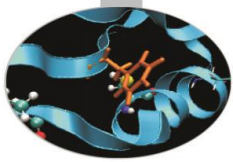


A General-Purpose Parallel Computing Architecture

CUDA (Compute Unified Device Architecture) (NVIDIA 2007)

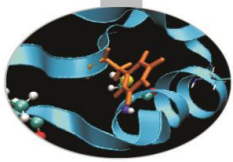
- un'architettura *general purpose* che fornisce un modello "semplice" di programmazione delle GPU
- il cuore di CUDA è basato su un nuovo set di istruzioni architetturali chiamato PTX (Parallel Thread eXecution)
  - fornisce un nuovo paradigma di programmazione parallela *multi-threaded* gerarchico
  - espone la GPU come una *many-core virtual machine*
- fornisce un set di estensioni al linguaggio C/C++ e Fortran che consentono l'utilizzo del paradigma PTX tramite un linguaggio di programmazione a più alto livello



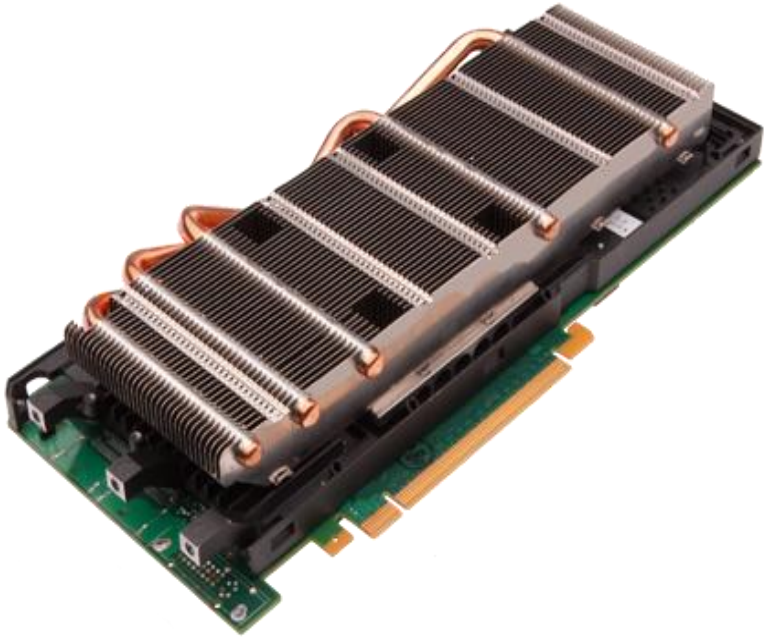


Soluzioni alternative a CUDA per GPU Computing

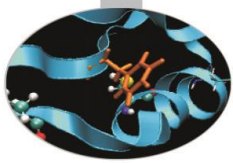
- ATI/AMD solution: ATI Stream
- Microsoft DirectCompute (parte di DirectX 11 API, supportata da GPU con DirectX 10)
- OpenCL (Open Computing Language): modello di programmazione standardizzato sviluppato con accordo delle maggiori produttrici di hardware (Apple, Intel, AMD/ATI, Nvidia). Come CUDA, definisce estensioni di linguaggio e runtime API (Khronos Group)
- Acceleratori (PGI Accelerator, Caps HMPP, ….)



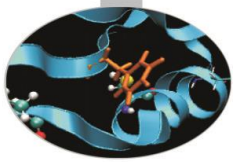
- Modello CUDA e threads
  - co-processore/memoria
  - generazione threads
  - granularità fine
- Come scrivere un codice CUDA
  - 4 passi per iniziare
  - esempio Matrix-Matrix Add
  - gestire la memoria
    e il trasferimento dati CPU-GPU
  - scrivere e lanciare un kernel CUDA



Modello CUDA

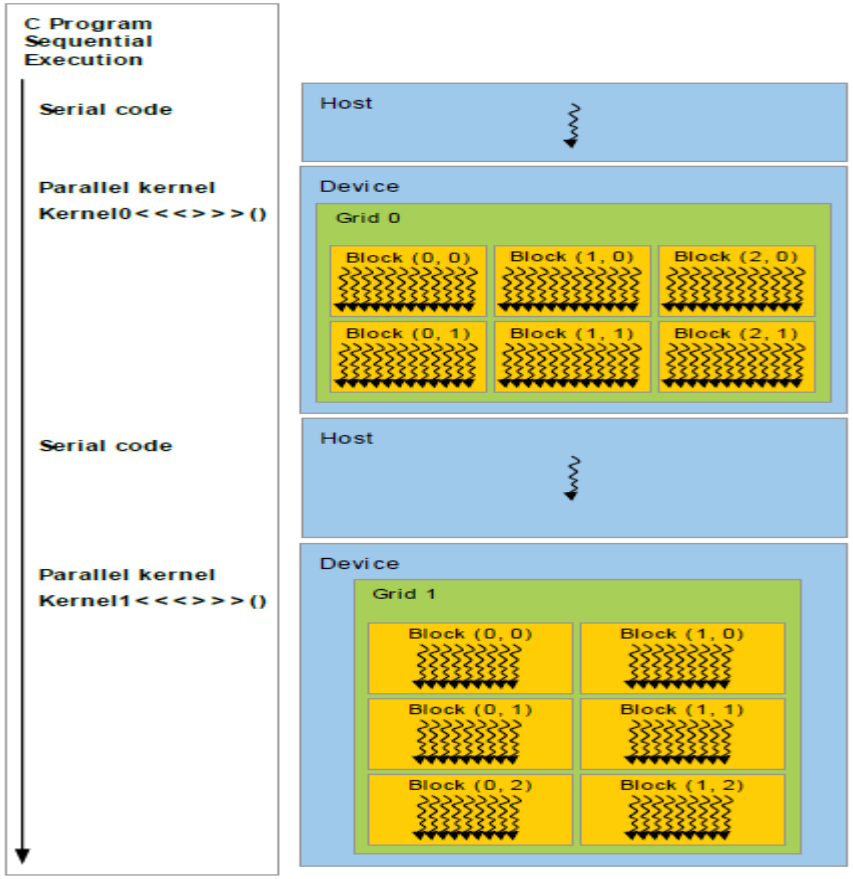


- La GPU è vista come un coprocessore della CPU dotato di una sua memoria e capace di eseguire centinaia di thread in parallelo
  - le parti *data-parallel* e *computational-intensive* di una applicazione sono delegate al device GPU
  - ogni corpo computazionale delle parti *data-parallel* definisce una funzione kernel che viene eseguita identicamente da tutti i *thread* sul *device*
  - ogni thread elabora uno o più elementi in modo indipendente
  - il parallelismo è di tipo:
    - SPMD (*Single-Program Multiple-Data*)
    - SIMT (*Single-Instruction Multiple-Thread*)
- Differenze tra i thread GPU e CPU
  - i thread GPU sono estremamente leggeri
    - nessun costo per la loro attivazione/disattivazione (*content-switch*)
  - la GPU richiede migliaia di threads per la piena efficienza
    - una CPU multi-core richiede un thread per core

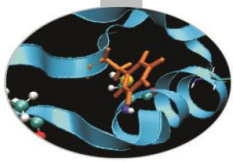


Modello CUDA

- Le parti seriali o a basso parallelismo girano sulla CPU (*host*)
  - i kernel CUDA sono lanciati dal programma principale che gira sulla CPU (da cuda 5.0 è possibile lanciare kernel anche lato device)



Modello Cuda



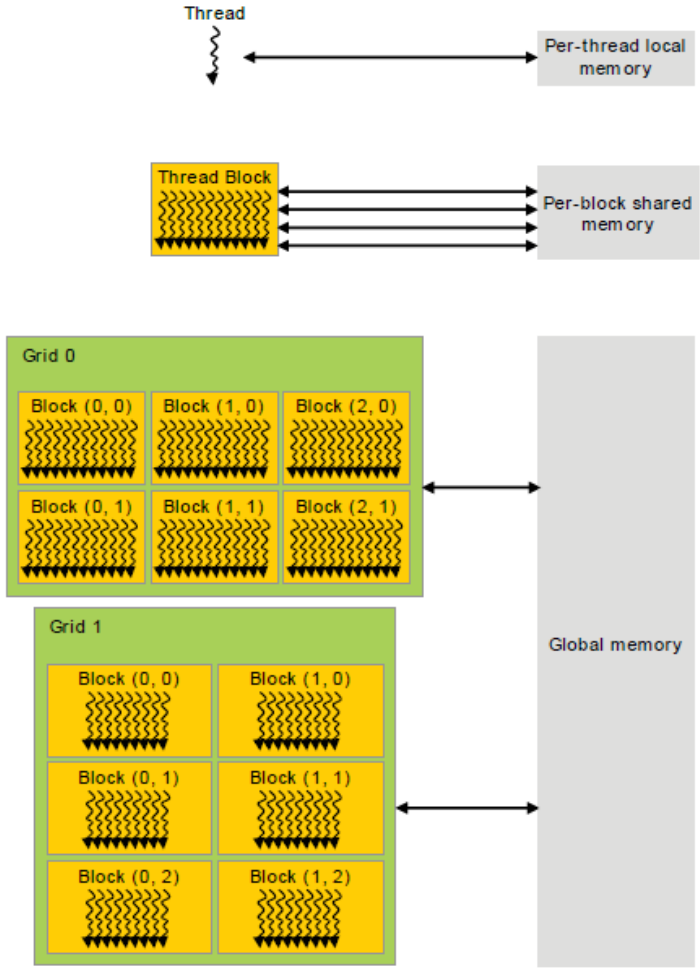
Gerarchia di memoria:

I threads hanno accesso a diverse aree di memoria

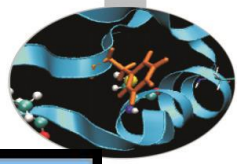
- Ogni threads ha accesso ad un'area di memoria

Locale e privata

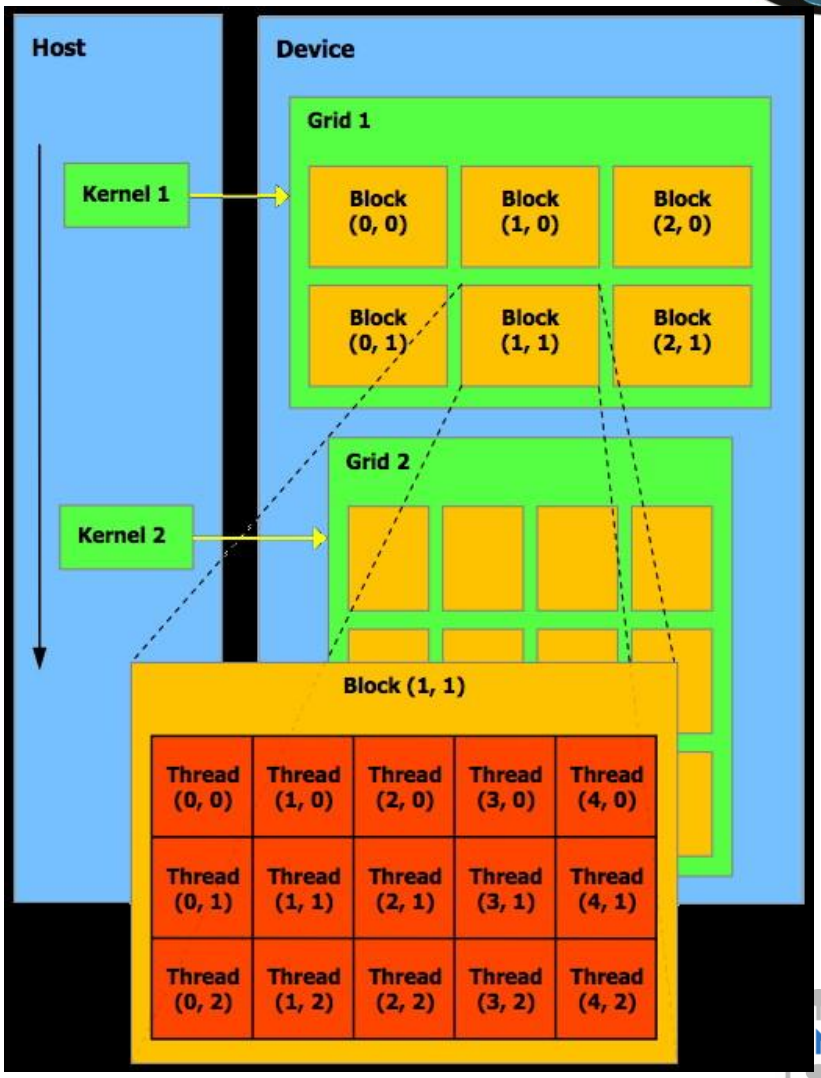
- Ogni blocco ha accesso ad una memoria shared
- Tutti i threads hanno accesso alla stessa area di memoria globale

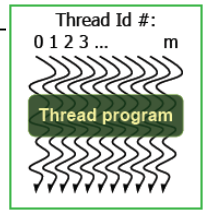


Griglia e blocchi



- CUDA definisce un'organizzazione gerarchica a due livelli per i threads: griglia e blocchi.
- Ogni kernel è eseguito come una griglia di blocchi.
- I thread all'interno di un blocco possono cooperare.
- I threads in blocchi differenti non possono cooperare.



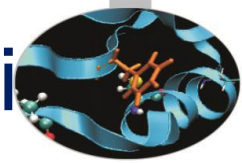


Blocchi di threads

Set di threads che possono cooperare attraverso

- Barriere di sincronizzazione, usando la funzione `__syncthreads();`
- shared memory.

Variabili predefinite per la gestione di griglie e blocchi



```
struct dim3 {unsigned int x,y,z;}
```

La configurazione esatta della griglia è definita in fase di lancio del kernel. La dimensione del blocco e la dimensione della griglia sono contenute nelle variabili:

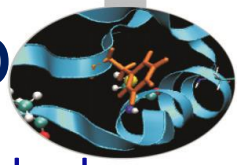
- **dim3 gridDim:** dimensione della griglia in blocchi (1D,2D o 3D)
- **dim3 blockDim**: dimensione del blocco di threads (1D,2D o 3D)

L'identificativo del blocco e del thread vengono assegnati a runtime e sono definiti in variabili built-in preinizializzate:

- **dim3 blockId**: Indice del blocco all'interno della griglia
- **dim3 threadIdx**: Indice del thread all'interno del blocco

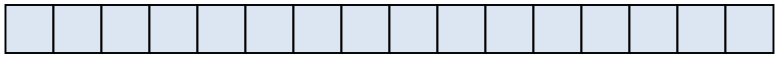
- In vector_types.h ci sono altri tipi predefiniti **(float1, float2,float3,float4,longlong2,…,longlong4,double1,… ,double4)**

ID di blocco e di thread: esempio



Le variabili built-in sono usate per calcolare l'indice globale del thread, in modo da determinare l'area dei dati di competenza.

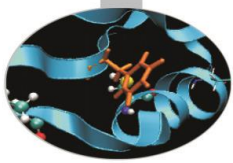
- Supponiamo di avere un array di 16 elementi. Utilizziamo una griglia 1D; se **`gridDim`**=4, si avrà un griglia di 4 blocchi con 4 threads ciascuno.



blockId.x=0	blockId.x=1	blockId.x=2	blockId.x=3
blockDim.x=4	blockDim.x=4	blockDim.x=4	blockDim.x=4
threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3
idx=0,1,2,3	idx=4,5,6,7	idx=8,9,10,11	idx=12,13,14,15

La relazione `int idx=blockDim.x*blockId.x+threadIdx.x`

associa l'indice locale `threadIdx.x` ad un indice globale `idx`

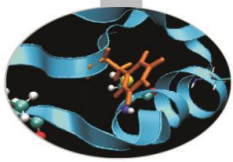


Codice CUDA

1. identificare le parti *data-parallel* computazionalmente onerose e isolare il corpo computazionale
2. individuare i dati coinvolti da dover trasferire sul *device*
3. modificare il codice in modo da girare sulla GPU
   a. allocare la memoria sul *device*
   b. trasferire i dati necessari dall'host al *device*
   c. specificare la modalità di esecuzione del kernel CUDA
   d. trasferire i risultati dal *device* all'*host*
4. implementare il corpo computazionale in un kernel CUDA

Somma di Matrici

1. identificare le parti data-parallel
2. individuare i dati coinvolti da trasferire

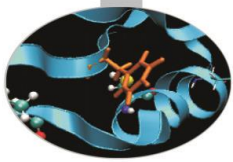


```
int main(int argc, char *argv[]) {
  int i, j;
  const int N = 1000, M = 1000;
  double A[N][M], B[N][M], C[N][M];

  initMatrix(A, N, M, 1.0);
  initMatrix(B, N, M, 2.0);
  initMatrix(C, N, M, 0.0);

  printMatrix(A, N, M);
  printMatrix(B, N, M);

  // C = A + B
  for (i=0; i<N; i++)
    for (j=0; j<M; j++)
      C[i][j] = A[i][j] + B[i][j];

  printMatrix(C, N, M);


  return 0;
}
```

```
program matrixadd
integer :: i, j
integer, parameter :: N=1000, M=1000
real(kind(0.0d0)), dimension(N,M):: A, B,
C

call initMatrix(A, N, M, 1.0)
call initMatrix(B, N, M, 2.0)
call initMatrix(C, N, M, 0.0)

call  printMatrix(A, N, M)
call  printMatrix(B, N, M)

! C = A + B
do j = 1,M
  do i = 1,N
    C(i,j) = A(i,j) + B(i,j)
  end do
end do

call printMatrix(C, N, M)

end program
```

Somma di Matrici

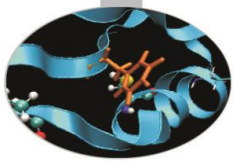
3. modificare il codice in modo da girare sulla GPU



- CUDA fornisce:
  - estensioni ai linguaggi C/C++ o Fortran per scrivere i kernel CUDA
  - alcune API (*Application Programming Interface*), che consentono di gestire da *host* le operazioni da eseguire sul *device*
  - una libreria runtime divisa in:
    - una componente comune che include il supporto per tipi predefiniti della libreria C e Fortran standard
    - una componente *host* per controllare ed accedere a una o più GPU dal lato *host* (CPU)
    - una componente *device* che fornisce funzioni specifiche per la GPU

Somma di Matrici

3. modificare il codice in modo da girare sulla GPU
   a) allocare la memoria sul device



- API CUDA C: `cudaMalloc(void **p, size_t size)`
  - alloca size byte nella memoria globale della GPU
  - restituisce l'indirizzo della memoria allocata sul *device*

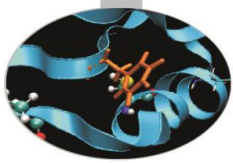
```
double *A_dev, *B_dev, *C_dev;


cudaMalloc((void **)&A_dev, N * M * sizeof(double));
cudaMalloc((void **)&B_dev, N * M * sizeof(double));
cudaMalloc((void **)&C_dev, N * M * sizeof(double));
```

- in CUDA Fortran basta dichiarare degli array con attributo **device** e allocarli con **allocate**

```
real(kind(0.0d0)), device, allocatable, dimension(:,:) :: A_dev,
B_dev, C_dev
allocate( A_dev(N,M), B_dev(N,M), C_dev(N,M) )
```

Somma di Matrici

3. modificare il codice in modo da girare sulla GPU
   a) allocare la memoria sul device

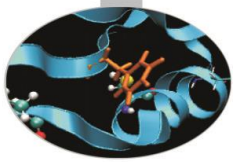


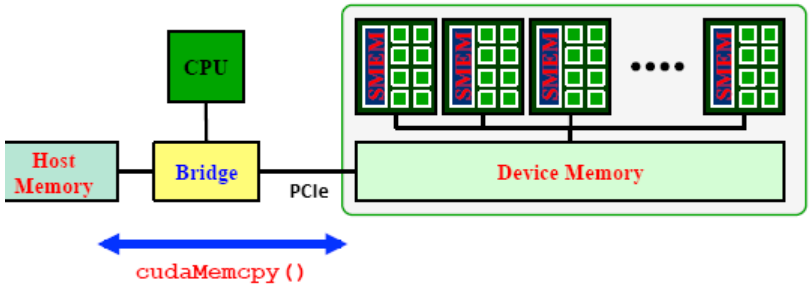
```
double *A_dev;

cudaMalloc((void **) &A_dev, N*M*sizeof(double));
```

- &A_dev
  - A_dev è una variabile che risiede sull'host
  - è una variabile che dovrà ospitare un indirizzo di memoria ad un double (indirizzo di memoria del *device*)
  - il C passa gli argomenti alle funzioni per valore (copia)
    - per far si che venga modificato il contenuto di A_dev, dobbiamo passargli l'indirizzo della variabile, non il valore
- (void **) è un cast per far si che cudaMalloc possa gestire puntatori ad aree di memoria di qualsiasi tipo

Somma di Matrici

3. modificare il codice in modo da girare sulla GPU
b) trasferire i dati necessari dall'host al device





- API CUDA C:

```
cudaMemcpy(void *dst, void *src, size_t size, direction)
```

  - copia size byte in dst a partire da src

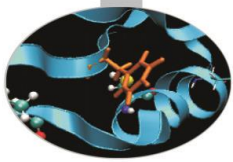
```
cudaMemcpy(A_dev, A, sizeof(A), cudaMemcpyHostToDevice);
cudaMemcpy(B_dev, B, sizeof(B), cudaMemcpyHostToDevice);
```

- in CUDA Fortran basta assegnare le variabili dichiarate e allocate con l'attributo **device**

```
A_dev = A ; B_dev = B
```

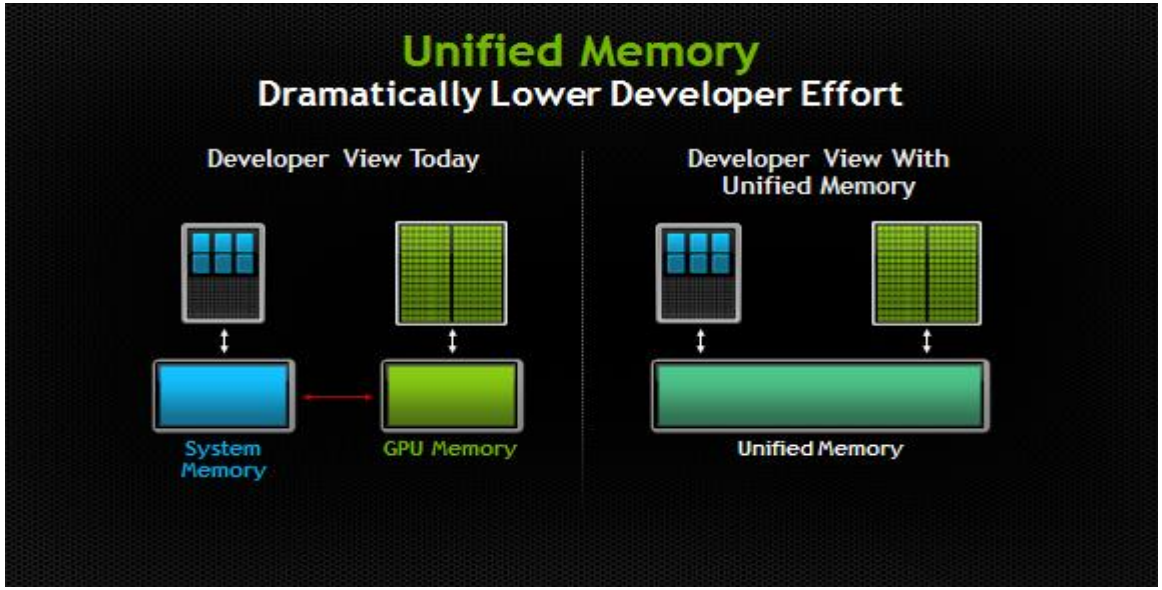
Somma di Matrici

3. modificare il codice in modo da girare sulla GPU
b) trasferire i dati necessari dall'host al device



- `cudaMemcpy(void *dst, void *src, size_t size, direction)`
  - dst: Puntatore alla destinazione
  - src: Puntatore alla sorgente
  - size: Numero di byte da copiare
  - direction: tipo di trasferimento
    - da CPU a GPU: cudaMemcpyHostToDevice
    - da GPU a CPU: cudaMemcpyDeviceToHost
    - da GPU a GPU: cudaMemcpyDeviceToDevice
  - inizia a copiare solo quando tutte le precedenti chiamate CUDA sono state completate
  - blocca il thread CPU fino a quando tutti i byte non sono stati copiati
  - ritorna solo quando la copia è completa

Somma di Matrici CUDA 6.5 Unified Memory



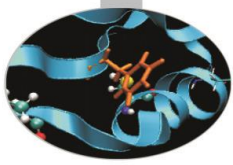
3. modificare il codice in modo da girare sulla GPU

a) Allocare memoria sul device
b) trasferire i dati necessari dall'host al device

- `cudaError_t cudaMallocManaged(void **devPtr, size_t size, unsigned int flags=0);`
  - Vengono allocati sulla GPU "size" bytes e viene ritornato un puntatore all'area di memoria sul Device in "devPtr".
  - Il puntatore è valido sia lato Host che Device, un solo puntatore è ora necessario!
  - Non è più necessaria la chiamata a cudaMemcpy!
  - Le copie esplicite HostToDevice e DeviceToHost sono ora automatizzate

Somma di Matrici

3. modificare il codice in modo da girare sulla GPU
c) specificare la modalità di esecuzione

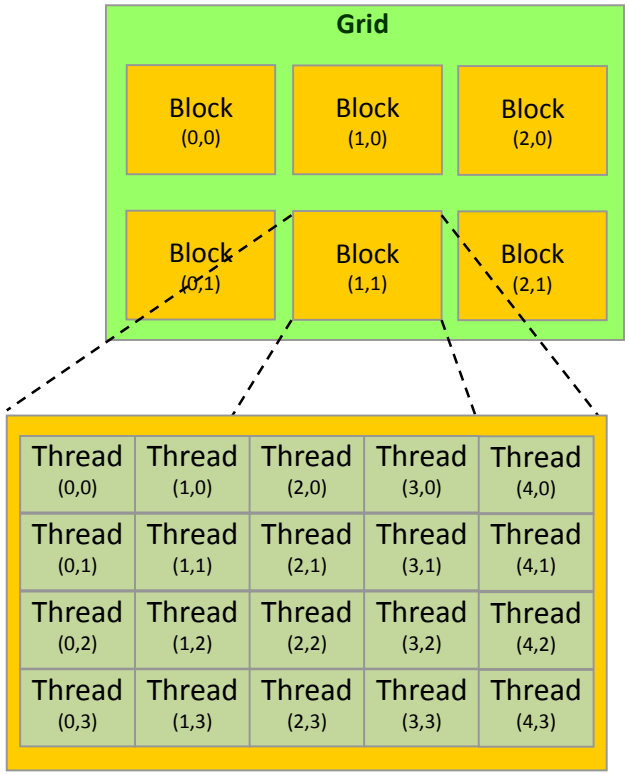


Ogni chiamata a un kernel CUDA deve specificare il numero di thread con cui il kernel verrà eseguito sulla GPU:

- **execution configuration:** estensione CUDA alla sintassi di chiamata a funzione per il lancio dei kernel CUDA:

```
kernelCUDA<<<numBlocks,numThreads>>>(arg1,
                    arg2,...)
```

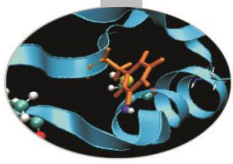
- i thread sono organizzati in modo gerarchico in una griglia di blocchi di thread:
  - **numBlocks:** specifica le dimensioni della griglia in termini di numero di blocchi lungo ogni dimensione
  - **numThreads:** specifica la dimensione di ciascun blocco in termini di numero di thread lungo ciascuna direzione (1D,2D,3D)



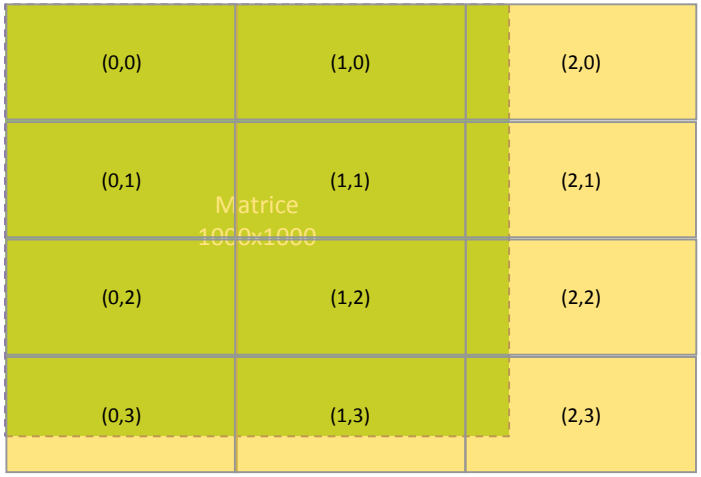
Con questa organizzazione è possibile mappare ogni elemento da elaborare ad un set di *thread*

Somma di Matrici

3. modificare il codice in modo da girare sulla GPU
c) specificare la modalità di esecuzione



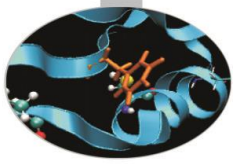
- ogni *thread* della griglia elabora in parallelo un solo elemento della matrice
  - determinare la griglia in modo da generare un numero di *thread* sufficiente a ricoprire tutta la superficie da elaborare
  - scelta la dimensione del blocco, la dimensione della griglia si può adattare al problema
  - alcuni *thread* potrebbero uscire dal dominio (nessun problema)




```
dim3 numThreads(32, 16);
dim3 numBlocks( N/numThreads.x + 1, M/numThreads.y + 1);
gpuMatAdd<<<numBlocks, numThreads>>>( A_dev, B_dev, C_dev, N, M );

type(dim3) :: numBlocks, numThreads
numThreads = dim3( 32, 16, 1 )
numBlocks = dim3( N/numThreads%x + 1, M/numThreads%y + 1, 1 )
call gpuMatAdd<<<numBlocks,numThreads>>>( A_dev, B_dev, C_dev, N, M )
```

Somma di Matrici

3. modificare il codice in modo da girare sulla GPU
d) trasferire i risultati dal device all'host

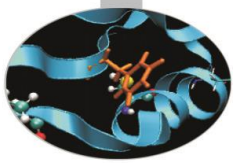


```
double *A_dev, *B_dev, *C_dev;
cudaMalloc((void **)&A_dev, N * M * sizeof(double));
cudaMalloc((void **)&B_dev, N * M * sizeof(double));
cudaMalloc((void **)&C_dev, N * M * sizeof(double));

cudaMemcpy(A_dev, A, sizeof(A), cudaMemcpyHostToDevice);
cudaMemcpy(B_dev, B, sizeof(B), cudaMemcpyHostToDevice);

dim3 numThreads(32, 16);
dim3 numBlocks( N/numThreads.x + 1, M/numThreads.y + 1);
gpuMatAdd<<<numBlocks, numThreads>>>( A_dev, B_dev, C_dev, N, M );
cudaMemcpy(C, C_dev, N * M * sizeof(double), cudaMemcpyDeviceToHost);
```

```
real(kind(0.0d0)), device, allocatable, dimension(:,:) :: A_dev, B_dev, C_dev
type(dim3) :: numBlocks, numThreads
allocate( A_dev(N,M), B_dev(N,M), C_dev(N,M) )
A_dev = A; B_dev = B

numThreads = dim3( 32, 16, 1 )
numBlocks = dim3( N/numThreads%x + 1, M/numThreads%y + 1, 1 )
call gpuMatAdd<<<numBlocks,numThreads>>>( A_dev, B_dev, C_dev, N, M )
C = C_dev
```

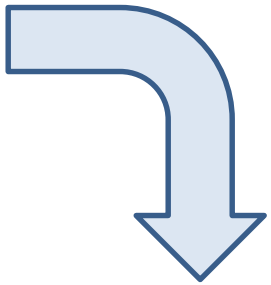
Somma di Matrici

4. Implementare il Kernel CUDA

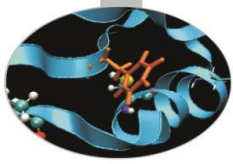


- ogni *thread* esegue lo stesso kernel, ma opera su dati diversi:
  - trasformiamo il corpo computazionale in una funzione
  - assumiamo che ogni *thread* abbia un identificativo univoco
  - indirizziamo ogni elemento da elaborare mediante questo identificativo

```
const int N = 1000, M = 1000;
double A[N][M], B[N][M], C[N][M];

// C = A + B
for (i=0; i<N; i++)
  for (j=0; j<M; j++)
    C[i][j] = A[i][j] + B[i][j];
```



```
void gpuMatAdd (const double *A, const double *B, double *C, int N, int M)
{
  // for (i=0; i<N; i++)
  //     for (j=0; j<M; j++)
  // index is a unique identifier of each GPU thread
  int index =  ....  ;
  C[index] = A[index] + B[index];
}
```

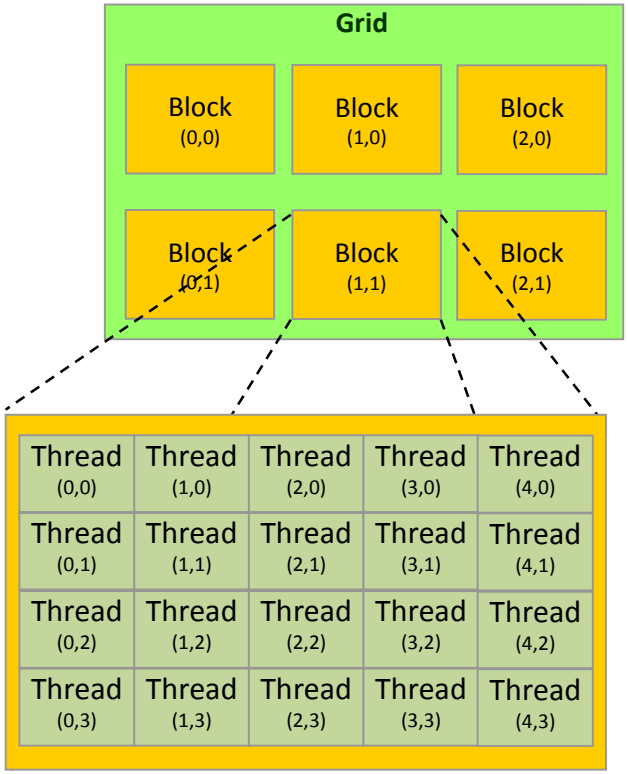


Somma di Matrici

4. Implementare il Kernel CUDA

- possiamo identificare un *thread* in modo **univoco** mediante coordinate globali o attraverso un indice con cui accedere agli elementi da elaborare:
  - ogni *thread* all'interno di un blocco può essere identificato da un set di coordinate cartesiane
  - ogni blocco all'interno della griglia può essere identificato mediante un set di coordinate cartesiane bidimensionale
- CUDA mette a disposizione le seguenti variabili:

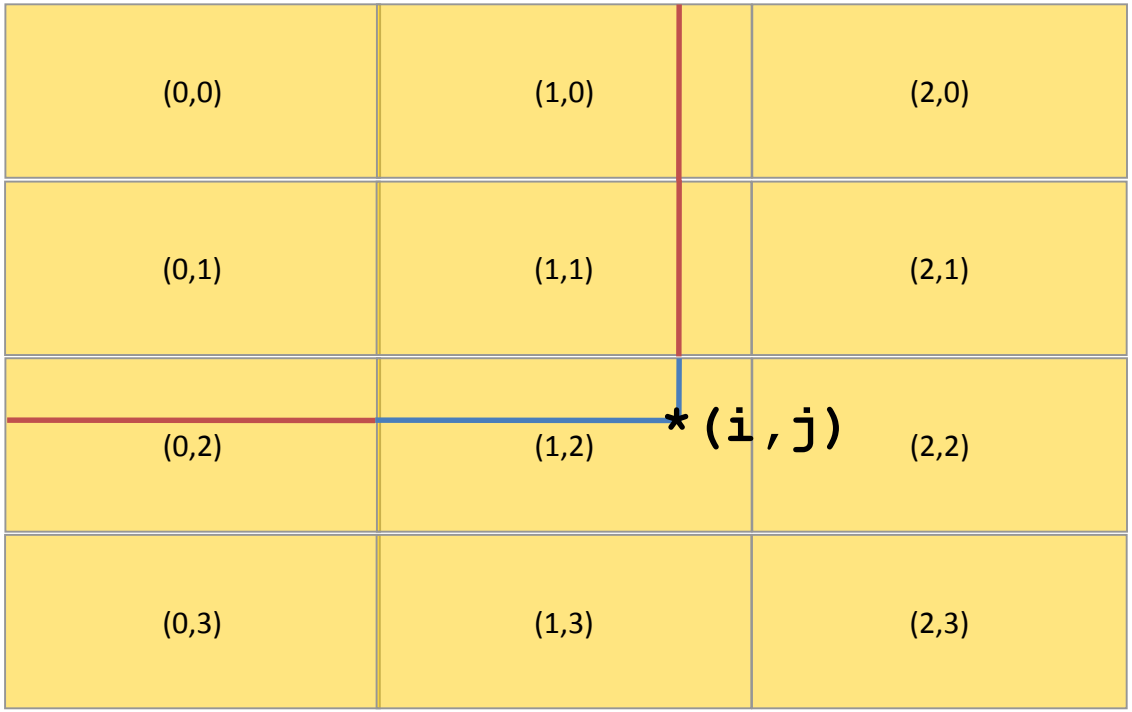
```
uint3 threadIdx, blockIdx
dim3  blockDim,  gridDim
```



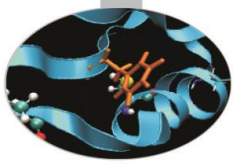

```
i = blockIdx.x * blockDim.x + threadIdx.x;
j = blockIdx.y * blockDim.y + threadIdx.y;
index = j * gridDim.x * blockDim.x + i;
```

Somma di Matrici

4. Implementare il Kernel CUDA

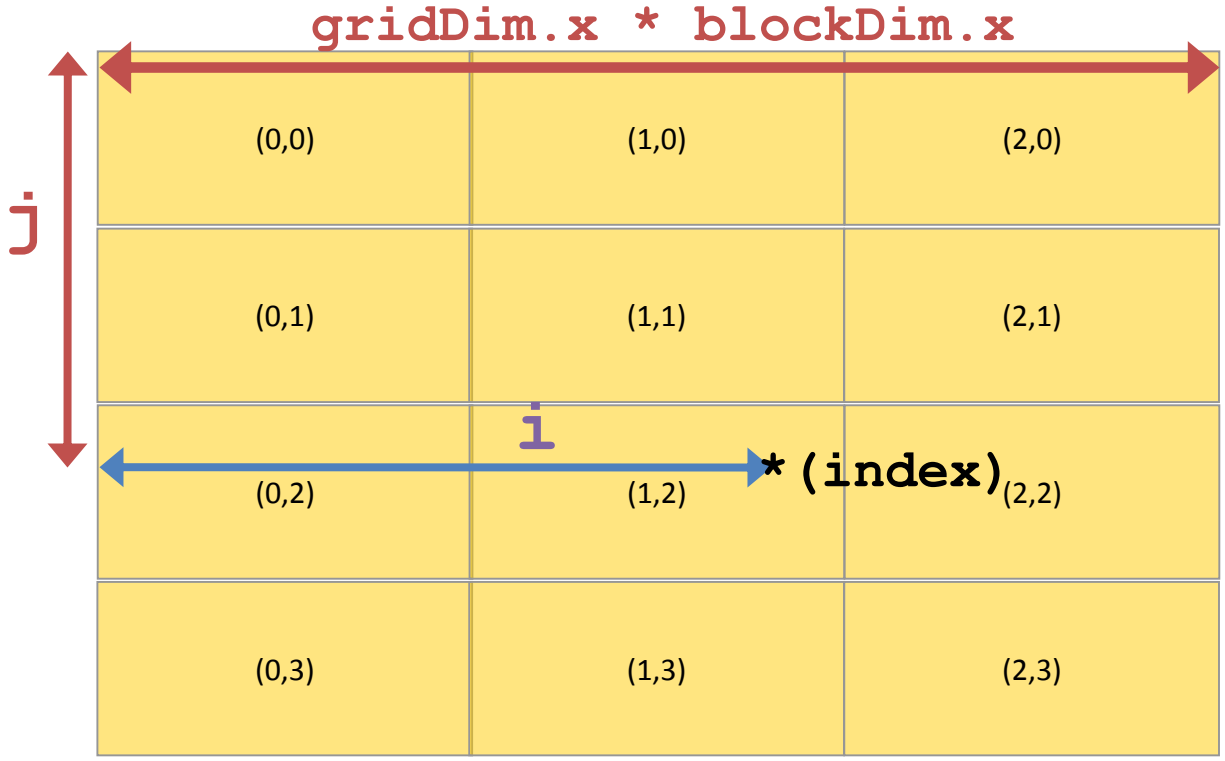



```
i = blockIdx.x * blockDim.x +threadIdx.x;
j = blockIdx.y * blockDim.y +threadIdx.y;
```



Somma di Matrici

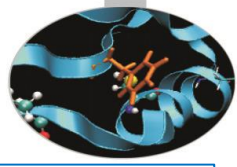
4. Implementare il Kernel CUDA




```
i = blockIdx.x * blockDim.x + threadIdx.x;
j = blockIdx.y * blockDim.y + threadIdx.y;

index = j * gridDim.x * blockDim.x + i;
```

Somma di Matrici

4. Implementare il Kernel CUDA

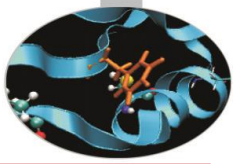


```
__global__  gpuMatAdd (const double *A, const double *B, double *C, int N, int M)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;

  if (i >= M || j >= N) return; // handle out-of-bound threads

  int index = j * M  +  i;
  C[index] = A[index] + B[index];
}
```

- **__global__** è un qualificatore che definisce un kernel GPU
- le funzioni con qualificatore __global__
  - possono essere invocate dall'*host* e dal *device* per i devices che supportano il **dynamic parallelism (compute capability >= 3.5)**
  - devono essere invocate con una *execution configuration*
  - devono avere un tipo di ritorno *void*
  - possono essere ricorsive per i devices che supportano il **dynamic parallelism (compute capability >= 3.5)**
  - sono asincrone (ritornano il controllo all'*host* prima del loro completamento)
- **attributes(global)** in fortran

Somma di Matrici

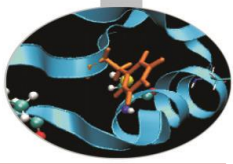
4. Implementare il Kernel CUDA



```
module matrix_algebra_cuda
use cudafor
contains
attributes(global) subroutine gpuMatAdd (A, B, C, N, M)
  implicit none
  integer, intent(in), value :: N, M
  real, intent(in) :: A(N,M), B(N,M)
  real, intent(inout) :: C(N,M)
  integer :: i,j

  i = ( blockIdx%x - 1 ) * blockDim%x + threadIdx%x
  j = ( blockIdx%y - 1 ) * blockDim%y + threadIdx%y

  if (i .gt. N .or. j .gt. M) return

   C(i,j) = A(i,j) + B(i,j)
end subroutine
end module matrix_algebra_cuda
```

Somma di Matrici

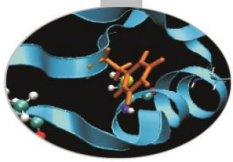
4. Implementare il Kernel CUDA



```
attributes(global) subroutine gpuMatAdd (A, B, C, N, M)
  ...
end subroutine

program matrixAdd
use cudafor
implicit none
interface
  attributes(global) subroutine gpuMatAdd (A, B, C, N, M)
    integer, intent(in), value :: N, M
    real, intent(in) :: A(N,M), B(N,M)
    real, intent(inout) :: C(N,M)
    integer :: i,j
  end subroutine
end interface
  ...

end program matrixAdd
```

se non definiti in un modulo, i kernel devono specificare l'interfaccia

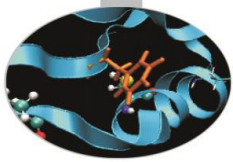
Definizione di funzioni in CUDA



	Eseguito sulla:	Invocabile solo dalla:
`__device__ float DeviceFunc()`	GPU	GPU
`__global__ void KernelFunc()`	GPU	CPU
`__host__ float HostFunc()`	CPU	CPU

- **`__global__`** definisce una funzione kernel che viene seguita sulla GPU
- **`__global__`** deve avere ritorno void

Gestione della memoria globale



- Allocazione della memoria globale

```
cudaMalloc(void** bufferPtr, size_t n)
cudaMallocManaged(void **devPtr, size_t size, unsigned int flags=0);
cudaFree(void* bufferPtr)
```

- Inizializzazione della memoria globale

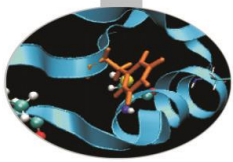
```
cudaMemset(void* devPtr, int value, size_t count)
```

- Trasferimento dati host-device-host

```
cudaMemcpy(void* destPtr,void* srcPtr, int size, cudaMemcpyKind kind)
```

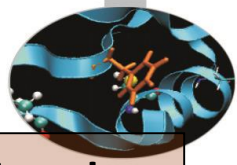
Dove:

enum **cudaMemcpyKind**

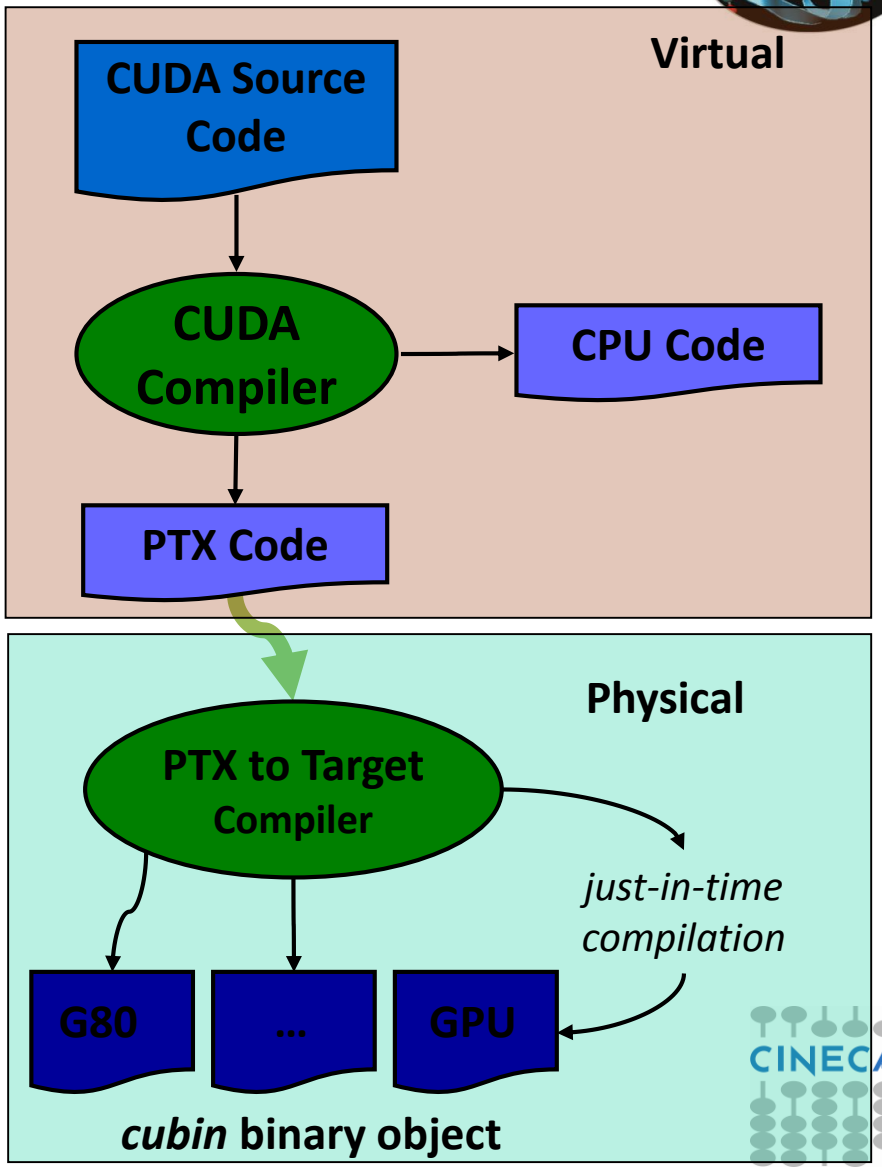


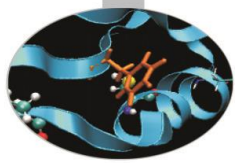
- Compilatore e passi
  - PTX, cubin, what's inside
  - computing capability
- Esercitazione:
  - ambiente e utility devQuery
  - compilazione esempio A+B
  - modifica A+B in A = A + c*B
    - variare la griglia di threads

Compilare Codice CUDA



- Ogni file sorgente contenente estensioni CUDA deve essere compilato con un compilatore CUDA compliant
  - nvcc per CUDA C (NVIDIA)
  - pgf90 -Mcuda per CUDA Fortran (PGI)
- il compilatore processa il sorgente separando codice *device* dall'*host*
  - il codice *host* viene rediretto a un compilatore standard di default
  - il codice *device* viene tradotto in PTX
- a partire dal PTX prodotto è possibile:
  - produrre codice oggetto binario (*cubin*) specializzato per una particolare architettura GPU
  - produrre un eseguibile che include PTX e/o codice binario (*cubin*)

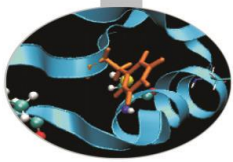




La Compute Capability

- Con *compute capability (c.c)* di una scheda si identificano:
  - Il set di istruzioni e di feature supportate dalla scheda
  - Le regole di coalescenza
  - I limiti delle risorse della scheda
  - Il througput di alcune istruzioni (implementazione hardware)
- La *compute capability* di un device è definita da un major revision number e da un minor revision number X.y.
    - major number (X): identifica l'architettura base del chip
    - minor number (y): aggiunge nuove caratteristiche
- I device attuali hanno c.c:
  - 5.x architettura Maxwell
  - 3.x architettura Kepler
  - 2.x architettura Fermi
  - 1.x architettura Tesla

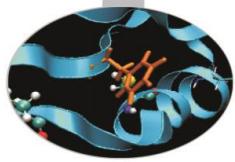
La Compute Capability



Features:

Feature Support (Unlisted features are supported for all compute capabilities)	Compute Capability 1.0	1.1	1.2	1.3	2.x, 3.0	3.5
Atomic functions operating on 32-bit integer values in global memory (Atomic Functions)	No	Yes				
atomicExch() operating on 32-bit floating point values in global memory (atomicExch())						
Atomic functions operating on 32-bit integer values in shared memory (Atomic Functions)	No		Yes			
atomicExch() operating on 32-bit floating point values in shared memory (atomicExch())						
Atomic functions operating on 64-bit integer values in global memory (Atomic Functions)						
Warp vote functions (Warp Vote Functions)						
Double-precision floating-point numbers	No			Yes		

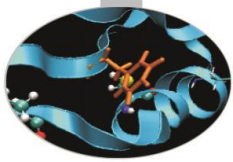
Feature Support (Unlisted features are supported for all compute capabilities)	Compute Capability 1.0	1.1	1.2	1.3	2.x, 3.0	3.5
Atomic functions operating on 64-bit integer values in shared memory (Atomic Functions)	No				Yes	
Atomic addition operating on 32-bit floating point values in global and shared memory (atomicAdd())						
__ballot() (Warp Vote Functions)						
__threadfence_system() (Memory Fence Functions)						
__syncthreads_count(), __syncthreads_and(), __syncthreads_or() (Synchronization Functions)						
Surface functions (Surface Functions)						
3D grid of thread blocks						
Funnel shift (see reference manual)	No					Yes



La Compute Capability

Resources Constraints

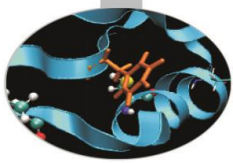
	Compute Capability						
Technical Specifications	**1.0**	**1.1**	**1.2**	**1.3**	**2.x**	**3.0**	**3.5**
Maximum dimensionality of grid of thread blocks	2				3		
Maximum x-dimension of a grid of thread blocks	65535					2^{31}-1	
Maximum y- or z-dimension of a grid of thread blocks	65535						
Maximum dimensionality of thread block	3						
Maximum x- or y-dimension of a block	512				1024		
Maximum z-dimension of a block	64						
Maximum number of threads per block	512				1024		
Warp size	32						
Maximum number of resident blocks per multiprocessor	8					16	
Maximum number of resident warps per multiprocessor	24		32		48	64	
Maximum number of resident threads per multiprocessor	768		1024		1536	2048	
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K	64 K	
Maximum number of 32-bit registers per thread	128				63		255
Maximum amount of shared memory per multiprocessor	16 KB				48 KB		
Number of shared memory banks	16				32		
Amount of local memory per thread	16 KB				512 KB		
Constant memory size	64 KB						
Cache working set per multiprocessor for constant memory	8 KB						
Cache working set per multiprocessor for texture memory	Device dependent, between 6 KB and 8 KB				12 KB		Between 12 KB and 48 KB



La Compute Capability

Instruction Throughput

	Compute Capability					
	1.0 1.1 1.2	1.3	2.0	2.1	3.0	3.5
32-bit floating-point add, multiply, multiply-add	8	8	32	48	192	192
64-bit floating-point add, multiply, multiply-add	N/A	1	16(*)	4	8	64
32-bit floating-point reciprocal, reciprocal square root, base-2 logarithm (`__log2f`), base 2 exponential (`exp2f`), sine (`__sinf`), cosine (`__cosf`)	2	2	4	8	32	32
32-bit integer add, extended-precision add, subtract, extended-precision subtract, compare, minimum, maximum	10	10	32	48	160	160
32-bit integer multiply, multiply-add, extended-precision multiply-add, sum of absolute difference, population count, count of leading zeros, most significant non-sign bit	Multiple instructions	Multiple instructions	16	16	32	32
24-bit integer multiply (`__[u]mul24`)	8	8	Multiple instructions	Multiple instructions	Multiple instructions	Multiple instructions
32-bit integer absolute value, negate	8	8	16	16	32	32



I passi del compilatore CUDA

- Al compilatore va sempre specificata:
  - l'architettura virtuale con cui generare il *PTX code*
  - l'architettura reale per creare il codice oggetto (*cubin*)

```
nvcc -arch=compute_10 -code=sm_10,sm_13
```

selezione architettura virtuale (*PTX code*)

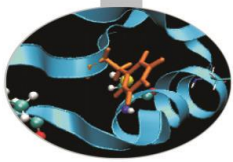
selezione architetture reali (*cubin*)

  - nvcc ammette l'uso dello shortcut `nvcc -arch=sm_XX.`

Esempio di GPU Fermi:

```
nvcc -arch=sm_20
```

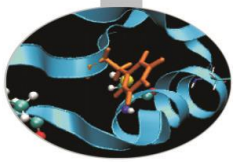
equivalente a: `nvcc -arch=compute_20 -code=sm_20`



Compatibilità del Codice

- Il codice *PTX* viene caricato e compilato a runtime dal device driver. La *just-in-time compilation* aumenta il load time dell'applicazione ma consente all'applicazione di beneficiare delle nuove features del compilatore e di poter girare su device non presenti nel momento in cui il *PTX* è stato generato.
- Il codice *cubin* generato è device-dependent. La compatibilità binaria è garantita tra una minor revision e quella successiva.

Compatibilità del Codice



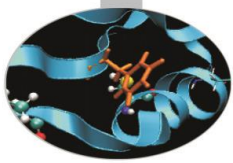
- Gli eseguibili CUDA possono incorporare un codice PTX e più codici cubin per compute capability differenti:

```
nvcc src.cu -arch=compute_10 -code=compute_10,sm_10,sm_13
```

oppure

```
nvcc src.cu -gencode arch=compute_10 code=\'compute_10,sm_10,sm_13\'
```

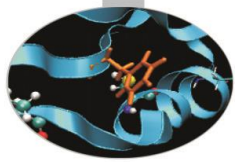
- Il precedente comando crea un eseguibile che incorpora
  - cubin per cc 1.0 e 1.3 (`-code=sm_10,sm_13`)
  - cubin e PTX per cc 1.0 (`-code=compute_10 `)
- l'eseguibile seleziona a *runtime* il codice più appropriato da caricare e eseguire:
  - *cubin* 1.0 per device con cc 1.0
  - *cubin* 1.3 per device con cc 1.3
  - *cubin* compilato *just-in-time* dal PTX 1.0 per device con cc 1.1, 1.2



Il compilatore Nvcc: opzioni

- Alcune delle opzioni più importanti:

-arch sm_13 (o sm_20)	Attiva la possibilità di utilizzare la doppia precisione
-G	Compila per il debug del codice GPU
--ptxas-options=-v	Mostra l'utilizzo dei registri e della memoria (shared e constant)
--maxregcount<N>	Limita il numero di registri
-use_fast_math	Utilizza la libreria matematica veloce
--compiler-options	Specificare opzioni direttamente al compilatore /preprocessore



CUDA API

CUDA è composto da due APIs:

- CUDA runtime API
- CUDA driver API

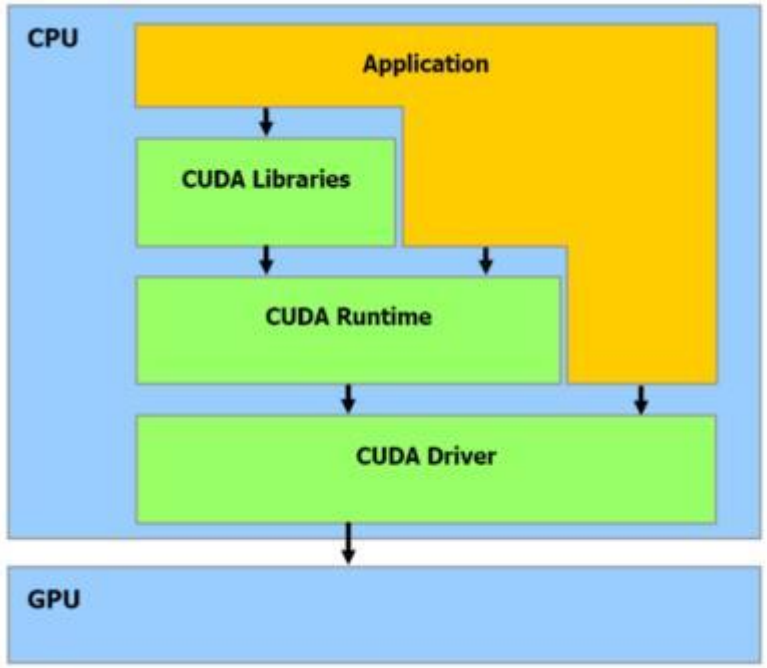
Sono mutualmente esclusive.

Runtime API:

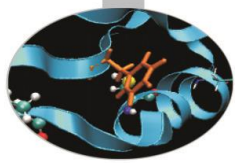
- Più semplici
- Estensione C al linguaggio

Driver API:

- Richiedono più codice
- Garantiscono più controllo
- Non richiedono l'nvcc per compilare il codice host.



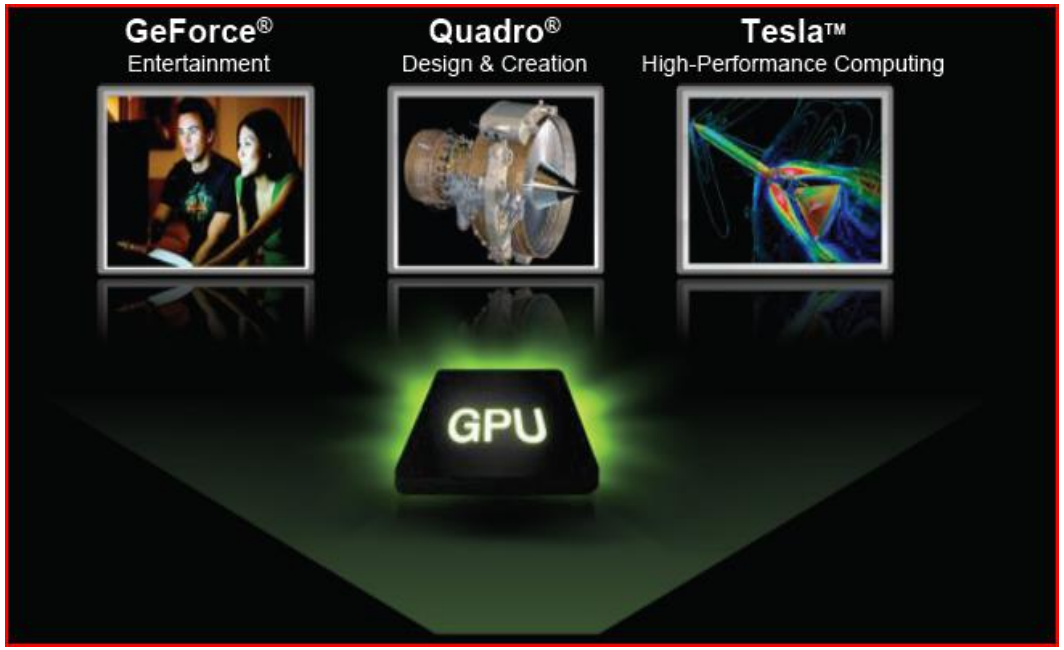
CUDA API



```
// driver API
// initialize CUDA
err = cuInit(0);
err = cuDeviceGet(&device, 0);
err = cuCtxCreate(&context, 0, device);

// setup device memory
err = cuMemAlloc(&d_a, sizeof(int) * N);
err = cuMemAlloc(&d_b, sizeof(int) * N);
err = cuMemAlloc(&d_c, sizeof(int) * N);

// copy arrays to device
err = cuMemcpyHtoD(d_a, a, sizeof(int) * N);
err = cuMemcpyHtoD(d_b, b, sizeof(int) * N);

// prepare kernel launch
kernelArgs[0] = &d_a;
kernelArgs[1] = &d_b;
kernelArgs[2] = &d_c;
// load device code (PTX or cubin. PTX here)
err = cuModuleLoad(&module, module_file);
err = cuModuleGetFunction(&function, module,kernel_name);
// execute the kernel over the <N,1> grid
err = cuLaunchKernel(function, N, 1, 1,  // Nx1x1 blocks
                     1, 1, 1, // 1x1x1 threads
                     0, 0, kernelArgs, 0);
```

```
// runtime API
// setup device memory
err = cudaMalloc((void**)&d_a, sizeof(int) * N);
err = cudaMalloc((void**)&d_b, sizeof(int) * N);
err = cudaMalloc((void**)&d_c, sizeof(int) * N);

// copy arrays to device
err=cudaMemcpy(d_a, a, sizeof(int) * N, cudaMemcpyHostToDevice);
err=cudaMemcpy(d_b, b, sizeof(int) * N, cudaMemcpyHostToDevice);

// launch kernel over the <N, 1> grid
 matSum<<<N,1>>>(d_a, d_b, d_c); // yum, syntax sugar!
```

Cuda – enabled GPUs



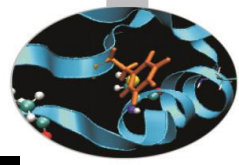
Mainstream & laptops: GeForce
• Target: videogames and multi-media

Workstation: Quadro
• Target: graphic professionals who use CAD and 3D modeling applications
•Maggiore memoria e driver specifici per l'acellerazione delle applicazioni.
GPGPU: Tesla
• Target: High Performance Computing

http://developer.nvidia.com/cuda-gpus

Cuda Driver e Toolkit

- https://developer.nvidia.com/cuda-downloads



NVIDIA. DEVELOPER ZONE

DEVELOPER CENTERS ▸ TECHNOLOGIES ▸ TOOLS ▸ RESOURCES ▸ COMMUNITY

CUDA ZONE

Home > CUDA ZONE > Tools & Ecosystem > CUDA Toolkit

CUDA Downloads

CUDA 5.5 PRODUCTION RELEASE

The CUDA 5.5 installers include the CUDA Toolkit, SDK code samples, Nsight Visual Studio edition (for Window
Eclipse Edition (for Linux / Mac OS X), and developer drivers.

Download CUDA 5.5 and share your feedback with us.

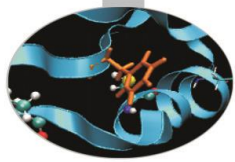
All the CUDA Toolkits are subject to the license terms in this End User License Agreement.

CUDA 5.5 Production Release Release Notes.

Operating System	Distribution	Architecture x86 64-Bit	Architecture x86 32-Bit	Architecture ARMv7	Related Documentation
Windows	Vista, 7, 8 - Notebook	64-Bit	32-Bit		Windows Getting Started Guide
	Vista, 7, 8 - Desktop	64-Bit	32-Bit		
	XP - Desktop*	64-Bit	32-Bit		
Linux	RHEL 6	RPM RUN			Linux Getting Started Guide RPM / DEB Installation Instructions RUN Installation Instructions
	RHEL 5.5	RUN			
	Fedora 18	RPM RUN			
	OpenSUSE 12.2	RPM RUN			
	SLES 11 (SP1 & SP2)	RPM RUN			
	Ubuntu 12.04	DEB** RUN	DEB** RUN	DEB	
	Ubuntu 12.10	DEB RUN	DEB RUN		
	Ubuntu 10.04	RUN	RUN		
Mac OSX	10.7 & 10.8	PKG			Mac Getting Started Guide
	10.9	PKG			

Esercitazione

Informazioni per le esercitazioni



- Accedere al cluster eurora con il comando:

  ```
  ssh a08traXX@login.eurora.cineca.it
  ```

  Dove XX va da 49 a 68.

- Scaricare gli esercizi con il comando :

```
wget https://hpc-forge.cineca.it/files/CoursesDev/public/2014/Introduction_to_Scientific_Programming_using_GPGPU_and_CUDA/Milan/CUDA_exercises.zip --no-check-certificate
```

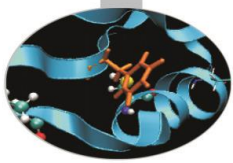
- Configurare l'ambiente per CUDA con i comandi:

  ```
  module load profile/advanced
  module load autoload cuda/6.5.14
  ```

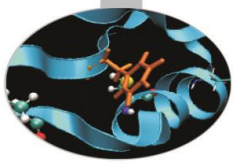
- Andare su un nodo di calcolo con il comando:

  ```
  qsub -I -V -A train_cgpM2014 -q R1422392
  ```

Esercizi



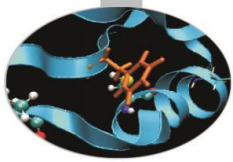
- **deviceQuery (SDK): informazioni sui device grafici presenti**
- **"Compilare un programma CUDA"**
  - `cd Esercizi/MatrixAdd.  Compilare:`
  - `nvcc matrixadd_cuda.cu -o matrixadd_cuda`
  - `nvcc -arch=sm_20 matrixadd_cuda.cu -o matrixadd_cuda`
  - `nvcc -arch=sm_20  -ptx matrixadd_cuda.cu`
  - `nvcc -arch=sm_20  -keep matrixadd_cuda.cu -o matrixadd_cuda`
  - `nvcc -arch=sm_20  -keep  -clean matrixadd_cuda.cu -o matrixadd_cuda`
  - Eseguire:
  - `./matrixadd_cuda`
- **copy:**
  - Vengono allocati 3 array: h_A,h_R, d_A. I dati vengono inizializzati sull'host nel vettore h_A e trasferiti sul device in d_A. Successivamente ricopiati sull'host nel vettore h_B. Gli array sull'host h_A e h_B vengono poi confrontati (cudaMemcpy, cudaMemset, cudaMalloc)
- **incrementArray:**
  - il programma somma a tutti gli elementi dell'array A di dimensione N lo scalare b: A = A + b
  - completare il sorgente CUDA C
  - compilare ed eseguire
  - Aumentare la dimensione di N e controllare il risultato.
- **Variazione sul tema di matrixAdd:**
  - cd MatrixAddInPlace
  - modificare il kernel in modo che A= A + c*B
  - modificare la griglia di blocchi da (16,16) a (32,32) a (64,64)



Codice CUDA

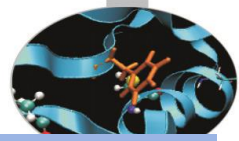
1. identificare le parti *data-parallel* computazionalmente onerose e isolare il corpo computazionale
2. individuare i dati coinvolti da dover trasferire sul *device*
3. modificare il codice in modo da girare sulla GPU
   a. allocare la memoria sul *device*
      C: cudaMalloc. Fortran: attributo device e allocazione
   b. trasferire i dati necessari dall'host al *device*
      C: cudaMemcpy. Fortran: M_dev = M
   c. specificare la modalità di esecuzione del kernel CUDA
      CudaKernel<<<numBlocks, numThreads>>>( ); in C
      call CudaKernel<<<numBlocks,numThreads>>> ( ) in Fortran
   d. trasferire i risultati dal *device* all'*host*
      C: cudaMemcpy. Fortran: M = M_dev
4. implementare il corpo computazionale in un kernel CUDA

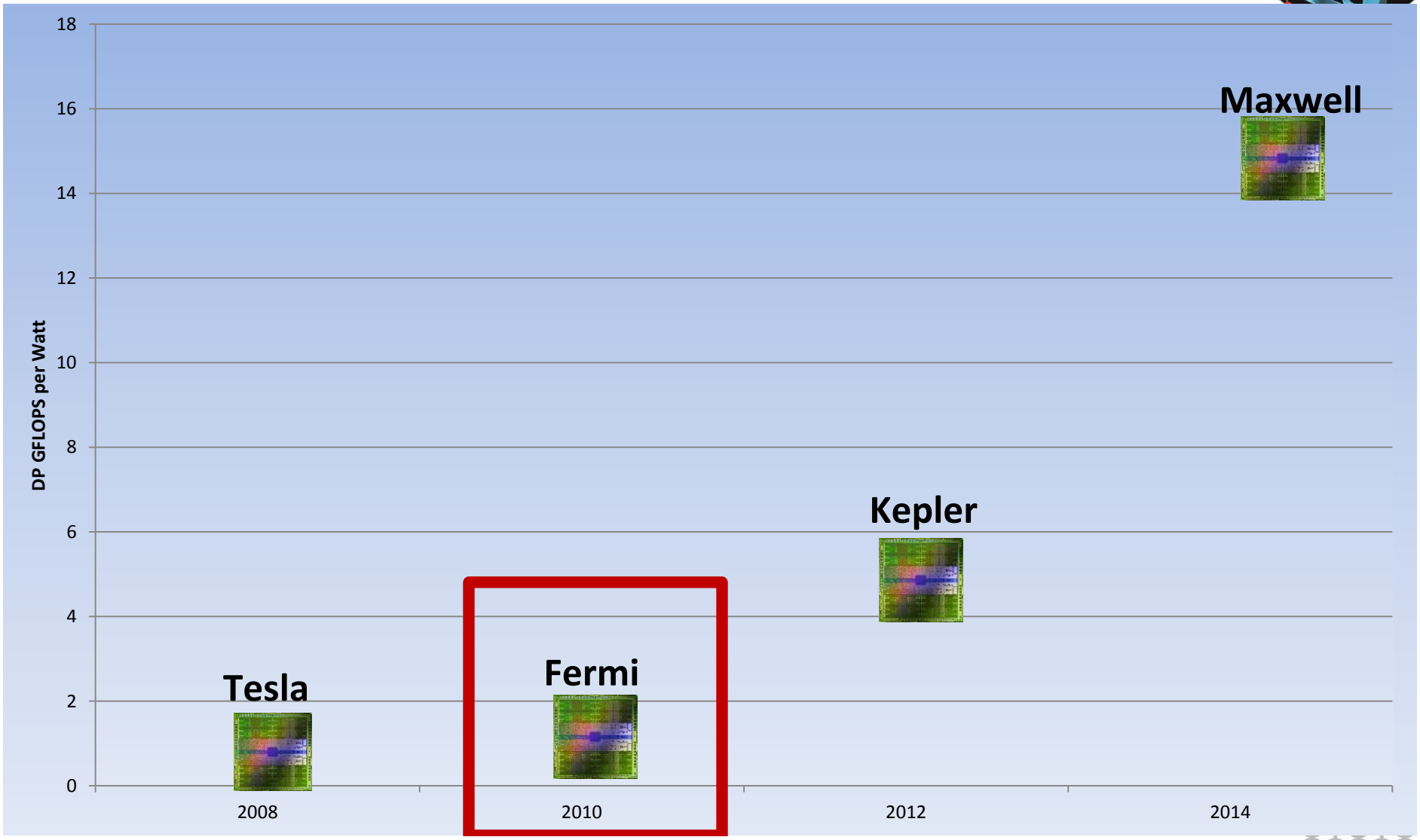
GPU Hardware e architettura Fermi



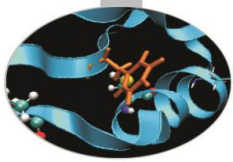
- GPU Hardware e architettura Fermi e Kepler
  - lo Streaming Multiprocessor (SM)
  - il modello di esecuzione
    - mappa software/hardware

CUDA GPU Roadmap

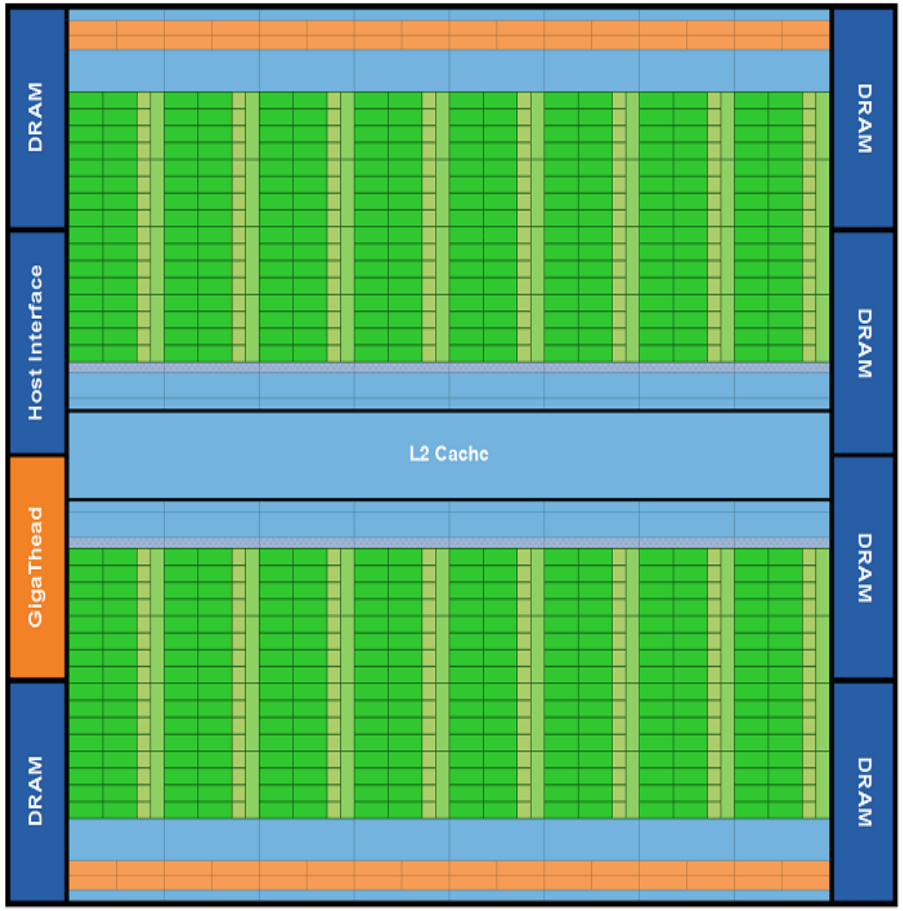




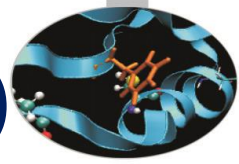
L'architettura hardware NVIDIA Fermi



- 16 chip multi-core per calcolo general purpose (Streaming Multiprocessors, SM)
- 6 banchi di memoria GDDR5 (ampiezza a 384-bit) con indirizzamento a 64-bit (>4GB)
- NVIDIA Parallel DataCache
  - L1 configurabile per SM
  - L2 comune a tutti gli SM
- due controller indipendenti per trasferimento dati da/verso *host* tramite PCI-Express
- uno scheduler globale (GigaThread global scheduler) distribuisce i blocchi di thread agli scheduler degli SM

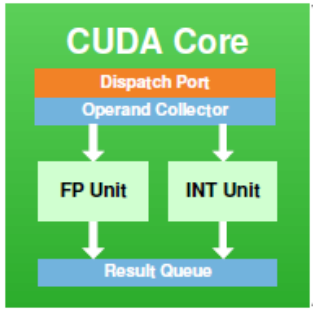


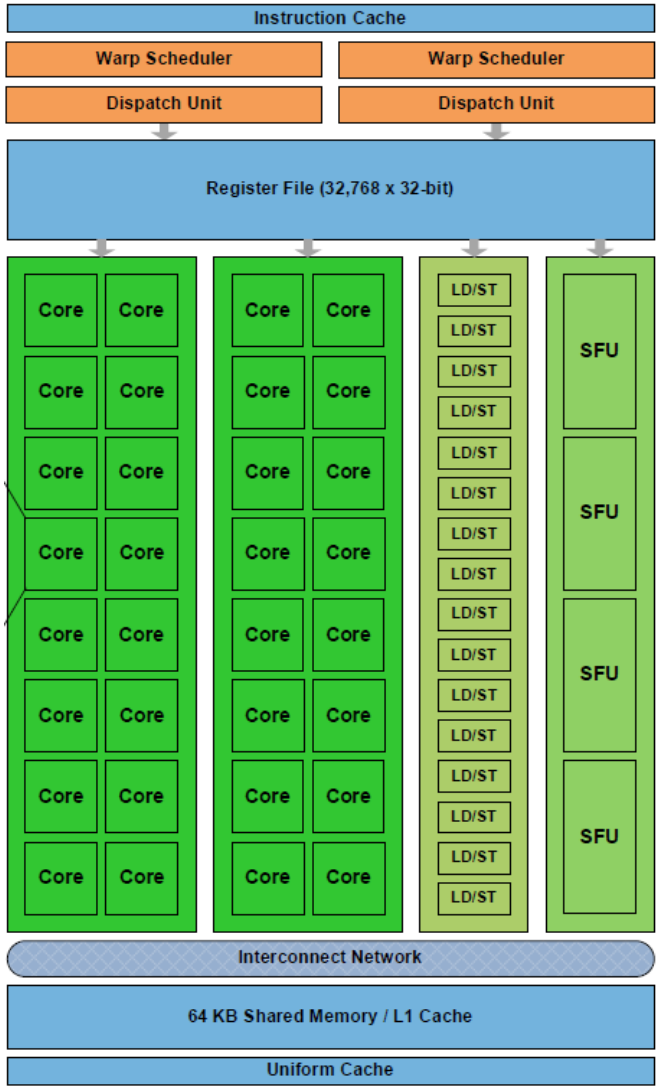
NVIDIA Streaming Multiprocessor(SM)



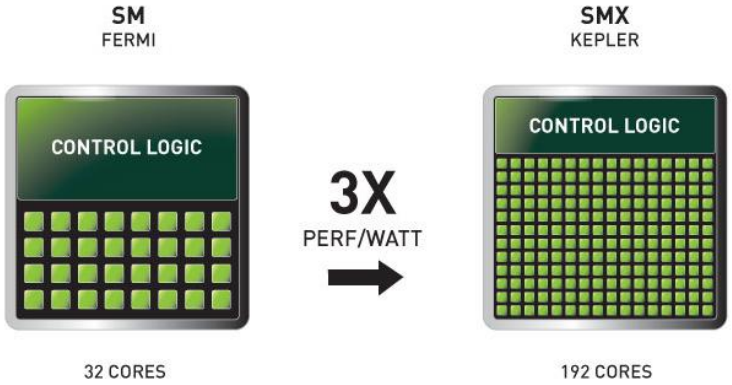
Ogni Streaming Multiprocessor è composto da:

- 32/48 core CUDA ognuno dei quali contiene una unità logica aritmetica (arithmetic logic unit - ALU) per interi e una per floating point (floating point unit - FPU) completamente *pipelined*
- 32^3 = 32768 registri (32-bit)
- 64KB shared-memory/cache L1 configurabili
  - 48-16KB o 16-48KB shared/L1 cache
- 16 unità load/store
- 4 Special Function Unit (SFU) per il calcolo di funzioni transcendenti
  (sin, sqrt, recp-sqrt,..)
- aderente allo standard IEEE 754-2008 a 32-bit e a 64-bit
  - **fused multiply-add** (FMA) in singola e doppia precisione





Novità Kepler

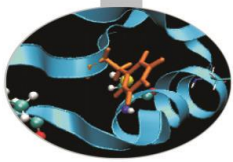


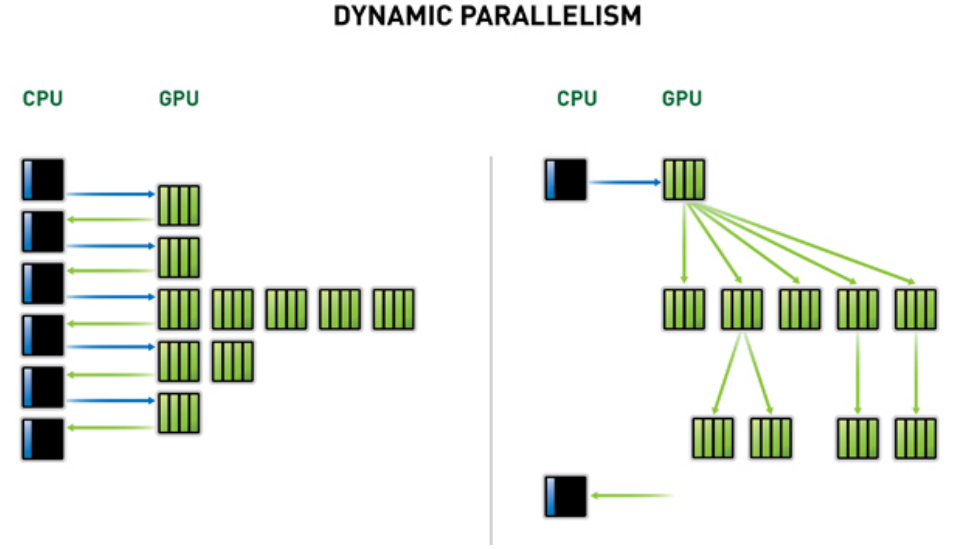
SMX

Nuovo streaming multiprocessor con più cores e una control logic unit ridotta

	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessor	32768	32768	65536	65536
Max Registers / Thread	63	63	63	255
Max Threads / Thread Block	1024	1024	1024	1024
Shared Memory Size Configurations (bytes)	16K 48K	16K 48K	16K 32K 48K	16K 32K 48K
Max X Grid Dimension	2^16-1	2^16-1	2^32-1	2^32-1
Hyper-Q	No	No	No	Yes
Dynamic Parallelism	No	No	No	Yes

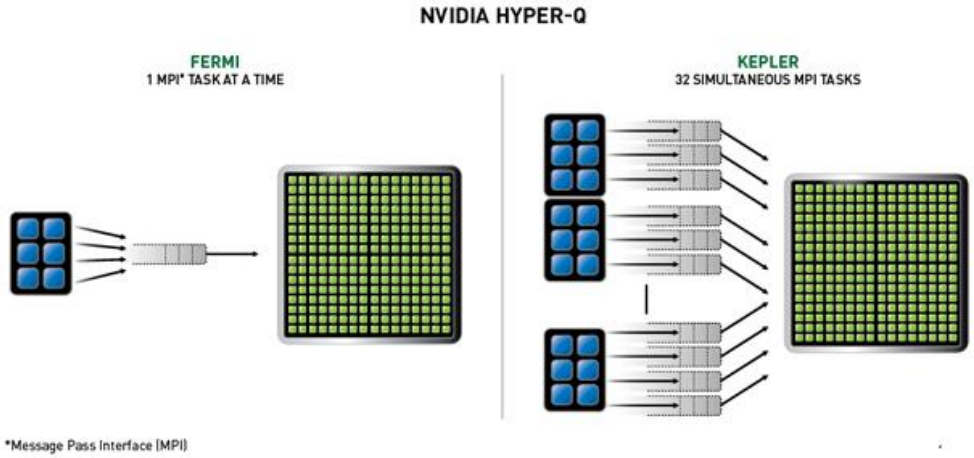
Novità Kepler





Dynamic Parallelism

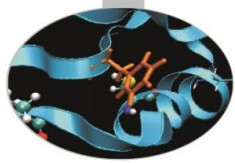
La gpu è in grado di lanciare kernels e generare nuovi threads in modo autonomo senza passare per l'host

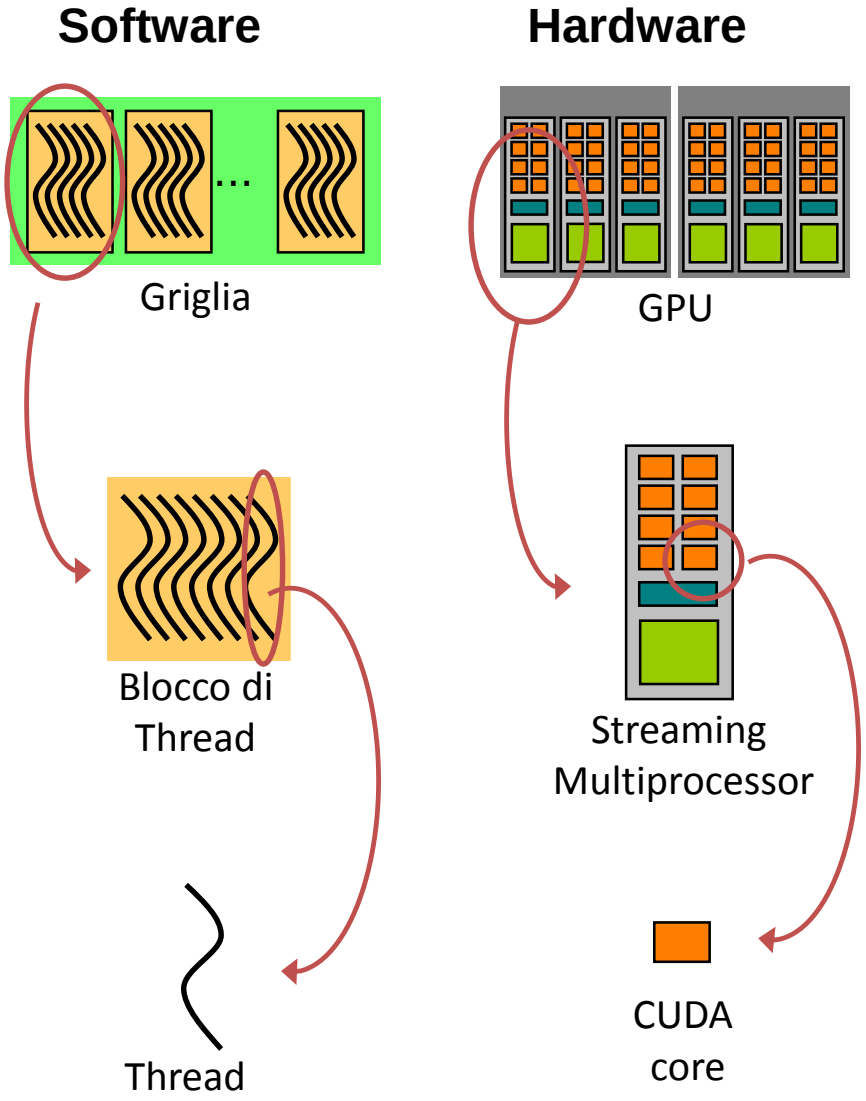


Hyper-Q

Consente, in modo molto più efficiente, ai vari core della CPU di utilizzare simultaneamente una singola GPU Kepler

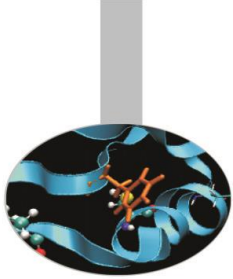
Il Modello di Esecuzione CUDA





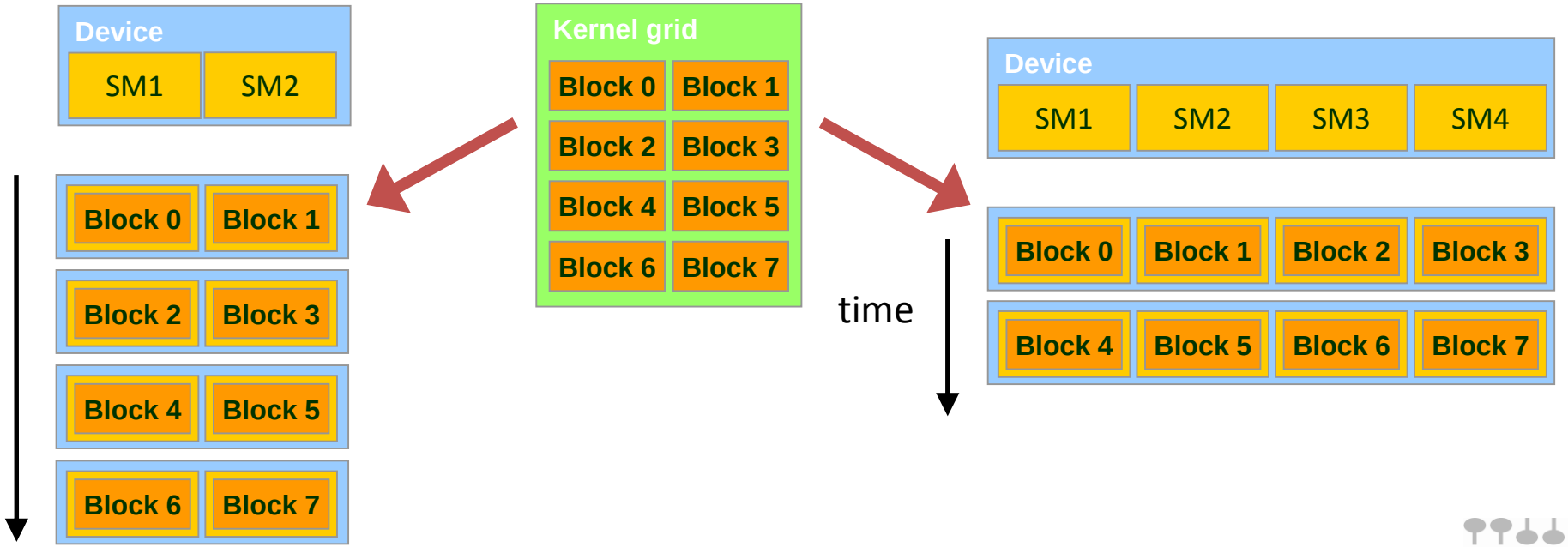
Al lancio di un kernel CUDA:

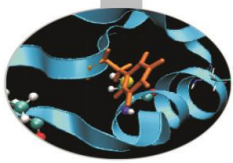
- ogni blocco della griglia è assegnato ciclicamente a uno Streaming Multiprocessor
  - ogni SM può ospitare fino a 8 blocchi: il limite è definito dalle risorse disponibili sul multiprocessore (memoria shared e registri) e da quelle richieste per l'esecuzione del kernel
  - eventuali bocchi non assegnati vengono allocati non appena un SM termina un blocco
  - non è possibile fare alcuna ipotesi sull'ordine di esecuzione dei blocchi! (**nessuna sincronizzazione**!)
- **i blocchi**, una volta assegnati**, non migrano**
- i *thread* in ciascun blocco sono raggruppati in gruppi di **32 *thread*** di indice consecutivo (detti ***warp***)
- **Il warp**, e non il singolo thread**, è l'unità fondamentale di esecuzione per lo scheduler dello Streaming Multiprocessor**
- lo scheduler seleziona da uno dei blocchi a suo carico uno dei warp pronto per l'esecuzione
- le istruzioni vengono dispacciate per *warp*
  - ogni CUDA core elabora uno dei 32 *thread* del warp



Scalabilità "Trasparente"

- un kernel CUDA scala su un qualsiasi numero di Streaming Multiprocessor
  - lo stesso codice può girare al meglio su architetture diverse

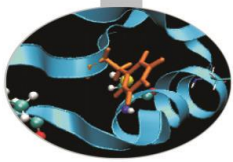




Occupancy

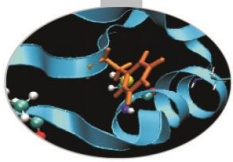
- Le istruzioni dei thread sono eseguite sequenzialmente, l'unico modo di nascondere la latenza è eseguire altri warp in modo da tenere occupato l'hardware.
- Occupancy=numero di warp che effettivamente sono eseguiti in maniera concorrente su un multiprocessore diviso per il numero massimo di warp che potrebbero essere eseguiti in maniera concorrente.
- Limitata dalle risorse utilizzate:
  - Registri
  - Memoria shared

Scelta della dimensione del blocco

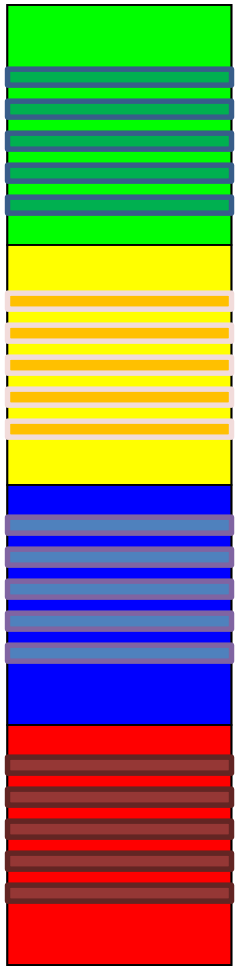


- Per un codice che utilizza più blocchi, quale è la dimensione ottimale del blocco?
  - Con un blocco 8X8 abbiamo 64 threads/blocco. Dato che ogni SM (Fermi) supporta 1536 threads, potremmo avere 24 blocchi. Purtroppo ogni SM supporta solo 8 blocchi, quindi solo 512 threads possono essere caricate su ogni SM!
  - Con un blocco 32X32 avremmo 1024 threads/blocco. Le GPU pre-Fermi hanno un limite di 512 thread per blocco. Con le Fermi potremmo avere un massimo di un blocco per SM!
  - Con un blocco 16X16 abbiamo 256 threads/blocco. Con 6 blocchi possiamo avere piena occupazione del SM (1536 threads) a meno di altri vincoli.

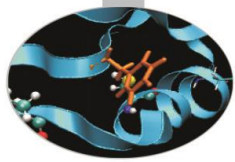
Gestione dei Registri



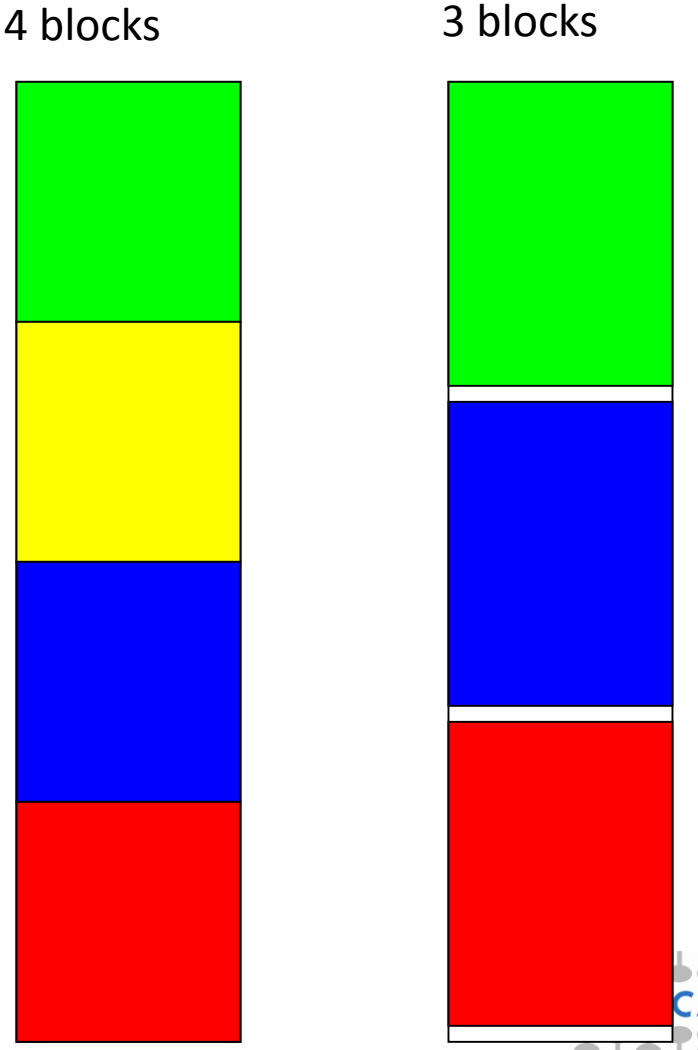
- Ogni SM ha 32768 registri (architettura Fermi)
  - i registri sono partizionati dinamicamente tra tutti i blocchi assegnati ad un SM
  - i thread di un blocco accedono SOLO ai registri che gli sono stati assegnati



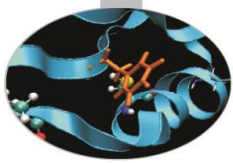
Considerazioni sul numero di registri



- Se un blocco ha dimensioni 32x8 thread ed il kernel usa 30 registri, quanti blocchi possono girare per SM?
  - ogni blocco richiede 30x32x8 = 7680 registri
  - 32768/7680 = **4** + "resto"
  - solo 4 blocchi (invece di 8) possono girare su un SM
- Cosa succede se, cambiando implementazione del kernel, aumenta l'uso dei registri del 10%?
  - ogni blocco ora richiede 33x32x8 = 8448 registri
  - 32768/8448 = **3** + "resto"
  - solo 3 blocchi!
    - 25% di riduzione del parallelismo!

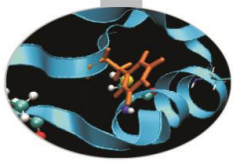


Control Flow



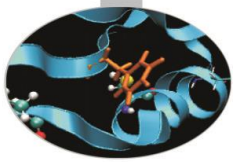
- È fondamentale evitare divergenze del flusso di esecuzione
  - quando thread facenti parte dello stesso warp seguono diversi percorsi l'esecuzione è serializzata
    - I percorsi di esecuzione presi dai thread sono seguiti uno per volta fino a quando non si esauriscono
  - warp differenti possono eseguire codice diverso senza penalità di performance

- Attenzione in particolare ai casi di divergenza quando la condizione di branch è una funzione del thread ID
  - Esempio con divergenza:
    - `If (threadIdx.x > 2) { }`
    - Crea due diversi percorsi di esecuzione per i threads dello stesso warp
    - I threads 0, 1 e 2 seguono un percorso di esecuzione diverso dagli altri thread del primo warp (3, 4, …)
  - Esempio senza divergenza:
    - `If (threadIdx.x / WARP_SIZE > 2) { }`
    - Anche in questo caso si hanno due diversi percorsi di esecuzione per I threads del blocco, ma in ogni warp tutti i thread seguono lo stesso percorso



- Gestione errore
- Misura prestazioni
  - Eventi
  - Libreria CUDA utility
  - Libreria CUDA helper

Segnalazione degli errori CUDA alla CPU



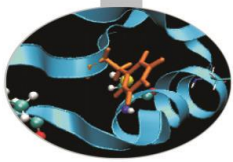
- Tutte le chiamate API CUDA ritornano un codice di errore di tipo cudaError_t
  - il codice cudaSuccess significa che non si sono verificati errori
  - i kernel sono asincroni e void (non ritornano codice di errore)
  - le funzioni asincrone resituiscono un errore che si riferisce solo alla fase di chiamata da host

```
char* cudaGetErrorString(cudaError_t code)
```

  - ritorna una stringa di caratteri (NULL-terminated) che descrive l'errore a partire dal codice passato

```
cudaError_t cerr = cudaMalloc(&d_a,size);


if (cerr != cudaSuccess)
fprintf(stderr, "%s\n", cudaGetErrorString(cerr));
```

Segnalazione degli errori CUDA alla CPU

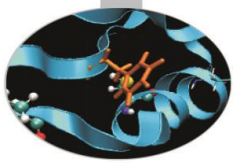


- L'errore viene comunque mantenuto in una variabile interna modificata dalle funzioni in errore

  `cudaError_t cudaGetLastError(void)`

  - ritorna il codice di stato della variabile interna (cudaSuccess o altro)
  - resetta la variabile interna: quindi l'errore che dà cudaGetLastError può riferirsi a una qualunque delle funzioni dalla precedente chiamata di cudaGetLastError stesso
  - è necessaria prima una sincronizzazione chiamando **`cudaDeviceSynchronize()`** in caso di controllo di funzioni asincrone (e.g., kernel) che ha sostituito la funzione **`cudaThreadSynchronize()`** deprecata da cuda 4.0 in poi.

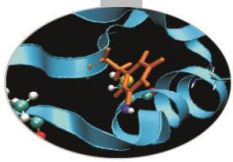
```
cudaError_t cerr;
...
cerr = cudaGetLastError(); // reset internal state
kernelGPU<<<dimGrid,dimBlock>>>(arg1,arg2,...);
cudaDeviceSynchronize();
cerr = cudaGetLastError();
if (cerr != cudaSuccess)
fprintf(stderr, "%s\n", cudaGetErrorString(cerr));
```



Timing del codice

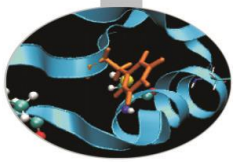
- Le chiamate CUDA e l'esecuzione dei kernel possono essere misurate usando usando i timer della CPU o usando i timer GPU.
- CPU timer:
  - Molte funzioni CUDA API sono asincrone e ritornano il controllo alla CPU prima del loro completamento .
  - E' necessario sincronizzare il thread della CPU con la GPU con la funzione `cudaDeviceSyncronize()` prima di iniziare e di terminare il timing:

```
cudaDeviceSynchronize();
clock_t begin = clock();

MyKernel<<< x, y >>>( z, w );

cudaDeviceSynchronize();
clock_t end = clock();

// Print time difference: ( end - begin )
```



API CUDA: gli “eventi”

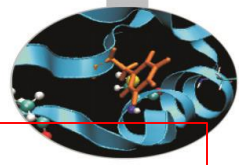
- Gli eventi sono una sorta di marcatori che possono essere utilizzati nel codice per:
  - misurare il tempo trascorso (elapsed) durante l’esecuzione di chiamate CUDA
    (precisione a livello di ciclo di clock)
  - bloccare la CPU fino a quando le chiamate CUDA precedenti l’evento sono state completate
    (maggiori dettagli su chiamate asincrone in seguito...)



API CUDA: gli "eventi"

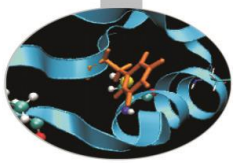
- Le principali fuzioni per la gestione degli eventi sono:
  - cudaError_t **cudaEventCreate** (cudaEvent_t *event) // **crea l'evento**
  - cudaError_t **cudaEventRecord** (cudaEvent_t event, cudaStream_t stream) // **registra l'evento**
  - cudaError_t **cudaEventSynchronize** (cudaEvent_t event) // **attende il completamento dell'evento**
  - cudaError_t **cudaEventElapsedTime** (float *ms, cudaEvent_t start, cudaEvent_t end) // **calcola il tempo fra due eventi in millisecondi con una risoluzione di circa 0.5 microsecondi**
  - cudaError_t **cudaEventDestroy** (cudaEvent_t event) // **distrugge un evento**

Eventi per misurare il tempo



```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start);
...
kernel<<<grid, block>>>(...);
...
cudaEventRecord(stop);
cudaEventSynchronize(stop);

float elapsed;
// tempo tra i due eventi
// in millisecondi
cudaEventElapsedTime(&elapsed, start, stop);

cudaEventDestroy(start);
cudaEventDestroy(stop);
```

```
integer ierr
type (cudaEvent) :: start, stop
real elapsed

ierr = cudaEventCreate(start)
ierr = cudaEventCreate(stop)

ierr = cudaEventRecord(start, 0)
...
call kernel<<<grid,block>>>()
...
ierr = cudaEventRecord(stop, 0)
ierr = cudaEventSynchronize(stop)

 ierr = cudaEventElapsedTime&
      (elapsed,start, stop)

ierr = cudaEventDestroy(start)
ierr = cudaEventDestroy(stop)
```



Timing del codice

CUDA Command Line Profiler

Cuda dispone di un profiler testuale che permette di individuare il tempo speso nel kernel, il numero di accessi alla memoria, il numero di chiamate etc etc.

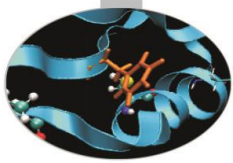
Come utilizzarlo?

- Configurare l'ambiente:

```
export CUDA_PROFILE=1   //cuda 4
export CUDA_PROFILE_CSV=1 // cuda 4
export CUDA_PROFILE_LOG=result.csv // cuda 4
export CUDA_PROFILE_CONFIG=cudaprofile.txt // cuda 4
   (optional)
```

Da **CUDA 5** il prefisso "**CUDA**" è stata sostituito da "**COMPUTE**"

Timing del codice



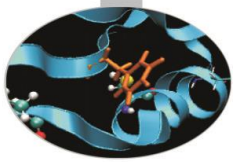
- **Lanciare l'eseguibile da profilare**
- **Analizzare i risultati prodotti (result.csv) nella cartella di lancio.**

Esempio (vectAdd.cu)

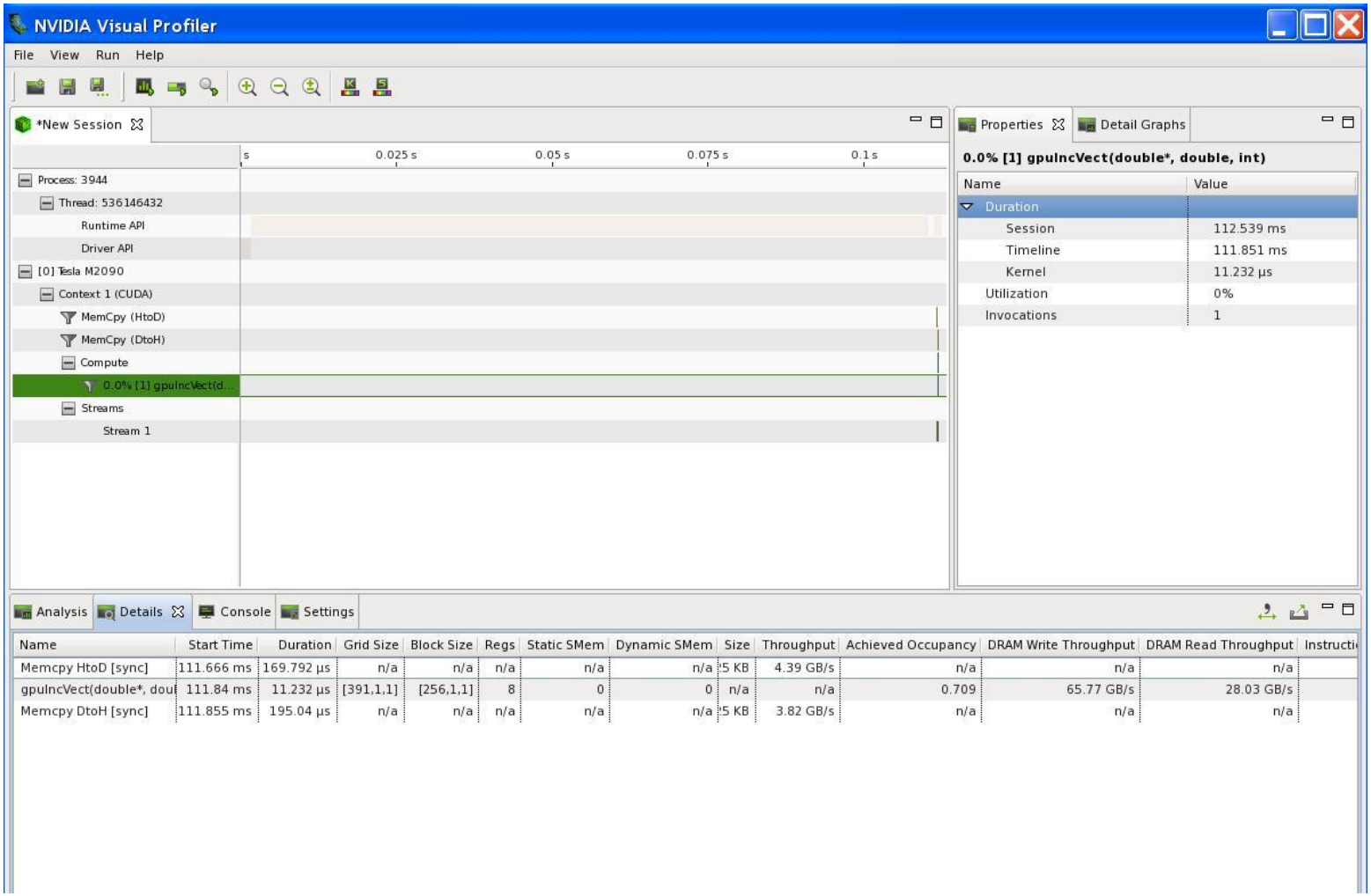
```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 Tesla M2050
# CUDA_PROFILE_CSV 1
# TIMESTAMPFACTOR fffff6e0019dd990
method                  gputime cputime    occupancy
memcpyHtoD,              35.296,115.000
memcpyHtoD,              35.264,109.000
_Z6VecAddPKfS0_Pfi,  7.232, 22.000,     1.000
memcpyDtoH,             34.272,246.000
```

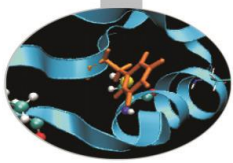
- **Il tempo è misurato in microsecondi in singola precisione**

Timing del codice



- Utilizzare il profiler grafico *nvvp*



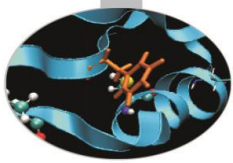


La libreria CUDA utililty

- Il pacchetto SDK contiene, oltre a numerosi esempi, anche alcune librerie di utilità

- Fino a cuda 4 la libreria *cutils* forniva alcune funzionalità tra cui:
  - il parsing degli argomenti di linea di comando
  - confronto di array tra GPU e CPU
  - timers
  - MACROs per il controllo dei codici di errore

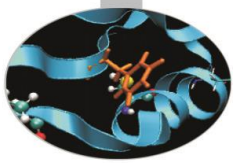
- si trovava in: `NVIDIA_CUDA_SDK/C/common/`



La libreria CUDA utililty

- controllo dell'errore: `CUDA_SAFE_CALL(call)`
  - wrapper alle chiamate API del runtime CUDA
    - controlla il codice di errore di ritorno
    - esce con un messaggio di errore comprensibile

- timing:
  - `cut(Create|Destroy)Timer(name)`:
    - crea o distrugge un timer cumulativo con il nome *name*
  - `cut(Start|Stop|Reset)Timer(name)`:
    - Avvia, ferma o resetta il timer di nome *name*
  - `cutGet[Average]TimerValue(name)`:
    - elapsed [average] time in millisecondi del timer *name*

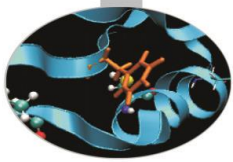


La libreria CUDA helper

- Da cuda 5 la libreria CUDA utility è stata sostituita dalla libreria CUDA helper
- controllo dell'errore: checkCudaErrors `(call)`
  - wrapper alle chiamate API del runtime CUDA
    - controlla il codice di errore di ritorno
    - esce con un messaggio di errore comprensibile

- timing:
  - `sdk(Create|Delete)Timer(name);`
    - crea o distrugge un timer cumulativo con il nome *name*
  - `sdk(Start|Stop|Reset)Timer(name);`
    - Avvia, ferma o resetta il timer di nome *name*
  - `sdkGet[Average]TimerValue(name);`
    - elapsed [average] time in millisecondi del timer *name*
- Si trova in : samples/common/inc

Performance



- Quale metrica utilizziamo per misurare le performance?

- Flops (numero di operazioni floating point per secondo):

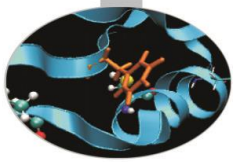
$$\text{flops} = \frac{N_{\text{OPERAZIONI FLOATING POINT}}\ (\text{flop})}{\text{Elapsed Time (s)}}$$

- Più comodo di solito usare Mflops, Gflops,... Interessante valutare questa grandezza a confronto con il valore di picco della macchina. Per schede NVIDIA Fermi valori di picco: 1Tflops in singola precisione, 0.5 Tflops in doppia precisione

- Bandwidth (larghezza di banda): quantità di dati trasferiti al secondo

$$\text{bandwidth} = \frac{\text{Quantità di dati trasferiti (byte)}}{\text{Elapsed Time (s)}}$$

- Di solito si usano GB/s o. Nel contesto GPU Computing possibili trasferimenti tra diverse aree di memoria (HosttoDevice, DevicetoHost, DevicetoDevice)

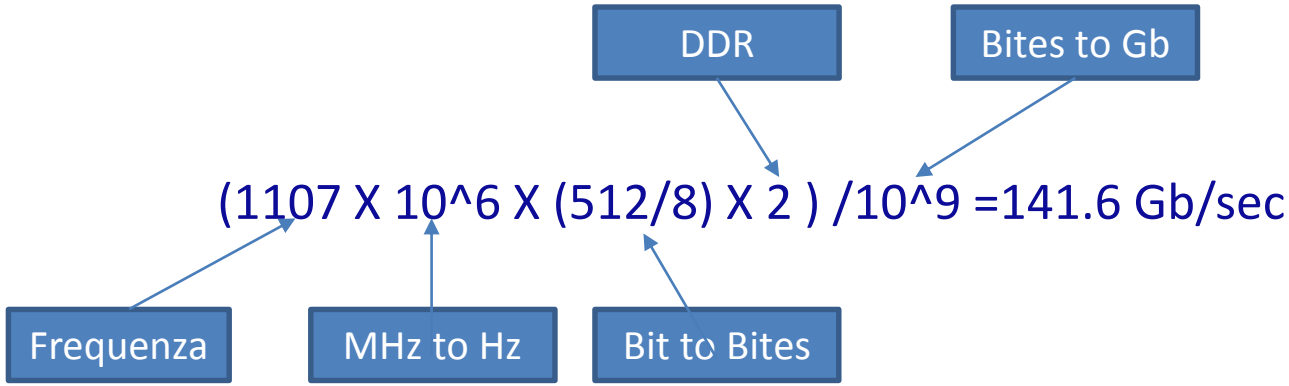


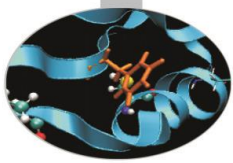
Misura della Bandwidth

- **Bandwidth teorica**

La bandwidth teorica può essere calcolata a partire dalle specifiche dell'hardware.

NVIDIA GeForce GTX 280 DDR RAM, 1.107 MHz, 512-bit memory interface:





Misura della Bandwidth

- **Bandwidth Effettiva**

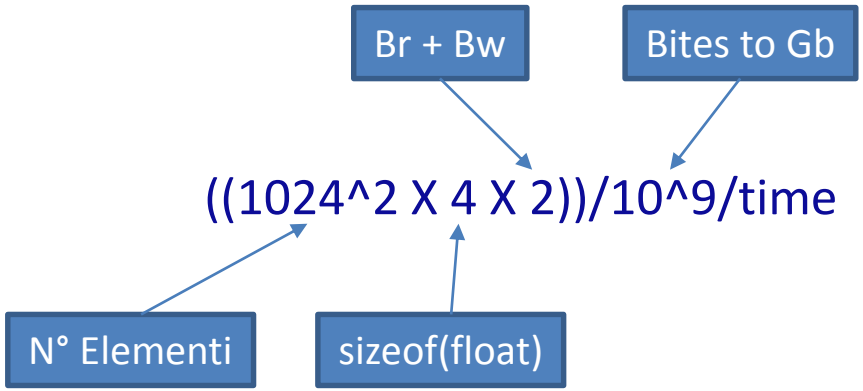
E' calcolata misurando l'attività effettiva del programma e i dati effettivamente usati.

Bandwidth Effettiva= ((Br+Bw)/10^9)/time

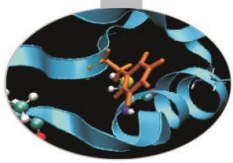
Br = #bytes read

Bw =#bytes write

Bandwidth effettiva per la copia di una matrice 1024 X 1024 di float:



Esercitazione



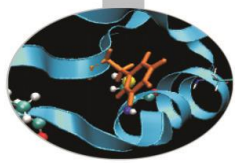
Misura della bandwidth

- cd Esercizi/Bandwidth: bandwidthTest.cu
  - cudaEventCreate(&start) ;
  - cudaEventCreate(&stop) ;
  - cudaEventRecord(start, 0);
  - cudaMemcpy(dst,src,memSize,cudaMemcpyHostToDevice);
  - cudaEventRecord(stop, 0);
  - cudaEventElapsedTime(&elapsedTimeInMs, start, stop);

- compilare: `compile.sh`
- lanciare

```
./bandwidthTest --mode=range --start=<B> --end=<B>
                --increment=<B>
```

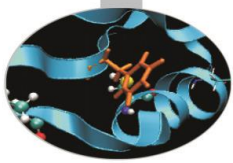
Misura della bandwidth: esercitazione



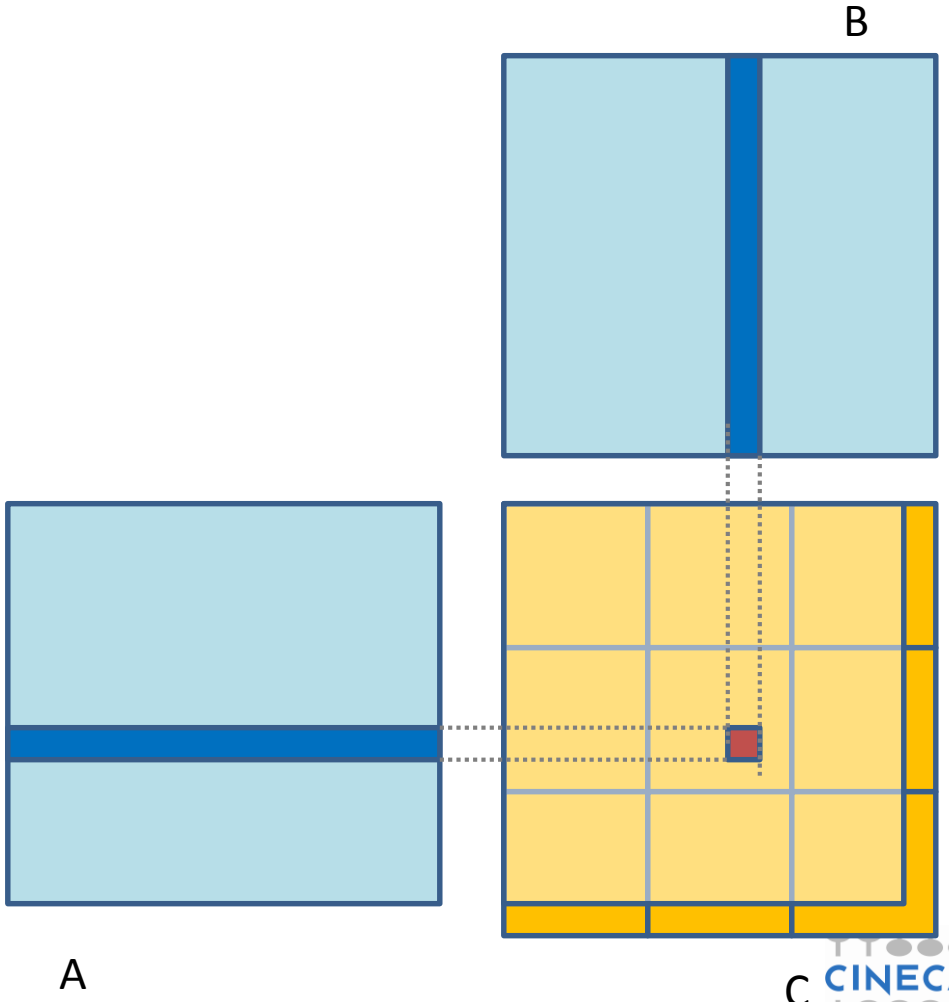
- cd Esercizi/Bandwidth
- Compilare
- riportare la bandwidth misurata all'aumentare del pacchetto dati trasferito da Host a Device, da Device to Host, da Device to Device

Size (MB)	HtoD	DtoH	DtoD
1			
10			
100			
1024			

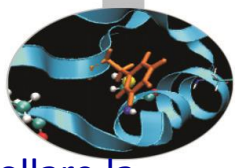
Prodotto matrice-matrice usando memoria globale



- Per semplicità consideriamo matrici quadrate di ordine N
- Ogni thread calcola un elemento della matrice C
- Dimensionamento griglia di thread come di consueto (distinguere il caso di N multiplo della dimensione del blocco).
- Thread attivi solo se corrispondono ad elementi di matrice C.
- Ogni thread legge dalla memoria globale scorrendo gli elementi di una riga di A e di una colonna di B, li moltiplica e accumula il loro prodotto in C



Matrice-matrice in memoria globale: errore



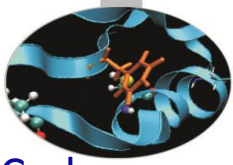
- Implementare il trattamento dell'errore sfruttando la funzione cudaGetLastError() per controllare la corretta esecuzione del kernel
- Attenzione:
  - ai tipi della variabile di errore: "cudaError_t" in C e "integer" in fortran
  - occorre chiamare la funzione anche prima del kernel per azzerare la variabile di errore
  - prima della seconda chiamata è necessaria una sincronizzazione perchè il kernel è asincrono

```
mycudaerror=cudaGetLastError() ;
          <chiamata kernel>
cudaDeviceSynchronize() ;
mycudaerror=cudaGetLastError() ;
if(mycudaerror != cudaSuccess)
fprintf(stderr,"%s\n",
cudaGetErrorString(mycudaerror)) ;
```

```
mycudaerror=cudaGetLastError()
          <chiamata kernel>
ierr = cudaDeviceSynchronize()
mycudaerror=cudaGetLastError()
if(mycudaerror .ne. 0) write(*,*)  &
  'Errore in kernel: ',mycudaerror
```

- Provare a lanciare un kernel con blocchi di thread più grandi del limite (32x32). Vedere quale tipo di errore si rileva.

Matrice-matrice in memoria globale: performance



- Implementare la misura del tempo del kernel tramite eventi e funzioni di timing Cuda
- Fasi:
  - Creazione eventi start e stop: *cudaEventCreate*
  - Record dell'evento start: *cudaEventRecord*
  - Chiamata kernel, eventualmente con controllo dell'errore
  - Record dell'evento stop: *cudaEventRecord*
  - Sincronizzazzione eventi: *cudaEventSynchronize*
  - Misura elapsed time tra eventi: *cudaEventElapsedTime*
  - Distruzione eventi: *cudaEventDestroy*

- Calcolare i Gflops (floating point operations per second) considerando che l'algoritmo richiede $2N^3$ operazioni

	C	fortran
Gflops		

- Idee per l'ottimizzazione?