Introduction to Scientific Programming using GPGPU and CUDA

Day 1

Luca Ferraro
l.ferraro@cineca.it

Sergio Orlandini
s.orlandini@cineca.it

Agenda

I° giorno:

- Introduzione al GPU Computing
- Il modello CUDA per GPGPU
- Architetture CUDA
- Altri modelli GPGPU

--- pausa pranzo ---

- Controllo errore
- Misura performance
- Esercitazione

II° giorno:

- Gerarchia della memoria GPU
- Concorrenza
- Interazione CPU-GPU
- Ambiente multi-GPU
- Esercitazione

--- pausa pranzo ---

- Tools CUDA-Toolkit
- Librerie CUDA enabled
- Esercitazione

Cosa è una GPU

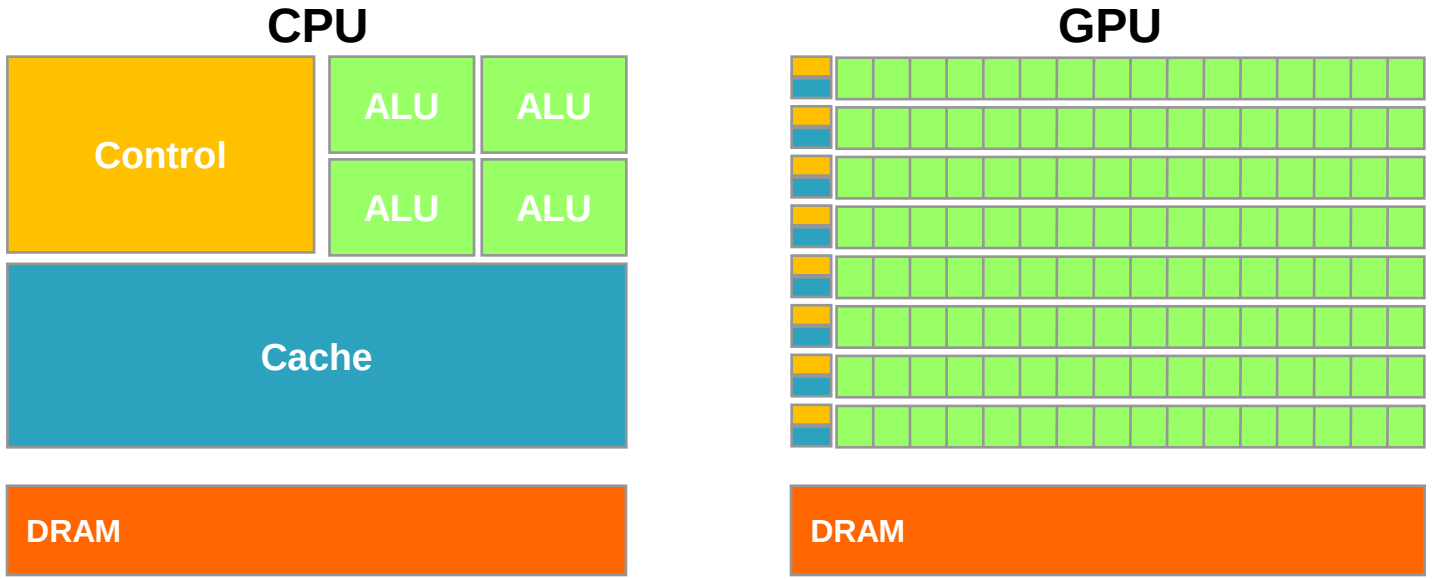
- **G**raphics **P**rocessing **U**nit
  - microprocessore altamente parallelo (*many-core*) dotato di memoria privata ad alta banda
- specializzate per le operazioni di rendering grafico 3D
- sviluppatesi in risposta alla crescente domanda di potenza grafica nel mercato dei videogiochi (grafica 3D ad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

Le operazioni di una GPU

- una scena 3D è composta di oggetti rappresentati da poligoni disposti in un mondo astratto (3D world space)
  - i poligoni sono rappresentati dalle coordinate dei loro vertici nello spazio 3D (*vertex*)
- le coordinate degli oggetti del mondo 3D sono trasformate secondo un riferimento dato dalla posizione e dall'orientazione del punto di vista della scena (*virtual camera*).
  Le trasformazioni sono realizzate mediante prodotti di matrici:
  - le trasformazioni più comuni sono traslazioni, scaling, rotazioni e proiezioni.
  - queste trasformazioni debbono essere applicate ogni volta che cambia il punto di vista della scena.
- dopo l'applicazione del punto di vista, vengono eseguite le operazioni di trasformazione di riempimento dei poligoni (*rendering*)
  - applicazioni dei colori, texture
  - filtraggi e altri effetti di "abbellimento" (anti-alias, smoothing, ecc)
  - applicazione delle luci e calcolo ombre
- la scena 3D deve essere trasformata in uno spazio piano di coordinate 2D (pixels) per creare l'immagine da visualizzare sullo schermo (*rastering*) .
- una tipica scena complessa è composta da circa 1M di vertici e da circa 6M pixel
- per dare l'impressione di una scena fluida, tutte queste operazioni devono essere applicate almeno 25 volte al secondo

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)
  - limitata località spaziale (parallelismo a granularità fine) e alta intensità aritmetica che nasconde le latenze di load/store (cache ridotta)

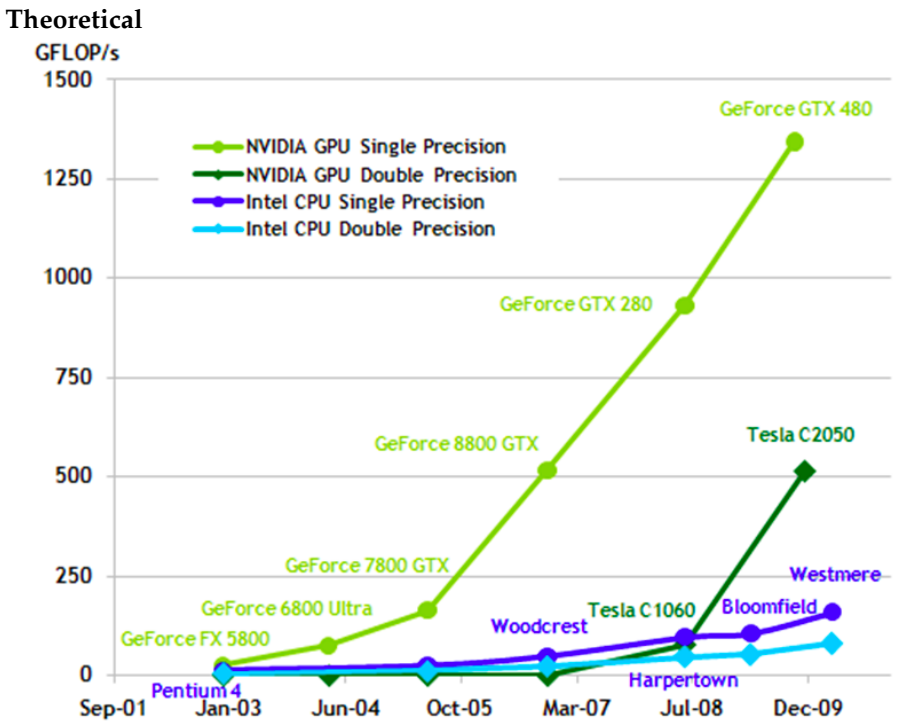


"The GPU devotes more transistors to Data Processing"
(NVIDIA CUDA Programming Guide)

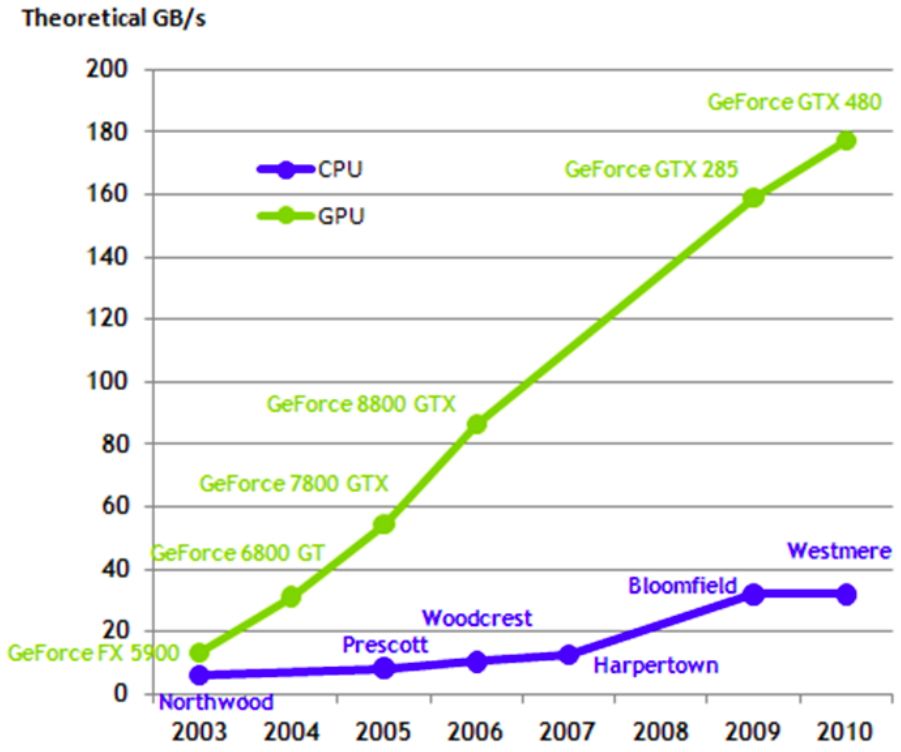
The concurrency revolution

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

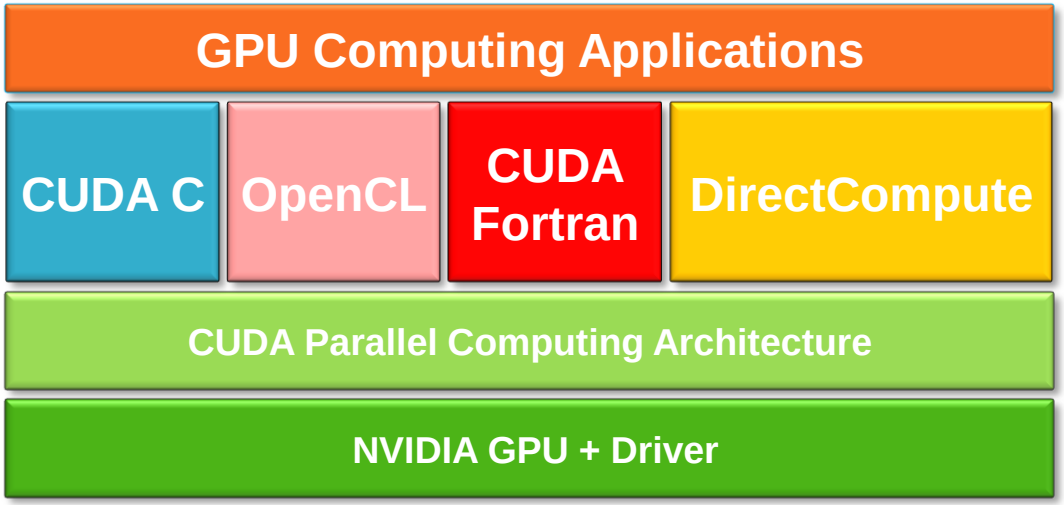
GPGPU (General Purpose GPU) e GPU computing

- Perché non utilizzare la potenza delle GPU per applicazioni diverse dall'elaborazione grafica?
- La GPU è vista come coprocessore matematico con una propria area di memoria (device memory)
- Evoluzione dei chip grafici: da soluzioni specializzate per elaborazioni di pixel e vertici (per rendering e calcolo di geometria) a motori di calcolo altamente programmabili

A General-Purpose Parallel Computing Architecture

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



CUDA APIs

- 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

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, ….)

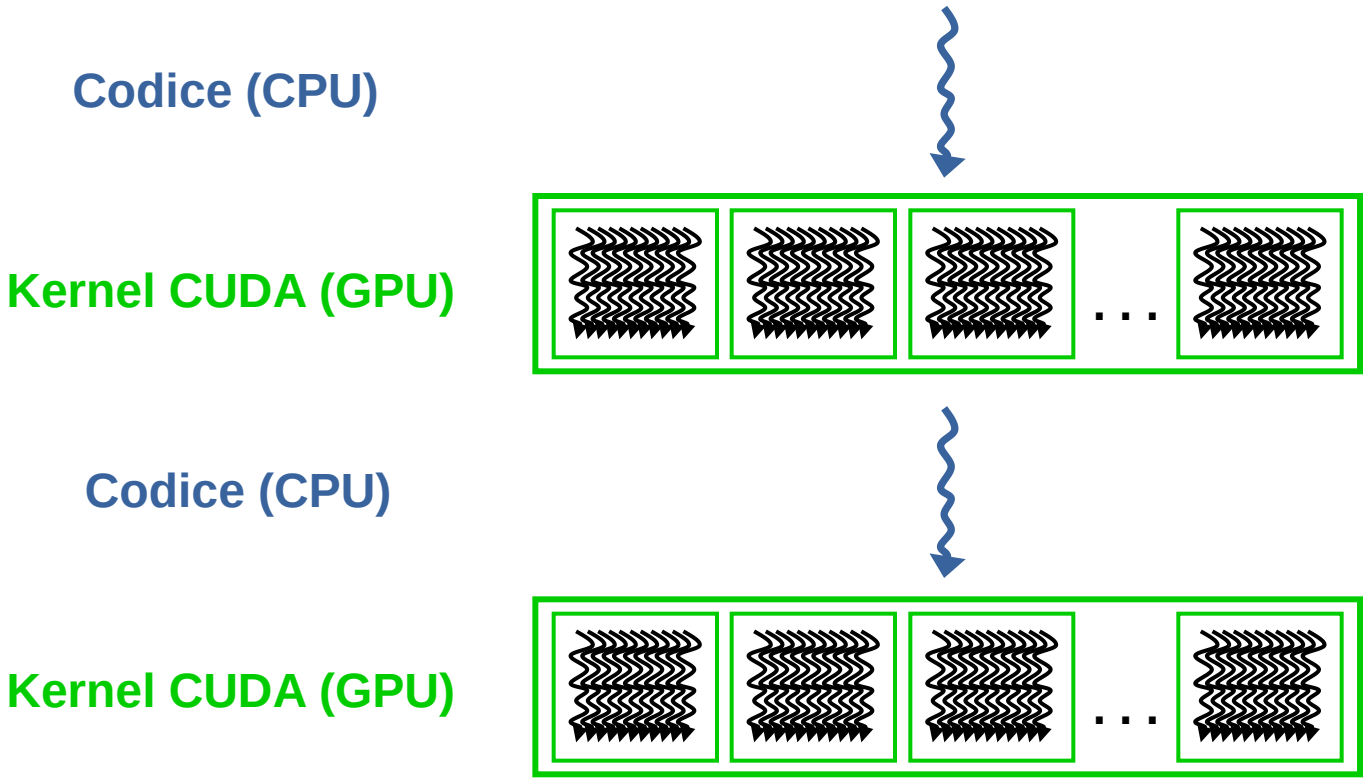
- Modello CUDA e threads
  - co-processore/memoria
  - generazione threads
  - granularità fine
- Come scrivere un codice CUDA
  - 4 passi per iniziare
  - esempio Vector-Vector Add
  - gestire la memoria
    e il trasferimento dati CPU-GPU
  - scrivere e lanciare un kernel CUDA

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



Multi-thread in azione

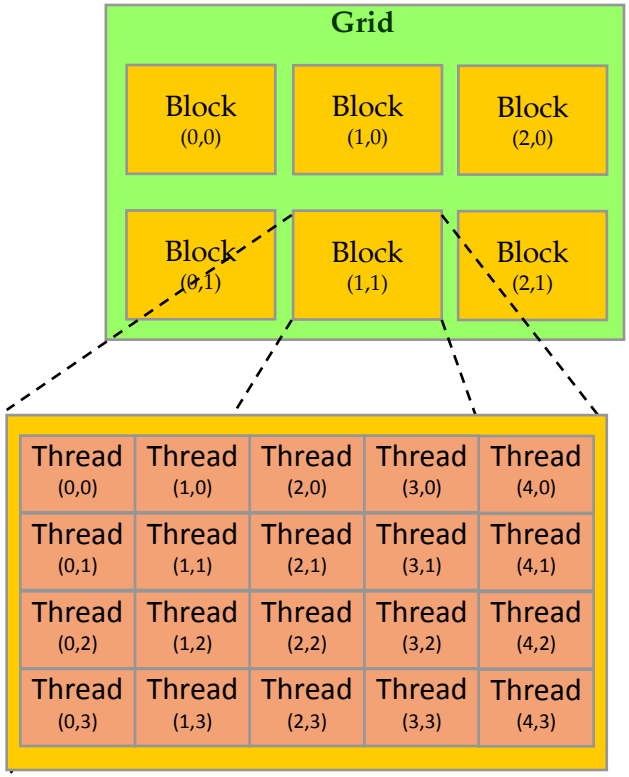
```
int main(int argc, char *argv[]) {
  int i;
  const int N = 1000;
  double u[N], v[N], z[N];

  initVector (u, N, 1.0);
  initVector (v, N, 2.0);
  initVector (z, N, 0.0);

  printVector (u, N);
  printVector (v, N);

  // z = u + v
  for (i=0; i<N; i++)
      z[i] = u[i] + v[i];

  printVector (z, N);

  return 0;
}
```

```
void gpuVectAdd( const double *u,
  const double *v, double *z)
{  // use GPU thread id as index
   i = getThreadIdx();
   z[i] = u[i] + v[i];
}
```

```
int main(int argc, char *argv[]) {

  ...

  // z = u + v
  {
      // run on GPU with N threads
      gpuVectAdd(u, v, z);
      // close threads
  }

  ...
}
```

CUDA Threads

- quando si esegue un kernel CUDA sul device GPU bisogna specificare il numero di thread che devono essere lanciati
- i thread sono raggruppati in blocchi
  - ogni *thread* all'interno di un blocco può essere identificato da un set di coordinate cartesiane 1D, 2D, 3D
- i blocchi di thread costituiscono gli elementi di una griglia
  - ogni blocco all'interno della griglia può essere identificato mediante un set di coordinate cartesiane bidimensionale
- in ogni CUDA kernel è possibile utilizzare le seguenti variabili per identificare le coordinate del thread corrente e del blocco



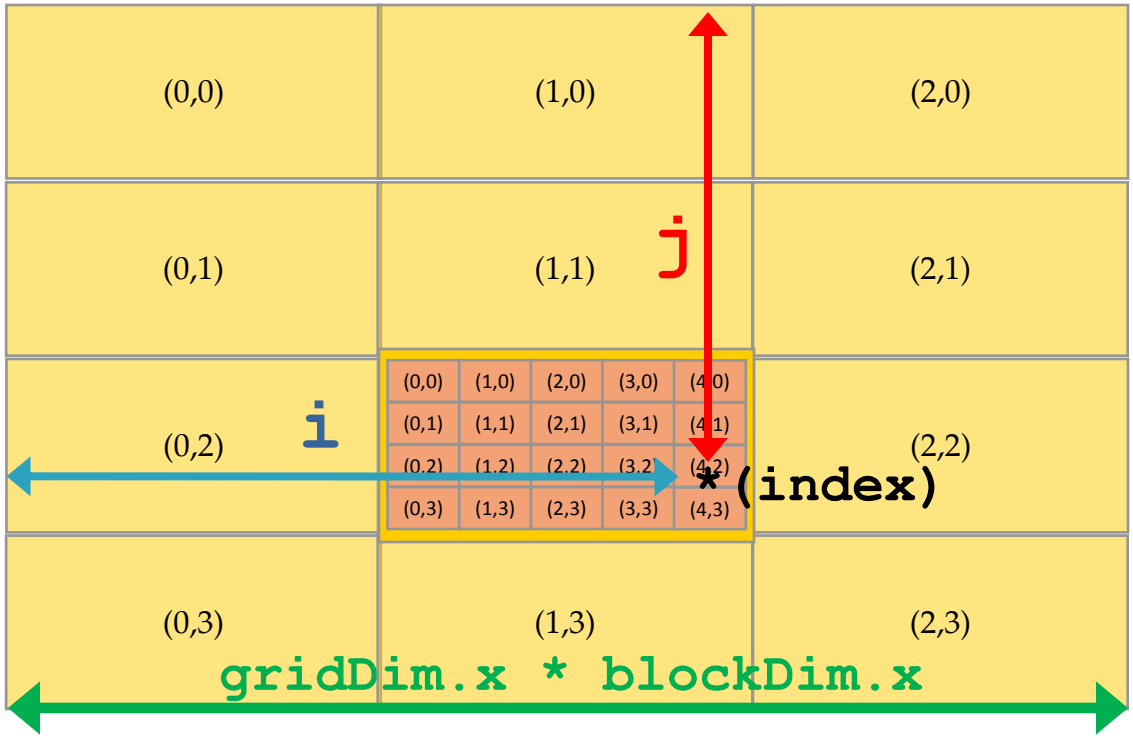
`threadIdx:`
thread coordinates inside a block

`blockIdx:`
block coordinates inside the grid

`blockDim:`
block dimensions in thread units

`gridDim:`
grid dimensions in block units

CUDA Thread Grid



`threadIdx`:
thread coordinates inside a block

`blockIdx`:
block coordinates inside the grid

`blockDim`:
block dimensions in thread units

`gridDim`:
grid dimensions in block units

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

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

4 Passi per un posting a 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. implementare il corpo computazionale in un kernel CUDA
   a) scrivere l'algoritmo che elabora il singolo elemento
   b) specificare la modalità di esecuzione del kernel CUDA
4. 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. trasferire i risultati dal *device* all'*host*

Somma di Vettori

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

```c
int main(int argc, char *argv[]) {
  int i;
  const int N = 1000;
  double u[N], v[N], z[N];

  initVector (u, N, 1.0);
  initVector (v, N, 2.0);
  initVector (z, N, 0.0);

  printVector (u, N);
  printVector (v, N);

  // z = u + v
  for (i=0; i<N; i++)
      z[i] = u[i] + v[i];

  printVector (z, N);

  return 0;
}
```

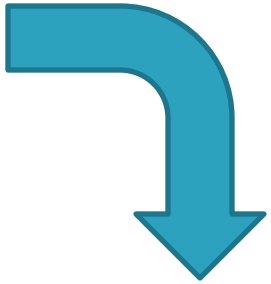
```fortran
program vectoradd
integer :: i
integer, parameter :: N=1000
real(kind(0.0d0)), dimension(N):: u, v, z

call initVector (u, N, 1.0)
call initVector (v, N, 2.0)
call initVector (z, N, 0.0)

call  printVector (u, N)
call  printVector (v, N)

! z = u + v
do i = 1,N
    z(i) = u(i) + v(i)
end do

call printVector (z, N)

end program
```

Somma di Vettori

3. Implementare il Kernel CUDA
a) scrivere l'algoritmo per il singolo elemento

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

```
const int N = 1000;
double u[N], v[N], z[N];

// z = u + v
for (i=0; i<N; i++)
    z[i] = u[i] + v[i];
```



```
void gpuVectAdd (const double *u, const double *v, double *z, int N)
{
  // index is a unique identifier of each GPU thread
  int index =  ....  ;
  if (index < N)
     z[index] = u[index] + v[index];
}
```

Somma di Vettori

3. Implementare il Kernel CUDA
b) specificare la modalità di esecuzione

La scelta delle dimensioni della griglia è dettata dal problema e dal *mapping* che si utilizza nel kernel CUDA tra l'indice degli elementi da elaborare e gli identificativi dei thread CUDA

- nel caso di un problema lineare come l'elaborazione degli elementi di un array, possiamo scegliere:

  - blocchi di thread monodimensionali (1D)
  - griglia monodimensionale di blocchi (1D)
- ogni *thread* della griglia elabora in parallelo un solo elemento del vettore
  - 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)

Somma di Vettori

3. Implementare il Kernel CUDA
b) specificare la modalità di esecuzione

execution configuration: estensione CUDA per il lancio dei kernel su GPU:

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

specifica la griglia dei thread:

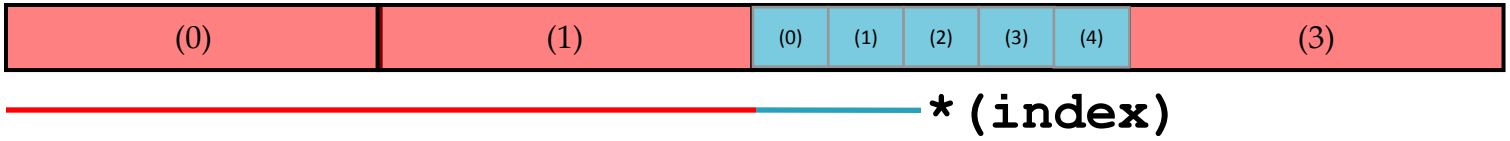
- **`numBlocks:`** specifica le dimensioni della griglia in termini di numero di blocchi lungo ogni dimensione
- **`numThreads:`** specifica la dimensione di ciascun blocco in termini di numero di thread lungo ciascuna direzione

```
dim3 numThreads(32);
dim3 numBlocks( ( N + numThreads - 1 ) / numThreads.x );
gpuVectAdd<<<numBlocks, numThreads>>>( u_dev, v_dev, z_dev, N );
```

```
type(dim3) :: numBlocks, numThreads
numThreads = dim3( 32, 1, 1 )
numBlocks = dim3( (N + numThreads%x - 1) / numThreads%x, 1, 1 )
call gpuVectAdd<<<numBlocks,numThreads>>>( u_dev, v_dev, z_dev, N )
```

Somma di Vettori

3. Implementare il Kernel CUDA




```
__global__ void gpuVectAdd (const double *u, const double *v, double *z, int N)
{
    // index is a unique identifier of each GPU thread
    int index =  blockIdx.x * blockDim.x + threadIdx.x  ;
    if (index < N)
      z[index] = u[index] + v[index]
}
```

__global__
qualificatore che definisce un kernel GPU

le funzioni con qualificatore __global__

- possono essere invocate solo dall'*host*
- devono essere invocate con una *execution configuration*
- devono avere un tipo di ritorno *void*
- sono asincrone (ritornano il controllo all'*host* prima del loro completamento)

Somma di Vettori

3. Implementare il Kernel CUDA

```
module vector_algebra_cuda
use cudafor
contains
attributes(global) subroutine gpuVectAdd (u, v, z, N)
  implicit none
  integer, intent(in), value :: N
  real, intent(in) :: u(N), v(N)
  real, intent(inout) :: z(N)
  integer :: i

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

   if (i .gt. N) return

   z(i) = u(i) + v(i)
end subroutine
end module vector_algebra_cuda
```

Somma di Vettori

3. Implementare il Kernel CUDA

```
attributes(global) subroutine gpuVectAdd (u, v, z, N)
  ...
end subroutine

program vectorAdd
use cudafor
implicit none
interface
  attributes(global) subroutine gpuVectAdd (u, v, z, N)
    integer, intent(in), value :: N
    real, intent(in) :: u(N), v(N)
    real, intent(inout) :: z(N)
    integer :: i
  end subroutine
end interface
  ...

end program vectorAdd
```

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

Somma di Vettori

4. 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 *u_dev, *v_dev, *z_dev;

cudaMalloc((void **)&u_dev, N * sizeof(double));
cudaMalloc((void **)&v_dev, N * sizeof(double));
cudaMalloc((void **)&z_dev, N * sizeof(double));
```

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

```
real(kind(0.0d0)), device, allocatable, dimension(:,:) :: u_dev, v_dev, z_dev

allocate( u_dev(N), v_dev(N), z_dev(N) )
```

Somma di Vettori

4. 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(u_dev, u, sizeof(u), cudaMemcpyHostToDevice);
cudaMemcpy(v_dev, v, sizeof(v), cudaMemcpyHostToDevice);
```

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

```
u_dev = u ; v_dev = v
```

Somma di Vettori

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

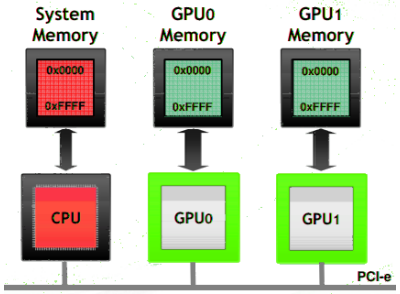
- `cudaMemcpy(void *dst, void *src, size_t size, direction)`
  - dst: Puntatore alla destinazione
  - src: Puntatore alla sorgente
  - size: Numero di byte da copiare
  - direction: tipo di trasferimento
    - da CPU a GPU: cudaMemcpyHostToDevice
    - da GPU a CPU: cudaMemcpyDeviceToHost
    - da GPU 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

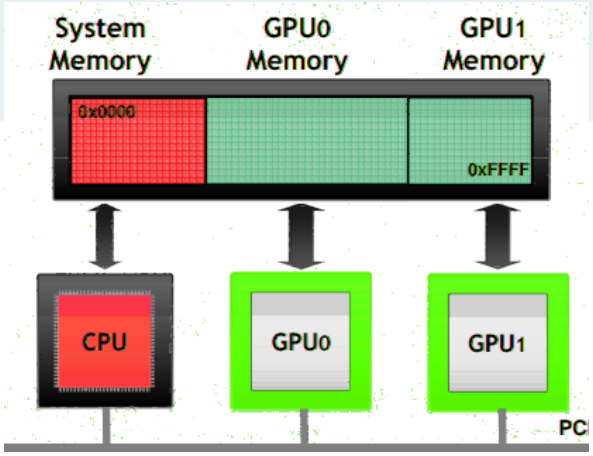
Unified Virtual Addressing

- CUDA 4.0 introduce un unico spazio di indirizzi (Unified Virtual Address) di memoria GPU e HOST:
  - la locazione di memoria fisica viene capita dall'indirizzo
  - consente di semplificare notevolmente le interfacce API

Pre-UVA	UVA
una definizione per ogni permitazione di sorgente/destinazione	una sola definizione direzionale

`cudaMemcpyHostToHost`
`cudaMemcpyHostToDevice`
`cudaMemcpyDeviceToHost`
`cudaMemcpyDeviceToDevice`
	`cudaMemcpyDefault`





Somma di Vettori: complete CUDA porting

```
double *u_dev, *v_dev, *z_dev;
cudaMalloc((void **)&u_dev, N * sizeof(double));
cudaMalloc((void **)&v_dev, N * sizeof(double));
cudaMalloc((void **)&z_dev, N * sizeof(double));

cudaMemcpy(u_dev, u, sizeof(u), cudaMemcpyHostToDevice);
cudaMemcpy(v_dev, v, sizeof(v), cudaMemcpyHostToDevice);

dim3 numThreads( 256); // 128-512 are good choices
dim3 numBlocks( (N + numThreads.x - 1) / numThreads.x );
gpuVectAdd<<<numBlocks, numThreads>>>( u_dev, v_dev, z_dev, N );
cudaMemcpy(z, z_dev, N * sizeof(double), cudaMemcpyDeviceToHost);
```

```
real(kind(0.0d0)), device, allocatable, dimension(:,:) :: u_dev, v_dev, z_dev
type(dim3) :: numBlocks, numThreads
allocate( u_dev(N), v_dev(N), z_dev(N) )
u_dev = u; v_dev = v

numThreads = dim3( 256, 1, 1 ) ! 128-512 are good choices
numBlocks = dim3( (N + numThreads%x - 1) / numThreads%x, 1, 1 )
call gpuVectAdd<<<numBlocks,numThreads>>>( u_dev, v_dev, z_dev, N )
z = z_dev
```

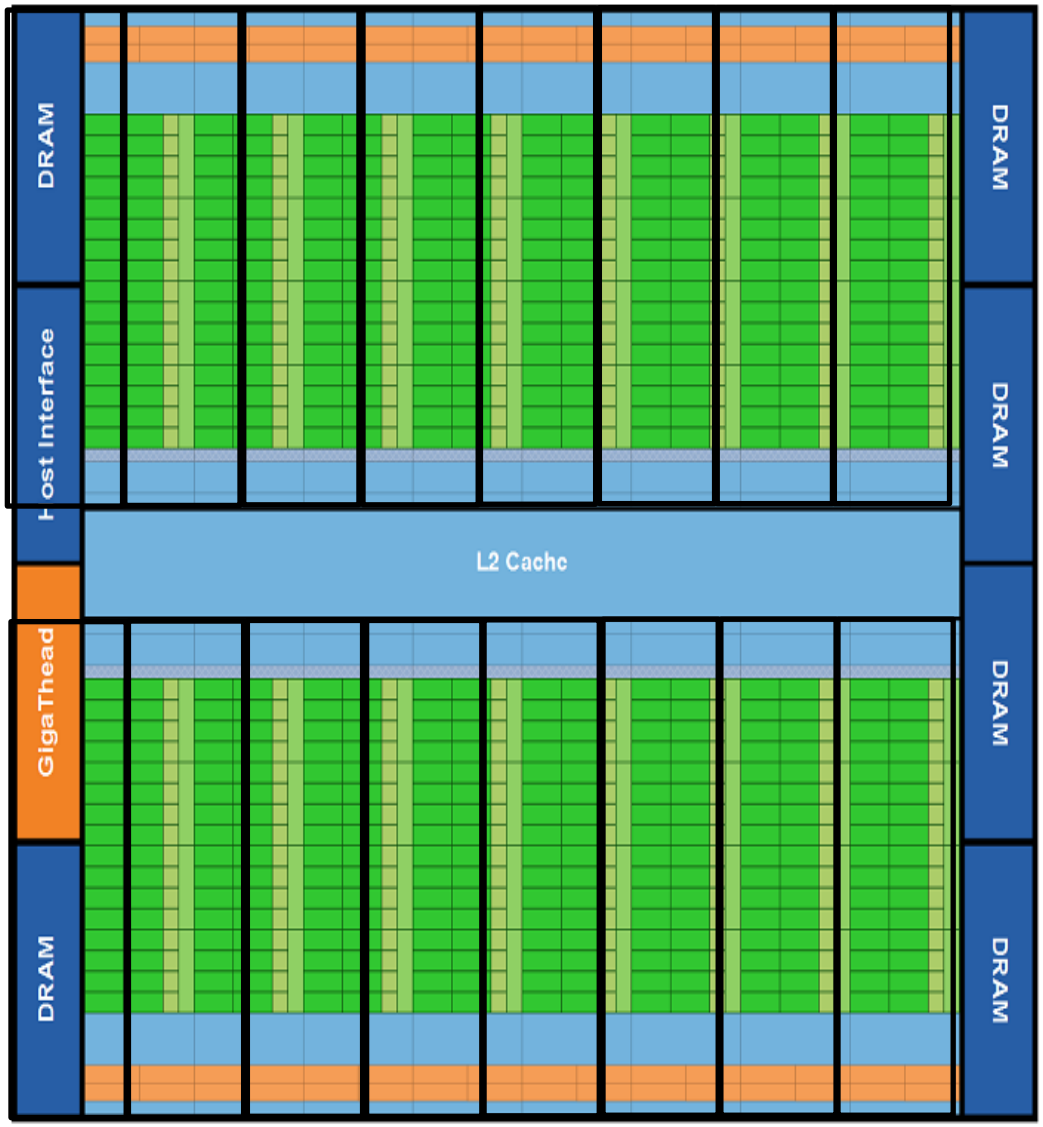
- Architettura CUDA
  - hardware FERMI e KEPLER
  - modello di esecuzione
- Compilatore e passi
  - PTX, cubin, what's inside
  - computing capability
- Altri modelli e acceleratori
  - OpenCL
  - OpenACC
  - Intel Xeon Phi (MIC) coprocessor

NVIDIA Architectures naming

- Mainstream & laptops: GeForce
  - Target: videogames and multi-media
- Workstation: Quadro
  - Target: gragica professionale per applicazioni CAD o modeling 3D
- GPGPU: Tesla
  - Target: High Performance Computing

L’architettura hardware NVIDIA Fermi

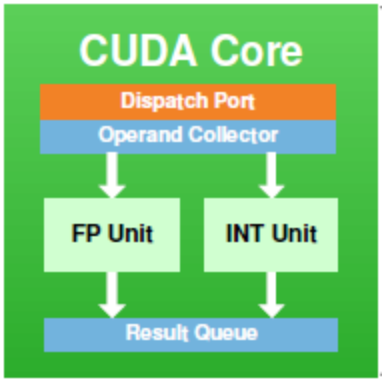
- 16 Streaming Multiprocessors (SM)
- 4-6 GB di memoria con ECC
- caches:
  - L1 (16-48KB) configurabile per SM
  - L2 (768KB) comune a tutti gli SM
- 2 controller indipendenti per trasferimento dati da/verso *host* tramite PCI-Express
- scheduler globale (GigaThread global scheduler) che distribuisce i blocchi di thread agli scheduler degli SM

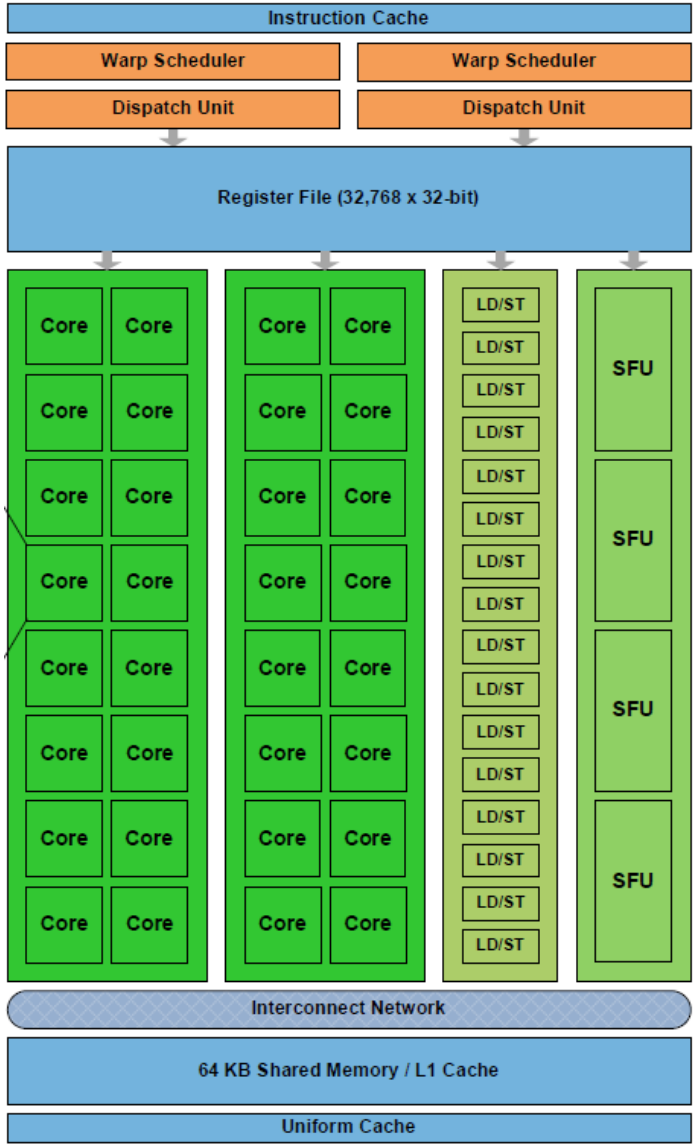


Fermi Streaming Multiprocessor (SM)

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*
- aderente allo standard IEEE 754-2008 a 32-bit e a 64-bit
  - **fused multiply-add** (FMA) in singola e doppia precisione
- 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,..)

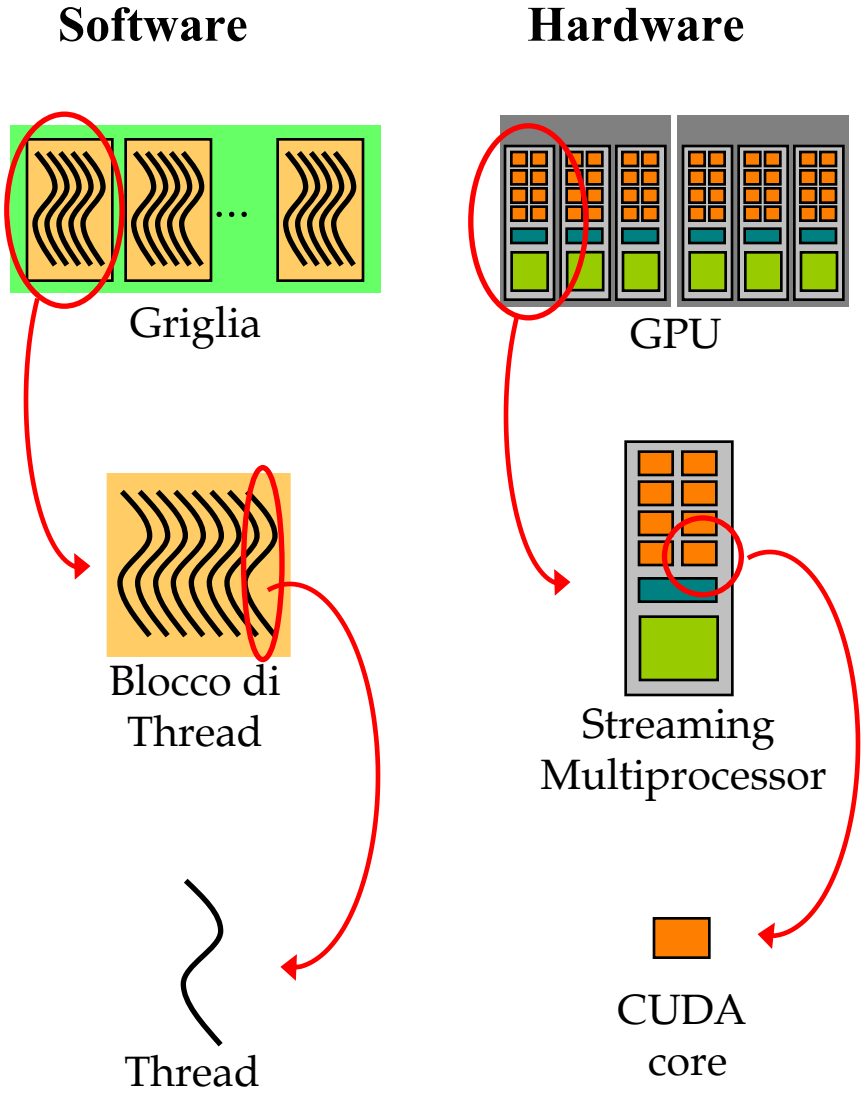




L’architettura hardware NVIDIA Kepler

- x3 performance/watt rispetto FERMI
  - litografia 28nm
- 192 core CUDA
- 4 warp scheduler (2 dispatcher)
  - *2 independent instruction/warp*
- standard IEEE 754-2008
- 65536 registri (32-bit)
- 32 unità load/store
- 32 Special Function Unit
- 1534KB L2 cache (x2 vs Fermi)
- 64KB shared-memory/cache + 48KB read-only L1 cache
- 16 texture units (x4 vs Fermi)

Il Modello di Esecuzione CUDA

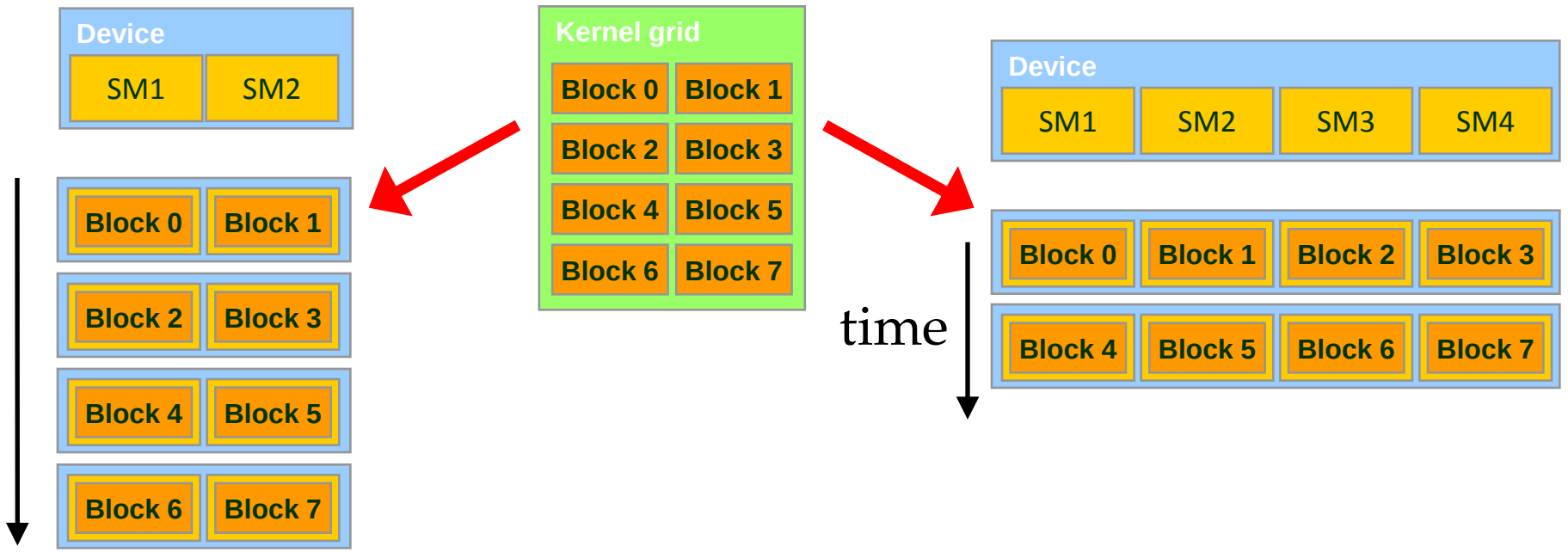


Al lancio di un kernel CUDA:

- ogni blocco della griglia è assegnato ciclicamente a uno Streaming Multiprocessor
  - Il numero massimo di blocchi in carico a ciascun SM dipende dalle caratteristiche hardware del multiprocessore (memoria shared e registri) e da quante risorse richiede il kernel da eseguire
  - eventuali bocchi non assegnati vengono allocati non appena un SM completa 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*)
- lo scheduler seleziona da uno dei blocchi a suo carico dei warp pronti per l'esecuzione
- le istruzioni vengono dispacciate per *warp*
  - ogni CUDA core elabora uno dei 32 *thread* del warp

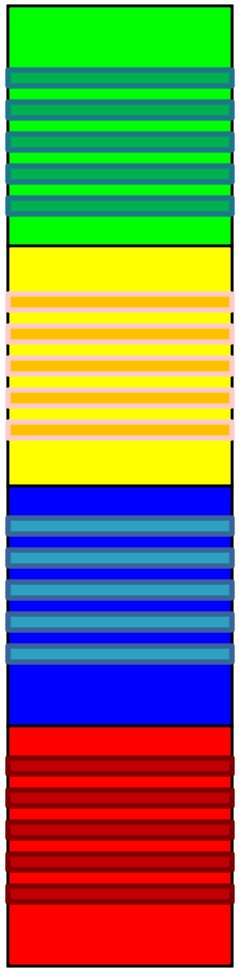
Scalabilità “Trasparente”

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



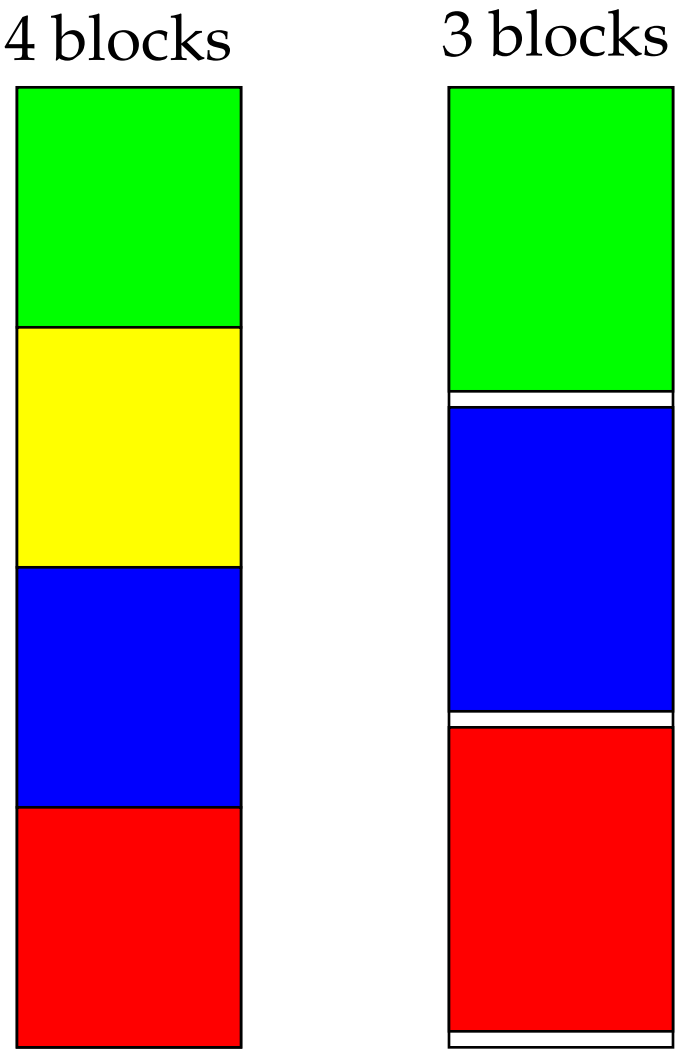
Gestione dei Registri

- 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
- lo zero-overload schedule è realizzato grazie al fatto che le informazioni di content switch dei warp rimangono inalterate nei registri del blocco



Assegnazione dei blocchi agli SM

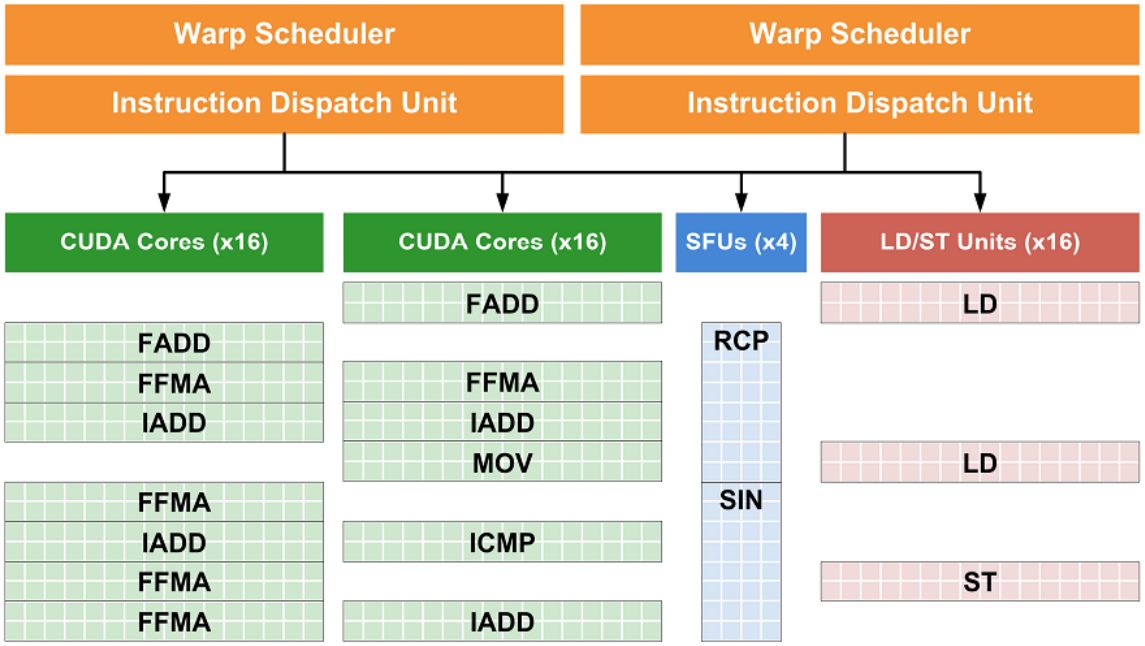
- Esempio di assegnazione di blocchi a SM:
  - Architettura Fermi: 32768 registri
  - Blocco 32x8 thread
  - kernel usa 30 registri
- Quanti blocchi possono girare per SM?
  - ogni blocco richiede 30x32x8 = 7680 registri
  - 32768/7680 = **4** + “resto”
  - assegnati solo 4 blocchi (invece di 8)
- Cosa 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 potenziale parallelismo!



Warps

- Il ***warp* è l'unità fondamentale di esecuzione**
- le potenzialità dello *scheduler* dipendono dalla *compute capability*:
  - 1.x : 8 core per SM - 1 scheduler - 1 istr. per *warp* - 1 istr. ogni 4 cicli
  - 2.0: 32 core per SM - 2 scheduler - 1 istr. per *warp* - 2 istr. ogni 2 cicli
  - 2.1: 48 core per SM - 2 scheduler - 2 istr. per *warp* - 4 istr. ogni 2 cicli

- ogni *warp* può eseguire una istruzione su
  - 16 CUDA core
  - 16 load/store units
  - 4 SFUs
- ci vogliono 2 cicli per completare una istruzione o una load/store per tutto il *warp*

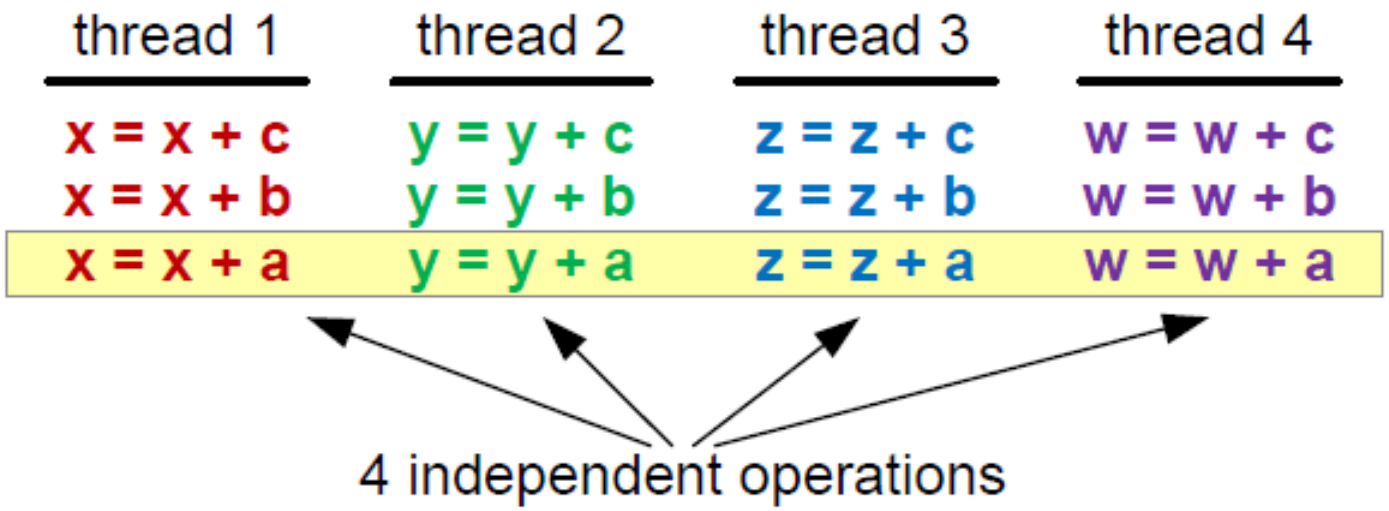


Nascondere le latenze

- Che cosa è la latenza?
  - numero di cicli necessari affinché una istruzione venga completata
  - ... e quindi il numero di cicli che devo attendere per poter iniziare una nuova istruzione dipendente dalla precedente
    - aritmetiche (18-24 cicli)
    - di accesso in memoria (400-800 cicli)
- Le latenze sono inevitabili, ma possiamo nasconderle. Come?
  - saturando le pipeline di calcolo
  - saturando la bandwidth
- Ciò si ottiene fornendo allo scheduler un numero sufficiente di **operazioni indipendenti** da svolgere, tramite *content-switch* , mentre attendiamo il completamento di quelle già istanziate
- Due paradigmi possibili (combinabili):
  - Thread-Level Parallelism (TLP)
  - Instruction-Level Parallelism (ILP)

Thread-Level Parallelism (TLP)

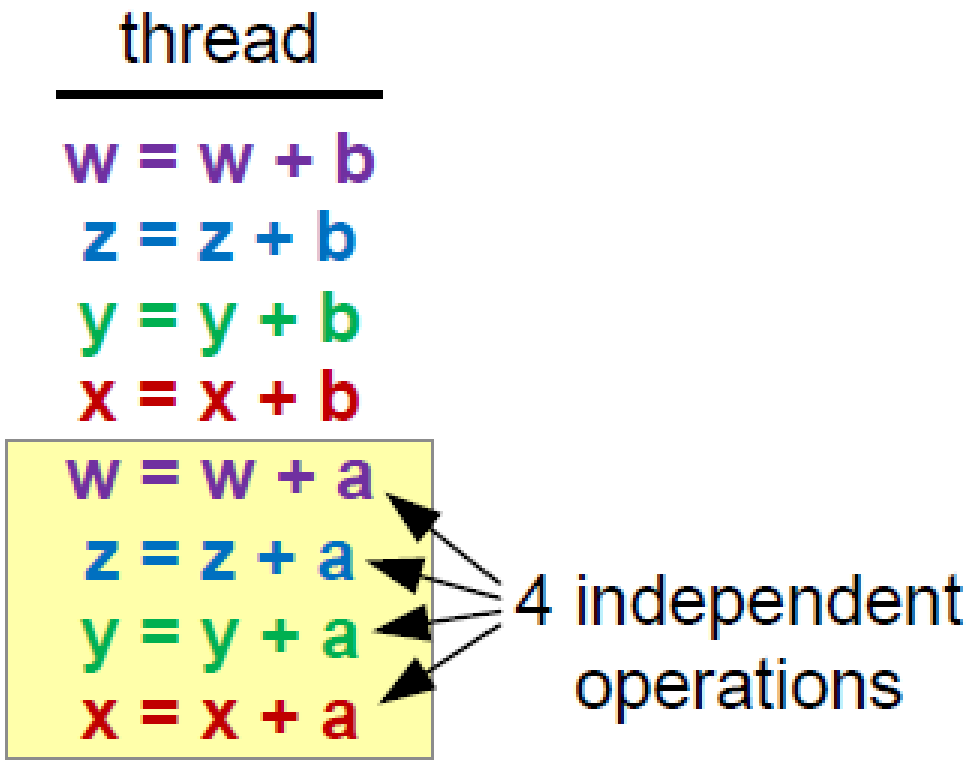
- Generalmente si consiglia di fornire molti thread per SM in modo da garantire il parallelismo minimo per coprire le latenze aritmetiche



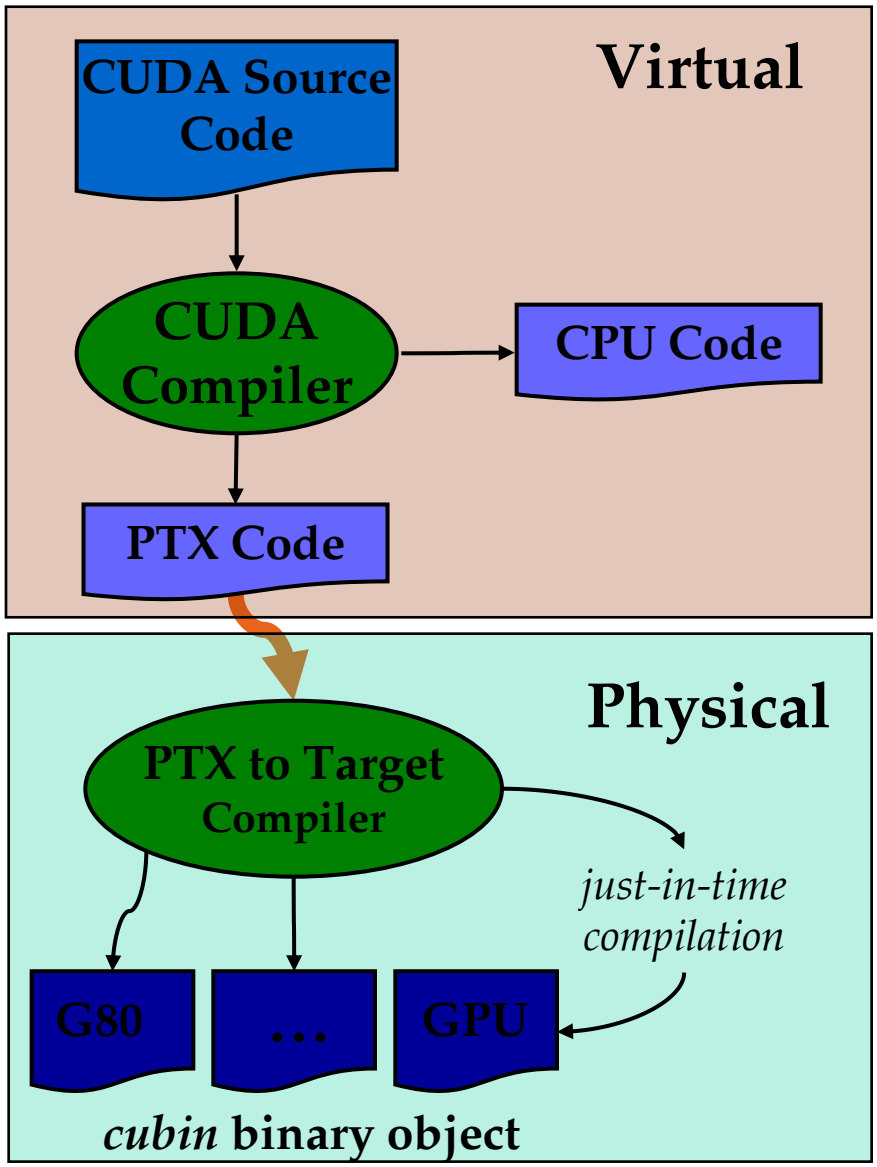
Instruction-Level Parallelism (ILP)

- una seconda alternativa è sfruttare il parallelismo interno alle istruzioni dello stesso thread, fornendo allo scheduler più operazioni indipendenti da accodare nella pipeline

- lo scheduler non passa a un nuovo thread se ci sono altre istruzioni da poter istanziare



Compilare Codice CUDA



- Ogni file sorgente contenente estensioni CUDA deve essere compilato con un compilatore CUDA compliant
  - `nvcc` CUDA C (NVIDIA)
  - `pgf90 -Mcuda` 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

Il codice PTX, pur descrivendo una macchina virtuale, viene esteso e arricchito nel tempo con nuove istruzioni per il nuovo hardware

- la *compute capability* specifica il set di istruzioni e *features* che si vuole supportare per la generazione del codice PTX
  - è identificata da dal codice "`compute_Xy`"
    - major number (X): identifica l'architettura base del chip
    - minor number (y): identifica varianti con più o meno features
  - specifica il set di istruzioni disponibili in compilazione del PTX code

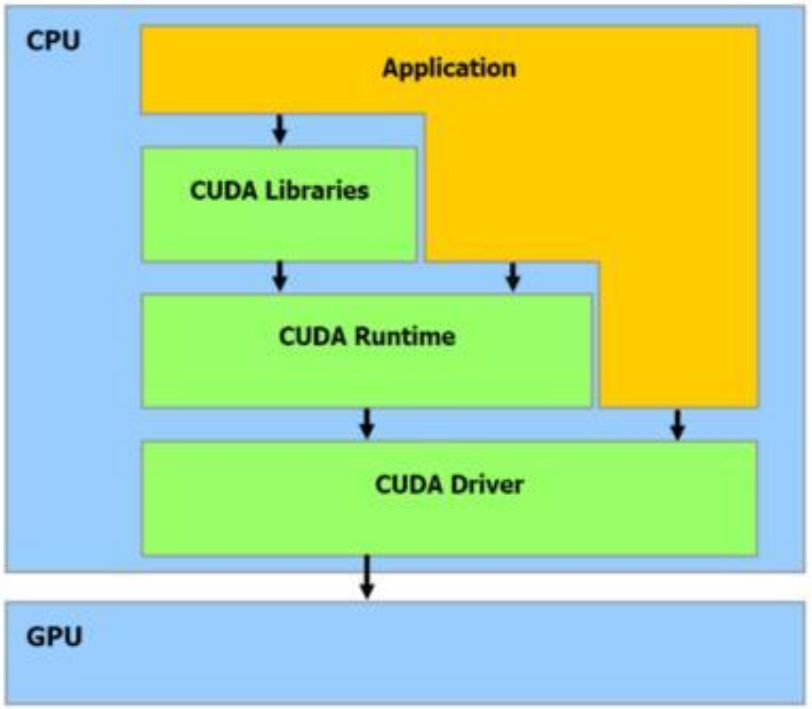
compute capability	*feature support*
compute_10	very basic features
compute_13	basic + double precision + atomics
compute_20	FERMI architecture (double precision)
compute_30	KEPLER K10 architecture (single precision)
compute_35	KEPLER K20, K20X, K40 architectures

Capability: resources constraints

Technical Specifications	Compute Capability						
	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						
Maximum width for a 1D texture reference bound to a CUDA array	8192				65536		

CUDA Driver Vs Runtime API

- CUDA offre due tipologie di API:
  - CUDA runtime API
  - CUDA driver API
- sono mutualmente esclusive



- Runtime API:
  - più semplice da programmare
  - facilita la gestione dei device
  - integra le estensioni al C per CUDA
- Driver API:
  - codice più elaborato da scrivere
  - gestione esplicita dei device e del lancio dei kernel
  - non ricorre a estensioni C per CUDA nel codice HOST

CUDA Driver API

- Le driver API sono implementate nella libreria dinamica nvcuda. Tutte le API hanno prefisso cu, invce di cuda (runtime API).
- E' una API imperativa, basata su handles: il programmatore costruisce una serie di oggetti opachi per la descrizione del device, contesto, kernel, parametri, ecc che vengono passati come argomento alle funzioni che li manipolano e utilizzano
- La driver API deve essere inizializzata con una chiamata a cuInit() prima di qualunque altra chiamata alla sua API.
- Deve essere creato un CUDA context per il controllo di ciascun device.
- All'interno di un CUDA context, i kernels sono caricati esplicitamente come stringhe di PTX o codice oggetto binario caricato come librerie dinamiche dall'host.
- I kernels sono lanciati utilizzando API entry points.

Vector add: driver Vs runtime 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); // CUDA C syntax sugar!
```

The Open Computing Language: OpenCL

- OpenCL è uno standard aperto multi-piattaforma
  - una API C a basso livello con binding C++
- può essere usato per programmare architetture eterogenee (multicore CPU, NVIDIA GPU, AMD GPU, Intel MIC, etc)
- OpenCL slogan: '*program once, run everywhere*'
  - sfortunatamente la portabilità del codice non si traduce in portabilità di performance
- Che aspetto ha il framework OpenCL?
  - fornisce supporto per il paradigma data parallel
  - poche differenze: un CUDA grid diventa un Ndrange in OpenCL, un warp si chiama wavefront, e così via…
  - per GPGPU: OpenCL è molto simile alla CUDA driver API

Vector add: OCL Host Code

```
// initialize OpenCL
err = clGetPlatformIDs(1, &cpPlatform, NULL);
err = clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
queue = clCreateCommandQueue(context, device_id, 0, &err);
// setup device memory
d_a = clCreateBuffer(context, CL_MEM_READ_ONLY, bytes, NUI
d_b = clCreateBuffer(context, CL_MEM_READ_ONLY, bytes, NUI
d_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY, bytes, NU
// copy array to the device
err = clEnqueueWriteBuffer(queue, d_a, CL_TRUE, 0, bytes, h_a, 0
err |= clEnqueueWriteBuffer(queue, d_b, CL_TRUE, 0, bytes, h_b,
NULL);
// prepare kernel launch
err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_a);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_b);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_c);
// load, *COMPILE* and *LINK* device code
program = clCreateProgramWithSource(context, 1,
                    (const char **) & kernelSource, NULL, &err);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
kernel = clCreateKernel(program, "vecAdd", &err);
// Execute the kernel over the NDrange
err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &globals
                                0, NULL, NULL);
```

```
// 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 the 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);
```

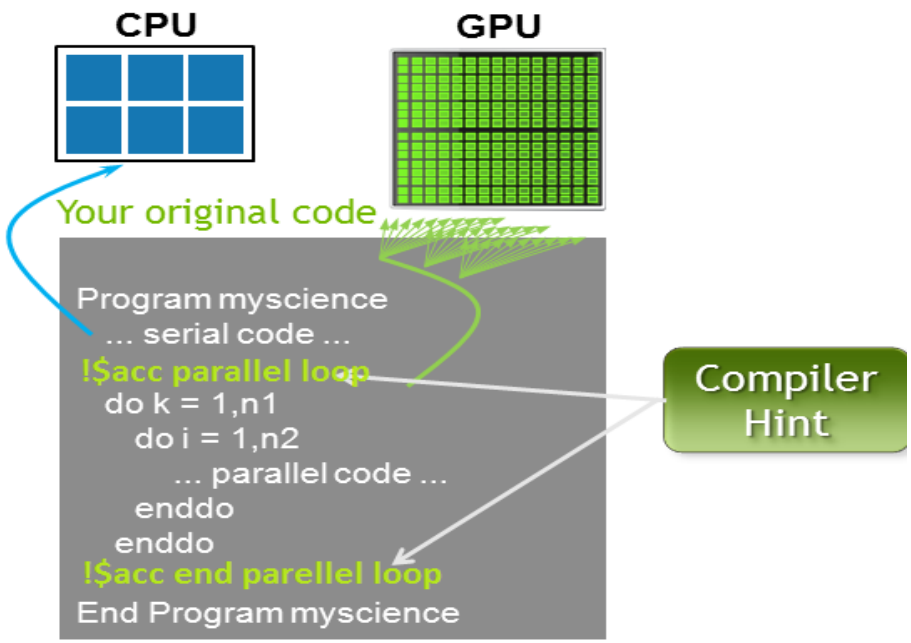
Vector add: OpenCL Device Code

```
// OpenCL kernel. Each work item takes care of one element of c
const char *kernelSource = \
"__kernel void vecAdd(  __global double *a,\n"\
"                       __global double *b,\n"\
"                       __global double *c)\n"\
"{\n"\
"//Get our global thread ID\n"\
"    int id = get_global_id(0);\n"\
"\n"\
"//Make sure we do not go out of bounds\n"\
"    if (id < n) \n"\
"        c[id] = a[id] + b[id];\n"\
"}\n" ;
```

- Anche i kernel sono molto simili a CUDA, ma ...
- i kernel sono delle stringhe caricate dall'HOST
- la compilazione e linking viene eseguita a runtime
- le similarità tra OpenCL/CUDA sono tali che esistono diversi progetti per la traduzione source-to-source (i.e: CU2CL)

OpenACC

- OpenACC è uno standard aperto per la programmazione parallela disegnato per rendere rapido e semplice l'accesso ad architetture eterogenee CPU/GPU
- OpenACC consente al programmatore di inserire nel codice dei "suggerimenti" al compilatore, dette direttive, per identificare le sezioni da accelerare



Maggiori vantaggi:

- High-Level: non richiede l'apprendimento e il ricorso a OpenCL, CUDA, etc.
- Single source: il sorgente rimane lo stesso per seriale/parallelo/acceleratori
- Portable: fornisce il supporto a diversi acceleratori di diversi vendor
- gran parte delle sue feature sono entrate nello standard OpenMP 4.0

OpenACC: A Simple Example

```
pgcc -acc -ta=nvidia -Minfo=accel saxpy.c

saxpy:
      3, Generating present_or_copyin(x[0:n])
         Generating present_or_copy(y[0:n])
         Generating compute capability 1.0 binary
         Generating compute capability 2.0 binary
      4, Loop is parallelizable
         Accelerator kernel generated
          4, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
             CC 1.0 : 8 registers; 48 shared, 0 constant, 0 local memory bytes
             CC 2.0 : 12 registers; 0 shared, 64 constant, 0 local memory bytes
```

Compiler is able to parallelize

```
int main(){

  int N = 1<<10;

  float *x, *y;
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));

  for (int i = 0; i < N; ++i) {
    x[i] = 2.0f; y[i] = 1.0f;
  }

  saxpy(N, 1.0f, x, y);

  return 0;
}
```

```
void saxpy (int n, float a,
    float *x, float *restrict y)
{

#pragma acc kernels
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];

}
```

OpenAcc performance

Example: Laplace equation in 2D

CPU: Intel Xeon X5680
6 Cores @ 3.33GHz
GPU: NVIDIA Tesla M2070

Execution	Time (s)	Speedup
CPU 1 OpenMP thread	69.80	--
CPU 2 OpenMP threads	44.76	1.56x
CPU 4 OpenMP threads	39.59	1.76x
CPU 6 OpenMP threads	39.71	1.76x
OpenACC GPU	13.65	2.9x

SpeedUp vs 1 CPU core

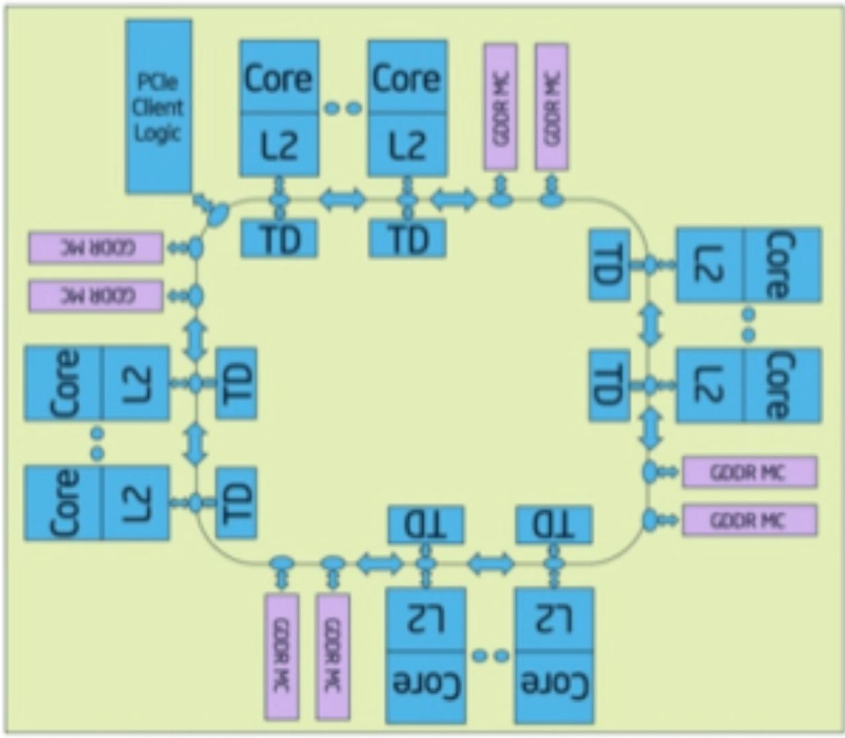
SpeedUp vs 6 CPU cores

Note: same code runs in 9.78s on NVIDIA Tesla M2090 GPU

SpeedUp = 4x

Intel Many Integrated Core (MIC)

- Intel Xeon Phi KNC (Knight Corner*) è un acceleratore multicore (60+1)-x86-core SMP.
  - 60 cores disponibili al calcolo
  - 1 riservato al sistema.
- ogni core ha una unità vettoriale SSE a 512-bit
- ogni core fornisce un hyper-threading 4way
- connessi da un bus bidirezionale a 512 bit
- 512 KB L2 cache comune
- 32KB L1 per core
- Xeon Phi KNC è montato su una scheda PCIe con 8 GB GDDR5 con ECC
  - theoretical peak perf. : 352 GB/s.
  - actual peak with ECC : 200 GB/s
- Intel dichiara: 'Up to 1 teraflops of double precision perfomance'



**Knight Corner* è il codename della prima generazione dell'architettura Intel MIC. La seconda generazione annunciata avrà nome *Knight Landing*.

Intel Xeon Phi: programming model

- Basata su OpenMP (o pthreads) e MPI
  - per partire con MIC non serve nessun nuovo linguaggio o paradigma di programmazione da imparare
- Il porting di una applicazione OpenMP/MPI su Xeon Phi si riduce semplicemente ad una ricompilazione dell'applicazione con qualche direttiva specifica e attivare dei flag di compilazione
- ... ma per prendere performance serve maggiore effort
  - Intel fornisce direttive di precompilazione per ottimizzazioni
  - OpenCL fornisce supporto all'architettura MIC

Intel Xeon Phi: accessing modes

- Offload mode:
  - utilizzando direttive Intel di precompilazione è possibile identificare le parti di codice da parallelizzare e da far eseguire sui processori Intel Xeon Phi
  - utilizzando le librerie ottimizzate Intel MKL (Math Kernel Library)
- Native mode:
  - ricompilando il sorgente per girare tutta l'applicazione in modo nativo direttamente sul coprocessore, come se si trattasse di un nodo di calcolo SMP Linux
  - oppure utilizzando ciascun coprocessore MIC come un nodo di calcolo e utilizzando MPI per distribuire i processi sui MIC disponibili e OpenMP sui core