GPU: processori manycore

Annalisa Massini
Calcolo Intensivo
a.a. 2011/2012

- Questa presentazione è tratta dalle slide e dai capitoli del libro presenti sul sito del corso

  Applied parallel programming

  http://courses.ece.illinois.edu/ece498/al/Syllabus.html
- Questo materiale rappresenta la versione preliminare del libro:
  - **Programming Massively Parallel Processors**
    D.B. Kirk , W.W. Hwu
    Morgan Kaufmann

Introduzione

- Negli ultimi 20 anni i microprocessori basati su una singola CPU hanno avuto un rapido incremento nelle prestazioni e una diminuzione dei costi.
- Questi microprocessori riescono ad arrivare a velocità dell'ordine dei GFLOPS, 10^9 operazioni in virgola mobile al secondo, nei computer da tavolo.
- L'evoluzione dell'hw ha permesso di aumentare la velocità delle applicazioni → lo stesso software va più veloce ogni volta che si introduce una nuova generazione di processori.

Introduzione

- Nel 2003 c'è stato un rallentamento perché consumi e problemi di riscaldamento hanno limitato l'incremento della frequenza del clock e delle attività che si possono eseguire in un ciclo di clock in una singola CPU.
- I produttori di microprocessori si sono orientati verso modelli in cui più unità di processo denominate **processing core**, sono presenti su un singolo chip, allo scopo di aumentare la potenza di calcolo.

Introduzione

- L'introduzione dei processing core ha avuto grande impatto sugli sviluppatori di sw.
- La stragrande maggioranza di applicazioni sono costituite da programmi sequenziali.
- Un programma sequenziale gira su un singolo core e non ci si aspetta che nei prossimi anni i core diventino più veloce di quelli in uso oggi.
- Quello che cambia le prestazioni è l'uso di programmi paralleli, in cui **più thread di esecuzione cooperano** per completare il lavoro più velocemente.

Introduzione

- L'uso della programmazione parallela non è nuovo.
- La comunità per High Performance Computing HPC sviluppa programmi paralleli da decine di anni.
- I calcolatori utilizzati sono molto grandi e molto costosi.
- Solo alcune applicazioni giustificano l'utilizzazione di calcolatori così costosi, quindi la pratica della programmazione parallela è ristretta ad un piccolo gruppo di sviluppatori.
- Adesso che tutti i microprocessori sono calcolatori paralleli si ha la necessità di sviluppare le nuove applicazioni come programmi paralleli.

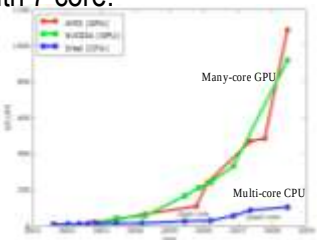
Introduzione

- Dal 2003 l'industria di semiconduttori per la progettazione di microprocessori ha seguito principalmente due linee.
- La linea di progettazione dei ***multicore*** cerca di aumentare la velocità di esecuzione dei programmi sequenziali attraverso l'uso dei diversi core presenti.
- Nei primi processori multicore erano presenti due core.
- Il processore Intel Core i7 (novembre 2008) ha 4 core, virtualizzati a 8 grazie alla tecnologia Hyper-Threading che sfrutta la duplicazione di alcune unità interne dei chip (2 hardware hyperthread) per mostrare al sistema operativo il doppio dei core "fisici" presenti nel sistema.

Introduzione

- Al contrario la linea di progettazione ***many-core*** è volta all'esecuzione di applicazioni parallele.
- All'inizio i manycore erano dotati di pochi core.
- Il processore grafico (GPU) nVidia Geforce GTX280 ha 240 core, ognuno con multithread che condivide controllo e cache istruzioni con altri 7 core.

I processori many-core, e in particolare le GPU, hanno mostrato un aumento sempre crescente delle prestazioni rispetto al calcolo in virgola mobile.



Introduzione

- La grande differenza di performance tra many-core GPU e multicore CPU è dovuta alle diverse filosofie di progettazione dei due tipi di processore.
- Una **CPU multicore** è ottimizzata rispetto alle prestazioni durante l'esecuzione di codice sequenziale.
- Si fa uso di una sofisticata logica di controllo per permettere che istruzioni provenienti da un singolo thread di esecuzione vengano eseguite in parallelo o anche in un ordine sequenziale diverso (out-of-order), mantenendo la logica di una esecuzione sequenziale.
- L'uso di memorie cache grandi e a più livelli riduce la latenza nell'accesso ai dati e alle istruzioni.

Introduzione

- Un aspetto importante per la valutazione delle prestazioni è la larghezza di banda per l'accesso in memoria.
- I chip grafici hanno larghezza di banda 10 volte più grande rispetto a quella disponibile sui chip per CPU.
- La difficoltà nell'aumentare la banda per le CPU è dovuta alle limitazioni poste dai sistemi operativi, dalle applicazioni e dai sistemi di I/O.
- Al contrario i progettisti di GPU possono raggiungere una maggiore larghezza di banda grazie alle minori limitazioni ereditate (legacy) e a modelli di memoria più semplici.

Introduzione

- La filosofia di progetto delle GPU è modellata dall'industria dei videgiochi che preme per ottenere grande velocità per le operazioni in virgola mobile.
- L'hw è progettato in modo che un numero grande di thread di esecuzione continui a lavorare anche quando alcuni thread sono in attesa di dati dalla memoria, minimizzando la logica di controllo per ogni thread .
- L'uso di piccole memorie cache permette di evitare che più thread abbiamo bisogno contemporaneamente di dati dalla DRAM.
- La maggior parte dell'area del chip è dedicata alle operazioni in virgola mobile.

Introduzione

- Le GPU sono progettate come motori per il calcolo numerico.
- **CPU e GPU** sono adatte ad eseguire **compiti diversi** e non danno buone prestazioni se utilizzate per compiti diversi da quelli che sanno fare.
- Molte applicazioni hanno bisogno sia di buone CPU che di GPU, in modo da eseguire le parti sequenziali sulle CPU e parti di calcolo numerico intensivo su GPU.
- Nel 2007 è stato introdotto da nVidia il modello di programmazione CUDA (Compute Unified Device Architecture) progettato per supportare l'esecuzione di applicazioni congiuntamente su CPU e GPU.

Introduzione

- Fino al 2006 per programmare i chip grafici era necessario usare API (Application Programming Interface) grafiche.
- Il tipo di applicazioni sviluppabili era comunque ridotto anche se si sono ottenuti ottimi risultati.
- La diffusione però è rimasta molto limitata.
- L'introduzione di CUDA permette di usare C/C++ senza bisogno di usare interfacce grafiche.

Introduzione

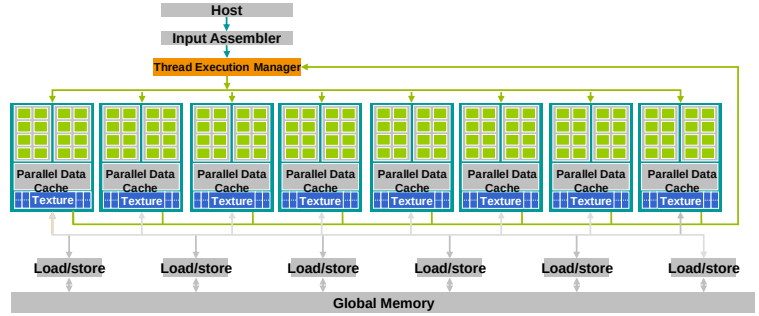
- Più recentemente un progetto, a cui hanno partecipato fra gli altri anche Apple, Intel, AMD/ATI e nVidia, ha sviluppato un altro modello di programmazione standardizzata che si chiama **OpenCL**.
- Un'applicazione **OpenCL** può girare senza modifica su tutti i processori che supportano **OpenCL**.
- Essendo nato dopo CUDA, **OpenCL** è ancora un po' indietro rispetto ad esso.
- Le caratteristiche chiave dei due modelli di programmazione presentano molte similarità.

Introduzione

- Un aspetto importante che ha reso le GPU più interessanti è stata l'adozione dello **standard IEEE** per la rappresentazione in **virgola mobile**.
- Questo comporta che i risultati ottenuti su GPU sono comparabili con quelli prodotti dalle CPU.
- Il problema principale era l'uso della singola precisione, poiché le applicazioni che hanno bisogno della doppia precisione non potevano girare su GPU.
- Dal 2008 non è più così: GPU attuali (Geforce G280 e Tesla) adottano la **doppia precisione**.

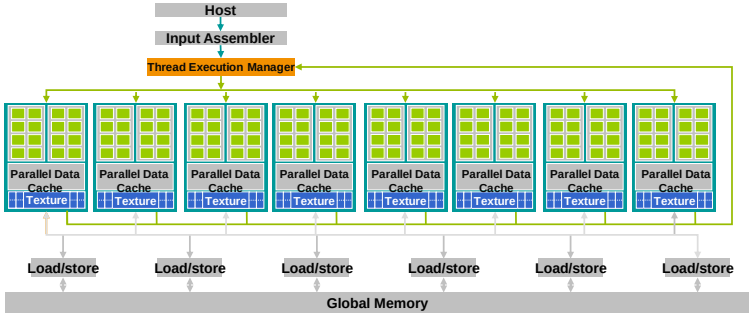
Architettura di una GPU

- Ecco come può essere organizzata una GPU:
  - Array di Streaming Multiprocessor (SM)
  - 2 SM in ogni building block → il numero di SM in un blocco può cambiare da una generazione all'altra



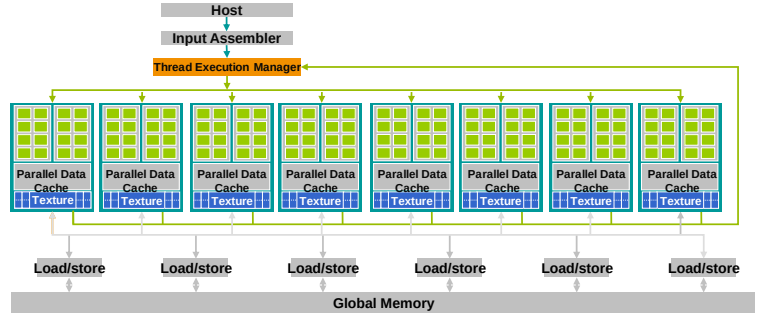
Architettura di una GPU

- In uno SM ci sono un certo numero di Streaming Processor (SP) che condividono la logica di controllo e la cache istruzioni.



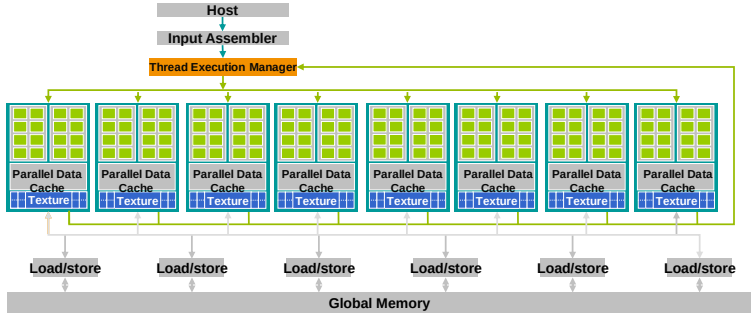
Architettura di una GPU

- Si può avere una memoria DRAM grafica Double Data Rate (GDDR) di dimensioni fino a 4 Gigabyte - differisce dalla DRAM di sistema (scheda madre) - è usata principalmente per grafica.
- Nel caso di calcolo, questa memoria presenta una very-high-bandwidth che compensa la maggiore latenza.



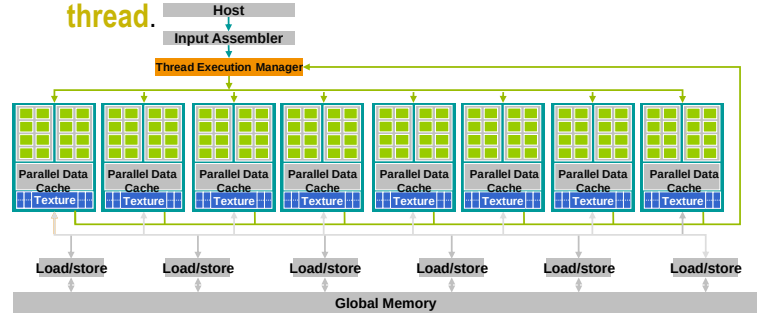
Architettura di una GPU

- La G80 ha 128 SP (16 SM, ognuno con 16 SP)
- Ogni SP ha un'unità multiply-add (MAD) e un'unità multiply aggiuntiva.
- In totale si producono 500 **gigaflops**.



Architettura di una GPU

- Le CPU della Intel supportano 2 o 4 **thread**.
- Il chip G80 supporta fino a 768 thread per SM, cioè 12.000 **thread** in totale
- Il chip GT200 con i suoi 240 SP supporta quasi 30.000 **thread**.



Parallelismo di dati: esempio

- In molte applicazioni, alcune sezioni presentano alto parallelismo tra dati → usare strutture dati su cui eseguire operazioni aritmetiche in modo simultaneo.
- Consideriamo un esempio semplice:
  prodotto matrice-matrice - **P=MxN**
- **Ogni elemento** della matrice prodotto **P:**
  - viene generato eseguendo il prodotto tra una riga della matrice M e una colonna della matrice N
  - non influisce sul calcolo degli altri elementi.
- Il calcolo di elementi diversi di P può essere fatto contemporaneamente.

Parallelismo di dati : esempio

- In una matrice **1000x1000** si devono calcolare **1.000.000** di elementi tra loro indipendenti ognuno dei quali richiede **1.000** moltiplicazioni e **1.000** addizioni.
- Una GPU può accelerare notevolmente questo calcolo.
- Certo il parallelismo di dati non è sempre così semplice come in questo caso!
- Usando CUDA il programma viene suddiviso in più fasi:
  - le fasi che non hanno parallelismo sono implementate nel **codice host** ed **eseguite sull'host**, cioè la CPU
  - le fasi che presentano grado di parallelismo alto sono implementate nel **codice device** ed **eseguite sulla GPU**

Parallelismo di dati

- Un programma CUDA è un sorgente che racchiude sia il **codice host** che il **codice device**.
- Il **compilatore C nVidia** (**nvcc**) separa i due codici durante il processo di compilazione.
- Il **codice host** è compilato poi con un compilatore C standard e gira come processo sulla CPU.
- Il **codice device** è esteso con etichette per le funzioni parallele, dette **kernel**, e per le strutture dati associate, viene ricompilato con **nvcc** e eseguito su GPU.
- I **kernel** generano tipicamente un grande numero di **thread** per sfruttare il parallelismo tra i dati.

Osservazioni

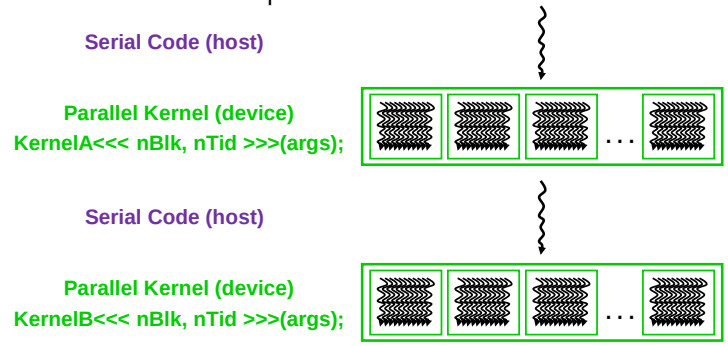
- Nell'esempio della moltiplicazione tra matrici, il calcolo può essere completamente realizzato come **kernel** e ogni **thread** viene usato per calcolare un elemento della matrice → vengono generati 1.000.000 di thread.
- Naturalmente questi thread sono molto più leggeri dei thread CPU e richiedono pochissimi cicli per essere generati e schedulati.

Kernel, grid, thread

- L'esecuzione di un tipico programma CUDA:
  - comincia con l'esecuzione dell'**host** (CPU)
  - quando viene invocata una funzione **kernel**, l'esecuzione si sposta nel **device** (GPU) dove viene generato un grande numero di **thread** allo scopo di sfruttare il parallelismo dei dati
  - tutti i **thread** generati da un **kernel** sono collettivamente chiamati **grid**

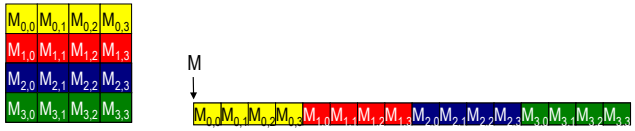
Kernel, grid, thread

- quando tutti i **thread** di un **kernel** sono completati, la corrispondente **grid** termina e l'esecuzione continua sull'**host** fino a quando viene lanciato un altro **kernel**.



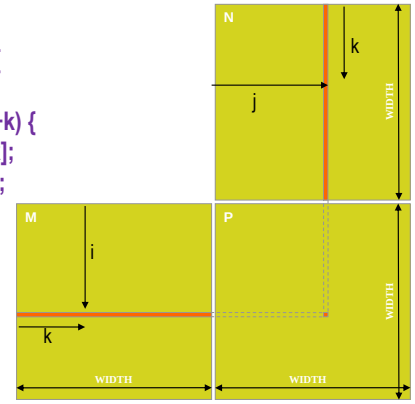
Moltiplicazione tra matrici

- Consideriamo una funzione per la moltiplicazione tra matrici.
- Consideriamo l'accesso agli elementi delle matrici secondo l'indirizzamento lineare della memoria, ricordando che per il C la convenzione è quella per righe:
  - tutti gli elementi di una riga sono in posizioni di memoria consecutive
  - le righe sono memorizzate una dopo l'altra.



Moltiplicazione tra matrici

```
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
for (int i = 0; i < Width; ++i)
    for (int j = 0; j < Width; ++j) {
      float sum = 0;
      for (int k = 0; k < Width; ++k) {
          float a = M[i  * width + k];
          float b = N[k * width + j];
          sum += a * b;
       }
       P[i * Width + j] = sum;
    }
}
```



Moltiplicazione tra matrici

- Modifichiamo la funzione affinchè il calcolo degli elementi della matrice avvenga su un device CUDA.

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
  int size = Width * Width * sizeof(float);
   float* Md, Nd, Pd;
  …
1. // Allocate device memory for M, N, P and
        // load M, N to device memory
2. // Kernel invocation code to have the device to perform
        // the actual matrix multiplication
3. // copy P from the device
         // Free device matrices
   }
```

Osservazioni

- Essenzialmente la nuova funzione si occupa di:
  - mettere i dati sul device
  - attivare il calcolo sul device
  - recuperare il risultato dal device.
- Per capire cosa succede effettivamente, guardiamo come è organizzata la memoria
- Per CUDA host e device hanno spazi di memoria separati → infatti tipicamente il device sta su una scheda hw separata e ha una sua DRAM.

Esecuzione del kernel

- Per eseguire il kernel su device, il programmatore deve:
  - allocare memoria sul device
  - trasferire dati dalla memoria dell'host alla memoria allocata nel device
- Analogamente, dopo l'esecuzione il programmatore:
  - deve trasferire i risultati nella memoria dell'host
  - liberare la memoria del device che non utilizza più
- Il sistema runtime di CUDA fornisce delle API che permettono di eseguire queste attività.

Codice host e codice device

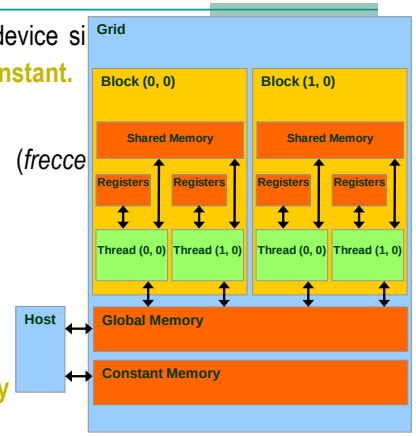
Per le comunicazioni tra host e device si hanno le memorie **global** e **constant.**

Il codice **host** può:

- trasferire dati al/dal device (*frecce bidirezionali*)

Il codice **device** può:

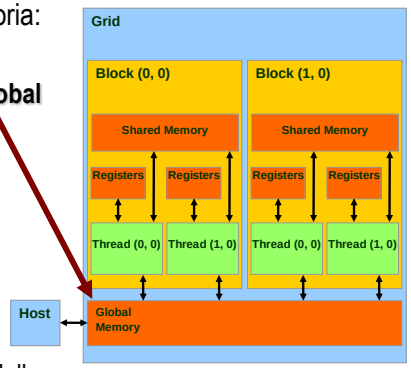
- R/W per-thread **registers**
- R/W per-thread **local memory**
- R/W per-block **shared memory**
- R/W per-grid **global memory**
- Read-only per-grid **constant memory**



API per la memoria del device

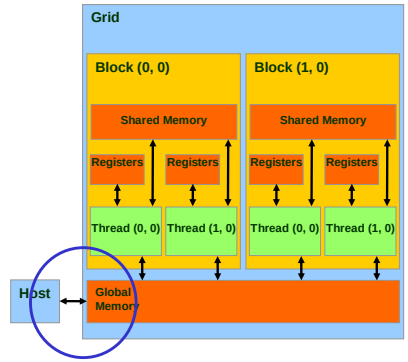
API per allocare e liberare memoria:

- **cudaMalloc()**
  - alloca l'oggetto dati nella **Global Memory** del device
  - richiede due parametri
    - **indirizzo del puntatore** all'oggetto allocato
    - **dimensione** dell'oggetto allocato
- **cudaFree()**
  - libera lo spazio di memoria dalla Global Memory del device
  - parametro: **puntatore** all'oggetto



API per la memoria del device

- **cudaMemcpy()**
  - API per il trasferimento di dati in memoria
  - richiede 4 parametri
    - puntatore alla destinazione
    - puntatore alla sorgente
    - numero di byte copiati
    - tipi di trasferimento
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
  - il trasferimento è asincrono



API per la memoria del device

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
  int size = Width * Width * sizeof(float);
  float* Md, Nd, Pd;
  …
1. // Allocate and Load M, N to device memory
   cudaMalloc(&Md, size);
   cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
   cudaMalloc(&Nd, size);
   cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);
   // Allocate P on the device
   cudaMalloc(&Pd, size);
2.  // Kernel invocation code – to be shown later
   …
3.   // Read P from the device
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
     // Free device matrices
    cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
  }
```

*N.B. le variabili che terminano con "**d**" si riferiscono a oggetti nello spazio di memoria del device*

N.B. **cudaMemcpyHostToDevice** *é una delle costanti predefinite e riconosciute nell'ambiente di programmazione CUDA*

Keyword _global_ _device_ _host_

- In CUDA, una funzione **kernel** specifica il codice che deve essere eseguito da tutti i thread durante una fase parallela.
- Per specificare che una funzione è un **kernel** si usa la keyword **__global__**, che viene quindi chiamata da un host per generare un **grid** di **thread** sul **device**
- Le altre due keyword che possono essere usate nella dichiarazione di una funzione sono:
  - **__device__** è una funzione eseguita su un device CUDA e può essere chiamata da una funzione kernel o da un'altra funzione device
  - **__host__** è la funzione C classica e può essere chiamata solo nell'host

Keyword threadIdx.x e threadIdx.y

- Altre importanti keyword sono **threadIdx.x** e **threadIdx.y** che permettono di riferire i “thread indices” di un thread in modo che ognuno di essi possa identificarsi e operare sulla parte di struttura dati su cui devono lavorare.
- Queste keyword identificano variabili predefinite.
- Thread differenti vedono valori differenti nelle loro variabili **threadIdx.x** e **threadIdx.y**
- Un thread è indicato dalle sue due coordinate e il fatto che le coordinate siano due riflette un’organizzazione matriciale dei thread.

Matrix multiplication kernel

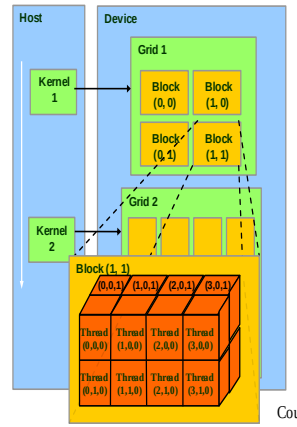
```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
// 2D Thread ID
int tx = threadIdx.x;
int ty = threadIdx.y;
// Pvalue stores the Pd element that is computed by the thread
float Pvalue = 0;
for (int k = 0; k < Width; ++k)
      {
      float Mdelement = Md[ty * Md.width + k];
      float Ndelement = Nd[k * Nd.width + tx];
      Pvalue += Mdelement * Ndelement;
      }
// Write the matrix to device memory each thread writes one element
Pd[ty * Width + tx] = Pvalue;
}
```

Thread e blocchi

- Quando un kernel viene lanciato parte l'esecuzione di una grid di thread paralleli.
- Una grid di thread può essere composta di migliaia o milioni di thread per ogni invocazione di kernel.
- Per creare un numero adeguato di thread per sfruttare in pieno l'hw è necessario un alto parallelismo tra i dati, come nell'esempio della moltiplicazione tra matrici.

Thread e blocchi

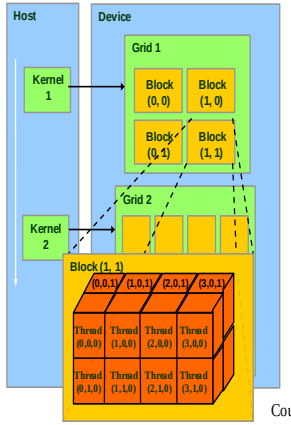
- I thread sono organizzati secondo una gerarchia a **due livelli**.
- A *livello superiore* una **grid** consiste di uno o più blocchi
- Ogni blocco ha lo stesso numero di thread.
- Nella figura Grid 1 è organizzata come un array 2 x 2 di 4 blocchi.
- Ogni blocco è identificato da da una coppia di coordinate individuata dalle keyword blockIdx.x e blockIdx.y



Courtesy: NDVIA

Thread e blocchi

- Ogni blocco è organizzato come un array 3 D di thread
- Le coordinate dei thread sono definite da **threadIdx.x, threadIdx.y** e **threadIdx.z**
- Non tutte le applicazioni usano tutte e tre le dimensioni
- Nell'esempio semplificato ci sono
  - 4 blocchi
  - ogni blocco è un array di thread 4 x 2 x 2
  - grid 1 consiste di 4 x 16 = 64 thread.



Courtesy: NDVIA

Parametri di *execution configuration*

- Quando il codice host chiama un **kernel**, le dimensioni della **grid** e del **blocco** vengono impostati tramite i parametri di '*execution configuration*'
- *Esempio*

```
// Setup the execution configuration
dim3 dimGrid(1, 1);
dim3 dimBlock(Width, Width);
// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

- Vengono dichiarate due struct di tipo dim3:
  - La prima definisce una grid 1 x1 quindi un solo blocco
  - La seconda definisce la configurazione dei blocchi
- L'ultima linea di codice invoca il **kernel**.
- Viene definita la dimensione della **grid** in termini di numero di **blocchi** e la dimensione dei **blocchi** in termini di numero di **thread**.

Thread e blocchi

In generale:

- una **grid** è organizzata come un array 2D di blocchi
- ogni **blocco** è organizzato come un array 3D di thread

I parametri usati per definire la configurazione d'esecuzione sono tipi `dim3`, che sono essenzialmente `struct` C con tre campi interi unsigned

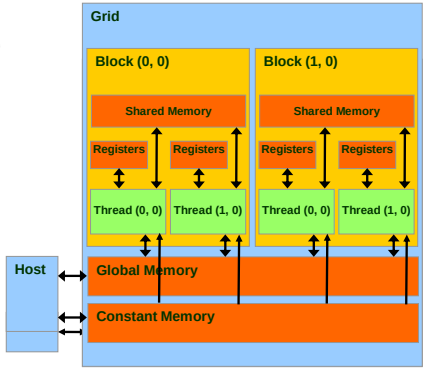
N.B. essendo una **grid** un array 2D il terzo campo è ignorato.

Syncthreads

- Cuda permette ai thread in uno stesso blocco di coordinare le proprie attività usando una funzione per generare una barriera di sincronizzazione `__syncthreads()`
- Quando un kernel chiama `__syncthreads()`, l'istruzione `__syncthreads()` viene eseguita da tutti i thread in un blocco
- Un thread che esegue la `__syncthreads()` aspetta rimane in attesa fino a quando tutti i thread hanno completato il loro lavoro.

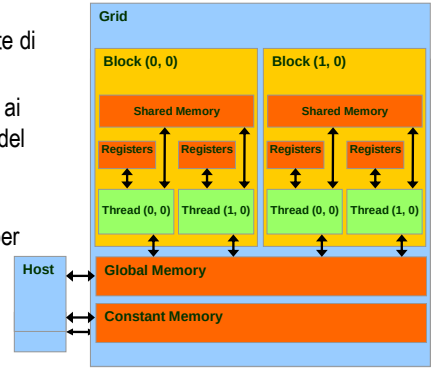
La memoria nella GPU

- Cuda supporta diversi tipi di memoria che possono essere usati dal programmatore per raggiungere alta velocità nell'esecuzione dei kernel
- Nella figura vengono mostrati
  - **registers**
  - **shared memory**
  - **global memory**
  - **constant memory**
- **Registri** e **shared memory** sono memorie on-chip



La memoria nella GPU

- **Registers** vengono usati per variabili private di uso frequente di ogni thread.
- La **shared memory** è allocata ai blocchi di thread: tutti i thread del blocco possono accedere alle variabili nella shared memory.
- La **shared memory** è usata per permettere ai thread di condividere dati di input o dati intermedi.
- R/W per-grid **global memory**
- R only per-grid **constant memory**



La memoria nella GPU

- Nella tabella è riportata la sintassi per dichiarare le variabili nei vari tipi di memoria.
- **Scope** indica quali thread possono accedere ad una variabile.
- Se lo **scope** di una variabile è un singolo thread, viene creata una versione privata della variabile per ogni thread.
- Il **lifetime** specifica quando la variabile è disponibile per essere usata rispetto all'esecuzione del programma.

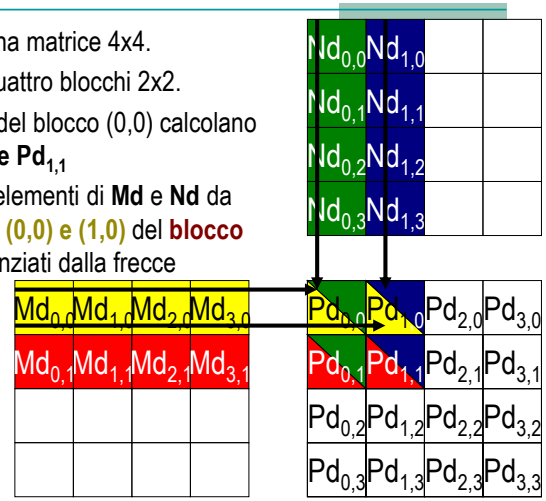
Variable declaration	Memory	Scope	Lifetime
Variabili automatiche	register	thread	kernel
`__device__ __shared__ int SharedVar;`	shared	block	kernel
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	application

Strategie di programmazione

- La memoria globale risiede sul device e presenta accessi più lenti rispetto alla **shared memory**
- Un modo per sfruttare la velocità della **shared memory** è dividere i dati in **tile**, cioè
  - partizionare i dati in **tile** della dimensione della **shared memory**
  - gestire ogni **tile** con un **blocco di thread**:
    - caricare il tile dalla global memory alla shared memory
    - usare thread multipli per sfruttare il parallelismo a livello di memoria
    - eseguire il calcolo sul tile dalla shared memory; ogni thread può usare più volte un elemento in memoria
    - copiare i risultati dalla shared alla global memory

Esempio della moltiplicazione

- Consideriamo una matrice 4x4.
- Consideriamo quattro blocchi 2x2.
- I quattro thread del blocco (0,0) calcolano **$Pd_{0,0}$ $Pd_{1,0}$ $Pd_{0,1}$ e $Pd_{1,1}$**
- Gli accessi agli elementi di **Md** e **Nd** da parte dei **thread (0,0) e (1,0)** del **blocco (0,0)** sono evidenziati dalle frecce



Moltiplicazione usando i blocchi

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
// Calculate the row index of the Pd element and M
int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
// Calculate the column index of Pd and N
int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

float Pvalue = 0;
// each thread computes one element of the block sub-matrix
for (int k = 0; k < Width; ++k)
  Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];

  Pd[Row*Width+Col] = Pvalue;
}
```

Moltiplicazione usando i blocchi

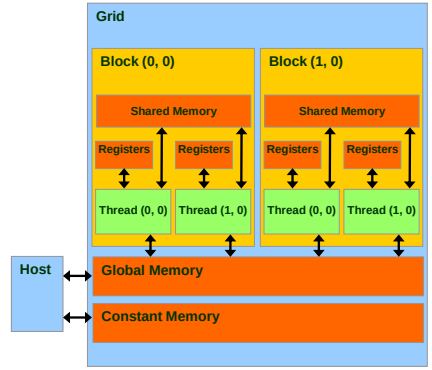
- Nella tabella vengono mostrati gli accessi alla global memory fatti da tutti i thread del blocco (0,0).
- Ogni thread accede a 4 elementi di Md e a 4 elementi di Nd,con un'evidente sovrapposizione
- Esempio: thread (0,0) e thread (1,0) accedono entrambi a tutta la riga 0 di M

$P_{0,0}$ $thread_{0,0}$	$P_{1,0}$ $thread_{1,0}$	$P_{0,1}$ $thread_{0,1}$	$P_{1,1}$ $thread_{1,1}$
$M_{0,0} * N_{0,0}$	$M_{0,0} * N_{1,0}$	$M_{0,1} * N_{0,0}$	$M_{0,1} * N_{1,0}$
$M_{1,0} * N_{0,1}$	$M_{1,0} * N_{1,1}$	$M_{1,1} * N_{0,1}$	$M_{1,1} * N_{1,1}$
$M_{2,0} * N_{0,2}$	$M_{2,0} * N_{1,2}$	$M_{2,1} * N_{0,2}$	$M_{2,1} * N_{1,2}$
$M_{3,0} * N_{0,3}$	$M_{3,0} * N_{1,3}$	$M_{3,1} * N_{0,3}$	$M_{3,1} * N_{1,3}$

Access order ↓

Performance

- Nell'esempio tutti i threads devono accedere alla global memory per prelevare gli elementi della matrice
  - Due accessi in memoria (8 bytes) per ogni operazione floating point
- La potenziale riduzione del traffico in memoria globale dipende dalla dimensione dei blocchi.
- Con blocchi NxN si può avere una riduzione pari a N,



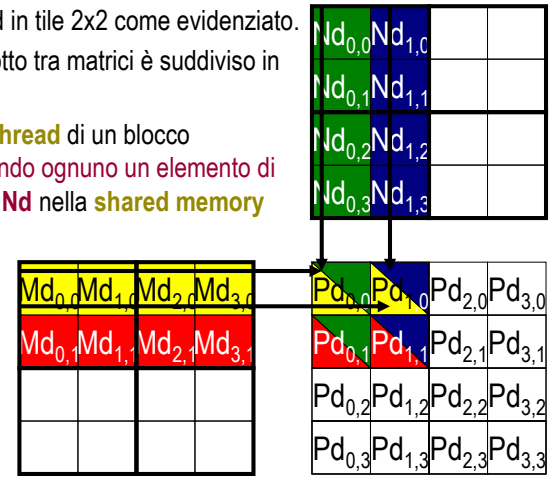
Usare la Shared Memory

- I thread devono collaborare per ridurre il traffico con la memoria globale.
- In particolare i thread devono collaborare nel caricare gli elementi di **Md** e **Nd** nella **shared memory**.
- La **shared memory** è piccola → nella divisione delle matrici **Md** e **Nd** si devono ottenere tile della dimensione della **shared memory**.
- Il modo più semplice di procedere è avere tile della dimensione dei blocchi.
- L'**obiettivo** è progettare algoritmi **tiled** in modo che
  - più threads collaborino nel caricare i dati nella shared memory
  - usino la copia locale (**shared memory**) dei dati

  per ridurre il traffico con la memoria globale

Suddividere Md e Nd in tile

- Dividiamo Md e Nd in tile 2x2 come evidenziato.
- Il calcolo del prodotto tra matrici è suddiviso in **due fasi**
- In ogni fase tutti i **thread** di un blocco collaborano caricando ognuno un elemento di **Md** e un elemento **Nd** nella **shared memory**



Fasi di esecuzione dei thread di un blocco

- Gli array nella shared memory per **Md** e **Nd** sono **Mds** e **Nds**

	Fase 1			Fase 2		
$T_{0,0}$	$\mathbf{Md_{0,0}}$ ↓ $Mds_{0,0}$	$\mathbf{Nd_{0,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$	$\mathbf{Md_{2,0}}$ ↓ $Mds_{0,0}$	$\mathbf{Nd_{0,2}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$
$T_{1,0}$	$\mathbf{Md_{1,0}}$ ↓ $Mds_{1,0}$	$\mathbf{Nd_{1,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$	$\mathbf{Md_{3,0}}$ ↓ $Mds_{1,0}$	$\mathbf{Nd_{1,2}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$
$T_{0,1}$	$\mathbf{Md_{0,1}}$ ↓ $Mds_{0,1}$	$\mathbf{Nd_{0,1}}$ ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$	$\mathbf{Md_{2,1}}$ ↓ $Mds_{0,1}$	$\mathbf{Nd_{0,3}}$ ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$
$T_{1,1}$	$\mathbf{Md_{1,1}}$ ↓ $Mds_{1,1}$	$\mathbf{Nd_{1,1}}$ ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$	$\mathbf{Md_{3,1}}$ ↓ $Mds_{1,1}$	$\mathbf{Nd_{1,3}}$ ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$

time →

Fasi di esecuzione dei thread di un blocco

	Fase 1		
$T_{0,0}$	$\mathbf{Md_{0,0}}$ ↓ $Mds_{0,0}$	$\mathbf{Nd_{0,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$
$T_{1,0}$	$\mathbf{Md_{1,0}}$ ↓ $Mds_{1,0}$	$\mathbf{Nd_{1,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$
$T_{0,1}$	$\mathbf{Md_{0,1}}$ ↓ $Mds_{0,1}$	$\mathbf{Nd_{0,1}}$ ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$
$T_{1,1}$	$\mathbf{Md_{1,1}}$ ↓ $Mds_{1,1}$	$\mathbf{Nd_{1,1}}$ ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$

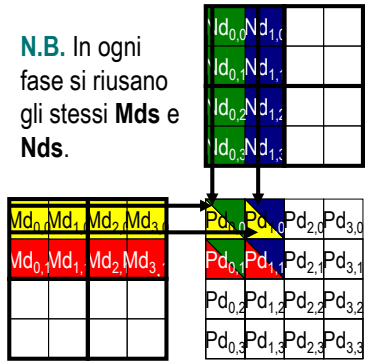
- All'inizio della **Fase 1** i thread del blocco (0,0) caricano un tile di **Md** nella shared memory.
- Analogamente un tile di **Nd**.
- I valori caricati vengono utilizzati nel calcolo del prodotto.
- **N.B.** Ognuno dei valori viene usato **due volte**.
  Nell'esempio si vede l'uso di $\mathbf{Md_{0,1}}$ e di $\mathbf{Nd_{1,0}}$.

Fasi di esecuzione dei thread di un blocco

	Fase 2		
$T_{0,0}$	$\mathbf{Md_{2,0}}$ ↓ $Mds_{0,0}$	$\mathbf{Nd_{0,2}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}*Nds_{0,0}$ + $Mds_{1,0}*Nds_{0,1}$
$T_{1,0}$	$\mathbf{Md_{3,0}}$ ↓ $Mds_{1,0}$	$\mathbf{Nd_{1,2}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}*Nds_{1,0}$ + $Mds_{1,0}*Nds_{1,1}$
$T_{0,1}$	$\mathbf{Md_{2,1}}$ ↓ $Mds_{0,1}$	$\mathbf{Nd_{0,3}}$ ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}*Nds_{0,0}$ + $Mds_{1,1}*Nds_{0,1}$
$T_{1,1}$	$\mathbf{Md_{3,1}}$ ↓ $Mds_{1,1}$	$\mathbf{Nd_{1,3}}$ ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}*Nds_{1,0}$ + $Mds_{1,1}*Nds_{1,1}$

- Nella **Fase 2** si caricano i tile di Md e Nd **opportuni** per completare il calcolo del prodotto.

N.B. In ogni fase si riusano gli stessi **Mds** e **Nds**.



Funzione kernel tiled

Funzione **kernel tiled** per l'uso della **shared memory**

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int
    Width)
{
1. __shared__float Mds[TILE_WIDTH][TILE_WIDTH];
2. __shared__float Nds[TILE_WIDTH][TILE_WIDTH];

3. int bx = blockIdx.x;  int by = blockIdx.y;
4. int tx = threadIdx.x; int ty = threadIdx.y;

// Identify the row and column of the Pd element to work on
5. int Row = by * TILE_WIDTH + ty;
6. int Col = bx * TILE_WIDTH + tx;
```

Funzione kernel tiled

```
7.  float Pvalue = 0;
// Loop over the Md and Nd tiles required to compute the Pd element
8.  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
// Collaborative loading of Md and Nd tiles into shared memory
9.    Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
10.   Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
11.   __syncthreads();
12.   for (int k = 0; k < TILE_WIDTH; ++k)
13.     Pvalue += Mds[ty][k] * Nds[k][tx];
14.     synchthreads();
      }
15. Pd[Row*Width+Col] = Pvalue;
  }
```

Tiled Multiply

- Ogni **blocco** calcola una sottomatrice Pd_{sub} di dimensioni TILE_WIDTH
- Ogni thread calcola un elemento di Pd_{sub}