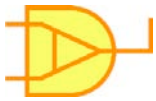
CUDA
computing with the GPU

Federico Pedersini

Università degli studi di Milano
Dipartimento di scienze dell'informazione



Outline

- Introduzione
- CUDA hardware
- Programmazione CUDA
  - CUDA software architecture
  - GPU memory management
  - Ottimizzazione dell'accesso ai dati
  - Sincronizzazione
  - Aspetti di calcolo
- Conclusioni

NVIDIA CUDA world...

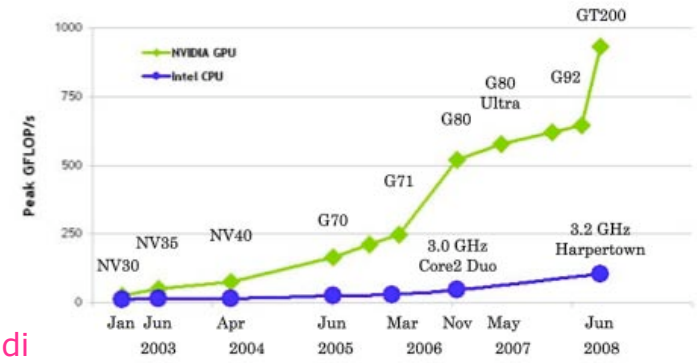
- CUDA hardware
  - Schede grafiche NVIDIA (*GeForce*, *Quadro*)
  - Sistemi NVIDIA *high-performance computing (HPC)* (*Tesla*)



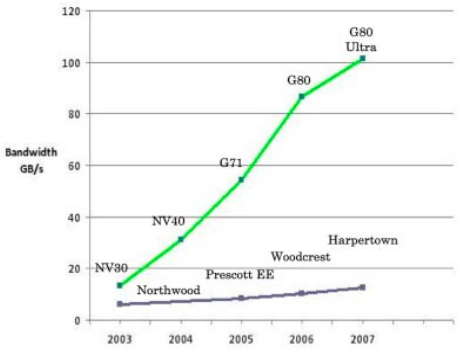
- CUDA software
  - ***CUDA*** language (estensione di C)
  - Librerie per programmazione ad alto livello
    - ***CUBLAS*** (algebra lineare), ***CUFFT*** (calcolo FFT): non è necessario imparare CUDA per utilizzarle!

Perchè fare calcolo con le GPU?

- Performance...
  - **GeForce GT 280: > 900 MFLOPS/s**
  - Xeon 3.2 GHz: ~ 90 MFLOPS/s



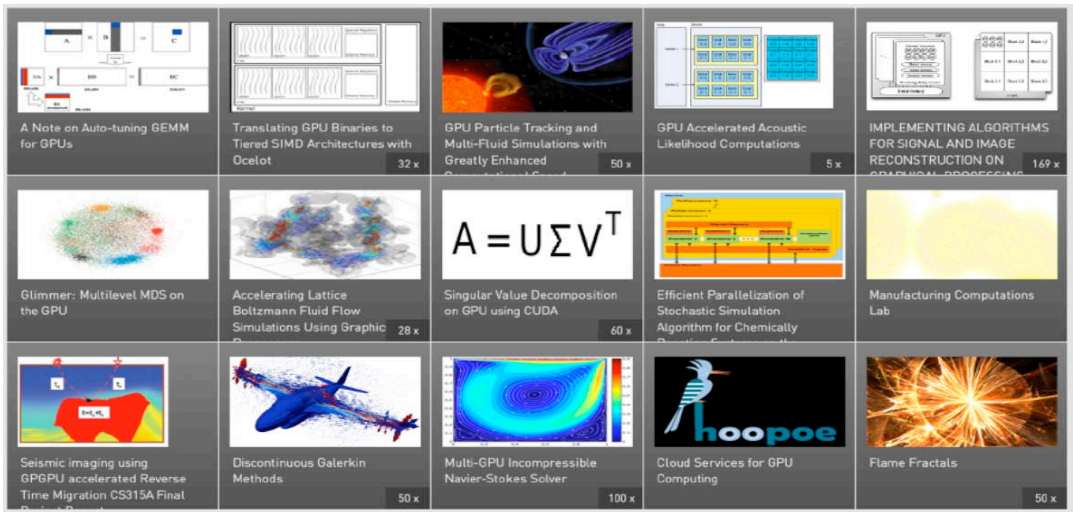
- ...a costi da produzione di massa!
  - **GeForce GT 280:**
    - 240 processori
    - 1 GB RAM a bordo
    - **< 400 €**



GT200 = GeForce GTX 280	G71 = GeForce 7900 GTX	NV35 = GeForce FX 5950 Ultra
G92 = GeForce 9800 GTX	G70 = GeForce 7800 GTX	NV30 = GeForce FX 5800
G80 = GeForce 8800 GTX	NV40 = GeForce 6800 Ultra	

Applicazioni

- Applicazioni tipiche: tutte le applicazioni di calcolo parallelo e "massivo" (algoritmi semplici su quantità enormi di dati)
  - Astrofisica (simulazioni astronomiche)
  - Meccanica computazionale (fluidodinamica, cinematica, modelling acustico)
  - Meteorologia
  - ...



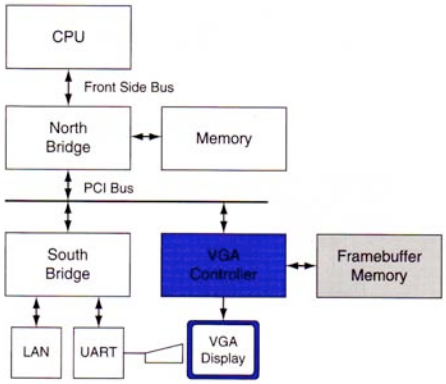
CPU vs. GPU architecture

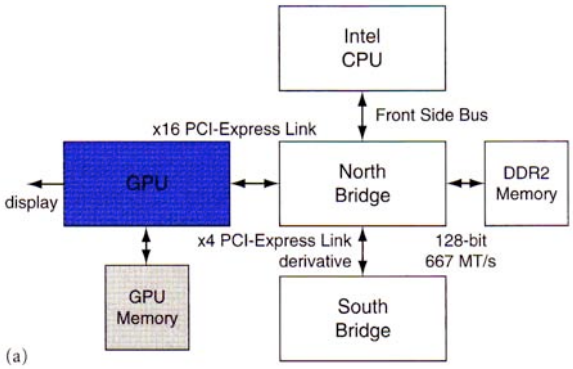
- Le GPU sono processori specializzati per impieghi fortemente paralleli e di calcolo intensivo.
  - Stesso algoritmo su molti dati (SIMD) ➔ controllo di flusso molto semplice
  - Strutture di memoria semplici e con bassa latenza, anziché ad accesso ottimizzato, ma complesso (cache)

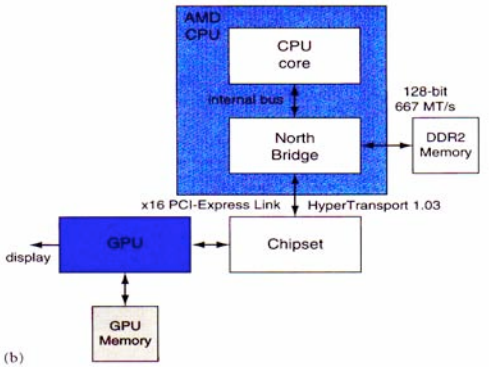
"The GPU devotes more transistors to Data Processing"
(NVIDIA CUDA Programming Guide)

Architettura PC / GPU

- Architettura PC tradizionale (1990)
  - Sottosistema grafico (VGA) è una delle periferiche
  - Comunicazione attraverso bus generico (PCI)
- Architettura PC moderno
  - Presenza di **GPU** (Graphic Processing Unit)
  - Memoria dedicata a ciascun processore
  - Comunicazione GPU–CPU ad alta velocità (PCI-E: 16 GB/s)

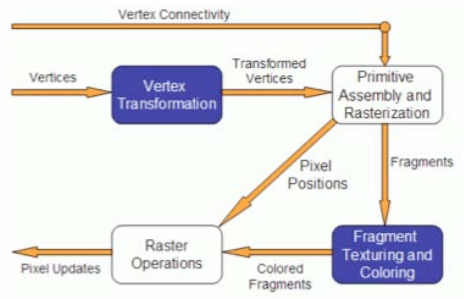




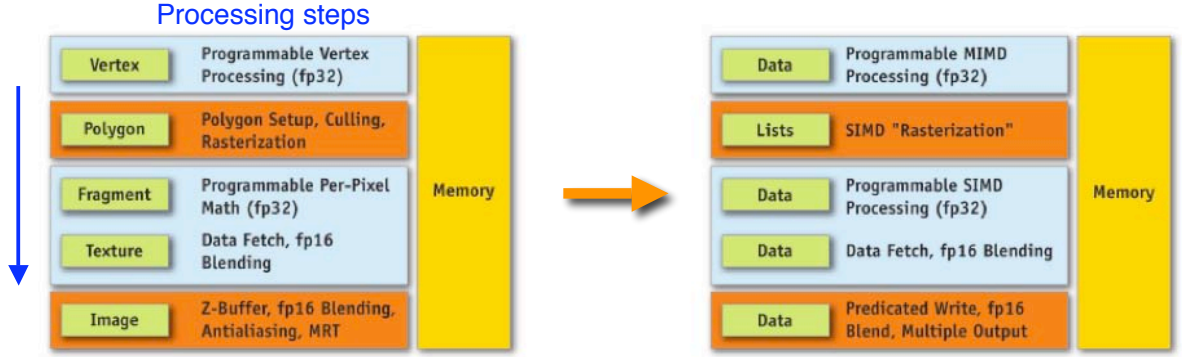


Traditional GP-GPU architecture

- Pipeline grafica tradizionale
  - **Vertex shader:**
    - Dai vertici 3D alla loro proiezione prospettica ➔ poligoni sul piano immagine
  - **Fragment shader:**
    - dai poligoni ai pixel, per la rasterizzazione
  - Ogni **shader** fa operazioni semplici e facilmente parallelizzabili

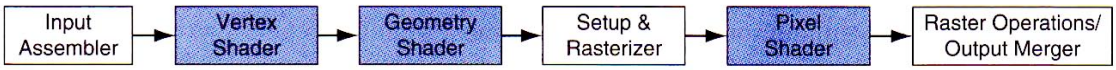


Processing steps

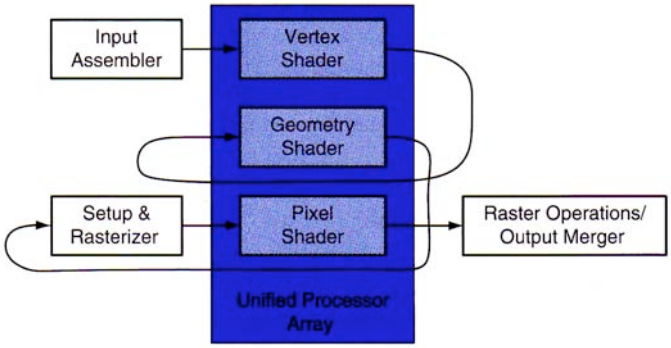


The unified architecture

- Progetto classico GPU:
  - struttura pipeline, blocchi funzionali fissi / programmabili (shaders)
  - Possibilità (limitata) di programmazione shaders



- Progetto moderno GPU:
  - blocchi funzionali completamente programmabili
  - Comunicazione fra blocchi funzionali configurabile: (pipeline, parallelo, blocchi indipendenti, ...)

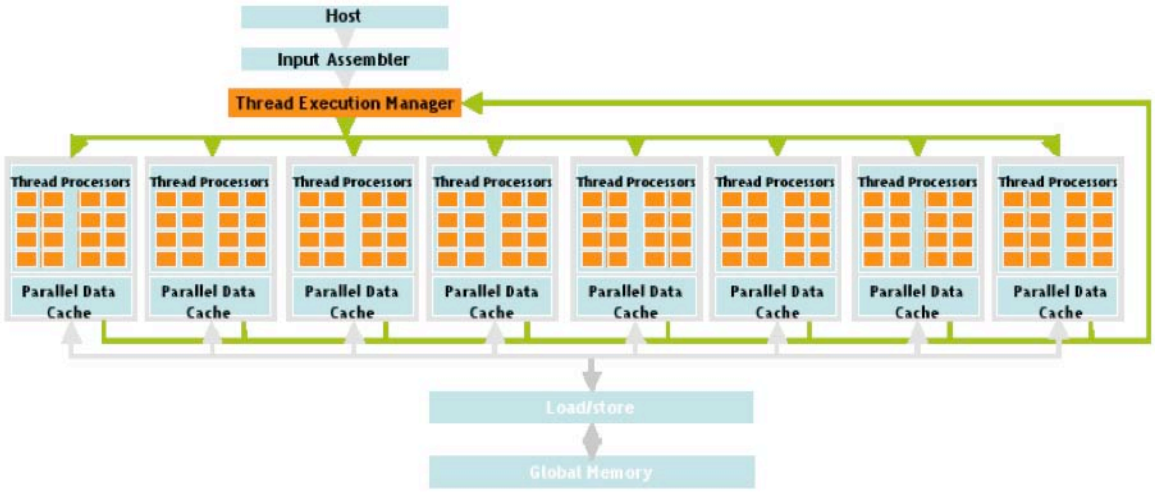


***CUDA**: Compute Unified Device Architecture*

- Da shaders in cascata a shaders paralleli, general-purpose
- Ogni shader è una Processing Unit

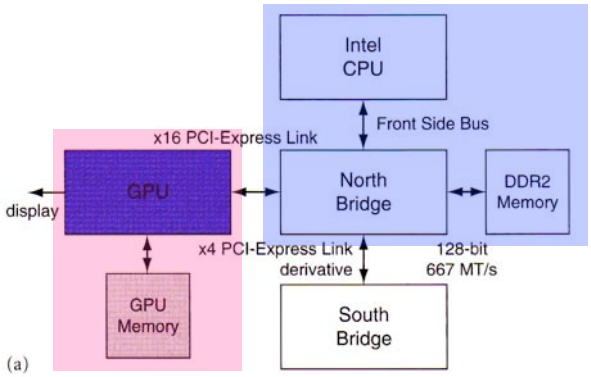
CUDA: Compute Unified Device Architecture

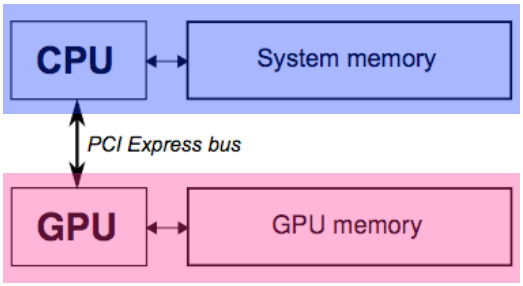
- *NVIDIA GeForce-88xx* / GeForce-98xx series architecture
  - 128 general-purpose **Streaming Multiprocessors (SM)**
  - GPU memory – transfer rate fino a 90 GB/s



Il sistema CPU-GPU

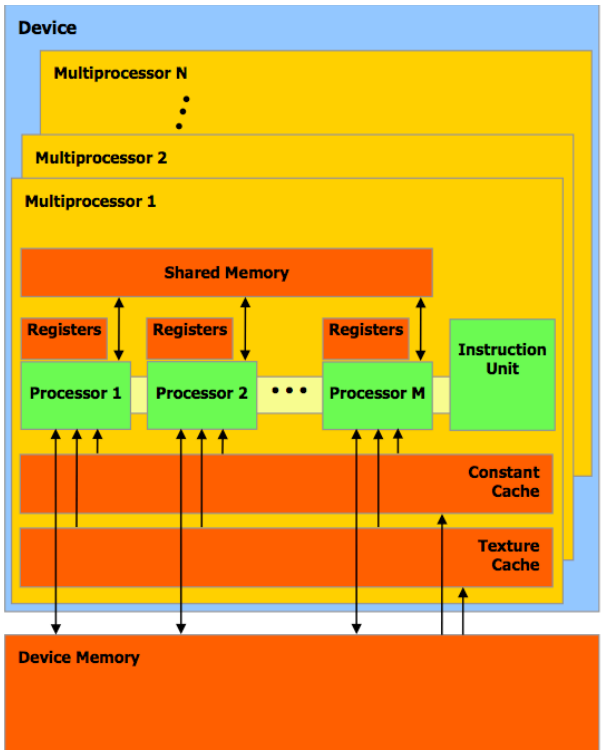
- **CUDA approach: CPU = host, GPU = device**
  1. CPU invia dati alla GPU per l'elaborazione (CPU memory ➜ GPU memory)
  2. GPU elabora i dati e genera risultati
  3. CPU recupera i risultati (GPU memory ➜ CPU memory)





CUDA: architettura hardware

- Device CUDA: **array di Multiprocessori**
- **Streaming Multiprocessor (SM)**
  - Uno SM contiene M processori, in grado di eseguire threads in parallelo in modalità **SIMD:** eseguono la stessa istruzione su dati diversi
  - Uno SM è in grado di eseguire un ***warp*** di threads (32 threads)
- Organizzazione della memoria
  - **Register file** spazio privato di ogni singolo processore
  - **Shared memory**, condivisa da tutti i processori di uno stesso Multiprocessore
  - Read-only **constant cache**, per lettura accelerata di costanti
  - Read-only **texture cache**, ottimizzata per la lettura di textures
  - **Device memory**, esterna ai Multiprocessori: spazio condiviso



Terminologia CUDA

Thread: codice concorrente, eseguibile in parallelo ad altri threads su un device CUDA.

- È l'unità fondamentale del parallelismo in CUDA
- Rispetto ai threads CPU, nei threads CUDA i costi (tempi) di creazione e di commutazione e l'utilizzo delle risorse sono molto minori.

Warp: un gruppo di threads che possono essere eseguiti *fisicamente* in parallelo (SIMD – o *SPMD, Single Program, Multiple Data*)

- ***Half-warp***: una delle 2 metà di un warp (spesso eseguiti sullo stesso multiprocessore)

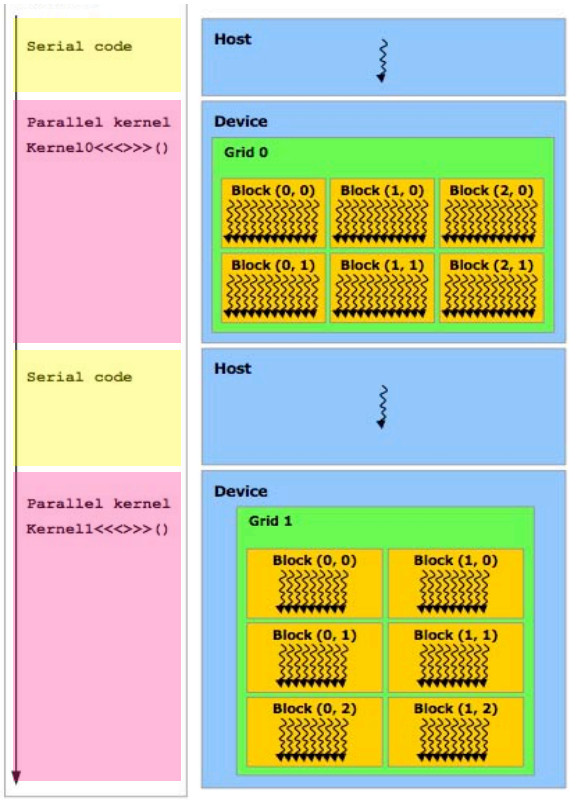
Block: un insieme di threads eseguiti sullo stesso Multiprocessore, e che quindi possono condividere memoria (stessa *shared memory*)

Grid: un insieme di thread blocks che eseguono un singolo kernel CUDA, in parallelismo *logico*, su una singola GPU

Kernel: il codice CUDA che viene lanciato (dalla CPU) su una o più GPU

Modello di esecuzione

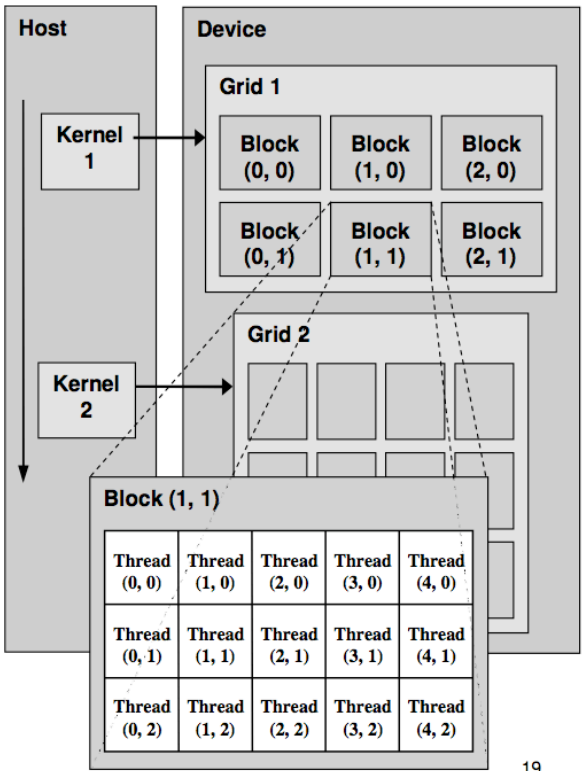
- Un codice CUDA alterna porzioni di codice seriale, eseguito dalla CPU, e di codice parallelo, eseguito dalla GPU.
- Il codice parallelo viene lanciato, ad opera della CPU, sulla GPU come **kernel**
  - La GPU esegue un solo kernel alla volta
- Un kernel è organizzato in **grids** di blocks
  - Ogni block contiene lo stesso numero di threads
- Ogni block viene eseguito da un solo multiprocessore: non può essere spezzato su più SM, mentre più blocks possono risiedere, ed essere eseguiti in parallelo, dallo stesso multiprocessore
  - Tutti i threads di un block condividono la shared memory
  - Il numero max. dipende dalle dimensioni del multiprocessore
  - Il **register file** e la **shared memory** vengono **suddivisi** tra tutti i threads residenti



Multidimensionalità degli IDs

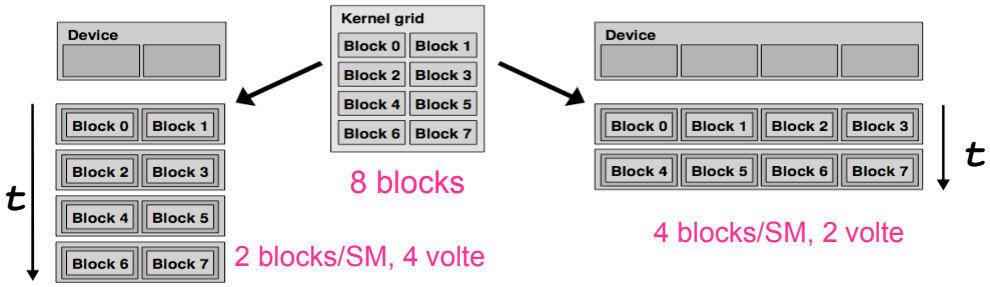
- Ogni blocco ed ogni thread vengono identificati univocamente da un ID:
- Block_ID: **1-D** or **2-D**
- Thread_ID: **1-D**, **2-D** or **3-D**

- IDs multidimensionali risultano molto utili nell'impostazione di calcoli con parallelismo multi-dimensionale
  - Image processing (2-D)
  - Volume processing
    - Soluzione di equazioni differenziali su domini 2D/3D
    - Tomografia



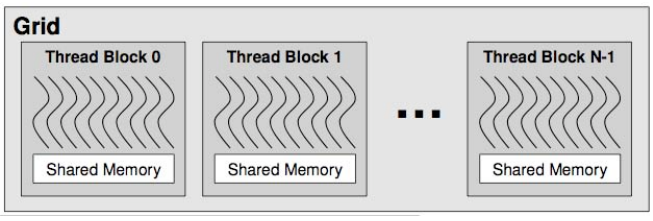
Gestione dei threads

Transparent scalability: l'utente decide la suddivisione in grids/blocks l'hardware decide in maniera autonoma la distribuzione dei blocks su ogni multiprocessore. Ogni kernel viene "scalato" sui multiprocessori disponibili.



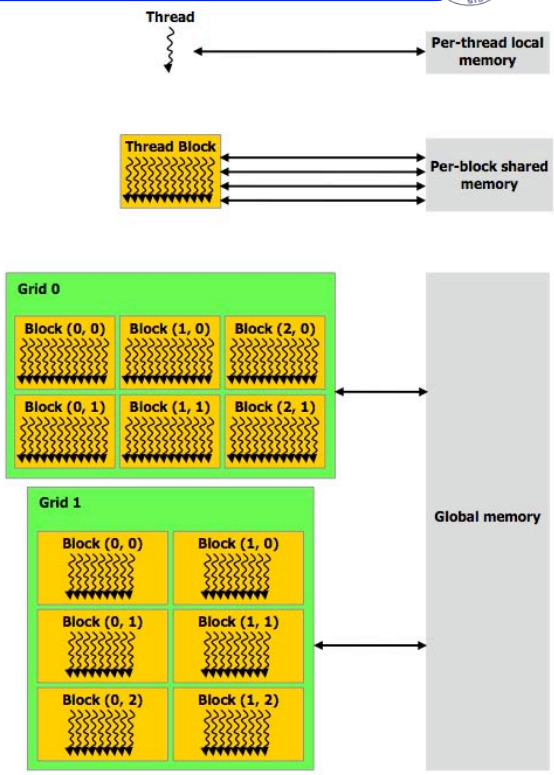
Visibilità della shared memory

- I threads di un block vedono una shared memory con cui cooperare
- threads in blocks differenti vedono shared memory diverse



Organizzazione gerarchica della memoria

- **Register file:** area di emoria privata di ciascun thread (var. locali)
- **Shared memory:** accessibile a tutti i threads dello stesso block. Può essere usata sia come spazio privato che come spazio condiviso
- Tutti i threads accedono alla medesima **global memory** (off-chip DRAM).
- Memorie read-only accessibili da tutti i threads: **constant** e **texture memory**.
  - dotate di **cache locale** in ogni SM
- *Global, constant e texture memory sono memorie **persistenti tra differenti lanci di kernel** della stessa applicazione.*

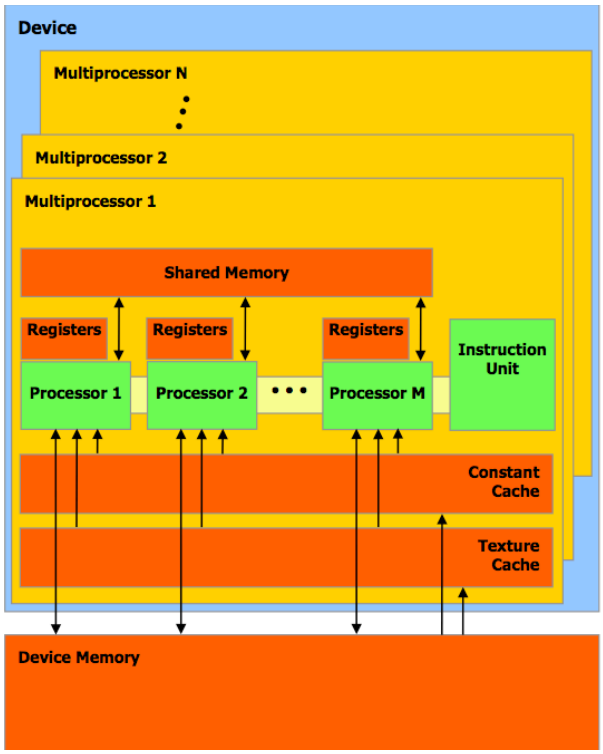


Organizzazione gerarchica della memoria

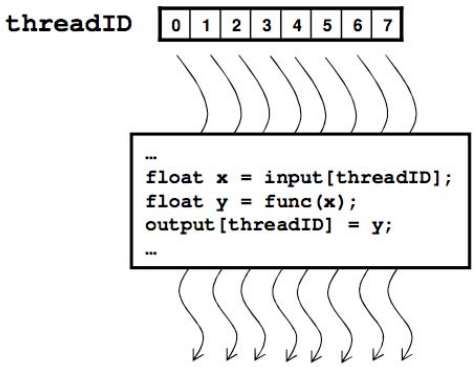
Gerarchia di memorie in devices CUDA:

- **Register file**
  - velocissimi, locali al thread
- **Shared memory**
  - velocissima, globale ai threads di un block, locali al block (SM)
- **Device memory**
  - velocissimi, locali al thread
  - cache: **constant, texture memory**

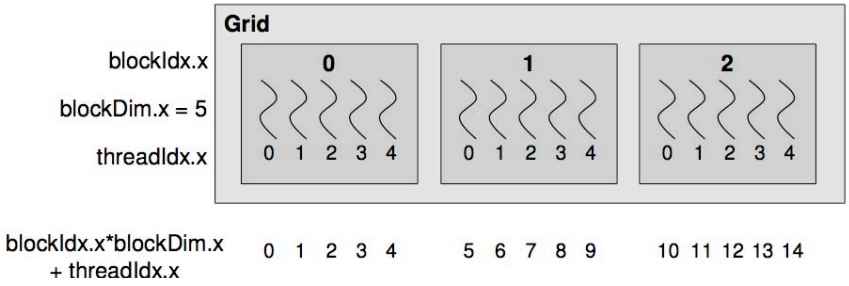
- Tale organizzazione gerarchica fornisce una notevole flessibilità di utilizzo e permette di ottimizzare la velocità di accesso ai dati ed il transfer rate tra memoria e processore
- Si minimizza così il **transfer rate bottleneck**



Distribuzione dei dati sui threads



- Un kernel CUDA è eseguito da un array di threads
  - Tutti i threads eseguono lo stesso codice
  - Ogni thread è identificato da un ID (**`threadID`**) che viene utilizzato come parametro per accedere ai dati.
- Sono definite le varabili globali:
  - **`threadIdx.x`**: **thread ID** nel **block**
  - **`blockIdx.x`**: **block ID** nella **grid**
  - **`blockDim.x`**: **n. di threads** in un **block**



Considerazioni sul parallelismo in CUDA

CUDA offre una duplice possibilità di parallelismo:

1. Parallelismo funzionale

- Differenti parti di codice possono essere elaborate indipendentemente da unità funzionali separate, su differenti unità computazionali
- Non viene definita una associazione esplicita fra il kernel ed i multiprocessori
- Non c'è parallelismo di kernel (viene sempre eseguito un solo kernel alla volta)

2. Parallelismo sui dati

- I dati vengono elaborati in parallelo mediante loro distribuzione sulle unità di elaborazione, le quali eseguono tutte pressoché la medesima funzione algoritmica.

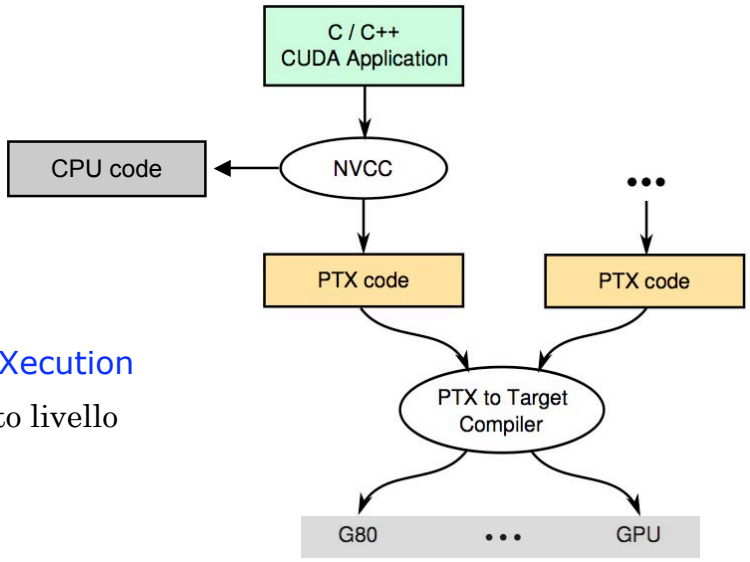
- **Streams and kernels** approach:
  - uno o più streams (flussi consistenti di dati omogenei) in ingresso possono essere elaborati da kernels e trasformati in uno o più streams in uscita
  - Il lancio del kernel funge da meccanismo di sincronizzazione

Outline

- Introduzione
- CUDA hardware
- Programmazione CUDA
  - CUDA software architecture
  - GPU memory management
  - Ottimizzazione dell'accesso ai dati
  - Sincronizzazione
  - Aspetti di calcolo
- Conclusioni

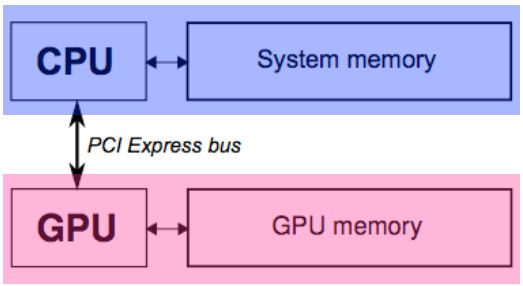
Compilazione di codice CUDA

- **CUDA API:** come estensione del linguaggio **C**
  - GPU Memory management
  - Comunicazione GPU–CPU
  - Programmazione e lancio di kernel
- Compilatore CUDA:
  nvcc
- **PTX**: Parallel Thread eXecution
  - GPU Assembly ad alto livello
  - Device-independent



Gestione memoria della GPU

- CPU e GPU dispongono di aree di memoria separate



- E' sempre **l'host** (CPU) a gestire la memoria del **device** (GPU):
  - Gestisce soltanto la global device memory (GPU RAM)

- **CUDA API** fornisce funzioni per effettuare:
  - **Allocate / free**
  - **Copia** di dati tra host e device (**device_to_host** e **host_to_device**)

Gestione memoria della GPU

- GPU Memory Allocation / Release

```
cudaMalloc(void** pointer, size_tnbytes)
cudaMemset(void* pointer, int value, size_tcount)
cudaFree(void* pointer)
```

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int *d_a = 0;
cudaMalloc( (void**)&d_a,  nbytes);
cudaMemset( d_a, 0, nbytes);
...
cudaFree(d_a);
```

- CPU–GPU Data transfer:

```
cudaMemcpy( void* dst, void* src,
            size_tn bytes, cudaMemcpyKind direction);
```

  - `direction` specifica sorgente e destinazione (host or device):

```
cudaMemcpyKind := {HostToDevice, DeviceToHost, DeviceToDevice}
```

  - Funzione bloccante: ritorna quando la copia è completa, e non comincia la copia prima che tutte le CUDA calls precedenti siano completate.

Primitive CUDA – definizione di funzioni

- ***Function type qualifiers:*** *prefissi di caratterizzazione delle funzioni*
  - `__global__` : definisce una funzione GPU KERNEL. Può essere invocata solo dalla CPU (host), non dalla GPU (code must return `void`)
  - `__device__` : definisce un GPU KERNEL che può essere invocata solo dalla GPU stessa (device), non dalla CPU (code must return `void`)
  - `__host__` : funzione eseguibile ed invocabile solo dalla CPU (host) – (default)

- **Lancio di un kernel**
  - Call syntax:

```
kernel_func <<< dim3 grid, dim3 block >>> (...)
```

  - **Execution Configuration**: `<<< >>>`:
  - **grid** dimensions: x, y
  - **thread-block** dimensions: x, y, z
  - input arguments:

```
dim3 grid(16,16);
dim3 block(16,16);
kernel<<<grid, block>>>(...);

kernel<<<32, 512>>>(...);
```


Esempio: codice CUDA

Confronto:

codice seriale
vs.
codice CUDA parallelo

Computing y = ax + y with a serial loop:

```
void saxpy_serial(int n, float alpha, float *x, float *y)
{
    for(int i = 0; i<n; ++i)
        y[i] = alpha*x[i] + y[i];
}

// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

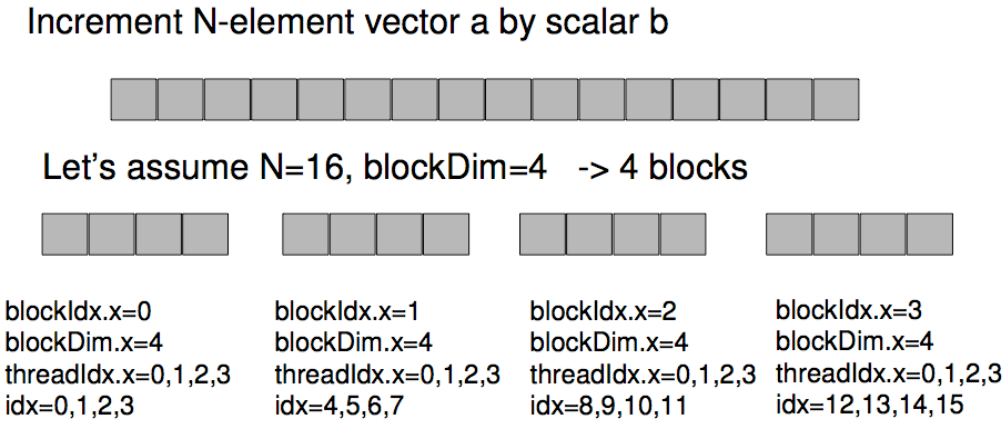
Computing y = ax + y in parallel using CUDA:

```
__global__
void saxpy_parallel(int n, float alpha, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if( i<n )  y[i] = alpha*x[i] + y[i];
}

// Invoke parallel SAXPY kernel (256 threads per block)
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

Esempio: somma di uno scalare ad un vettore

- L'indice del vettore è funzione del thread ID (thread_ID, block_ID)
  - Vettore di 16 elementi
  - **`BlockDim = 4 → threadId = 0,1,2,3`**




```
int idx = blockDim.x * blockId.x + threadIdx.x;
```


Example: Adding a scalar to a vector

CPU program

```
void increment_cpu(float *a, float b, int N)
{
        for (int idx = 0; idx<N; idx++)
            a[idx] = a[idx] + b;
}

void main()
{
   .....
      increment_cpu(a, b, N);
}
```

CUDA program

```
__global__ void increment_gpu(float *a, float b, int N)
{
      int idx = blockIdx.x * blockDim.x + threadIdx.x;
      if (idx < N)
          a[idx] = a[idx] + b;
}

void main()
{
   ...
      dim3 dimBlock (blocksize);
      dim3 dimGrid( ceil( N / (float)blocksize) );
      increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

Esempio: codice **host**

Operazioni da compiere:

1. Caricamento dati nella GPU
2. Chiamata kernel
3. Recupero risultati dalla GPU

```
// allocate host memory
int numBytes= N * sizeof(float);
float* h_A = (float*) malloc(numBytes);

// allocate device memory
float* d_A = 0;
cudaMalloc((void**)&d_A, numbytes);

// copy data from host to device
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);

// execute the kernel
increment_gpu<<< N/blockSize, blockSize>>>(d_A, b);

// copy data from device back to host
cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);

// free device memory
cudaFree(d_A);
```


Tipi di dato predefiniti

- In CUDA sono predefiniti alcun tipi di dato, la cui gestione ed elaborazione è ottimizzata. Possono essere usati sia in codice GPU che CPU.

- **Scalar data types:**
  - Standard C types: `[u]char, [u]short, [u]int, [u]long, float`
  - Manca il `double`...

- Built-in **vector types**
  - `[u]char[1..4],`
  - `[u]short[1..4],`
  - `[u]int[1..4],`
  - `[u]long[1..4],`
  - `float[1..4]`

- Structures accessed with **x, y, z, w** fields:

```
uint4 param;
int y = param.y;
```

- `dim3`
  - Based on `uint3`
  - Used to specify dimensions

Variabili in CUDA

Variable Qualifiers (GPU code): prefissi di caratterizzazione delle variabili

- `__device__`
  - La variabile sarà allocata in **device memory** (large memory, high latency, no cache)
  - Allocazione mediante: `cudaMalloc( __device__ qualifier ... )`
  - Accessibile da tutti i threads (variabile globale)
  - Lifetime: applicazione
  -
- `__shared__`
  - La variabile sarà memorizzata nella **shared memory** (very low latency)
  - Accessibile dai threads appartenenti al medesimo block
  - Lifetime: esecuzione del kernel

- Variabili senza caratterizzazione
  - Scalari e vettori built-in: memorizzati nei **registers**
  - Arrays di più di 4 elementi: memorizzati in **device memory**

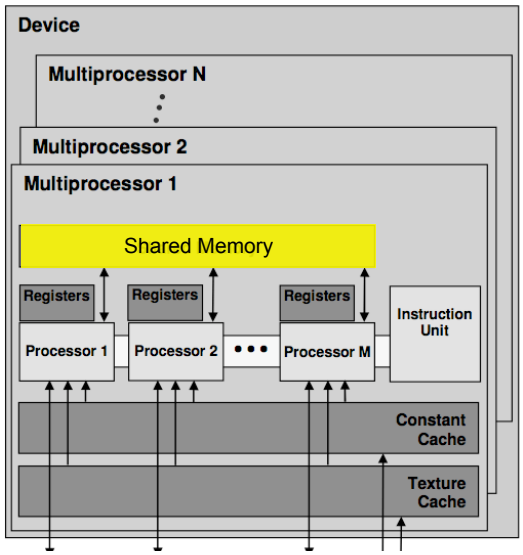
Accesso ottimizzato alla memoria

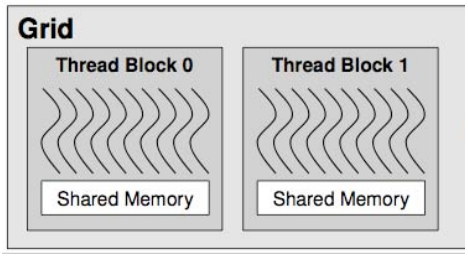
Tecniche di ottimizzazione:

- Ottimizzazione accesso alla Global memory: Coalescence
- Ottimizzazione accesso alla Shared memory: Bank conflicts
- Ottimizzazione accesso ai dati: Texture memory

Shared Memory

- Approx. 100 volte più veloce della global memory
  - Conviene sfruttarla per ridurre l'accesso alla global memory
- I threads di un block possono condividere informazioni attraverso la shared memory



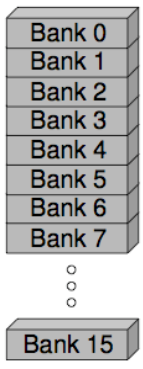


Shared Memory

- **Parallel Memory Architecture**
  - di regola molti threads accedono alla shared memory simultaneamente
- La Shared memory è divisa in banchi: **banks**
  - Ogni banco può "servire" un thread per ciclo di clock
  - La shared memory può "servire" simultaneamente tanti threads quanti sono i banchi (**GeForce 8x**: 16 threads per SM ➔ 16 banks)
- **Accesso simultaneo...**

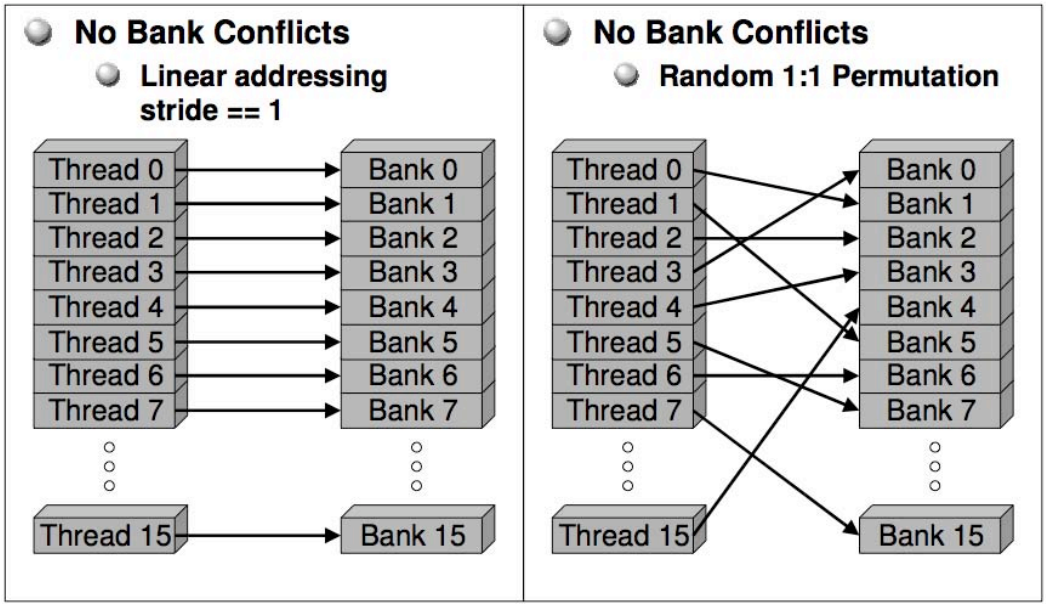
  ... a banchi differenti avviene **in parallelo (1 ciclo di clock)**

  ... allo stesso banco causano un **bank conflict**
  - l'accesso avviene quindi in modo **serializzato**
- La Shared memory è veloce come i registri, se non ci sono conflitti !



Shared memory

- Esempio: nessun conflitto

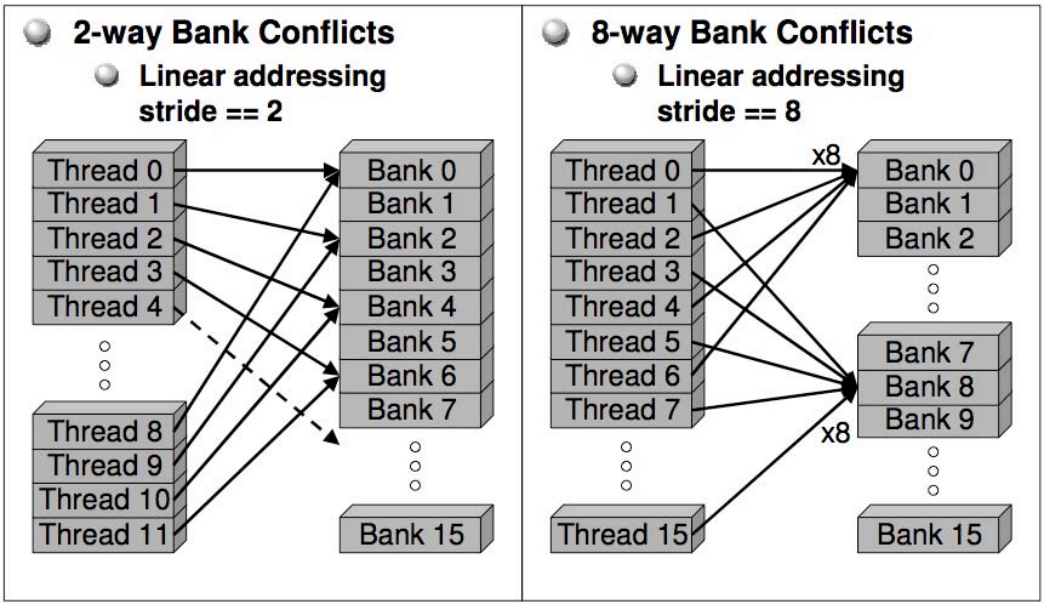


Shared memory

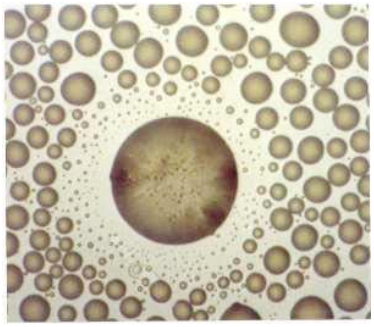
- Esempio: bank conflicts

2 cicli di clock

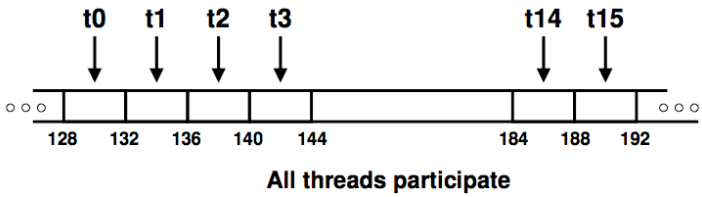
8 cicli di clock



Coalescence



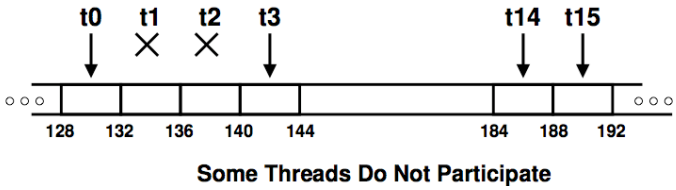
- *"The global memory access by all threads of a warp is coalesced into a single memory transaction if the words accessed by all threads lie in the same segment of size equal to:*
  - *32 bytes if all threads access 1-byte words,*
  - *64 bytes if all threads access 2-byte words,*
  - *128 bytes if all threads access 4- or 8-byte words"*
- Inoltre valgono le seguenti restrizioni:
  - **L'indirizzo di partenza (starting address)** deve essere un **multiplo** della dimensione della regione
  - Il **thread k-esimo** dell'half-warp deve accedere al ***k-esimo* elemento**.
- *Example: reading float*
- **Accesso "Coalesced"**
  - Accesso ordinato
  - "Starting address" multiplo di 128
  - Tutti i threads partecipano all'accesso



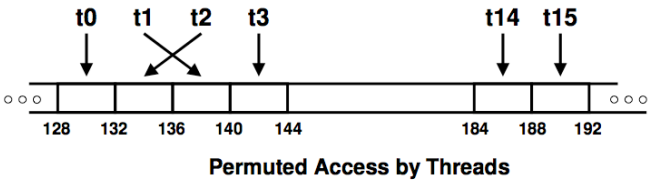
Coalescence

Reading float:

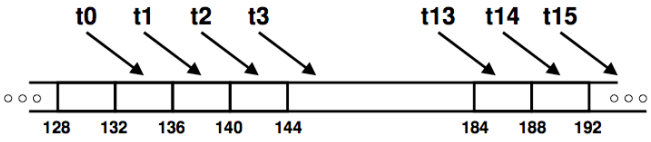
- **Coalesced access:**
  - data padding



- **Uncoalesced access:**
  - Permuted access
  - Accesso ordinato, ma starting address disallineato (non multiplo di 128)



- Tempi:
  - Accesso Uncoalesced è approx. **10 volte più lento** che coalesced.

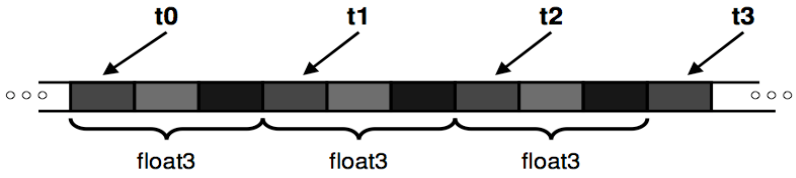


Coalescence

- *Example: uncoalesced float3 code*

```
__global__ void accessFloat3(float3 *d_in, float3 d_out)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  float3  a = d_in[index];
  a.x += 2;
  a.y += 2;
  a.z += 2;
  d_out[index] = a;
}
```

- float3 is 12 bytes, ≠ 4, 8, or 16
  - Each thread ends up executing 3 reads
  - Half-warp reads three (16*4=64 Byte)-wide non-contiguous regions

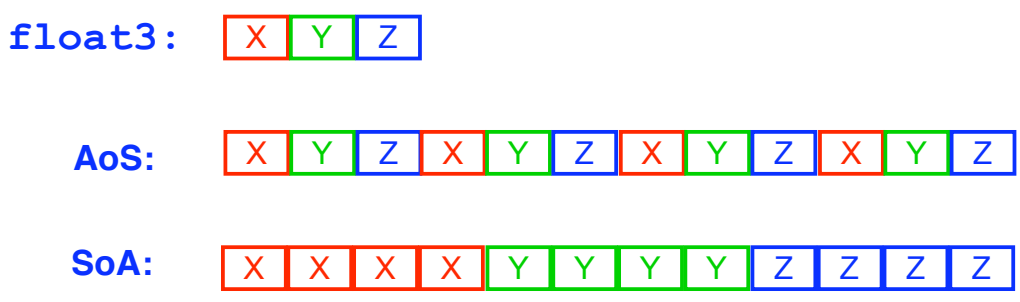


Coalescence: SoA vs. AoS

Strategie per evitare l'accesso uncoalesced:

- Uso di **Structure of Arrays (SoA)**
  al posto di **Array of Structures (AoS)**

- Se SoA non è utilizzabile:
  - Forzo l'allineamento: `__align(X)` , X = 4, 8, or 16
  - Uso Shared Memory



Coalescence: use of Shared Memory

- Use shared memory to allow coalescing
  - Need `sizeof(float3)*(threads/block)` bytes of SMEM
  - Each thread reads 3 scalar floats:
  - **Offsets: 0, (threads/block), 2*(threads/block)**
  - These will likely be processed by other threads ➔ **coalescence!**

- Processing
  - Each thread retrieves its float3 from SMEM array
  - Use **thread ID** as index
  - Rest of the code does not change!

Coalescence: use of Shared Memory

- Coalesced float3 code:

```
__global__ void accessInt3Shared(float *g_in, float *g_out)
{
  int index = 3 * blockIdx.x * blockDim.x + threadIdx.x;
  __shared__ float s_data[256*3];
  s_data[threadIdx.x]       = g_in[index];
  s_data[threadIdx.x+256] = g_in[index+256];
  s_data[threadIdx.x+512] = g_in[index+512];
  __syncthreads();
  float3 a = ((float3*)s_data)[threadIdx.x];

  a.x += 2;
  a.y += 2;
  a.z += 2;

  ((float3*)s_data)[threadIdx.x] = a;
  __syncthreads();
  g_out[index]       = s_data[threadIdx.x];
  g_out[index+256] = s_data[threadIdx.x+256];
  g_out[index+512] = s_data[threadIdx.x+512];
}
```

Read the input through SMEM

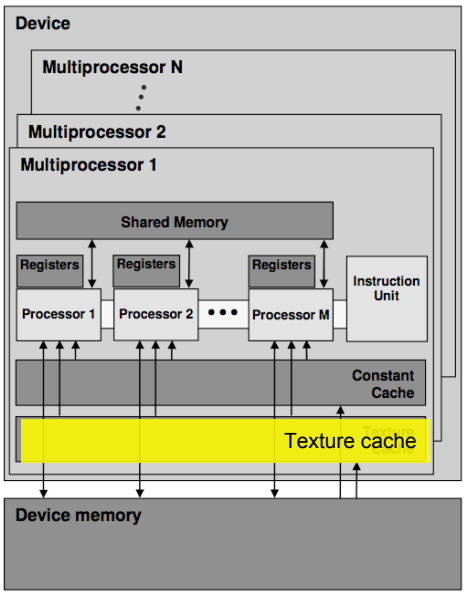
Compute code is not changed

Write the result through SMEM

Textures

- **`Texture`** è un oggetto per lettura dati, che presenta diversi vantaggi rispetto alla Constant Memory:
  - Presenza di **cache** (ottimizzata per località **2D**)
  - **Interpolazione** hardware (linear, bi-, tri-linear)
  - **Wrap modes** (per indirizzi "out-of-bounds")
  - Indirizzabile in 1D, 2D, or 3D
  - Possibilità di accesso con coordinate integer o **normalized coordinates**
- Uso:
  - **CPU** code: data binding a texture object
  - **Kernel** accede alla texture chiamando una fetch function:

    `tex1D(),tex2D(),tex3D()`

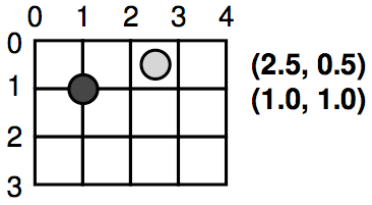


Texture Addressing

Interpolazione:

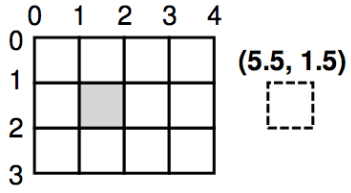
- Es: interpolazione bilineare:

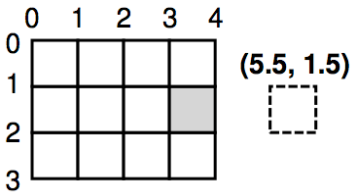
$$V(2.5,0.5) = \frac{1}{4}\left(V(2,0) + V(3,0) + V(2,1) + V(3,1)\right)$$



Wrap modes:

- Wrap
  - Coordinate Out-of-bounds coordinate vengono "arrotolate" (modulo DIM)
- Clamp
  - Coordinate Out-of-bounds sono sostituite dal bordo più vicino





Sincronizzazione tra threads

```
void __syncthreads();
```

- Sincronizza tutti i threads di un block
- Genera una barriera di sincronizzazione: nessun thread può superare la barriera finché non l'hanno raggiunta tutti.
- Usato per evitare gli RAW / WAR / WAW hazards nell'accesso alla shared memory
- È permesso in codice condizionale (presenza di branch) solo se la condizione è uniforme su tutti i threads del blocco.

Atomic operations su interi memorizzati nella global (device) memory:

- Operazioni associative su interi signed/unsigned: *add, sub, min, max, and, or, xor, increment, decrement, compare, swap, ...*
- `atomic_add(), atomic_sub(), ...`

Math issues

- Double precision mancante. (probabilmente nella prossima generazione di devices)
- Deviazioni dallo standard IEEE-754
  - Divisione non-compliant – Non tutti i rounding modes sono supportati
  - Numeri denormalizzati non supportati – Alcune eccezioni FPU non segnalate

	G8x	SSE	IBM Altivec	Cell SPE
Format	IEEE 754	IEEE 754	IEEE 754	IEEE 754
Rounding modes for FADD and FMUL	Round to nearest and round to zero	All 4 IEEE, round to nearest, zero, inf, -inf	Round to nearest only	Round to zero/truncate only
Denormal handling	Flush to zero	Supported, 1000's of cycles	Supported, 1000's of cycles	Flush to zero
NaN support	Yes	Yes	Yes	No
Overflow and Infinity support	Yes, only clamps to max norm	Yes	Yes	No, infinity
Flags	No	Yes	Yes	Some
Square root	Software only	Hardware	Software only	Software only
Division	Software only	Hardware	Software only	Software only
Reciprocal estimate accuracy	24 bit	12 bit	12 bit	12 bit
Reciprocal sqrt estimate accuracy	23 bit	12 bit	12 bit	12 bit
log2(x) and 2^x estimates accuracy	23 bit	No	12 bit	No

Conclusioni

- Caratteristiche principali di CUDA ("Design Goals"):

  Hardware:
  - Scalabilità verso centinaia di cores / migliaia di threads paralleli
  - Gerarchia di memoria semplice e potente

  Software:
  - Estensione minimale al linguaggio C
  - Veloce curva di apprendimento: permette ai programmatori di concentrarsi sugli algoritmi, anziché sulle regole di programmazione parallela

- CUDA programming "golden rules":
  - Use parallelism efficiently
  - Coalesce memory accesses if possible
  - Take advantage of shared memory
  - Explore other memory spaces: Texture, Constant
  - Reduce bank conflicts

Conclusioni

- È CUDA il futuro del calcolo ad alte prestazioni?

- Multicore vs. DSP:
- **NVIDIA G8x:**
  - 200 € / 16 Multiprocessors
  - complete system < 1,000 €
  - Software development tools: 0 €

- **DSP (TI):**
  - 10 € / processor
  - complete system (hi-perf) > 10,000 €
  - Software development tools: ~ 5000 €

- Altre soluzioni multicore: IBM/Toshiba **CELL Processor (PS3)**

- Riferimenti:

  www.nvidia.com/cuda
  - **Documentazione:** CUDA Programming Guide, CUDA Reference Manual
  - **Software (free):** CUDA SDK (Windows, Linux, Mac OS/X), CUDA drivers, ...
  - CUDA forum