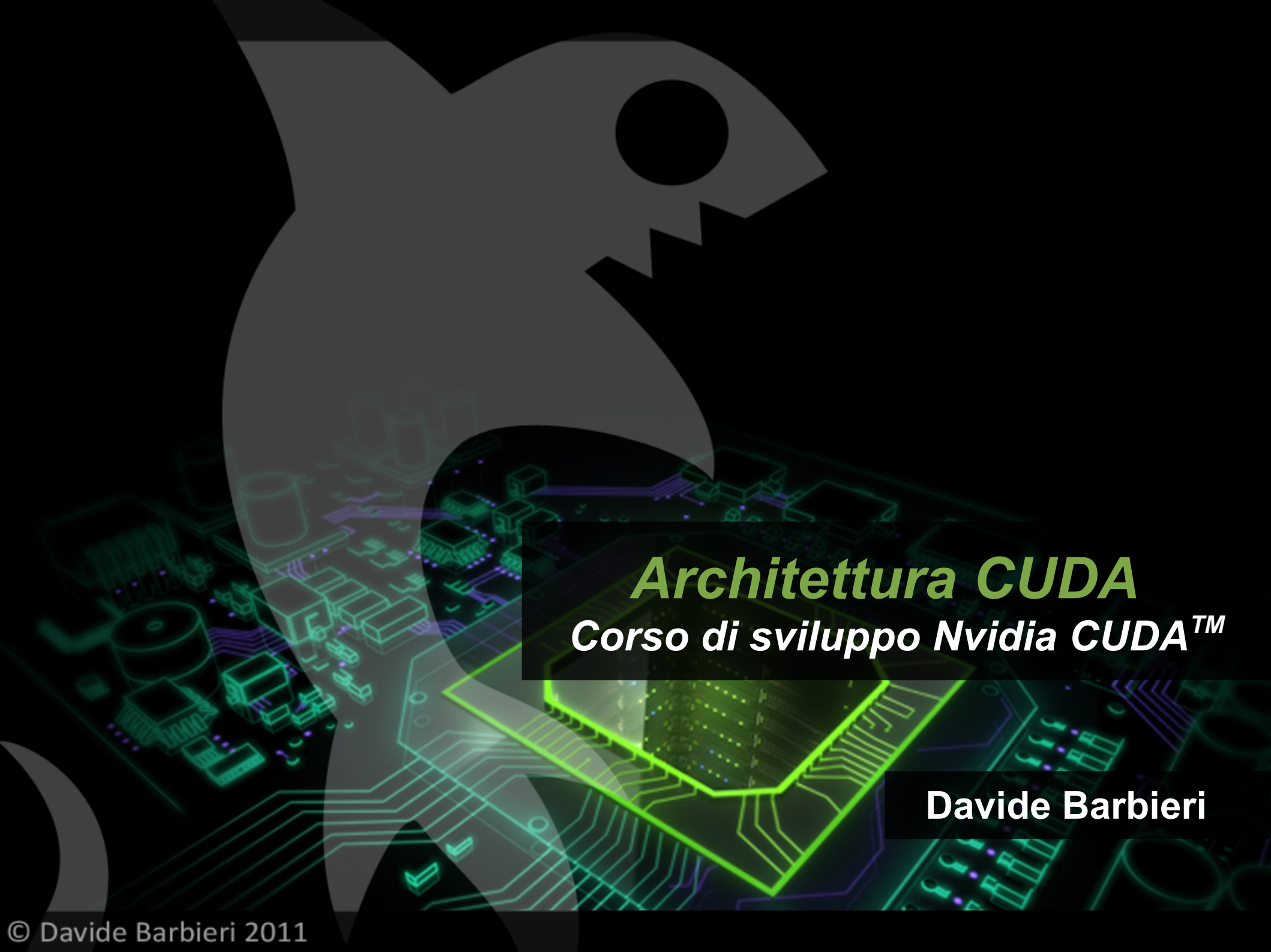


Architettura CUDA

Corso di sviluppo Nvidia CUDA™

Davide Barbieri

Panoramica Lezione

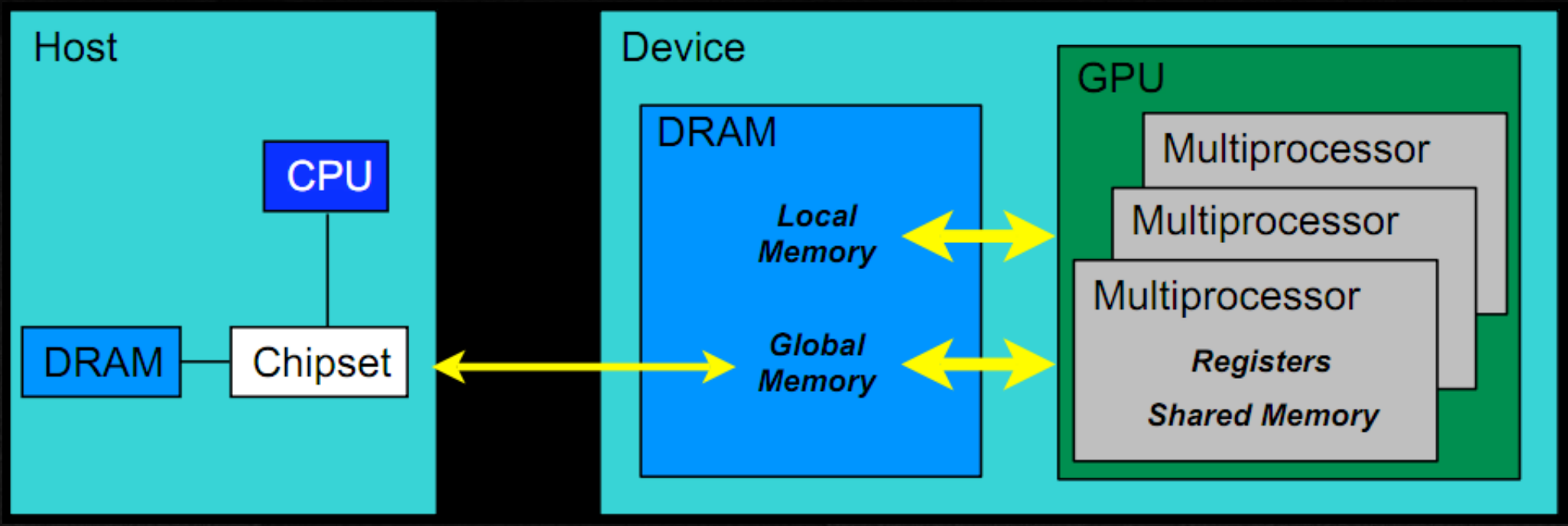
- Modello Architetturale CUDA
- Modello di programmazione CUDA
- “Hello World” in CUDA
- Gestione degli errori

Terminologia

- *Host* – La CPU e la sua memoria
- *Device* – La GPU e la sua memoria
- *Kernel* – Una funzione eseguita su GPU tramite una *chiamata remota*

Modello Architettura CUDA

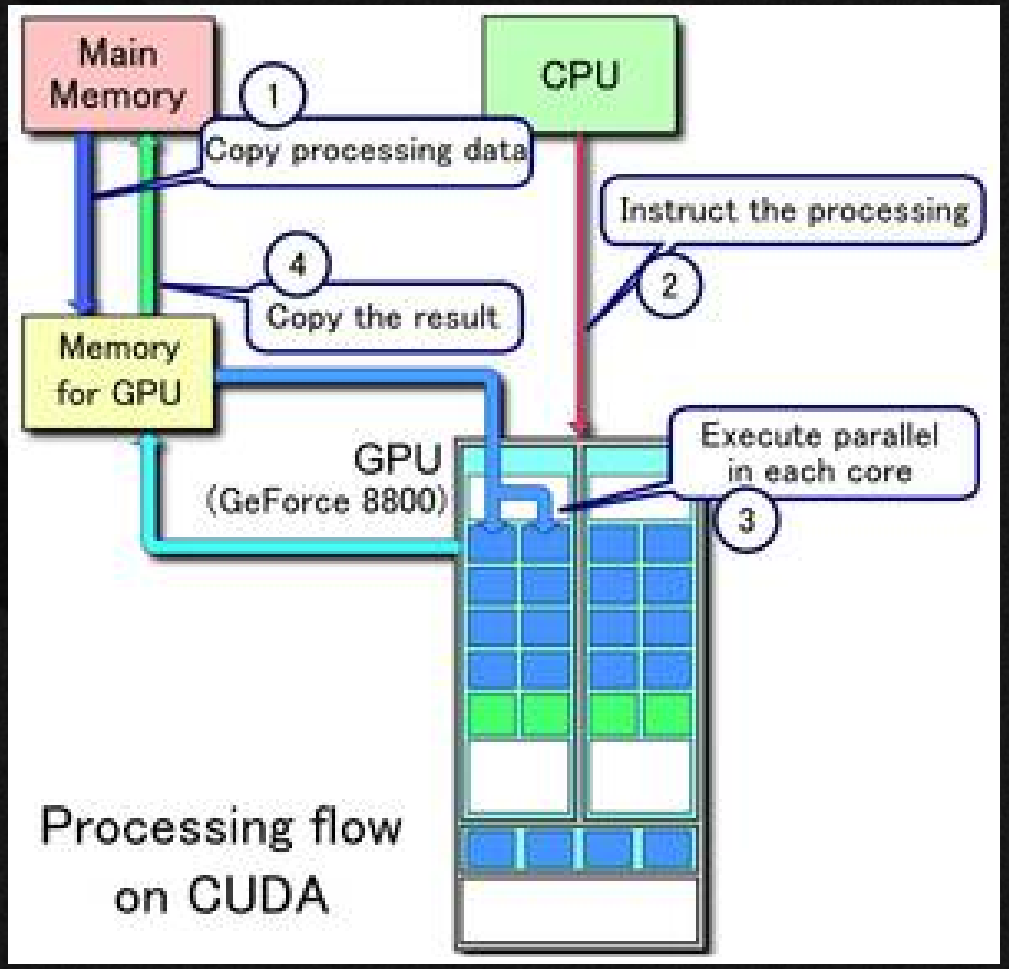
- Array di M ***Streaming Multiprocessor*** (***SM***)
- Ogni SM ha N processori scalari
- Ogni SM ha una memoria **On-Chip** veloce
  - (latenza ~ accesso a registro)
- ***Diverse Astrazioni per accedere alla m***emoria *RAM GDDR* dedicata
- Solo nelle ultime versioni, una gerarchia di *cache*



- L'host è connesso al device tramite il BUS **PCI-Express**
  - PCI-Express 2.0 → banda effettiva inferiore agli 8 GB/s
  - Pochi rispetto ai possibili 192 GB/s tra ***GPU*** ↔ Memoria GDDR
- L'host può leggere e scrivere dati dalla/sulla RAM GPU
  - Meglio limitare al più possibile queste *transazioni*
    - *Input del problema*
    - *Output della soluzione*

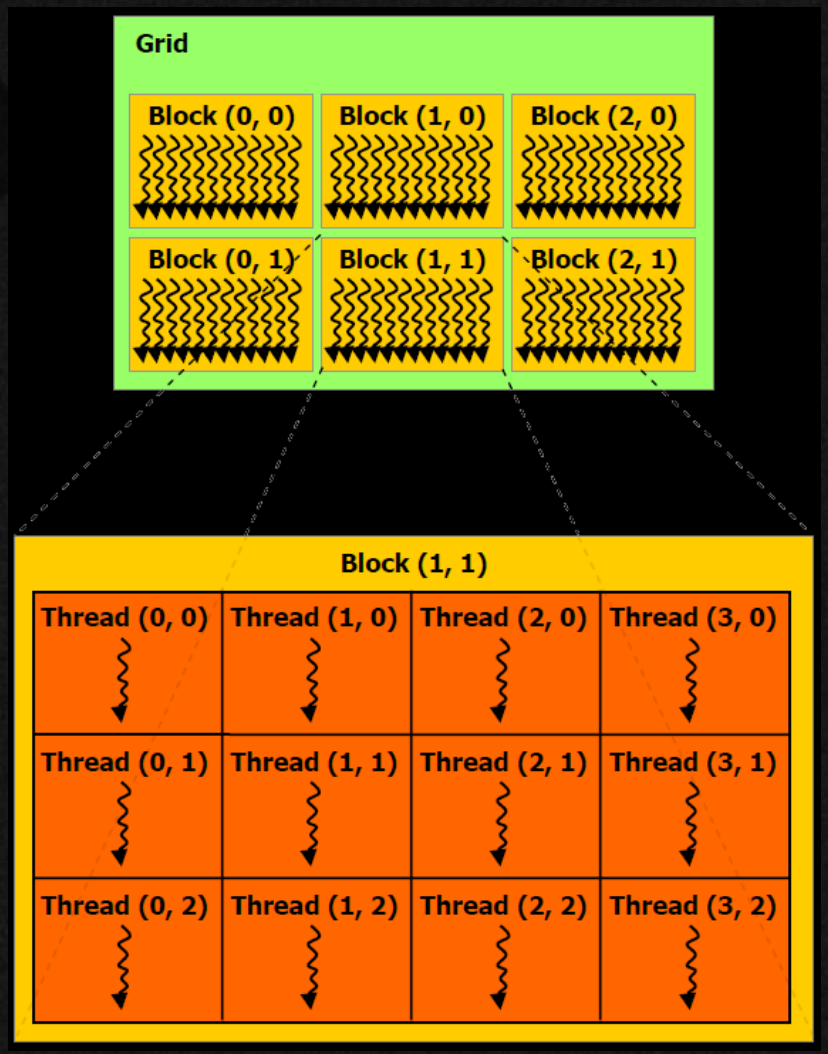
Flusso dei dati CPU ↔ GPU

1. Copio i dati di *input* da Host a Device
2. L'host effettua una chiamata remota verso il device
3. La GPU esegue il kernel
4. Copio i dati di output dal Device all'*Host*

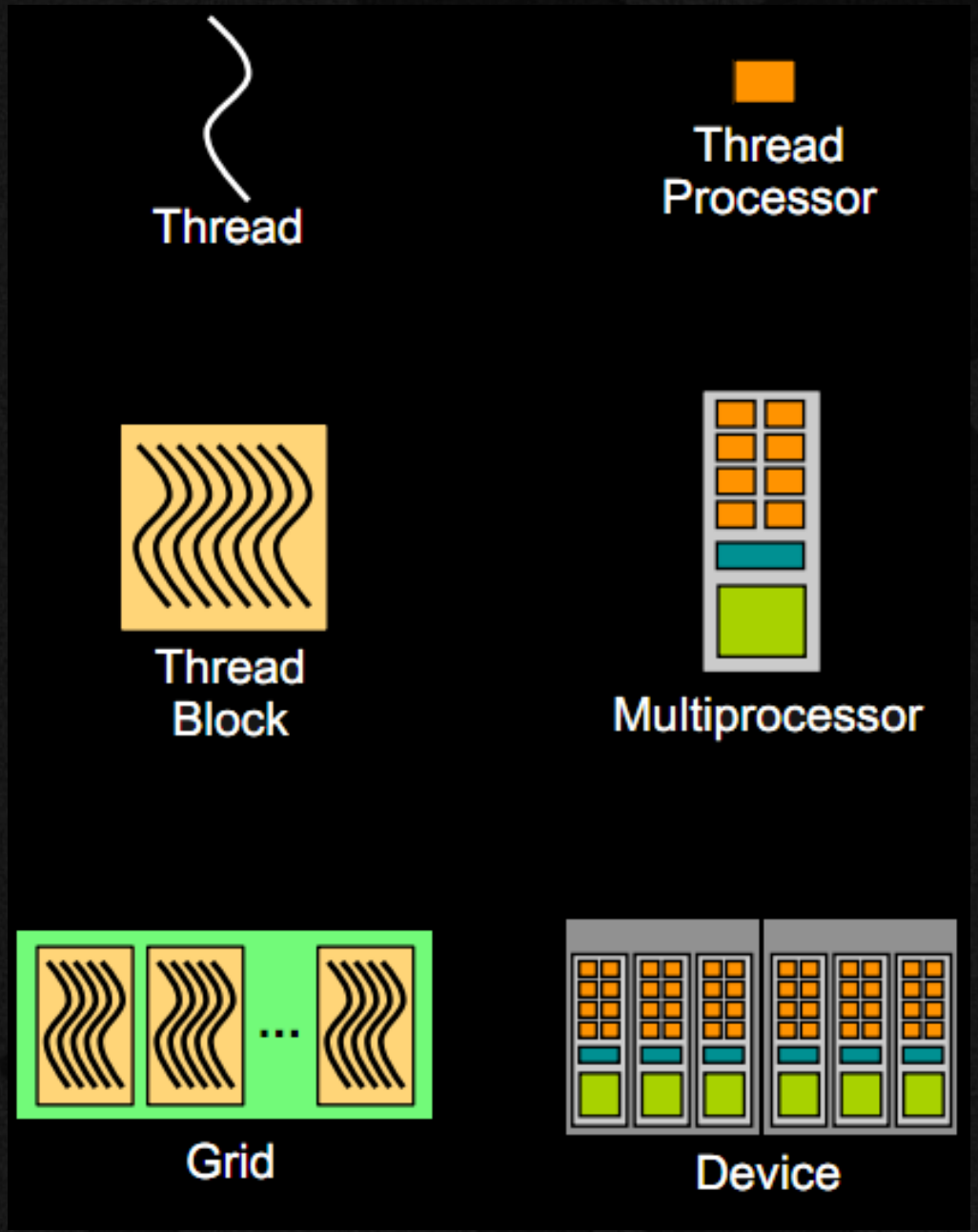


Modello di programmazione

- Gerarchia di threads
  - Thread
  - Blocco di thread
  - Griglia di blocchi
- Il programmatore stabilisce:
  - # thread per blocco
  - # blocchi
- Chiamata remota alla GPU
  - Blocco *schedulati* **automaticamente** su uno **SM** con **risorse disponibili**
- Il thread si specializza tramite:
  - Id del thread nel blocco
  - Id del blocco nella griglia



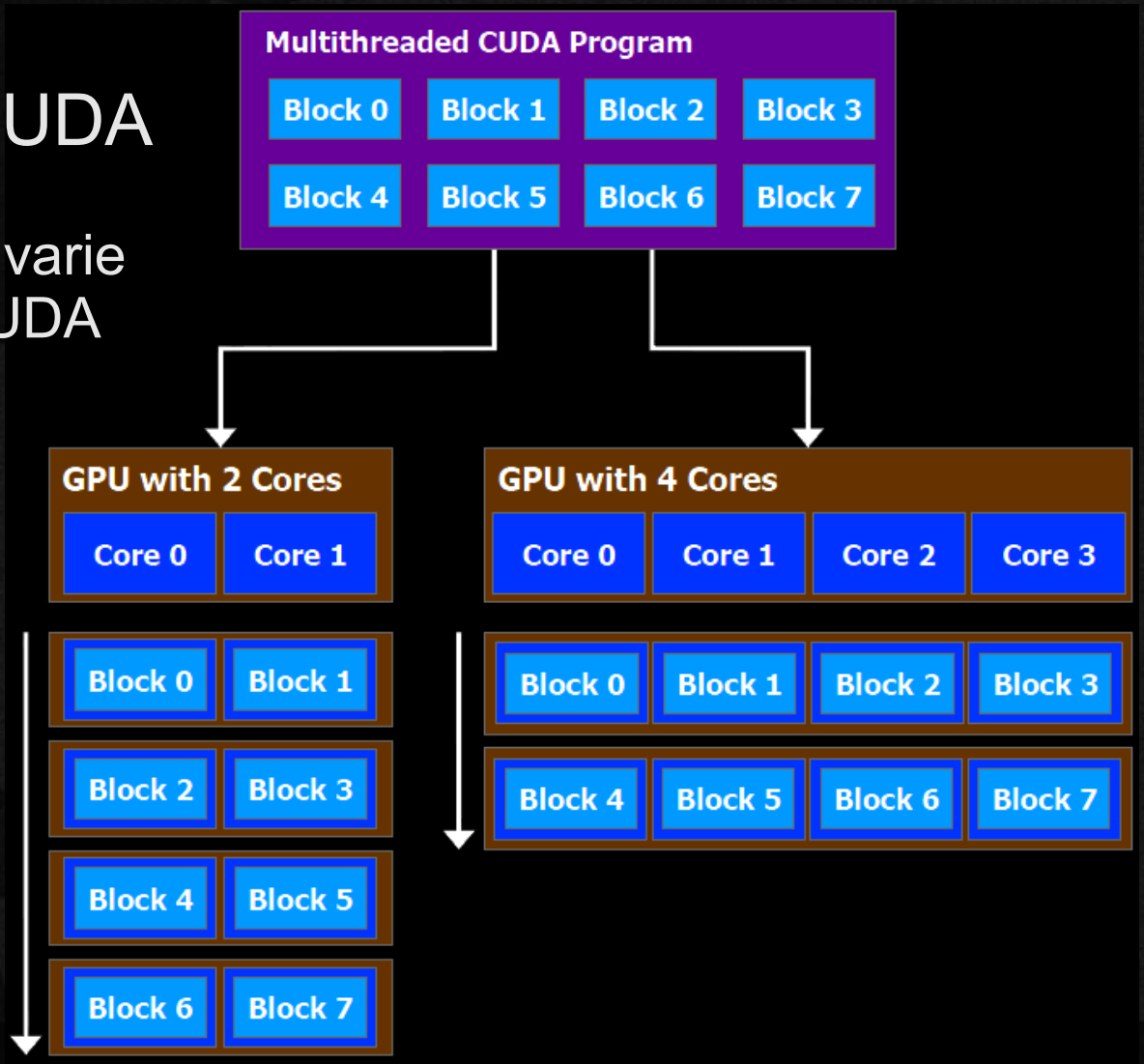
Mapping Software - Hardware

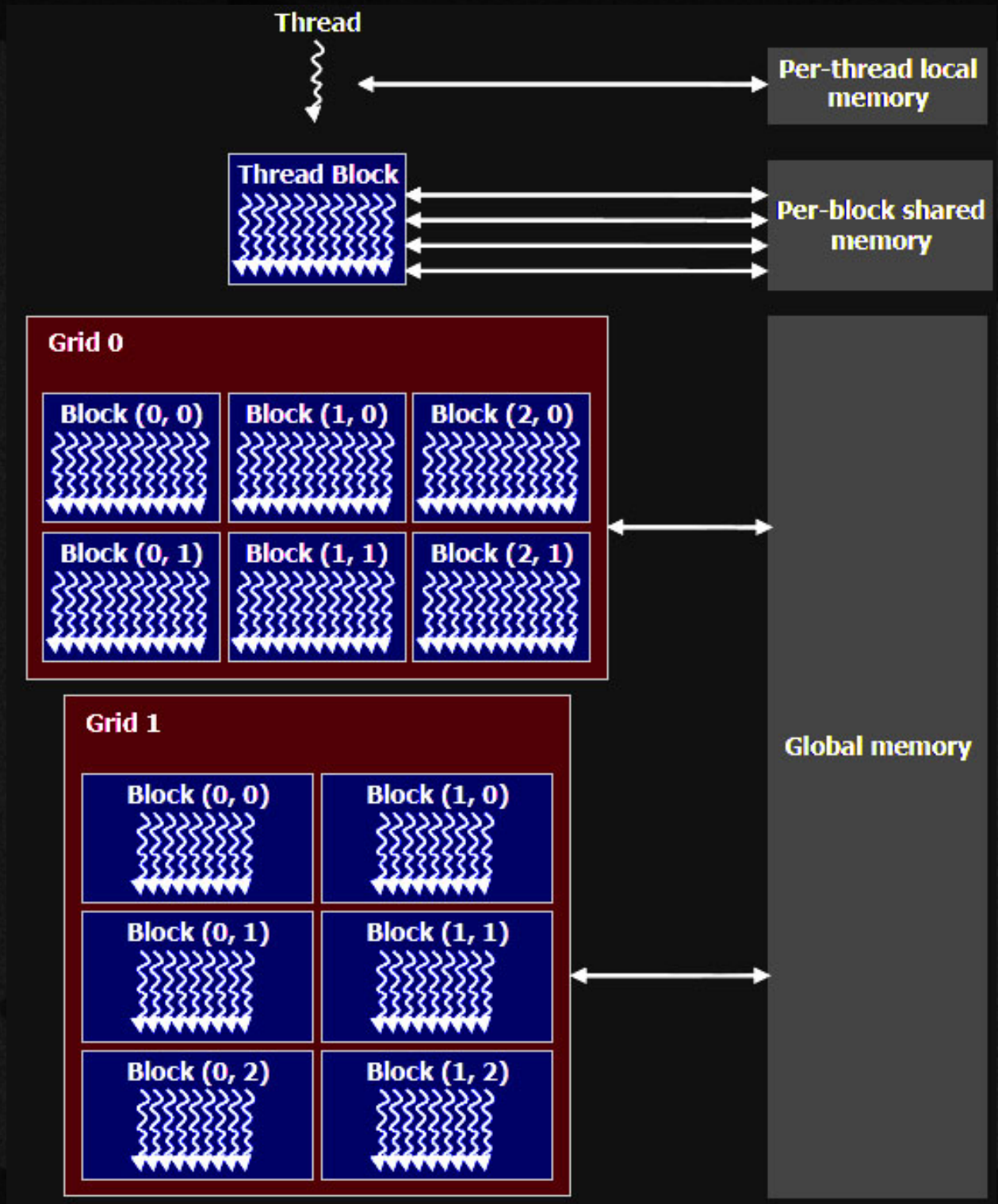


- Thread eseguiti dai CUDA core
- Blocchi eseguiti su multiprocessori che hanno risorse disponibili
- Una volta distribuiti, mantengono le risorse fino a fine esecuzione
- Una griglia viene eseguita su un dispositivo

Scalabilità Automatica

- Stabilito il numero di blocchi per una chiamata
  - Una GPU con più multiprocessori esegue in parallelo più blocchi alla volta
- Progettando un kernel CUDA
  - Ottimizziamo il codice per le varie *versioni* di architettura CUDA
  - Dopodiché, GPU con più multiprocessori eseguiranno **automaticamente** più velocemente il nostro kernel





- Ogni thread possiede:
  - Registri
  - Memoria locale (molto lenta rispetto ai registri)
- Ogni blocco possiede:
  - Memoria condivisa
- Tutti i thread accedono:
  - Memoria RAM device
  - Attraverso astrazioni differenti
    - Global Memory
    - Constant Memory
    - Texture Cache
    - Altro

Modello di Memoria

Memoria Globale

- Spazio di indirizzi lineari della memoria RAM
  - Accessibile e condivisa da ogni thread
  - Allocazioni hanno tempo di vita dell'applicazione

Memoria Locale

- Porzione di memoria RAM riservata ai thread
  - NON condivisa
  - Assegnata in compilazione quando non bastano registri e memoria condivisa
  - Tempo di vita della chiamata

Memoria Condivisa

- Memoria on-chip, accessibile velocemente dai thread dello stesso blocco

Memoria Texture

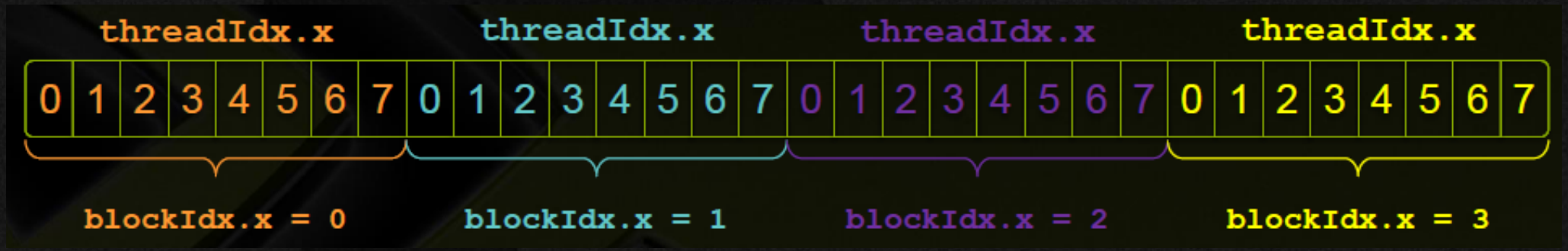
- *Astrazione per accedere a memoria RAM attraverso cache con località spaziale 2D*

Hello World: somma tra vettori

- Somma in parallelo tra due array A e B
  - Ogni thread *i* esegue la somma *A[i] + B[i]*
- Su cpu:
  - ```
    for (int i=0; i<N; ++i)
     C[i] = A[i] + B[i];
    ```
- Su gpu:
  - Avviamo una gliglia composta in totale da N thread
  - Ogni thread avrà un id nella griglia (**0 ≤ id < N**)
  - `C[id] = A[id] + B[id];`
    ```

Indicizzare i thread

- Variabili *built-in* a 3 dimensioni
  - *threadIdx* – Id del thread corrente nel blocco
  - *blockIdx* - Id del blocco a cui appartiene il thread corrente
  - *blockDim* - dimensione del blocco
  - *gridDim* – dimensione della griglia



- Nell'esempio:
  - *int index = threadIdx.x + blockIdx.x * blockDim.x;*
  - Unico per ogni thread nella griglia
  - Equivalente all'id dell'elemento da computare

Hello World!

```
__global__ void somma(int n, float* a, float* b, float* c)
{
 int id = blockIdx.x*blockDim.x + threadIdx.x;

 c[id] = a[id] + b[id];
}
```

- Codice Device all'interno dello stesso file del codice Host
  - Divisione tra C++ e CUDA da parte di *nvcc (compiler driver)*
- *__global__*
  - La funzione gira sul device ed è chiamabile dall'*host*
  - I puntatori *a,b* e *c* puntano a memoria device
- E' necessario considerare il caso in cui
  - *n* non sia multiplo della grandezza del blocco

Hello World!

```
__global__ void somma(int n, float* a, float* b, float* c);

int main(void) {
 ...
 /// somma 256 valori in memoria GPU
 somma<<<1,256>>>(256,a,b,c);
 return 0;
}
```

- Nello stesso file della funzione __global__
- *funzioneGlobal<<<dimGriglia,dimBlocco>>>()*
  - Esegue una chiamata remota alla GPU

Hello World

- Gli array *a*, *b* e *c* sono nella memoria device

- Gestione della memoria device tramite funzioni host:
  - cudaMalloc()
    - malloc() per le allocazioni device
  - cudaFree()
    - free() per le allocazioni device
  - cudaMemcpy()
    - memcpy() per copie
      Host → Device,
      Device → Device,
      Device → Host

- cudaDeviceSynchronize()

Hello World!

```
__global__ void somma(int n, float* a, float* b, float* c)
{

 int id = blockIdx.x*blockDim.x + threadIdx.x;
 if (id >= n)
   return;

 c[id] = a[id] + b[id];
}

int main(void) {
 ...
 /// somma 256 valori in memoria GPU
 somma<<<8,32>>>(256,a,b,c);

 /// somma N valori in memoria GPU
 somma<<<(N+31)/32,32>>>(N,a,b,c);

 return 0;
}
```

Funzioni __device__

- Abbiamo visto le funzioni *__global__*
  - Eseguite su **device**, richiamabili da h**ost**
- Funzioni *__device__*
  - Eseguite su **device**, richiamabili solo da **device**
  - Default, viene effettuato l'*inline*
    - ***Lo stack non esiste su GPU***
      (a parte sulle ultime GPU Nvidia)

```
__device__ float sum(float a, float b)
{
 return a + b;
}

__global__ void myKernel(float* a)
{
  //...
 a[id] = sum(a[id], a[id + 1]);
}
```

Cooperazione Thread

- I thread all'interno dello stesso blocco possono cooperare facilmente
  - Memoria Condivisa
  - Primitive di sincronizzazione

- Thread di blocchi differenti difficilmente cooperano
  - Ordine di esecuzione dei blocchi casuale

  - Un blocco possiede parte delle risorse di un multiprocessore fino a completa esecuzione

  - Nella maggiorparte dei casi per poter ordinare scritture/letture dalla memoria globale:
    - Iterazioni multiple della griglia di thread
      → Più chiamate remote al kernel

Memoria condivisa

- Dati condivisi tra tutti i thread del blocco
- Estensione del linguaggio CUDA
  - `__shared__ float temp[16];`
  - All'interno delle funzioni *device*
- Utile per:
  - Scambiare dati tra thread nel blocco
  - Evitare caricamenti ripetuti dalla memoria globale
  - Ottenere pattern di accesso alla memoria efficienti
    - (Vedremo nella prossima lezione cosa vuol dire)

__syncthreads()

- Funzione device built-in
  - void __syncthreads();
- Sincronizza i thread di un blocco
  - Ordinare letture e scritture su memoria condivisa
    - Esempio:
      - Ogni thread calcola la media dei 3 elementi vicini in memoria condivisa (tralasciando il controllo aggiuntivo per i thread del *bordo*)

```
__shared__ sMem[BLOCK_SIZE];

float center = sMem[threadIdx.x] = x[id];

__syncthreads();

float left = sMem[threadIdx.x-1];
float right = sMem[threadIdx.x+1];

float result = (center + left + right)/3.0f;
```

__syncthreads()

- Tutti i thread la devono eseguire
  - Attenzione a blocchi di codice condizionale

  - ERRATO:
    - e.g. if (threadIdx.x > 5) {
      A();
      __syncthreads();
      B();
      }

  - CORRETTO:
    - if (threadIdx.x > 5) { A(); }
      __syncthreads();
      if (threadIdx.x > 5) { B(); }

Hello World

- Perchè dovrei utilizzare un blocco con più di un thread?
  - E' indispensabile per permettere ai thread di **scambiarsi dati**
  - Anche se non si scambiano dati, per motivi di ***performance***

- I thread hanno un proprio contesto:
  - Program Counter
  - Registri
  - Memoria Locale

- A livello concettuale sono indipendenti
- L'hardware non li esegue però in maniera indipendente
  - Eseguiti in gruppi di 32, chiamati *Warp*
  - Se il warp esegue la stessa istruzione
    → Parallelo
  - Altrimenti
    → Branch divergenti vengono serializzati

Warp

- I thread in *CUDA* hanno propri contesti **indipendenti**
  - *Registri, Program counter, Memoria locale...*

- L'hardware però non li esegue indipendentemente
- I thread vengono eseguiti in gruppi di **32**, detti ***Warp***
- Il warp di appartenenza dipende dall'id del thread nel blocco

  tId = threadIdx.x + threadIdx.y*blockDim.x +
  threadIdx.z*blockDim.x*blockDim.y

  - *0 ... 31 → Warp 0,*
  - *32 ... 63 → Warp 1,*
  - *64 … 95 → Warp 2 , …*

- *Se i 32 thread del warp eseguono istruzioni diverse, il warp viene spezzato*
  - *Numero di Sotto-warp pari al numero di istruzioni diverse*
  - *Eseguiti in sequenza, anziché in parallelo (**Warp divergenti**)*

Come progettare un kernel CUDA

- Indagare il parallelismo a livello di dati
  - Stesse operazioni su molteplici dati
- Scomporre il problema in sottoproblemi di grandezza fissa (a meno di poche eccezioni)
- Sviluppare il codice del gruppo di thread che lo risolve
  - Dimensioni del blocco fisse
  - Dimensione della griglia variabili con la grandezza del problema
  - Prima il codice con input relativi ai casi favorevoli ad alte prestazioni
  - Poi Controllo/padding per gestire le eccezioni a questi casi
- Ulteriori ottimizzazioni

Gestione Errori (1/2)

- Tutte le chiamate CUDA ritornano un codice di errore di tipo *cudaError_t*
  - *Tranne le chiamate ai kernel*

- Per verificare un errore anche banale (e.g.dereferenziazione di un puntatore NULL):
  - *cudaError_t cudaGetLastError(void)*
    - Ritorna l'ultimo errore generato
  - *char* cudaGetErrorString(cudaError_t code)*
    - Rilascia la stringa che descrive l'errore

Gestione Errori (2/2)

- Le chiamate ai kernel sono asincrone
  - Errori conosciuti solo a fine esecuzione
- Necessario usare *cudaDeviceSynchronize()* prima
- Consiglio:
  - Controllare errori dopo ogni chiamata compilando in Debug

```
#ifdef _DEBUG
  cudaDeviceSynchronize();
  cudaError_t error = cudaGetLastError();
  if(error != cudaSuccess)   {
    printf("CUDA error: %s\n", cudaGetErrorString(error));
    exit(-1);
  }
#endif
```

Risorse

- CUDA Programming Guide
- CUDA Best Practices Guide
- CUDA Zone
  http://developer.nvidia.com/category/zone/cuda-zone