GPU Computing

1. Quali sono i meccanismi di sincronizzazione?

Solution: Si possono avere più **livelli di sincronizzazione**:

- **Livello di sistema**: per attendere che un dato task venga completato su host e device; la primitiva

  ```
  cudaError_t cudaDeviceSynchronize(void);
  ```

  blocca l'applicazione host finché tutte le operazioni CUDA su tutti gli stream non sono completate. Si tratta di una funzione host-side only (una volta usata lato device per gestire il parallelismo dinamico, ma ora deprecata);

- Non c'è una primitiva esplicita per la sincronizzazione a **livello di grid**, ma la si può ottenere (da CC 6 in avanti) lanciando un kernel cooperativo

  ```
  cudaLaunchCooperativeKernel(
      (void*)myKernel,
      gridDim, blockDim,
      kernelArgs, /*sharedMemBytes=*/0, /*stream=*/0);
  ```

  e all'interno del kernel

  ```
  grid_group grid = this_grid();
  // work work work ...
  // waits for _all_ blocks in *this* kernel
  grid.sync();
  ```

  Non ci devono essere ulteriori kernel attivi all'interno del device;

- **Livello di blocco**: per attendere che tutti i thread in un blocco raggiungano lo stesso punto di esecuzione. La primitiva

  ```
  __device__ void __syncthreads(void);
  ```

  impone a tutti i thread nel blocco corrente di attendere fino a quando tutti gli altri thread dello stesso blocco non hanno raggiunto quel particolare punto di esecuzione. Lo scopo principale è garantire la visibilità degli accessi alla memoria (rendere visibile le modifiche), in modo da evitare conflitti e race conditions. Se non tutti i thread all'interno del blocco arrivano alla primitiva si può avere un deadlock;

- **Livello di warp**: per attendere che tutti i thread all'interno di un warp raggiungano lo stesso punto di esecuzione. La primitiva

```
__device__ void __syncwarp(mask);
```

permette di avere una barriera esplicita per garantire la ri-convergenza del warp per le istruzioni successive. L'argomento `mask` è composto da una sequenza di 32 bit che permette di definire quali warp partecipano alla sincronizzazione (se omessa, di default tutti, ovvero `0xFFFFFFFF`).

Sincronizzazione **tramite stream**: tra stream non-NULL diversi non si ha nessuna dipendenza od ordinamento, mentre lo stream di default (`0`) ha un comportamento diverso, può essere:

- legacy: bloccante rispetto a tutti gli altri stream, un'operazione lanciata nel default stream non può iniziare finché non sono completate tutte le operazioni precedenti in qualsiasi altro stream (e viceversa);
- per-thread: disponibile da CUDA 7, ogni thread host ottiene il suo default stream, diventa non-bloccante rispetto agli altri stream

Sincronizzazione **tramite eventi**: all'interno degli stream si possono creare degli eventi tramite i quali è possibile avere anche sincronizzazione:

- Host-side: la primitiva

  ```
  cudaError_t cudaEventSynchronize(cudaEvent_t event);
  ```

  permette di attendere lato host finché l'evento specificato non viene completato; esiste una variante non-bloccante:

  ```
  cudaError_t cudaEventQuery(cudaEvent_t event)
  ```

  che permette di controllare se un evento è stato completato o meno, senza bloccare l'host;

- Stream-to-stream: per far attendere a uno stream il completamente di un evento su un altro stream. La primitiva:

  ```
  cudaError_t cudaStreamWaitEvent(
      cudaStream_t stream , cudaEvent_t event);
  ```

  permette di aspettare un evento su un altro stream (anche su altri device).

Sincronizzazione **implicita** dovuta a operazioni bloccanti: alcune operazioni causano sincronizzazione in quanto implicano un blocco su tutte le operazioni precedenti sul device corrente. In questo gruppo rientrano molte operazioni relative alla gestione della memoria.

2. Cosa sono local e constant memory?

Solution: In CUDA è presente una gerarchia di memorie, con diversi tipi di memoria al suo interno, ciascuno con dimensioni, banda e scopi specifici.

Local e constant memory sono due tipi di memoria programmabile esposti al programmatore:

- **Local memory**: memoria off-chip (quindi molto lenta), locale ai thread; risiede in global memory. Da CC 2.0 parti di questa sono in cache L1 e L2.

  Viene usata per variabili "grandi" (o la cui dimensione non è nota a compile time), oltre che per lo spilling dei registri (quando il kernel usa troppe variabili).

- **Constant memory**: si tratta di uno spazio di memoria di sola lettura, accessibile da tutti i thread. La si può dichiarare usando il qualificatore `__constant__`. Sono 64k per tutte le CC off-chip, con 8k di cache dedicata in ogni SM. Ha scope globale va dichiarata staticamente al di fuori dei kernel.

  Viene usata quando tutti i thread devono leggere dalla stessa locazione (raggiunge l'efficienza dei registri); in altri casi le performance sono significativamente minori.

In sintesi: local memory è lenta, serve quando registri e shared memory non bastano, la constant memory è una zona di sola lettura, ideale per accessi broadcast a piccole tabelle condivise.

3. Come si distingue SIMT di CUDA da SIMD? Fare un esempio in cui CUDA si comporta in maniera SIMD.

Solution: Single Instruction Multiple Data SIMD: Si tratta di un modello in cui, secondo la tassonomia di Flynn, sono presenti più unità di elaborazione e tutte eseguono lo stesso flusso di istruzioni, ciascuna operando su dati diversi.

Single Instruction Multiple Thread SIMT: Modello introdotto da CUDA che estende SIMD, fornendo a ogni unità di esecuzione (thread) la possibilità di divergere dalle altre, in base ai dati.

Il flusso di controllo parte parallelo, ma, in base ai dati, ogni thread può intraprendere un flusso diverso. Per fare ciò è necessario che ogni unità di esecuzione possieda un program counter e register set. In realtà, all'interno di CUDA, il PC è uno per ogni warp (gruppo di 32 thread), i quali eseguono le istruzioni in lock-step e nel caso di divergenza i diversi path vanno eseguiti serialmente.

Oltre al costo "architetturale", si ha un costo in termini di performance quando si incontra una divergenza (i path di esecuzione non sono allineati).

Quando tutti i thread eseguono la stessa istruzione, senza divergenze, il modello SIMT si comporta ugualmente a quello SIMD: si ha un'unica istruzione su dati diversi in parallelo.

Banalmente, qualsiasi codice senza possibilità di divergenze si comporta come SIMD

```
__global__ void vectorAdd(const float* A, const float* B,
    float* C, int N) {
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        if (i < N) {
            C[i] = A[i] + B[i];
        }
    }
```

In questo modo tutti i thread all'interno di un warp eseguono la stessa istruzione.

4. Meccanismi di sincronizzazione tra GPU e modalità di trasmissione tra queste.

Solution: Tra diverse GPU ci sono più metodi di **sincronizzazione** possibili:

- Il metodo più semplice è lasciare che sia l'host a sincronizzare tutte le GPU, la primitiva `cudaDeviceSynchronize()` permette di attendere il completamento di tutte le operazioni su tutte le GPU (il comando va ripetuto per ogni device)
- Per una gestione più flessibile si possono usare gli eventi CUDA; un evento è un marcatore all'interno di una stream su un device, un'altra GPU può "ascoltare" per attendere il completamento di un evento su un altro device, tramite `cudaStreamWaitEvent()` (bloccante) o `cudaStreamQueryEvent()` (non bloccante)
- La libreria NCCL (Nvidia Collective Communications Library) fornisce primitive di comunicazione con sincronizzazione implicita

Anche per **trasmettere dati** tra più GPU ci sono diverse modalità:

- La più semplice è via host: i dati vengono copiati sull'host e poi passati ai device a cui servono (tramite `cudaMemcpy()`)
- Se il P2P è abilitato, esistono primitive che permettono lo scambio dati tra GPU diverse, come ad esempio `cudaMemcpyPeer()`; esistono anche primitive asincrone come `cudaMemcpyAsync()` e `cudaMemcpyPeerAsync()`

- Usare unified memory: la memoria unificata permette di avere uno spazio di indirizzamento condiviso tra host e device, allocando con `cudaMallocManaged()` si può usare lo stesso puntatore su tutti i dispositivi
- Per primitive di comunicazione altamente ottimizzate per la comunicazione collettiva la libreria NCCL offre throughput elevato e bassa latenza

5. Spiegare la warp divergence e come ovviarla nel caso della reduction.

Solution: In CUDA, un warp è un insieme di 32 thread che vengono eseguiti sullo stesso Streaming Multiprocessor SM; la warp divergence si ha quando thread all'interno di uno stesso warp prendono path di esecuzione differenti (causa istruzioni di controllo condizionale).

Quando c'è una divergenza, all'interno di un warp, l'hardware deve serializzare i path di esecuzione, eseguendoli uno dopo l'altro, ogni volta disabilitando i thread che non devono entrare in quel ramo di esecuzione. Riduce il parallelismo all'interno del warp, degradando, anche significativamente le prestazioni.

Per la parallel reduction, l'approccio "naive" consiste nell'imitare la somma strided ricorsiva: al passaggio i si sommano elementi a distanza 2^i; in parallelo, questo attiverebbe 1 thread ogni 2^{i+1}, causando divergenza crescente (a ogni step si usano la metà dei thread precedenti, divisi sullo stesso numero di warp).

Per risolvere questo problema vogliamo usare thread adiacenti per fare le somme, "disaccoppiando" l'indice del dato dall'indice del thread. Calcoliamo l'indice del dato di cui si deve occupare ogni thread come `2*stride*tid`, in questo modo thread adiacenti si occupano di tutte le somme, rimuovendo la divergenza (i thread che andrebbero oltre la dimensione dell'array vanno disattivati).

Riorganizzare i pattern di accesso ai dati per "convertire" gli indici in modo che l'utilizzo dei thread sia allineato alla granularità del warp.

6. Mostrare l'architettura di uno SM.

Solution: Le GPU sono costituite da array di Streaming Multiprocessor SM, ognuno dei quali è pensato per supportare l'esecuzione concorrente di centinaia di thread. Si divide in gruppi di 32 thread chiamati "warp".

Ogni SM al suo interno è composto da:

- CUDA Core: le ALU per le operazioni intere o floating point

- Warp scheduler: a ogni ciclo di clock decidono quali warp sono pronti e possono essere mandati in esecuzione
- Dispatch unit: invia le istruzioni del warp selezionato alle varie execution unit
- Special Function Unit SFU: usate per calcoli complessi, svolti in modo hardware
- Eventuali unità specializzate, come Tensor Core o FP64
- Load/Store Unit LSU: per la gestione delle operazioni di lettura/scrittura in shared memory/cache L1
- Register file: insieme dei registri per i thread di uno SM, la dimensione limita il numero di thread residenti concorrentemente
- Cache L1/shared memory: memoria condivisa tra i thread del blocco, a bassa latenza
- Cache L2: condivisa tra tutti gli SM, gestisce il traffico verso la memoria globale
- Instruction Cache: per ridurre la latenza dovuta al fetch di istruzioni
- Texture & constant cache: cache separate per accessi read-only in maniera non sequenziale

7. Spiegare i diversi tipi di stream.

Solution: Stream