GPU Computing

Massimo Perego

Sezioni

1 Sincronizzazione

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. 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

3. Sincronizzazione tra più device che condividono lo stesso bus.

Solution: La sincronizzazione tramite stream ed eventi si può usare anche tra diversi device: un device può controllare il completamento di un evento appartenente a uno stream in esecuzione su un diverso device. La primitiva

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

permette di attendere l'esecuzione di un evento in maniera bloccante, altrimenti

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

permette di controllare se l'evento è completato o meno in maniera non bloccante.

Rimane sempre un opzione la sincronizzazione host-side tramite `cudaDeviceSynchronize()`, da effettuare per ogni dispositivo.

Esistono inoltre primitive di sincronizzazione esplicite per kernel cooperativi multi-device (`this_grid().sync()`).

La libreria NCCL (Nvidia Collective Communications Library) per la comunicazione collettiva fornisce primitive con sincronizzazione implicita al loro interno.

4. Cos'è la divergenza e qual è la sua relazione con la sincronizzazione a livello di blocco.

Solution: La divergenza è un fenomeno per cui i thread all'interno di uno stesso warp (gruppo di 32 thread) seguono percorsi di esecuzione diverse a causa di istruzioni di branching (`if`, `switch`, …). Questo porta a un degrado delle performance in quanto i flussi di esecuzione diversi sono eseguiti in modo seriale dall'hardware (si ha un singolo PC per warp), disattivando i thread "non interessati".

CUDA fornisce primitive per la sincronizzazione esplicita a livello di blocco:

```
__syncthreads();
```

Ovvero una barriera che devo raggiungere tutti i thread prima che la computazione possa proseguire.

Se una primitiva `__syncthreads()` è "nascosta" dietro a istruzioni condizionali e non viene raggiunta da tutti i thread del blocco si incorre in un deadlock: i thread che hanno raggiunto l'istruzione rimarranno fermi.

Non si può risolvere la divergenza a livello di blocco, anche se la sincronizzazione è posta in maniera corretta (non causa deadlock), la serializzazione dei percorsi di esecuzione avviene ugualmente. Per risolvere la divergenze si possono usare primitive di sincronizzazione a livello di warp (`__syncwarp()`) o tecniche per evitare del tutto istruzioni condizionali.

2 Memoria

1. 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.

2. Cosa sono le memorie pinned, zero copy e unified.

Solution: La memoria **pinned** è memoria host non paginabile dal sistema operativo, ovvero non può essere fatto lo swap su disco di quella zona di memoria. La si può allocare con `cudaHostAlloc()`, permette trasferimenti asincroni con maggiore throughput rispetto alla memoria paginabile (evita overhead dovuto al pinning temporaneo). Da notare che l'allocazione eccessiva potrebbe degradare le performance host.

La memoria **zero copy** si basa su memoria pinned mappata nello spazio di indirizzamento del device, permette alla GPU di accedere direttamente a pagine di memoria host senza copie esplicite di memoria. Può semplificare la programmazione, ma ha latenza più alta della global memory (i dati devono passare su PCIe) e banda limitata.

La memoria **unified** è un modello di memoria automatico in cui host e device condividono lo spazio di indirizzamento, tutte le CPU e GPU del sistema possono accedere a questa memoria. Il sistema sottostante si occupa di gestire le migrazioni di memoria secondo necessità, in maniera trasparente all'applicazione. Può essere allocata con

`cudaMallocManaged()` e permette di semplificare notevolmente la programmazione, evitando tutte le copie di dati esplicite, ma richiede overhead di migrazione quando viene fatto l'accesso ai dati.

3. Come usare la shared memory per le convoluzioni.

Solution: Un possibile schema per la convoluzione usando la shared memory è quello del tiling: a ogni blocco della grid viene affidato un "tile" (sotto-blocco) dell'immagine in input, ogni thread nel blocco si occupa di un prodotto della convoluzione.

In shared memory vanno caricati tutti i possibili dati a cui il blocco deve fare accesso, ovvero la porzione dell'immagine di cui si occupa, assieme all'area data dal raggio della maschera di convoluzione (un "halo" attorno al sotto-blocco stesso).

Rimane il problema di come caricare i blocchi in shared memory: i dati sono più del blocco stesso, sarebbe ottimale dividere il più equamente possibile il carico di lavoro.

Un'idea può essere dare un ordine ai thread e "ripetere" il thread block sopra la sotto-matrice formata da tutti i dati da caricare.

L'uso della shared memory permette di limitare gli accessi alla memoria globale, incrementando la velocità degli accessi successivi, ma sono da tenere in conto possibili bank conflict (la shared memory è divisa in bank, se due o più thread vogliono accedere alla stessa bank vanno serviti serialmente) e la quantità di memoria shared utilizzata influisce sul numero di blocchi attivi concorrentemente su un singolo SM.

4. Modalità di accesso alla device memory e performance.

Solution: In CUDA si ha una gerarchia di memorie:

- Registri: allocati per thread, estremamente veloci (latenza praticamente nulla), ma hanno capienza limitata e un uso eccessivo porta a spilling in memoria locale
- Shared Memory: memoria condivisa tra i thread di uno SM, latenza bassa, organizzata in bank: accessi senza conflitti di bank garantiscono throughput massimo. Dimensione comunque limitata e l'uso che ne fa ogni blocco determina il numero di blocchi che possono essere in esecuzione su uno SM concorrentemente
- Global Memory: memoria più grande e a più alta latenza ($400 \sim 800$ cicli), accessibile da tutti i thread e dall'host. Permette una buona latenza, ma solo se gli accessi sono coalescenti (raggruppati in transazioni larghe)

- Local Memory: risiede fisicamente nella global memory, quindi ha la stessa latenza, viene usata per variabili molto grandi o per lo spilling dei registri
- Constant & Texture memory: risiedono nella device memory, ma hanno una cache all'interno di ogni SM, sono usate per accessi uniformi read-only, ovvero quando tutti i thread devono accedere a una stessa zona di memoria (hanno la banda della device memory altrimenti). La texture memory è ottimizzata per dati e operazioni su dati espressi sotto forma di matrici

3 Architettura

1. Parallelismo dinamico.

Solution: Il parallelismo dinamico è una funzionalità introdotta dalle CC 3.5 che permette a un kernel in esecuzione di lanciare altri kernel, senza passare dall'host.

Elimina la necessità di comunicare con la CPU e permette pattern di programmazione ricorsivi e data-dependent. Si possono generare dinamicamente kernel in base ai dati, senza doverli richiedere alla CPU. Il lavoro può essere adattato in base a decisioni data-driven.

Da tenere sotto controllo il numero di kernel lanciati, solitamente non è necessario che ogni thread lanci un nuovo kernel.

Si ha una sincronizzazione implicita tra padre e figlio: il padre non può terminare prima del figlio. Rimane la possibilità di avere sincronizzazione esplicita.

2. Spiegare i diversi tipi di stream.

Solution: Uno stream CUDA è una sequenza di operazioni CUDA asincrone, eseguite nell'ordine fornito dall'host dalla GPU. Ogni stream è asincrono rispetto all'host ed è indipendente rispetto ad altri stream.

I tipi di stream sono:

- NULL stream o default stream: si tratta dello stream predefinito, dichiarato implicitamente, usato per i lanci di kernel quando non specificato altrimenti (oppure usando `0` come parametro). Il suo comportamento varia in base alla flag di compilazione `--default-stream`
  - `legacy`: gli stream NULL sono bloccanti rispetto agli altri stream, quindi le operazioni sullo stream NULL possono essere seguite solo quando hanno terminato tutti gli altri stream e viceversa

  - `per-thread`: ogni thread host ottiene il suo stream di default e si comportano come stream regolari, non sono bloccanti

- Stream dichiarati esplicitamente o non-NULL: si possono creare stream tramite primitive come `cudaStreamCreate()` e `cudaStreamCreateWithFlags()`. Di default sono indipendenti tra loro e bloccanti rispetto al NULL-stream, ma questo comportamento può essere modificato tramite flag

Si possono anche avere stream con priorità, `cudaStreamCreateWithPriority()`: uno stream con priorità più alta può prelazionare lavoro in esecuzione con priorità più bassa.

3. 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

4. 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.

5. 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 Librerie

1. Schema d'uso per la libreria cuBLAS.

Solution: La libreria cuBLAS è la versione accelerata su GPU delle routine classiche di algebra lineare, come prodotti di matrici, vettori e scalari.

Le funzioni hanno 3 livelli, rispettivamente per operazioni vettore-vettore, matrice-vettore e matrice-matrice.

Da notare che lavora in ordine column-major (come Fortran) e non row-major (come C/C++).

Uno schema d'uso tipico per usare cuBLAS è:

1. Creare un handle con `cublasCreateHandle()`
2. Allocare la memoria sul dispositivo
3. Popolare la device memory, ad esempio con `cublasSetVector()`
4. Effettuare le chiamate a libreria per le operazioni necessarie
5. Recuperare i dati dalla device memory, ad esempio con `cublasGetVector()`
6. Una volta terminato, rilasciare le risorse CUDA e cuBLAS con `cudaFree()` e `cublasDestroy()`

2. Schema d'uso per la libreria cuRAND.

Solution: La libreria cuRAND fornisce semplici generatori di numeri. Permette sequenze pseudo-random e quasi-random. Si compone di due header, la seconda è per generatori su device (opzionale).

Uno schema d'uso generico di cuRAND:

1. Creare un nuovo generatore del tipo desiderato con `curandCreateGenerator()`
2. Settare i parametri del generatore
3. Allocare la memoria device
4. Generare i valori casuali, ad esempio con `curandGenerate()`
5. Uso dei valori
6. Quando non serve più il generatore, va distrutto con `curandDestroyGenerator()`

3. Pseudocodice utilizzo di curand per simulare il lancio di n dadi.

Solution: Seguendo l'iter tipico di utilizzo di cuRAND:

```
// Set up di cuRAND
curandCreateGenerator(&generator, CURAND_RNG_PSEUDO_DEFAULT);
curandSetPseudoRandomGeneratorSeed(generator, seed_value);

// Allocazione della memoria
cudaMalloc(&d_raw,    N * sizeof(unsigned int));
cudaMalloc(&d_dice,   N * sizeof(unsigned char));

// Generazione dei valori
curandGenerate(generator, d_raw, N);

// Kernel per normalizzare i valori nel range 1-6
kernel_normalize_values<<<blocks, threads>>>(d_raw, d_dice, N);

// Trasferimento dei valori all'host
cudaMallocHost(&h_dice, N * sizeof(unsigned char));
cudaMemcpy(h_dice, d_dice, N * sizeof(unsigned char),
cudaMemcpyDeviceToHost);

// Clean up
curandDestroyGenerator(generator);
```

```
cudaFree(d_raw);
cudaFree(d_dice);
cudaFreeHost(h_dice);
```