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();
  ```

  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

  ```
  __syncthreads();
  ```

  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

```
__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

Il comportamento può essere configurato tramite parametro `--default-stream` durante compilazione o direttiva nel codice.

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 [1].

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 [2].

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);
```

[1] Da notare che la parte di multi-GPU non è più trattata nel corso
[2] Come sopra

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, effettuabile tramite la primitiva `cudaDeviceSynchronize()` 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, disattivando i thread "non interessati".

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

```
__syncthreads();
```

Ovvero una barriera che devono 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, gli altri non la raggiungeranno mai; la correttezza dell'uso sta al programmatore.

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 le divergenze si possono usare primitive di sincronizzazione a livello di warp (`__syncwarp()`) o tecniche per evitare del tutto istruzioni condizionali.

5. Cosa sono e per cosa possono essere utilizzati gli eventi.

Solution: Un evento è un marker all'interno di uno stream associato a un punto del flusso di operazioni. Possono assumere due stati: occorso/non occorso. I due utilizzi principali sono:

- Sincronizzare l'esecuzione tra stream: si può avere sincronizzazione esplicita tra stream tramite gli eventi, la primitiva

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

  permette di far attendere a uno stream l'occorrenza di un evento su un altro stream (non necessariamente sulla stessa GPU)

- Monitorare il progresso del device: dal lato host si possono usare le primitive

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

  per, rispettivamente, bloccare l'host fino all'occorrenza di un evento oppure controllarne l'avvenimento in maniera non bloccante

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 si ha overhead di migrazione quando viene fatto l'accesso ai dati.

3. 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 un buon throughput, 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

4. Memorie statiche e dinamiche.

Solution: Le allocazioni di memoria possono essere statiche o dinamiche. Le allocazioni statiche hanno dimensione nota compile-time e possono risiedere a livello di:

- Global memory: la si può dichiarare tramite `__device__` (o `__constant__` se di sola lettura), ha un accesso lento, scope e lifetime globale

- Shared memory: la si può dichiarare tramite `__shared__` all'interno del kernel, latenza bassa, ha scope a livello di blocco e lifetime del kernel
- Registri: variabili locali ai thread, si tratta della memoria più veloce, ma di dimensione limitata

Le allocazioni dinamiche invece sono usate quando la dimensione dei dati è nota solo in esecuzione. Possono essere:

- Device malloc: tramite la primitiva `cudaMalloc()`, richiede passaggi di memoria espliciti da host a device (e viceversa, `cudaMemcpy()`), disponibile fino all'istruzione `cudaFree()`, risiede nella memoria globale del device
- Pinned memory: non all'interno del device, ma la primitiva `cudaHostAlloc()` permette di allocare memoria non paginabile dal sistema operativo per trasferimenti più veloci; questa può essere mappata nello spazio di indirizzamento della GPU (memoria zero copy)
- Unified Memory: la primitiva `cudaMallocManaged()` permette di avere una zona di memoria con indirizzamento unico per host e device, lasciando la gestione della migrazione dei dati a CUDA; semplice da utilizzare, ma può portare a maggiori latenze se non usata correttamente
- Shared memory: la memoria condivisa può essere allocata dinamicamente tramite una dichiarazione all'interno del kernel di una variabile adimensionale:

  ```
  extern __shared__ float s[];
  ```

  Va poi passato come terzo argomento tra variabili angolari durante il lancio del kernel la dimensione della memoria condivisa:

  ```
  kernel<<<grid, block, shared_bytes>>>();
  ```

5. Significato e uso della memoria unificata.

Solution: La memoria unificata è un modello di gestione della memoria che permette l'utilizzo di un singolo spazio di indirizzamento (puntatore unico) per accedere a dati sia host che device.

Ha lo scopo di semplificare la gestione della memoria, rendendo trasparenti al programmatore i trasferimenti, gestiti da CUDA.

La si può allocare tramite la funzione `cudaMallocManaged()`, è anche presente un

parametro `flag` per specificare se la memoria è condivisa solo con l'host o anche con tutte le altre GPU.

Semplifica lo sviluppo CUDA, ma può portare a un maggiore overhead dovuto alla migrazione (gestita a livello di pagina) rispetto alla gestione della memoria con trasferimenti espliciti.

6. Significato e utilità di pattern di accesso alla memoria globale. Distinzioni tra lettura e scrittura.

Solution: La global memory è una memoria off-chip (DRAM, divisa dal chip dell'SM su cui il thread è in esecuzione), quindi ad alta latenza (400 $\sim$ 800 cicli di clock), con scope e lifetime globale.

I pattern di accesso alla memoria globale sono le strutture riguardanti "come" viene effettuato l'accesso alla memoria da parte dei thread. La global memory ha, infatti, latenza elevata, ma throughput ampio quando gli accessi sono coalescenti.

Gli accessi alla memoria del device possono avvenire in transazioni da 32, 64 o 128 byte. Spesso le applicazioni sono limitate dal throughput effettivo della memoria, quindi per rendere efficienti i trasferimenti si vuole minimizzare il numero di transazioni.

Per migliorare le prestazioni è bene ricordare che le istruzioni vengono eseguite a livello di warp, per un dato indirizzo si esegue una operazione di loading/storing e i 32 thread presentano una singola richiesta di accesso, da servire in una o più transazioni.

Gli accessi possono essere:

- Allineati: quando il primo indirizzo della transazione è un multiplo della granularità della memoria usata per servire la transazione (32 byte per L1, 128 per L2)
- Coalescenti: quando i 32 thread accedono a un blocco contiguo di memoria

Per sfruttare al meglio le transazioni di memoria bisogna rispettare allineamento e coalescenza. Al contrario, effettuare accessi non coalescenti/strided significa rendere necessarie più transazioni per la stessa quantità di dati (fino a 32 diverse).

Può essere importante strutturare i dati in modo da avere accessi coalescenti (array of structures vs structures of array).

Le operazioni di scrittura non usano la cache L1, le `store` vengono cachate solo in L2, prima di essere inviati alla device memory in 1, 2 o 4 segmenti da 32 byte. Accessi allineati e coalescenti sono ugualmente importanti per lettura e scrittura. Inoltre,

per la sola lettura esistono zone di memoria dedicate (constant e texture memory); in generale la lettura possiede una gerarchia di memorie più complessa.

7. Bank e bank conflict.

Solution: La shared memory è una memoria condivisa a livello di blocco, locata all'interno dello SM. Questa è organizzata in "banks" paralleli che permettono l'accesso simultaneo ai dati.

La shared memory è quindi suddivisa in blocchi identici, solitamente da 4 byte/una word, e ciascun blocco può servire un solo accesso per ciclo di clock.

Se più thread accedono in parallelo a indirizzi mappati a bank diversi, questi accessi possono avvenire in parallelo, altrimenti, se due o più thread vogliono accedere allo stesso bank si ha un bank conflict e gli accessi vengono serializzati. Si vogliono quindi evitare i conflitti per utilizzare la massima banda disponibile.

8. Constant memory e il suo utilizzo.

Solution: Si tratta di una memoria che risiede nella device memory, con una cache dedicata per ogni SM. La si può definire tramite l'attributo `__constant__`, permette di ospitare dati di sola lettura, ideale per accessi uniformi. Ha scope globale, va dichiarata al di fuori dei kernel e staticamente.

Ideale per quando tutti i thread all'interno di un warp devono leggere dallo stesso indirizzo di memoria, raggiungendo efficienza simile a quella dei registri, altrimenti le letture vengono serializzate, degradando significativamente le performance.

Può essere inizializzata dall'host usando

```
cudaError_t cudaMemcpyToSymbol(const void* symbol,
    const void* src, size_t count);
```

Si può anche leggere lato host tramite

```
cudaError_t cudaMemcpyFromSymbol(const void* dst,
    const void* symbol, size_t count);
```

9. Come viene mappata logicamente e fisicamente la shared Memory.

Solution: La shared memory è una memoria programmabile a bassa latenza, con scope di blocco (visibile da tutti thread all'interno di uno stesso blocco) e lifetime pari a quello del kernel. La si può dichiarare tramite il qualificatore `__shared__`, sia staticamente che dinamicamente (aggiungendo `extern` e specificando la dimensione in fase di chiamata del kernel). Serve come comunicazione a bassa latenza tra thread dello stesso blocco. La dimensione della shared memory è limitata e il numero di blocchi concorrenti è determinato anche dalla smem disponibile.

Fisicamente, si tratta di una memoria on-chip, quindi posizionato sullo stesso chip dello SM, organizzata linearmente in moduli chiamati bank (tipicamente 32). Gli accessi alla shared memory (sia `load` che `store`) vengono emessi per warp e la banda massima si ha quando tutti gli warp accedono a bank diversi (word consecutive). Se due thread all'interno dello stesso warp tentano di accedere allo stesso bank si ha un bank conflict, che richiede la serializzazione degli accessi e causa un degrado delle prestazioni.

10. Vantaggi della Unified Memory ed esempio di utilizzo.

Solution: La memoria unificata (unified memory), introdotta da CUDA 6.0, semplifica la gestione della memoria creando uno spazio di indirizzamento unico tra CPU e GPU. Tramite un unico puntatore si può accedere sia alla memoria host che device.

Il trasferimento di dati avviene in maniera trasparente al programmatore, gestito dal sistema CUDA. Tra i vantaggi possiamo notare:

- semplicità di programmazione, si tratta di un modello più semplice, non richiede gestione manuale della memoria e porta a codice più comprensibile
- coerenza automatica della memoria, CUDA si occupa di mantenere automaticamente la coerenza tra host e device, a volte a scapito delle performance
- over-subscription della memoria, si può allocare più memoria di quanta sia fisicamente disponibile sulla GPU, le pagine verranno spostate in automatico quando necessario, permettendo di usare data set più grandi

Esempio di utilizzo:

```
// Dichiarazione e popolamento dei dati
float *A, *B;
cudaMallocManaged(&A, size);
cudaMallocManaged(&B, size);
for (int i = 0; i < N; ++i) {
    A[i] = float(i);
```

```
}

// Esecuzione del kernel
kernel<<<grid, block>>>(A, B);
cudaDeviceSynchronize();

// Uso dei risultati da parte della CPU
printf("%f", B[0]); // Esempio banale

// Rilascio delle risorse
cudaFree(A);
cudaFree(B);
```

11. Come si usa la SMEM?

Solution: La shared memory è una memoria a bassa latenza con lifetime del kernel e scope a livello di blocco. La si può dichiarare tramite il qualificatore `__shared__`:

```
__shared__ float s[size];
```

Se la dimensione della memoria condivisa non è nota a compile time ma deve essere dinamica, si usa la keyword `extern` e la dimensione va indicata come terzo parametro tra parentesi angolari quando viene chiamato il kernel. Dichiarazione:

```
extern __shared__ float s[];
```

Dimensione:

```
kernel<<<grid, block, sharedMemorySize>>>();
```

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 di cui è composta la GPU. Quali sono le unità hardware?

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
- Barrier & Synchronization Unit: implementano primitive di sincronizzazione a livello di blocco
- Interfacce verso L2 e rete di connessione tra SM

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. I thread all'interno di un warp 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.

6. Ciclo di vita dei thread e organizzazione gerarchica. Come conviene organizzarli per lavorare su matrici?

Solution: I thread sono l'unità di esecuzione fondamentale in CUDA, il codice viene scritto in CUDA C per l'esecuzione sequenziale la quale viene estesa a migliaia di thread. Ogni thread, durante la sua vita, può passare per più stati di esecuzione, legati anche al modello di esecuzione SIMT e allo scheduling:

- In esecuzione/selezionato: il thread è pronto per essere eseguito e il warp scheduler manda in esecuzione il relativo warp; sta eseguendo istruzioni
- In attesa/blocked: il thread non è parte degli warp "selezionati" dal warp scheduler e non può esserlo in quanto "bloccato" da qualcosa (come l'attesa di un trasferimento di memoria)
- Candidato: il thread è pronto, può essere il prossimo ad essere mandato in esecuzione, ma il warp scheduler non lo ha ancora selezionato

In realtà lo stato riguarda il warp di cui fa parte il thread, non il thread singolo.

I thread sono suddivisi (logicamente) su due livelli:

- Block: collezione ordinata di thread, che possono cooperare tra loro, indipendenti l'uno con l'altro; ogni block ha dimensione massima di 1024 thread
- Grid: insieme di tutti i blocchi che eseguono lo stesso kernel

A livello fisico:

- i thread vengono mappati sui CUDA Core

- i blocchi vengono mappati sugli SM, ogni SM può eseguire contemporaneamente uno o più blocchi (in base alle caratteristiche degli stessi)
- la grid viene mappata al device, il quale può eseguire una o più grid concorrentemente, appartenenti a kernel diversi
- i thread fisicamente sono raggruppati in gruppi di 32, detti warp; questi dovrebbero, tra loro, avere modello di esecuzione SIMD, pena degrado delle prestazioni

I thread possono essere organizzati su blocchi e griglie a 1, 2 o 3 dimensioni. Per lavorare su matrici lo schema più intuitivo è quello di usare blocchi e grid bidimensionali in modo da coprire tutto l'input (grid) dividendolo in sotto-matrici (blocchi).

7. Cosa sono i warp in SIMT e qual è la loro utilità.

Solution: Nella architettura SIMT di CUDA, un warp è il raggruppamento minimo di thread all'interno di uno SM che la GPU esegue in lock-step. Un warp è composto da 32 thread e ognuno di questi esegue la stessa istruzione nello stesso ciclo di clock.

All'interno del modello SIMT l'esecuzione parte parallela, ma ogni thread può intraprendere un percorso di esecuzione diverso, raggruppando i thread in gruppi da 32 si può ridurre l'overhead dovuto a controllo e fetching delle istruzioni. Questo può però portare a problemi di divergenza: thread all'interno di un singolo warp devono eseguire la stessa istruzione, se sono presenti path di esecuzione diversi questi vanno eseguiti serialmente, degradando le prestazioni.

Inoltre, uno SM decide quali warp mandare in esecuzione tra quelli candidati/pronti, rendendo inattivi gli warp che richiedono attesa, mascherando così la latenza dovuta a operazioni come trasferimenti di memoria. Si vuole massimizzare l'occupancy, ovvero il rapporto tra thread attivi e thread massimi sostenibili, misurato in warp.

8. Cos'è la scalabilità? Come vengono misurati i tempi e lo speedup.

Solution: La scalabilità è una proprietà desiderabile per qualsiasi applicazione parallela, fa riferimento alla capacità di eseguire parti di un calcolo in modo concorrente per risolvere velocemente problemi anche di grandi dimensioni; di conseguenza, è la capacità di un'applicazione di trarre vantaggio dall'aumento delle risorse, in maniera efficiente.

Lo speed-up è definito come il rapporto tra il tempo di esecuzione sequenziale $T(1)$ e il tempo di esecuzione su p dispositivi $T(p)$:

$$S(p) = \frac{T(1)}{T(p)}$$

L'efficienza è lo speeup normalizzato per il numero di risorse.

I tempi, in ambito CUDA, richiedono di tenere conto dell'esecuzione parallela o asincrona:

- Per tenere traccia solamente dell'esecuzione dei kernel si possono usare gli eventi di timing forniti da CUDA. Tra due eventi, i quali si possono registrare tramite `cudaEventRecord()`, si può fare `cudaEventElapsedTime()` per trovare il tempo tra loro (in ms). Questo non tiene conto dell'overhead dovuto ad allocazione e trasferimento della memoria
- I tempi possono essere misurati dal "punto di vista" dell'host tramite normali primitive di sistema/C, in questo modo si può considerare anche l'overhead dovuto ad allocazione e trasferimento della memoria; da ricordare che il lancio dei kernel è asincrono, quindi prima di prendere la misurazione del tempo finale è necessario sincronizzare il device tramite `cudaDeviceSynchronize()`
- Gli strumenti di profiling forniti da Nvidia permettono un'analisi dettagliata delle prestazioni, inclusi i tempi di esecuzione del kernel

9. Cosa sono le operazioni atomiche e come vengono usate per il calcolo dell'istogramma di immagini RGB.

Solution: Le operazioni atomiche sono operazioni (solamente matematiche) la quale esecuzione non può essere interrotta da altri thread. Sono funzioni, come ad esempio `atomicAdd()` che vengono tradotte in operazioni singole. Servono a evitare race conditions per l'aggiornamento di dati da parte di thread multipli.

Il calcolo dell'istogramma di immagini RGB è una analisi delle frequenze per ogni tono presente in un'immagine. Un'idea per calcolarlo in maniera parallela è:

- Allocare una struttura dati per memorizzare i valori delle frequenze (basta un array lungo `3 * 256`)
- Assegnare a ogni thread un pixel
- Ogni thread aumenta di 1 il valore della cella corrispondente al tono di colore del pixel assegnatogli

Quest'ultimo incremento deve essere atomico, altrimenti si potrebbe incorrere in race conditions e di conseguenza valori non consistenti al termine dell'esecuzione.

10. Cosa indica la compute capability di una GPU.

Solution: Con il termine "compute capability" si intende la versione dell'architettura CUDA supportata da una GPU Nvidia, espressa sotto forma di numero.

Permette di stabilire le feature e funzionalità hardware disponibili. Viene usato in fase di compilazione per stabilire l'architettura per cui compilare.

11. Come vengono schedulati i blocchi sugli SM?

Solution: Quando viene invocato un kernel, si ha una coda di blocks che aspettano di andare in esecuzione, il block scheduler assegna ogni blocco al primo SM che abbia risorse libere a sufficienza. Vengono assegnati dinamicamente agli SM in base alla disponibilità. Un blocco di thread viene assegnato a un solo SM e vi rimane fino al termine della sua esecuzione; molteplici blocchi possono risiedere su un singolo SM.

Ogni SM ha un limite di risorse (in termini di registri, shared memory, numero di thread, …) e possiede uno o più warp scheduler che si occupano di decidere, a ogni ciclo di clock, quali warp, tra quelli attivi, mandare in esecuzione; un warp attivo può essere in uno di tre stati:

- Candidato: pronto per essere mandato in esecuzione
- Selezionato: in esecuzione
- Bloccato: in attesa di qualcosa, ad esempio un trasferimento di memoria

Il warp scheduler cambia dinamicamente il warp da mandare in esecuzione in modo da mantenere alta l'occupancy (rapporto tra warp attivi e warp teoricamente sostenibili) e nascondere la latenza dovuta ad alcuni tipi di operazioni.

12. Che livello di concorrenza permettono di ottenere i CUDA streams.

Solution: I CUDA Streams sono sequenze di operazioni eseguite sulla GPU in maniera asincrona rispetto all'host e (generalmente) indipendenti l'uno dall'altro (stream non-NULL non sono bloccanti a vicenda).

L'esecuzione di un kernel può cominciare alla fine dei trasferimenti di memoria necessari, ma con grandi quantità di dati sarebbe preferibile caricare una porzione dei dati e cominciare l'esecuzione, lasciando al DMA il caricamento dei dati restanti, parallelamente all'esecuzione del kernel. Gli stream permettono di fare questo, abilitano la concorrenza tra trasferimenti di memoria e kernel.

Permettono la sovrapposizione di trasferimento dati ed esecuzione del kernel.

13. Come può la divergenza causare deadlock?[3]

Solution: La divergenza accade quando thread all'interno di uno stesso warp seguono flussi di esecuzione separati. Per risolvere il problema esistono barriere di sincronizzazione a livello di warp come `__syncwarp(mask)`, queste lasciano i thread che la raggiungono in attesa fino a quando tutti i thread di uno stesso warp non arrivano a quel punto nell'esecuzione.

~~Le barriere di sincronizzazione possono causare problemi quando condizionate: se solo una porzione dei thread all'interno di un warp raggiunge il `__syncwarp()` dietro un `if` in cui gli altri non ci arriveranno mai, lasciando gli altri in attesa infinita: deadlock~~.

[3]non sono sicuro, penso non causi deadlock perché tutto lock-step, ma non ho capito e sono stanco

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);
```

4. Dire a cosa serve la libreria cuBLAS.

Solution: La libreria cuBLAS, sviluppata da Nvidia, è un'implementazione accelerata su GPU dei Basic Linear Algebra Subproblem, quindi fornisce API per la risoluzione di problemi di algebra lineare, divisa in 3 livelli:

1. Operazioni vettore-vettore
2. Operazioni vettore-matrice
3. Operazioni matrice-matrice

5 Python

1. Commentare un kernel CUDA python semplice.

Solution: Un semplice kernel potrebbe essere:

```
@cuda.jit
def increment_kernel(data):
    # Calcola la posizione assoluta
    #   del thread nella griglia
    idx = cuda.grid(1) # L'1 indica l'indice su 1 dimensione
    # Controlla il bound dell'array
    if idx < data.size:
        data[idx] += 1 # Incrementa il valore
```

Il lancio del kernel potrebbe essere:

```
from numba import cuda
import numpy as np

# Move data to the device
device_array = cuda.to_device(host_array)
# Call the kernel
increment_kernel[grid, block](device_array)
# Move the results back
result_array = device_array.copy_to_host()
```

presupponendo che i parametri necessari siano già impostati.

2. Dare il kernel della somma matriciale in Numba.

Solution: Per effettuare la somma tra matrici:

```
@cuda.jit
def matsum (A, B):
    # Indice 2D della griglia
    row, col = cuda.grid(2)
    # Controllo dei bound
    if row < A.shape[0] and col < A.shape[1]:
        A[row, col] = A[row, col] + B[row, col]
```

I parametri sono solamente le due matrici, il risultato viene posto nella prima.

6 Pattern

1. Pseudocodice di Jones-Plassman per la colorazione parallela

Solution: L'algoritmo di Jones-Plassman è un algoritmo approssimato per ambienti distribuiti per risolvere il problema di graph coloring, con una qualità delle soluzioni simile a quella degli algoritmi sequenziali.

Dato un grafo $G = (V, E)$ e un insieme di colori $\mathcal{C}$:

- Ogni vertice $v \in V$ riceve una priorità casuale
- Ogni vertice può decidere il proprio colore quando ha priorità maggiore di tutti i vicini non ancora colorati
- Viene iterato lo step di decisione, fino a completamento

All'interno di ogni round, il passo di decisione e selezione può essere svolto in parallelo. Per un grafo sparso e con distribuzioni casuali dei pesi converge con *alta probabilità* per $O(\log |V|)$.

Pseudocodice (inventato al momento):

```
1 S ← ∅;
2 R ← V;
3 while R ≠ ∅ do
4 |   ∀v ∈ R, π(v) ← rand() ;  // parallelo
5 |   if π(v) > π(n), ∀n ∈ N(v) non colorato
  |    then
6 |   |   C ← ⋃_{n∈N(v)} c(n);
7 |   |   c(v) ← colore minimo ∉ C;
8 |   |   S ← S ∪ {v};
9 |   |   R ← R − {v};
```

Dove $\pi : V \to \mathbb{N}$ restituisce la priorità del nodo e $c : V \to \mathcal{C}$ restituisce il colore del nodo.

2. Illustrare il concetto di parallel reduction.

Solution: La parallel reduction è un pattern di programmazione parallela comune che consiste nell'eseguire un'operazione commutativa e associativa su un vettore $a =$

$\{a_0, \ldots, a_{n-1}\}$ tale da ottenere il vettore risultato $b = \{a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus \cdots \oplus a_{n-1})\}$. In sintesi:

$$b[k] = \sum_{i=0}^{k} a[i] \quad \forall k \in 0, \ldots, n-1$$

L'approccio sequenziale è semplice, in parallelo la strategia di base prevede:

- suddividere il vettore di input
- assegnare thread per calcolare somme parziali
- combinare i risultati parziali

I due approcci "naive" (presupponendo che la lunghezza n del vettore in input sia potenza di 2) possibili sono:

- coppie contigue: a ogni iterazione i un thread somma elementi a distanza 2^i (partendo da $i = 0$)
- coppie equispaziate: a ogni iterazione i un thread somma elementi a distanza $n/2^{i+1}$ (partendo da $i = 0$)

Un problema di performance che può sorgere con questi approcci è la divergenza crescente a livello di warp se non si determinano in maniera oculata quali thread sono attivi per fare quali somme, l'approccio ingenuo introduce divergenza crescente disattivando metà dei thread attivi a ogni step.

3. Implementazione di Horn per prefix sum.

Solution: Avendo a disposizione molti processori, si vuole ridurre il tempo delle somme prefisse a $O(\log n)$, anche a costo di qualche operazione in più.

Viene usato un approccio iterativo in $d = \lceil \log_2 n \rceil$ passi, a ogni passo, da 1 a d:

- Calcolo l'offset $\Delta = 2^i$
- Per ogni elemento k dell'array (in parallelo)
  - se $k \geq \Delta$: $x[k] \leftarrow x[k] + x[k - \Delta]$
  - se $k < \Delta$: $x[k] \leftarrow x[k]$

Questo algoritmo effettua $\Theta(\log n)$ step, ma $\Theta(n \log n)$ operazioni, quindi non è work efficient.

4. Schema per l'implementazione work efficient della prefix sum.

Solution: La prefix sum, dato un operatore binario associativo (in questo caso la somma) e due vettori di dimensione n (a input, b output) vuole fare in modo che:

$$b[k] = \sum_{i=0}^{k} a[i] \quad \forall k \in [0, n-1]$$

La strategia work efficient usa (concettualmente) un albero bilanciato sui dati in input, in cui le foglie sono i valori dell'array di input.

Due fasi:

1. reduce o up-sweep: per costruire le somme parziali fino alla radice, ogni nodo a livelli intermedi sarà la somma dei nodi figlio
2. down-sweep: per distribuire i prefissi corretti alle foglie; si scorrono i livelli dalla radice verso le foglie, si setta la radice a zero e
   - ogni figlio sinistro sarà pari al padre
   - ogni figlio destro sarà la somma del padre e del valore (prima che venga aggiornato) del figlio sinistro

In totale sono:

- $2 \log n$ passi, ovvero $\log n$ per fase
- $2n - 1$ operazioni, una per nodo interno dell'albero per fase

5. Spiega il bitonic MergeSort

Solution: Una sequenza bitonica è una sequenza $s = \{a_0, \ldots, a_{n-1}\}$ su cui vale la proprietà

$$\exists i \in [0, n-1] \text{ t.c. } a_0 \leq \cdots \leq a_i \geq \ldots a_{n-1}$$

oppure esiste una permutazione ciclica degli indici per il quale la proprietà vale.

Se la sequenza $s = \{a_0, \ldots, a_{n-1}\}$ è bitonica, allora

$$s_1 = \{\min(a_0, a_{n/2}), \min(a_1, a_{n/2+1}), \ldots, \min(a_{n/2-1}, a_{n-1})\}$$
$$s_2 = \{\max(a_0, a_{n/2}), \max(a_1, a_{n/2+1}), \ldots, \max(a_{n/2-1}, a_{n-1})\}$$

sono anch'esse sequenze bitoniche e tutti gli elementi di s_1 sono minori di tutti gli elementi di s_2.

Si può ripetere questo passaggio fino a sequenze di 2 elementi, ordinabili banalmente ($\log n - 1$ passi).

6. Spiegare la bitonic merge network.

Solution: Una bitonic merging network con una sequenza bitonica lunga n in input produce una sequenza

- crescente con il comparatore $\oplus BM[n]$
- decrescente con il comparatore $\ominus BM[n]$

Per una rete con $n = 2$ si tratta di comparatori che semplicemente "scambiano" i valori se sono nell'ordine sbagliato.

Per una rete con n fili, una $BM[n]$ ha

- $\log_2 n$ colonne
- $n/2$ comparatori a coppie per colonna

Ogni colonna ha comparatori a distanza metà di quella precedente, a partire da distanza $n/2$, fino a 1 (dopo $\log_2 n$ colonne). In questo modo, a partire da una sequenza bitonica in entrata, si ha una sequenza ordinata in uscita.

7. Spiega l'ordinamento bitonico

Solution: A partire da una sequenza qualsiasi, si può creare una sequenza bitonica tramite una rete con $\log_2 n - 1$ colonne in cui ogni i-esima colonna usa $\oplus BM[2^i]$ e $\ominus BM[2^i]$ alternati su tutto l'input per creare sequenze bitoniche lunghe 2^{i+1} (a partire da $i = 0$, un valore singolo soddisfa banalmente le proprietà di una sequenza bitonica).

Alla fine si ha una sequenza bitonica ordinabile con una bitonic merging network $\oplus BM[n]$ (o $\ominus BM[n]$).

Per una sequenza lunga n, la complessità è $\Theta(\log^2 n)$.

7 Codice

1. 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 "alone" 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. Tiling della zona da caricare in smem.

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.

Pseudocodice:

```
1 nblocks = numero di blocchi per il tiling;
2 for i, j ∈ numblocks do
3 |   // Calcola indici
4 |   // Carica nella smem
5 __syncthreads();
6 sum = 0;
7 for i, j ∈ MASK_SIZE do
8 |   sum ← sum + smemValue * maskValue;
9 // Scrivere l'output
```

Codice:

```
__global__ void conv2D(Matrix A, Matrix B, Matrix M) {
    // Indici per colonna e riga della matrice A
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
```

```
// Allocazione della smem
__shared__ float smem[TILE_SIZE][TILE_SIZE];

// Caricare i dati nella smem
for (int row = 0; row <= TILE_SIZE / blockDim.y; row++) {
    for (int col = 0; col <= TILE_SIZE / blockDim.x; col++) {
        // Indici per i dati originali
        int d_row = y + blockDim.y * row - RADIUS;
        int d_col = x + blockDim.x * col - RADIUS;
        // Indici per la smem
        int s_row = threadIdx.y + blockDim.y * row;
        int s_col = threadIdx.x + blockDim.x * col;

        // Controlla che il range sia valido, per smem e dati
        if (s_row < TILE_SIZE && s_col < TILE_SIZE) {
            if (d_row >= 0 && d_row < A.height
                        && d_col >= 0 && d_col < A.width) {
                smem[s_row][s_col] =
                        A.elements[d_row * A.width + d_col];
            } else {
                smem[s_row][s_col] = 0.0f;
            }
        }
    }
}
__syncthreads();

// Convoluzione
float sum = 0.0f;
for (int i = 0; i < MASK_SIZE; i++) {
    for (int j = 0; j < MASK_SIZE; j++) {
        int r = threadIdx.y + i;
        int c = threadIdx.x + j;
        if (r >= 0 && r < TILE_SIZE && c >= 0 && c < TILE_SIZE) {
            sum += smem[r][c] * M.elements[i * MASK_SIZE + j];
        }
    }
}

// Scrive l'output
if (y < A.height && x < A.width) {
    B.elements[y * B.width + x] = sum;
```

```
    }
}
```

2. Pseudocodice del trasferimento in memoria shared per il prodotto matriciale.

Solution: L'idea dietro l'uso della smem è: vogliamo suddividere in tile le zone di memoria della matrici di cui fare il prodotto (divise in `nblocks` tile della dimensione del block), in modo da poterle caricare con una suddivisione del lavoro più equa possibile. Per il numero di tile `nblocks` quindi si ripete il processo di

- caricare i valori delle due matrici in smem
- calcolare la somma parziale data dai valori parziali caricati, per ogni cella della matrice prodotto

Una volta terminato, scrivere i valori in memoria globale.

Pseudocodice:

```
__shared__ As[WIDTH][WIDTH];
__shared__ Bs[WIDTH][WIDTH];
nblocks = numero di block contenuti nelle sotto-matrici da
  caricare;
for i ∈ nblocks do
    As[tidy][tidx] = A[tidx_abs + i *
     WIDTH][tidy_abs];
    Bs[tidy][tidx] = B[tidx_abs][tidy_abs + i *
     WIDTH];
    __syncthreads();
    for j ∈ WIDTH do
        partial_sum += As[tidy][j] * Bs[j][tidx];
    __syncthreads();
// Scrivere il risultato in memoria globale
```

Dove:

- `WIDTH` è la dimensione del blocco di dati da caricare
- `tidx` e `tidy` sono le coordinate del thread all'interno del blocco
- `tidx_abs` e `tidy_abs` sono le coordinate del thread rispetto alla grid
- `nblocks` rappresenta il ceil della larghezza (o altezza) della matrice fratto la larghezza (o altezza) del block

3. Codice per un kernel che svolge il prodotto di matrici triangolari superiori.

Solution: Senza ottimizzazioni particolari, un kernel potrebbe essere:

```
// A e B matrici di input, C output, N dimensione
__global__ void upperTriangularMatMul(float *A, float *B,
                float *C, int N) {
    // Indici assoluti della cella
    int row = threadIdx.y + blockIdx.y * blockDim.y;
    int col = threadIdx.x + blockIdx.x * blockDim.x;
    // Bound check
    if (row < N && col < N) {
        float sum = 0.0f;
        if (row <= col) {
            // Somma solo per k = row..col, dato che upper-triangular
            for (int k = row; k <= col; ++k) {
                sum += A[row * N + k] * B[k * N + col];
            }
        }
        // Scrivere in memoria il risultato
        C[row * N + col] = sum;
    }
}
```

4. Mostrare un uso pratico degli stream CUDA.

Solution: Gli stream vengono utilizzati per sovrapporre trasferimenti di memoria con computazioni da parte della GPU. Uno schema di utilizzo può essere:

```
// Si suppongono già fatte le allocazioni di memoria

//Creazione degli stream
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

// Trasferimenti e lancio kernel separati
cudaMemcpyAsync(dst, src, len, cudaMemcpyHostToDevice, stream1);
cudaMemcpyAsync(dst, src, len, cudaMemcpyHostToDevice, stream2);
kernel<<<grid, block, smem, stream1>>>();
kernel<<<grid, block, smem, stream2>>>();
```

```
cudaMemcpyAsync(dst, src, len, cudaMemcpyDeviceToHost, stream1);
cudaMemcpyAsync(dst, src, len, cudaMemcpyDeviceToHost, stream1);

// Attendere che gli stream termina
cudaStreamSynchronize(stream1);
cudaStreamSynchronize(stream2);

// Rilascio delle risorse
cudaStreamDestroy(stream1);
cudaStreamDestroy(stream2);
```

5. Uso degli stream per il prodotto di matrici diagonali a blocchi MQDB.

Solution: Gli stream sono una sequenza di operazioni CUDA incapsulate in una coda FIFO, eseguite dalla GPU in maniera asincrona rispetto all'host, sempre rispettando l'ordine di esecuzione fornito. Una MQDB è una matrice che ha valori non nulli solo in blocchi di dimensioni $d_1, \ldots, d_n$ attorno alla diagonale.

L'uso degli stream permette di dividere il carico di lavoro e i trasferimenti di memoria, permettendo un parallelismo tra trasferimenti ed esecuzione dei kernel (DMA permettendo). Sapendo che il prodotto di due matrici MQDB è anch'esso una matrice con le stesse proprietà, lo schema ovvio per l'utilizzo degli stream è quello di utilizzare stream diversi per ogni blocco di dati, permettendo il parallelismo tra trasferimenti e kernel.

Codice (forse funzionante):

```
__global__ void streamMQDB(int *A, int *B, int *R) {
    int tid = threadIdx.x + threadIdx.y * blockDim.x;
    __shared__ int As[BS][BS];
    __shared__ int Bs[BS][BS];

    As[threadIdx.y][threadIdx.x] = A[tid];
    Bs[threadIdx.y][threadIdx.x] = B[tid];
    __syncthreads();

    float sum = 0.0f;
    for (int k = 0; k < BS; ++k) {
        sum += As[k][threadIdx.y] * Bs[threadIdx.x][k];
    }
    R[tid] = sum;
}
```

6. Esempio e vantaggi di loop unrolling. Esempio nella reduction.

Solution: Il loop unrolling è una tecnica usata per ottimizzare i cicli: questi vengono "srotolati" in modo da ridurre l'overhead dovuto alle operazioni di controllo, ridurre il branching e i salti condizionali, aumentando così il livello di parallelismo. Si copia il corpo del loop un numero n di volte, chiamato unroll factor. Lo si può fare manualmente, in alternativa, CUDA fornisce direttive di compilazione.

Esempio semplice di loop unrolling:

```
for (int i = 0; i < 4; i++){
    A[base] += B[base + i];
}
```

Può diventare:

```
A[base] += B[base + 0];
A[base] += B[base + 1];
A[base] += B[base + 2];
A[base] += B[base + 3];
```

In alternativa, usando le direttive:

```
#pragma unroll 4
for (int i = 0; i < 4; i++){
    A[base] += B[base + i];
}
```

Lo si dovrebbe usare quando il corpo del ciclo è semplice e il numero di iterazioni è multiplo dell'unroll factor. Un unrolling troppo spinto potrebbe causare bloating del codice e riduzione delle performance dovuta a mancanza di spazio in cache/instruction buffer.

Esempio nella reduction: loop all'interno del kernel, prima dell'unroll

```
for (int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
    if (threadIdx.x < stride) {
        smem[threadIdx.x] += smem[threadIdx.x + stride];
    }
    __syncthreads();
}
```

Mentre la versione con unroll è

```
if (blockDim.x >= 1024 && threadIdx.x < 512) {
    smem[threadIdx.x] += smem[threadIdx.x + 512];
    __syncthreads();
}
if (blockDim.x >= 512 && threadIdx.x < 256) {
    smem[threadIdx.x] += smem[threadIdx.x + 256];
    __syncthreads();
}
if (blockDim.x >= 256 && threadIdx.x < 128) {
    smem[threadIdx.x] += smem[threadIdx.x + 128];
    __syncthreads();
}
if (blockDim.x >= 128 && threadIdx.x < 64) {
    smem[threadIdx.x] += smem[threadIdx.x + 64];
    __syncthreads();
}
// Within a single warp
if (threadIdx.x < 32) {
    volatile float *vsmem = smem;
    vsmem[threadIdx.x] += vsmem[threadIdx.x + 32];
    vsmem[threadIdx.x] += vsmem[threadIdx.x + 16];
    vsmem[threadIdx.x] += vsmem[threadIdx.x + 8];
    vsmem[threadIdx.x] += vsmem[threadIdx.x + 4];
    vsmem[threadIdx.x] += vsmem[threadIdx.x + 2];
    vsmem[threadIdx.x] += vsmem[threadIdx.x + 1];
}
```

7. Esempio di utilizzo della constant memory.

Solution: La constant memory è spesso utilizzata per coefficienti che devono essere letti da tutti i thread contemporaneamente. Esempio:

```
__constant__ float pi = 3.12;

// Array di cerchi, dato il raggio calcolare l'area
__global__ void circlesArea (float *area, float *radius, int N) {
    int tid = threadIdx.x + blockDim.x * blockIdx.x;
    // Bound check
    if (tid < N)
        area[tid] = radius[tid] * pi * pi;
}
```