GPU Computing

Massimo Perego

Indice

1 Introduzione all'High Performance Computing HPC

L'uso delle GPU permette di incrementare significativamente le performance, per avere speed-up anche nell'ordine delle migliaia, per problemi altamente parallelizzabili. Si parlerà di paradigma **GP-GPU** (**General Purpose - GPU**).

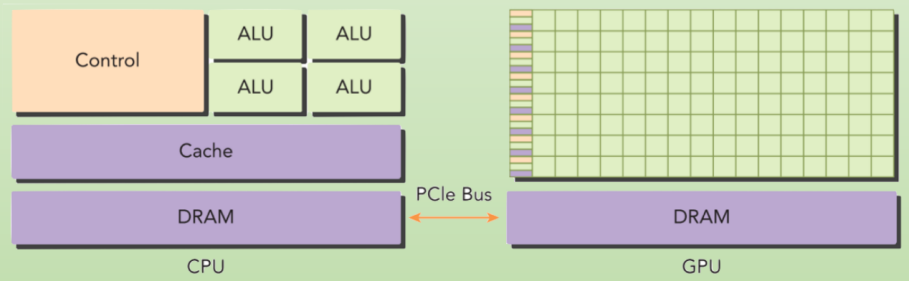
Esistono molti sistemi che si basano su operazioni semplici ma ripetute numerose volte. Esempio: il prodotto matriciale è il prodotto vettore-vettore ripetuto. Questi sistemi sono facilmente parallelizzabili (se non ci sono interdipendenze tra i risultati).

Parallelismo: Vogliamo accelerare il tempo, il parallelismo è la capacità di eseguire parti di un calcolo in modo concorrente. Esistono problemi che sarebbero impensabili senza l'accelerazione permessa dal parallelismo. Permette di risolvere problemi più grandi nello stesso tempo, o problemi di dimensione fissa in tempo più breve.

Il parallelismo sarà gestito a livello di **thread**: unità di esecuzione costituita da una sequenza di istruzioni e gestita dal sistema operativo o da un sistema di runtime.

Paradigma GP-GPU: Fa riferimento all'uso di GPU (Graphics Processing Unit) per eseguire computazioni di carattere generale, di qualsiasi tipo. Implica l'uso di CPU e GPU in maniera congiunta, la computazione parte comunque dalla CPU (chiamata **host**) la quale effettuerà richieste alla GPU (chiamata **device**), quest'ultima diventa un coprocessore. Viene separata la parte sequenziale dell'applicazione e di controllo, che va sulla CPU, dalla parte a maggior intensità computazionale, lasciata alla GPU.

La GPU non è una piattaforma stand-alone, ma è un coprocessore che opera congiuntamente alla CPU, comunicando tramite bus PCI-Express. Sono necessari trasferimenti e la CPU orchestra la "sincronizzazione".



L'uso, dal punto di vista dell'utente, di un sistema con GPU è **trasparente**, si ha un risultato più veloce ma dall'esterno non cambia l'esperienza.

Le funzioni vanno riscritte in modo da poterle esporre al parallelismo sulla GPU. I "**kernel**" sono le funzioni demandate alla GPU. Le applicazioni ibride avranno parti di codice host, eseguito sulla CPU, e parti di codice device, eseguito sulla GPU.

La CPU si occupa della gestione dell'ambiente, dei dati per il device stesso ed è ottimizzata per tutte le sequenze di operazioni con un flusso di controllo impredicibile, mentre la GPU è ideale per flussi di controllo semplici.

Un problema da considerare è l'uso di energia: si vuole massimizzare la potenza di calcolo minimizzando l'energia consumata.

La differenza di esecuzione è

- CPU: pochi core ottimizzati per l'elaborazione sequenziale
- GPU: architettura massicciamente parallela che consiste di migliaia di core che cooperano in modo efficiente per trattare molteplici task in maniera concorrente

Il calcolo parallelo può essere realizzato in vari modi, tra cui:

- parallelismo nei **dati**: suddivisone dei dati in parti uguali per essere elaborati simultaneamente su più processori
- parallelismo sui **task**: il lavoro viene suddiviso in attività indipendenti ed ogni task viene eseguito dal suo processore. Nel processo di parallelizzazione bisogna tenere in considerazione le dipendenze tra i task
- parallelismo di **istruzioni**: un programma viene diviso in istruzioni ed ognuna di queste parti indipendenti viene eseguita simultaneamente su più processori

L'ambito di utilizzo del parallelismo dato dalle GPU è con **dimensioni dei dati abbastanza ampie** che allo stesso tempo permettono buon parallelismo.

Tassonomia di Flynn: I modelli di computazione fondamentali sono:

- **SISD Single Instruction Single Data:** una unità che esegue una operazione (sequenziale); questo è il modello di Von Neumann
- **SIMD Single Instruction Multiple Data:** una singola istruzione per molteplici unità di calcolo, applicata su molti dati
- **MISD Multiple Instruction Single Data:** il parallelismo è solo a livello di istruzioni, molte unità sugli stessi dati; non ha implementazioni realistiche
- **MIMD Multiple Instruction Multiple Data:** molteplici unità che possono accedere a molteplici dati, ognuna con istruzioni proprie

SIMT Model: Modello Single Instruction Multiple Thread, introdotto da CUDA. Ogni thread ha la possibilità di "scegliere una strada" in base al dato. Il flusso di controllo parallelo parte assieme ma può portare a branch differenti, in base ai dati. Estende il concetto di SIMD permettendo flussi individuali per ogni thread, con il costo relativo a gestire la decisione locale sui thread (program counter e registri).

Le caratteristiche del modello SIMT che non troviamo all'interno di SIMD sono:

- Ogni thread ha il proprio instruction address counter
- Ogni thread ha il proprio register state e in generale un register set
- Ogni thread può avere un execution path indipendente

1.1 Architettura Nvidia

Streaming Multiprocessor SM: Le GPU sono costituite di array di SM, ognuno dei quali composto da gruppi di 32 CUDA core, chiamati **warp**. Ogni SM in una GPU è progettato per supportare l'esecuzione concorrente di centinaia di thread. In un warp tutti i thread dovrebbero essere SIMD, ovvero eseguire la stessa istruzione allo stesso tempo.

Questo è il modello iniziale, nel tempo si è evoluto aggiungendo elementi come una gerarchia di cache migliorata, altri core dedicati ad applicazioni specifiche (ad esempio, i tensor core per il calcolo matriciale). Ogni CUDA core ha i suoi registri e unità di calcolo (FP e INT).

All'interno di un SM troviamo quindi:

- **Warp scheduler**: da 2 a 4, scelgono, ad ogni ciclo di clock, quale warp "pronto" mandare in esecuzione
- **Dispatch unit**: invia le istruzioni del warp selezionato ai vari execution unit (ALU, SFU, load/store)
- **CUDA Cores**: ALU scalari a 32 bit
- **Special Function Units SFU**: usate per calcoli complessi
- **Tensor Cores**: presenti da Volta in poi, unità specializzate per operazioni matriciali
- **Load/Store Units LSU**: per la gestione delle operazioni di lettura/scrittura dalla shared memory e cache L1
- **Register file**: insieme dei registri per ogni thread; la dimensione globale limita il numero di thread residenti
- **Shared memory/L1 Cache**: Regione di memoria condivisa tra i thread del block, a bassa latenza
- **L2 Cache**: Condivisa fra tutti gli SM; gestisce traffico verso memoria globale

L'organizzazione a livello di parallelismo, dal basso verso l'alto, è:

- **Thread**: ogni SM esegue migliaia di thread in modalità time-multiplex.
- **Warp**: gruppi di 32 thread eseguiti in lock-step (Single Instruction, Multiple Threads).
- **Block**: un blocco (block) di CUDA threads viene schedulato su uno SM fino a quando non termina o subisce uno swap out.
- **Grid**: collezione di blocchi distribuiti sui vari SM.

Ogni **CUDA Core** all'interno degli SM è una unità scalare che esegue operazioni aritmetico-logiche sui dati. Ogni core di questo tipo implementa:

- Pipeline di esecuzione scalare: FP e INT Unit

- Unità di decode e dispatch: riceve l'istruzione dal warp scheduler, ne effettua il decode e la invia al datapath corretto
- Operand Fetch & Write-Back: I dati (operand) vengono letti dai registri, elaborati, poi scritti nuovamente all'interno dei registri o inoltrati alle unità di memoria (per `store`)

In generale, ogni 32 core (warp) devono eseguire la stessa istruzione, ognuno sul proprio dato, con i propri registri.

Sono privi di cache proprie, si appoggiano ai registri condivisi e alle cache.

Compute Capability CC: Rappresenta la versione dell'architettura CUDA supportata da una GPU Nvidia. Definisce le funzionalità hardware disponibili, come il numero di core CUDA, il supporto per le istruzioni avanzate, uso della memoria, risorse, ecc. Viene usato in fase di compilazione per determinare l'architettura per cui compilare.

CUDA Toolkit: Fornisce tutti gli strumenti per la programmazione in CUDA C/C++ (e oltre). Permette compilazione, profilazione e debugging, assieme a librerie ecc.; tutto ciò che serve per sviluppare.

CUDA APIs: Sono presenti due livelli di API per la gestione della GPU e l'organizzazione dei thread:

- **CUDA Runtime API**
- **CUDA Driver API**

Le driver API sono API a basso livello e piuttosto difficili da programmare ma danno un maggior controllo della GPU.

Runtime porta una astrazione maggiore, per un utilizzo più user-friendly ma richiede di compilare con `nvcc` e dipendono dalla versione del driver. Le funzioni cominciano con il nome "`cuda`".

2 Modelli per sistemi paralleli

Un **modello di programmazione parallela** rappresenta un'**astrazione** per un sistema di calcolo parallelo in cui è conveniente esprimere algoritmi concorrenti/paralleli.

Si possono avere diversi livelli di astrazione:

- **Modello macchina**: livello più basso che descrive l'hardware e il sistema operativo (registri, memoria, I/O); il linguaggio assembly è basato su questo livello di astrazione
- **Modello architetturale**: rete di interconnessione di piattaforme parallele, organizzazione della memoria e livelli di sincronizzazione tra processi, modalità di esecuzione delle istruzioni di tipo SIMD o MIMD
- **Modello computazionale**: modello formale di macchina che fornisce metodi analitici per fare predizioni teoriche sulle prestazioni (in base a tempo, uso delle risorse, …). Per esempio il modello RAM descrive il comportamento del modello architetturale di Von Neumann (processore, memoria, operazioni, …) Il modello PRAM estende RAM per architetture parallele

2.1 Modello PRAM

Si tratta del più semplice modello di calcolo parallelo: **memoria condivisa**, n processori; la memoria gestita in questo modo permette di scambiare facilmente valori tra i processori.

Il calcolo procede per passi: ad ogni passo ogni processore può fare una operazione sui dati con possesso esclusivo; leggere o scrivere nella memoria condivisa. Si può selezionare un insieme di processori che eseguono tutti la stessa istruzione (su dati generalmente diversi - **SIMD**). Gli altri processori restano inattivi; i processori attivi sono sincronizzati (eseguono la stessa istruzione simultaneamente).

SIMD: I modelli SIMD sono basati su unità funzionali contenute in processori general purpose. Le ALU SIMD possono effettuare operazioni multiple simultaneamente in un ciclo di clock. Usano registri che effettuano `load` e `store` di molteplici elementi in una sola transazione. La popolarità del modello SIMD deriva dall'uso esplicito di linguaggi di programmazione parallela sfruttando il parallelismo dei dati. Permette di semplificare il controllo in quanto univoco.

Modello di programmazione parallela: Specifica la "vista" del programmatore sul computer parallelo, definendo come si possa codificare un algoritmo; al suo interno

- Comprende la **semantica** del linguaggio di programmazione, librerie, compilatore, tool di profiling
- Dice di che **tipo** sono le **computazioni parallele** (instruction level, procedural level o parallel loops)
- Permette di dare **specifiche implicite** o **esplicite** (da parte dell'utente) per il parallelismo
- Modalità di **comunicazione tra unità di computazione** per lo scambio di informazioni (shared variable)
- Meccanismi di **sincronizzazione** per gestire computazioni e comunicazioni tra diverse unità che operano in parallelo
- Molti forniscono il concetto di **parallel loop** (iterazioni indipendenti), altri di **parallel task** (moduli assegnati a processori distinti eseguiti in parallelo)
- Un **programma parallelo** è eseguito da processori in un ambiente parallelo tale che in ogni processore si ha uno o più flussi di esecuzione, quest'ultimi sono detti processi o thread
- Ha una **organizzazione dello spazio di indirizzamento**: per esempio, distribuito (no variabili shared quindi uso del message passing) o condiviso (uso di variabili shared per lo scambio di informazioni

2.2 Processi UNIX

Con "**processo**" si definisce un programma in esecuzione con diverse risorse allocate (stack, heap, registri, …). Un processo con un solo thread può eseguire una sola attività alla volta, se ci sono più processi in esecuzione è necessario alternarli e di conseguenza avere un context switch (costoso, gestito dal sistema operativo). I processi possono essere creati a runtime.

Thread Unix: Un thread (su CPU) è una estensione del modello di processo (*lightweight process* perché possiedono un contesto più snello rispetto ai processi). Si tratta di un flusso di istruzioni di un programma e viene schedulato come unità indipendente nelle code di esecuzione dei processi della CPU (scheduler).

Condivide lo spazio di indirizzamento con gli altri thread del processo: rappresentato da un thread control block (TCB) che punta al PCB del processo contenitore. Dal punto di vista del programmatore, l'esecuzione del thread è sequenziale, quindi un'istruzione eseguita alla volta, con un puntatore alla prossima istruzione da eseguire e verificando costantemente l'accesso ai dati.

Esistono meccanismi di sincronizzazione tra thread per evitare race condition (accesso a variabili condivise o in generale comportamenti non deterministici).

Ogni processo ha il proprio contesto ed è pensato per eseguire codice sequenzialmente; l'astrazione dei thread vuole consentire di eseguire procedure in maniera concorrente. Ciascuna procedura eseguita in parallelo sarà un thread.

Un thread è quindi un singolo flusso di istruzioni, con le strutture dati necessarie per realizzare il proprio flusso di controllo. Una procedura che lavora in parallelo con le altre.

Stati di un thread: Gli stati di un thread possono essere:

- **Newly generated**: il thread è stato generato e non ha ancora eseguito operazioni
- **Executable**: il thread è pronto per l'esecuzione, ma al momento non è assegnato a nessuna unità di calcolo
- **Running**: il thread è in esecuzione
- **Waiting**: il thread è in attesa di un evento esterno (es. I/O) quindi non può andare in esecuzione fino a che l'evento non si verifica
- **Finished**: il thread ha terminato tutte le operazioni

3 Modello CUDA

3.1 Thread in CUDA

Pensare in parallelo significa avere chiaro quali feature la GPU espone al programmatore

- Conoscere l'architettura della GPU per scalare su migliaia di thread come fosse uno
- Gestione basso livello cache permette di sfruttare principio di località
- Conoscere lo scheduling di blocchi di thread e la gerarchia di thread e di memoria (ridurre latenze)
- Fare impiego diretto della shared memory (riduce latenze come le cache)
- Gestire direttamente le sincronizzazioni (barriere tra thread)

Si scrive codice in CUDA C (estensione di C) per l'esecuzione sequenziale e lo si estende a migliaia di thread (permette di pensare "ancora" in sequenziale ma eseguire codice in parallelo).

L'host ha una serie di processi in esecuzione e controlla l'ambiente, compreso il lancio delle funzioni kernel sul device. Con "kernel" si intende un programma sequenziale eseguito dalla GPU.

Ogni kernel è asincrono: la CPU lancia il kernel e passa oltre, almeno finché non è necessaria sincronizzazione, come ad esempio per i trasferimenti tra memorie.

Il compilatore `nvcc` genera codice eseguibile per host e device (fat-binary).

Esempio di **processing flow**:

- Copiare dati da CPU a GPU, tutto parte dalla CPU
- Caricare il programma GPU, con tutto il setup necessario, svolto da parte della GPU
- Al termine della computazione i risultati vengono copiati da GPU a CPU

La "ricetta" base per cucinare in CUDA:

1. Setup dei dati su host (CPU-accessible memory)

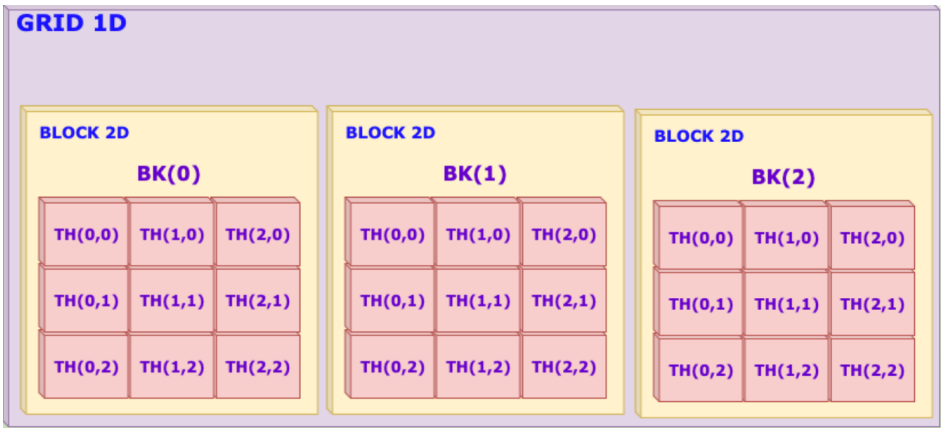
2. Alloca memoria per i dati sulla GPU
3. Copia i dati da host a GPU
4. Alloca memoria per output su host
5. Alloca memoria per output su GPU
6. Lancia il kernel su GPU
7. Copia output da GPU a host
8. Libera le memorie

3.1.1 Organizzazione dei thread

CUDA presenta una **gerarchia astratta di thread** strutturata su **due livelli** che si decompone in

- **grid**: una griglia ordinata di blocchi
- **block**: una collezione ordinata di thread

Grid e block possono avere 1, 2 o 3 dimensioni. Sono possibili 9 combinazioni, ma solitamente si usa la stessa per grid e block. La scelta delle dimensioni è da definire a seconda della struttura dei dati in uso.



Tutti i blocchi devono essere uguali, in struttura e numero di thread. La griglia replica blocchi tutti uguali.

In qualsiasi caso, in **ogni blocco** ci possono essere **al più 1024 thread**; esempi di dimensioni: $(1024, 1, 1)$ o $(32, 16, 2)$. Il totale non può superare 1024.

Mapping logico-fisico dei thread: Ragionando su come sono i thread sono organizzati e su come è composta la GPU, si può pensare al seguente mapping:

Thread	$\rightarrow$	CUDA Core
Thread block	$\rightarrow$	SM
Grid	$\rightarrow$	Device (GPU)

Thread block: Un blocco di thread è un gruppo di thread che possono cooperare tra loro mediante:

- **Block-local synchronization**
- **Block-local shared memory**

La memoria più veloce è condivisa solo dallo stesso blocco, quindi da CUDA 9.0 e CC 3.0+ thread di differenti blocchi possono cooperare come Cooperative Groups.

Tutti i thread in una grid condividono lo stesso spazio di global memory. Una grid rappresenta un processo, ogni processo lanciato dall'host ha una sua grid associata (ogni kernel).

I thread vengono identificati univocamente dalle coordinate:

- `blockId` (indice del blocco nella grid)
- `threadId` (indice di thread nel blocco)

Sono variabili built-in, ognuna delle quali con 3 campi: `x,y,z` (una per ogni possibile dimensione).

Dimensioni di blocchi e thread: le dimensioni di grid e block sono specificate dalle variabili built-in:

- `blockDim` (dimensione di blocco, misurata in thread)
- `gridDim` (dimensione della griglia, misurata in blocchi)

Sono di tipo `dim3`, un vettore di interi basato su `uint3`. I campi sono sempre `x,y,z`. Ogni componente non specificata è inizializzata a 1.

Linearizzare gli indici: Ovviamente gli indici in blocchi a più dimensioni si possono linearizzare: con due indici x, y posso unificarli facendo $x + y \cdot D_x$, dove D_x è la dimensione della riga.

Possiamo tradurlo in un indice unico per i thread: per griglie e blocchi a 1D ciascuno:

```
IDth = blockIdx.x * blockDim.x + threadIdx.x
```

Si può ovviamente scalare a più dimensioni, per ottenere un'indicizzamento unico tra tutte le dimensioni.

Lanciare un kernel: Per lanciare un kernel CUDA si aggiungono tra triple parentesi angolari le dimensioni di grid e block.

```
kernel_name <<<grid, block>>>(argument list);
```

Runtime API: Alcune funzioni:

- `cudaDeviceReset()` distrugge tutte le risorse associate al device per il processo corrente, non molto usato ma si può fare
- `cudaDeviceSynchronize()` aspetta che la GPU termini l'esecuzione di tutti i task lanciati fino a quel punto, sincronizzazione host device

Per effettuare debugging, la `Synchronize` permette di "scaricare" tutti i `printf` quando servono. Altrimenti, dato che le chiamate sono asincrone, si rischia che l'applicazione lato CPU termini prima che i `printf` abbiano avuto modo di essere mostrati.

Un altro mezzo di debugging è `Kernel<<<1,1>>>`: forza l'esecuzione su un solo blocco e thread, emulando comportamento sequenziale sul singolo dato.

Proprietà dei kernel:

Qualificatori	**Esecuzione**	**Chiamata**
`__global__`	Eseguito dal device	Dall'host e dalla compute cap. 3 anche dal device
`__device__`	Eseguito dal device	Solo dal device
`__host__`	Eseguito dall'host	Solo dall'host

Restrizioni del kernel:

- Accede alla sola memoria device
- Deve restituire un tipo `void`

- Non supporta il numero variabile di argomenti
- Non supporta variabili statiche
- Non supporta puntatori a funzioni
- Esibisce un comportamento asincrono rispetto all'host

Gestione degli errori: Si ha un `enum cudaError_t` come valore di ritorno di ogni chiamata cuda. Può essere `success` o `cudaErrorMemoryAllocation`. Si può usare cudaError_t cudaGetLastError(void) per ottenere il codice dell'ultimo errore.

Misurare tempo con la CPU: Per misurare il tempo di esecuzione con la CPU serve aspettare il termine dell'esecuzione del kernel lanciato, quindi bisogna sincronizzare host e device prima di prendere il tempo di fine:

```
1 double iStart = cpuSecond();
2 kernel_name<<<grid, block>>>(argument list);
3 cudaDeviceSynchronize();
4 double iElaps = cpuSecond() - iStart;
```

3.2 Warp

Ogni thread vede:

- i suoi **registri privati**
- la **memoria condivisa** del blocco di thread

L'architettura SIMT (vedi 1) si basa sugli **warp**, (tradotto in "*trama*", termine che viene dalla tessitura), l'idea è che ci sono delle file di thread (warp), collegate assieme dall'ordito. Rappresenta i blocchi di thread, sono blocchi da 32.

Ogni Streaming Mutiprocessor SM esegue i thread in gruppi di 32, chiamati warp. Idealmente, tutti i thread in un warp eseguono la stessa cosa in parallelo allo stesso tempo (SIMD all'interno del warp).

Ogni thread ha il suo program counter e register state di conseguenza può seguire cammini distinti di esecuzione delle istruzioni (parallelismo a livello thread, disponibile da Volta in poi, prima c'era un PC solo per ogni warp).

Il valore 32 è l'unità minima di esecuzione che permette grande efficienza nell'uso della GPU, concettualmente i blocchi di 32 dovrebbero avere modello SIMD, anche se nella pratica è SIMT (più flessibile ma potenzialmente meno efficiente).

Dove si può si deve **evitare la divergenza di esecuzione** all'interno del warp. I **blocchi** vengono **divisi in warp**, quindi è meglio avere blocchi con thread multipli di 32, per evitare divergenza.

I blocchi di thread possono essere configurati logicamente in 1, 2 o 3 dimensioni, ma a livello hardware sarà una sola dimensione con id progressivo, con un warp ogni 32 thread.

Sarà quindi necessario uno scheduling per i warp (il numero di blocchi richiesto è maggiore, chi va prima in esecuzione?) all'interno dei blocchi, vengono mandati in esecuzione quando sono liberi. Ad ogni colpo di clock lo scheduler dei warp decide quale mandare in esecuzione tra quelli che

- non sono in attesa di dati dalla device memory (alta latenza, memory latency)
- non stanno completando un'istruzione precedente (pipeline delay)

Questi dettagli sono trasparenti al programmatore, serve solo a garantire un elevato numero di warp in esecuzione; vogliamo massimizzare l'occupancy (percentuale di risorse usate in ogni SM).

Se all'interno di un warp dei thread devono eseguire istruzioni diverse (e.g., per colpa di un `if`), la GPU le eseguirà sequenzialmente al posto che in parallelo, disabilitando i thread inattivi. Questa è una **divergenza** ed ha impatto negativo sull'efficienza, a volte anche in maniera significativa.

Ogni warp ha un contesto di esecuzione (runtime), trasparente al programmatore, che consta di:

- Program counters
- Registri a 32-bit ripartiti tra thread
- Shared memory ripartita tra blocchi

Di conseguenza, la memoria locale ad ogni thread è limitata, bisogna prestare attenzione alle risorse richieste simultaneamente per ogni thread, altrimenti il numero totale di thread che possono essere attivi concorrentemente si riduce.

I registri sono usati per le variabili locali automatiche scalari (che non sono array quindi) e le coordinate dei thread. I dati nei registri sono privati ai thread (scope) e ogni multiprocessor ha un insieme di 32-bit register che sono partizionati tra i warp.

Il numero di blocchi e warp che possono essere elaborati insieme su un SM per un dato kernel dipende

- dalla quantità di registri e di shared memory usata dal kernel
- dalla quantità di registri e shared memory resi disponibili dallo SM

Ogni architettura ha i suoi vincoli e noi vogliamo avvicinarci il più possibile ai limiti massimi, in modo da rendere il più efficiente possibile il programma. C'è un numero massimo di thread/blocchi/warp per SM, vogliamo fare in modo di avere l'utilizzo maggiore possibile.

Un warp attivo può essere di 3 tipi:

- Selezionato: in esecuzione su un dato path
- Bloccato: non pronto all'esecuzione
- Candidato: può essere il prossimo ad andare in esecuzione

Warp Divergence: Tutti i thread all'interno di un warp devono eseguire la stessa istruzione, quindi se sono presenti path diversi (per esempio per un `if`) si ha una *divergenza*.

Quando all'interno di un warp è presente divergenza, i molteplici path di esecuzione presenti vanno eseguiti in serie: i path vengono eseguiti uno dopo l'altro, disabilitando i thread non appartenenti a quel flusso di esecuzione.

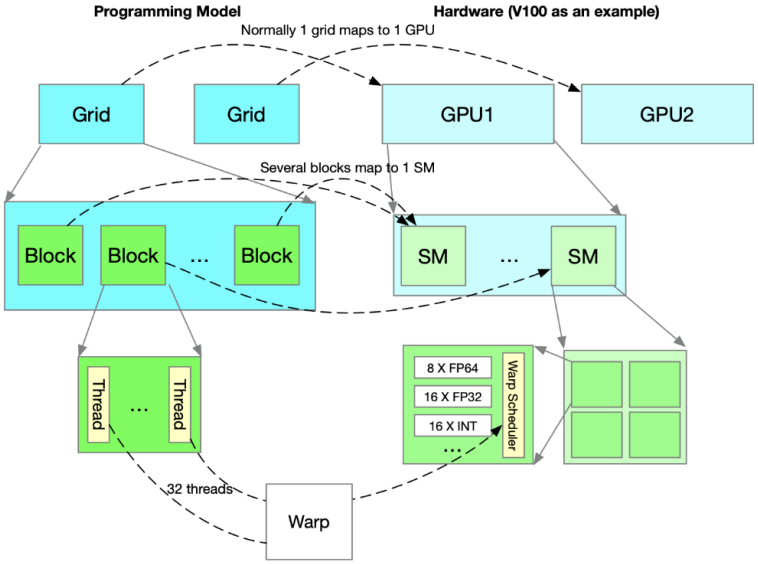
Questo porta ad un peggioramento delle prestazioni fino a 32 volte. Da notare che questo fenomeno avviene solo all'interno del warp, non vale per thread di warp differenti.

Latency Hiding: La "latenza" è il numero di cicli necessari al completamento di un'istruzione. Per massimizzare il throughput occorre che lo scheduler abbia sempre warp eleggibili a ogni ciclo di clock. Si ha così latency hiding scambiando la computazione tra warp.

Tipi di istruzioni che inducono latenza:

- Istruzioni aritmetiche: tempo necessario per la terminazione dell'operazione (`add`, `mult`, ...); 10-20 cicli di clock
- Istruzioni di memoria: tempo necessario al dato per giungere a destinazione (`load`, `store`); 400-800 cicli di clock

La griglia viene suddivisa in blocchi, il blocco in thread, i blocchi vanno all'SM.



Sincronizzazione a più livelli: Le prestazioni decrescono con l'aumentare della divergenza nei warp. **Primitive di sincronizzazione** sono necessarie per evitare race conditions in cui diversi thread accedono simultaneamente alla stessa locazione di memoria.

Si possono avere più livelli di sincronizzazione:

- **System-level**: attesa che venga completato un dato task su entrambi host e device

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

  Blocca l'applicazione host finché tutte le operazioni CUDA non sono completate;

- **Block-level**: attesa che tutti i thread in un blocco raggiungano lo stesso punto di esecuzione

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

Sincronizza i thread all'interno di un blocco: attende fino a che tutti raggiungono il punto di sincronizzazione

- **Warp-level**: attesa che tutti i thread in un warp raggiungano lo stesso punto di esecuzione

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

Sincronizza i thread all'interno di un warp: attende fino a che tutti raggiungono il punto di sincronizzazione (riconverge)

La sincronizzazione a livello di blocco va usata con attenzione, può anche portare a deadlock, un esempio semplice può essere una sincronizzazione dentro un `if-else`: potrebbero esserci thread che non entreranno mai nel ramo con la sincronizzazione, causando deadlock.

Il compilatore ha tecniche di ottimizzazione per evitare divergenza all'interno del warp (es: per un `if` calcola entrambi i branch).

3.3 Parallel reduction

Un'operazione comune che va sotto il nome di **reduction** è la somma di elementi di array, solitamente di grandi dimensioni.

Per un operatore di reduction le proprietà richieste sono commutatività e associatività; con queste gli elementi possono essere riordinati e combinati in qualsiasi modo.

L'approccio sequenziale è molto semplice, in parallelo un'idea potrebbe essere:

- Suddividere il vettore in parti più piccole
- Attivare i thread per la somma parziale sui pezzi
- Sommare tra loro i risultati parziali ottenuti

Questo approccio si può implementare in due modi:

- sommando le coppie di elementi contigui: al passaggio i il thread `tid` somma valori a distanza 2^i: `A[tid] + A[tid + 2^i]`
- sommando coppie di elementi equispaziati (stride): al passaggio i il thread `tid` somma elementi a distanza `len(A)/2^(i+1)`: `A[tid] + A[tid + len(A)/2^(i+1)]`

Queste presupponendo che la lunghezza del vettore sia una potenza di 2 e che i parta da 0.

Somma strided: La strategia parallela è simile alla somma ricorsiva con stride:

- Ad ogni passo un numero di thread pari alla metà degli elementi effettua le somme parziali (riduzione)
- Il numero di thread attivi si dimezza ad ogni passo (rinnovare la stride)
- Occorre sincronizzare il comportamento dei thread affinché al passo `t` abbiano tutti terminato il compito prima di andare al passo `t+1` (analogo alla chiamata ricorsiva)

Un possibile codice:

```
for (int stride = 1; stride < blockDim.x; stride *= 2) {
    if ((tid % (2 * stride)) == 0)
        idata[tid] += idata[tid + stride];
}
```

Questa soluzione può introdurre divergenza crescente a livello di warp: i thread attivi sono "distanti" a livello di warp, nel primo passo si attivano solo i thread con indici pari, nella seconda iterazione solo la metà pari degli indici precedenti (un quarto dei thread), …

Vogliamo eliminare la divergenza, fondamentalmente usando thread adiacenti (quindi all'interno dello stesso warp): "convertiamo" gli indici in modo da usare sempre i `tid` più bassi possibili:

```
for (int stride = 1; stride < blockDim.x; stride *= 2) {
    int index = 2 * stride * tid;
    if (index < blockDim.x)
        idata[index] += idata[index + stride];
    __syncthreads();
```

Da notare come viene eliminata la clausola `if` in tutti i thread. La locazione delle somme parziali non cambia, viene modificato solo l'indice dei thread.

3.4 Operazioni Atomiche

Per evitare race conditions, le **operazioni atomiche** in CUDA eseguono (solo) operazioni matematiche senza interruzione da altri thread. Si tratta di funzioni che vengono tradotte in istruzioni singole.

Le operazioni basilari sono:

- Matematiche: add, subtract, maximum, minimum, increment, and decrement
- Bitwise: AND, bitwise OR, bitwise XOR
- Swap: scambiano valore in memoria con uno nuovo

3.4.1 Calcolo dell'istogramma per immagini RGB

L'istogramma del colore è una rappresentazione della distribuzione (frequenza relativa/assoluta) tonale dei colori in un'immagine. Può essere costruito per ogni spazio colore, come RGB o HSV.

Per calcolare l'istogramma bisogna:

- Considerare separatamente i canali RGB dell'immagine
- Usare le operazioni atomiche per il calcolo delle frequenze
- Restituire 3 istogrammi distinti (uno per canale) di 256 valori di intensità colore

L'idea è molto semplice: si alloca una struttura dati (array lungo `3 * 256`) per mantenere i valori dell'istogramma e, per ogni pixel dell'immagine si effettua una `atomicAdd()` per incrementare il valore della frequenza dei colori presenti nel pixel.

Esempio per una immagine PPM:

```
__global__ void ppm_histGPU(PPM ppm, int *histogram) {
        uint x = blockIdx.x * blockDim.x + threadIdx.x;

        if (x >= ppm.width * ppm.height)
                return;

        color R = ppm.image[3 * x];
        color G = ppm.image[3 * x + 1];
        color B = ppm.image[3 * x + 2];
```

```
10
11          atomicAdd(&histogram[R], 1);
12          atomicAdd(&histogram[G + 256], 1);
13          atomicAdd(&histogram[B + 512], 1);
14  }
```

Se l'incremento non fosse atomico, un thread potrebbe venire interrotto da altri mentre tenta di incrementare una cella, possibilmente portando a race conditions e risultati inconsistenti.

3.5 Memoria CUDA

Per il programmatore esistono due tipi di memorie:

- **Programmabile**: controllo esplicito di lettura e scrittura per dati che transitano in memoria
- **Non programmabile**: nessun controllo sull'allocazione dei dati, gestiti con tecniche automatiche (e.g., memorie CPU e cache L1 e L2 della GPU)

Nel modello di memoria CUDA sono esposti diversi tipi di memoria programmabile:

1. registri
2. shared memory
3. local memory
4. constant memory
5. texture memory
6. global memory

Cache su GPU: Come nel caso delle CPU, le cache su GPU **non sono programmabili**. Sono presenti 4 tipi:

- **L1**, una per ogni SM
- **L2**, condivisa tra tutti gli SM
- **Read-only constant**
- **Read-only texture** (L1 da CC 5.0)

La **cache L1** è presente all'interno di ogni SM; in alcune architetture (Fermi e successive) la dimensione può essere configurata, con una porzione assegnabile a memoria condivisa. Capacità limitata ma permette di sfruttare località dei dati.

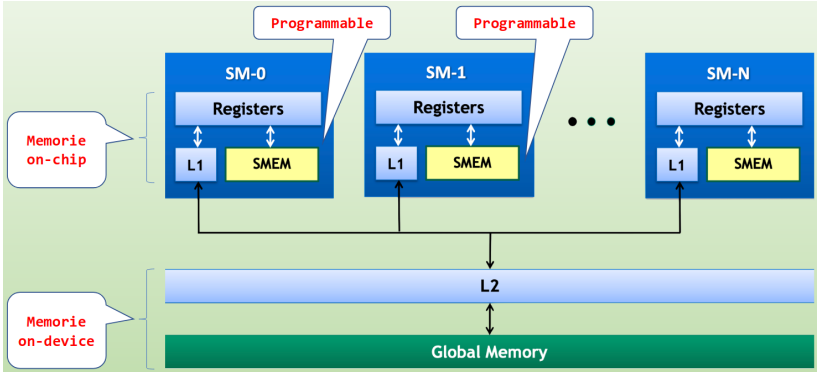
La **cache L2** ha dimensione maggiore ed è condivisa tra tutti gli SM, funziona da intermediario tra memoria globale e cache L1 dei singoli SM. Raccoglie i dati necessari a tutti gli SM e contribuisce a mantenere la coerenza dei dati tra vari SM.

L1 e L2 sono usate per memorizzare dati in memoria locale e globale, incluso lo spilling dei registri (eccessi nell'uso di local memory).

Ogni SM ha anche una **read-only constant cache** e **read-only texture cache** (non sempre fisiche) usate per migliorare le prestazioni in lettura dai rispettivi spazi di memoria sul device.

Read-only constant cache è ottimizzata per dati globali costanti condivisi tra tutti i thread, con accesso uniforme e caching efficiente. Read-only texture cache è ideale per dati in sola lettura con accesso non coalescente, sfruttando la località spaziale e offrendo funzionalità di interpolazione e filtraggio hardware.

Suddivisione fisica:



Nel tempo, è stata gradualmente aumentata la dimensione delle cache L1 e L2, allo stesso tempo incrementando la memoria condivisa tra gli SM, fino all'introduzione di una L0 instruction cache in Volta, oltre a 128kB di cache L1 unita alla shared memory (smem).

3.5.1 Cooperating Threads/Shared Memory

Un blocco può avere della **memoria condivisa** e tutti thread all'interno del blocco hanno la stessa visuale su questa memoria; la memoria è unica per blocco e inaccessibile ad altri blocchi. Viene dichiarata tramite `__shared__`.

La SMEM è suddivisa in moduli della stessa ampiezza, chiamati **bank**. Ogni richiesta di accesso fatta di n indirizzi che riguardano n distinti bank sono serviti simultaneamente.

Ogni SM ha una quantità limitata di shared memory che viene ripartita tra i blocchi di thread. La smem serve come base per la comunicazione inter-thread: i thread all'interno di un blocco possono cooperare scambiandosi dati memorizzati in shared memory. L'accesso deve essere sincronizzato per mezzo di `syncthreads()`.

Organizzazione fisica: La smem è suddivisa in blocchi da 4 byte (word), ogni accesso legge almeno la word di appartenenza (anche se viene richiesto un solo byte).

Dati 32 bank, ogni word è memorizzata in bank distinti, a gruppi di 32. Dato l'indirizzo del byte:

- diviso 4 si ottiene l'indice della word
- l'indice della word modulo 32 è l'indice della bank

Smem a runtime: La memoria viene ripartita tra tutti i blocchi residenti in un SM. Maggiore è la shared memory richiesta da un kernel, minore è il numero di blocchi attivi concorrenti.

Il contenuto della shared memory ha lo stesso lifetime del blocco a cui è stata assegnata.

Pattern di accesso: Se un'operazione di `load` o `store` eseguita da un warp richiede al più un accesso per bank, si può effettuare in una sola transizione il trasferimento dei dati dalla shared memory al warp. In alternativa sono richieste diverse (≤ 32) transazioni, con effetti negativi sulla bandwidth globale.

L'accesso ideale è una singola transazione per warp.

Ci possono essere dei **conflitti**: un **bank conflict** accade quando si hanno diversi indirizzi di shared memory che insistono sullo stesso bank.

L'hardware effettua tante transazioni quante ne sono necessarie per eliminare i conflitti, diminuendo la bandwidth effettiva di un fattore pari al numero di transazioni separate necessarie (vengono serializzati gli accessi).

Osservazioni:

- **Latency hiding**: il ritardo tra richiesta dei thread alla smem e l'ottenimento dei dati, in generale, non è un problema, anche in caso di bank conflict; lo scheduler passa a un altro warp in attesa che quelli sospesi completino il trasferimento dei dati dalla smem
- **Inter-block**: non esiste conflitto tra thread appartenenti a blocchi differenti, il problema sussiste solo a livello di warp dello stesso blocco
- **Efficienza massima**: il modo più semplice per avere prestazioni elevate è quello di fare in modo che un warp acceda a word consecutive in memoria shared
- **Caching**: con scheduling efficace, le prestazioni (anche in presenza di conflitti a livello smem) sono molto migliori rispetto alla cache L2 o global memory

3.5.2 Allocazione della SMEM

Allocazione statica: Una variabile in shared memory può anche essere dichiarata locale a un kernel o globale in un file sorgente. Viene dichiarata con il qualificatore `__shared__`. Può essere dichiarata sia staticamente sia dinamicamente.

Se statica, può essere 1D, 2D o 3D, con dimensione nota compile time.

Allocazione dinamica: Per allocare la shared memory dinamicamente (in bytes), occorre indicare un terzo argomento all'interno della chiamata del kernel:

```
1 kernel <<<grid, block, N*sizeof(int)>>>(...)
```

Se la dimensione non è nota compile time, è possibile dichiarare una variabile adimensionale con la keyword `extern`. Dinamicamente si possono allocare solo array 1D.

Allocazione dinamica multipla: Non si possono allocare multiple allocazioni di shared memory, quindi bisogna fare un'allocazione unica e utilizzare puntatori con offset all'interno dell'area allocata.

Uso tipico della shared memory:

1. Carica i dati dalla device memory alla shared memory
2. Sincronizza i thread del blocco al termine della copia che ognuno effettua sulla shared memory (così che ogni thread possa elaborare dati certi andando avanti)
3. Elabora i dati in shared memory
4. Sincronizza (se necessario) per essere certi che la shared memory contenga i risultati aggiornati
5. Scrivi i risultati dalla device memory alla host memory

3.5.3 Prodotto Convolutivo con SMEM

La `convoluzione` sono una serie di somme e prodotti

$$y(n) = h(n) \cdot x(n) = \sum_{k=0}^{N-1} h(k)x(n-k)$$

Si prende un segnale, si considera un kernel/finestra su tale segnale, si fanno i prodotti. Il tutto viene fatto un numero *molto elevato* di volte. Si può scalare il processo a più dimensioni.

Ci sono sempre problemi di bordo: cosa faccio quando la maschera considerata arriva al bordo dei dati? Andrebbe fuori, quindi devo popolare nel modo corretto i dati mancanti (`0`? Li invento?).

Tiling: Divido i dati in blocchi (ad esempio, 16 elementi in 4 blocchi da 4 thread), ogni thread nel blocco fa un prodotto della convoluzione. Per ridurre l'accesso alla global memory, in cache/memoria condivisa si tengono i dati a cui l'accesso è fatto più frequentemente, ovvero i valori del "blocco di dati" assegnato al block (tutti i thread devono calcolare sullo stesso insieme di dati, o quasi), tenendo conto della dimensione della maschera (serve avere i dati "adiacenti" al blocco (sarebbe molto utile un'immagine, si capirebbe subito)).

I dati da caricare in smem sono più dei thread nel blocco ("alone" che va al di fuori del blocco di dati stesso); in smem carico tutti i possibili dati a cui il blocco deve fare l'accesso. Chi carica che dati in memoria? I dati esterni al blocco potrebbero essere anche più del blocco stesso (maschera "grossa"). La soluzione è dare un ordine ai thread e dividere il più equamente possibile i caricamenti in memoria tra i thread del blocco.

3.5.4 Prodotto matriciale con SMEM

L'approccio più ovvio è quello di lasciare a ogni thread il calcolo di una cella della matrice prodotto. Ogni blocco si occuperebbe di una sezione della matrice prodotto risultante.

In memoria condivisa andrebbero caricate tutte le righe e colonne delle matrici da cui fare il prodotto. Per distribuire equamente il lavoro tra i thread, si può fare tiling anche delle aree da caricare (multiple della dimensione del blocco, se definite correttamente/le dimensioni in ingresso lo permettono). Una volta finito il caricamento e la sincronizzazione a livello di blocco, ogni thread può calcolare la sua entry della matrice.

Pseudocodice del kernel:

```
1  __shared__ As[WIDTH][WIDTH];
2  __shared__ Bs[WIDTH][WIDTH];
3  nblocks = numero di block nelle sotto-matrici da
     caricare;
4  for i ∈ nblocks do
5  |  As[tidy][tidx] = A[tidx_abs + i *
   |   WIDTH][tidy_abs];
6  |  Bs[tidy][tidx] = B[tidx_abs][tidy_abs + i *
   |   WIDTH];
7  |  __syncthreads();
8  |  for j ∈ WIDTH do
9  |  |  partial_sum += As[tidy][j] * Bs[j][tidx];
10 |  __syncthreads();
   ;    // Srivere il risultato in memoria globale
```

L'idea è:

- Caricare il primo tile
- Effettuare la somma parziale per ogni cella

- Ripetere per il numero di tile contenuti nelle zone delle matrici da caricare
- Scrivere i risultati finali in memoria globale

3.6 Global Memory

Nei computer moderni esiste una gerarchia di memorie per minimizzare latenze e massimizzare throughput. In genere, si ha l'illusione virtuale di una grande memoria, tutta a bassa latenza, anche se la memoria con effettivamente bassa latenza è poca e si ha una memoria ad alta capacità e alta latenza.

All'interno delle GPU abbiamo, dalla latenza più alta alla più bassa:

- Device Memory
- L2 Cache
- L1/shared
- Registers

Le gerarchie di memorie, comprese quelle CUDA, fanno fede ai principi di:

- **Località spaziale**: se l'istruzione all'indirizzo i viene eseguita, probabilmente dopo verrà eseguita quella all'indirizzo $i + \Delta i$
- **Località temporale**: se un'istruzione viene eseguita al tempo t, probabilmente verrà eseguita anche al tempo $t + \Delta t$ (dove Δt è piccolo)

Registers: Le memorie più veloci in assoluto, con lifetime del kernel. Vengono ripartiti tra i warp attivi, le variabili dichiarate nel codice device senza qualificatori generalmente risiedono in un registro.

Meno registri usa il kernel, più blocchi di thread è probabile che risiedano sull'SM (il compilatore usa un'euristica per ottimizzare questo parametro). **Register spilling**: se si usano più registri di quelli consentiti le variabili si riversano nella local memory.

Local Memory: Si tratta di una memoria *lenta* (collocata off-chip, alta latenza, bassa bandwidth). Si tratta di una memoria locale ai thread.

Usata per contenere le variabili automatiche (grandi) non contenute nei registri, o per le variabili al di fuori causa spilling. La local memory risiede nella device memory, pertanto gli accessi hanno stessa latenza e ampiezza di banda della global memory e sono soggetti anche agli stessi vincoli di coalescenza-

Da CC 2.0 ci sono parti poste in cache L1 a livello di SM e in cache L2 a livello di device. Il compilatore `nvcc` si preoccupa della sua allocazione e non è controllata dal programmatore.

Constant Memory: Risiede nella device memory (64K per tutte le CC) ed ha una cache dedicata in ogni SM (8K). Definibile tramite l'attributo `__constant__`.

Ospita dati in sola lettura, ideale per accessi uniformi. Ha scope globale, va dichiarata al di fuori di qualsiasi kernel e viene dichiarata staticamente, quindi è visibile a tutti i kernel nella stessa unità di compilazione.

La constant memory può essere inizializzata dall'host usando:

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

Lavora bene quando tutti i thread di un warp leggono dallo stesso indirizzo di memoria (raggiunge l'efficienza dei registri); se i thread di un warp leggono da indirizzi diversi allora le letture vengono serializzate, riducendo l'efficienza.

Texture Memory: Risiede nella device memory e (può avere) una read-only cache per-SM ed è acceduta solo attraverso di essa. La cache include un supporto hardware efficiente per filtraggio o interpolazione floating-point nel processo di lettura dei dati.

Ottimizzata per la località spaziale 2D, quindi dati espressi sotto forma di matrici. I thread in un warp che usano la texture memory per accedere a dati 2D hanno migliori prestazioni rispetto a quelle standard, quindi è adatta per applicazioni in cui servono classiche elaborazioni di immagini/video. Per altre applicazioni l'uso della texture memory potrebbe essere più lento della global memory.

Global Memory: La più grande, con più alta latenza e più comunemente usata memoria su GPU. Ha scope e lifetime globale (da qui il nome).

Dichiarazione (codice host):

Statica	`__device__ int a[N];`
Dinamica	`cudaMalloc((void **)&d_a, N);  cudaFree(d_a);`

Corrisponde alla memoria fisica, con "global" si intende una divisione logica. L'accesso da parte di thread appartenenti a blocchi distinti può potenzialmente portare a modifiche incoerenti. La global memory è accessibile attraverso transazioni da 32, 64, o 128 byte; le transazioni avvengono solo per gruppi di valori, non si può accedere a un valore singolo.

I valori contenuti nella memoria allocata non sono inizializzati, ma si possono inizializzare con dati provenienti dall'host (`cudaMemcpy`) oppure con un valore specifico

```
1 cudaError_t cudaMemset(void* devPtr, int value, size_t count)
```

Assegna il valore `value` a tutti gli indirizzi contenuti nel blocco di memoria.

La memoria allocata è opportunamente allineata per ogni tipo di variabile. La `cudaMalloc` restituisce `cudaErrorMemoryAllocation` in caso di fallimento.

Lo **specificatore `__device__`** indica una variabile che risiede unicamente sul device. Risiede nella memoria globale (e quindi oggetti distinti per device diversi), ha il lifetime del contesto CUDA in cui è stata creata. Può essere acceduta da tutti i thread e dall'host tramite la libreria runtime:

- `cudaGetSymbolAddress()`, `cudaGetSymbolSize()`: per ottenere indirizzo e dimensione di una variabile, rispettivamente
- `cudaMemcpyToSymbol()`, `cudaMemcpyFromSymbol()`: per copiare verso e da una variabile, rispettivamente

Riassunto dichiarazione di variabili:

QUALIFIER	VARIABLE	MEMORY	SCOPE	LIFESPAN
	`float var`	Register	Local	Thread
	`float var[100]`	Local	Local	Thread
`__shared__`	`float var`	Shared	Block	Block
`__device__`	`float var`	Global	Global	Application
`__constant__`	`float var`	Constant	Global	Application

3.7 Pinned memory

La pinned memory (o page-locked memory) in CUDA è una tecnica che serve per ottimizzare il trasferimento dei dati tra la memoria del sistema (RAM) e la memoria della GPU (VRAM).

Si vuole evitare il page fault della virtual memory (CPU, di default la memoria host allocata è paginabile). Esistono delle primitive per definire una memoria pinned, ovvero viene tolta la pagina dal meccanismo di virtualizzazione in modo che l'host non possa "toglierla" mentre il device la deve usare. Blocca la memoria in modo da poter fare trasferimenti asincroni al device.

Una volta "pinnata", la memoria non sparirà dal sistema di virtualizzazione automatico della memoria host, quindi si può lavorare su quella memoria in maniera asincrona. La pinned memory può essere acceduta direttamente dal device, in modalità asincrona. Può essere letta e scritta con una bandwidth più alta rispetto alla memoria paginabile.

Da notare che eccessi di allocazione di pinned memory potrebbero far degradare le prestazioni dell'host (ridurre la memoria paginabile inficia l'uso della virtual memory), Per allocare esplicitamente memoria pinned:

```
1 cudaError_t cudaMallocHost(void **devPtr, size_t count);
```

E per deallocarla:

```
1 cudaError_t cudaFreeHost(void *devPtr);
```

Questa allocazione sostituisce la malloc "normale", su host. Rende i trasferimenti host-device significativamente più veloci, al costo di un tempo più alto di allocazione.

3.8 Unified Virtual Addressing UVA

Si vuole avere un unico spazio di indirizzamento tra CPU e tutte le GPU. Tutti i puntatori (CPU e GPU) appartengono allo stesso spazio di indirizzi virtuali, di conseguenza è possibile passare un puntatore da host a device e viceversa senza ambiguità, entrambi possono "capire" a cosa punta quell'indirizzo.

La unified memory è una memoria "*comoda*", fornisce un puntatore unico per tutte le CPU e GPU presenti nel sistema. Spazio di indirizzamento unico per CPU e GPU.

Con "**Managed Memory**" si fa riferimento ad allocazioni della unified memory. All'interno di un kernel si possono usare entrambi i tipi di memoria:

- managed memory, controllata dal sistema
- un-managed memory, controllata esplicitamente dall'applicazione

Tutte le operazioni CUDA valide sulla memoria del dispositivo sono valide anche sulla memoria managed.

Per fare allocazione dinamica:

```
cudaError_t cudaMallocManaged(void **devPtr, size_t size,
    unsigned int flags=0)
```

"rimpiazza" `cudaMalloc`, la `flag` indica chi condivide il puntatore con il device:

- `cudaMemAttachHost`: solo la CPU
- `cudaMemAttachGlobal`: anche tutte le altre GPU

Nuova **keyword**: `__managed__`, si tratta di un qualifier che denota scope globale, accessibile da CPU e GPU.

Con l'uso "misto" di memoria bisogna porre attenzione alla sincronizzazione tra CPU e GPU, onde evitare problemi.

3.9 Pattern di Accesso alla Global Memory

Gli accessi alla memoria del dispositivo possono avvenire in transazioni da 32, 64 o 128 byte. Le applicazioni GPU tendono (a volte) ad essere limitate dalla memory bandwidth, quindi massimizzare il throughput effettivo è importante.

In generale, per rendere efficienti le transazioni in memoria:

- minimizzare il numero di transazioni per servire il massimo numero di accessi
- considerare che il numero di transazioni e throughput ottenuto variano con la CC

Per migliorare le prestazioni in lettura e scrittura occorre ricordare che:

- le istruzioni vengono eseguite a livello di warp e gli accessi in memoria dipendono dalle operazioni svolte nel warp
- per un dato indirizzo si esegue un'operazione di loading o storing (gestione diversa)
- i 32 thread presentano una singola richiesta di accesso, che viene servita da una o più transazioni in memoria

In base a come sono distribuiti gli indirizzi di memoria, gli accessi alla stessa possono essere classificati in pattern distinti. Tutti gli accessi a memoria globale passano dalla cache L2, molti passano anche dalla L1. Se entrambe le memorie vengono usate gli accessi sono da 128 byte, altrimenti, se viene usata solo la L2, gli accessi sono a 32 byte.

Per le architetture che usano cache L1, queste possono essere esplicitamente abilitate o disabilitate a compile time.

Bisogna rispettare allineamento e coalescenza per sfruttare al meglio le transazioni di memoria; per avere accessi in memoria efficienti è necessario combinare in un unica transazione accessi multipli a memoria allineati e coalescenti.

Accesso **allineato**: quando il primo indirizzo della transazione è un multiplo pari della granularità della cache che viene usata per servire la transazione (32 byte per la cache L2 o 128 byte per la cache L1).

Accesso **coalescente:** quando tutti i 32 thread in un warp accedono a un blocco contiguo di memoria.

In un SM i dati seguono pipeline attraverso i seguenti tre cache/buffer paths dipendentemente da quale tipo di device memory si accede:

- L1/L2 cache
- Constant cache
- Read-only cache

L1/L2 cache rappresenta il default path. Il fatto che un'operazione di `load` in global memory passi attraverso la cache L1 dipende da CC e compiler options.

Scritture: Le write vengono servite in modo diverso, non viene usata la cache L1, ma le `store` sono cachate solo in L2, prima di essere inviate alla

device memory in segmenti da 32 byte; vengono trasferiti 1,2 o 4 segmenti alla volta.

Quando forzati a fare accessi (letture/scritture) non coalescenti si può usare la shared memory come "passaggio" per rendere le operazioni effettive in memoria coalescenti.

AoS vs SoA: I dati possono essere divisi in:

- **Array of Structures AoS**:

```
struct Particle { float x, y, z; };
Particle* P;
float x = P[idx].x;  // stride = sizeof(Particle)
```

  Questo porta a distanza tra accessi (stride) alta, rompendo la coalescenza

- **Structure of Arrays SoA**:

```
float *Px, *Py, *Pz;
float x = Px[idx];   // stride = 1
```

  In questo modo lo stride è ridotto, riducendo così il numero di transazioni necessarie

TL;DR: Un warp può effettuare accessi

- coalescenti: i 32 thread leggono dati contigui, massima efficienza
- non coalescenti/strided: dati a distanza > 1, possono servire più transazioni per la stessa quantità di dati, fino a 32 diverse

In generale (per CC superiori a 2) le transazioni coprono 128 byte. All'interno di un singolo segmento da 128 byte, la memoria è organizzata in "banks" (4 da 32 byte solitamente), anche se un thread legge solo 4 byte, dovrà trasferire l'intero bank.

La cache L1 serve load/store anche con granularità a 32 byte.

4 Ottimizzazione delle Prestazioni

4.1 Risorse Hardware

Device Query: Per indagare le feature disponibili sul device, scoprire le proprietà. Ad esempio: quanti SM sono disponibili, quanta memoria, ...

Per farlo ci sono `Funzioni delle API runtime di CUDA` e la CLI utility `nvidia-smi`. Quest'ultimo permette di gestire e monitorare le GPU presenti.

Le funzioni:

```
cudaError_t cudaGetDeviceCount(&dev_count)
cudaError_t cudaGetDeviceProperties(cudaDeviceProp* prop,
    int device);
```

Permettono di indagare il numero di device disponibili sul sistema e restituire le proprietà del device nella struttura `cudaDeviceProp` (rispettivamente).

4.2 Gestione ottimizzata delle risorse

L'ottimizzazione delle performance si basa su 4 strategie principali:

- massimizzare l'utilizzazione tramite massimo parallelismo
- ottimizzare l'utilizzo di memoria per avere il throughput di memoria massimo
- ottimizzare l'uso di istruzioni per avere il massimo throughput
- minimizzare il memory thrashing

Che strategie permettono di ottenere le migliori performance per una determinata applicazione dipende da qual'è il fattore limitante all'interno della stessa. Gli sforzi per l'ottimizzazione vanno quindi costantemente direzionati monitorando i fattori che limitano le performance, tramite strumenti come il CUDA profiler.

Register spilling: Il massimo numero di registri per thread può essere definito manualmente compile time con l'opzione `-maxrregcount` e si può indagare (sempre compile time) con `--ptxas-options=-v`.

Limitare il numero porta a fare spilling (quindi usare la memoria locale), ma permette di aumentare il numero di blocchi in esecuzione concorrentemente.

4.3 Profiling

Nvidia mette a disposizione dei **developer tools** per effettuare profiling e monitorare le applicazioni.

Nsight Compute: Profiler di livello kernel che fornisce informazioni dettagliate sulle metriche di esecuzione dei kernel CUDA. Permette una misurazione dettagliata delle prestazioni dei kernel (latency, throughput, utilizzo delle risorse, ecc.), analisi delle performance a livello di istruzione e accesso alla memoria, supporto per personalizzare la raccolta di metriche e approfondire l'ottimizzazione delle singole funzioni CUDA. `ncu, ncu-ui`, CLI e GUI.

Nsight Systems: Offre un'analisi a livello di sistema, ideale per identificare bottleneck nell'interazione tra CPU e GPU. Fornisce una visione d'insieme dell'intero flusso applicativo, monitorando la sincronizzazione tra processi e thread, il trasferimento dei dati e l'esecuzione complessiva. Permette di analizzare come le attività CUDA si integrino con il resto dell'applicazione, evidenziando le possibili ottimizzazioni per bilanciare meglio l'utilizzo di tutte le risorse hardware. `nsys, nsys-ui`, CLI e GUI.

4.4 Loop Unrolling

Il loop unrolling può essere utile per ottimizzare i cicli: questi vengono espansi ("srotolati") in modo da ridurre l'effettivo numero di iterazioni necessarie durante l'esecuzione del kernel. Il corpo del ciclo viene riscritto più volte. Utile quando il numero di iterazioni è conosciuto a priori.

Questo ha diversi vantaggi, tra cui:

- riduzione dell'overhead dovuto ai controlli del ciclo
- eliminazione di salti e riduzione della logica di controllo
- aumento del livello di parallelismo

Il numero di copie del corpo del loop create si chiama **unrolling factor** (quanto è stato "srotolato" il ciclo). Questa tecnica è efficace quando il numero di iterazioni è noto a priori.

Warp unrolling: L'ottimizzazione si può anche migliorare sfruttando il concetto di warp. Tutti i 32 thread all'interno di un solo warp eseguono lo stesso codice in maniera sincrona, si usa questa caratteristica per unrollare il codice di un ciclo in maniera esplicita, eliminando controlli ed eventuali divergenze tra thread.

Dato che tutti gli warp eseguono lo stesso codice, l'unrolling garantisce che il flusso di esecuzione rimanga uniforme, riducendo la divergenza.

4.5 Parallelismo dinamico

Ci siamo mai chiesti se si può lanciare un kernel all'interno di un kernel? Not really, ma potrebbe essere utile (come ad esempio per la ricorsione). Nuova feature introdotta dalle CC 3.5: ogni kernel può lanciare un altro kernel e gestire dipendenze inter-kernel.

Elimina la necessità di comunicare con la CPU, rende più semplice creare e ottimizzare pattern di esecuzione ricorsivi e data-dependent. Senza parallelismo dinamico la CPU deve occuparsi di lanciare ogni kernel.

L'idea dietro il parallelismo dinamico è generare dinamicamente kernel in base ai dati: se ci sono elementi diversi/zone della matrice di lavoro che richiedono sforzi diversi possiamo fare in modo che i kernel siano *ad hoc* per migliorare l'efficienza.

Senza permettere al kernel di lanciare altri kernel il modello di esecuzione è inefficiente: la CPU non può essere conscia dei dati, ma è lei che deve lanciare *tutti* i kernel. In questo modo la GPU può valutare se è necessario lanciare nuovi kernel (in base ai dati) e tali informazioni vanno passate nuovamente alla CPU per lanciare nuovi kernel.

La soluzione è il **parallelismo dinamico**: la GPU può lanciare nuovi kernel, permettendo di ridurre la dipendenza dalla CPU e migliorare il throughput del kernel (se fatto bene). Consente carichi di lavoro dinamici senza penalizzare le prestazioni.

Vogliamo mettere carico di lavoro dove serve e scegliere la granularità del lavoro in base ai dati. Possiamo posporre la decisione delle dimensioni di blocchi e griglia fino a runtime. Possiamo adattare il lavoro in base a **decisioni data-driven**, non da schemi fissi come visto fino ad ora.

Esempio: un kernel figlio viene chiamato all'interno di un kernel padre e quest'ultimo può utilizzare i risultati prodotti dal figlio senza nessuna inter-

azione da parte della CPU

```
__global__ ChildKernel(void* data) {
    //Operate on data
}
__global__ ParentKernel(void* data) {
    ChildKernel<<<16, 1>>>(data);
}
// In Host Code
ParentKernel<<<256, 64>>(data);
```

Sarebbe da limitare un attimo l'annidamento: se ogni thread facesse una chiamata a kernel figlio *potrebbero* diventare tanti kernel lanciati; sarebbe carino inserire **control flow attorno ai lanci**, per esempio limitando il lancio ad 1 per blocco del padre (`threadIdx.x == 0`).

Sincronizzazione: Si ha una **sincronizzazione implicita**, il padre non può terminare prima dei figlio, un kernel non è considerato completato finché ha figli attivi. Rimane la possibilità di avere sincronizzazione esplicita, altrimenti il kernel padre non ha garanzie di poter vedere i dati elaborati dal figlio.

4.6 Librerie CUDA

Le librerie sono comode e quelle CUDA sono accelerate dalla GPU. Le API di molte di queste sono volutamente simili a quelle della libreria standard. Permettono porting di codice da sequenziale a parallelo con *minimo sforzo*, nessun tempo di mantenimento della libreria.

Esempi di librerie CUDA:

Libreria	**Dominio**
cuFFT (NVIDIA)	Fast Fourier Transforms Linear
cuBLAS (NVIDIA)	Linear Algebra (BLAS Library)
cuSPARSE (NVIDIA)	Sparse Linear Algebra
cuRAND (NVIDIA)	Random Number Generation
NPP (NVIDIA)	Image and Signal Processing
CUSP (NVIDIA)	Sparse Linear Algebra and Graph Computations
CUDA Math Library (NVIDIA)	Mathematics
Trust (terze parti)	Parallel Algorithms and Data Structures
MAGMA (terze parti)	Next generation Linear Algebra

Workflow tipico: Per l'utilizzo di una libreria CUDA, il workflow generico è:

1. Creare un **handle** specifico della libreria (per la gestione delle informazioni e relativo contesto in cui essa opera, es. uso degli stream)
2. **Allocare la device memory** per gli input e output alle funzioni della libreria (convertirli al formato specifico di uso della liberia, es. converti array 2D in column-major order)
3. **Popolare con i dati** nel formato specifico
4. **Configurare** le computazioni per l'esecuzione (es. dimensione dei dati)
5. Eseguire la **chiamata della funzione** di libreria che avvia la computazione sulla GPU
6. **Recuperare i risultati** dalla device memory
7. Se necessario, **(ri)convertire i dati** nel formato specifico o nativo dell'applicazione
8. **Rilasciare le risorse** CUDA allocate per la data libreria

4.6.1 cuBLAS - Basic Linear Algebra Subproblems

Usata per calcolo scientifico ed ingegneristico per problemi di algebra lineare numerica

- risoluzione di sistemi lineari
- ricerca di autovalori e/o autovettori
- calcolo della SVD (valori e vettori singolari)
- fattorizzazione di matrici

Come BLAS, le funzioni di cuBLAS sono divisi in livelli:

- Livello 1: per operazioni vettore-vettore
- Livello 2: per operazioni vettore-matrice
- Livello 3: per operazioni matrice-vettore

Usa **column-major order** (leggo le colonne dall'alto verso il basso) perché chiunque ha scritto la libreria è stronzo (colpa di Fortran). Esempio:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 4 & 7 & 2 & 5 & 8 & 3 & 6 & 9 \end{bmatrix} \quad I(r,c) = c \cdot M + r$$

Dove (r, c) sono le coordinate del valore cercato e M è l'altezza della matrice (dimensioni $M \times N$).

Operare con cuBLAS: L'iter tipico per usare cuBLAS è

1. creare un handle con `cublasCreateHandle`
2. allocare la memoria sul device con `cudaMalloc`
3. popolare la device memory con gli input necessari usando `cublasSetVector` e `cublasSetMatrix`
4. effettuare le chiamate di libreria necessarie
5. recuperare i risultati dalla device memory usando `cublasGetVector` e `cublasGetMatrix`
6. rilasciare le risorse CUDA e cuBLAS con `cudaFree` e `cublasDestroy`, rispettivamente

Funzioni all'interno di cuBLAS: Per trasferire vettori da CPU a GPU:

- Copia `n` elementi di dimensione `elemSize` da `cpumem` sulla CPU ad un vettore `gpumem` sulla GPU

```
cublasSetVector(int n, int elemSize, const void *cpumem,
    int incx, void *gpumem, int incy)
```

- L'inverso di prima (da GPU a CPU)

```
cublasGetVector(int n, int elemSize, const void *gpumem,
    int incx, void *cpumem, int incy)
```

Per trasferire matrici (sempre column-major order):

- copia una matrice `rows` $\times$ `cols`, di elementi grossi `elemSize`, da `A` nella memoria CPU a `B` nella memoria GPU

```
cublasSetMatrix(int rows, int cols, int elemSize,
    const void *A, int lda, void *B, int ldb)
```

esiste anche il corrispettivo `cublasGetMatrix()` che fa l'inverso

- come `cublasGetMatrix()`, ma asincrono (rispetto all'host), usando il parametro `stream` fornito

```
cublasGetMatrixAsync(int rows, int cols, int elemSize,
    const void *A, int lda, void *B,
    int ldb, cudaStream_t stream)
```

Per gestire la libreria serve un **handle**, il quale si può generare tramite

```
cublasCreate(cublasHandle_t* handle)
```

Viene passato ad ogni chiamata di funzione della libreria successiva. Al termine

```
cublasDestroy(cublasHandle_t* handle)
```

per distruggerlo. Il tipo dell'handle è `cublasHandle_t`. Esiste un tipo `cublasStatus_t` usato per il report degli errori.

Per trasferimenti device-device: copia `n` elementi da `x` a `y`:

```
cublasScopy(handle, n, x, incx, y, incy)
```

In generale la libreria segue una naming convention `cublas<T>operation`, dove `<T>` può essere:

- `S` per parametri di tipo `float`
- `D` per `double`
- `C` per `complex floats`
- `Z` per `complex double`

Ad esempio, per l'operazione `axpy` le funzioni disponibili sono `cublasSaxpy`, `cublasDaxpy`, `cublasCaxpy`, `cublasZaxpy`.

Si usa un valore di tipo `cublasOperation_t` per indicare operazioni su matrici all'interno di funzioni:

- `CUBLAS_OP_N` per non-transpose
- `CUBLAS_OP_T` per transpose
- `CUBLAS_OP_C` per conjugate transpose

Per fare

$$result = \sum_{i=1}^{n} x[k] \cdot y[j], \quad k = 1 + (i-1) \cdot incx, \quad j = 1 + (i-1) \cdot incy$$

tra vettori `x` e `y` di `n` elementi (dimensione dei tali nella naming convention) e mettere il risultato in `result`

```
cublasStatus_t cublasSdot(cublasHandle_t handle, int n,
    const float *x, int incx, const float *y,
    int incy, float result)
```

Per fare

$$y[i] = \alpha \cdot x[i] + y[i] \quad \forall i \in n$$

con vettori `x` e `y` di dimensione `n`, risultato nel secondo vettore `y`

```
cublasStatus_t cublasSaxpy(cublasHandle_t handle, int n,
    const float *alpha, const float *x, int incx,
    const float *y, int incy)
```

Per fare

$$y = \alpha Ax + \beta y$$

dove α e β sono scalari, A è una matrice, x e y sono vettori

```
cublasStatus_t cublasSgemv(cublasHandle_t handle,
    cublasOperation_t trans, int m, int n,
    const float *alpha, const float *A,
    int lda, const float *x, int incx,
    const float *beta, float *y, int incy)
```

Per fare

$$C = \alpha AB + \beta C$$

dove α e β scalari, A, B e C matrici

```
cublasStatus_t cublasSgemm(cublasHandle_t handle,
    cublasOperation_t transa, cublasOperation_t transb,
    int m, int n, int k, const float *alpha,
    const float *A, int lda,
    const float *B, int ldb,
    const float *beta, float *C, int ldc)
```

4.6.2 cuRAND

La libreria cuRAND fornisce semplici ed efficienti **generatori di numeri**. Permette sequenze:

- Pseudo-random: soddisfa proprietà statistiche di una vera sequenza random, ma generata da un algoritmo deterministico
- Quasi-random: sequenza di punti n-dimensionali uniformemente generati secondo un algoritmo deterministico

La libreria si compone di due parti:

- `curand.h` per l'host
- `curand_kernel.h` per il device

Host API: Dalla `documentazione`, passaggi:

1. Crea un **nuovo generatore** del tipo desiderato con `curandCreateGenerator()`
2. Setta i **parametri** del generatore; ad esempio: `curandSetPseudoRandomGeneratorSeed()` per settare il seed
3. Alloca la memoria device con `cudaMalloc()`
4. Genera i valori casuali necessari con `curandGenerate()` (o altre funzioni)
5. Usa i valori
6. Quando non serve più il generatore va distrutto con `curandDestroyGenerator()`

Alcune funzioni per l'host:

- Per creare il generatore

```
curandCreateGenerator(&g, GEN_TYPE)
```

  Dove il parametro `GEN_TYPE` può essere `CURAND_RNG_PSEUDO_DEFAULT`, oppure `CURAND_RNG_PSEUDO_XORWOW` (differenze trascurabili)

- Per impostare il seed

```
curandSetRandomGeneratorSeed(g, SEED)
```

  ma importa poco, uno qualunque va bene (e.g, `time(NULL)`)

- Per generare una distribuzione

```
curandGenerate_____(...)
```

dipende dalla distribuzione che si vuole generare, ad esempio: `curandGenerateUniform(g, src, n)` oppure `curandGenerateNormal(g, src, n, mean, stddev)`.

- Per distruggere il generatore

```
curandDestroyGenerator(g)
```

La funzione `curandGenerate()` permette di generare valori in maniera asincrona, molto più veloce per quantità elevate di valori. Usare questa libreria richiederebbe poi di dover passare i dati generati alla GPU (`src` all'interno della funzione è un puntatore host), introducendo overhead. Per risolvere si può usare la Device API.

Device API: Per generare valori sul device:

1. Pre-allocare un set di cuRAND state objects nella device memory per ogni thread (gestiscono lo stato)
2. Opzionale, pre-allocare device memory per tenere i valori generati (se devono poi essere passati all'host o essere mantenuti per kernel successivi)
3. Inizializzare lo stato di tutti gli state objects con una kernal call
4. Chiamare una funzione cuRAND per generare valori casuali usando gli state objects allocati
5. Opzionale, trasferire i valori all'host (se è stata allocata la memoria in precedenza)

4.7 Stream e Concorrenza

Si possono avere diversi gradi di concorrenza in CUDA:

- **CPU/GPU concurrency** (modello ibrido): si tratta di dispositivi distinti e operano indipendentemente
- **Memcpy/kernel processing concurrency**: grazie al DMA il trasferimento tra host e device può avere luogo mentre gli SM processano i kernel
- **Kernel concurrency**: si possono eseguire fino a 128 kernel in parallelo, anche da thread di CPU distinti

- **Grid-level concurrency**: uso di stream multipli per operazioni indipendenti
- **Multi-GPU concurrency**: si può ripartire il carico tra multiple GPU che lavorano in parallelo

4.7.1 CUDA Streams

Uno **stream CUDA** è riferito a sequenze di operazioni CUDA asincrone eseguite dal device, nell'ordine che viene stabilito dal codice host. Queste operazioni vengono inserite in una coda FIFO (incapsulate dallo stream), per poi essere gestita dallo scheduling (devono essere serviti).

Operazioni tipiche possono essere: trasferimento dati, lancio kernel, gestione eventi di sincronizzazione. L'esecuzione di operazioni in uno stream è sempre asincrona rispetto all'host.

Le operazioni appartenenti a stream distinti non hanno restrizioni sull'ordine di esecuzione l'uno con l'altro (ma possono essere imposte); ogni stream è asincrono rispetto all'host e sono tutti indipendenti l'uno con l'altro.

Parallelismo Grid-level: Dal punto di vista CUDA le operazioni di stream distinti vengono eseguite in parallelo (concorrentemente). I comandi immessi su uno stream possono essere eseguiti quando tutte le dipendenze del comando sono soddisfatte. Le dipendenze possono essere comandi lanciati in precedenza sullo stesso flusso o dipendenze da altri flussi; i.e., ogni stream è indipendente da tutti gli altri, idealmente vengono eseguiti tutti in parallelo.

Il completamento con successo della chiamata di sincronizzazione garantisce il completamento corretto di tutti i comandi lanciati.

Creare API Stream: Bisogna inserire degli oggetti che si chiamano "stream". Passaggi:

- creare uno stream non nullo

```
cudaError_t cudaStreamCreate(cudaStream_t* pStream );
```

- lancio del kernel

```
kernel_name<<< grid, block, sharedMemSize, pStream >>>
    (argument list);
```

- eliminazione di stream:

```
cudaError_t cudaStreamDestroy(cudaStream_t pStream );
```

Anche le operazioni di trasferimento: per **allocare** spazio su **pinned memory**:

```
cudaError_t cudaMallocHost(void **ptr, size_t size);
cudaError_t cudaHostAlloc(void **pHost, size_t size,
    unsigned int flags);
```

Alloca su host memoria non paginabile (pinned memory), `flag` indica specifiche proprietà di allocazione (se `0` le due API sono uguali). In seguito, per fare **trasferimento asincrono** basato su pinned memory

```
cudaError_t cudaMemcpyAsync(void* dst, const void* src,
    size_t count, cudaMemcpyKind kind,
    cudaStream_t stream );
```

Tipi di stream: Le operazioni CUDA vengono eseguite esplicitamente o implicitamente su uno stream. Ne esistono di due tipi:

- dichiarato implicitamente (NULL stream o deault stream, si può indicare esplicitamente con `0` al posto del valore di stream nella chiamata a kernel)
- dichiarato esplicitamente (non-NULL stream)

Il default stream interviene quando non viene usato esplicitamente uno stream. Il comportamento in relazione agli altri stream dipende dalla flag di compilazione:

- `--default-stream legacy` (or noflag): vecchio comportamento in cui un lancio di `cudaMemcpy` o del kernel sullo stream predefinito si blocca/sincronizza con altri stream
- `--default-stream per-thread`: nuovo comportamento in cui il default stream non influenza gli altri

Maintaining Occupancy: La situazione ideale è avere kernel grandi che occupano completamente il device. Kernel piccoli possono occupare il device in maniera meno organizzata, portando a sequenzializzazione all'interno dello stream.

Dividere su più stream i kernel "piccoli" permette di mantenere l'efficienza togliendo dei vincoli di sequenzialità che porterebbe ad una situazione di bassa occupancy.

Meglio usare stream (asincroni concorrenti) non default per:

- sovrapporre articolate computazioni host e device
- sovrapporre computazioni host e trasferimento dati host-device
- sovrapporre trasferimento dati host-device e computazioni device
- computazioni concorrenti su device

Dalla Cuda Programming Guide:

- le applicazioni gestiscono la concorrenza attraverso gli stream
- uno stream è una sequenza di comandi (anche da thread host diversi) eseguiti in ordine
- stream diversi potrebbero eseguire i comandi senza rispettare l'ordine relativo tra loro o in maniera concorrente

Insomma, l'idea è alzare un'altra volta il grado di concorrenza, non abbiamo più kernel sequenziali le cui istruzioni sono eseguite in parallelo, anche i kernel possono essere eseguiti parallelamente tra loro su stream diversi.

Overlapping behavior: Si possono avere diversi tipi di sovrapposizioni:

- overlap **traferimento dati ed esecuzione kernel**: alcuni dispositivi possono avere trasferimenti asincroni da o verso la GPU concorrentemente all'esecuzione di kernel; per controllare se presente proprietà `asyncEngineCount` (non zero vuol dire supportata); la memoria host deve essere page-locked
- esecuzione di **kernel concorrenti**: da CC 2.x in su si possono avere multipli kernel concorrenti; si può verificare il supporto tramite la proprietà `concurrentKernels`; il numero massimo di kernel concorrenti possibili dipende dalla CC, 128 recentemente
- **trasferimenti dati concorrenti**: si possono sovrapporre copie da e verso il device (per i device che supportano la cosa, `asyncEngineCount` a 2); per avere sovrapposizione la memoria host coinvolta deve essere page-locked

Stream Synchronize: Tutte le operazioni sono asincrone, può essere utile controllare se tutte le operazioni in uno stream sono state completate o meno.

Blocco dell'host sullo stream:

```
cudaError_t cudaStreamSynchronize(cudaStream_t stream);
```

Forza il blocco dell'host fino a che tutte le operazioni dello stream sono state completate. Da notare che `cudaDeviceSynchronize()` blocca l'host finché non sono stati completati tutti i comandi su tutti gli stream.

Controllo stream completato:

```
cudaError_t cudaStreamQuery(cudaStream_t stream);
```

Controlla se le operazioni sono completate ma non forza blocco dell'host in caso negativo. Ritorna `cudaSuccess` o `cudaErrorNotReady`.

Sovrapporre kernel e trasferimento dati: Devono essere verificati diversi requisiti perché si possa effettuare questa sovrapposizione:

1. Il device deve essere capace di "concurrent copy and execution", indagato con il campo `deviceOverlap` della struct `cudaDeviceProp` (tutti i device con compute capability $\geq$1.1 hanno questa capacità)
2. Il kernel e trasferimento dati devono appartenere a differenti non-default stream
3. La host memory coinvolta nel trasferimento deve essere pinned memory

Se si possono fare trasferimenti ed esecuzione dati parallelamente, potrebbe essere conveniente dividere un blocco di dati grande N in M sotto-gruppi da elaborare, permettendo di sovrapporre trasferimenti H2D (e poi D2H) con l'esecuzione di kernel. I trasferimenti sono gestiti tramite DMA; usiamo N/M stream.

Default Stream prima di CUDA 7: Il funzionamento è cambiato da CUDA 7, ma il default stream è utile quando la concorrenza non è cruciale al fine delle performance. Prima di CUDA 7, ogni device ha un defult stream usato per tutti i thread host, il quale porta a sincronizzazione implicita.

Sincronizzazione rispetto NULL-stream: Un NULL-stream blocca tutte le precedenti operazioni dell'host con la sola eccezione del lancio kernel. Anche se i non-NULL stream sono non-bloccanti rispetto all'host, possono essere sincroni o asincroni rispetto al NULL-stream.

Per questo gli stream non-NULL possono essere di due tipi:

- **Blocking** stream: lo stream NULL è bloccante
- **Non-blocking** stream: lo stream NULL non è bloccante

Gli stream creati usando `cudaStreamCreate` sono bloccanti: l'esecuzione di operazioni in questi stream vengono bloccate in attesa del completamento di operazioni dello stream NULL.

Il NULL stream è implicitamente definito e sincronizza con tutti gli altri stream bloccanti nello stesso contesto CUDA. In generale il NULL stream non si sovrappone con nessun altro stream bloccante.

Dal punto di vista dell'host ogni kernel è asincrono e non-bloccante, ma nell'esempio:

```
1 kernel_1<<<1, 1, 0, stream_1>>>();
2 kernel_2<<<1, 1>>>();
3 kernel_3<<<1, 1, 0, stream_2>>>();
```

`kernel_2` non può partire finché non termina `kernel_1` e similmente `kernel_3` con `kernel_2`.

CUDA runtime permette di definire il comportamento di uno stream non-NULL in relazione al NULL stream:

- NULL stream e non-NULL stream sono generalmente bloccanti tra loro
- `cudaStreamCreateWithFlags(*stream, flag)` permette di aggiungere la flag `cudaStreamNonBlocking` per rendere lo stream creato non bloccante

Inoltre, in generale, stream non-NULL sono tra loro non bloccanti.

Post Cuda 7: Prima di CUDA 7, ogni device ha un singolo default stream usato per tutti i thread dell'host che causano sincronizzazione implicita.

CUDA 7 ha introdotto la nuova opzione `per-thread default stream`, che ha due effetti:

1. Assegna a ogni thread dell'host il proprio default stream (comandi inviati al default stream da diversi thread dell'host possono eseguire concorrentemente)
2. I default stream sono stream regolari (comandi nel default stream possono eseguire in concorrenza con quelli in un non-default stream)

Per abilitare per-thread default stream compilare con `nvcc` command-line option `--default-stream per-thread`, o definire la macro per il preprocessore `#define CUDA_API_PER_THREAD_DEFAULT_STREAM`.

Priorità negli stream: Si possono creare stream con priorità (da CC 3.5). Una grid con più alta priorità può prelazionare il lavoro già in esecuzione con più bassa priorità.

Hanno effetto solo su kernel e non su data transfer. Priorità al di fuori del range vengono riportate automaticamente nel range.

Per creare e gestire uno stream con priorità si usano le funzioni:

```
cudaError_t cudaStreamCreateWithPriority(
    cudaStream_t* pStream, unsigned int flags, int priority);
```

Crea un nuovo stream con priorità intera e ritorna l'handle in `pStream`.

```
cudaError_t cudaDeviceGetStreamPriorityRange(
    int *leastPriority, int *greatestPriority);
```

Restituisce la minima e massima priorità del device (la più alta è la minima).

Host Functions (Callback): Si ha la possibilità di inserire una funzione host, senza introdurre sincronizzazione o interrompere il flusso dello stream. Questa funzione viene eseguita sull'host una volta che tutti i comandi forniti allo stream prima della chiamata sono stati eseguiti.

4.7.2 CUDA Event

Un **evento** è un **marker all'interno di uno stream** associato a un **punto del flusso di operazioni**. Serve per controllare se l'esecuzione di uno

stream ha raggiunto un dato punto o anche per la sincronizzazione inter-stream. Permettono controllo e sincronizzazione tra stream.

Può essere usato per due scopi base:

- **Sincronizzare l'esecuzione** di stream
- **Monitorare il progresso** del device

Le API CUDA forniscono funzioni che consentono di inserire eventi in qualsiasi punto dello stream. Oppure effettuare delle query per sapere se lo stream è stato completato.

Eventi sullo stream di default sincronizzano con tutte le precedenti operazioni su tutti gli stream.

Creazione:

```
cudaEvent_t event;
cudaError_t cudaEventCreate(cudaEvent_t* event);
```

Crea un nuovo evento di nome `event`.

Gli eventi nello stream zero vengono completati dopo che tutti i precedenti comandi in tutti gli stream sono stati completati. Ogni evento ha uno stato booleano: occorso/non occorso.

Per distruggerlo

```
cudaError_t cudaEventDestroy(cudaEvent_t event);
```

Questo completa il rilascio di risorse.

Per usarli, si registra un evento su uno stream:

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

Poi si possono usare altre funzioni per:

- sincronizzare l'host rispetto all'evento, bloccarlo finché non si verifica l'evento

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

- controllare l'avvenimento di un evento, senza bloccare

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

- far attendere uno stream sull'occorrenza dell'evento su un altro stream

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

Sincronizzazione Esplicita: CUDA runtime supporta diversi modi di sincronizzazione esplicita a livello di grid in un programma CUDA, si può sincronizzare rispetto

- al device
- ad uno stream
- a un evento all'interno di uno stream
- a diversi stream (tra loro), usando un evento

Si può bloccare l'host fino a che il device non ha completato i task precedenti:

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

Si può bloccare l'host fino a che tutte le operazioni in uno stream sono completate (`cudaStreamSynchronize`) oppure eseguire un test non-bloccante (`cudaStreamQuery`):

```
cudaError_t cudaStreamSynchronize(cudaStream_t stream);
cudaError_t cudaStreamQuery(cudaStream_t stream);
```

Un CUDA event può anche essere usato per sincronizzare host e device:

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

5 CUDA Python

L'idea è quella di avere codice python che sfrutta runtime CUDA per sfruttare il parallelismo esposto dalla GPU.

pyCUDA: L'idea di PyCUDA è di fare un "wrap" del codice C.

PyTorch: Uno dei più usati, spesso per il ML, orientato anche a non programmatori CUDA (generalmente chi lo usa lo fa in modo trasparente, senza sapere il funzionamento della GPU sottostante). L'idea è quella di replicare librerie all'interno di torch, usando CUDA in maniera trasparente (ad esempio numPy).

Rapids: Azienda che produce molteplici software, include alcune librerie come CuPy (NumPy e SciPy su GPU).

5.1 Numba for CPU

Per la CPU, Numba nasce con l'idea di accelerare tramite parallelismo su CPU. Numba è un package compilato Just-In-Time, non richiede uno step di compilazione dedicato, compila solo le funzioni che servono, usa LLVM per la traduzione a linguaggio macchina. Numba si integra con l'ecosistema python (NumPy, Pandas, …) e permette di usare solo codice Python per fare tutto.

Decoratori in Python: Un decoratore è un oggetto usato per modificare una funzione, metodo o classe per trasformarla. Per Numba, si usa il decoratore `@numba.jit` prima della funzione.

Ufuncs: Le funzioni universali sono un concetto introdotto da NumPy per indicare funzioni che operano elemento per elemento su un array NumPy. Permettono di eliminare la necessità di scrivere esplicitamente i ciclo `for` e sono (solitamente) compilate in C per efficienza.

Vectorize Decorator: Le operazioni vettorizzate eliminano i loop espliciti e permettono:

- maggiore velocità: non bisogna interpretare più volte la stessa operazione e le computazioni possono avvenire in parallelo

- minore overhead di memoria: minimizzando le variabili temporanea ed effettuando gli scambi in place, con conseguente miglior utilizzo della cache (e quindi performance)
- miglior uso del modello SIMD della CPU
- si può anche avere accelerazione GPU

Chiamare una funzione `@jit`:

- determinare il tipo degli argomenti forniti
- controllare se esiste una versione compilata a codice macchina e, nel caso, utilizzarla
- compilare, se necessario, una versione in linguaggio macchina ottimizzata
- convertire i parametri, questi vengono convertiti in valori compatibili con il codice macchina
- eseguire il codice ottimizzato, viene chiamata direttamente la funzione compilata
- convertire il risultato in una valore compatibile con Python

5.2 Numba for GPU

Anche qui si possono costruire le funzioni universali e vettorizzazione, si può dire che il target è CUDA, "dirottando" la compilazione verso CUDA tramite un semplice decoratore. Si tratta però di un modello piuttosto rigido.

Si possono invece definire kernel sulla GPU con `@cuda.jit`. Si possono lanciare kernel con `kernel[nBlocks, nThreads](args)`, supportano shared, pinned e local memory. Viene usato `nvcc` per compilare.

Dichiarazione dei kernel: Non possono tornare esplicitamente valori, quindi tutti i dati devono essere scritti su array passati alla funzione. Va dichiarata esplicitamente la thread hierarchy (numero di grid e blocchi). Una funzione può essere chiamata più volte, con diversi parametri e thread hierarchy, ma viene compilata una sola volta.

Thread Hierarchy: I valori della gerarchia possono essere ottenuti con:

- `numba.cuda.threadIdx`: indice del thread all'interno del blocco, da `0` a `numba.cuda.blockDim-1`
- `numba.cuda.blockDim`: dimensione del blocco di thread, per come dichiarato per l'istanza del kernel
- `numba.cuda.blockIdx`: indice del blocco all'interno della griglia di thread, da `0` a `numba.cuda.gridDim-1`
- `numba.cuda.gridDim`: dimensione della griglia di blocchi, numero totale di blocchi lanciati per l'istanza del kernel

Mentre le posizioni assolute si possono ottenere con:

- `numba.cuda.grid(ndim)`: torna la posizione assoluta del thread all'interno dell'intera griglia di blocchi; `ndim` deve corrispondere al numero di dimensioni dichiarate per l'istanza del kernel, se `i=1` torna un solo interno, altrimenti una tupla contenente quel numero di interi
- `numba.cuda.gridsize(ndim)`: torna la dimensione assoluta ("forma") in thread dell'intera griglia di blocchi

Vale la pena ricordare che in Python il tipo di default è `f64` (o qualcosa di simile, idk), quindi senza specificare il tipo anche la libreria userà quello come tipo.

Magari non è richiesta tale precisione (in genere in CUDA non si lavora con f64), quindi bisogna specificare esplicitamente il tipo. Ad esempio usando `.astype(np.float32)`.

Funzioni device: Si possono scrivere funzioni richiamabili solo dall'interno del device, con il decoratore `@cuda.jit(device=True)`, ad esempio:

```
1 cuda.jit(device=True)
2 def a_device_function(a, b):
3     return a + b
```

5.3 Gestione della memoria

Numba trasferisce automaticamente gli array NumPy quando viene invocato il kernel, ma lo può fare solo in modo "conservativo", quindi trasferisce sempre la memoria device back to the host quando finisce. Si possono gestire manualmente i trasferimenti per evitare di passare inutilmente array read-only.

Api per allocare e trasferire:

- Alloca un ndarray device (similmente a `numpy.empty()`)

```
numba.cuda.device_array(shape, dtype=...,
    strides=..., stream=0)
```

- Chiama `device_array()` con informazioni dall'array

```
numba.cuda.device_array_like(ary, stream=0)
```

- Alloca e trasferisce un ndarray numpy o uno scalare strutturato al device

```
numba.cuda.to_device(obj, stream=0, copy=True, to=None)
```

Pinned e mapped memory: La memoria pinned a mapped si può gestire tramite:

- Un context manager per pinnare temporaneamente una sequenza di ndarray host

```
numba.cuda.pinned(*arylist)
```

- Alloca un ndarray con un buffer pinnato (pagelocked)

```
numba.cuda.pinned_array(shape, dtype=...,
    strides=..., order='C')
```

- Chiama un array pinned con le informazioni dall'array

```
numba.cuda.pinned_array_like(ary)
```

- Un context manager per mappare temporaneamente una sequenza di ndarray host

```
numba.cuda.mapped(*arylist, **kws)
```

Deallocazione: In generale, la deallocazione è gestita in modo automatico, tracciata per-contex. Nei casi di gestione asincrona la deallocazione automatica potrebbe causare problemi, quindi `numba.cuda.defer_cleanup()` permette di fermare la deallocazione (usata tramite blocco with).

Esempio:

```
with defer_cleanup():
    # all cleanup is deferred in here
```

```
    do_speed_critical_code()
# cleanup can occur here
```

Static shared memory:

- Alloca un array shared della dimensione e tipo specificato; la funzione deve essere chiamata dall'interno del device

```
numba.cuda.shared.array(shape, type)
```

`shape` può essere un intero o una tupla di interi, rappresenta le dimensioni del'array; deve essere una espressione semplice

- Sincronizza tutti i thread all'interno dello stesso blocco

```
numba.cuda.syncthreads()
```

Dynamic shared memory: Per usare la memoria shared dinamica, nel kernel va dichiarato un array shared di dimensione `0`; esempio:

```
@cuda.jit
def kernel_func(x):
    dyn_arr = cuda.shared.array(0, dtype=np.float32)
```

Durante la chiamata a kernel va specificata la dimensione in byte della shared memory:

```
kernel_func[32, 32, 0, 128](x)
```

L'ultimo parametro è la shared memory.

Tutta la memoria dinamica diventa un alias allo stesso array; dichiarando più array dinamici quello che succede sarà che ci saranno solamente più puntatori ad uno stesso array, con interpretazioni differenti (stessi dati).

Una soluzione al problema può essere invertire un array, raddoppiando la dimensione totale durante la chiamata:

```
f32_arr = cuda.shared.array(0, dtype=np.float32)
i32_arr = cuda.shared.array(0, dtype=np.int32)[1:]
```

In questo modo uno viene letto dall'inizio, uno dal fondo. Servono visioni disgiunte degli array.

5.4 Atomic Operations

Si possono usare operazioni atomiche per evitare race conditions, le quali possono accadere nei casi di **read-after-write** (un thread prova a leggere una cella di memoria nello stesso momento in cui un altro sta scrivendo) oppure **write-after-write** (due thread provano a scrivere nello stesso momento).

Il namespace per le operazioni atomiche è la classe `numba.cuda.atomic`. Ad esempio, `add(ary, idx, val)` svolge l'operazione atomica `ary[idx] += val`; supportata su `i32`, `f32` e `f64`.

5.5 Streams

Gli **stream** possono essere passati alle funzioni, vengono usati durante la configurazione per il lancio del kernel, in modo che le operazioni siano eseguite in maniera asincrona.

La funzione

```
numba.cuda.stream()
```

crea uno stream CUDA, il quale rappresenta la coda dei comandi per il dispositivo.

La funzione

```
numba.cuda.default_stream()
```

restituisce lo stream di default. Solitamente, il default stream si comporta in maniera legacy o per-thread on base alla API CUDA in uso.

La funzione

```
Stream.synchronize()
```

permette la sincronizzazione all'interno dello stream, i.e., aspetta che tutti i comandi all'interno dello stream vengano eseguiti.

6 Pattern Paralleli

6.1 Scan (Prefix-sum)

L'operazione **parallel scan** (anche chiamata **prefix-sum**) considera un operatore binario associativo $\oplus$ e un array di n elementi $a = [a_0, a_1, \ldots, a_{n-1}]$ e restituisce l'array $b = [a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus \cdots \oplus a_{n-1})]$. In sintesi:

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

Questo pattern viene utilizzato come "blocco" all'interno di diverse applicazioni, come alcuni algoritmi di sorting, operazioni su alberi, distribuzioni di probabilità cumulative, …

La versione sequenziale è molto semplice:

```
1 void prefix_sum(float *a, float *b, int n) {
2     b[0] = a[0];
3     for (int i = 1; i < n; i++)
4         b[i] = b[i-1] + a[i];
5 }
```

Ed ha complessità:

- di step $O(n)$
- di operazioni svolte $O(n)$

Quindi, per un array di n elementi servono un numero lineare di operazioni: efficiente.

6.1.1 Implementazione di Horn per GPU

Avendo a disposizione molti processori in parallelo, si vuole ridurre il tempo da $O(n)$ a $O(\log n)$, anche a costo di qualche operazioni in più.

Horn usa un approccio iterativo in $d = \lceil \log_2 n \rceil$ passi. A ogni passo i (da 1 a d):

1. Calcolo l'offset
$$\Delta = 2^{i-1}$$
2. Per ogni elemento k dell'array (in parallelo)

- se $k \geq \Delta$

$$x[k] \leftarrow x[k] + x[k - \Delta]$$

- se $k < \Delta$

$$x[k] \leftarrow x[k]$$

A livello pratico:

- Vengono caricati i dati in memoria shared

```
smem[tid] = input[tid];
__syncthreads();
```

- Loop per effettuare l'operazione: ogni thread somma il valore della cella di smem relativa con quella a distanza determinata dall'offset

```
for (int d = 1; d < BLOCK_SIZE; d *= 2) {
    if (tid >= d)
        smem[tid] += smem[tid - d];
        __syncthreads();
}
```

- Scrivere i risultati in memoria globale

Scan parallelo per lunghezze arbitrarie: Come affrontare grandi quantità di dati?

- Partizionare i dati in blocchi che possono essere memorizzati all'interno della memoria shared
- Sommare i singoli blocchi
- Memorizzare la somma dei singoli blocchi in memoria ausiliaria
- Ogni elemento della memoria ausiliaria contiene la somma del blocco precedente
- Lanciare un kernel per la somma dei risultati parziali con gli elementi dei blocchi

Analisi efficienza: Per la versione parallela su un solo blocco in shared memory: tutti i thread iterano al più fino a $\log n$, con $n =$ dimensione del blocco. A ogni passo il numero di thread che non effettua operazioni è pari allo stride.

Complessità:

- step complexity: $\Theta(\log n)$
- work efficiency: $\Theta(n \log n)$

Quindi peggiore del caso sequenziale: poco efficiente.

6.1.2 Strategia work efficient

Per evitare il fattore $\log_2 n$ extra dell'algoritmo naive, si usano **alberi bilanciati**: viene costruito un binary tree bilanciato sui dati in input e viene "passato" da e verso la radice per computare le somme prefisse.

Un albero binario con n foglie ha $d = \log_2 n$ livelli, ogni livello d ha 2^d nodi. Viene effettuata una somma per nodo, quindi in totale $O(n)$.

Non si memorizza l'albero come struttura dati effettiva, è solo un concetto usato per determinare cosa devono fare i thread a ciascun passaggio.

L'algoritmo consiste di due fasi:

- fase di reduce o **up-sweep**: per costruire le somme parziali fino alla radice
- **down-sweep**: per distribuire i prefissi corretti alle foglie

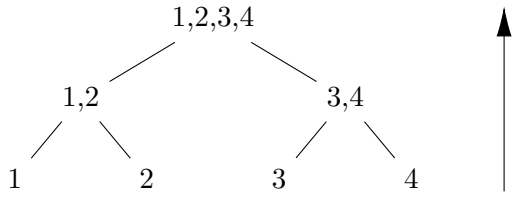
Gli n elementi in input vengono rappresentati come foglie di un albero binario pieno di altezza d.

Fase di up-sweep: Si scorrono i livelli da foglie a radice: a livello i, ogni nodo j, calcola

$$T[i][j] = T[i-1][2j] + T[i-1][2j+1]$$

dove $T[0][k] = x[k]$ (le foglie corrispondono al vettore di input).

Questo è come dire che ogni nodo a livello i sarà la somma dei nodi figlio. Alla fine, la radice sarà la sommatoria di tutti i valori.

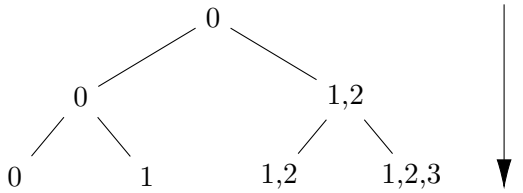


Fase di down-sweep: Prima di "scendere", si setta la radice a 0

$$T[d][0] \leftarrow 0$$

Poi si scorrono i livelli, dalla radice verso le foglie (i da d a 0): ogni nodo j a livello i ha due figli $2j$ e $2j+1$ a livello $i-1$, quindi

$$\begin{aligned} T[i-1][2j+1] &\leftarrow T[i][j] + T[i-1][2j] \\ T[i-1][2j] &\leftarrow T[i][j] \end{aligned}$$



Sostanzialmente:

- il figlio destro prende la somma dei valori di padre e figlio sinistro (prima che venga aggiornato)
- il figlio sinistro prende il valore del padre

In totale sono:

- $2 \log n$ passi: $\log n$ per fase
- $2(n-1)$ operazioni: una operazione per nodo interno dell'albero $(n-1)$ per ogni fase

NB: L'ultimo elemento viene "perso" in quanto si aggiunge uno zero per "allineare" i valori. Prima dell'esecuzione bisogna ricordare l'elemento finale e aggiungerlo nuovamente alla fine (tempo costante).

6.2 Graph Coloring

Grafi k-colorabili: Una k-colorazione dei vertici di un grafo G consiste nell'assegnazione di k colori $1, 2, \ldots, k$ ai vertici di G in modo tale che due vertici adiacenti non abbiano lo stesso colore.

Diremo **k-colorabile** ogni grafo G che ammette una k colorazione.

Dato k, trovare una k colorazione di G è $\mathcal{NP}$-Hard.

6.2.1 Colorazione Greedy sequenziale

Dato un grafo $G = (V, E)$ e k colori, si può avere un semplice algoritmo greedy:

1. Considero i vertici in un ordine specifico

$$v_1, \ldots, v_n$$

2. Assegno a v_i il più piccolo colore disponibile non utilizzato dai vicini, aggiungendo un nuovo colore se necessario

In questo modo la qualità della colorazione risultante dipende dall'ordinamento prescelto. Esiste un ordinamento che conduce a una colorazione con il numero ottimale (difficile da trovare).

6.2.2 Jones-Plassman Coloring

Si tratta di un algoritmo approssimato per ambienti distribuiti. Non garantisce una soluzione ottimale, ma la qualità è simile a quella di algoritmi sequenziali greedy.

Dato un grafo $G = (V, E)$, l'idea è:

- Ogni vertice $v \in V$ riceve una priorità casuale
- Ogni vertice può decidere il proprio colore quando ha priorità maggiore di tutti i suoi vicini non ancora colorati
- Viene iterato lo step di decisione, a ogni round colorando i vertici che soddisfano la condizione, rimuovendoli dai round successivi, fino a completamento

All'interno di ogni round, il passo di selezione e colorazione può essere eseguito in parallelo.

Per un grafo sparso e con distribuzioni casuali dei pesi converge con alta probabilità per $O(\log |V|)$.

Pseudocodice:

```
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};
```

6.3 Insieme Indipendente Massimale MIS

Dato un grafo $G = (V, E)$, un **Indipendent Set** IS è un sottoinsieme $U \subseteq V$ dei nodi del grafo tale che non ci siano nodi in U adiacenti $\forall u, v \in U$, $(u, v) \notin E$.

Un IS è *massimale* se nessun nodo può essere aggiunto a U senza violare l'IS.

Noi cerchiamo l'IS massimale di cardinalità massima.

Trovare un IS massimale è facile (su processore singolo), trovare l'IS massimo è $\mathcal{NP}$-Hard.

6.3.1 Algoritmo Greedy sequenziale

Un semplice algoritmo greedy è, partendo dall'insieme soluzione S vuoto: dato un ordinamento sui vertici, ogni vertice viene aggiunto alla soluzione se non ha vicini già all'interno di S.

6.3.2 Parallel Randomized MIS Algorithm

L'algoritmo di Luby per il MIS permette di trovare un IS massimale (non massimo, non c'è nessuna garanzia sulla cardinalità della soluzione).

Schema dell'algoritmo:

1. Inizializzazione
    - L'insieme soluzione S viene segnato come vuoto

- In un altro insieme C, si tiene traccia dei vertici "attivi", all'inizio $C = V$

2. Iterazione, finché sono presenti vertici attivi $C \neq \emptyset$ ripete
   - Ogni vertice attivo $v \in C$ sceglie un valore casuale r_v
   - Un vertice v si propone per la soluzione S se

     $$r_v < r_u \quad \forall u \in N(v) \cap C$$

     dove $N(v)$ rappresenta i vicini di v; questo vuol dire che il vertice entra nella soluzione se ha la priorità minima tra i suoi vicini.
   - Tutti i vertici che soddisfano il criterio vengono aggiunti a S
   - Da C vengono rimossi tutti i vertici selezionati e tutti i loro vicini (per garantire l'indipendenza)
3. Termina quando non rimane alcun vertice attivo $C = \emptyset$: la soluzione S è un MIS

L'algoritmo molto probabilmente finisce in $O(\log |V|)$ round. A ogni iterazione, una frazione dei nodi viene eliminata, portando a una terminazione "rapida".

6.4 Sorting

La complessità computazionale si basa sulla valutazione del numero di operazioni elementari necessarie (confronti, scambi); si misura come funzione del numero n di elementi della sequenza.

Algoritmi sequenziali possono raggiungere complessità di $O(n \log n)$ (lower e upper bound provato), per gli algoritmi paralleli la complessità desiderata sarebbe $O(\log n)$.

6.4.1 Bitonic MergeSort

Si tratta di un algoritmo di ordinamento parallelo basato su sorting network per sequenze bitoniche.

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

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

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

Bitonic split: Se la sequenza $s = \{a_0, \ldots, a_{n-1}\}$ è bitonica e n è una potenza di 2, allora per le sequenze

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

Vale che

- s_1 e s_2 sono ancora sequenze bitoniche
- $a_i < a_j$, $\forall a_i \in s_1$, $\forall a_j \in s_2$, i.e., tutti gli elementi di s_1 sono minori degli elementi di s_2

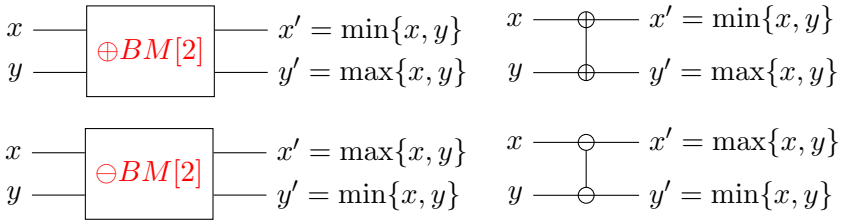
Si può quindi applicare ricorsivamente questa procedura alla sequenza dati per ordinarla.

Dopo $\log n - 1$ passi, ogni sequenza bitonica avrà solo 2 elementi: ordinabili banalmente.

Bitonic Merging Networks: 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 $n = 2$ banalmente:

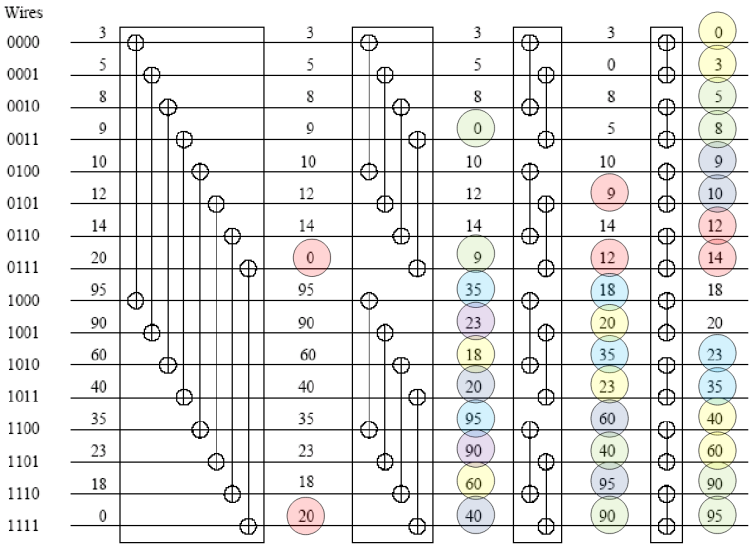


Sulla destra una semplificazione in quanto reti con $n = 2$ sono dei comparatori.

Per una rete con n fili, $\oplus BM[n]$, (quindi sequenza bitonica lunga n in ingresso e ordinata in uscita), servono:

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

Ogni colonna ha comparatori a distanza pari a metà della colonna precedente (per $n = 16$, prima colonna a distanza 8, seconda a distanza 4, ..., fino a distanza 1 in $\log_2 n$ passi). Esempio per rete con $n = 16$:



Costruzione di sequenza bitonica: Con input una sequenza arbitraria, vogliamo ottenere una sequenza bitonica in output. Si può fare alternando i segni dei BM.

L'idea è quella di usare una rete con $(\log_2 n) - 1$ colonne: ogni i-esima colonna:

- prende in input una sequenza bitonica di lunghezza 2^i, a partire da $i = 1$, una sequenza lunga 2 è banalmente bitonica
- si costruisce una sequenza bitonica lunga 2^{i+1} alternando $\oplus BM[2^i]$ con $\ominus BM[2^i]$, da fornire in input alla colonna successiva

L'ultima colonna avrà una $\oplus BM[n/2]$ e una $\ominus BM[n/2]$, per creare una sequenza bitonica lunga n.

6.4.2 Ordinamento bitonico

Si può ordinare una sequenza qualsiasi avendo:

- $(\log n) - 1$ step per costruire una sequenza bitonica a partire dall'input (come visto prima)
- aggiungere un comparatore $\oplus BM[n]$ alla fine per ordinare la sequenza

Complessità: Per una sequenza lunga n, tutte le $\log n$ colonne richiedono:

$$\begin{aligned} T(n) &= T(n/2) + \log(n) \\ &= \log(n) + \log(n/2) + \cdots + 1 \\ &= \sum_{i=0}^{\log n} i \\ &= \frac{\log n \left((\log n) + 1\right)}{2} \\ &= \Theta(\log^2 n) \end{aligned}$$