GPU Computing

Massimo Perego

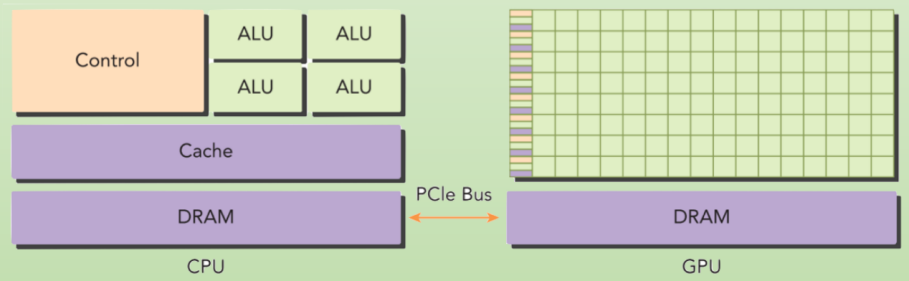
Contents

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 (possono esserci fino a decine di migliaia di core), 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, tutto parte comunque dalla CPU (Host) la quale effettuerà richieste alla GPU (Device), diventa un coprocessore. Viene separata parte sequenziale dell'applicazione e di controllo che va sulla CPU mentre la parte a maggior intensità computazionale va sulla GPU.
La GPU non è una piattaforma standalone, ma è un coprocessore che opera congiuntamente alla CPU, comunicando tamite 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 esporle 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 SMD permettendo flussi individuali per ogni thread, con il costo relativo a gestire la decisione locale sui thread (program counter e registri).

1.1 Architettura Nvidia

Streaming Multiprocessor SM: Le GPU sono costituite di array di SM, 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, eseguire la stessa istruzione allo stesso tempo.
Questo è il modello iniziale, nel tempo si è evoluto con cose come una maggiore gerarchia di cache e altri core dedicati ad applicazioni specifiche, come i tensor core per il calcolo matriciale. Ogni CUDA core ha i suoi registri e

unità di calcolo (FP e INT).

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 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 permette 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; può 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 di dati in una sola transizione. La popolarità 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 del computer parallelo, definendo come si possa codificare un algoritmo

- Comprende la **semantica** del linguaggio di programmazione, librerie, compilatore, tool di profiling
- Dice di che **tipo** sono le c**omputazioni parallele** (instruction level, procedural level o parallel loops)
- Permette di dare **specifiche implicite** o **esplicite** (da parte 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 alternali 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. Vi sono meccanismi di sincronizzazione tra thread per evitare race conditon (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 concorrentemente. 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).

L'host ha una serie di processi in esecuzione e controlla tutto, lancio delle funzioni kernel sul device compreso. Con "kernel" si intende programma sequenziale eseguito dalla GPU.
Ogni kernel è asincrono, la CPU lancia il kernel e passa a dopo, almeno finché non è necessaria la 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:

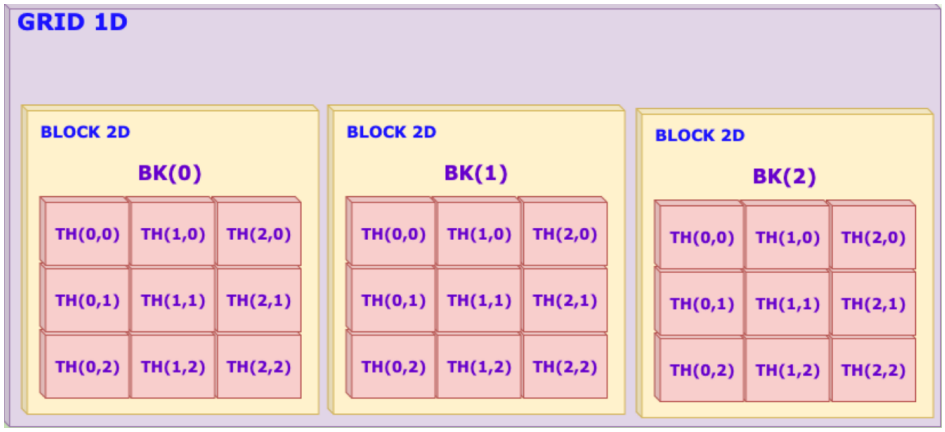
1. Setup dei dati su host (CPU-accessible memory)
2. Alloca memoria per i dati sulla GPU
3. Copia i dati da host a GPU
4. Alloca memoria per output su host
5. Alloca memoria per output su GPU
6. Lancia il kernel su GPU
7. Copia output da GPU a host
8. Libera le memorie

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 essere 1D, 2D o 3D. 9 combinazioni ma di solito 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, ogni blocco ha thread 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.

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.

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`.

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 scalare a più 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.

3.2 Warp

Ogni thread vede:

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

Single Instruction Multiple Thread; l'architettura è basata sul **warp**, (tradotto in "trama" nella 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 e può seguire cammini distinti di esecuzione delle istruzioni (parallelismo a livello thread, 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 un `if`), la GPU le eseguirà sequenzialmente al posto che in parallelo, disabilitando i thread inattivi. Questa è una **divergenza** e riduce l'efficienza, a volte anche significativamente.

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 di thread che possono essere attivi in maniera concorrente 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

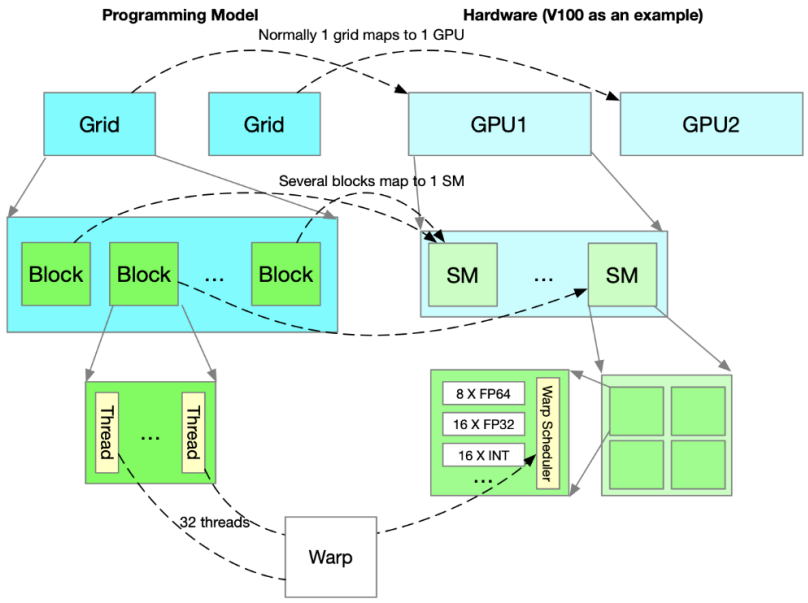
- dalla quantità di registri e di shared memory usata dal kernel
- dalla quantità di registri e shared memory resi disponibili dallo SM

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 multiprocessor, vogliamo fare in modo di avere l'utilizzo maggiore possibile.

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 ad ogni ciclo di clock. Si ha così latency hiding intercambiando 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

  ```
  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, deadlock.

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

3.3 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 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 (es. 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)

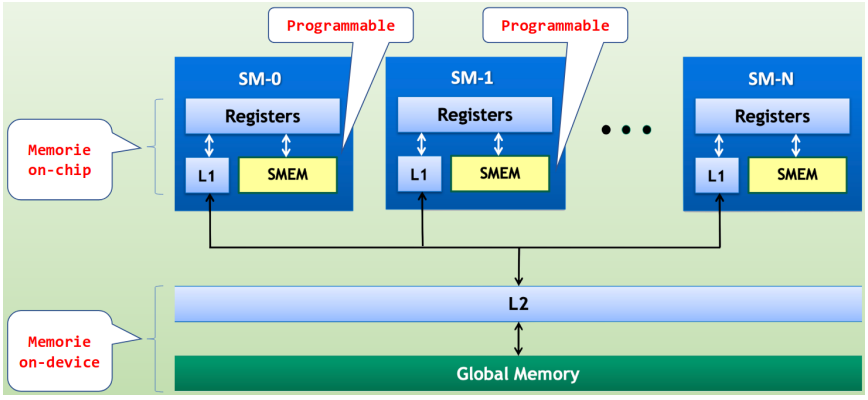
La cache L1 è presente all'interno di ogni SM; in alcune architetture (Fermi e successive) la dimensione può essere configurata, con una porzione assegnabile a memoria condivisa. Capacità limitata ma permette di sfruttare località dei dati.
La cache L2 ha dimensione maggiore ed è condivisa tra tutti 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).

3.4.1 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 ed inaccessibile ad altri blocchi. Viene dichiarata tramite `__shared__`.

La SMEM è suddivisa in moduli della stessa ambiezza, 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 ($\leq$ 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 non è in generale un problema, anche in caso di bank conflict; lo scheduler passa ad 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 lo scheduling efficace, le prestazioni (anche in presenza di conflitti a livello smem) sono molto migliori rispetto alla cache L2 o global memory

3.4.2 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.
Dinamicamente si possono allocare solo array 1D.

Se la dimensione non è nota compile time, è possibile dichiarare una variabile adimensionale con la keyword `extern`.

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

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

3.4.3 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 Global Memory

Nei computer moderni esiste una gerarchia di memoria 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 sul multiprocessore (il compilatore usa un'euristica per ottimizzare questo parametro). Register spilling: se si usano più registri dei 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, ma 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:

```
cudaError_t cudaMemcpyToSymbol(const void* symbol, 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 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 poten-

zialmente 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 ad 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

```
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` restitusce `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 a 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

3.6 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:

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

E per deallocarla:

```
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.

3.7 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.

3.8 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 compute capability

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 cache dipende da compute capability 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 opreazioni effettive in memoria coalescenti.

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);
```

Indaga il numero di device disponibili sul sistema e restituisce 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 dell'applicazione 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 aumentando il numero di blocchi concorrenti.

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.

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 (percorsi di codice differenti all'interno del medesimo warp).

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 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 sian *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, la GPU valuta 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, permette 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, 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 figli senza nessuna interazione 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. poplare 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)

4.6.3 cuFFT - Fast Fourier Transform

La libreria cuFFT fornisce una implementazione della `Fast Fourier Transform` `FFT` (segnali dal dominio del tempo a frequenze e relativa inversa). L'input della FFT è un sampling di un segnale. La librerie permette trasformate 1D, 2D o 3D. Si basa sugli algoritmi per FFT di `Cooley-Tukey` e `Bluestein`. Permette una esecuzione asincrona con valori che possono essere `real, complex, float,` double.