1. Che cos'è un'architettura eterogenea? Perchè l'abbiamo esplorata?

Un computer con un'architettura eterogenea è un tipo di computer che integra diversi tipi di processori o core di elaborazione al suo interno combinandone i vantaggi (e bypassando i svantaggi, le cose in cui non sono bravo le faccio fare ad un altro più bravo di me). Il computer, avendo a disposizione unità di calcolo eterogenee, può far eseguire su ognuna di esse il task per cui sono più portate velocizzando in questo modo i calcoli e ottenendo flessibilità.

Ad esempio: un computer con GPU e CPU, può eseguire sulla CPU i task sequenziali sfruttando la sua minore latenza delle operazioni, mentre può eseguire sulla GPU i task altamente paralleli sfruttandone l'alto throughput. A questo punto è il programmatore che decide dove far eseguire cosa per ottenere le migliori prestazioni.

MODULO 1

SIMD

2. Descrivi in generale le caratteristiche del paradigma SIMD

SIMD è un paradigma di elaborazione di istruzioni in cui una singola istruzione SIMD elabora multipli dati eseguendo su di essi la medesima operazione. Questi dati sono memorizzati in **registri speciali detti estesi** della CPU pensabili a come un array di registri normali.

Questo paradigma:

- Incrementa il parallelismo agendo a livello dei dati
- Può essere utilizzato in combinazione con altre strategie di parallelismo (Pipelining, superscalarità, multithreading, ...)
- Richiede un numero di modifiche all'hardware limitato rispetto a SISD con impatto modesto in termini di maggiori risorse utilizzate
- Integrazione di ALU/unità di calcolo addizionali (una per ogni lane)
- Integrazione dei registri estesi
- Integrazione delle nuove istruzioni nel decoder

3. Quali sono delle tipiche istruzioni supportate da un estensione SIMD?

In generale quelle classiche di un processore, con in aggiunta istruzioni per la manipolazione dei dati all'interno dei registri estesi:

- Load e store dei registri estesi da/verso la memoria principale di blocchi di dati (Condizione necessaria per l'efficacia del paradigma SIMD è che i registri estesi possano essere letti/scritti agevolmente)
- somme, sottrazioni, moltiplicazioni, etc tra registri estesi
- operazioni logiche e di confronto bitwise tra registri estesi
- manipolazione e riarrangiamento dei dati intra e inter registro esteso (blend, packing, unpacking, ...)
- consentire operazioni con saturazione e operazioni molto comuni come SAD (Sum of Absolute Differences) o FMA (Fused Multiply Add)

4. Che cos'è una operazione con saturazione e come mai è utile?

Una operazione con saturazione è una operazione che non va in overflow/underflow ma "satura" al valore massimo/minimo. è utile quando serve solo sapere se un valore è molto grande o molto piccolo.

5. Parlami del branching in SIMD

Con il paradigma SIMD, il branching condizionata al valore di un registro esteso non è supportato in quanto non applicabile (se 3 lane su 8 rispettano la condizione di branching salto o non salto? non ha senso...).

Se si desidera compiere azioni differenti sui dati all'interno di un registro esteso, si utilizzano apposite maschere prodotte da apposite istruzioni di confronto SIMD e operazioni logiche. Queste maschere filtrano le lane che rispettano le "condizioni di branch" e a cui applicare le operazioni desiderate.

Dealing with real numbers

6. Descrivi le tecniche principali di codifica dei reali?

Esistono principalmente due codifiche:

- Fixed point:
  - Codifica parte intera e parte decimale di un reale con un numero costante di bit, mantenendo la virgola in una posizione fissa (fixed point)
  - bit di segno per dati signed
  - ha un range dinamico molto limitato rispetto a floating point
  - tuttavia, mantiene una accuratezza (e quindi un errore) costante su tutto il range dei valori rappresentabili
- Floating point:
  - Codifica i reali nella loro rappresentazione scientifica. Composto da:
    - Un certo numero di bit per la parte decimale chiamata **mantissa** (la parte intera è sempre 1 dato che siamo in binario e utilizziamo notazione scientifica)
    - Un certo numero di bit per l'esponente (da qui floating point, la virgola si sposta in base al valore dell'esponente)
    - un bit di segno
  - ha un range dinamico molto ampio; rappresenta sia valori molto piccoli che molto grandi
  - tuttavia, l'accuratezza varia al variare dell'esponente
    - all'aumentare dell'esponente, aumenta in maniera proporzionale anche il valore dell'ULP (Unit in the Last Place) della mantissa. Questo significa che per i numeri grandi in valore assoluto, si ha una accuratezza bassa (è possibile anche che vengano saltati dei valori interi)
    - al diminuire dell'esponente invece accade il contrario. Per numeri piccoli in valore assoluto, si ha una accurattezza tanto più precisa tanto quanto è basso il valore dell'esponente (questo causa la necessità di valori subnormali).
    - **TRUCCO**: Un modo di pensarla è che stiamo dividendo sempre nello stesso numero di bin (definito dal numero di bit della mantissa) un intervallo di valori che varia in grandezza al variare dell'esponente

Floating-point è senza discussione il formato più usato a causa del suo largo range dinamico. Tuttavia, fixed-point trova un caso d'uso nel rimpiazzare quelle che sarebbero costose operazioni floating-point, con

operazioni tra interi.

NB: entrambi i formati sono solo una astrazione approssimata dei reali in quanto con un numero finito di bit è ovvio che non posso rappresentare l'infinito non numerabile dei reali.

7. Che tipi di errori sono possibili con floating-point?

- **Errori di codifica/approssimazione**: tutti quei valori reali con non ricadono in uno dei '*bin*' possibili definiti dai bit di mantissa ed esponente, vengono per forza approssimati al bin più vicino.
    - Aumentando il numero di bit della mantissa avremo a disposizione un ULP con un valore sempre più 'fine' e quindi una accuratezza maggiore; tuttavia, rimaranno sempre dei valori che non saranno esprimibili come potenze di due e che quindi dovranno per forza venire approssimati (i.e. 1,21).
    - Per capire "quale bin è il più vicino" è necessario usare durante la codifica un ulteriore bit di rounding; in questo modo, l'errore massimo sarà val(ULP)/2
    - questi errori di approssimazione si propagano poi nelle operazioni aritmetiche
    - (quanto detto sopra vale in realtà anche per fixed-point)
- **Trocamenti durante le Somme**:
    - Il risultato di una somma tra due numeri a n bit, ha n+1 bit. Questo bit aggiuntivo va troncato e causa un'ulteriore approssimazione.
    - **Allineamento degli esponenti durante le somme**: sommare un numero molto grande con un numero molto piccolo causa dei troncamenti nei due addendi in quando è necessario allineare i due esponenti. è anche possibile, nel caso la differenza tra i due valori sia molto grande, che uno dei due addendi sparisca.
- **Troncamenti durante le Moltiplicazioni**:
    - Il risultato di un prodotto tra due numeri a n bit, ha 2*n bit. Questo bit aggiuntivi vanno troncati, questo è causa di un'ulteriore approssimazione.

In generale, a causa di questi errori bisogna stare attenti a fare confronti di uguaglianza tra float, più safe usare un *epsilon*. Inoltre proprità associativa e distributiva di somme e prodotti non sono realmente valide; questo può causare errori difficili da debuggare.

8. Parlami degli altri formati per i reali oltre a quelli definiti da IEEE? Come mai si è sentito il bisogno di crearne di altri?

In molti casi d'uso (i.e. training di modelli di AI), non è necessaria una alta accuratezza ma si ha bisogno di un largo range dinamico. Appurato cio, diventerebbe allora conveniente compattare la codifica dei float in formati più piccoli, che sacrificano molti bit di mantissa e pochi (se non zero) bit di esponente. Questo porterebbe a tre principali vantaggi:

- **Consumo energetico ridotto**: il consumo di energia per operazioni aritmetiche con float scala con (circa) il quadrato della lungehzza della mantissa
- **Computazioni pià veloci**: con un formato che usa meno bit ho che
    - posso trasferire più dati con la stessa bandwidth riducendo lo stallo delle unità di calcolo per operazioni memory bound (Molta sinergia con il calcolo parallelo).
        - detta in altri termini, aumenta la mia Intensità Aritmetica il che mi sposta più a destra nel diagramma di roofline. Questo mi fa ottenere più FLOPS nel caso in cui fossi stato memory bound.
    - nella stessa cache ci stanno più dati

- **Memory footprint ridotto a runtime**

Un esempio che può sembrare estremo, ma sorprendentemente efficace, di questi formati è: ***minifloat E5M2***. Con 128 bin (8-1 bit togliendo il bit di segno) ha un range dinamico pari a [-(1+3/4)2^15, +(1+3/4)2^15]

9. Che altre tecniche conosci per ridurre il memory footprint di un programma?

Abbiamo anche parlato di *weight sharing (paletizzazione)*. Questa tecnica consiste in:

- quantizzazione dei dati in un unico valore (media dell'intervallo quantizzato, centroide)
- salvataggio dei centroidi in una LUT
- a runtime, utilizzo gli indici della LUT (pochi bit per indice) piuttosto che i valori reali che utilizzerebbero più bit

10. Qual'è il caso d'uso di fixed-point?

Fixed-point permette di effettuare calcoli con dati reali utilizzando solo operazioni tra interi. L'idea consiste in:

1. trasformare i dati, inizialmente in fixed-point, in formato intero mediante uno shift pari al numero di cifre decimali (molto facile con fixed-point)
2. effettuare le operazioni desiderate con i valori interi
3. recuperare il risultato reale facendo lo shift inverso alla fine

Con questa tecnica si hanno due vantaggi:

- Speedup nel calcolo: operazioni tra interi richiedono un numero di clock minore rispetto ad operazioni tra float
- Non c'è bisogno di unità di calcolo FP: quest'ultime potrebbero essere non disponibili (microcontrollori) oppure troppo costose in termini di spazio/potenza consumata(FPGA)

NOTA: La tecnica del passare al mondo degli interi con una moltiplicazione per un fattore appropriato in realtà si può applicare anche nel contesto floating-point. In questo caso però, il range dinamico molto largo porta ad avere dei fattori moltiplicativi enormi se si moltiplica, ad esempio, un float piccolo (in valore assoluto) per uno molto grande. Questo porta ad ottenere dei valori interi che hanno bisogno di un numero di bit enorme per essere rappresentati e fa perdere i vantaggi del passare alla rappresentazione intera.

11. Quando è necessario utilizzare floating point?

Utilizzare floating point è necessaria quando si ha bisogno del suo elevato range dinamico. In generale, se si riesce ad evitare (ad esempio utilizzando la tecnica possibile con fixed-point) è meglio in quanto è costoso, sia in termini di performance, che di energia consumata.

Se si ha la necessità di usare floating-point, è importante utilizzare il formato più compatto possibile. Se invece floating point non è strettamente necessario, potrebbe essere più intelligente utilizzare fixed-point e passare al mondo degli interi.

MODULO 2

Modello di programmazione CUDA

11. Che cos'è il modello di programmazione CUDA?

Il Modello di Programmazione definisce la struttura e le regole per sviluppare applicazioni parallele su GPU. In particolare definisce:

- Gerarchia di Thread: organizza l'esecuzione parallela in thread, blocchi e griglie, ottimizzando la scalabilità su diverse GPU.
- Gerarchia di Memoria: Offre tipi di memoria (globale, condivisa, locale, costante, texture) con diverse prestazioni e scopi, per ottimizzare l'accesso ai dati.
- API: Fornisce funzioni e librerie per gestire l'esecuzione del kernel, il trasferimento dei dati e altre operazioni essenziali.

12. Che cos'è un thread CUDA

Un thread CUDA rappresenta un'unità di esecuzione elementare nella GPU. Ogni thread CUDA si occupa di un piccolo pezzo del problema complessivo, eseguendo calcoli su un sottoinsieme di dati in maniera sequenziale. Il parallelismo si ottiene coprendo l'intero spazio dei dati del problema, lanciando contemporaneamnete migliaia di thread (SIMT). Ogni thread esegue lo stesso codice del kernel ma opera su dati diversi, determinati dai suoi identificatori univoci (threadIdx,blockIdx).

13. Qual'è il tipico workflow in CUDA?

1. Inizializzazione e Allocazione Memoria su CPU e GPU
2. Trasferimento Dati (Host → Device)
3. Esecuzione (asincrona) del Kernel (Device)
4. (eventuali calcoli lato host)
5. Recupero Risultati (Device → Host)
6. Post-elaborazione (Host)
7. Liberazione Risorse

14. Come vengono organizzati i thread in CUDA?

Abbiamo due livelli di organizzazione:

- i thread vengono raggruppati in blocchi
- i blocchi vengono raggruppati in griglie Entrambe le struttura possono poi essere ulteriormente strutturate in maniera 1D, 2D o 3D in base al problema da risolvere.

15. Come mai c'è bisogno di una gerarchia di thread?

La gerarchia di thread permette di scomporre problemi complessi in unità di lavoro parallele più piccole e gestibili, rispecchiando spesso la struttura intrinseca del problema stesso. Ad esempio si possono distinguere sottoproblemi paralleli diversi in griglie diverse e la griglia può essere strutturata al suo interno in blocchi che logicamente rispecchiano la risoluzione del sottoproblema.

Il programmatore poi, può controllare la dimensione dei blocchi (e della griglia) per adattare l'esecuzione alle caratteristiche specifiche dell'hardware e del problema, ottimizzando l'utilizzo delle risorse della GPU a disposizione. La gerarchia risulta quindi scalabile e permette di adattare l'esecuzione a GPU con diverse

capacità e numero di core. Il codice CUDA, quindi, risulta più portabile e può essere eseguito su diverse architetture GPU.

Inoltre, la distinzione tra blocchi e griglie permette operazioni, come sincronizzazione e allocazione di memoria condivisa, che sarebbero troppo costose a livello globale.

16. Che cos'è un kernel CUDA?

Un kernel CUDA è una funzione che viene eseguita in parallelo sulla GPU da migliaia/milioni di thread. Al suo interno vi è definito che cosa il singolo thread dovrà fare, ed il mapping ai dati che dovrà elaborare.

17. Ci sono dei limiti per quanto riguarda il dimensionamento di blocchi e griglie?

Si, il numero massimo totale di thread per blocco è 1024 per la maggior parte delle GPU. Inoltre, le dimensioni di griglie e blocchi (anche 3D) sono limitate. I valori variano in base a alla compute capability della GPU.

La Compute Capability (CC) di NVIDIA è un numero che identifica le caratteristiche e le capacità di una GPU NVIDIA in termini di funzionalità supportate e limiti hardware.

18. Che influenza ha il dimensionamento di blocchi e griglie sulle performance?

Il dimensionamento di blocchi è griglie ha conseguenze dirette sull'utilizzo delle risorse della GPU e sull'occupancy raggiungibile.

Ad esempio:

- una configurazione con una manciata (< num SM) di blocchi con tanti thread, non attiva tutti gli SM della GPU diminuendo il parallelismo.
- al contrario, una configurazione con tanti blocchi composti da una manciata di thread causa un overhead eccessivo per lo scheduler dei blocchi e **potrebbe diminuire l'occupancy siccome si è limitati dal numero di blocchi assegnabile ad un SM**.

Altri fattori possono essere:

- blocchi più grandi che accedono a dati localmente vicini possono anche sfruttare meglio la cache L1 rispetto a blocchi più piccoli.
- blocchi con dimensione non multipla di 32 causano l'esecuzione di warp con molti thread disabilitati con spreco delle unità di calcolo

19. Che influenza ha il mapping dei dati ai thread sulle performance?

Innanzitutto, è ovvio che il mapping dei dati deve essere corretto, ovvero bisogna garantire che il mapping permetta di ottenere un risultato del calcolo parallelo uguale a quello del calcolo sequenziale. Per fare questo il mapping deve: garantire una copertura completa dei dati senza ripetizioni.

Più interessante è il fatto che il mapping dei dati influenzi la scalabilità ed il parallelismo che si riesce ad ottenere da un kernel. Con un mapping inappropriato il kernel potrebbe non scalare al crescere della dimensione dei dati (pensa all'esempio del kernel in cui un thread somma un'intera colonna tra due matrici) oppure potrebbe diventare proprio impossibile risolvere il problema (pensa ad un mapping con solo threadId -> si è limitati dalla dimensione del blocco).

Altri aspetti influenzati dal mapping dei dati sono: l'accesso alla memoria che idealmente deve essere allineato e coalescente, e il bilanciamento del carico tra i thread che idealmente deve essere uniforme.

Riassumendo il mapping deve garantire:

- Copertura completa dei dati
- Scalabilità per diverse dimensioni dei dati
- Coerenza dei risultati con l'elaborazione sequenziale
- Accesso efficiente alla memoria

20. Esistono diversi metodi di mapping dei dati?

Si, noi ne abbiamo visti due:

- metodo lineare: più comodo quando si ha a che fare con configurazioni 1D
- metodo per coordinate: più comodo quando si ha a che fare con configurazioni 2D/3D Metodi diversi possono produrre indici globali differenti per lo stesso thread, impattando in questo modo le prestazione del kernel per aspetti come la coalescenza degli accessi in memoria.

Modello di esecuzione CUDA

21. Che cos'è il modello di esecuzione?

Il modello di esecuzione è un modello che fornisce una visione di come i kernel lanciati lato host vengano effettivamente eseguiti sulla GPU. Studiare il modello di esecuzione è utile in quanto:

1. Fornisce indicazioni utili per l'ottimizzazione del codice
2. Facilita la comprensione della relazione tra il modello di programmazione e l'esecuzione effettiva.

22. Che cos'è un SM e da che cosa è composto?

Gli SM sono dei processori, ovvero cio che esegue le istruzioni specificate dai thread. Ogni SM, al suo interno, contiene:

- diverse **unità di calcolo** (INT unit, FP32 unit, FP64 unit, SFU, Tensor Core, ecc...), ognuna delle quali è in grado di eseguire un thread in parallelo con altri nel medesimo SM (Un SM ospita più Blocchi e quindi multipli warp/thread).

- almeno una coppia (ma di solito di più) di **warp-scheduler** e **dipatch unit**; queste due unità si occupano rispettivamente di: selezionare quali sono i warp pronti all'esecuzione (all'interno di un blocco assegnato al SM) e di assegnare effettivamente ai warp selezionati le unità di calcolo appropriate.

- un'insieme di registri, che vengono spartiti ai thread in esecuzione all'interno dell'SM per la memorizzazione ed il calcolo di dati temporanei.

- Shared memory/L1 cache: una memoria super veloce condivisa tra i thread di un blocco (ecco perchè la shared memory è shared solo all'interno del blocco -> è fisicamente presente solo all'interno del SM in cui il blocco è stato assegnato). La stessa memoria è divisa tra shared memory (cache programmabile) e cache, ed è il programmatore a decidere/consigliare quanta memoria assegnare all'una rispetto all'altra in base al problema da risolvere.

In realtà, questi sono i componenti principali di SM vecchi. Gli SM delle GPU moderne suddividono poi gli SM in vari SMSP, e sono loro ad essere fatti così. In questo modo si aumenta ulteriormente il parallelismo disponibile a livello di hardware.

Infine, a livello di architettura di GPU globale, sono notevoli anche:

- la cache L2: che è condivisa tra tutti gli SM e (quindi tutti i blocchi)
- il giga thread engine: Scheduler globale per la distribuzione dei blocchi.

23. Come vengono distribuiti i blocchi tra i vari SM?

- Quando un kernel viene lanciato, i blocchi di vengono automaticamente e dinamicamente distribuiti dal GigaThread Engine agli SM.
- Le variabili di identificazione e dimensione: gridDim, blockIdx, blockDim, e threadIdx sono rese disponibili ad ogni thread
- Una volta assegnati a un SM, i thread di un blocco eseguono esclusivamente su quell'SM.
- Più blocchi possono essere assegnati allo stesso SM contemporaneamente.
- **Lo scheduling dei blocchi dipende dalla disponibilità delle risorse dell'SM (registri, memoria condivisa) e dai limiti architetturali di ciascun SM (max blocks per SM, max threads per SM, max warp per SM)** (Gigathread engine fa load balancing)
- Parallelismo multi-livello nell'esecuzione:
    - Parallelismo a Livello di Istruzione: Le istruzioni all'interno di un singolo thread sono eseguite in pipeline.
    - Parallelismo a Livello di Thread: Esecuzione concorrente di gruppi di thread (warps) sugli SM (SIMT).
    - (considerando l'intera GPU, abbiamo anche parallelismo a livello di griglia: SM diversi possono processare blocchi appartenenti a griglie diverse)

24. Parlami di SIMT e delle sue differenze con il modello SIMD

SIMT è un modello di esecuzione dei thread adottato in CUDA in cui:

- Per prima cosa, i thread di un blocco vengono divisi in **warp**, ovvero gruppi di thread di dimensione fissa (32)
- Successivamente (similmente a quanto accade in SIMD) tutti i thread di uno stesso warp eseguono la stessa istruzione
- La differenza principale con SIMD sta nel fatto che questo modello ammette divergenza!
    - Quando thread appartenenti allo stesso warp divergono si ha semplicemente esecuzione seriale dei due percorsi.
    - Quando si intraprende un percorso i thread che non appartengono a quest'ultimo vengono disabilitati
    - Questa esecuzione seriale fa perdere parallelismo e quindi seppure la divergenza sia possibile, è lo stesso da evitare (diminuisce la branch efficiency)

25. Parlami di più dei warp ed in particolare del warp scheduling

Innanzitutto abbiamo che:

- Un warp viene assegnato a una sub-partition (dell'SM del blocco a cui appartiene) dove rimane fino al completamento.

- Una sub-partition gestisce un “pool” di warp concorrenti di dimensione fissa (es., Turing 8 warp, Volta 16 warp).
    - altro limite architetturale definito dalla CC

Successivamente, si ha che **un warp ha un contesto di esecuzione** (similmente ad un processo in un normale SO), esso contiene:

- Warp ID (PID)
- PC (per thread per >=Volta)
- Stack (per thread per >=Volta)
- Blocchi di Registri e Shared memory (l'offset viene calcolato grazie al warpid)
- Stato di esecuzione (in esecuzione, pronto, in stallo)
- Thread-mask **NB**: Notevole il fatto che questo contesto venga salvato on-chip per tutta la durata d'esecuzione del warp. In questo modo **il cambio di contesto è senza costo** quando si vuole eseguire un altro warp (operazione fondamentale per latency hiding)

L'attività di warp scheduling consiste nel selezionare dal pool dei warp attivi, un warp appartenente al sottoinsieme dei warp pronti da mandare in esecuzione su un SMSP. Similmente ad uno scheduler classico di un normale SO, se un warp in esecuzione entra in stallo, viene fatto immediatamente un cambio di contesto (senza costo) e viene messo in esecuzione un altro warp. Questo meccanismo è alla base del **latency hiding** e permette di mantere alta l'occupazione delle risorse del SM nascondendo la latenza delle operazioni costose come gli accessi alla memoria globale.

Le unità che si occupano del warp scheduling sono:

- I warp scheduler all'interno di un SMSP che selezionano i warp eleggibili ad ogni ciclo di clock e li inviano alle dispatch unit
- Le dispatch unit, responsabili dell’assegnazione effettiva alle unità di esecuzione Più Warp scheduler e dispatch unit si ha disposizione all'interno di un SMSP, più in fretta si riesce a riempire quest'ultimo e più in fretta si riesce a sostituire molti warp che entrano in stallo.

La tecnica di sopra rientra nella categoria del Thread Level Parallelism. Un'ulteriore tecnica adottabile (ILP) consiste nel fare emettere al warp scheduler istruzioni indipendenti apparteneti allo stesso warp.

TLP e ILP contribuiscono a mantenere le unità di calcolo attive e occupate riducendo i tempi morti dovuti alle operazioni a latenza elevata come accessi alla memoria globale (latency hiding).

26. Parlami di come si può ottenere il latency hiding massimo

Siccome il latency hiding si ottiene sostituende il warp correntemente in stallo con un warp pronto, una condizione necessaria per massimizzare quest'ultimo è avere a disposizione "tanti" warp pronti.

Una formalizzazione di questo concetto più operativa è data dalla **Legge di Little**. Questa legge ci aiuta a calcolare quanti warp (approssimativamente) devono essere in esecuzione/pronti per ottimizzare il latency hiding e mantenere le unità di elaborazione della GPU occupate.

```
Warp Richiesti = Latenza × Throughput
```

Con:

- Latenza = Tempo di completamento di un'istruzione (in cicli di clock).
- Throughput = Numero di warp (e, quindi 32 operazioni) eseguiti per ciclo di clock.
- Warp Richiesti = Numero di warp pronti necessari per nascondere la latenza ed ottenere il throughput desiderato

Inoltre, una ulteriore strategia possibile è cercare di scrivere operazioni all'interno del kernel il più possibile indipendenti in modo che possano essere emesse in sequenza dal warp scheduler(vedi ILP). Questo è molto **sinergistico con il loop unrolling**, che di suo riduce il numero di istruzioni di controllo da eseguire, ma inoltre aumenta il numero di operazioni indipendenti (vedi riduzione parallela)

27. Che cos'è Indipendent Thread Scheduling?

Prima di ITS il livello di concorrenza minimo era tra Warp siccome era l'intero warp ad avere un PC ed uno stack. Con ITS ogni thread mantiene il proprio stato di esecuzione, inclusi program counter e stack. Di conseguenza, dopo ITS, il livello di concorrenza minimo diventa quello dei singoli thread, anche appartenenti a warp diversi o a rami diversi dello stesso warp.

Prima di ITS

- Quando c'è divergenza, i thread che prendono branch diverse perdono concorrenza fino alla riconvergenza.
- Possibili deadlock tra thread in un warp, se i thread dipendono l'uno dall'altro in modo circolare. Con ITS entrambe queste situazioni vengono mitigate:
- Posso eseguire concorrentemente(non parallelamente) istruzioni appartenenti a rami diversi all'interno di un warp
- Un ramo può attendere un altro ramo Infine, un ottimizzatore di scheduling raggruppa i thread attivi dello stesso warp in unità SIMT mantenendo l'alto throughput dell'esecuzione SIMT, come nelle GPU NVIDIA precedenti.

28. Perchè sono necessarie le operazioni atomiche in CUDA?

Quando più thread accedono e modificano la stessa locazione di memoria contemporaneamente si ha una corsa critica che produce risultati imprevedibili. Le operazioni atomiche garantiscono la correttezza del risultato impattando pesantemente però sulle performance siccome i thread vengono sequenzializzati durante l'esecuzione di quest'ultima.

Una soluzione a questo problema è duplicare i dati sulla SMEM limitando la sequenzializzazione a livello di blocco. Questa idea ci è stata anche mostrata dal professor Mattoccia nel contesto di OpenMP ed è quindi applicabile anche al di fuori di CUDA.

29. Che cos'è il resource partitioning in CUDA?

Il Resource Partitioning riguarda la suddivisione e la gestione delle risorse hardware limitate all'interno di una GPU, in particolare all'interno di ogni SM. L'obiettivo è massimizzare l'occupancy ed il parallelismo (permettendo l'assegnamente di più blocchi possibile all'interno di un SM)

Le risorse richieste da un blocco (e quindi da partizionare dentro ad un SM) sono tre:

1. La dimensione del blocco, ovvero il numero di thread che devono essere concorrenti (in realtà anche numero di blocchi e numero di warp).
2. Memoria Condivisa

3. Registri

In caso di problemi di occupancy, ridimensionare la quantità di smem, oppure il numero di registri necessari ad un thread, potrebbe risolvere il problema permettendo l'assegnamento di più blocchi allo stesso SM. **Cio che si desidera è raggiungere il numero di thread massimo gestibile dall'SM** (corrisponde ad una occupancy del 100%).

30. Che cos'è l'occupancy in CUDA?

L'occupancy è definita come il rapporto tra i warp attivi e il numero massimo di warp supportati per SM

```
Occupancy [%] = Active Warps / Maximum Warps
```

Una occupancy alta permette di fare latency hiding siccome ci saranno molti warp disponibili a sostituire quelli entrati in stallo; è chiaro che con un resource partitioning infelice il numero di thread occupati (warp) sarà molto minore rispetto al massimo supportato e questo si rifletterà sull'occupancy.

Alcuni fattori che possono abbassare l'occupancy sono:

- Dimensioni del blocco di thread: Se i blocchi sono troppo piccoli, si potrebbe essere limitati dal numero massimo di blocchi assegnabili ad un SM e non raggiungere il numero massimo di warps attivi.
- Uso dei registri: Se un kernel utilizza troppi registri per thread, il numero totale di blocchi assegnabili ad un SM sarà limitato. Si potrebbe diminuire la dimensione del blocco ma questo causa il problema di sopra.
- Memoria condivisa: Se un kernel utilizza molta memoria condivisa, potrebbe ridurre il numero di blocchi assegnabili un SM.

31. Parlami di CUDA Dynamic Parallelism

CDP è una estensione del modello di programmazione CUDA che permette la creazione e sincronizzazione dinamica (a runtime) di nuovi kernel direttamente dalla GPU.

- Molto utile per implementare algoritmo ricorsivi sulla GPU.
- È possibile posticipare a runtime la decisione su quanti blocchi e griglie creare sul device (vedi raffinamento adattivo della griglia nella simulazione di fluido dinamica)
- Elimina in alcuni casi continui trasferimenti di memoria tra CPU e GPU
- La GPU non è più un coprocessore totalmente governato dalla CPU ma diventa indipendente!

Come funziona:

- Il kernel/griglia parent continua immediatamente dopo il lancio del kernel child (asincronicità).
- Attenzione a sincronizzare accessi ad aree di memoria comuni tra parent e child. La memoria è coerente tra i due solo:
    - All'avvio della griglia child: il child vede tutto quello che ha fatto il padre prima del suo lancio
    - Quando la griglia child completa: il padre vede tutto quello ha fatto il figlio dopo che si è sincronizzato
- Puntatori a Memoria locale e smem l'uno dell'altro non sono accessibili in quanto rappresentano memoria privata

- Un parent si considera completato solo quando tutte le griglie child create dai suoi thread (tutti) hanno terminato l'esecuzione (sincronizzazione implicita se il padre termina prima dei child).
    - Sincronizzazione esplicita possibile con *CudaDeviceSynchronize()*

Modello di memoria CUDA

32. Parlami di kernel compute bound e kernel memory bound, come mai è importante distinguere queste due categorie? Un kernel è memory bound quando il tempo di esecuzione è limitato dalla velocità di accesso alla memoria piuttosto che dalla capacità di elaborazione dei core.

- Le unitò di calcolo della GPU trascorrono più tempo in attesa dei dati rispetto a eseguire calcoli
- poche operazioni per byte letto/scritto.
- Accessi frequenti alla memoria.
- Banda di memoria insufficiente rispetto ai requisiti del kernel. **NB**: è notevole il fatto che la peak performance teorica di una GPU considerando solo le unità di calcolo sia molto maggiore rispetto alla peak performance considerando la sua bandwidth. Questo significa che una GPU è intrinsecamente limitata da quanto velocemente può alimentare le sue unità di calcolo con i dati (collo di bottiglia nel caso di workload memory-bound).

Un kernel è compute Bound quando il tempo di esecuzione è limitato dalla capacità di calcolo della GPU, con sufficiente larghezza di banda per i dati.

- La GPU trascorre più tempo a eseguire calcoli rispetto all’attesa dei dati.
- Operazioni aritmetiche intensive
- Molte operazioni per byte letto/scritto

Questa distinzione è utile i quanto per ottimizzare un kernel è cruciale comprendere se il collo di bottiglia risiede negli accessi alla memoria o nella capacità computazionale della GPU. Questa distinzione determina le strategie di ottimizzazione da adottare. Ad esempio:

- In un contesto memory bound, è di notevole importanza ottimizzare gli accessi alla memoria considerando quale sia il tipo di memoria pià opportuna da usare e i pattern di accesso. Questo massimizza la bandwith effettiva di trasferimento dei dati
    - Distinguiamo tra:
        - bandwidth teorica (considera solo i dati "lordi" trasferiti)
        - bandwidth effettiva (byte effettivamente letti/scritti; considera gli sprechi)
- In un contesto compute bound, è di notevole importanza massimizzare l'occupancy delle unità di eleaborazione.
- In entrambi i casi la scelta del tipo di dato influisce sulle performance
    - memory bound: un tipo più piccolo mi permette di trasferire meno dati
    - compute bound: tipi di dato diversi possono avere un numero diverso di unità di elaborazione

33. Come possiamo capire se un kernel è memory bound o compute bound? Che cos'è il diagramma di roofline? Innanzitutto un kernel memory bound su una GPU potrebbe diventare compute bound su di un altra e viceversa. Questo perchè GPU diverse hanno bandwidth e velocità di elaborazione diverse. Per stabilire la tipologia di un kernel utilizziamo quindi due metriche:

- Intensità aritmetica
    - dipende solo dal kernel

    - definita come il rapporto tra la quantità di operazioni di calcolo e il volume di dati trasferiti dalla/verso la memoria di un kernel: AI = FLOPs / Byte richiesti
    - misura quante operazioni il kernel fa per byte trasferito
- Soglia di intensità aritmetica
    - dipende dalla GPU in considerazione
    - definita come il rapporto tra la peak performance per una determinata operazione e la bandwidth massima (teorica) della GPU Soglia(AI) = Theoretical Computational Peak Performance (FLOPs/s) / Bandwidth Peak Performance (Bytes/s)
    - misura quante operazioni la GPU è in grado di eseguire per byte trasferito

Unendo le informazioni di queste due metriche possiamo definire un kernel come:

- memory bound: se AI < Soglia(AI)
    - ovvero il kernel ha bisogno di eseguire meno operazioni per byte trasferito rispetto a quante ne supporta la GPU
- compute bound: se AI > soglia(AI)
    - il contrario di sopra

Il diagramma di roofline è solo una visualizzazione grafica di quanto detto sopra:

- la soglia(AI) è il punto in cui la linea diventa parallela all'asse x
- in base alla AI, il kernel si può collocare prima o dopo il punto di svolta definito dalla soglia(AI) rendendo immediatamente visibile se esso è memory bound o compute bound
- abbiamo roofline diverse per tipi di dato e memorie diverse

34. Che tipi di memoria esistono in CUDA?

- **Registri**:
    - Ci vanno dentro le variabili locali all'interno dei kernel
    - Strettamente privati per thread con durata limitata all'esecuzione del kernel.
    - Allocati dinamicamente tra warp attivi in un SM; Minor uso di registri per thread permette di avere più **blocchi** concorrenti per SM (maggiore occupancy).
    - Limite di [63-255] registri per thread
    - Register Spilling verso la memoria locale
- **Memoria locale**:
    - off-chip (DRAM)
    - Privata per thread
    - Utilizzata per variabili che non possono essere allocate nei registri a causa di limiti di spazio (array locali, grandi strutture)
    - Variabili che eccedono il limite di registri del kernel finiscono qua (register spill)
- **SMEM e Cache L1**:
    - Ogni SM ha memoria on-chip (molto veloce) limitata (es. 48-228 KB), condivisa tra smem e cache L1
    - SMEM praticamente una cache programmabile condivisa tra thread di un blocco; Cache L1 Serve tutti i thread dell'SM

    - SMEM richiede sincronizzazione per prevenire corse critiche
    - La quantità da assegnare alla smem rispetto alla cache L1 è configurabile

- **Memoria costante**:

    - off-chip (DRAM)
    - scope globale (visibile a tutti i kernel)
    - Inizializzata dall'host e read-only per i kernel

- **Memoria Texture**

    - ...

- **Memoria Globale**

    - Memoria più grande e pià lente, e più comunemente usata sulla GPU.
    - Memoria principale off-chip (DRAM) della GPU, accessibile tramite transazioni da 32, 64, o 128 byte.
    - Scope e lifetime globale (da qui global memory)
        - Accessibile da ogni thread di tutti i kernel
        - Occhio alle corse critiche ed alla sincronizzazione (no sincronizzazione tra blocchi)
    - Fattori chiave per l'efficienza:
        - Coalescenza: Raggruppare accessi di thread adiacenti a indirizzi contigui.
        - Allineamento: Indirizzi di memoria allineati con dim delle transazioni

- **Vari tipi di cache**

    - Cache L1
        - Ogni SM (non SMSP) ne ha una propria
    - Cache L2
        - Unica e condivisa tra SM.
        - Funge da ponte tra le cache L1 più veloci e la memoria principale più lenta.
    - Constant Cache (sola lettura, per SM)
    - Texture Cache (sola lettura, per SM)

Le cache GPU, come quelle CPU, sono memorie on chip **non programmabili** utilizzate per memorizzare temporaneamente porzioni della memoria principale per accessi più veloci.

35. Come vengono trasferiti i dati dalla memoria dell'host alla memoria del device? A che cosa bisogna stare attenti in questo processo?

I dati nell'host vengono trasferiti sul device tramite il bus PCIe. Questo bus ha una bandwidth notevolemente minore rispetto a quella della memoria del device e potrebbe essere quindi un collo di bottiglia nell'esecuzione dell'applicazione. Diventa essenziale quindi massimizzare la velocità di trasferimento dati tra host e device; per fare questo bisogna prima capire come vengono effettivamente trasferiti i dati su i due dispositivi.

Memoria Pageable:

- La memoria allocata dall’host di default è pageable (soggetta a swap-out notificati tramite page fault).

- La GPU non può accedere in modo sicuro alla memoria host pageable (mancanza di controllo sui page fault).

Come avviene allora il trasferimento da Memoria Pageable?

- Il driver CUDA alloca temporaneamente memoria host pinned (non soggetta a swap-out, bloccata in RAM).
- Copia i dati dalla memoria host sorgente alla memoria pinned.
- Trasferisce i dati dalla memoria pinned alla memoria del device (in modo sicuro siccome c'è la garanzia che i dati siano effettivamente presenti in RAM).

NB: L'overhead dovuto alla allocazione temporanea di memoria pinned e copia dei dati su quest'ultima è la causa del peggioramento di performance di quando si fanno trasferimenti multipli rispetto a un trasferimento grande (Con un singolo trasferimento pago il costo di allocazione e copia 1 volta, con *n* trasferimenti lo pago *n* volte).

Cio a cui bisogna fare attenzione è quindi evitare trasferimenti multipli quando non necessario. In alternativa posso, allocare direttamente della memoria pinned sull'host e, non solo non mi devo preoccupare di raggrupare i trasferimenti, ma evito anche proprio il costo di copia che avevamo prima da memoria paginabile a pinned (inoltre, la memoria pinned mi permette anche di effettuare dei trasferimenti asincroni). La memoria pinned però è più costosa da allocare e se ne alloco troppa possono degradare le prestazioni dell'host.

36. Che cos'è la memoria zero copy?

La memoria "Zero-Copy" è una tecnica che consente al device di accedere direttamente alla memoria dell'host senza la necessità di copiare esplicitamente i dati tra le due memorie. In pratica è memoria pinned dell’host che è **mappata nello spazio degli indirizzi del device** e di conseguenza è accessibile a quest'ultimo (stessa memoria fisica ma puntatori comunque diversi).

- È un'eccezione alla regola che l'host non può accedere direttamente alle variabili del dispositivo e viceversa.
- Utile per dati a cui si accede raramente, evitando copie in memoria device e riducendo l'occupazione di quest'ultima
    - Evita trasferimenti espliciti (impliciti per il PCIe)
- Peggiora le prestazioni se utilizzata per operazioni di lettura/scrittura frequenti o con grandi blocchi di dati in quanto ogni transazione alla memoria mappata passa per il bus PCIe che ha una bandwidth piccola.
- Inoltre, si ha accesso concorrente a un area di memoria comune da parte di host e device... Necessità di sincronizzazione altrimenti corse critiche

37. Che cosa sono Unified Virtual Addressing (UVA) e Unified Memory (UM)?

UVA è una tecnica che permette alla CPU e alla GPU di condividere lo stesso spazio di indirizzamento **virtuale** (la memoria fisica rimane distinta). Elimina quindi la distinzione tra un puntatore (adesso virtuale) host e uno device.

NB: Con UVA però, si ha comunque bisogno di sapere se si sta allocando memoria su GPU o CPU dato che questa tecnica **non gestisce la migrazione dei dati** nelle corrette memoria fisiche, richiedendo trasferimenti manuali espliciti.

UM è una estensione di UVA che oltre allo spazio di indirizzamento unico gestisce in maniera automatica anche i trasferimenti di memoria tra i vari dispositivi. Si parla di *managed memory* (essa è specificabile con opportune keyword).

- Non è più necessario esplicitare i trasferimenti (come con la memoria zero copy) e non è più necessario distinguere tra puntatori host e device (come con UVA). In pratica posso fare *cudaMallocManaged/malloc* e non preoccuparmi più di niente come se stessi utilizzando un unico dispositivo.

Queste due tecniche astraggono i dettagli di gestione della memoria tra i vari dispositivi eliminando la necessità di duplicare i puntatori, fare trasferimenti, ecc. Come ogni astrazione però, peggiora la performance dato che il sistema deve capire dove e come trasferire i dati ed inoltre il posizionamento dei dati potrebbe non essere ottimale. Per le performance massime, la gestione "classica" è migliore.

38. Che cosa si intende con pattern di accesso alla memoria? Come mai è importante avere un pattern ottimo per le performance di un kernel?

Facciamo un passo indietro e definiamo prima che cos'è una transazione di memoria. Abbiamo che:

- Le operazioni di memoria sono emesse ed eseguite per warp (32 thread).
- Ogni thread fornisce l'indirizzo di memoria a cui vuole accedere, e la dimensione della richiesta del warp dipende dal tipo di dato (es. 32x4B per int, 32x8B per double).
- La richiesta (lettura o scrittura) è servita da una o più transazioni di memoria.
- Quest'ultime sono delle operazioni atomiche di lettura/scrittura tra la memoria globale e gli SM della GPU.
- Le transazioni di memoria avvengono in blocchi di dimensioni variabili, come 128 byte o 32 byte

Con i pattern di accesso alla memoria si classifica come sono distribuiti gli indirizzi di una richiesta di accesso alla memoria globale da parte dei thread di un warp. In particolare si distinguono due caratteristiche:

1. Allineamento: quando l'indirizzo iniziale della transazione è multiplo della dimensione di quest'ultima.
2. Coalescenza: quando tutti i 32 thread di un warp accedono ad un blocco contiguo di memoria. Entrambe queste caratteristiche sono desiderabili per ridurre al minimo il numero di transazioni richieste per servire la memoria richiesta dal warp. Quando una di questa due proprietà è assente (non sempre è possibile averle entrambe) il numero di transazioni diventa maggiore del minimo teorico e, siccome esse corrispondono ad accessi alla memoria aggiuntivi sequenziale, si ha un peggioramento delle performance.

Con un pattern di accesso ottimale si riesce a massimizzare la bandwidth effettiva nella lettura/scrittura dei dati, e siccome la maggior parte delle applicazioni GPU è limitata dalla larghezza di banda della memoria DRAM, ottimizzare l'uso della memoria globale è fondamentale per le prestazioni del kernel.

39. Puoi farmi qualche esempio di utilizzo della SMEM

1. Come ogni forma di memoria condivisa può essere utilizzata come canale di comunicazione per i thread appartenenti allo stesso blocco.
2. Memoria scratch pad temporanea per elaborare dati on-chip e **migliorare i pattern di accesso alla global memory**. Ad esempio, nel caso della trasposta di una matrice, piuttosto che leggere

direttamente le colonne in maniera non coalescente, posso prima far caricare a tutti i thread (del blocco, siamo limitati in dimensione ma posso gestire con i tile) i dati della matrice in smem **in maniera coalescente**, e poi accedere alla smem in maniera NON coalescente, siccome però stiamo accedendo alla smem e non alla memoria globale, alla peggio ci saranno dei bank conflict.

3. Memoria scratch pad temporanea per limitare le sincronizzazioni dovute alle operazioni atomiche come nel caso dell'istogramma. Se preparo una copia dei dati in smem per ogni blocco e faccio lavorare i thread sulla smem piuttosto che sulla memoria globale, limito di molto la concorrenza nell'accesso ai dati condivisi e quindi le relative sincronizzazioni.
4. Come memoria temporanea in cui carico i dati per evitare accessi multipli alla memoria globale come nel caso della riduzione parallela. Se ogni thread carica un dato in smem, i thread che prima facevano accessi multipli alla memoria globale ora ne fanno solo uno (per caricare il proprio dato), e gli accessi multipli si spostano verso la smem.

40. Come si utilizza la SMEM?

1. Caricamento in Shared Memory (Global → Shared)
2. Sincronizzazione Post-Caricamento
    - in modo che tutti i thread vedano nella smem i dati caricati dagli altri
3. Elaborazione Dati
4. (Opzionale) Sincronizzazione Post-Elaborazione
    - in modo che tutti i thread vedano nella smem i dati caricati dagli altri
    - opzionale se non mi serve vedere le modifiche degli altri thread
5. Scrittura dei Risultati (Shared → Global)

41. Che cos'è un bank conflict?

La shared memory è suddivisa in 32 banchi (warp size) di memoria, ovvero 32 bin contenenti una word. Questa suddivisione permette ai thread di uno stesso warp di accedere contemporaneamente alla smem **se i thread accedono ad indirizzi di quest'ultima corrispondenti a banchi diversi**.

Quando più tread tentano di accedere ad **indirizzi diversi corrispondenti allo stesso banco** si verifica un bank conflict che ha come conseguenza la sequenzializzazione delle risposte ai vari thread. Questo è chiaramente non desiderabile in quando diminuisce la potenziale bandwidth.