

Parallel Computing (Revisited)

Tradicionalmente il software è scritto per eseguire in maniera sequenziale, quindi un'istruzione a volta, in maniera molto semplice la **computazione parallela** è l'utilizzo simultaneo di più risorse per risolvere un problema, ad esempio eseguendo il problema su più CPU rompendolo in più parti.

Tra i motivi che abbiamo di usare la computazione parallela abbiamo questioni economiche ma anche di tempo e risorse, ci consente inoltre di risolvere problemi molto più grandi e complessi. Possiamo anche utilizzare risorse di calcolo non locali (cloud computing) inoltre il calcolo seriale presenta dei limiti.

Concetti essenziali

Tra le classi di computer individuiamo:

- desktop computer
- embedded computer
- clusters
- server
- internet of things: embedded con internet
- smart systems: IoT + sensori + attuatori
- supercomputers

Architettura di Von Neumann

È un'architettura che virtualmente tutti i computer seguono, composta di 4 componenti principali:

- **Memoria(RAM)**: mantiene istruzioni e programmi
- **Control Unit(CU)**: recupera istruzioni e dati dalla memoria e coordina le operazioni
- **Arithmetic Logic Unit(ALU)**: fa le operazioni aritmetiche
- **Input/Output**: interfacce

Tassonomia di Flynn

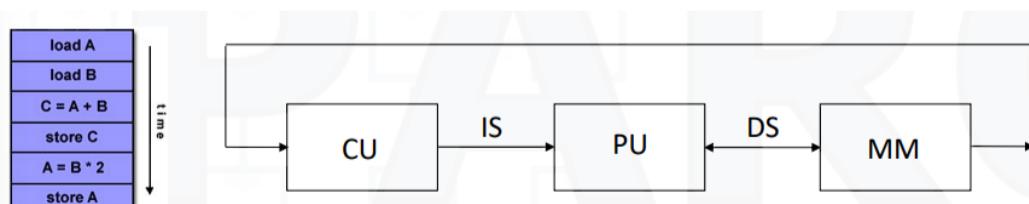
Distingue le architetture di computer multi-processore utilizzando due dimensioni, **istruzioni e dati**:

S I S D Single Instruction, Single Data	S I M D Single Instruction, Multiple Data
M I S D Multiple Instruction, Single Data	M I M D Multiple Instruction, Multiple Data

Consideriamo i seguenti componenti:

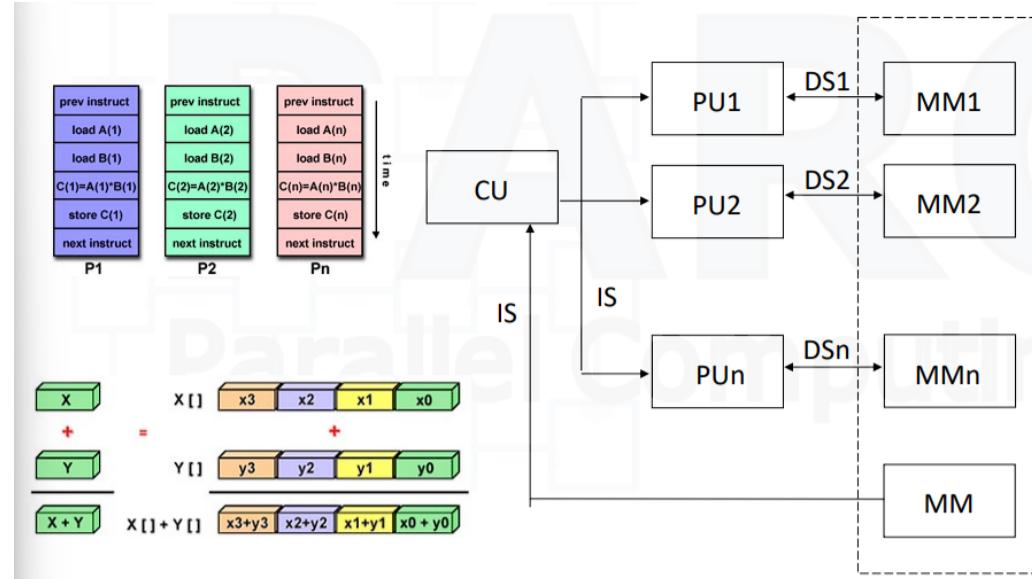
- IS: instruction stream
- CU: control unit (fa fetch decode, execute)
- MM: main memory
- DS: data stream
- PU: processing unit (fatta da ALU+registri)

SISD



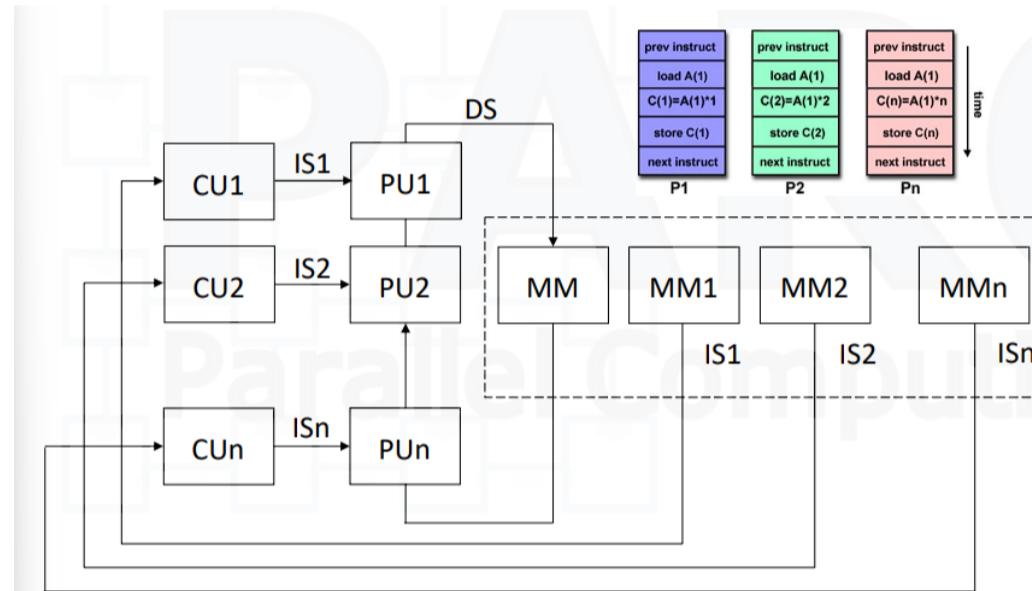
Si recuperano le istruzioni dalla memoria, la PU esegue queste ultime e aggiorna i dati in memoria, un singolo programma sta eseguendo e per questo c'è un solo data stream e un solo instruction stream. È ovviamente seriale

SIMD



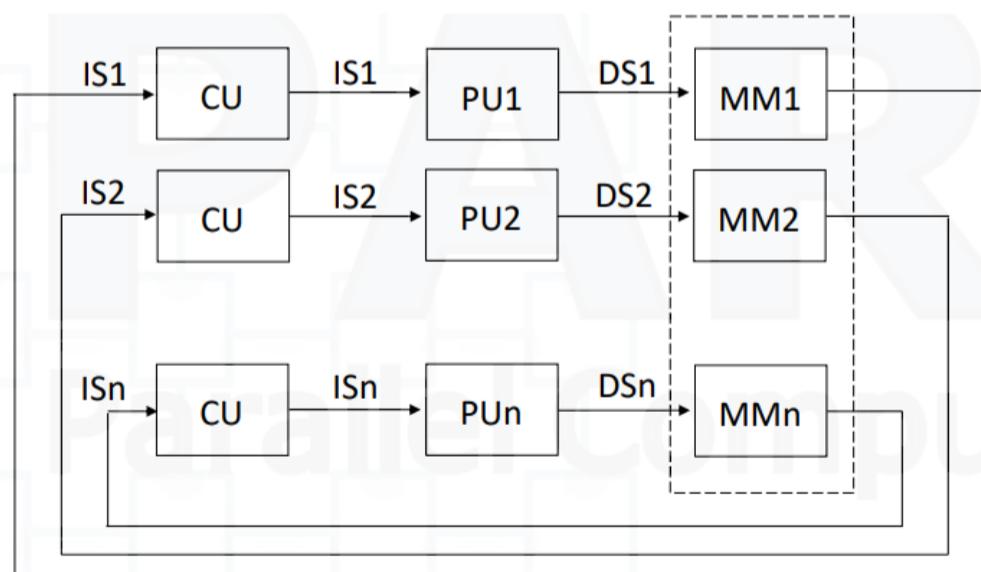
Ci sono più PU che eseguono la stessa istruzione (single instruction stream), bisogna considerare che ogni PU può eseguire con dati diversi(multiple data stream), ottima per problemi ad esempio di image processing (applico un filtro ad un'immagine, devo fare la stessa operazione su tutti i pixel). È essenziale che ci sia sincronizzazione, fatta con lockstep(ogni PU esegue e quando termina aspetta le altre).

MISD



Ci sono più control unit che recuperano più istruzioni, queste vengono mandate alle PU che però usano un solo data stream. Questo tipo di architettura è stata implementata in via sperimentale ma subito accantonata.

MIMD



Ci sono più CU che recuperano e mandano più istruzioni a più PU che usano un data stream ciascuna, essenzialmente abbiamo più programmi eseguiti in parallelo, ognuno con i propri dati.

Quasi tutti i computer comuni sono in questa categoria, l'esecuzione qui può essere sia sincrona che asincrona, deterministica o non.

Molte architetture MIMD comprendono sottocomponenti che fanno esecuzioni SIMD.

Terminologia

- task:** sezione discreta di lavoro computazionale. Tipicamente un programma o un set di istruzioni eseguite dal processore

- **parallel task**: un task eseguibile su più processori in maniera safe (senza creare errori)
- **esecuzione seriale**: esecuzione di un programma un'istruzione alla volta
- **esecuzione parallela**: esecuzione di un programma con più task alla volta (anche diverse)
- **pipelining**: rompere un task in vari step eseguiti da varie unità del processore, con gli input che scorrono nella pipe
- **memoria condivisa**: da un punto di vista hardware abbiamo un'architettura dove tutti i processori hanno accesso diretto a una memoria fisica comune. Da un punto di vista di programmazione ci sono task paralleli che accedono e indirizzano le stesse locazioni di memoria indipendentemente da dove essa sia fisicamente.
- **symmetric multi-processor(smp)**: architettura hardware dove più processori condividono un singolo spazio degli indirizzi e accesso alle risorse
- **distributed memory**: da un punto di vista hardware si riferisce all'accesso a memorie fisiche su altri hw tramite una rete. Da un punto di vista di programmazione le task vedono solo la memoria della macchina in cui sono e per accedere alle altre memorie devono comunicare con le altre macchine.
- **communication**: lo scambio di dati tra task paralleli
- **sincronizzazione**: coordinazione delle task parallele in real time, solitamente associata con la comunicazione. Tipicamente si stabilisce un punto di sincronizzazione dove una task non procede fino a che altre task coinvolte non arrivano allo stesso punto
- **granularità**: è una misura qualitativa che misura il ratio di computazione/comunicazione:
 - grossa(coarse): viene fatto molta computazione tra due eventi di comunicazione
 - fine: viene fatta poca computazione tra due eventi di comunicazione
- **speedup osservato**: è il guadagno ottenuto nell'accelerare l'esecuzione del codice:
$$\text{speedup} = \frac{t_{exe} \text{ seriale}}{t_{exe} \text{ parallelo}}$$
- **overhead parallelo**: quantità di tempo necessaria per coordinare le task parallele, include fattori come:
 - sincronizzazione
 - comunicazioni
 - software overhead
 - tempo di start e di terminazione
- **parallelizzazione massiva**: si riferisce all'hardware che compone un sistema parallelo con molti processori. (molti = centinaia di migliaia al momento, può cambiare la concezione di "molti" con lo sviluppo delle tecnologie)
- **parallelizzazione imbarazzante**: risoluzione di molti compiti simili ma indipendenti simultaneamente, poca o nessuna necessità di coordinamento
- **scalabilità**: si riferisce alla capacità di un sistema parallelo(hw/sw) di incrementare proporzionalmente all'aumentare dei processori disponibili lo speedup. Se con 10 processori ho uno speedup di 17 e con 1000 processori ho uno speedup di 20, non scala.
- **multi-core processor**: processori multipli su un singolo chip
- **cluster computing**: utilizzare una combinazione di unità di calcolo per costruire un sistema parallelo
- **supercomputing**: uso delle macchine più grandi e potenti al mondo per risolvere problemi
- **edge computing**: paradigma di computazione distribuita che porta la computazione e i dati più vicini alla locazione dove sono necessari, migliorando tempo, banda, sicurezza e privacy.

Shared Memory

In generale abbiamo tutti i processori che accedono alla memoria come se fosse uno spazio degli indirizzi globale, possono operare indipendentemente sulle stesse risorse di memoria e le modifiche fatte da uno sono visibili a tutti.

Si possono dividere principalmente in due classi le **UMA** e le **NUMA**.

Uniform Memory Access(UMA)

Più comunemente rappresentate al giorno d'oggi dalle macchine **SMP(symmetric multiprocessor)**, ci sono processori identici e c'è tempo di accesso alla memoria equivalente da parte di tutti, può essere implementata anche la **CC-UMA(cache coherent)**, si verifica quando dopo la modifica da parte di un processore anche tutti gli altri vengono a sapere di questo cambiamento.

Non Uniform Memory Access(NUMA)

E' spesso fatta collegando fisicamente **due o più SMP**, un SMP può accedere direttamente alla memoria di un altro SMP. Non tutti i processori hanno tempo di accesso a tutte le memorie equivalenti, ovviamente l'accesso alle memorie tramite i collegamenti è più lento, se è mantenuta la coerenza di cache vengono chiamate **CC-NUMA**.

Vantaggi delle Shared Memory

Lo spazio degli indirizzi globale fornisce una prospettiva di programmazione user friendly, lo share dei dati tra task è veloce e uniforme grazie alla prossimità della memoria alle cpu.

Svantaggi delle Shared Memory

Sicuramente lo svantaggio più importante è quello della mancanza di scalabilità tra memoria e CPUs, aumentare il numero di cpu difatti può aumentare il traffico nel percorso cpu-memoria e di conseguenza per i cache coherent anche il traffico associato alla gestione cache/memoria.

Inoltre è il programmatore che deve occuparsi di gestire la sincronizzazione degli accessi alla memoria, infine diventa costoso e difficile produrre macchine con un alto numero di processori.

Distributed Memory

Richiedono una rete di comunicazione per connettere le varie memorie dei vari processori, ogni processore difatti ha la sua singola memoria, che viene condivisa con gli altri processori, memoria che però non si mappa ad altri processori quindi non esiste un concetto di spazio degli indirizzi globali, difatti ci sarà l'indirizzo x12 in un processore e può ripresentarsi anche in un altro processore.

Non è necessario introdurre il concetto di cache coherency in quanto ogni processore agisce indipendentemente e quindi non vanno notificati gli altri di eventuali cambiamenti nella sua memoria. Ovviamente un processore può però accedere alle memorie degli altri processori e quando deve fare ciò solitamente è compito del programmatore definire come e quando i dati vengono comunicati, così come è responsabilità del programmatore sincronizzare i task.

Vantaggi delle Distributed Memory

La memoria è scalabile con il numero dei processori, basta incrementare il numero di processori e la dimensione della memoria proporzionalmente, inoltre ogni processore può accedere alla sua memoria senza interferenza e overhead per mantenere la cache coherency, inoltre è meno costosa.

Svantaggi delle Distributed Memory

Il programmatore è responsabile di vari dettagli riguardanti la comunicazione tra processori, potrebbe essere difficile mappare le strutture dati esistenti, basate su memoria globale su questo tipo di organizzazione. Abbiamo infine tempi di accesso non uniformi

Hybrid Distributed Memory

I computer più grandi e potenti del giorno d'oggi hanno entrambe le architetture di memoria condivisa e distribuita.

Il componente shared memory è solitamente una macchina SMP cache coherent.

Il componente di memoria distribuita invece è la rete di SMP che si crea e permette la comunicazione delle varie SMP. Probabilmente sarà l'architettura standard dei PC del futuro.

Modelli per la programmazione parallela

Ci sono vari modelli che si usano comunemente:

- shared memory
- threads
- message passing
- data parallel
- hybrid

Vengono usati ed esistono per astrarre sull'architettura hardware e della memoria, ognuno di essi teoricamente può essere implementato su qualsiasi hardware (non sempre è vantaggioso).

Scgliere il modello da utilizzare è spesso una combinazione tra ciò che è disponibile (ciò che abbiamo a disposizione) e ciò che più si adatta a quello che dobbiamo fare. Non c'è un modello migliore di un altro, ma c'è un implementazione migliore (nel senso di più adatta) di un'altra.

Shared Memory

Le task condividono uno spazio di indirizzi comune, che leggono e scrivono in modo asincrono, l'accesso a questi è regolamentato con lock e semafori.

Un vantaggio di questo modello è che viene a mancare il concetto di proprietà dei dati, tanto che non serve specificare le comunicazioni tra task in quanto ogni task è libera indipendentemente di leggere o modificare dati nella memoria che è condivisa tra tutte le task.

Tra gli svantaggi il più importante è in termini di performance, diventa più difficile capire e gestire i dati localmente
↳ dovuta ai lock

Implementazioni

Nelle piattaforme a memoria condivisa i compilatori traducono le variabili del programma in indirizzi di memoria effettivi, che sono globali: non esistono però implementazioni effettive

Threads

Un singolo processo può avere cammini di esecuzione concorrenti

Implementazioni

Sono spesso associati con architetture shared memory e sistemi operativi.

Da una prospettiva di programmazione solitamente troviamo una libreria di sottoroutine chiamate all'interno del codice parallelo o anche un insieme di direttive di compilazione, tra le implementazioni troviamo ad esempio Posix Thread o OpenMp.

OpenMp si basa sulle direttive al compilatore, essenzialmente diamo dei consigli al compilatore che porteranno a parallelizzare un codice che è sequenziale

Message Passing Model

Tra le caratteristiche di questo modello abbiamo:

- un insieme di task che usa la propria memoria locale durante la computazione
- le task scambiano dati mandandosi e ricevendo messaggi
- il trasferimento dei dati solitamente richiede di fare delle operazioni cooperando (esempio send deve avere un corrispondente receive dall'altra parte).

Implementazione

Comunemente in termini di programmazione c'è una libreria di sottoroutine embeddate nel codice sorgente.

MPI è un'implementazione di message passing model.

Data parallel model

La maggior parte del lavoro parallelo si concentra sull'eseguire operazioni su un dataset, il dataset è tipicamente organizzato in una struttura comune come un array o cubo.

Un insieme di task lavorano collettivamente sulla stessa data structure, ovviamente ogni una su una diversa partizione, le task fanno la stessa operazione sulle diverse partizioni.

Su un'architettura a memoria condivisa tutte le task ^{hanno} potrebbero aver accesso alla stessa struttura della memoria globale. Se siamo su una distribuita invece i chunk della struttura vengono messi ognuno nella memoria locale di ogni task

Single Program Multiple Data

È un modello ad alto livello che può essere costruito combinando i precedentemente visti.

Un singolo programma è eseguito da tutte le task simultaneamente, in qualsiasi momento le task possono eseguire la stessa o differenti istruzioni nello stesso programma

A differenza di SIMD, SPMD ha vari processori autonomi che eseguono simultaneamente senza l'ausilio di lockstep per sincronizzarsi, inoltre tutte le task possono usare dati diversi.

Multiple Program Multiple Data

Anche quest'ultima è un modello ad alto livello costruito combinando i precedenti, tipicamente ci sono più programmi che eseguono con più dati anche diversi tra di loro.

Misurare le Performance

Per capire quanto più veloce è una computazione x di una y usiamo lo **speedup** dato dalla formula:

$$\text{speedup} = \frac{t_{exe\ y}}{t_{exe\ x}} = \frac{\frac{1}{\text{Performance}_y}}{\frac{1}{\text{Performance}_x}} = \frac{\text{Performance}_x}{\text{Performance}_y}$$

Uno strumento sicuramente utile per misurare le performance sono i **benchmark** che sono di vario tipo:

- **kernels**: piccoli, pezzi chiave di applicazioni reali
- **toy programs**: programmi di 100 righe non molto complessi (es. Quicksort)
- **synthetic benchmarks**: programmi fake inventati per corrispondere con il comportamento di applicazioni reali (Linpack, dhystone). Sollecitano di proposito alcune componenti dell'architettura
- **benchmark suites**

Principio quantitativo

Quando vogliamo parallelizzare cercando di aumentare lo speedup incontriamo vari casi:

- dobbiamo concentrarci sul trovare i casi più eseguiti piuttosto che quelli più remoti

- spesso quelli più eseguiti sono anche i più semplici motivo per cui possono anche ottenere speed up migliori

Legge di Amdahl

Il miglioramento della performance che può essere ottenuto usando una qualsiasi modalità di esecuzione più veloce è comunque legato dalla frazione di tempo in cui quella modalità può essere usata.

Essenzialmente quindi devo cercare di capire quali funzioni vengono eseguite più spesso, in maniera da aumentare il più possibile la frazione di codice ottimizzato, cosicché lo **speedup globale** sia più alto possibile, speedup globale che è così calcolato:

$$\text{Speedup}_{\text{global}} = \frac{T_{\text{exe old}}}{T_{\text{exe new}}} = \frac{1}{(1 - \text{Fraction}_{\text{improved}}) + \frac{\text{Fraction}_{\text{improved}}}{\text{Speedup}_{\text{improved}}}} \leq \frac{1}{1 - \text{Fraction}_{\text{improved}}}$$

Lo **speedup globale massimo** è calcolabile abbastanza immediatamente come segue, il denominatore corrisponde alla percentuale di codice non parallelizzabile:

$$\frac{1}{(1 - \text{Fraction}_{\text{improved}})}$$

Va da se che quindi lo **speedup globale** sarà sempre minore di 1 su la frazione non migliorabile di codice.

Tempo come misura di performance

Essenzialmente quindi il tempo di esecuzione è la misura della performance di un computer.

Abbiamo vari tempi:

- **response time:** rappresenta la latenza per compiere una task includendo anche gli accessi alle memorie e I/O.
- **cpu time:** non include il tempo per I/O o esecuzione di altre task, comprende lo user time + cpu system time, unità di misura [secondi/task].

$$CPU_{\text{time}} = CPU_{\text{clock cycle x task}} * \text{clock cycle time} = \frac{CPU_{\text{clock cycle x task}}}{\text{clock frequency}}$$

Altro modo di definire il CPU time è utilizzando il CPI(clock cycle x instruction)che è così ottenuto:

$$CPI = \frac{CPU_{\text{clock cycle x task}}}{\# \text{istruzioni}}$$

$$CPU_{\text{time}} = \# \text{istruzioni} * CPI * \text{clock cycle duration} = \frac{\# \text{istruzioni} * CPI}{\text{clock freq}}$$

Bisogna comunque ricordarsi che il CPU time dipende da 3 parametri:

- **clock cycle(frequenza):** che dipende dalla tecnologia e organizzazione dell'hardware.
- **cpi:** che dipende dall'organizzazione e instruction set
- **numero di istruzioni:** che dipende dall'instruction set architecture e dalla tecnologia del compilatore

Se voglio andare a vedere il numero totale di cicli di clock di un certo programma devo utilizzare la seguente formula:

$$CPU_{\text{clock cycles}} = \sum_{i=1}^n (CPI_i * I_i)$$

dove i componenti sono:

- **I_i** = numero di volte che l'istruzione *i* ha eseguito in una task
- **CPI_i** = numero medio di cicli di clock spesi in un'istruzione generica

Ne risulta quindi che possiamo scrivere il tempo di cpu come:

$$CPU_{\text{time}} = \sum_{i=1}^n (CPI_i * I_i) * T_{\text{clock}} = CPU_{\text{clock cycles}} * T_{\text{clock}}$$

Mips e Gips

Quando si misurano le performance di un calcolatore è facile incontrare le seguenti misure:

- **Mips:** milioni di istruzioni per secondo
- **Gips:** miliardi di istruzioni per secondo

$$MIPS = \frac{\# \text{istruzioni}}{\text{exe time} * 10^6}$$

$$GIPS = \frac{\# \text{istruzioni}}{\text{exe time} * 10^9}$$

Sono facili da capire, dato che misurano la frequenza delle operazioni per unità di tempo, semplicemente una macchina che ha un alto Mips vuol dire che è molto veloce.

Presentano però diversi problemi tra cui il fatto che è una valore dipendente dall'Instruction Set, quindi macchine con instruction set diversi sono difficili da confrontare, inoltre varia in base al programma considerato e infine può variare in maniera inversamente proporzionale alla performance.

Presentano un problema però, non riesce a distinguere le istruzioni, potrebbero essere istruzioni semplici o complesse

Gflops

Rappresenta miliardi di istruzioni floating point per secondo, serve per misurare la performance in relazione a operazioni con floating point, non può quindi essere usata in altri contesti

$$GFLOPS = \frac{\# \text{operazioni Fp in program}}{\text{exe time} * 10^9}$$

Essendo basata su operazioni, piuttosto che su istruzioni, è una buona misura per comparare elementi tra differenti architetture, considerando l'ipotesi che lo stesso programma su diverse architetture, può eseguire diverse istruzioni in base all'ISA ma sicuramente performerà lo stesso numero di operazioni FP.

Ipotesi che però non regge, infatti le operazioni FP non sono consistenti tra architetture diverse, inoltre i gflops cambiano anche cambiando il rateo tra operazioni intere e FP, ma anche cambiando il mix tra FP veloci e lente.

Introduciamo quindi i GFLOPS Normalizzati, in cui associamo un bias ad ogni operazione FP, che ne rappresenta il peso:

Real operations	Bias
ADD, SUB, COMPARE, MULTI	1
DIVIDE, SQRT	4
EXP, SIN, ...	8

Quindi per esempio un frammento di codice che ha rispettivamente ADD → DIV → SIN è calcolato avendo 13 operazioni FP normalizzate.

Prospettive sulla parallelizzazione

Tra i vari casi d'impiego della computazione parallela abbiamo ad esempio la simulazione delle correnti oceaniche, modellata come una griglia bidimensionale che discretizza spazio e tempo, troviamo inoltre la simulazione dell'evoluzione di una galassia o per concludere il rendering di varie scene con ray tracing.

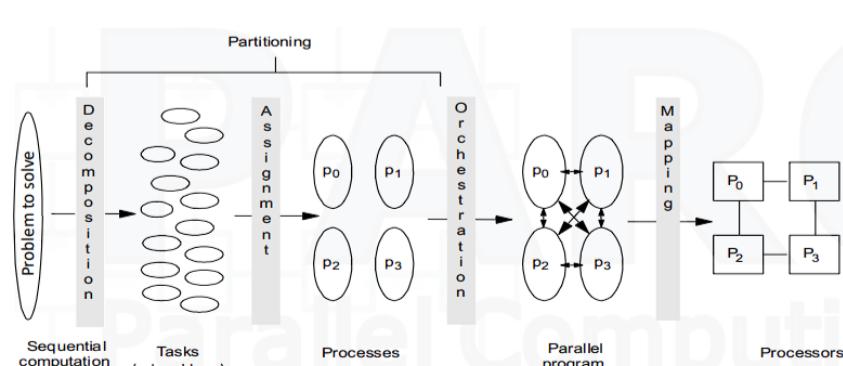
Ricordiamo alcuni concetti utili:

- **task:** pezzo di lavoro arbitrario in una computazione parallela (pezzo di codice)
- **process:** pezzo di codice (task) in esecuzione
- **processor:** entità fisica su cui il processo esegue

Spesso però i calcoli non sono indipendenti, deve esserci sincronizzazione tra gli effetti, se volessi sincronizzazione massima mi ridurrei ad avere sequenzialità senza aver speed up

Step per creare un programma parallelo

- Scomposizione di una computazione in task
- Assegnamento di task ai processi
- Orchestrione degli accessi ai dati, comunicazione e sincronizzazione
- Mapping dei processi ai processori



Prima di passare a queste 4 fasi bisogna ovviamente capire il problema e il programma, se questo non fosse parallelizzabile non avrebbe senso perderci tempo. Ad esempio un problema non parallelizzabile è Fibonacci, perché nel momento in cui devo calcolare un risultato questo dipende dai risultati precedenti.

Cosa importante da fare quando capiamo un problema è trovarne gli **hotspot**, ossia quei punti maggiormente eseguiti, lo si fa con tool di profiling e di performance analysis.

Altri elementi da identificare sono i **bottleneck** ossia aree di codice che sono molto lente e che di conseguenza rallentano globalmente il programma(ad esempio accesso in memoria I/O)

Troviamo anche gli **inibitori al parallelismo** ad esempio uno dei più comuni è la dipendenza dei dati

Scomposizione

Si identifica la concorrenza e si decide a che livello sfruttarla, si divide la computazione in task da dividere in vari processi, essenzialmente abbastanza task da tenere i processi occupati ma non troppi altrimenti otteniamo il risultato inverso. La responsabilità del processo è del programmatore.

Ci sono due modi principali per partizionare che sono:

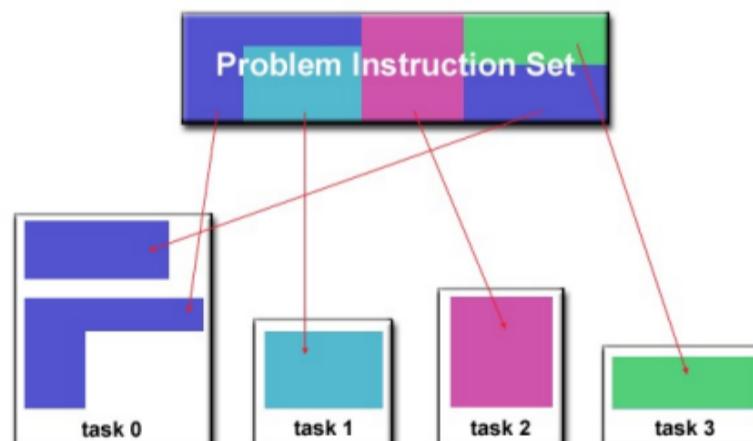
Scomposizione di dominio

Si scompongono i dati associati ad un problema, una volta fatto ciò ogni task parallela lavora su una di queste porzioni di dati.



Scomposizione funzionale

In questa scomposizione ci si concentra di più sulla computazione da eseguire piuttosto che sui dati manipolati da quest'ultima. Quindi il problema è scomposto sulla base del lavoro che deve essere fatto e quindi ogni task avrà una parte del lavoro totale da eseguire



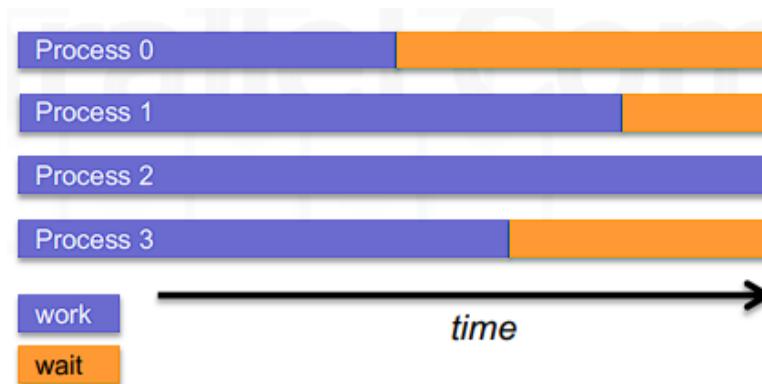
In genere comunque il responsabile è sempre il programmatore che riesce a farlo molto meglio dei tool automatici

Assegnamento

Si va a specificare il meccanismo per dividere le task tra vari processi, sicuramente l'obiettivo principale è avere un carico di lavoro bilanciato tra le task, riducendo le comunicazioni e i costi di gestione, è quindi importante fare load balancing.

Load Balancing

Si riferisce alla pratica di distribuire le task tra i processi in maniera che tutti i processi rimangano impegnati per tutto il tempo dell'esecuzione, è importante per questioni di performance, difatti se tutti i processi sono soggetti a un punto di sincronizzazione, il task più lento di tutti determinerà la performance generale



Per riuscire ad avere load balance si può:

- Partizionare equamente il lavoro(task) per ogni processo:
 - per operazioni su array/matrici dove ogni processo lavora similmente agli altri, distribuire tra i processi pezzi pari.
 - per iterazioni loop distribuire il loop equamente tra i vari processi
 - se sta venendo usato un mix eterogeneo con caratteristiche varie, usare performance analysis tool per scovare eventuali imbalances
- Usare assegnamento del lavoro dinamico, utile perché:
 - Certe classi di problemi risultano avere del load imbalance anche se i dati sono distribuiti equamente tra i processi
 - array sparsi: alcuni processi possono avere dati su cui lavorare, altri magari trovano solo "zeri".
 - quando il lavoro di ogni processo sarà intenzionalmente variabile o difficile da predire, potrebbe risultare utile usare un approccio **scheduler-task pool** dove quando un processo termina il suo lavoro, viene messo in coda per ottenere un nuovo lavoro

Granularità

Serve per misurare il rateo computazione/comunicazione, sappiamo che può essere:

- fine: poco lavoro tra i vari eventi di comunicazione, facilita il load balancing e rappresenta un'occasione mancata per migliorare le performance, se è troppo fine è possibile che l'overhead introdotto per la comunicazione e sincronizzazione diventi più alto di quanto effettivamente non sia lunga la computazione
- grossa: grosse quantità di lavoro compiute tra le varie comunicazioni, implica una migliore possibilità di aumentare le performance, più difficile fare del load balancing efficientemente.

La granularità migliore da usare dipende sia dall'algoritmo che dall'hardware.

Orchestrazione

Mira a strutturare la comunicazione, implementare la sincronizzazione e organizzare strutture dati e schedulare task nel tempo.

L'obiettivo è quello di ridurre i costi di comunicazione e sincronizzazione, schedulare task per soddisfare le dipendenze presto e ridurre l'overhead della gestione del parallelismo.

Comunicazione

La comunicazione tra task dipende dal problema che affrontiamo, può succedere che non serva la necessità di scambiare dati tra task, succede spesso ad esempio nei problemi "imbarazzantemente paralleli" che sono molto diretti e hanno delle piccole comunicazioni tra task. Può invece succedere che serva come è nella maggior parte dei casi.

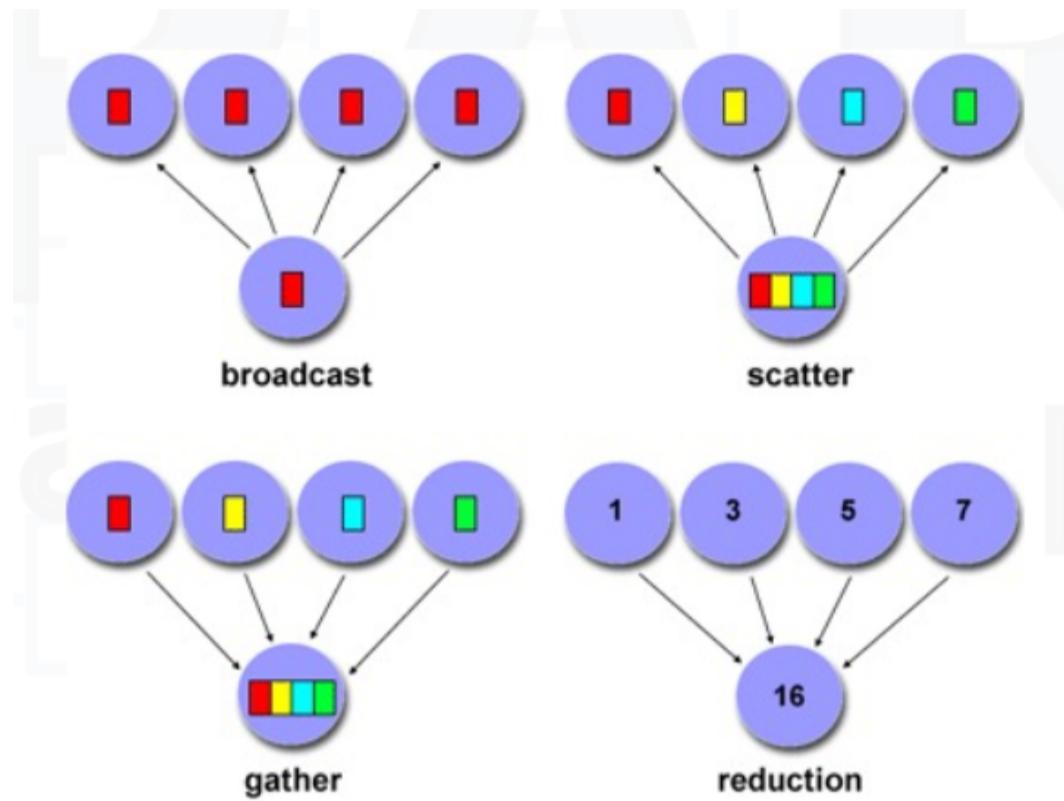
Quando andiamo a progettare la comunicazione tra task dobbiamo comunque considerare varie cose tra cui:

- **Costo della comunicazione:** virtualmente implica sempre overhead, cicli e risorse che potrebbero essere usate per computazioni vengono usati per trasmettere dati, spesso ci sono delle sincronizzazioni e ciò implica altro tempo perso e infine si può saturare la banda.
- **Latenza vs Bandwidth:**
 - latenza: tempo per mandare 0 byte da A a B
 - bandwith: quantità di dati trasmissibile per unità di tempo

Mandare piccoli messaggi può far sì che la latenza domini l'overhead della comunicazione, spesso è più efficiente impacchettare messaggi piccoli in messaggi più grandi incrementando ovviamente la bandwidth della comunicazione.

- **Visibilità delle comunicazioni:** con il modello Message passing le comunicazioni sono esplicite e visibili, con il data parallelle le comunicazioni sono trasparenti, particolarmente sulle architetture a memoria distribuita
 - **Comunicazioni sincrone e asincrone:** possono essere sincrone o anche asincrone e se asincrone possono essere bloccanti o non bloccanti.

- **Scope della comunicazione:** sapere quale task comunica con quale altra non è banale durante la progettazione del codice parallelo, i seguenti scope possono essere sia sync che async:
 - **Point to Point:** due task che si scambiano i dati, una produce e manda, l'altra riceve e consuma
 - **Collective:** più di due task comunicano, spesso appartenenti a un gruppo comune o collettivo



- Broadcast: manda a tutti lo stesso dato
- Gather: raccoglie informazioni da tutti
- Scatter: manda a tutti dati diversi
- Reduction: risultato di operazioni su dati diversi

Overhead e complessità della comunicazione spesso sono molto pesanti

- Tipi di sincronizzazione
 - **barrier:** serve per far attendere fino a che nprocessi arrivano a un dato punto
 - **lock/semafori:**
 - **operazioni di comunicazioni sincrone:** sono coinvolti sottoinsiemi di processi, vige il concetto di producer-consumer

Mapping

Bisogna considerare due aspetti:

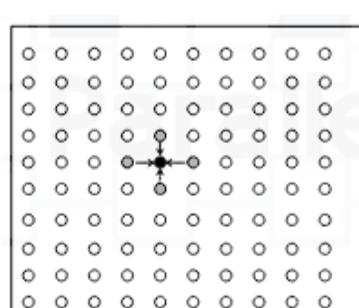
- Quali processi runnano su quale processore?
- Più processi runnano sullo stesso processore?

Gli obiettivi sono:

- alte performance
- ridurre uso di risorse e costo

Esempio: Risolutore di equazioni iterativo

Per fare l'update dei punti nella griglia si usa la formula:

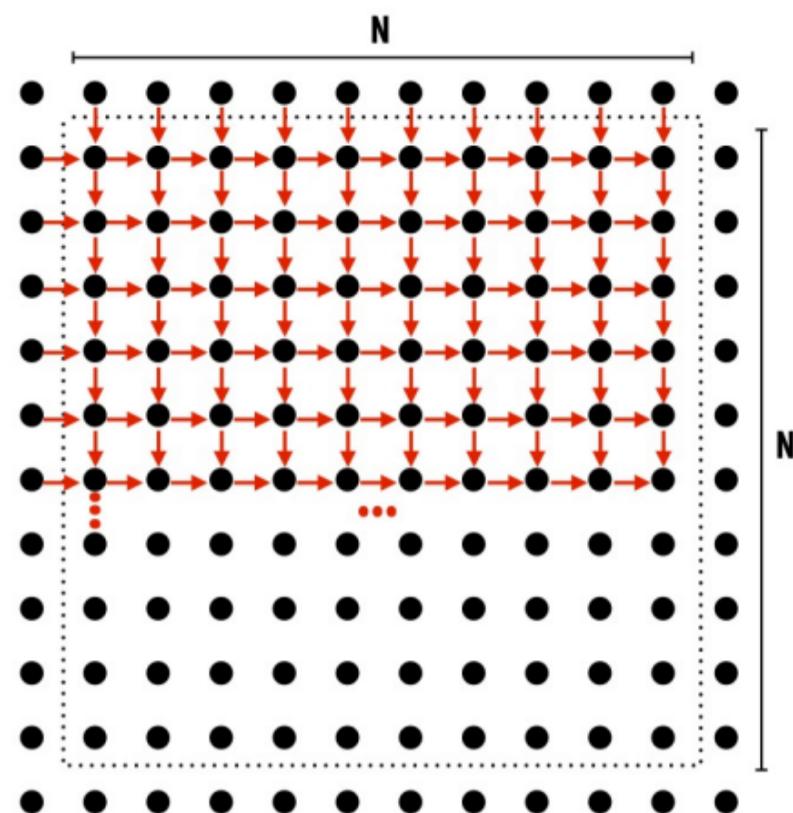


$$A[i, j] = 0.2 * (A[i, j] + A[i, j - 1] + A[i - 1, j] + A[i, j + 1] + A[i + 1, j])$$

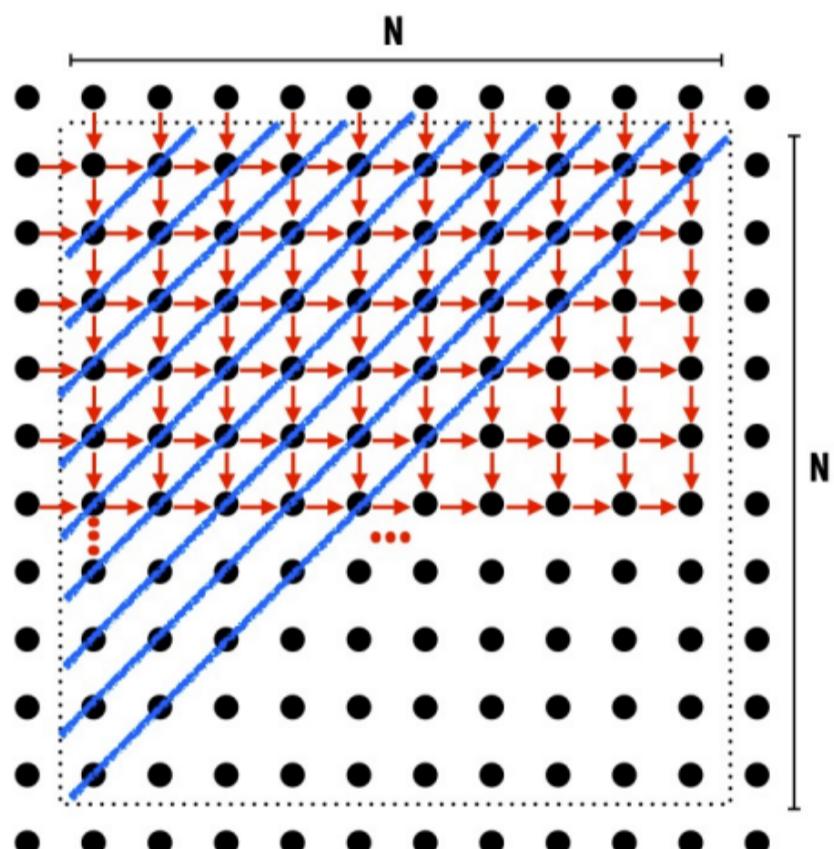
Si effettuano iterazioni finché non giungo a convergenza

Per capire se possiamo parallelizzare l'algoritmo sequenziale che lavora su questa griglia, iniziamo con il capire quali sono le dipendenze:

- Ogni elemento in una riga dipende dall'elemento sulla sinistra
- Ogni riga dipende dalla riga precedente

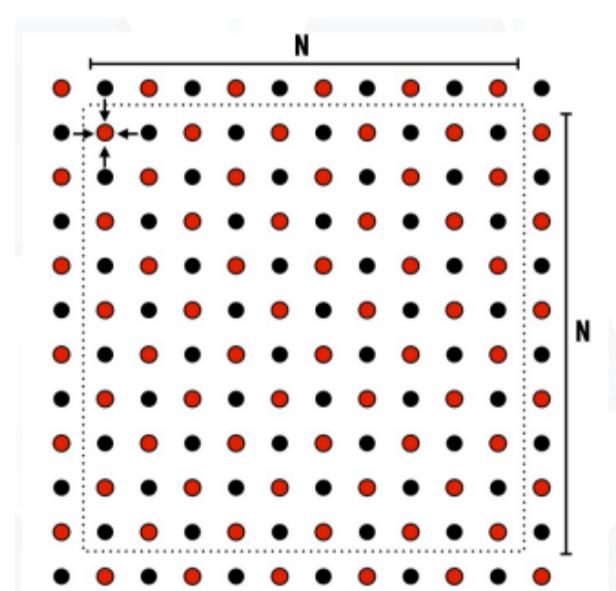


Una cosa che possiamo subito dire è che l'algoritmo così com'è non si riesce a parallelizzare, notiamo però che se consideriamo le diagonali, queste non hanno dipendenze rispetto a niente, quindi potrebbe risultarne un buon parallelismo. Considerare le diagonali però non basta, non ottengo poi troppo parallelismo e ci sono frequenti sincronizzazioni alla fine di ogni diagonale.



Un'idea che possiamo considerare è quella di cambiare totalmente l'algoritmo in un algoritmo che si adatta meglio al parallelismo

Grid solver red black, considero lo stesso funzionamento ma divido i nodi in rossi e neri, in maniera che quelli rossi dipendano da quelli neri e viceversa



Scomposizione: dividiamo la tabella in blocchi

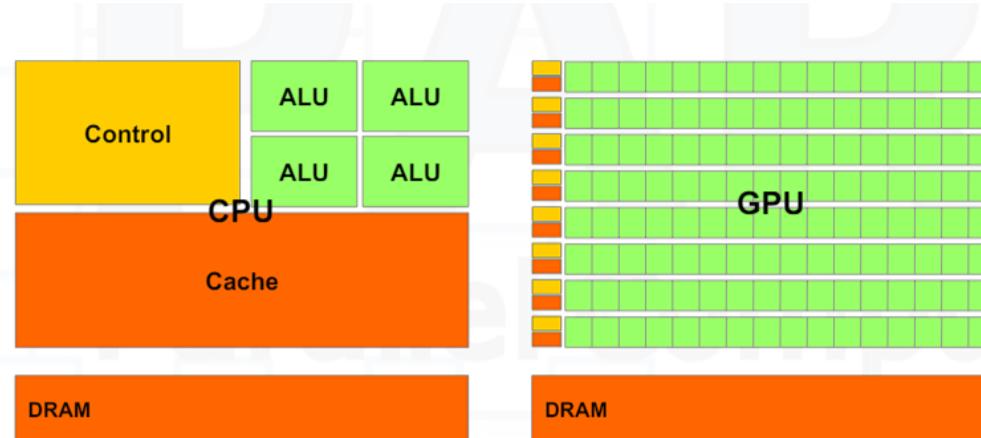
Assegnamento: potremmo decidere di assegnare gruppi di righe a ogni thread oppure di assegnare una riga per ogni thread, tutto ciò dipende dal sistema in cui facciamo girare il programma.

Comunicazione: bisogna tenere a mente che in entrambi i casi almeno una riga deve essere scambiata tra thread perché c'è della dipendenza

Introduzione alle GP-Gpu(General Purpose-Graphics Processing Unit)

Sono schede acceleratrici lanciate per fare elaborazioni di image processing(dati su matrice), note per avere molti thread.

Architettura CPU vs GPU



le ALU sono quelli che chiamiamo core

CPU

- tanta cache per abbreviare i tempi di accesso in memoria che portano via la maggior parte del tempo nella computazione
- il controllo è sofisticato, fa branch prediction e data forwarding (ottimizzazione nella pipeline) per ridurre la latenza dei dati
- le ALU sono poche ma potenti a tanti hertz per diminuire la latenza delle operazioni

GPU

- hanno meno cache perché ci si concentra sul memory throughput, in genere più utili per il sequenziale che non per il parallelo
- il controllo è semplice non fa branch prediction e data forwarding.
- tante ALU ma meno potenti (per ridurre il consumo energetico, se aumento di poco la frequenza il consumo cresce di molto → exp), pesantemente pipelined
- Per tollerare la maggiore latenza è ovviamente richiesto un altissimo numero di threads

In conclusione non è detto che la GPU sia migliore, sicuramente la combinazione tra le due cose da risultati migliori. Infatti se pretendo di far girare un codice fortemente sequenziale su una GPU non avrò alcun vantaggio, anzi.

Usiamo la CPU per le parti in cui la latenza è importante, e la GPU per le parti in cui è importante il throughput

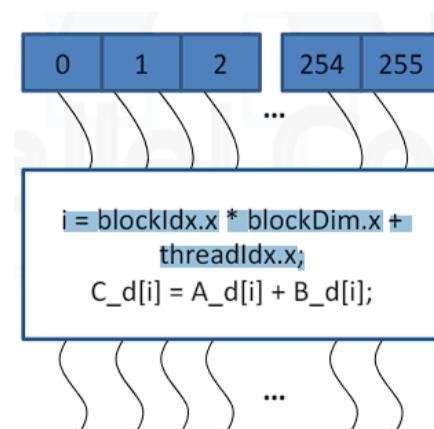
Introduzione a CUDA C

È il linguaggio inventato da NVIDIA per programmare GPU.

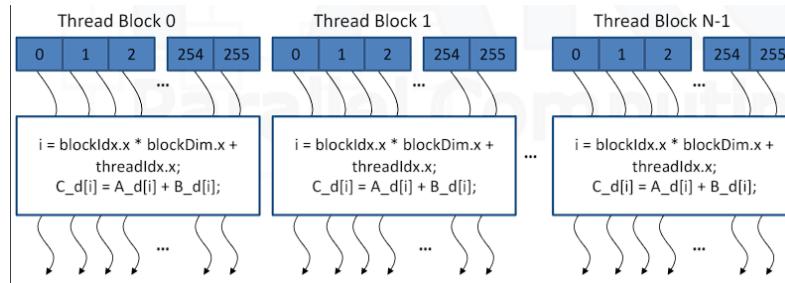
Chiameremo la CPU **host** e la GPU **device**, brevemente avremo che il programma, che come sappiamo parte dal main, inizierà la sua esecuzione in CPU e alla chiamata di una funzione **kernel** comincerà la sua esecuzione parallela su GPU, per poi ritornare su CPU e via così.

Un **CUDA kernel** è eseguito da una grid (array) di thread, tutte le thread eseguono lo stesso codice che è scritto nel kernel, ovviamente bisognerà identificare ogni thread, in modo che ognuna esegua lo stesso codice in maniera "diversa" rispetto alle altre.

Attenzione: X identifica la colonna, Y la riga di un thread, ovviamente anche di un blocco



Una precisazione importante da fare è che le thread non sono aggregate tutte insieme nella grid, bensì sono divise in blocchi, essenzialmente quindi abbiamo una grid di blocchi di thread, questi hanno delle proprietà molto utili che aiutano il parallelismo, ad esempio riescono a far comunicare e sincronizzare le thread dello stesso blocco molto bene, questo avviene perché c'è una memoria condivisa in ogni blocco, va da sé che thread in blocchi diversi difficilmente cooperano.



BlockIdx e ThreadIdx

Abbiamo quindi:

- Thread: singola unità d'esecuzione
- Blocco: un gruppo di thread che può essere mono, bi o tridimensionale
- Grid: un insieme di blocchi, la griglia può essere mono, bi o tridimensionale

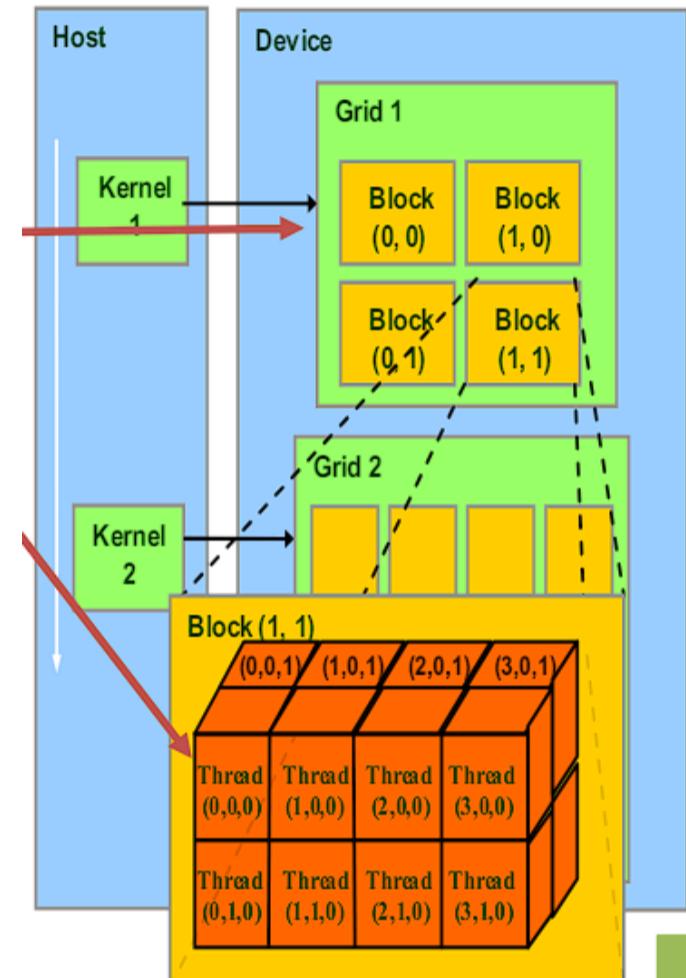
Vediamo ora come identificare le thread, ogni thread è spesso identificata considerando uno spazio fino a 3 dimensioni $\rightarrow (x, y, z)$.

Consideriamo di avere un'array di 128 elementi da processare, a questo punto la struttura è monodimensionale, assegnerò una thread per ciascun elemento avendo così 128 thread, e queste le identificherò su una dimensione usando solo la dimensione (x).

Se consideriamo invece di avere una matrice di 20×20 , ossia di 400 elementi, quello che posso fare è considerare di avere blocchi bidimensionali di thread, dove identifico le thread su due dimensioni (x,y), in maniera tale da processare $\text{mat}[i,j]$ con $(\text{thread}.x, \text{thread}.y)$.

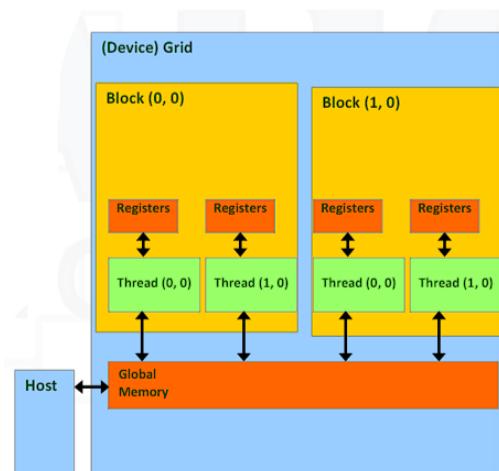
Via così per la 3^a dimensione considerando come esempio un cubo.

Lo stesso discorso vale per i blocchi che come le thread possono essere identificati a loro volta utilizzando fino a 3 dimensioni (x,y,z).



CUDA API per Device Memory Management

Consideriamo innanzitutto la struttura generale di un device abilitato a CUDA:

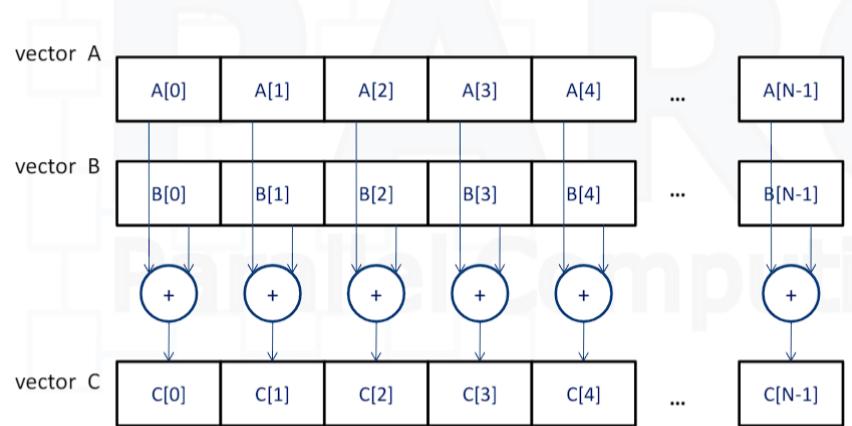


Tra le api più note che incontriamo abbiamo sicuramente:

- **cudaMalloc()**: alloca oggetti nella memoria globale del device, necessita ovviamente di due parametri:
 - l'indirizzo di un puntatore all'oggetto allocato
 - size of dell'oggetto allocato

- **cudaFree()**: libera l'oggetto allocato nella memoria globale del device, necessita di un parametro:
 - puntatore all'oggetto
- **cudaMemcpy()**: trasferisce i dati in memoria da un'indirizzo CPU/GPU a un'altro GPU/CPU, necessita di quattro parametri:
 - puntatore alla destinazione
 - puntatore alla sorgente
 - numero di bytes da copiare
 - direzione del trasferimento (device→host oppure host→device)

Esempio somma tra 2 vettori



Per questo tipo di operazione abbiamo il seguente codice sequenziale:

```
// Compute vector sum C = A+B
void vecAdd(float* A, float* B, float* C, int n)
{
    for (i = 0, i < n, i++)
        C[i] = A[i] + B[i];
}

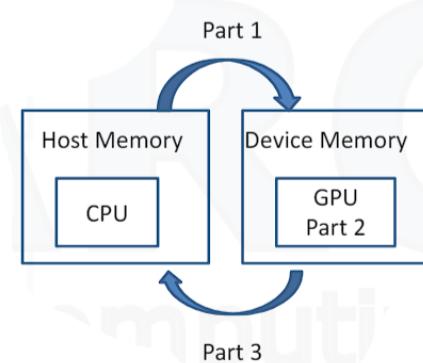
int main()
{
    // Memory allocation for A_h, B_h, and C_h
    // I/O to read A_h and B_h, N elements
    ...
    vecAdd(A_h, B_h, C_h, N);
}
```

A_h → vettore A che c'è su host(CPU)

Ora parallelizziamo, incontreremo sempre 3 fasi:

1. **allocazione della device memory per A,B e C**
2. **kernel lancia il code**
3. **copia il risultato C dalla device memory alla host memory**
4. **Liberazione della memoria del device**

Il risultato di questi step sarà un codice come segue:



```

void vecAdd(float* A, float* B, float* C, int n)
{
    int size = n * sizeof(float);
    float* A_d, B_d, C_d;

    1. // Transfer A and B to device memory
    cudaMalloc((void **) &A_d, size);
    cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &B_d, size);
    cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

    // Allocate device memory for
    cudaMalloc((void **) &C_d, size);

    2. // Kernel invocation code – to be shown later
    ...
    3. // Transfer C from device to host
    cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
    // Free device memory for A, B, C
    4. cudaFree(A_d); cudaFree(B_d); cudaFree(C_d);
}

```

Host Code

```

// Compute vector sum C = A + B
// Each thread performs one pair-wise addition
__global
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}

int vectAdd(float* A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256), 256>>>(A_d, B_d, C_d, n);
}

```

Device Code

Host Code

- Il kernel viene lanciato con una configurazione specifica di griglia e blocchi
 - una griglia di $\text{ceil}(n/256)$
 - blocchi di 256 thread
 - blocchi

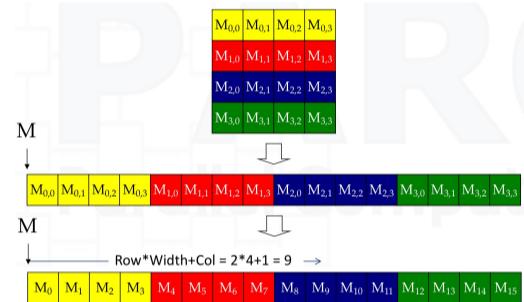
N.B. spesso i blocchi sono scelti di dimensione multipla di 32, che è la dimensione di una warp, l'unità di schedulazione di una GPU

Definizione di funzioni

- `_device_` type `fun()` → funzione per device chiamata da device
- `_global_` type `fun()` → funzione per device chiamata da host
- `_host_` type `fun()` → funzione per host chiamata da host

Si può comunque usare `_device_` e `_host_` insieme

Lavorare con matrici in C/C++ - Row Major Layout



Streaming Multiprocessor e Single Processor

Streaming Multiprocessor (SM): è una componente fondamentale dell'architettura delle GPU NVIDIA. È l'unità principale di elaborazione che gestisce l'esecuzione dei thread nei programmi CUDA. L'SM è responsabile di programmare e gestire l'esecuzione dei thread. Gestisce gruppi di 32 thread chiamati warp. Ogni SM contiene diverse unità di elaborazione più piccole, chiamate Single Processor (SP) che eseguono operazioni aritmetiche e logiche. Gli SM hanno una memoria condivisa che i thread al loro interno possono usare per condividere dati. E hanno anche dei registri usati per memorizzare le variabili locali dei thread e una cache per diminuire la latenza degli accessi in memoria globale.

Single Processor (SP): anche conosciuto come **Core CUDA**, è una singola unità di elaborazione all'interno di uno SM. È responsabile dell'esecuzione delle istruzioni per i thread. Usano modello (Single Instruction Multiple Data).

Relazione tra i due: Un SM contiene molti SP. Ad esempio, in alcune architetture recenti, un SM può contenere 64 SP o più. Gli SP all'interno di uno SM eseguono i thread appartenenti alle warp. I thread eseguiti dagli SP possono sincronizzarsi e condividere dati attraverso la memoria condivisa dell'SM.

Funzionamento

Quando un kernel CUDA è lanciato, i blocchi di thread sono assegnati agli SM disponibili. All'interno di ogni SM, i thread sono organizzati in warp. Gli SP all'interno di uno SM eseguono i thread delle warp in parallelo.

Esempio Pratico

Consideriamo una GPU con 4 SM, dove ogni SM ha 64 SP. Quando un kernel CUDA viene lanciato:

- Assegnazione dei Blocchi:** I blocchi di thread sono distribuiti tra i 4 SM disponibili.
- Esecuzione nei Warp:** Ogni SM esegue i thread in warp di 32 thread. I 64 SP all'interno di un SM possono eseguire simultaneamente due warp (2 warp da 32 usano 64 Single Processor).
- Condivisione dei Dati:** I thread di un blocco, eseguiti all'interno di uno SM, possono condividere dati tramite la memoria condivisa e sincronizzarsi con `__syncthreads()`.

Scalabilità trasparente

Partiamo con il ricordare che i thread in uno stesso blocco condividono i dati e si sincronizzano (tramite ad esempio con la `__syncthreads()` che funziona a livello di blocco) mentre fanno il loro lavoro condiviso, ciò non vale invece per thread di blocchi diversi, infatti ragionando a un livello superiore ogni blocco esegue in un ordine a caso rispetto agli altri blocchi e possono comunicare e sincronizzarsi solo tramite la memoria globale che come sappiamo è molto lenta.

Parliamo ora di **scalabilità trasparente**, sappiamo che l'hardware è libero di assegnare blocchi a ogni processore in qualsiasi momento e in maniera automatica, più precisamente se parliamo di GPU, per processore intendiamo uno streaming multiprocessor (SM),

all'interno del quale ci sono più single processor(SP).

Quando ciò avviene è importante notare che ci sono 3 vincoli importanti che vanno rispettati:

- non può succedere che ci siano più di 8 blocchi mappati su un SM Fermi(16 se parliamo di serie Kepler, 32 per Pascal).
- Possono esserci al massimo 1536 thread in un SM serie Fermi, 2048 se serie Kepler o Pascal.
- Non possono esserci più di 1024 thread per blocco(cuda), pascal 1024

Warp

Fino ad ora sapevamo che i thread eseguiti sono mantenuti in blocchi che a loro volta sono parte di una grid, in realtà a tempo di esecuzione succede che lo scheduler mette in esecuzione dei **warp** che rappresentano l'unità di scheduling. I warp sono composti da gruppi di 32 thread che vengono mandati in esecuzione sugli SM. Ci potremmo chiedere cosa succede quindi a blocchi di più di 32 ~~bit~~ ^{thread}, semplicemente questi vengono divisi per 32 e messi in esecuzione in N warps $\rightarrow 256/32 = 8$, che sono i warp che vengono messi sull'SM. Le prestazioni massime le otteniamo quando un warp è formato da 32 thread appunto, se ne ho di meno perdo sempre un po' di potenza di calcolo.

Control Flow

Tra le istruzioni che possiamo eseguire, abbiamo anche quelle di control flow, come il jump e sono dette anche di branching. Tra le principali preoccupazioni sulla performance legate al branching c'è la **divergenza dei warp** che è un problema che si verifica quando i thread all'interno di un warp prendono percorsi di esecuzione diversi a causa di istruzioni condizionali (if-else, switch), il problema è che nelle attuali GPU i diversi percorsi di controllo sono serializzati, quindi abbiamo a prescindere delle thread in un warp che non eseguono perché bisogna aspettare che le altre finiscano il percorso di controllo precedente (quando le thread che eseguono il ramo then stanno andando quelle del ramo else sono inattive e viceversa). Se volessimo portare la divergenza al minimo dovremmo riuscire a introdurre un controllo a livello di warp, quindi thread di un warp nel ramo then e thread di un warp nel ramo else, essendo che così la divergenza si crea a livello di warp saremmo appunto.

Block Granularity: considerazioni su matrix multiplication

Pensando al problema della Matrix Mul, quale dimensioni dei blocchi può tornare più sensata per uno speedup maggiore?

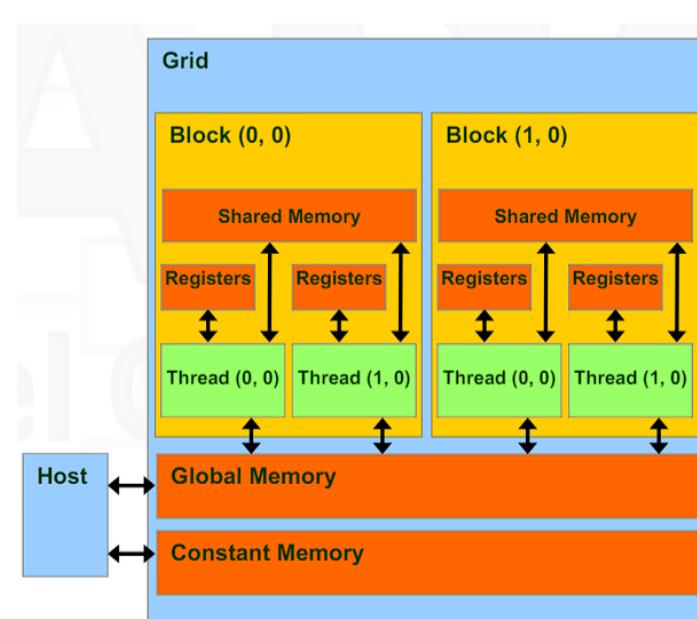
- 8×8: abbiamo 64 thread a blocco, ogni SM può prendere al massimo 1536 ¹⁵³⁶ thread, abbiamo quindi 24 blocchi, sappiamo inoltre che ogni SM può avere al massimo 8 blocchi, quindi arriviamo ad avere solo 512 thread in ogni SM, una quantità troppo piccola per raggiungere grandi speedup
- 16×16: 256 thread per blocco, quindi 6 blocchi, con questa configurazione dovremmo riuscire ad ottenere velocità massima a meno di altre considerazioni riguardanti le risorse non discusse
- 32×32: 1024 thread per blocco e solo un blocco che può stare nell'SM altrimenti supereremmo il vincolo di 1536 thread, ciò significa che usiamo solo 2/3 della capacità di un SM

GPU Memory Model

Vediamo più precisamente lo schema delle memorie presenti in una gpu:

Se parliamo di velocità di accesso in ordine abbiamo, ogni thread impiega:

- 1 ciclo di clock per accedere al registro
- 5 cicli di clock per accedere alla shared memory
- 5 cicli di clock con caching per accedere in sola lettura alla constant memory
- 500 cicli di clock per accedere alla memoria globale



Fondamentale è quindi la shared memory, interpretabile come la "cache" della CPU, ne abbiamo una per ogni blocco, essenziale perché se dovessimo tornare in global memory ad ogni accesso perderemmo troppo tempo.

Come si mettono le variabili nelle varie memorie in CUDA:

Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	thread
<code>__device__ __shared__ int SharedVar;</code>	shared	block	block
<code>__device__ int GlobalVar;</code>	global	grid	application
<code>__device__ __constant__ int ConstantVar;</code>	constant	grid	application

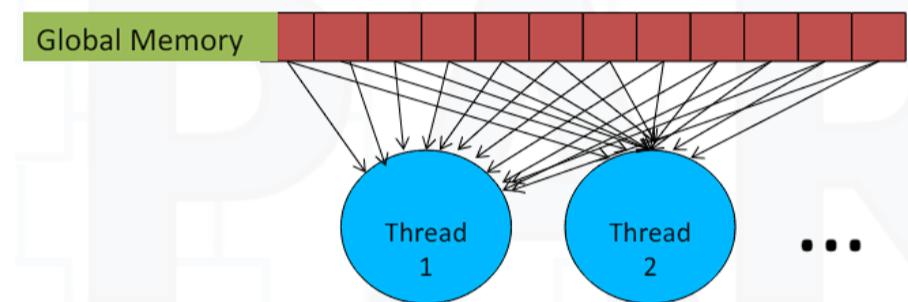
Le variabili usano registri che sono fisici e sono una quantità finita, per questo abbiamo i vincoli da rispettare per ogni architettura.

Per esempio, consideriamo la moltiplicazione tra matrici, per velocizzare il tutto potremmo mettere le due matrici in shared memory, l'unico problema è che per matrici grandi non riusciamo ad avere abbastanza spazio in memoria e quindi l'approccio non funziona.

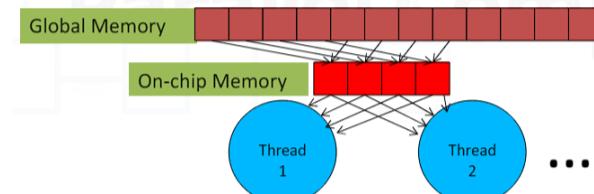
Una strategia comune è quella di tagliare i dati in input per sfruttare a pieno la shared memory. Per "tagliare" intendiamo partizionare i dati in sottoinsiemi che riescano ad entrare nella shared memory. A questo punto quello che si va a fare è gestire ogni partizione dei dati con un blocco di thread nella seguente maniera:

- caricare il subset da global a shared memory usando più threads per sfruttare il parallelismo a livello di memoria
- eseguire la computazione sul sottoinsieme caricato in shared
- copiare i risultati dalla shared a global memory

si passa da una situazione come questa



a una situazione di questo genere



Vediamo da un punto di vista di performance reale cosa succede con la moltiplicazione tra matrici, con e senza shared memory, considerando una scheda GPU con 1 TeraFlops di potenza di calcolo e 150 GB/s di bandwidth di trasferimento dei dati dalla memoria:

- **Senza shared memory**

$$\frac{\# \text{computation}}{\# \text{communication}} = \frac{2}{2} = 1$$

Questo perché per ogni operazione eseguita da un thread, ossia un prodotto e una somma, quindi 2 computazioni, abbiamo 2 accessi in memoria.

A questo punto considerando il fatto che una flops(floating point operation), utilizza 4 B/s di bandwidth, per raggiungere il picco teorico di un teraflops, serve una bandwidth di 4000 GB/s, ovvero 4 B/s * 1 Teraflops. Essendo la larghezza di banda di soli 150 GB/s abbiamo però:

$$\frac{150 \text{ GB/s}}{4} * 1 = 37.5 \text{ GFlops}$$

Per migliorare le prestazioni ciò che dobbiamo fare è diminuire drasticamente il numero di accessi a memoria

- **Con shared memory**

consideriamo 256 thread in ciascun blocco (con dimensione 16×16)

Va a prendersi tutta la mattonella e la carica in shared memory, avrà 256×2 accessi a memoria:

$$\frac{\# \text{computation}}{\# \text{communication}} = \frac{256 * (2 * 16)}{256 * 2} = 16$$

Dove (2*16) sono i calcoli per ciascuna singola thread

Facendo ora il medesimo calcolo:

$$\frac{150 \text{ GB/s}}{4} * 16 = 600 \text{ GFlops}$$

GPU Performance Consideration

Nelle DRAM l'informazione è mantenuta in condensatori che in base alla loro carica rappresentano un bit a 0 o a 1. Capire se quest'ultimo è a 0 o 1 però ha un costo non trascurabile, considerando quindi molti accessi ciò ci porta a un'importante perdita di

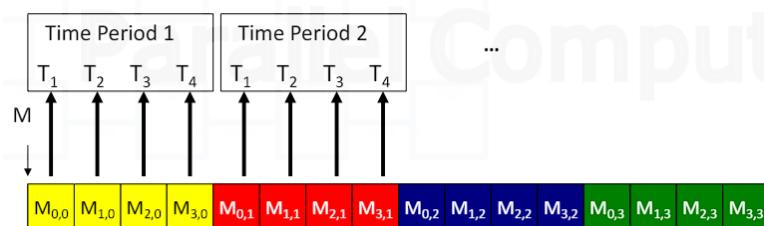
tempo.

Memory Coalescing

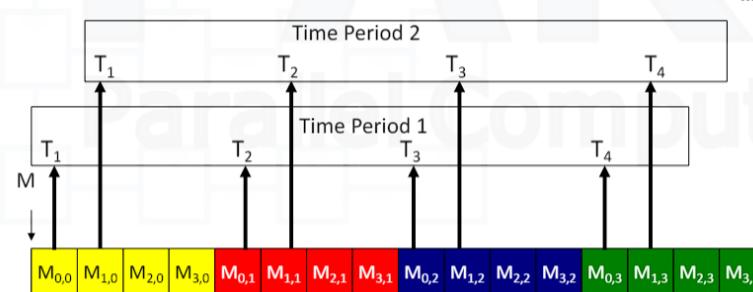
Proprio per il motivo sopra viene introdotto il coalescing, semplicemente se si nota che tutti i thread in un warp accendono a zone di memoria successive (quindi thread 0 accede a n , $i+1$ a $n+1$, $i+2$ a $n+2$ e via così), quello che si fa è recuperare tutto il blocco di memoria acceduto dai thread nel warp dalla memoria globale, così riduciamo da 32 accessi alla memoria per warp ad un unico accesso consolidato alla memoria per warp arrivando al picco massimo di performance.

È importante notare che parliamo di coalescing quando accediamo alla memoria globale. Se come nella moltiplicazione tra matrici, adattiamo la nostra soluzione ad utilizzare la shared memory con il tiling questo concetto viene a mancare.

Esempio di accesso coalescente:



Esempio di accesso non coalescente:



Ripartizione dinamica delle risorse di esecuzione

Tra le risorse di un SM sappiamo che abbiamo: registri, shared mem, thread block slots.

Sappiamo che un device può avere 1536 threads/SM, quindi:

- Se abbiamo blocchi da 512 thread, usiamo 3 blocchi/SM → può andare bene
- Se abbiamo blocchi da 128 thread, usiamo 12 blocchi, ma sappiamo che il massimo è 8 per SM, quindi in questo caso avremmo un sottoutilizzo delle risorse

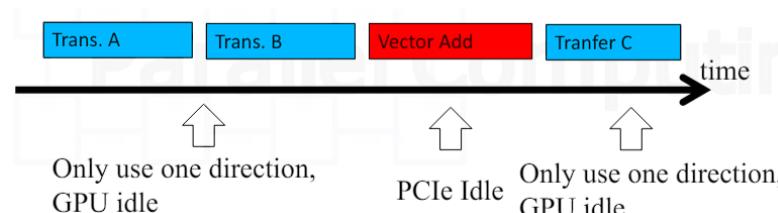
Consideriamo anche 16384 registri/SM, abbiamo che:

- Se il kernel stanzia 10 variabili (32 bit ciascuna, mantenute ciascune di esse in un registro) per thread
 - Se abbiamo blocchi da 256 threads, $256 \times 10 = 2560 \rightarrow$ registri per blocco
 - Abbiamo $1536/256 = 6$ blocchi nell'sm
 - Quindi in totale $6 \times 2560 = 15360$ registri in utilizzo → può andare bene
 - Se aggiungiamo 2 variabili in più per il kernel $\rightarrow 12 \times 256 = 3072 \rightarrow$ registri per blocco
 - Abbiamo quindi $1536/256 = 6$ blocchi nell'sm
 - Quindi in totale ho $6 \times 3072 = 18432$ registri in utilizzo che eccede il vincolo, quindi diventa necessario ridurre i blocchi per rientrare nel vincolo, ci può andare bene ma ora abbiamo un blocco in meno che esegue nell'SM solo per due variabili in più

È importante quindi non usare più variabili di quanto sia veramente necessario.

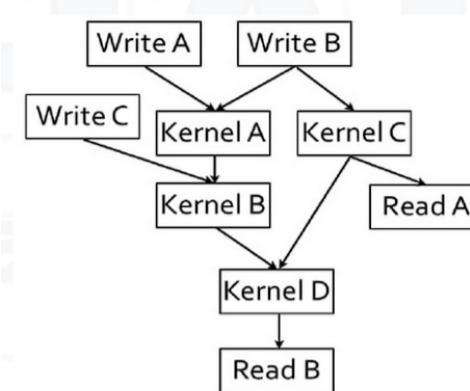
Task parallelism for data transfer

Al momento la maniera in cui abbiamo utilizzato cudaMemcpy serializza il data transfer e computazioni di GPU, come si può vedere prima si carica in memoria tutto ciò che serve, poi si esegue la computazione e successivamente si ricopia in memoria dell'host il risultato.



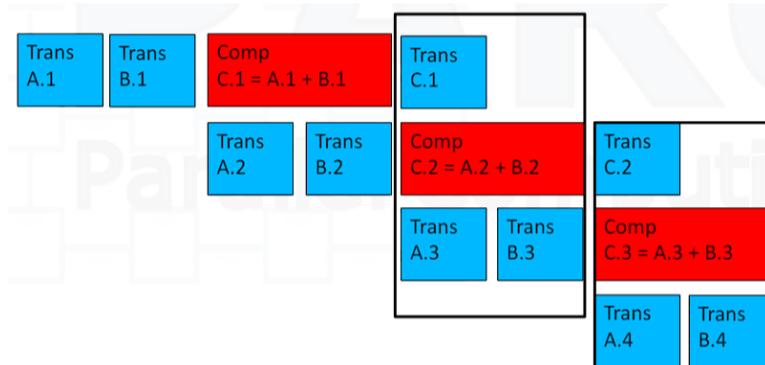
Ciò indubbiamente rallenta notevolmente l'esecuzione generale, osserviamo però che quando mettiamo in coda un kernel riusciamo a capire tutto ciò che deve essere successo prima che questo esegua, riuscendo a specificare un grafo di dipendenza tra le esecuzioni dei kernel e i trasferimenti di memoria

Ad esempio nel grafo delle dipendenze a lato vediamo come il kernel A non può eseguire se non sono state fatte le write A e B, ma può potenzialmente eseguire finché si sta facendo la write C.



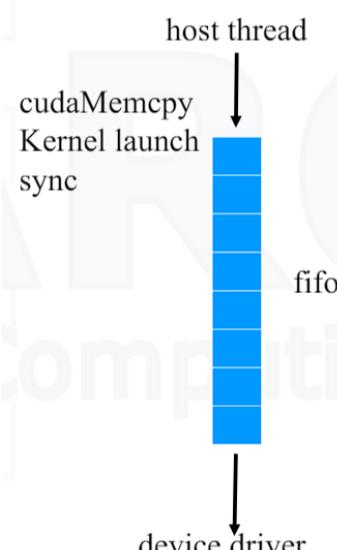
Device Overlap

Alcuni dispositivi CUDA supportano il device overlap, che permette banalmente di sovrapporre l'esecuzione di un kernel a dei trasferimenti dati

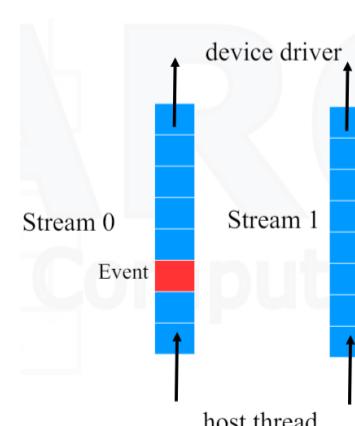


CUDA supporta l'esecuzione parallela di kernel e cudaMemcpy utilizzando gli **streams**, dove ogni stream non è altro che una coda di operazioni e per operazioni in stream differenti possiamo sfruttare il parallelismo detto **task parallelism**

Se vogliamo implementare la pipeline ci servono più stream (oltre allo stream 0 che presente di default), dobbiamo quindi crearne altri.



Da un punto di vista reale quello che succede è descritto nella foto, nel primo stream avrò il primo chunk. Nel secondo l'altro chunk e via così, ovviamente può bastare un for che alterna l'assegnazione dei chunk successivi agli stream.



Vector Add

```

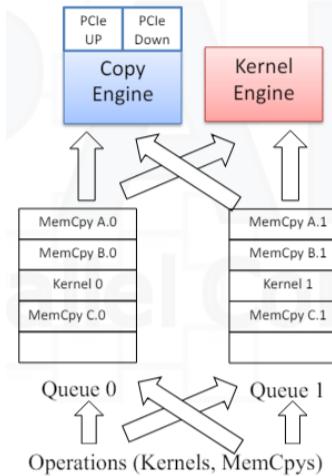
cudaStream_t stream0, stream1;
cudaStreamCreate( &stream0);
cudaStreamCreate( &stream1);
float *d_A0, *d_B0, *d_C0; // device memory for stream 0
  
```

```

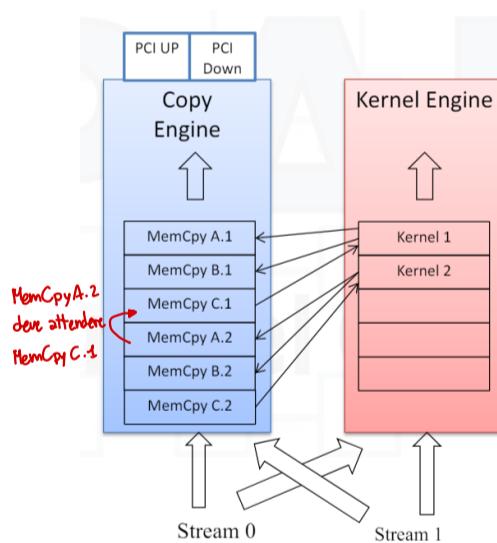
float *d_A1, *d_B1, *d_C1; // device memory for stream 1
// cudaMalloc for d_A0, d_B0, d_C0, d_A1, d_B1, d_C1 go here
for (int i=0; i<n; i+=SegSize*2) {
    //PRIMO CHUNK
    cudaMemCpyAsync(d_A0, h_A+i; SegSize*sizeof(float),..., stream0);
    cudaMemCpyAsync(d_B0, h_B+i; SegSize*sizeof(float),..., stream0);
    vecAdd<<<SegSize/256, 256, 0, stream0>>>(d_A0, d_B0, ...);
    cudaMemCpyAsync(d_C0, h_C+i; SegSize*sizeof(float),..., stream0);
    //SECONDO CHUNK
    cudaMemCpyAsync(d_A1, h_A+i+SegSize, SegSize*sizeof(float),..., stream1);
    cudaMemCpyAsync(d_B1, h_B+i+SegSize, SegSize*sizeof(float),..., stream1);
    vecAdd<<<SegSize/256, 256, 0, stream1>>>(d_A1, d_B1, ...);
    cudaMemCpyAsync(d_C1, h_C+i+SegSize, SegSize*sizeof(float),..., stream1);
}

```

Consideriamo a questo punto la vectorAdd, sappiamo che sicuramente servono in ordine 2 MemCopy → Kernel Exe → MemCopy, quello che succede a livello concettuale è la seguente cosa. Vediamo come nelle code si piazzino le varie attività in ordine, ovviamente come già detto prima con un for si può ciclare affinché ci sia un alternarsi dei chunk nelle code.



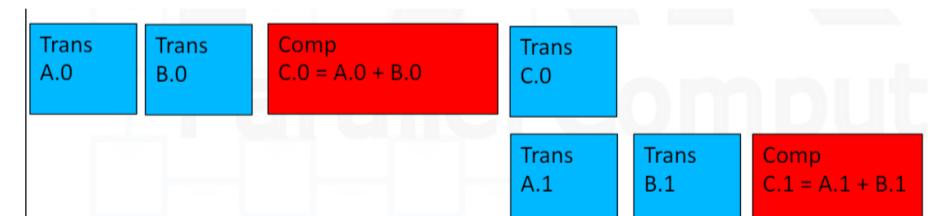
Andando poi a vedere cosa succede nei Copy e Kernel Engine, che gestiscono rispettivamente i trasferimenti dei dati e le esecuzioni dei kernel ci accorgiamo che non c'è bisogno di sincronizzare le task nella pipeline perché si sincronizzano già da sole con delle dipendenze che si creano.



Le operazioni del chunk2 attendono che siano partite tutte quelle del chunk1, non iniziando quindi l'esecuzione prima che inizi il trasferimento del risultato della prima computazione.

Sempre queste dipendenze però nelle architetture dove gli stream sono virtuali e non a livello hardware creano dei problemi, perché a livello hardware avremo poi una sola coda dove vengono raccolte tutte le operazioni sequenzialmente e quindi la pipeline risulta avere un **overlapping parziale**, che non è ciò che desideriamo.

L'unico problema di questa soluzione per stream virtuali quindi è che abbiamo delle dipendenze delle memcp per il kernel1 e via così, non siamo più nella pipeline di prima ma siamo ora in questa situazione:



Se però andiamo ad aggiustare il codice riusciamo a riportarci sulla strada corretta:

```

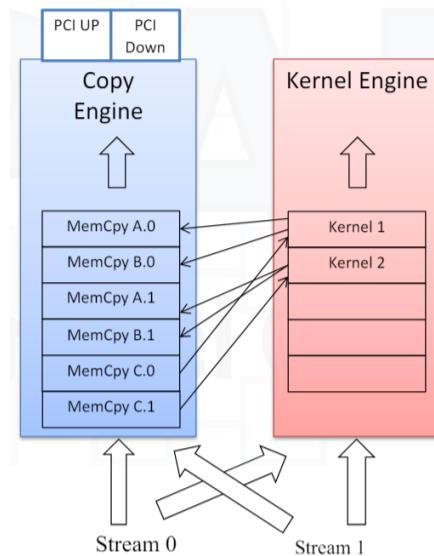
cudaStream_t stream0, stream1;
cudaStreamCreate( &stream0);
cudaStreamCreate( &stream1);
float *d_A0, *d_B0, *d_C0; // device memory for stream 0
float *d_A1, *d_B1, *d_C1; // device memory for stream 1
// cudaMalloc for d_A0, d_B0, d_C0, d_A1, d_B1, d_C1 go here
for (int i=0; i<n; i+=SegSize*2) {
    //CHUNK 1
    cudaMemCpyAsync(d_A0, h_A+i; SegSize*sizeof(float),..., stream0);
    cudaMemCpyAsync(d_B0, h_B+i; SegSize*sizeof(float),..., stream0);
    //CHUNK 2
    cudaMemCpyAsync(d_A1, h_A+i+SegSize; SegSize*sizeof(float),..., stream1);
}

```

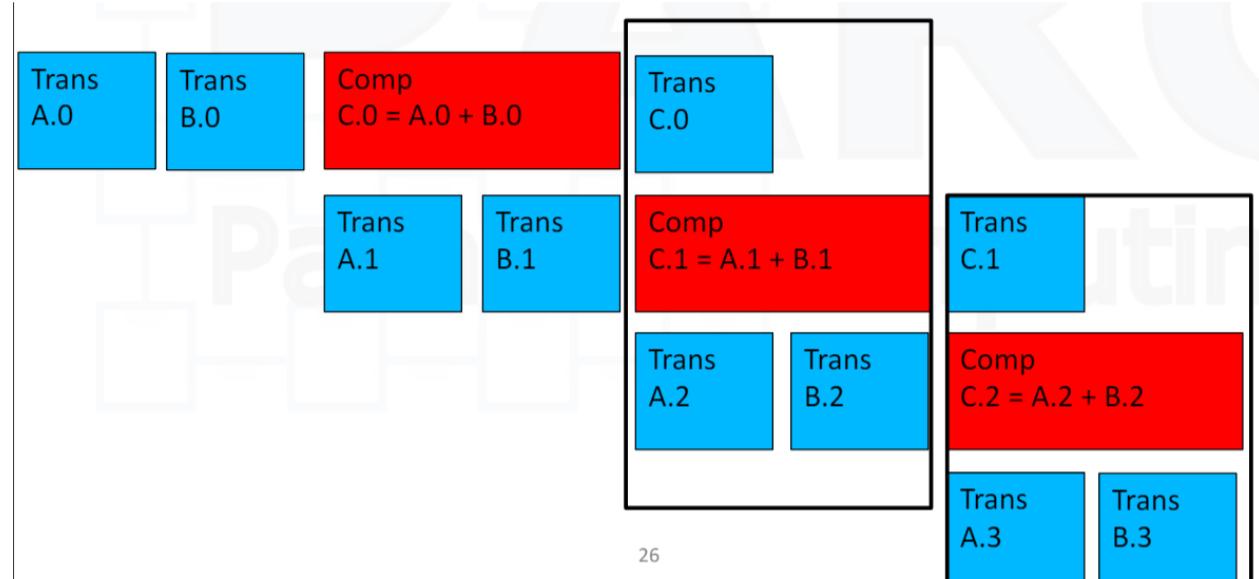
```

cudaMemCpyAsync(d_B1, h_B+i+SegSize; SegSize*sizeof(float),..., stream1);
//EXE 1
vecAdd<<<SegSize/256, 256, 0, stream0>>>(d_A0, d_B0, ...);
//EXE 2
vecAdd<<<SegSize/256, 256, 0, stream1>>>(d_A1, d_B1, ...);
cudaMemCpyAsync(d_C0, h_C+I; SegSize*sizeof(float),..., stream0);
cudaMemCpyAsync(d_C1, h_C+i+SegSize; SegSize*sizeof(float),..., stream1);
}

```



Notiamo che con un semplice cambiamento la pipeline è tornata a una situazione ottimale, perché ora appena finiscono le copy per il kernel1, kernel2 non deve più aspettare che venga fatta la memcpy risultato del kernel1 ma può cominciare le sue memcpy e successivamente l'esecuzione



Se invece parliamo di architetture che implementano la Hyper Queue (presenti dalla versione Kepler in poi), abbiamo proprio degli stream hardware e quindi la situazione cambia, lo si vede facilmente osservando che nelle virtuali il lavoro finisce comunque in una queue sequenzialmente,

implementati
a livello

poi l'esecuzione viene parallelizzato ma non al massimo, come invece avviene nelle hyper queue
che hanno vere e proprie queue hardware multiple



Esempio con Prefix-Scan(13:01)

L'operazione di prefix-scan consiste nell'utilizzo di un operatore associativo e un vettore di N elementi, il risultato che otterremo è:

Inclusive: $[x_0, (x_0 \oplus x_1), \dots, (x_0 \oplus x_1 \oplus \dots \oplus x_{N-1})]$

Exclusive: $[0, x_0, (x_0 \oplus x_1), \dots, (x_0 \oplus x_1 \oplus \dots \oplus x_{N-2})]$

Si può comunque passare da una versione all'altra in maniera banale con degli shift.

Exclusive Scan → inclusive scan: shift dell'array a sinistra di un elemento, aggiunta alla fine di ultimo elemento array + ultimo elemento scan

Inclusive Scan → exclusive scan: shift dell'array a destra di un elemento, e metto 0 a inizio array

È un'operazione molto importante in quanto utilizzata in moltissimi contesti come Sorting, Istogrammi, Moltiplicazione tra matrici sparse, ecc. Il prefix-scan è un'operazione cruciale in vari algoritmi paralleli e la sua implementazione efficiente può migliorare significativamente le prestazioni di sistemi paralleli.

Implementazione sequenziale

Vediamo le implementazioni sequenziali di entrambe:

```
//INCLUSIVA  
Out[0] = In[0];  
for (int i = 1; i < N; ++i)  
    Out[i] = Out[i - 1] + In[i];  
  
//ESCLUSIVA  
Out[0] = 0;  
for (int i = 1; i < N; ++i)  
    Out[i] = Out[i - 1] + In[i - 1];
```

La complessità dei precedenti algoritmi è di $O(n)$, per parallelizzarli però bisogna fare delle modifiche essendo che c'è della dipendenza tra i dati, infatti se osserviamo l' i -esimo output viene calcolato usando l' $(i-1)$ -esimo output

Implementazione Naive parallela

```
for (int level = 0; level < log2(N); level++) {  
    parallel_for ( Vi ∈ N ) {  
        if (i >= 2^level)  
            X[i] = X[i - 2^(level)] + X[i]  
    }  
}
```

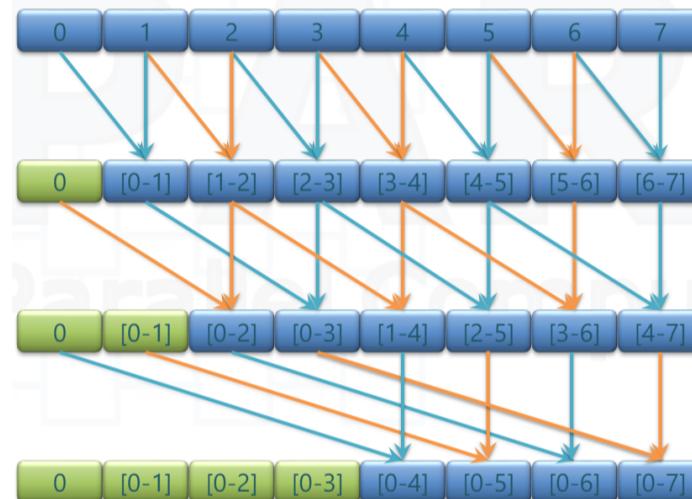
L'implementazione parallela mira a ridurre la complessità temporale tramite il parallelismo. La versione "naive" dell'implementazione parallela esegue l'operazione in diversi livelli, dove ad ogni livello ogni elemento prende il valore dell'elemento corrente più quello di un altro elemento che è ad una distanza potenza di due.

- **Livello 0:** Ogni elemento aggiunge il suo valore con quello del successivo, se esiste (distanza 2^0). Ad esempio, $X[1] = X[1] + X[0]$.
- **Livello 1:** Ogni elemento aggiunge il suo valore con quello a due posizioni di distanza, se esiste (distanza 2^1). Ad esempio, $X[2] = X[2] + X[0]$.
- **Livello 2:** Ogni elemento aggiunge il suo valore con quello a quattro posizioni di distanza, se esiste (distanza 2^2).

La complessità in questo caso è:

- Parallel Complexity (Span) $O(\log(n))$
↳ misura il tempo necessario con risorse infinite a disposizione
- Work complexity of parallel v.: rappresenta il lavoro totale svolto dall'algoritmo (somma totale del lavoro dei processori) (Work done) $O(n \cdot \log(n))$

L'implementazione Naive non è work efficient perché essendo $n \log n$ asintoticamente supera la complessità della sequenziale $O(n)$



Il diagramma mostra come i valori si propagano attraverso l'array ad ogni livello:

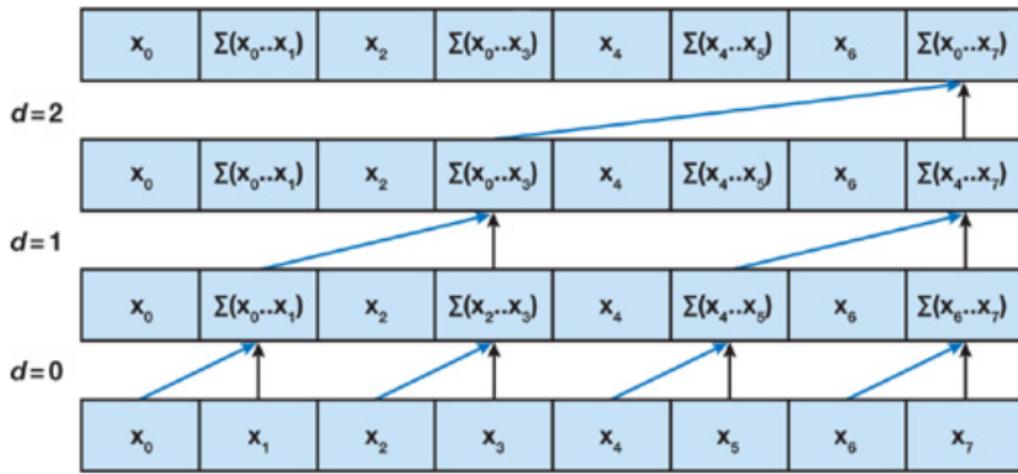
- Ogni cella blu rappresenta l'elemento corrente.
- Le frecce indicano gli elementi che vengono sommati insieme ad ogni livello.

Implementazione Up-Sweep(reduce) and Down-Sweep

L'idea alla base di questa soluzione è creare concettualmente un albero binario bilanciato partendo dall'array di input e percorrerlo in due fasi, dal fondo e dalla radice per calcolare la prefix-scan.

Fase di Up-Sweep:

- In questa fase, l'algoritmo costruisce un albero binario dalle foglie alla radice.
- Ogni nodo interno dell'albero contiene la somma parziale dei suoi figli.
- Questa fase è detta anche **parallel reduction** poiché ogni passo somma le coppie di elementi e salva i risultati nei nodi superiori.
(Alla fine nel nodo radice avrà la somma di tutti i valori)



Diagramma

- Partendo dal livello inferiore, ogni elemento viene sommato al suo successivo (quello a distanza 2^d).
- La somma risultante viene poi utilizzata nel livello superiore fino a raggiungere la radice dell'albero.

```

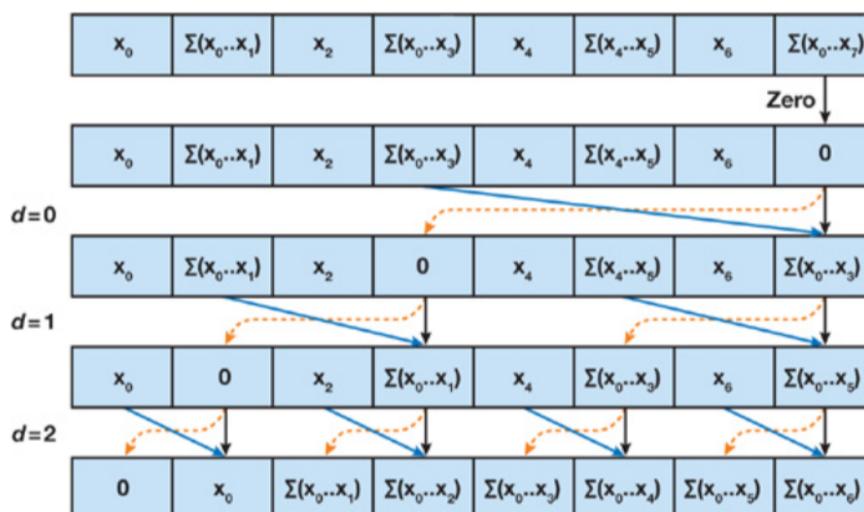
1: for d = 0 to  $\log_2 n - 1$  do
2:   for all k = 0 to  $n - 1$  by  $2^{d+1}$  in parallel do
3:      $x[k + 2^{d+1} - 1] = x[k + 2^d - 1] + x[k + 2^{d+1} - 1]$ 

```

- Work complexity: $O(n)$
- Parallel Complexity: $O(\log(n))$

Fase di Down-Sweep:

- In questa fase, l'albero viene percorso dalla radice alle foglie.
- Usando le somme parziali per costruire la prefix-scan dell'array, si comincia mettendo lo 0 nel nodo radice
- Ad ogni passo, ogni nodo al livello corrente passa il proprio valore al suo figlio sinistro, mentre al suo figlio destro passa il risultato della somma tra il suo valore e quello del figlio sinistro della fase di Up-Sweep



```

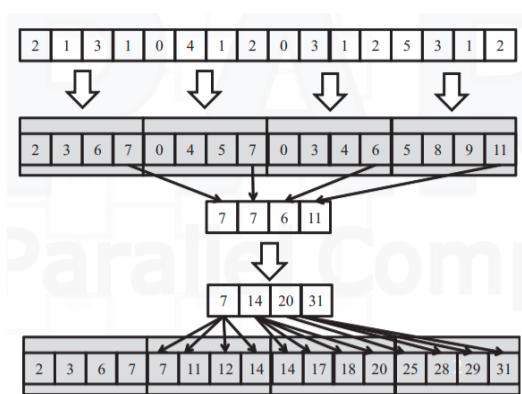
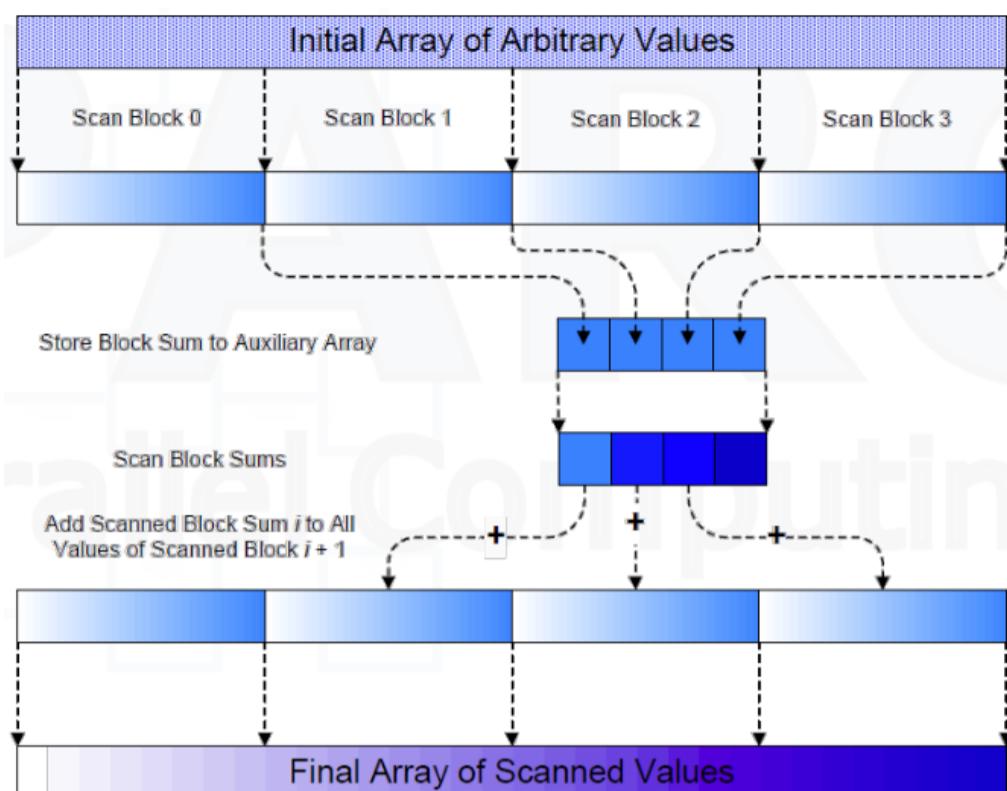
1:  $x[n - 1] \leftarrow 0$ 
2: for d =  $\log_2 n - 1$  down to 0 do
3:   for all k = 0 to  $n - 1$  by  $2^{d+1}$  in parallel do
4:      $t = x[k + 2^d - 1]$ 
5:      $x[k + 2^d - 1] = x[k + 2^d + 1 - 1]$ 
6:      $x[k + 2^d + 1 - 1] = t + x[k + 2^d + 1 - 1]$ 

```

- Work complexity: $O(n)$
- Parallel Complexity: $O(\log(n))$

Concludiamo dicendo che è un algoritmo che fa al massimo $2 * \log(n)$ iterazioni, $2 * (n-1)$ addizioni e un totale di $O(n)$ operazioni, il numero totale di addizioni non è più del doppio rispetto all'algoritmo sequenziale.

Device Wide Parallel Scan



- Ogni blocco scrive la somma della sua sezione in un array somma indicizzato da blockIdx.x
- Si esegue un scan kernel sull'array somma
- Si aggiungono i valori scannerizzati dall'array somma ad ogni elemento della sezione corrispondente

Algoritmi paralleli per grafi

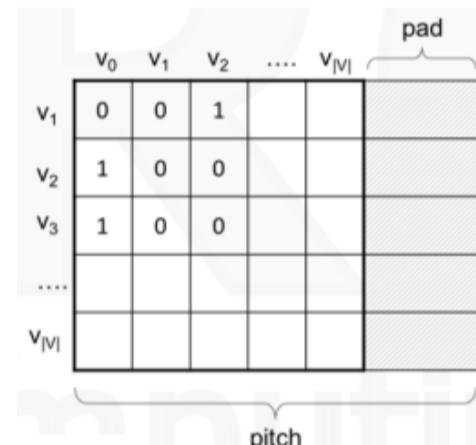
Sappiamo che i grafi possono essere rappresentati in vari modi:

- Matrici di adiacenza
- Liste di adiacenza
- Edge List

Matrici di adiacenza

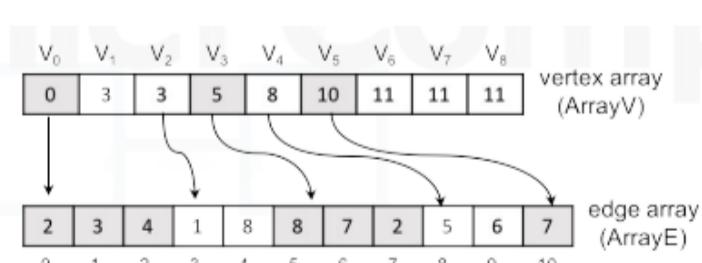
La dimensione è di $\# \text{nodi}^2$, su ogni cella c'è 1 o 0, 1 c'è arco, 0 no.

Vantaggiosa nel pensarla e implementarla, lo svantaggio è che richiede moltissima memoria ovvero $O(v^2)$ spazio. In generale sono utili se usate per rappresentare grafi piccoli e densi (numero di archi circa quello dei nodi al quadrato)



Liste di adiacenza

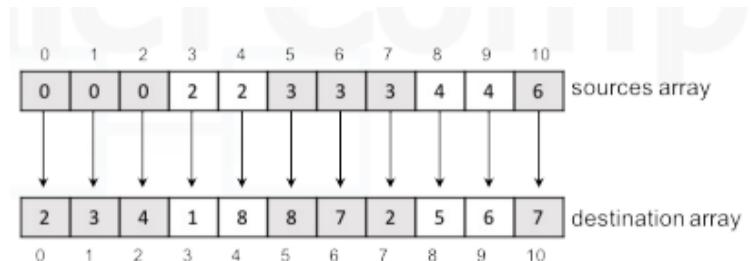
Si hanno due array, uno dei nodi e uno degli archi, sostanzialmente ogni cella del primo array punta a una sezione del secondo array dove sono contenuti i suoi figli, un po' più difficile da implementare ma sicuramente non occupa tanto spazio quanto le matrici. Sono la più comune rappresentazione scelta per grafi sparsi. Essendo che utilizzano i puntatori sono meno adatte alle GPU dove sono sostituite con le Compressed Sparse Row o con le Compressed Row Storage. Permettono comunque in genere di ottenere buone performance



L'array dei vertici tiene traccia della posizione dell'array degli archi in cui iniziano i figli del nodo considerato

Liste degli archi(edge list)

Chiamata anche Coordinate List, utilizza due array di dimensione $|E|$ dove uno salva i nodi di partenza e l'altro salva i nodi di arrivo, quindi in posizione i di A e posizione i di B trovo i due nodi corrispondenti in un arco. Molto adatti se utilizzo algoritmi che iterano su tutti i nodi per incrementare il memory coalescing sorgente destinazione e altre informazioni (come il peso) sull'arco possono essere memorizzate nella stessa struttura



Quale scelgo tra questi?

Scelgo la struttura più adatta in base a 3 cose che mi interessano :

- **memory footprint:** quantità di memoria che usa un programma
- **tempo richiesto per capire se un dato nodo è nel grafo**
- **tempo richiesto dato un nodo per capire i vicini**
- aggiuntivamente consideriamo anche:
 - **load balancing**
 - **memory coalescing**

	Space	$(u, v) \in E$	$(u, v) \in adj(v)$	Load Balancing	Mem. Coalescing
Adj Matrices	$O(V ^2)$	$O(1)$	$O(V)$	Yes	Yes
Adj Lists	$O(V + E)$	$O(d_{max})$	$O(d_{max})$	Difficult	Difficult
Edges Lists	$O(2 E)$	$O(E)$	$O(E)$	Yes	Yes

Background sulla BFS

Sappiamo che la BFS funziona nel seguente modo:

Preso un nodo di partenza che viene messo in una coda FIFO, fino a che la coda non è vuota estraggo dalla coda il nodo in testa, cerco i suoi figli e li aggiungo in coda, una volta fatto ciò il nodo viene segnato come visitato e se ci dovesse essere qualche altro nodo che ha un arco verso di lui non verrà aggiunto nuovamente alla coda. A questo punto si ripete il procedimento precedente con gli altri nodi nella coda fino a che tutti i nodi non sono stati visitati. **BFS ha complessità $O(|V|+|E|)$**

Frontier-Based Propagation

Quando parliamo di **propagazione frontier-based** parliamo di una tecnica dove riusciamo a trovare un albero con radice in S di cui abbiamo che tutti i vertici in ogni livello dell'albero formano una **frontiera F** (ovvero il limite dell'esplorazione dell'albero ad un dato momento).



Quindi quello che ci servirà per scrivere un algoritmo con questa tecnica sono 2 strutture dati:

- **FD:** contiene i nodi di un certo livello, quindi tutti a una certa distanza dalla radice, è letta da thread paralleli per iniziare lo step di propagazione.
- **FDnew:** scritta dai thread paralleli, è utilizzata nell'algoritmo per scrivere FD successiva.

Come possiamo vedere in FDnew abbiamo dei duplicati (ovvero il motivo per cui è stata introdotta) se osserviamo difatti tra FDnew e FD c'è una fase di filtraggio, che ci permette di non avere duplicati (che sono generati dalla visita allo stesso momento di due o più thread) che altrimenti mi farebbero perdere un sacco di tempo e risorse (crescono esponenzialmente) e inoltre ad avere una computazione corretta della BFS.

Una possibile soluzione al problema dei duplicati è l'utilizzo di tabelle di hash in Shared Memory, una per SM:

- Ogni Streaming Multiprocessor (SM) ha una tabella hash nella memoria condivisa.
- Quando un thread trova un nuovo vicino, calcola una posizione nella tabella hash usando una funzione di hash.

- La posizione calcolata è usata per salvare l'identificativo del nodo.
- Se un nodo è già presente nella posizione calcolata, significa che è già stato visitato e quindi viene ignorato.
- Altrimenti, il nodo viene aggiunto alla tabella hash e successivamente alla frontiera FDnew senza duplicati.

```
__device__ bool hash64 ( vertex v )
1:   H_SZ : Hash_Table_Size
2:   h = hash(v)           → h ∈ [0, H_SZ]
3*:  HashTable[h] = merge(v, thread_id)
4:   recover = HashTable[h];
5*:  (vR, thread_idR) = split(recover)
6:   return thread_id ≠ thread_idR ∧ v = vR
```

Usando una tabella hash non possiamo avere duplicati, ma possiamo avere invece dei conflitti, che avvengono in maniera molto più rara dei duplicati (questi avvengono in maniera esponenziale). Se avviene un conflitto siccome non posso capire da dove esso venga, se da un vertice valido o se da un duplicato mantengo il vertice aggiunto (agisco in maniera conservativa).

Tecniche per ottimizzare BFS parallela

Per migliorare le prestazioni del codice parallelo della BFS (Breadth-First Search) su GPU e altre architetture parallele, vengono utilizzate varie tecniche avanzate:

- Exclusive prefix-Sum:** per migliorare tempi di accesso e concorrenza tra thread nei passi di propagazione, le strutture della frontiera sono tenute in shared memory e gestite da una procedura prefix-sum(che può essere implementata tramite istruzioni di warp shuffle)
- Dynamic virtual warps:** presentata per load balancing è applicata per minimizzare lo spreco di risorse GPU e per ridurre la divergenza durante la fase di ispezione dei vicini
- Dynamic parallelism:** in caso di vertici con grado più alto di altri è possibile applicare il parallelismo dinamico al posto dei virtual warps (essenzialmente chiamo un kernel dentro un kernel)
- Edge-Discover:** i thread sono assegnati agli archi al posto che ai nodi per migliorare il workload balancing dei thread durante la frontier propagation
- Single-block versus Multi-block kernel:** two kernel implementation, sono usati alternati e combinati durante la frontier propagation
- Coalesced read/write memory accesses:** per ridurre l'overhead causato dagli accessi in memoria globale, possibile indurla tramite warp shuffle

Prefix-Sum Esclusiva

Nella BFS parallela, si lavora con frontiere di nodi (o vertici) che devono essere esplorati. Ogni thread può esplorare un nodo e i suoi vicini. La **frontiera** è la lista di nodi da esplorare nel livello corrente dell'algoritmo BFS. Quando i thread esplorano i vicini di un nodo, devono scrivere i risultati in un array di frontiera. Se ogni thread scrive i risultati senza coordinazione, si può creare un problema di sovrapposizione o di duplicati. L'**exclusive prefix-scan** viene utilizzato per calcolare in modo efficiente gli indici in cui ogni thread deve scrivere i risultati, evitando sovrapposizioni e garantendo un accesso alla memoria coalescente. Durante l'aggiornamento della frontiera, è possibile che diversi thread esplorino i vicini che sono gli stessi nodi. L'**exclusive prefix-scan** aiuta a organizzare e filtrare questi nodi per rimuovere duplicati in modo efficiente assicurandosi che ciascun nodo sia scritto una volta sola.

Virtual Warps

Un **virtual warp** è un'astrazione che permette di trattare gruppi di thread come se fossero warp, anche se il loro numero può essere diverso rispetto al warp fisico (ad esempio, meno di 32 thread). I **virtual warp** vengono utilizzati in situazioni in cui i dati da elaborare non si adattano perfettamente al numero di thread gestiti da un warp fisico, o quando si vuole aumentare la flessibilità nell'organizzazione e gestione dei thread. (*vo glie curamente accapponare thread simili e separare thread diverse*)

In sostanza, i **virtual warp** consentono un controllo più fine della granularità dell'esecuzione parallela, permettendo di sfruttare meglio le risorse della GPU e ottimizzare le prestazioni, specialmente in applicazioni che non sono facilmente divisibili in gruppi di 32 thread per warp.

Vantaggi e applicazioni

- Flessibilità:** Permette di organizzare e ottimizzare l'uso dei thread in base alla natura dell'applicazione.
- Efficienza:** Riduce il **thread divergence**, un fenomeno in cui i thread di un warp eseguono istruzioni diverse, causando inefficienze.

Parallelismo Dinamico

Permette di implementare la ricorsione nel kernel e quindi di creare thread e blocchi di thread dinamicamente a runtime senza aspettarsi un ritorno del kernel.

Nel contesto BFS l'idea è di invocare un multi-block kernel(child kernel) configurato appositamente per gestire lo sbilanciamento ottenuto da vertici di gradi diversi. Anche in questo caso si verifica un **problema**, l'overhead introdotto può annullare i vantaggi di questa tecnica se applicato incondizionatamente a tutti i vertici della frontiera

La soluzione sta nell'utilizzare questa tecnica solo su un numero limitato di vertici della frontiera, ossia quelli con un grado lontano dalla media partendo dai più alti.

Ci sono 3 possibilità per impostare il parallelismo dinamico, parlando di #block:

- **oversized child kernel:** i blocchi sono di più dei vicini del vertice e quindi concludono prima degli altri threads del parent kernel. Tuttavia, i molti blocchi comportano molte operazioni atomiche per aggiornare le strutture dati di frontiera e un sottoutilizzo delle veloci code locali.
- **undersized child kernel:** meno blocchi gestiscono più vicini del vertice, causa sbilanciamento nei confronti dei thread del parent, in quanto il parent deve aspettare tutte le thread prima di procedere con la prossima propagazione
- **trade off:** si configurano i blocchi del child seguendo la formula:

$$\#block = \frac{\text{vertex_Degree}}{K3 * \text{threadBlockSize}}$$

con tipicamente $K3 = 16$ e vertex_Degree è il grado del vertice di frontiera che sta applicando il parallelismo dinamico

Edge Discover

per miglior balancing (ogni arco ha 2 nodi associati, un nodo ha un numero variabile di vicini)

L'idea è di assegnare threads agli archi piuttosto che ai vertici, il problema è il costo del partizionamento e assegnamento dei thread che potrebbe rendere vani i vantaggi di questa tecnica

Single-block vs Multi-block kernel

Sappiamo che nel primo e nell'ultimo passo della propagazione il parallelismo è limitato, in questi periodi è più conveniente usare un singolo blocco di thread perchè permette di mantenere la frontiera in shmem e alle thread di sfruttare meccanismi di sincronizzazione e comunicazione efficienti. Si imposta un Threshold per decidere ad ogni passo di propagazione se usare un single o un multi-block kernel. In generale comunque il single è runnato nei primi e ultimi passi della propagazione, mentre il multi nei passi centrali.

Coalesced Read/Write memory accesses

Il problema sorge dal fatto che l'update di Fd_{new} è fatto in parallelo, dove ogni thread scrive sequenzialmente i propri vertici, questo fa sì che ci siano problemi di coalescenza dato che gli accessi in memoria si basano sul numero di vertici da scrivere in memoria globale capire se ha fatto anche questo slide 41. Si sfrutta l'accesso a memoria coalescente usando warp shuffle

Load balancing per algoritmi paralleli su grafi

La questione del work-balancing e il mapping del lavoro ai thread è fondamentale quando si tratta di sviluppare buoni software per gpu

Dato un workload da allocare alle thread della gpu, come dividiamo quest'ultimo?

Potremmo usare le prefix-sum per calcolare l'offset di accesso per ogni thread, ciò però in realtà non è a sufficienza per risolvere questo problema perchè:

- la scomposizione del workload e le strategie di mapping sono lasciate al designer
- come viene implementato il mapping influisce sulle prestazioni generali del software

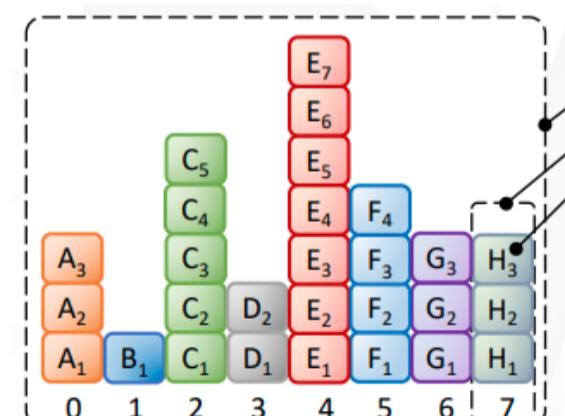
Tecniche di load balancing

Ci troviamo di fronte a 3 classi:

- **Mapping statico:** ideale per lavori con carichi regolari, dove il numero di operazioni per unità di lavoro è simile
- **Mapping semi-dinamico**
- **Mapping Dinamico:** necessario per lavori con carico irregolare. (nodi con numero di vicini molto vario)

Tutti e 3 sono basati sulla **prefix-sum array**

(A destra abbiamo il workload, ovvero tutti i nodi della frontiera)

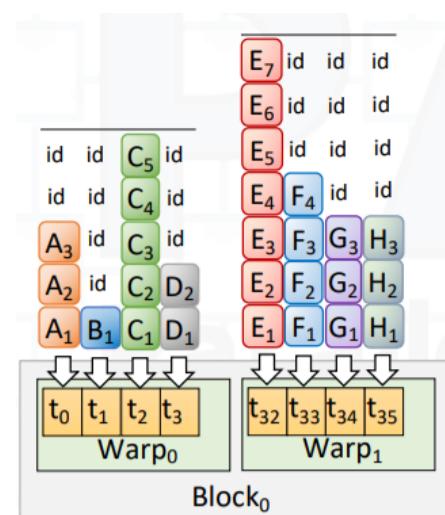


Statico

Statico 1 - Work-items to threads

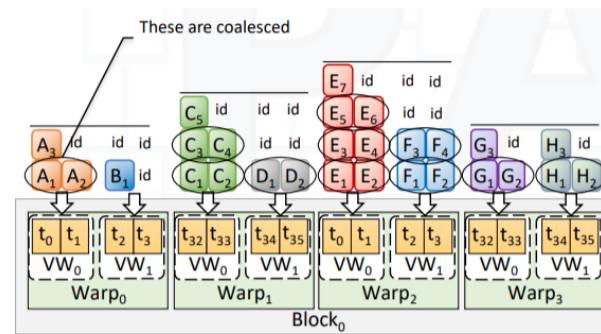
Molto semplice da implementare, non introduce overhead.

C'è però presenza di branch divergence, work-unit mappati a threads dello stesso warp ad uno stato di inattività, inoltre non c'è accesso di memoria coalescente.



Statico 2 - Virtual Warps

Si danno work-units a gruppi di thread chiamati virtual warps, i thread di un virtual warp appartengono allo stesso warp.



carico di lavoro
Il workload qui è quasi lo stesso per tutti i thread dello stesso gruppo, ridotta la divergenza e poco overhead introdotto, riusciamo ad avere coalescenza a warp-level.

In opposizione abbiamo che virtual warp troppo grandi portano a processare sequenzialmente i work-items, non abbiamo balancing tra thread di warp o blocchi differenti, alcuni thread elaborano molti item, mentre altri restano inattivi.

dei branch

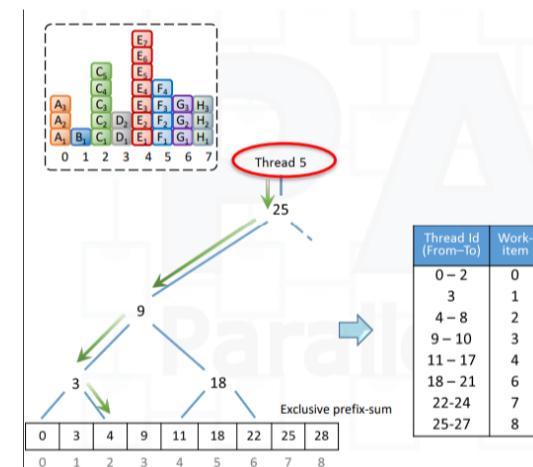
le work unit

Dinamico

Dinamico 1- Direct Search

Assegna una singola unità di lavoro a ciascuna thread usando una ricerca binaria sulla somma prefissa.

- Vantaggi: bilanciamento perfetto del carico tra thread, warps e blocchi;
- Svantaggi: overhead molto alto, dovuto alle molte ricerche binarie e oltre accessi a memoria non coalescenti;



Dinamico 2 - Two Phase Search

Per ridurre lo sbilanciamento tra blocchi, il carico di lavoro viene diviso in "chunk" bilanciati;

Fase 1: si esegue una ricerca binaria globale per trovare i confini dei chunk. Ciò avviene tramite una ricerca binaria sulla prefix sum. Ogni blocco di thread fa questa attività. Se ad esempio ogni blocco ha 128 thread e la memoria condivisa contiene 1300 slot, si utilizza un numero di blocchi che è il più grande multiplo che può stare nella shared memory. I confini vengono mantenuti in un array di partizione e sono necessari a mappare i thread del blocco al chunk corretto di lavoro.

Fase 2: tutti i thread di ciascun blocco caricano il chunk assegnato nella memoria condivisa. Ognuno di essi fa una ricerca binaria locale per trovare la prima unità di lavoro a lui assegnata. Dopo di ciò accede contiguumamente alle unità di lavoro del suo chunk.

Vantaggi:

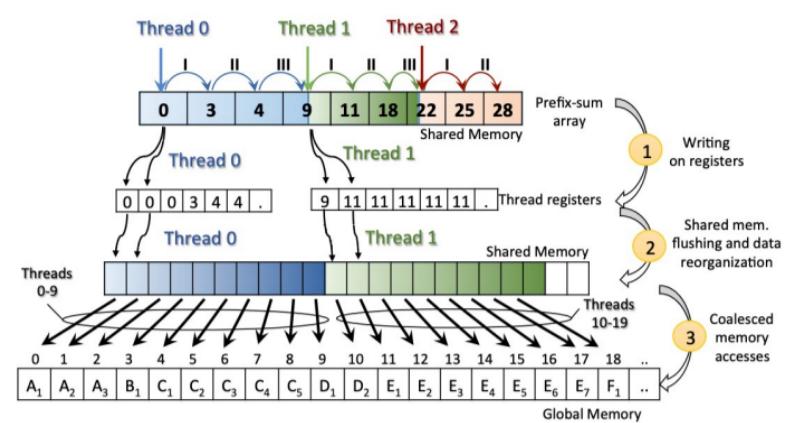
- Accesso coalescente
- Bilanciamento del carico
- Scalabilità'

Svantaggi:

- Overhead computazionale (ricerche binarie)
- Accessi non coalescenti nella fase 1
- Implementazione complessa

Dinamico 3 - Multi-Phase Mapping

L'idea è fare una ricerca binaria ottimizzata e una espansione coalizzata espansa



SSSP(single source shortest path)

Nello stato dell'arte abbiamo vari algoritmi tra cui Dijkstra($O(|V| + |E|)$) che non è adatto alla parallelizzazione, Bellman-Ford($O(|V||E|)$), studiato per la parallelizzazione ma meno efficiente energeticamente

Bellman-Ford

La complessità di questo algoritmo abbiamo detto essere $O(|V||E|)$

```

for all vertex u in V(G) do:
    d(u) = inf
d(s) = 0
for all edges (u,v) in E(G) do:
    RELAX(u,v,w)

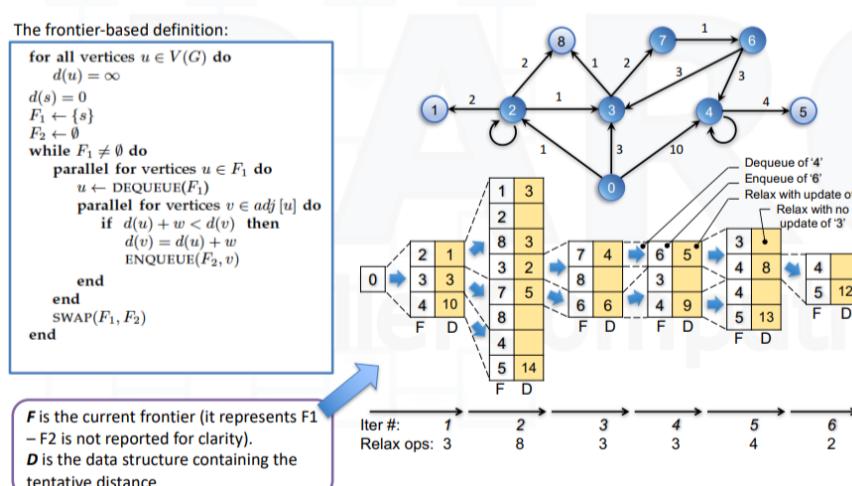
fun RELAX(u,v,w):
    if d(u) + w < d(v) then:
        d(v) = d(u) + w
    
```

Come possiamo notare l'operazione di relax è la più costosa, difatti ogni relax coinvolge un'istruzione atomica nella gestione di race condition.

Una possibile ottimizzazione è l'utilizzo di una coda fifo per tenere traccia dei vertici attivi, ossia:

- tutti quei vertici la cui distanza è stata modificata e che devono essere considerati dalla relax nella prossima iterazione
- se $d(v)$ non cambia, non c'è bisogno di fare la relax degli archi in uscita da lui nella prossima iterazione

Usiamo la **frontiera** per tenere traccia dei vertici attivi:



Fare ciò preserva la semantica perché il processamento dei nodi è indipendente dagli altri

Load Balancing per Bellman-Ford

Tutte le ottimizzazioni viste in precedenza per bfs si rispecchiano anche qui, abbiamo l'aggiunta di altre tecniche tra cui:

Edge Classification: durante ogni step di propagazione, gli archi uscenti dai vertici nella frontiera sono classificati (aggiunta di 2 bit) e processati quindi diversamente per semplificare il più possibile le relax. Quindi ci basiamo sulle proprietà di questi archi per ridurre le relax.

Tra le classi che incontriamo abbiamo:

- Self-loop:** arco che rientra nello stesso nodo, dato che il loop non può cambiare la distanza di u , la relax si può evitare

- **Source edge:** arco uscente dal vertice di partenza, la relax è sostituita con un update diretto della distanza per ogni vicino ????
- **In-degree edge:** archi che sono gli unici ad entrare in un certo vertice (vertice di arrivo ha in-deg = 1), in questi vertici non abbiamo visite concorrenti, quindi abbiamo un update diretto della distanza
- **Out-degree edge:** archi dove il nodo in cui entrano non ha nodi che escono