

Lezione 1

Inizialmente i software erano scritti per la computazione sequenziale:

- 1 per essere eseguiti su singoli computer con una singola Central Processing Unit
- 2 Il problema viene diviso in una serie discreta di istruzioni
- 3 L'esecuzione delle istruzioni avviene una alla volta
- 4 Avviene l'esecuzione di una sola istruzione in qualsiasi momento.

La programmazione parallela è l'utilizzo simultaneo di multiple risorse computazionali per risolvere un problema di computazione.

- 1 Eseguito usando più CPU
- 2 Il problema viene diviso in un numero discreto di componenti che possono essere eseguite concorrentemente
- 3 Ogni parte viene trasformata in una serie di istruzioni
- 4 Le istruzioni vengono eseguite simultaneamente su diverse CPU's

Utilizzo della programmazione parallela:

- modellare problemi scientifici e ingegneristici difficili.
- fisica applicata
- Computer science
-

La computazione parallela viene utilizzata

- per risparmiare tempo e soldi:

↳ assegnare una risorsa a un task renderà minore il tempo di computazione permettendo di risparmiare sui costi

- per risolvere problemi grandi

↳ molti problemi sono talmente grandi che non è possibile risolverli in un solo computer, specialmente avendo una limitata quantità di memoria

- Fornire concorrenza

↳ una singola risorsa di calcolo può fare una cosa per volta. Più risorse di calcolo possono fare più cose concorrentemente

- utilizzo di risorse non locali

↳ utilizzare risorse di calcolo presenti su una rete geografica estesa o addirittura su internet (Cloud Computing)

- limiti del calcolo seriale

↳ l'obiettivo di costruire computer seriali sempre più vicini incontra vari ostacoli

limiti della velocità della luce
limiti della trasmissione dei fili di rete
velocità di trasmissione
limiti di miniaturizzazione
limiti economici
consumo energetico
aumento il costo per rendere più veloce un singolo processore
core in parallelo consumano meno energia rispetto a un singolo core sequenziale equivalente

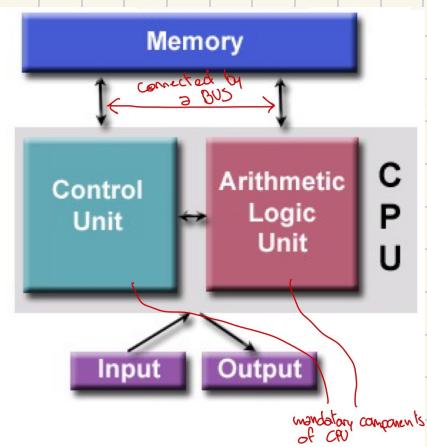
Ad oggi le architetture si stanno affidando sempre più al parallelismo hardware per migliorare le prestazioni

- Unità d'esecuzione multiple
- Istruzioni a pipeline
- Multicore
- Many core

L'utilizzo negli ultimi 30 anni di reti sempre più veloci, sistemi distribuiti e architetture multiprocessore (anche su pc) mostra come il parallelismo è il futuro della computazione

Architettura di Von Neumann

- Proposta da von Neumann nel '45 come requisiti generici per un general computer



· Random access memory: utilizzata per memorizzare sia istruzioni sia dati

· Control Unit: fa il fetch delle istruzioni o dati dalla memoria, decodifica le istruzioni e poi sequenzialmente coordina le operazioni per realizzare le task programmate

· Arithmetic Logic Unit: esegue operazioni aritmetiche di base

· Input / Output sono l'interfaccia con l'operatore

Tassonomia di Flynn

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

Componenti:

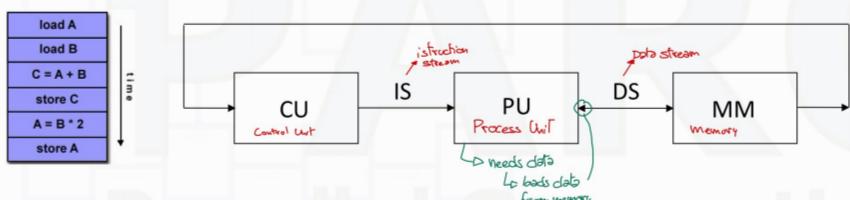
IS - Instruction Stream

DS - Data Stream

CU - Control Unit (performs fetch and decode)

PU - Processing Unit Elemento funzionale composto da ALU e registri che esegue istruzioni

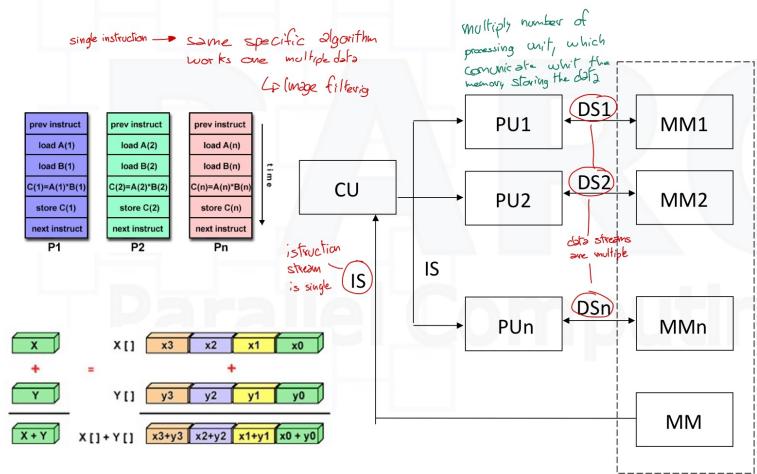
Single Instruction Single Data (SISD)



- CU fetches the instructions from MM, while PU executes the instructions interacting with MM to modify data.
- Standard Von Neumann structure, in which one single program is executing and relies on a single data flow.

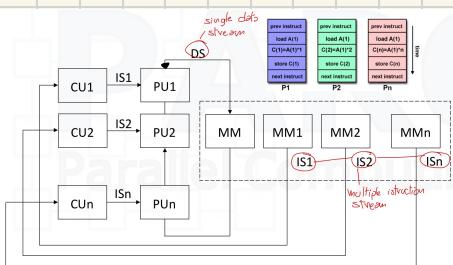
- Only one Data Stream
 - Only one Instruction Stream
 - Computer seriale, non parallelo
 - Esecuzione deterministica
- Esempi: vecchi mainframe, minicomputer e workstation

Single Instruction Multiple Data (SIMD)



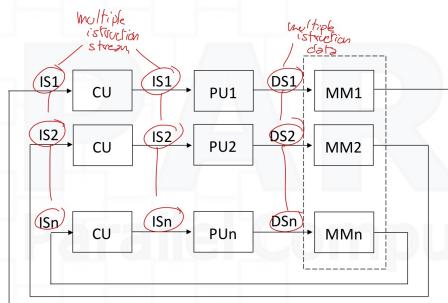
- Un tipo di calcolatore parallelo
- Sono presenti determinismo e sincronizzazione (lockstep)
- Oggi utilizzate particolarmente nelle graphics processor units (GPUs)

Multiple Instruction Single Data (MISD)



- Le operazioni avvengono sui dati in maniera indipendente
- Raramente o quasi mai utilizzate

Multiple Instruction Multiple Data



- Many programs executed in parallel, each one accessing its own data flow.

- Correntemente il più utilizzato, il più comune tipo di computer parallelo
- L'esecuzione può avvenire sia in maniera sincrona che asincrona

Lezione 2

Task: una sezione logica di un lavoro computazionale, che task è tipicamente un programma o qualcosa di simile a un programma come ad esempio insiemi di istruzioni eseguite da un processore

- Processo: task in esecuzione

- Processore: entità fisica su cui il processo esegue

Parallel task: una task che può essere eseguita da molti processori in maniera safe

Serial Execution: esecuzione di un programma sequenzialmente, una istruzione alla volta.

Esecuzione parallela: esecuzione di un programma attraverso più task alla volta, con ciascuna task che esegue le stesse o differenti istruzioni allo stesso momento

Pipelining: rompere una task in step e seguiti da processori differenti, sembra una catena di montaggio

Shared memory: Dal punto di vista hardware descrive un architettura in cui tutti i processori hanno accesso diretto a una memoria fisica comune.

Dal punto di vista della programmazione descrive un modello dove le task parallele hanno la stessa immagine di memoria indipendentemente da dove si trovi la memoria fisica

Symmetric multi-processor (SMP): architetture hardware dove multiprocessori condividono un singolo spazio degli indirizzi e l'accesso a tutte le risorse, shared memory computing

Memoria distribuita: Dal punto di vista hardware si riferisce a un memory access alla memoria fisica basato su reti che non è comune. Come modello di programmazione le task possono vedere solo logicamente la memoria locale e deve comunicare per accedere alla memoria su altre macchine dove stanno eseguendo altri task

Comunicazione: le task parallele tipicamente hanno bisogno di scambiare dati, ci sono molti modi in cui ciò può avvenire, come ad esempio un shared memory bus o attraverso una rete. Tale evento è detto comunicazione

Sincronizzazione: la coordinazione di task parallele in tempo reale, molto spesso associata alle comunicazioni. Spesso implementato dallo stabilire un punto di sincronizzazione in un'applicazione dove una task non dovrebbe procedere oltre fino a quando un'altra task raggiunge lo stesso o un punto logicamente equivalente

Granularità: nella computazione parallela la granularità è la quantità misurabile del rapporto tra computazione e comunicazione.

- **Granularità grossa:** vengono eseguite quantità relativamente grandi di lavoro computazionale tra gli eventi di comunicazione

- **Granularità fine:** vengono eseguite quantità relativamente piccole di lavoro computazionale tra gli eventi di computazione

Speedup: indicatore che indica quanto si è velocizzato il codice, definito come:

$$\text{Speedup} = \frac{\text{tempo di esecuzione seriale}}{\text{tempo di esecuzione parallela}}$$

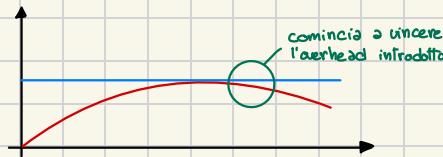
Overshead parallelo: la quantità di tempo richiesto per coordinare le task parallele, a scopo del lavoro effettivo.

- Tempo di avvio
- Sincronizzazioni
- Comunicazione dei dati
- Sottaccanto software
- Tempo di terminazione

Massivamente parallelo: si riferisce all'hardware che compone un determinato sistema parallelo, ovvero dotato di un elevatissimo numero di processori. (anche centinaia di migliaia)

Parallelizzazione banale: si riferisce a problemi o task che possono essere suddivisi in molteplici sottoproblemi o sotto-task completamente indipendenti che possono essere risolti simultaneamente. In questo tipo di parallelismo c'è poca o nessuna necessità di coordinamento tra i vari task.

Scalabilità: si riferisce a un sistema parallelo e consiste nell'avere una proporzionalità tra l'aumento dello speedup e l'aumento del numero dei processori.



Multicore process: processo multicore su un singolo chip

Cluster computing: utilizza una serie di componenti standard per costruire un sistema parallelo

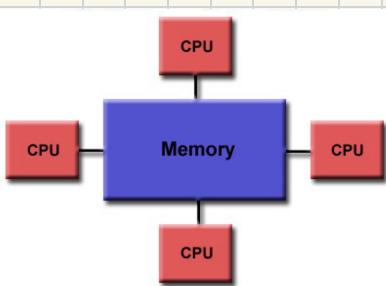
Supercomputing: utilizzare per calcolare i computer più veloci del mondo

Edge computing: computazione distribuita che porta il calcolo vicino al sensore. Utile per aumentare la velocità evitando problemi di rete e per migliorare la privacy dei dati trattati

Sistemi Shared memory

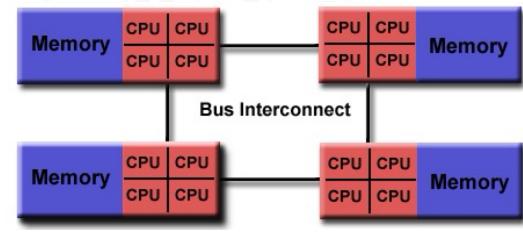
- I computer paralleli a shared memory presentano notevoli variazioni, ma generalmente hanno in comune la possibilità per tutti di processori di accedere a tutta la memoria come spazio di indirizzamento globale.
- Possono lavorare in maniera indipendente ma condividono le stesse risorse di memoria
- Le modifiche apportate a una posizione di memoria da un processore sono visibili a tutti gli altri
- Le architetture a memoria condivisa si dividono in:
 - UMA
 - NUMA

UMA: uniform memory access o Symmetric Multi processing (SMP)



- Sono il tipo di architettura più comune
- Tutti i processori sono identici in termini di architettura e prestazioni
- Tutti i processori possono accedere a qualsiasi posizione di memoria con la stessa velocità
- Può essere implementata a livello HW la coerenza delle cache, ciò consiste in una "notifica" agli altri processori dell'eventuale modifica di una locazione della shared memory da parte di un altro processore

NUMA: non uniform memory access

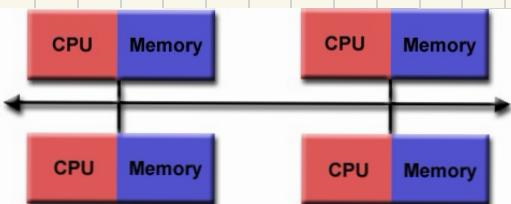


- Si ottengono collegando fra loro più UNA tramite rete
- Un UMA può accedere direttamente alla memoria di un altro UNA
- Non tutti i processori hanno lo stesso tempo di accesso a tutta la memoria, il tempo di accesso locale è più veloce di quello remoto
- Può essere introdotta coerenza

Vantaggi e svantaggi shared memory

- + spazio di indirizzamento globale
- + condivisione dei dati veloce e uniforme
- scalabilità limitata
- Sincronizzazione necessaria
- costo elevato

Sistemi a memoria distribuita

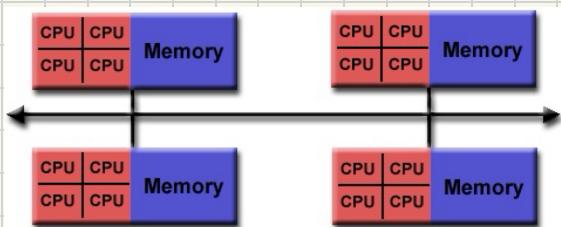


- Anche qui abbiamo varie implementazioni, la caratteristica fondamentale però è la richiesta di una rete di comunicazione per collegare le memorie dei singoli processori
- Ogni processore possiede una memoria locale indipendente, gli indirizzi di memoria di un processore non corrispondono con quelli di un altro, di conseguenza non esiste un unico spazio di indirizzamento globale condiviso da tutti i processori
- Ogni processore ha una propria memoria e opera autonomamente. Quando un processore modifica la propria memoria ciò non ha effetto su quella degli altri, perciò non è necessario introdurre l'implementazione di coerenza
- Il programmatore definisce come avviene la comunicazione, quando un processore necessita di avere dati presenti nella memoria di un altro processore
- posso avere qualsiasi tecnologia di comunicazione io voglia

Vantaggi e svantaggi memoria distribuita

- + scalabilità della memoria
- + accesso rapido ed indipendente
- + costo contenuto
- responsabilità del programmatore, ha la responsabilità di gestire gran parte dei dettagli di comunicazione
- difficoltà di adattamento delle strutture dati esistenti a questa organizzazione
- tempi di accesso non uniformi

Hybrid Distributed - Shared Memory



I computer più grandi e veloci del mondo oggi impiegano architetture a memoria sia condivisa che distribuita

Componente shared memory → solitamente una macchina UNA con coerenza

Componente a memoria distribuita → rete di più macchine SMP, permette la comunicazione delle varie SMP

Sembra l'architettura standard per il prossimo futuro

Modelli di programma

Ci sono molti modelli di programmazione parallela:

- Memoria condivisa
- Thread
- Passaggio di messaggi
- Dati paralleli
- Ibridi

I modelli esistono come astrazione sull'hardware e memoria dell'architettura

Questi modelli non sono specifici per un particolare tipo di architettura.

Ciascuno di essi può essere infatti implementato teoricamente implementato su qualsiasi hardware sottostante.

- modello a memoria condivisa su una macchina a memoria distribuita
questo approccio in generale è chiamato "memoria virtuale" condivisa, era fisicamente distribuita
ma appare come una singola memoria condivisa
- modello di passaggio messaggi su una macchina a memoria condivisa

Scegliere il modello da utilizzare è spesso una combinazione di ciò che è disponibile e dalla scelta personale.

Non c'è un modello migliore, ma c'è un'implementazione migliore dell'altro

Shared memory model

- I task condividono uno spazio di indirizzamento comune, che leggono e scrivono in modo asincrono

+ Vantaggi

- + mancanza del concetto di "proprietà" dei dati
- + non è necessario specificare esplicitamente la comunicazione dei dati fra task
- + lo sviluppo del programma può essere semplificato

- Svantaggi

- maggiore difficoltà nel comprendere e gestire la località dei dati:
 - mantenere i dati locali al processore che li elabora riduce gli accessi a memoria, i refresh della cache e il traffico sul bus che si verifica quando più processori utilizzano gli stessi dati
 - il controllo della località è complesso da comprendere e sfugge all'utente medio

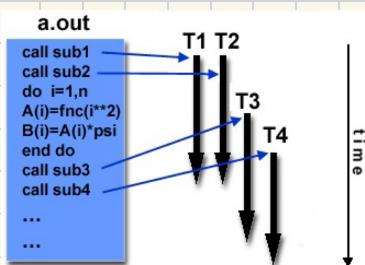
Implementazioni

Nelle piattaforme a memoria condivisa i compilatori nativi traducono le variabili del programma utente in indirizzi di memoria effettivi, che sono globali.

- + non esistono implementazioni effettive

Threads model

- Un singolo processo può avere molteplici percorsi di esecuzione concorrenti:



Implementazioni

Sono spesso associati con architetture shared memory e sistemi operativi.

Da una prospettiva di programmazione solitamente troviamo una libreria di subroutine 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 portiamo a parallelizzare.

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 recive dall'altra parte)

Implementazione

Comunemente nel codice c'è una libreria di subroutine embeddate nel codice sorgente. MPI è un'implementazione di message passing model nota e rilasciata nel 90

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 un cubo.

Un insieme di task lavorano collettivamente sulla stessa datastructure, ovviamente ognuna su una partizione diversa, le task fanno la stessa operazione su partitioni diverse.

Su un architettura a memoria condivisa tutte le task potrebbero aver accesso alla stessa struttura.

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, SPMV ha vari processi autonomi che eseguono simultaneamente senza l'aiuto 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 loro.

Misurare le Performance

- User: per ridurre il tempo di risposta anche detto tempo d'esecuzione o latenza.
Definito come il tempo passato tra l'awo e il completamento delle task.

Sistema: per aumentare il throughput, cuoro il lavoro totale sullo in un dato slot di tempo

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

$$\text{speedup} = \frac{t_{\text{execution } y}}{t_{\text{execution } x}} = \frac{\text{Performance } x}{\text{Performance } y}$$

Uno strumento 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
- synthetic benchmarks: programmi fake inventati per corrispondere col comportamento di applicazioni reali. Stimolano di proposito alcune parti 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 avere speedup migliore

Legge di Amdahl

Il miglioramento delle 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 dovo 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 sarà più alto possibile, speedup globale che è così calcolato:

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

Parte che non può essere parallelizzata

Lo speed globale massimo è calcolabile abbastanza immediatamente come segue:

Tempo come misura di performance

$$\text{Speedup globale} \leq \frac{1}{1 - \text{Fraction improved}}$$

codice non parallelo

Esempio

parallelizzando al massimo un codice parallellizzabile al 50%

$$\frac{1}{1 - 0.5} = 2 \quad \text{lo speedup è 2}$$

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

Abbiamo vari tempi:

- response time: rappresenta lo sforzo 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

$$\text{CPU}_{\text{time}} = \text{CPU}_{\text{clock cycle/task}} * \text{clock cycle time} = \frac{\text{CPU}_{\text{clock cycle}} * \text{task}}{\text{clock frequency}}$$

Altro modo per definire il CPU time è utilizzando il CPI che è così ottenuto

cicli di clock per istruzione

$$\text{CPI} = \frac{\text{CPU clock cycles for a task}}{\text{number of instructions}}$$

$$\text{CPU time} = \text{n}^{\circ} \text{ instructions} * \text{CPI} * \text{clock cycle duration}$$

$$= \frac{\text{n}^{\circ} \text{ instructions} * \text{CPI}}{\text{clock freq}}$$

Bisogna comunque ricordarsi che il CPU time dipende da 3 parametri

- programmatore non può cambiare
la frequenza al massimo cambia il
processore
- 1 Clock cycle (frequenza) dipende dalla tecnologia e organizzazione dell'hardware
 - 2 CPI
 - 3 Numero di istruzioni: dato dall' instruction set architecture e tipologia del compilatore

Per calcolare il numero di cicli di clock di cpu di un certo programma devo utilizzare la seguente formula:

$$\text{CPU clock cycles} = \sum_{i=1}^n (\text{CPI}_i * I_i)$$

I_i = number of times instruction I executed in a task

CPI_i = average number of clock cycles spent for a generic instruction

You can write the CPU time as follows:

$$T_{\text{CPU}} = \sum_{i=1}^n (\text{CPI}_i * I_i) * T_{\text{CLOCK}}$$

MIPS e GIPS (usate per misurare le performance di un sistema di calcolo)

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

$$\text{MIPS} = \frac{\# \text{istruzioni}}{\text{exec time. } 10^6}$$

$$\text{GIPS} = \frac{\# \text{istruzioni}}{\text{exec time. } 10^9}$$

Concepto semplice da capire, una macchina che ha un alto MIPS è molto veloce.

Presentano un problema però, non riesco a distinguere le istruzioni, potrebbero essere istruzioni semplici o complesse, dipendono dall'instruction set, e iS diversi tra loro non sono facilmente confrontabili.

Gflops

Gflops sta per miliardi di istruzioni floating point per secondo, serve per misurare le prestazioni rispetto alle operazioni floating point

$$\text{GFLOPS} = \frac{\# \text{operazioni fp}}{\text{exec time. } 10^9}$$

essendo basata su operazioni e non su istruzioni permette di confrontare architetture diverse.

L'ipotesi però non regge, infatti le operazioni non sono consistenti fra architetture differenti

Introduciamo i GFLOPS normalizzati, 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

Un pezzo di codice con ADD → DIV → SIN è calcolato eseguendo 13 operazioni FP normalizzate

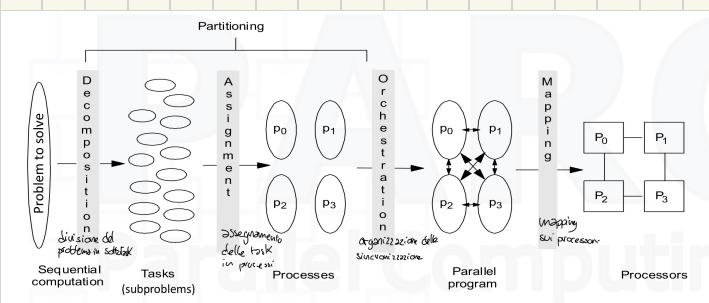
Prospettive sulla parallelizzazione

Tra i veri casi di utilizzo abbiamo la simulazione delle correnti oceaniche, modellate come una griglia bidimensionale, discartando spazio e tempo, trouiamo inoltre la simulazione dell'evoluzione di una galassia o il rendering di scene tracciando i raggi di luce.

Spesso però i calcoli non sono indipendenti, devo avere sincronizzazione tra gli effetti, ma se volessi sincronizzazione massima mi ridurrei ad avere sequenzialità perdendo tutto il vantaggio

Step per creare un programma parallelo

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



- Prima di passare alle 4 fasi è necessario capire il problema e il programma, se questo non fosse parallelizzabile non avrebbe senso perderci tempo. Ad esempio Fibonacci non è parallelizzabile, perché la computazione dipende dalle precedenti.

È importante una volta capito il problema individuare gli **hotspot**, ossia quei punti maggiormente eseguiti, attraverso tool di profiling e analisi delle performance

È altrettanto importante identificare i **bottleneck** ossia le aree di codice molto lente e che quindi rallentano globalmente il programma. Come accessi a memoria

Da ricercare sono anche gli inibitori al parallelismo, come ad esempio la dipendenza dei dati.

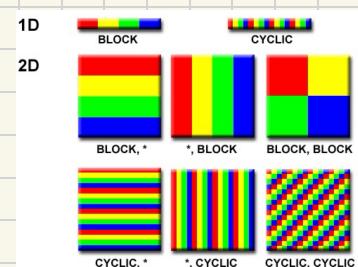
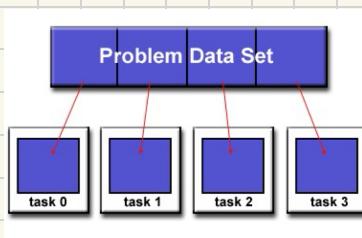
Step 1 : Scomposizione

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 della scomposizione è del programmatore che riesce a farlo meglio di molti tool automatici

Die tipi di scomposizioni

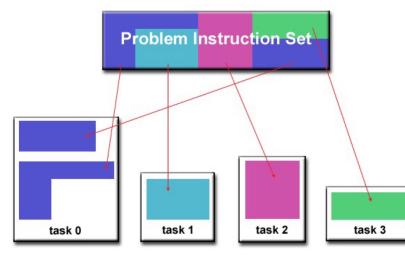
Scomposizione di dominio

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



Scomposizione funzionale

Ci si concentra sulla computazione da eseguire piuttosto che sui dati manipolati da quest'ultima. Il problema è scomposto sulla base del lavoro che deve essere fatto e quindi ogni task sarà una parte del lavoro totale da eseguire.

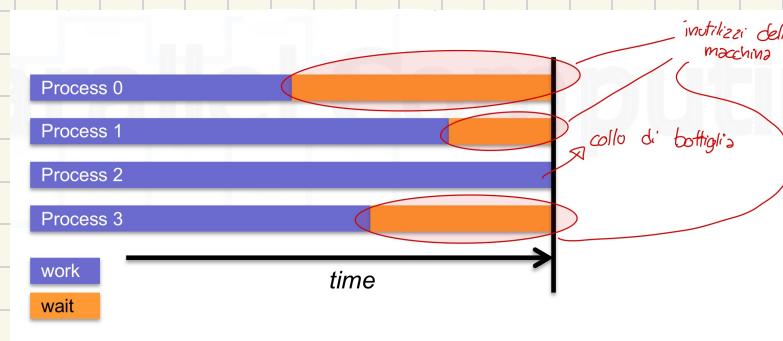


Step 2: Assegnamento

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.

Loadbalancing

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, infatti se abbiamo un punto di sincronizzazione, il task più lento determinerà la performance generale.



19

Per riuscire ad avere load balancing possiamo:

- Partizionare egualmente il lavoro per ogni processo
 - per operazioni su array/matrici dove ogni processo lavora similarmente agli altri, distribuire tra i processi il dataset
 - per iterazioni loop distribuire il loop egualmente tra i vari processi
 - se si sta utilizzando un mix eterogeneo con caratteristiche varie, usare performance analysis tool per scovare eventuali squilibri di carico
- Usare assegnamento del lavoro dinamico, utile perché:
 - certe classi di problemi risultano avere degli squilibri di carico anche se i dati sono distribuiti egualmente tra i processi
 - array sparsi: alcuni processi possono stare 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 gli eventi di comunicazione, facilita il load balancing e rappresenta un'occasione mancata per migliorare le performance, se è troppo fine è possibile che diventi eccessivo l'overhead per comunicazione e sincronizzazione
- **grossa**: grosse quantità di lavoro compiute tra le varie performance, implica una migliore possibilità di aumentare le performance, più difficile fare load balancing

Step 3: Orchestrazione

Mira a strutturare la comunicazione, implementare la sincronizzazione e organizzare strutture dati e schedolare task nel tempo.
L'obiettivo è ridurre i costi di comunicazione e sincronizzazione, schedolare task per soddisfare le dipendenze presto e ridurre l'overhead della gestione del parallelismo.

Comunicazione

La comunicazione fra 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 succedere invece che serva come nella maggior parte dei casi.

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

· **costo della computazione:** 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, infine può saturare la banda.

Latenza vs bandwidth:

· latenza: è il tempo per mandare un messaggio minimo (0 byte) da A a B

· bandwidth: 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 quindi la bandwidth della comunicazione.

· **Visibilità delle comunicazioni:** con il modello message passing le comunicazioni sono esplicite e visibili, con il dato parallel le comunicazioni sono trasparenti, particolarmente sulle architetture a memoria distribuita

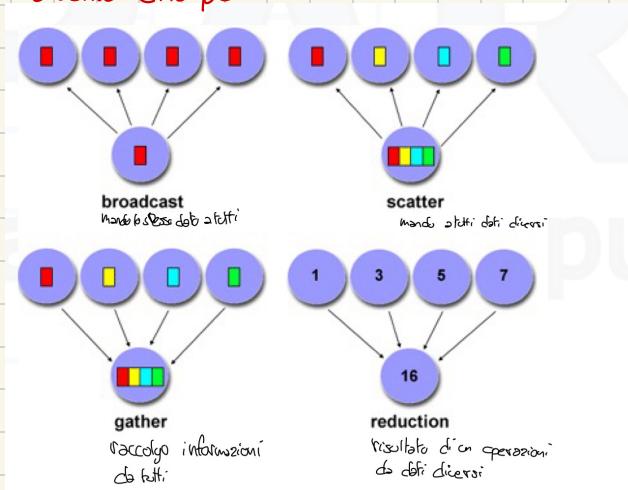
· **Comunicazioni sincrone e asincrone:** possono essere sincrone o anche asincrone e se asincrone possono essere bloccanti o non bloccanti

· **Scope della comunicazione:** sapere quale task comunica con quale altro non è banale durante la progettazione del codice parallelo i seguenti scope possono essere sia sync che async:

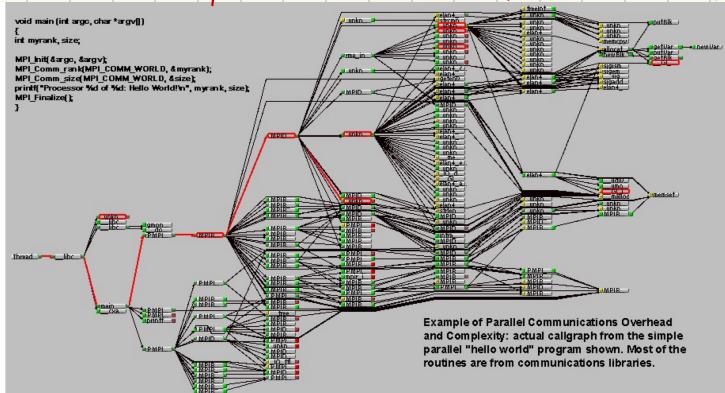
· **PTP:** due task che si scommettono i dati, una produce e manda, l'altra riceve e consuma, una notifica l'altro così può procedere

· **Collective:** più di due task comunicano, spesso appartenenti a un gruppo comune o collettivo

Collective example



Averhead e complessità della comunicazione, spesso è molto pesante



Tipi di sincronizzazione:

- barrier: serie per far attendere fino a che n processi arrivano ad un punto
- lock / semafori
- operazioni di comunicazione sincrone: sono coinvolti sottosinsiemi di processi, vige il concetto di producer - consumer

Step 4: Mapping

Bisogna Capire:

- quale processo esegue e su quale processore
- se più processi eseguono sullo stesso processore

Space Sharing: macchina divisa in sottosinsiemi con una sola applicazione alla volta in ogni sottosistema. I processi possono essere fissati:
a processori specifici o lasciati alla gestione del SO

Allocazione di sistema, l'utente specifica alcuni aspetti desiderati, mentre il sistema si occupa della restante gestione

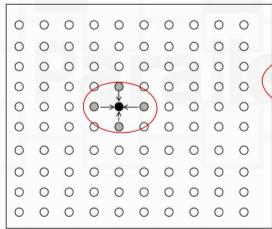
Generalmente si adotta una visione processo ↔ processore

Gli obiettivi ad alto livello sono i seguenti:

- Alte performance
- Basso uso di risorse e costo di sviluppo

Esempio di risolutore di equazioni iterative

Versione semplificata di simulazione di correnti oceaniche



Expression for updating each interior point:

$$A[i,j] = 0.2 \times (A[i,j] + A[i,j-1] + A[i-1,j] +$$

$$A[i,j+1] + A[i+1,j])$$

Non sono indipendenti perché



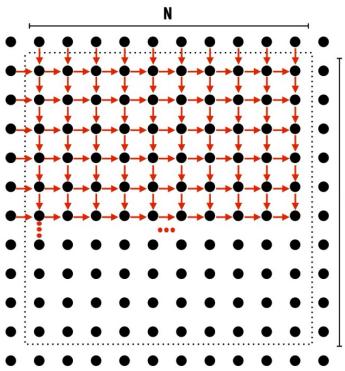
Si svolgono iterazioni di Gauss-Seidel fino a convergenza

- Ad ogni iterazione vengono aggiornati gli $n \times n$ punti interni di una griglia $(n+2) \times (n+2)$



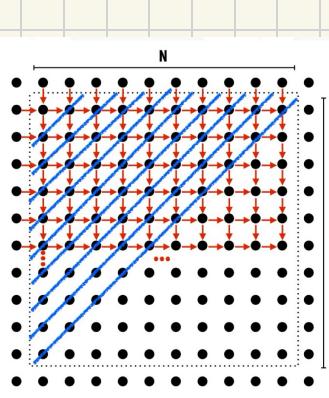
Gli aggiornamenti avvengono direttamente sulla griglia e viene calcolata la differenza rispetto al valore precedente (si verifica quando si è giunti a convergenza)

Identifichiamo le dipendenze, per capire se possiamo parallelizzare il codice sequenziale



- Ogni elemento di una riga dipende dall'elemento alla propria sinistra
- Ogni riga dipende dalla riga precedente

L'algoritmo così com'è non si riesce a parallelizzare



Notiamo però indipendenza lungo le diagonali

Perciò sappiamo che possiamo applicare del parallelismo

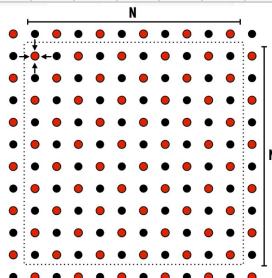
Come possiamo implementarlo?

- Partizionare le celle sulle diagonali in task
- Aggiornare i valori in parallelo
- Quando finito incaricarsi sulla diagonale successiva

Le problematiche sono che all'inizio e alla fine della computazione non c'è molto parallelismo
Frequente necessità di sincronizzare (dopo aver completato ogni diagonale)

Idea: cambiare algoritmo per migliorare le performance passando a uno più adatto al parallelismo

- Cambiare ordine di aggiornamento delle celle
- Il nuovo algoritmo giunge approssimativamente allo stesso soluzioone
ma converge differentemente entro un coefficiente d'errore

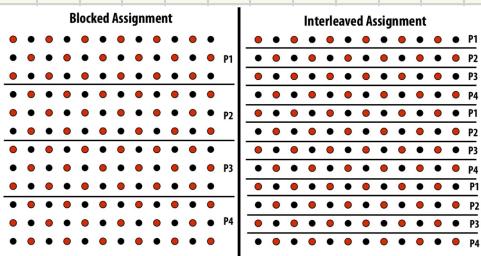


Ordiniamo rosso-nero

Differenti ordini di aggiornamento: potrebbe convergere più velocemente o più lentamente

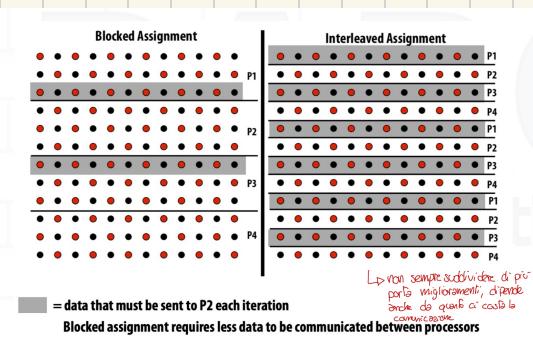
- Punti rossi e punti neri vengono eseguiti completamente in parallelo
- Abbiamo una sincronizzazione globale tra le fasi (conservativa ma conveniente)

- Scopriamo il problema in blocchi
- Assegnamento: potremmo decidere di assegnare grappi di righe a ogni thread oppure di assegnare una riga per ogni thread, tutto ciò dipende dal contesto del problema e dal sistema in cui gira il programma
- Comunicazione: bisogna tenere conto che in questo caso almeno una riga deve essere scambiata tra thread perché c'è della dipendenza



Dobbiamo infatti:

1. Eseguire computazione sui nodi rossi
2. Aspettare che tutti i processi terminino
3. Comunicare l'aggiornamento delle rosse agli altri processi
4. Eseguire computazione sui nodi neri
5. Aspettare che tutti i processi terminino
6. Comunicare l'aggiornamento dei neri agli altri processi
7. Ripeti

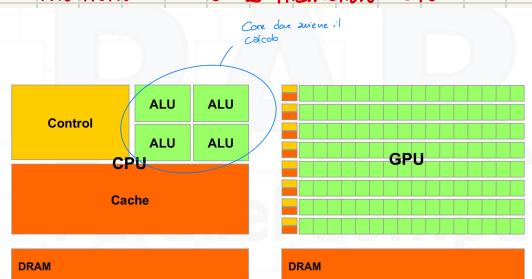


- La sincronizzazione è compito del programmatore
- Primitive di Sincronizzazione
 - Lock: uno solo può accedere alla zona critica per volta
 - Barrier: aspettare che le threads raggiunga quel punto

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

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

Architettura CPU vs Architettura GPU



CPU

- tanta cache per abbreviare i tempi di accesso in memoria che fanno 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 heartbeats per diminuire la latenza delle operazioni

GPU

- hanno meno cache perché si concentra sul memory throughput
- il controllo è semplice non fa branch prediction e data forwarding
- tante ALU ma meno potenti (per ridurre consumi, se aumenta di poco la freq. lo cresce del costo energetico e exp) molto pipelined per aumentare il throughput
- Per tollerare la latenza è necessario un altissimo numero di thread

Perciò non è detto che sia meglio la GPU, sicuramente la combinazione tra CPU e GPU è la cosa migliore. Se infatti esegui un codice completamente sequenziale su una GPU non avrai alcun vantaggio.

Introduzione a CUDA C

È il linguaggio inventato da NVIDIA per programmare GPU.

Chiameremo la CPU host e la GPU device. 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 tornare su CPU e così via.

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.

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 blocchi diversi difficilmente cooperano.

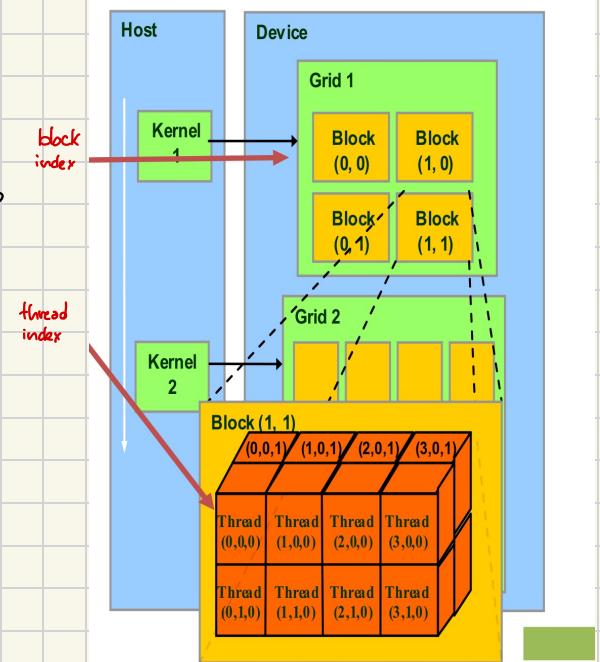
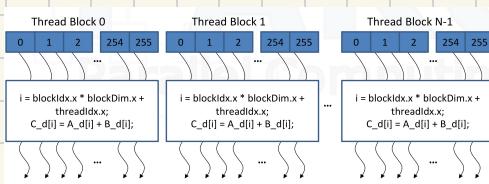
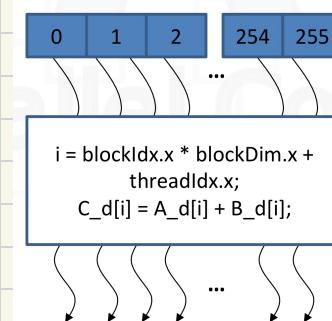
BlockIdx e ThreadIdx

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

Consideriamo di avere un array di 128 elementi da processare, a questo punto la struttura è monodimensionale, questo significa che per ovvie ragioni a ogni blocco 128 thread e queste le identifichero' su una dimensione usando solo la dimensione x .

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

Via così per la terza dimensione prendendo come esempio un cubo. Lo stesso vale per i blocchi che possono essere identificati a loro volta utilizzando fino a 3 dimensioni (x, y, z) .

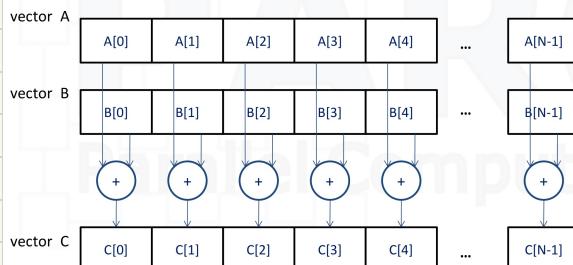
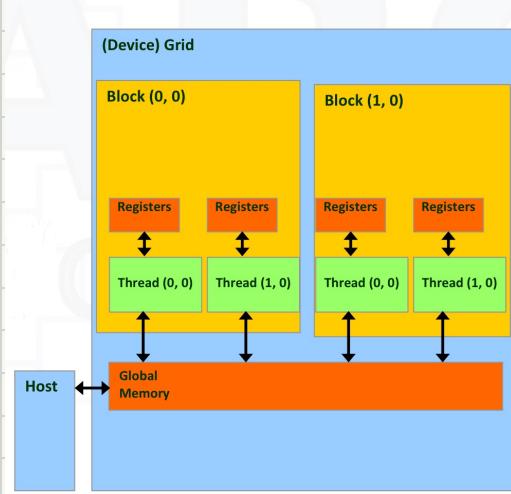


Cuda API per device memory management

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, necessariamente ovviamente di due parametri:
 - 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 4 parametri:
 - puntatore alla destinazione
 - puntatore alla sorgente
 - numero di bytes da copiare
 - direzione del trasferimento



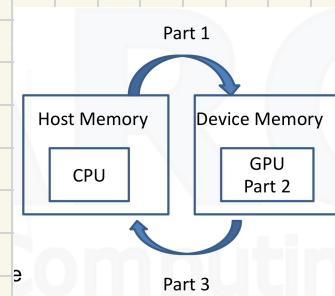
Per fare ciò abbiamo il seguente codice sequenziale:

```
// 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);
}
```

Ora parallelizziamo, in controvermo sempre 3 fasi:

- allocazione della device memory per A, B, C
- Kernel lancia il code
- Copia il risultato C dalla device memory alla host memory



Il risultato di questi step sarà un codice così:

```

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_d, C, size, cudaMemcpyDeviceToHost);
    // Free device memory for A, B, C
    cudaFree(A_d); cudaFree(B_d); cudaFree(C_d);
}

```

```

Device Code
__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);
}

```

Definizione di funzioni

- `--device-- type fun()` è una funzione eseguita dal device chiamata dal device
- `--global-- type fun()` è una funzione eseguita dal device chiamata da host
- `--host-- type fun()` è una funzione per host chiamata da host

• Si può comunque usare `--device--` e `--host--` insieme

Scalabilità trasparente

Parliamo con il ricordare che i thread in uno stesso blocco condividono i dati e si sincronizzano 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. Parliamo ora di **scalabilità trasparente**, sappiamo che l'hardware è libero di assegnare blocchi a ogni processore in qualsiasi momento, più precisamente se parliamo di GPU, per processore intendiamo un 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:

- ① non può succedere che ci siano più di 8 blocchi mappati su un SM (16 se parliamo della serie Kepler)
- ② Possono esserci al massimo 1536 thread in un SM serie Fermi, 2048 se serie Kepler.
- ③ Non possono esserci più di 1024 thread per blocco

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 32 thread che vengono mandati in esecuzione sugli SM. Ci potrebbero chiedere cosa succede quindi a blocchi di più di 32 bit, 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

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 sulle performance legate al branching c'è la **divergenza**, qui abbiamo che i thread prendono strade diverse, 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. Se volessimo portare la divergenza al minimo dovremmo riuscire a introdurre un controllo a livello di warp, quindi thread di un warp dà una parte e thread di un warp dell'altra, evitando che la divergenza si crea a livello di warp saremmo apposta.

Block Granularity: considerazioni su matrix multiplication

Pensando al problema della Matrix Mul, quale dimensione dei blocchi può fornire più senso per uno speedup maggiore?

• 8×8 : abbiamo 64 thread a blocco, ogni SM può prendere al massimo 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 avere grandi speedup.

• 16×16 : 256 thread per ogni blocco, quindi 6 blocchi con questa configurazione dovremo riuscire ad ottenere velocità massima a meno di altre considerazioni riguardanti le risorse non discuse.

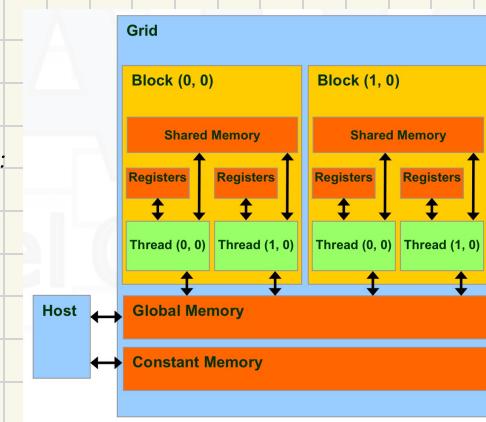
• 32×32 : 1024 thread per ogni blocco e solo un blocco che può stare nel SM altrimenti supereremo il vincolo dei 1536 thread, ciò significa che usiamo solo 2/3 della capacità di un SM.

GPU Memory Model

Vediamo più precisamente lo schema delle memorie presenti in una GPU

Se parliamo di velocità di accesso in ordine dalla meno alla più presente abbiamo:

- 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 const 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 dovesse tornare in global memory ad ogni accesso perderemo troppo tempo.

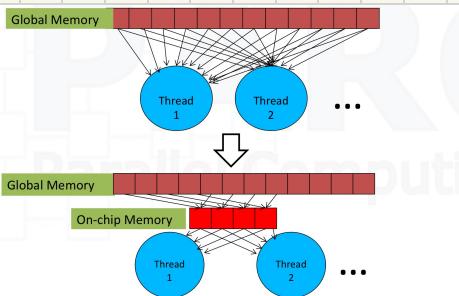
Come si mettono le variabili nelle varie memorie in CUDA

per questo abbiamo impostato i vincoli già prima per garantire la mappatura giusta.

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

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 sottinsiemi 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 sottinsieme caricato in shared memory
- copiare i risultati dalla shared memory alla global memory



• La memoria globale risiede nella device memory (DRAM)
accesso lento

Vediamo da un punto di vista di performance reali 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 consideriamo che una flops (floating point operation), utilizza 4 B/s di bandwidth, siamo a 4 TB/s

$$150 \text{ GB/s} \cdot 1 = 37.5 \text{ Gflops}$$

byte per floating point operation

- Consideriamo 256 th x Block (16x16) con shared memory,

Usa a prendersi tutta la mattonella e la carica in shared memory

256x2 accessi a memoria:

$$\frac{256}{256 \cdot 2} \cdot (2 \cdot 16) = 16$$

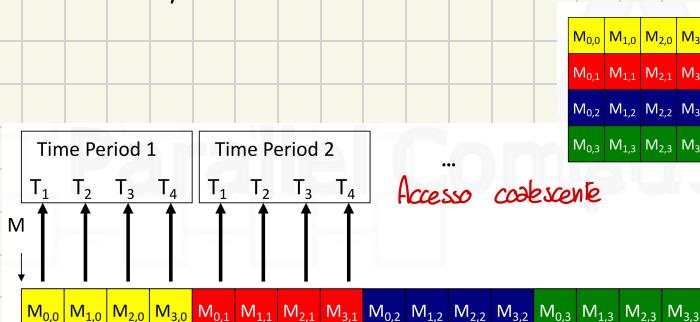
$$\Rightarrow \frac{150 \text{ GB/s}}{4} \cdot 16 \rightarrow 600 \text{ GFlops}$$

byte per operazione
floating point

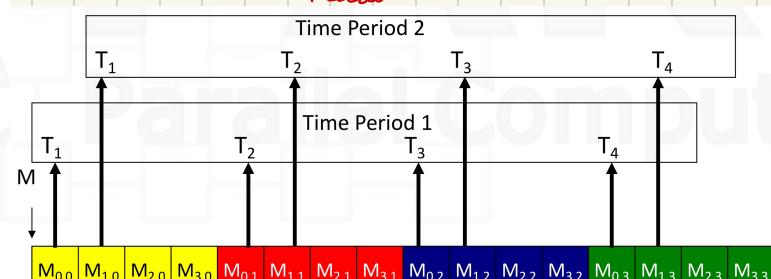
Memory Coalescing

Per tale motivo viene introdotto il coalescing, semplicemente se si vede che tutti i thread in un warp accedono a zone di memoria successive, quindi thread i accede a $n, n+1, n+2, \dots$, $n+2$ è $n+1$ e via così, quello che si fa è recuperare tutto il blocco di memoria accedito 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.

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



Accesso non coalescente



GPU Performance Consideration

- Nelle DRAM l'informazione è mantenuta in condensatori che in base alla loro carica rappresentano un bit a 0 o un bit a 1.
Capire se quest'ultimo è a 0 o a 1 però costa abbastanza, considerando perciò molti accessi ciò ci porta a un'importante perdita di tempo

Ripartizione Dinamica delle risorse di esecuzione

- Tra le risorse di un SM Sappiamo che abbiamo: registri, shared memory e thread block

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

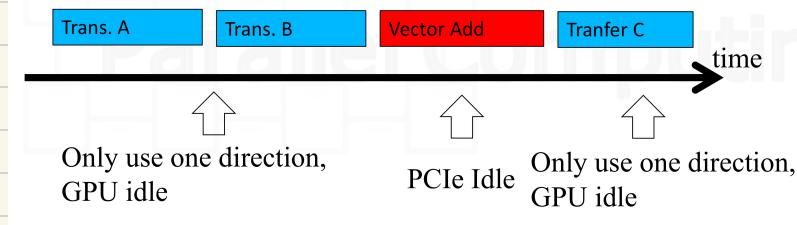
- Se abbiamo blocchi da 512 thread, usiamo 3 blocchi/SM \rightarrow è OK
- Se abbiamo blocchi da 128 thread, usiamo 12 blocchi, ma sappiamo che il massimo è 8 per SM, quindi ho sottoutilizzo delle risorse

Consideriamo anche 16384 registri/SM, abbiamo che:

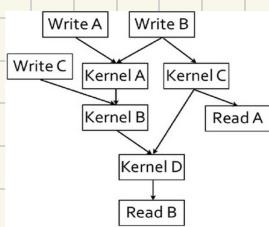
- Se il Kernel istanzia 10 variabili (32 bit per thread)
 - Se abbiamo blocchi da 256 threads, $256 * 10 = 2560 \rightarrow$ registri per blocco
 - Abbiamo $1536 / 256 = 6$ blocchi nell' SM
 - Quindi $6 * 2560 = 15360$ registri in utilizzo \rightarrow OK
- Se aggiungiamo due variabili in più per il Kernel $\rightarrow 12 * 256 = 3072 \rightarrow$ registri per blocco
 - Abbiamo $1536 / 256 = 6$ blocchi nell' SM
 - Quindi $6 * 3072 = 18432$ registri in utilizzo che eccede il vincolo quindi dovrà ridurre i blocchi in esecuzione per soddisfare il vincolo. Perdendo dunque grado di parallelismo
- È fondamentale non usare più variabili di quelle necessarie

Task parallelism for data transfer

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



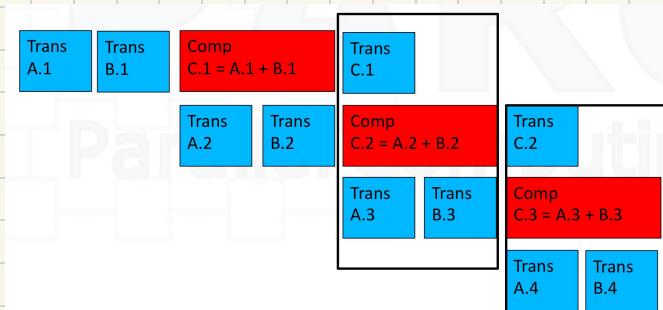
Cio' sicuramente rallenta l'esecuzione generale, osserviamo però che quando mettiamo in coda un Kernel riusciamo a capire tutto ciò che deve essere successo prima che questo esegua, riuscendo a specificare un grafo di dipendenze tra le esecuzioni dei Kernel e i trasferimenti di memoria



Ad esempio nel grafo vediamo come il Kernel A non può eseguire se non sono state fatte le write A e B, ma può potenzialmente eseguire finché avviene la write C

Device Overlap

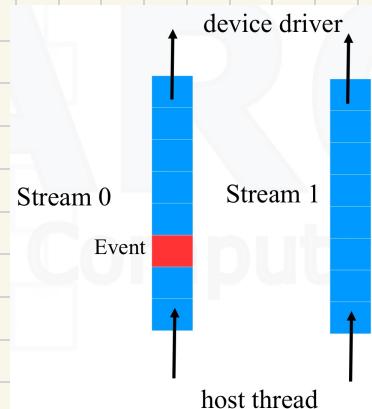
Alcuni dispositivi CUDA supportano il device overlap, che permette formalmente di sovrapporre l'esecuzione di un kernel a dei trasferimenti dati



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 diversi possiamo sfruttare il parallelismo detto task parallelism

• le device request fatte dal codice dell'host sono messe in una coda

- La coda è letta e processata in maniera asincrona dal driver e dal device
- Il driver assicura che i comandi nella coda vengano processati in sequenza.
- Le copie dei dati in memoria devono finire prima del lancio del Kernel
- Per permettere la concorrenza tra la copia da memoria e la Kernel execution sono necessari stream multipli



• Se vogliamo implementare la pipeline ci servono più stream con lo stream zero che è presente di default, dobbiamo quindi creare gli altri

• Da un punto di vista reale quello che accade è ciò che viene descritto nella foto, nel primo stream va il primo chunk nel secondo stream va il secondo chunk e così via.

Vector Add