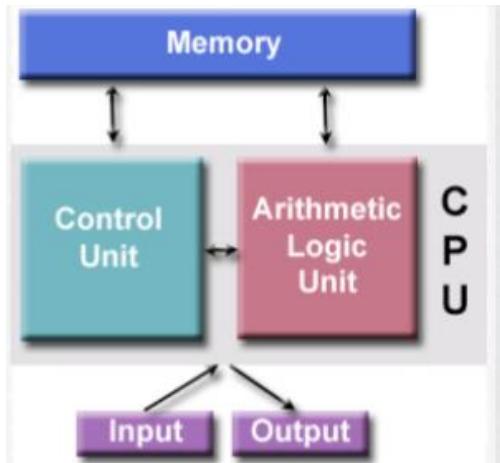


PROGRAMMAZIONE PARALLELA E HPC

ARCHITETTURA DI VON NEUMANN

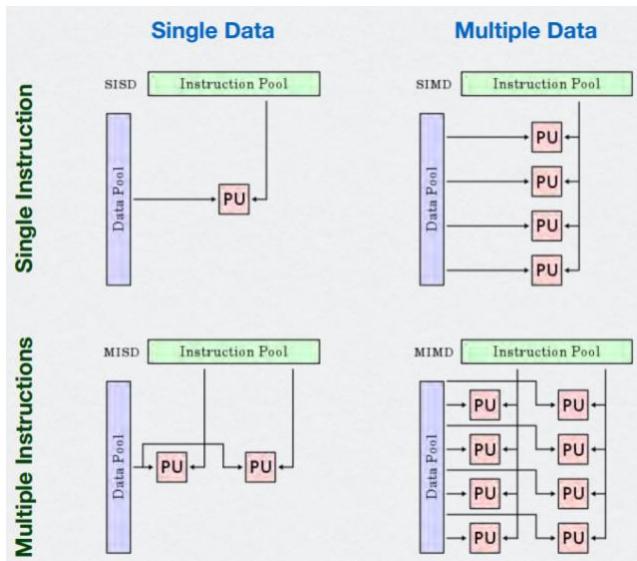


Programma e dati sono salvati in memoria in modo non differenziato. Il processore è scalare. Le varie istruzioni sono lette sequenzialmente ed eseguite un'istruzione alla volta. Per ottenere prestazioni maggiori: Parallelismo hw -> il sw deve adattarsi (so e applicazioni, gestione concorrenza)
Il fine è impiegare meno tempo, si misura con lo speedup: Tempo sequenziale/Tempo parallelo.

ARCHITETTURA PARALLELA

Logica: la vista dal lato dell'utente, tramite il sistema sw
Fisica: l'architettura hw vera e propria (indipendente dall'architettura logica)

STORIA



		Data Streams	
		SINGLE	MULTIPLE
Instruction Streams	Flynn's Taxonomy	SISD Pentium 4 (an x86 architecture)	SIMD SSE instructions on x86
	MULTIPLE	space shuttle flight control computer was MISD oriented	MIMD Intel Xeon e5345

SISD (single instruction stream - single data stream): uniprocessor

MISD (multiple instruction stream - single data stream): multiprocessor

SIMD: utile quando abbiamo le stesse operazioni da eseguire su più dati (array di dati). GPU NON è SIMD.

MODELLO DATA PARALLEL

In certe applicazioni, la maggior parte del lavoro è dedicato a fare operazioni su un insieme di dati con una struttura molto regolare (array, cubi, etc.), se le operazioni sono tutte uguali e indipendenti (si potrebbero eseguire con un ciclo for), il modello sarà data parallel. I dati possono essere divisi in diverse partizioni con la stessa struttura (es pezzi di array) ed elaborati in modo indipendente.

Con la memoria condivisa, tutti i task hanno accesso a tutta la struttura dati (es RAM e thread), con la memoria distribuita, la struttura dati è divisa in pezzi nella memoria locale di ciascun task.

Il problema di SIMD è quando dobbiamo fare operazioni differenti in base al tipo di dato e dobbiamo fare lo switch dell'operazione.

MODELLO IBRIDO

In un modello ibrido si combinano vari modelli classici.

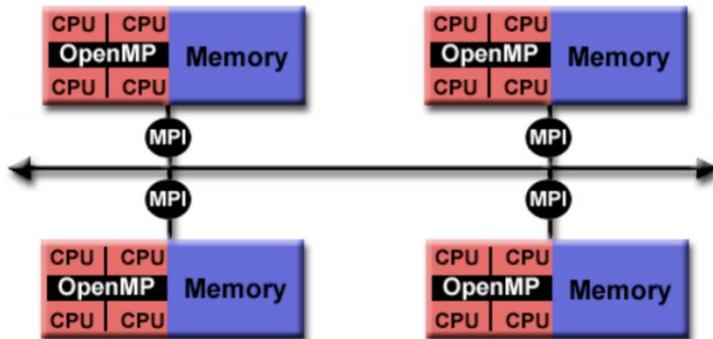


Immagine: Macchine MIMD con 4 core che comunicano tra loro tramite rete (freccia nera).

In questo caso è possibile lavorare su più aspetti del parallelismo contemporaneamente, a livello del singolo elaboratore possiamo lavorare con più thread (OpenMP) di esecuzione contemporaneamente, possiamo però anche far dialogare ciascuno dei programmi paralleli (più thread), scambiando messaggi con i vari processori (MPI). Avremo più elaboratori che lavorano su più thread, che però comunicano attraverso la rete, il risultato è quello di sfruttare più macchine multicore (cluster, il motivo è solitamente il costo, si fa lo stesso anche con le GPU).

MISD

È utile in pochi casi, ad esempio quando vogliamo utilizzare più filtri sullo stesso dato.

MIMD

È il più utilizzato, permette di eseguire con thread diversi, istruzioni diverse su dati diverse, in completo parallelismo. L'esecuzione può essere sincrona o asincrona, deterministica o non deterministica. All'interno di MIMD ci può essere la possibilità di eseguire operazioni di tipo SIMD come sottocomponenti (più unità operative a livello di singola esecuzione del thread).

ORGANIZZAZIONE PROGRAMMA PARALLELO

- Singolo programma che affronta diversi dati (SPMD – Single Programm Multiple Data)

Si progetta l'algoritmo in modo da poter lavorare con un certo sottoinsieme di dati, poi lanceremo lo stesso programma e lo lanceremo parallelamente, ma ogni programma lavorerà su un insieme di dati diverso. Bisogna progettare in modo il programma in modo da fargli sapere quali dati debba elaborare e riconosca e comunichi con le altre varie istanze del programma che saranno in esecuzione. L'organizzazione dei dati sarà regolare.

- Più programmi che lavorano su più dati (MPMD – Multiple Program Multiple Data)

Programmi diversi che lavorano indipendentemente e sono però coordinati e in contatto tra loro, la divisione è funzionale piuttosto che a livello di dati.

MEMORIA

Parallelismo:

Biblioteca -> RAM

Tavolo su cui metto i libri che mi interessano (per reperirli più velocemente) -> cache

Foglio per appunti -> registri

Rete di biblioteche -> memorie collegate attraverso la rete

Località:

- Temporale -> se ho utilizzato un elemento a breve, è probabile che dovrò riutilizzarlo nuovamente a breve (es. ciclo)

- Spaziale -> se ho utilizzato un elemento, è possibile che a breve debba utilizzare anche quelli i cui indirizzi sono ad esso vicino (es. array)

HPC cluster, solitamente posizionato nello stesso ambiente, così non si ha la latenza del viaggio in termini di spazio geografico.

Cache:

Hit: ho trovato quello che cercavo nella cache, altrimenti è miss e bisogna scendere ad un livello più basso (RAM) per cercare il dato

Miss penalty: tempo per rimpiazzare il blocco di alto livello con il blocco di basso livello

Blocco: minima unità di informazione

Hit time: tempo di accesso al livello superiore per determinare se si ha hit o miss

Con il MIMD ciascun core può avere la propria cache, se lo stesso dato “fotografato” sulla memoria principale cambia tra la memorizzazione in una e nell’altra cache, si rischia di avere due valori diversi per lo stesso dato (*cache coherence problem*, problema in hw da risolvere). Un sistema è coerente se:

1. Se CPU P legge la locazione X, seguita da una scrittura di P nella locazione X, e non ci sono scrittura su X da un altro processore nel frattempo, se il valore è stato cambiato da P, bisogna restituire il dato nuovo.
2. Se facciamo una lettura da parte di P di una locazione X, fatta dopo una scrittura da un altro processore P2, bisogna restituire il valore che l’altro processore ha scritto (serve aiuto hw), bisogna informare gli altri della modifica avvenuta.
3. Due scritture nella stessa locazione fatte da 2 processi diversi devono essere viste da tutti nello stesso ordine (devono essere serializzate).

CLUSTER

Idea: Si va a creare computer potenti semplicemente connettendone molti computer più piccoli, si ha una migliore performance per quanto riguarda la gestione del calore e una migliore scalabilità.

I clusters sono gruppi di computer collegati attraverso una rete (solitamente LAN) che possono operare come un unico multiprocessore, lo scopo è di ottenere una macchina astratta che si appoggia a tutti gli elaboratori per svolgere un compito più complesso. La prestazione si ottiene principalmente aumentando il numero di processori piuttosto che aumentare le singole potenze.

Ciascun chip ha più processori (a cui ci si riferisce come core).

PARALLELISMO A LIVELLO DI JOB/PROCESSI

Abbiamo un insieme di job/processi indipendenti che possono essere lanciati in parallelismo puro sui vari core del processore. Livello più diffuso e banale di parallelismo. Rischio di collo di bottiglia se si ha una sola RAM.

PARALLELISMO A LIVELLO DI PROGRAMMA

Un programma parallelo che si appoggia a più CPU contemporaneamente per portare a termine i propri compiti. Un programma che gira su più processori contemporaneamente.

TERMINOLOGIA

Nodo: un nodo è un computer che può essere usato singolarmente, composto da CPUs/core, il singolo nodo è autosufficiente (architettura di Von Neumann).

CPU: oggetto che esegue il programma, più CPU possono essere incorporate in un nodo e a loro volta ogni CPU può essere composta da più core (talvolta chiamato socket).

Task: sezione bene delimitata di lavoro da fare, tipicamente un programma o un insieme di istruzioni, un programma parallelo è un insieme di più task che girano parallelamente su più processori.

Pipeline: attività di spezzettare i vari passi, mettendole in sequenza di modo da sfruttare meglio l'hw che abbiamo a disposizione (è un tipo di parallelismo, ma è possibile farlo anche solo su un programma oltre che sul programma parallelo).

Memoria condivisa: dal punto di vista hw, è un'architettura in cui i vari CPUs hanno un accesso diretto ad una memoria fisica in comune. Dal punto di vista della programmazione, si ha un livello in cui i vari task paralleli vedono la stessa memoria (indipendentemente dal fatto che questa memoria fisica esista effettivamente, ad esempio perché composta dalle memoria di nodi diversi).

SMP (MultiProcessori Simmetrici): architettura hw in cui più CPUs condividono lo stesso spazio di indirizzamento alle varie risorse, non c'è differenza tra i vari elaboratori.

Memoria distribuita: al memoria logica a cui voglio accedere non è tutta sullo stesso dispositivo hw, ma sono vari pezzi di memoria interconnessi attraverso la rete. I vari task vedranno solo la memoria locale, per accedere alle altre parti di memoria dovrò usare la comunicazione (si crea un meccanismo asimmetrico, sulla RAM la lettura è più economica piuttosto che quella fatta tramite l'accesso alla rete). Lo scambio dei dati po' essere fatto in diversi modi (bus interno alla macchina, rete collegata all'esterno, etc.) e le prestazioni sono influenzate dalle scelte fatte. Insieme alla comunicazione bisognerà gestire anche la sincronizzazione in modo da coordinare i task paralleli per garantire delle comunicazioni corrette (solitamente si crea un punto di sincronizzazione in cui un task aspetta l'altro, che va gestito per non sprecare cicli di clock).

Granularità: è una misura qualitativa che misura il bilanciamento tra calcolo e comunicazione. Può essere grossolana (tanto calcolo e poche comunicazioni) o fine (poco calcolo e molte comunicazioni).

Speedup: rapporto tra il tempo utilizzato in un'esecuzione seriale e il tempo in un'esecuzione parallela dello stesso codice parallelizzato.

Overhead: lo speedup ideale non sarà lo speedup osservato a causa del tempo richiesto per coordinare i vari task paralleli, sincronizzare e comunicare, etc. Più task paralleli ci sono, più c'è il rischio che aumenti l'overhead parallelo.

Massimamente parallelo: oggetti di calcolo che hanno tanti (migliaia) processori. Hw in grado di sopportare una potenza di calcolo enorme.

Parallelamente imbarazzante: situazione in cui vogliamo calcolare tanti task indipendenti, in cui non è necessario coordinare i vari task, né farli comunicare. Di parallelo c'è solo il fatto che abbiamo un sistema costoso che ci permette di fare molte cose insieme, ma non lo stiamo usando al meglio.

Scalabilità: capacità di un sistema di seguire proporzionalmente con l'aumento del numero di CPUs, lo speedup parallelo. Oltre un certo numero lo speedup non riesce più a stare al pari con l'aumento di CPUs per problemi hw, algoritmo parallelo non adatto all'architettura sottostante, overhead, altre caratteristiche specifiche degli algoritmi o della codifica utilizzata, etc. In certi casi si può andare peggio che non usando pochi processori.

SCRIVERE UN PROGRAMMA PARALLELO

Prima di scrivere codice, dobbiamo capire se un problema si possa o meno parallelizzare. Esempi:

Calcolare il potenziale energetico per ciascuna delle migliaia conformazioni indipendenti di una molecola. -> si può parallelizzare perché i vari calcoli (sottoproblemi) sono indipendenti tra loro.

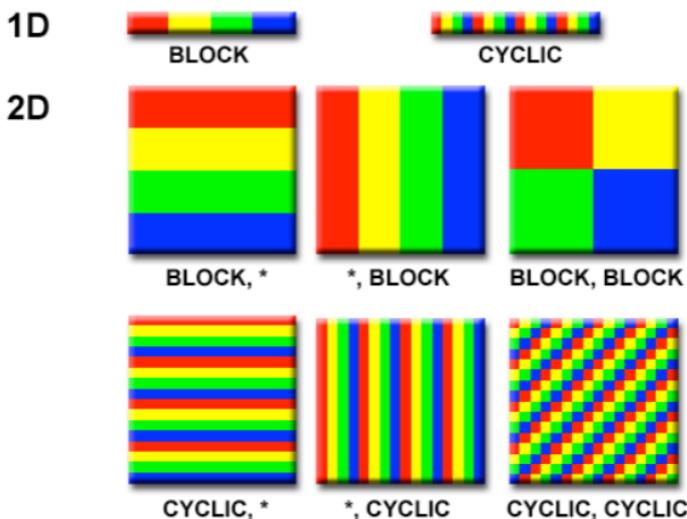
Calcolo della serie di Fibonacci -> non parallelizzabile, per calcolare $F(n)$ ho bisogno di $F(n-1)$ e $F(n-2)$, i sottoproblemi sono perciò dipendenti tra loro.

Una volta che capiamo se il programma è parallelo o meno, dobbiamo:

- ① - Identificare gli **hotspot** (dove il programma passa la maggior parte del suo tempo, la funzione più impegnativa dal punto di vista del calcolo, solitamente sono pochi)
- ② - Identificare i **bottlenecks** (colli di bottiglia, punti in cui il programma va troppo piano, sezioni di programmi paralleli che scalano male, andando molto peggio in pratica rispetto a quanto preventivato, una di questa cause potrebbe essere l'I/O, una riprogettazione di queste parti può ottimizzare)
- ③ - Identificare gli **inibitori del parallelismo** (tutte le situazioni, a livello anche della singola funzione, in cui non possiamo parallelizzare, a causa di dipendenza dei dati)

Tipicamente si agisce sull'organizzazione dei dati che dobbiamo andare a elaborare (*data parallelism*): osserviamo l'insieme dei dati e cerchiamo di dividerli in modo da creare un insieme di task più piccoli che possono essere eseguiti parallelamente. Si agisce con la **domain decomposition** (decomposizione dei dati): si prendono i dati e si suddividono in sottoinsiemi che vengono assegnati a vari lavoratori, nell'ipotesi che i vari task non debbano comunicare troppo tra loro (questo è un punto cruciale, dividere in modo diverso può causare quantità di scambi di dati molto diverse). Si può eseguire la **decomposizione funzionale**: il calcolo generale viene diviso in attività logiche che sono concettualmente indipendenti dalle altre, i vari task lavorano quindi su un aspetto diverso, è orientato alle istruzioni da eseguire più che ai dati.

Ci sono modi diversi di dividere i dati:



Solitamente quella che rende di più è la divisione block, block.

Ci sono modi diversi di eseguire la decomposizione funzionale (vedi slide 44).

COMUNICAZIONE

Sia per la decomposizione sul dominio che per quella funzionale, ci possiamo sempre aspettare che ci sarà comunicazione. Non c'è mai comunicazione solo nel caso in cui le attività sono totalmente indipendenti, solitamente è il caso dell'imbarazzantemente parallelo. Nella maggior parte delle applicazioni parallele invece è necessario condividere dei dati, solitamente molto spesso, per cui la comunicazione dev'essere progettata bene in modo da non avere colli di bottiglia. Ad esempio nelle simulazioni (ad esempio di termodinamica) c'è il problema dei vicini, con cui è necessario comunicare.

L'impatto della comunicazione:

Quando devo comunicare, per forza perdo tempo, in quanto un task deve aspettarne un altro, se devo aspettare tanto per comunicare, perdo tanto tempo e perdo quello che avevo guadagnato con il parallelismo.

Per poter comunicare è necessario preparare i dati e trasmetterli, operazione che ha un costo in termini di tempo; oltre al tempo di rete devo sottrarre dei cicli di clock per preparare la comunicazione.

Se è richiesta sincronizzazione tra i task, i task spendono tempo aspettando e non lavorando.

Se la comunicazione genera un traffico che satura i canali di comunicazione, dobbiamo aspettare che la rete venga decongestionata.

Latenza (tempo impiegato per ottenere il primo byte in una comunicazione) e larghezza di banda (quantità di informazione che può migrare da un punto all'altro) hanno un alto impatto nel costo della comunicazione. Potrei scegliere di inviare molto spesso dati piccoli (se la latenza è alta ci perdo) o inviare dati molto grandi poco spesso (va bene se la banda è grande, se la banda è molto piccola ci perdo, in ogni caso la latenza viene "pagata" solo una/poche volte). Per i dati grandi, possono essere divisi tra i vari task, mentre per la latenza ci si può fare poco. Questo è il motivo per cui i cluster hpc tendono ad avere una latenza molto bassa perché le comunicazioni tendono ad essere molto frequenti.

Con il modello di passaggio dei messaggi le comunicazioni sono visibili, esplicite e sotto il controllo del programmatore che, appoggiandosi alle API.

Quando invece si usano dati distribuiti, nel modello data parallel, le comunicazioni saranno impliciti o trasparenti, dato che i dati sono condivisi; il sistema passa il dato attraverso la rete, ma il programmatore non ne ha il controllo.

Le comunicazioni possono essere sincrone (bloccanti, bisogna aspettare che send e receive abbiano entrambe finiti) o asincrone (non bloccanti, faccio la send/receive e mi metto a fare altro, senza bloccarmi, di modo da riempire il tempo che avrei speso aspettando con il calcolo).

Le comunicazioni possono essere di vari tipi (broadcast, scatter, gather, reduction) e possono essere point-to-point o collettive.

Tipi di sincronizzazione: barriera (faccio aspettare tutti fino a quando tutti quanti non arrivano alla barriera (hanno finito di fare qualcosa di specifico), solitamente coinvolge tutti

i task), **lock e semafori** (numero variabile di task, usato generalmente per serializzare, solo un task alla volta può usare (essere proprietario) del lock o semaforo).

DIPENDENZA DEI DATI

Se usiamo una stessa locazione e ne facciamo un uso multiplo, abbiamo una dipendenza dei dati: se si ha almeno una scrittura e poi una lettura/scrittura successiva ad essa, si ha una dipendenza dei dati. Un esempio di dipendenza è quella dell'algoritmo di Fibonacci.

LOAD BALANCING

Se c'è una barriera di sincronizzazione o il termine dell'elaborazione, ma il più veloce deve aspettare il più lento, se abbiamo distribuito male il lavoro, allora l'attesa diventa significativa e si ha uno spreco consistente di tempo, non si scala quanto preventivamente. Devo distribuire il più equamente possibile il carico di lavoro. Non è detto però che anche dividendo i dati in modo equo, il carico di lavoro sia distribuito in modo equo: ci sono ad esempi i tempi di comunicazione (0 comunica con 1, ma 1 comunica con 0 e con 2, quindi il doppio); si scopre che magari la divisione migliore non è quella in cui si hanno blocchi tutti uguali, potrebbe essere meglio caricare task che comunicano meno per fare in modo che il tempo impiegato da tutti i task sia lo stesso o comunque distribuito equamente.

GRANULARITÀ

Granularità: è una misura qualitativa che misura il bilanciamento tra calcolo e comunicazione. Può essere grossolana (tanto calcolo e poche comunicazioni) o fine (poco calcolo e molte comunicazioni).

Se la granularità è troppo fine è possibile che il tempo richiesto per le comunicazione e la sincronizzazione tra i task richieda di più della computazione stessa.

Se la granularità è troppo grossa e sbaglio, rischio di far aspettare gli altri task per molto tempo, è più difficile da bilanciare, ma se fatto bene arrivo a prestazioni davvero ottime.

L'approccio migliore dipende dall'algoritmo e dall'hw utilizzato.

I/O

È un'attività pessima per il parallelismo, generalmente si vuole fare il meno possibile, se non evitarlo del tutto. Se proprio si desidera potrebbe andar bene dedicare 30 secondi di I/O (operazioni sulla memoria secondaria) ogni 30 minuti di calcolo.

PERFORMANCE

Scrivere un programma parallelo efficiente è un compito difficile (sicuramente più di scrivere un programma seriale) che richiede la considerazione di tanti aspetti, poiché bisogna:

- Trovare parti da parallelizzare (non tutti o non l'intero programma può essere parallelizzato)
- Valutare la granularità (quanto grande è il task parallelo)
- Valutare la località (dove li mettiamo e quanto costa accedervi, organizzazione e uso della memoria)
- Bilanciare il carico
- Coordinare e sincronizzare le operazioni e i dati
- Osservare le performance fare debugging, profiling ed eliminare eventuali colli di bottiglia, adattando il programma all'hw

Gli obiettivi sono:

- Provare a ridurre la comunicazione e la sincronizzazione (meno ce n'è, meno si aspetta)
- Bilanciare bene i task
- Garantire la coerenza di cache (di solito non ce ne occupiamo direttamente)
- Gestire bene l'I/O ed avere un hw di I/O performante

LEGGE DI AMDHAL

Se ho 100 processori, vorrei andare 90 volte più veloce, ma questo non sarà possibile, poiché ci saranno parti del programma non parallelizzabile.

È tutto in funzione della parte di codice parallelizzabile, se c'è poco codice parallelizzabile, potrò andare meno veloce.

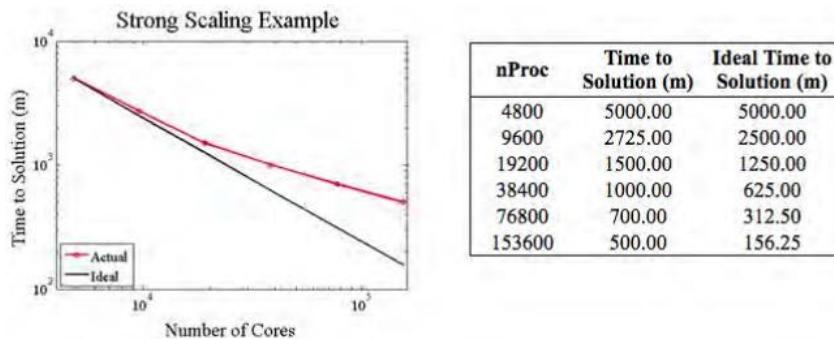
La legge di Amdahl afferma che se F è la frazione di un calcolo che è parallelizzabile (cioè che può beneficiare dal parallelismo), e $(1 - F)$ è la frazione che non può essere parallelizzata, allora l'aumento massimo di velocità che si può ottenere usando N processori è

$$\frac{1}{(1 - F) + \frac{F}{N}}$$

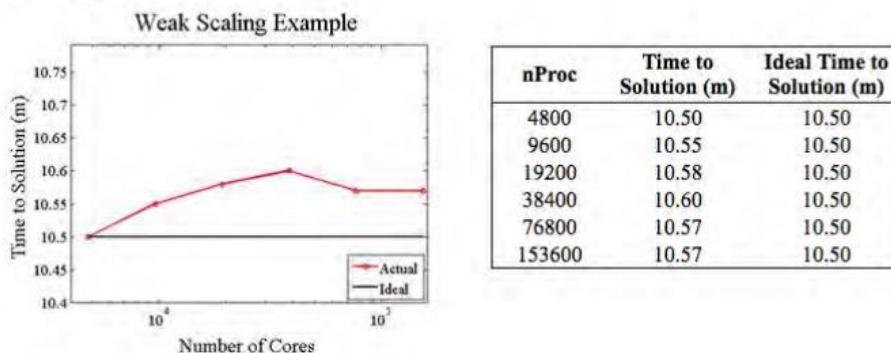
Quindi in un sistema parallelo, la parte non parallelizzabile, deve essere estremamente piccola (tipo massimo 1%), non possiamo permetterci altrimenti, dobbiamo cercare di portare l'efficienza vicino al massimo teorico.

SCALING

- Valuta la scalabilità di un programma, risolvendolo con più processori. Può essere:
 - Forte: speed-up su multiprocessori senza aumentare la grandezza del problema, lancia diverse istanze dello stesso programma ogni volta associate ad un numero crescente di processori (la legge di Amdahl è stata misurata così)



- Debole: speed-up su multiprocessori, mentre si aumenta proporzionalmente la grandezza del problema all'aumentare dei processori, ci aspettiamo, in teoria, che il tempo totale rimanga uguale, pur all'aumentare della grandezza del problema. Tuttavia bisogna fare delle considerazioni: se un lavoratore è più lento, andrà aspettato; bisogna considerare le comunicazioni, che possono creare problemi.

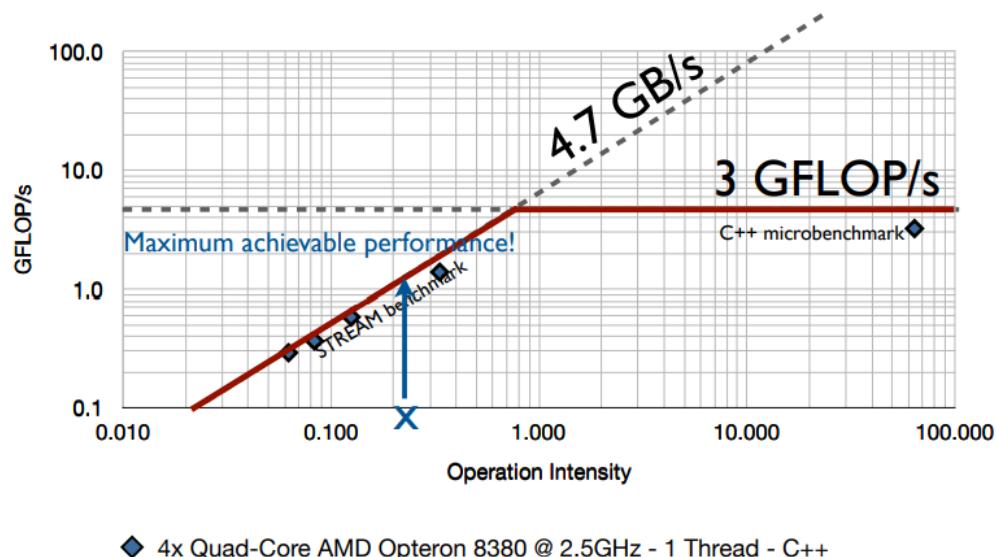


Se il weak scaling va male ad un certo numero di processori, con lo stesso numero di processori, lo strong scaling andrà peggio.

ROOFLINE MODEL

Serve per fare una stima delle performance raggiungibili dal programma parallelo, indicando anche su quale parte del programma andare a fare ottimizzazioni. Questo modello vuole mettere in relazione la potenza di calcolo [FLOP/s] con la capacità di trasferire i dati dal CPU alla memoria principale [GB], con le relative caratteristiche. Ciò ci permette di capire l'intensità operazionale [FLOP/Byte], per ogni operazione aritmetica vado a vedere quanti Byte ho dovuto trasferire. La curva risultante, chiamata appunto Roofline, costituisce un limite superiore alle prestazioni ottenibili, al di sotto della quale esistono le effettive performance per il dato kernel o applicazione. Suddetta curva include due limiti massimi specifici della piattaforma: un limite derivato dalla banda di memoria e un limite derivato dal picco di performance dell'unità di computazione. Il programma gira effettivamente al di sotto

della linea rossa (roofline), l'incrocio tra le righe rosse corrisponde al massimo delle prestazioni. Per ottenere queste informazioni utilizziamo un profiler a basso livello.

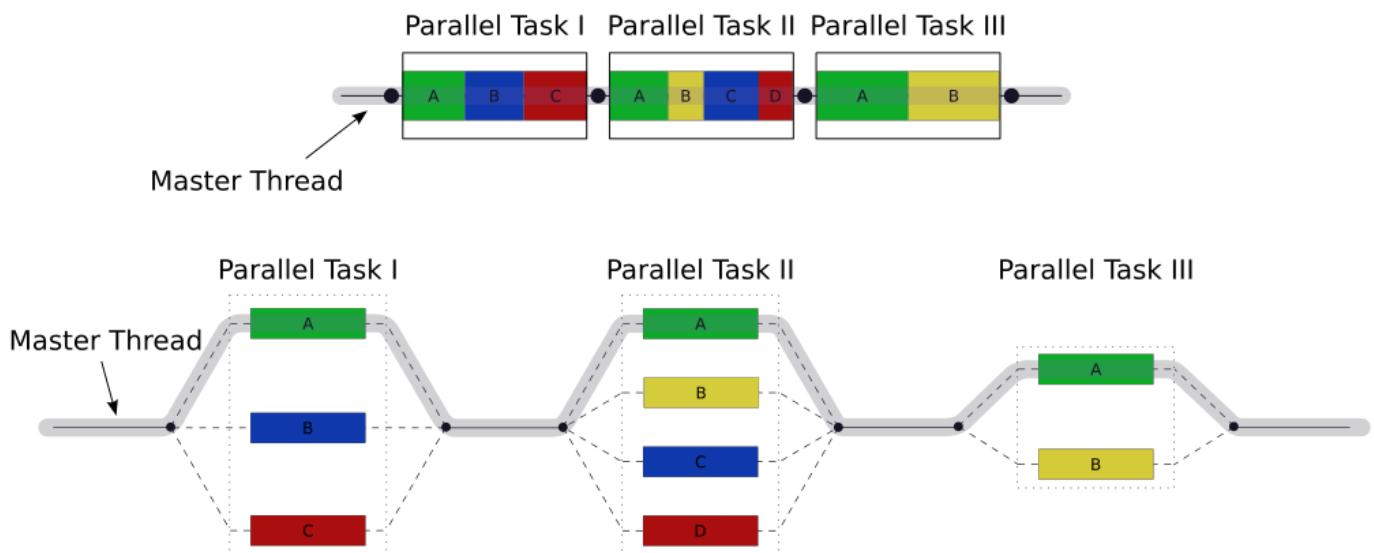


OPENMP(LAB)

(<https://wiki.smfi.unipr.it/dokuwiki/doku.php?id=roberto.alfieri:pub:openmp>)

È una libreria che ci permette di implementare facilmente il modello a memoria condivisa tramite i thread in C e C++. I vari thread condividono la memoria condivisa, ma ha anche un'area di memoria privata, ad esempio per le variabili.

Modello Fork-Join: quando il programma thread si ha il singolo thread master che crea dei thread a cui si affida il lavoro, alla fine del quale i thread collassano e si torna al singolo master thread. Questo viene fatto attraverso delle direttive per il compilatore, delle routine runtime e variabili d'ambiente.



Parallel è la direttiva che serve per creare una regione parallela, dichiarando lo scope della variabili attraverso private o shared, di default le variabili esterne sono condivise, mentre quelle dichiarate internamente alla regione parallela sono private.

Il valore di default dei thread è 2, questo numero si può modificare o modificando la variabile con `omp_set_num_threads(n)` o con env `OMP_NUM_THREADS=n`.

Critical è la direttiva che serve a gestire le sezioni critiche, facendo in modo che un solo thread per volta possa eseguire la sezione critica.

Reduction è la direttiva che si occupa di ricomporre i risultati calcolati dai diversi thread in un unico risultato finale (tramite varie operazioni come somma, differenza, etc.).

Section è la direttiva utile per i modelli a scomposizione funzionale, dichiaro delle sezioni e poi vado a descrivere cosa farà ciascuna sezione.

Leggi la wiki ed esegui i comandi.

MEMORIA CONDIVISA

Tramite questo meccanismo, tutti i processori vedono e utilizzano o spesso spazio globale di memoria. I processori possono agire indipendentemente, ma condividono le stesse risorse di memoria, se un processore modifica i dati in memoria, tutti gli altri processori lo vedono.

Pro:

- Avere un solo spazio di memoria rende la programmazione più facile
- La condivisione tra i vari task è più veloce e uniforme, data la vicinanza dei processori

Contro:

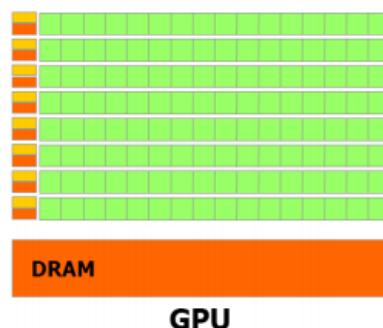
- Bassa scalabilità, un aumento nel numero di CPU e memorie causa un aumento del traffico sui canali di comunicazioni tra i vari processori e la memoria
- Il programmatore deve gestire i meccanismi di sincronizzazione per permettere un corretto accesso ai dati

Un SMP (Symmetric MultiProcessor) è un computer con molteplici processori identici sulla stessa scheda madre, condividendo tutti la stessa memoria. Dato il basso costo dei CPU e dei requisiti per la comunicazione, questa configurazione ha un buon rapporto di efficienza/costo, se è disponibile abbastanza memoria. Problemi: è complesso implementarla e mantenerla, non è scalabile e viene sfruttata male la cache.

Un processore multicore ha molteplici unità di esecuzione sullo stesso chip, tutte che condividono la stessa memoria. Sono MIMD

GPU

Anche nel caso delle GPU, l'hw è stata modificata rispetto all'esecuzione tradizionale. Nella GPU si implementano molte unità aritmetico logiche (ALU), mantenendo le unità di controllo molto semplici. Ogni thread avrà una dato diverso da lavorare e le varie ALU potranno fare operazioni uguali su dati diversi contemporaneamente; ci possiamo aspettare una traffico di dati molto alto con banda molto alta. In questo modo le GPU sono specializzate per fare computazioni intense e fortemente parallele (rendering grafico).



Le GPU montate in ateneo hanno 3500 cores e hanno capacità di 4.7 TeraFLOPS in operazioni double, i TeraFLOPS aumentano, ovviamente, riducendo la precisione,

MEMORIA DISTRIBUITA

Richiede comunicazione su una rete per fare scambio di informazioni, dato che i dati non sono più in locali, anche se ogni processore ha la sua memoria locale.

Pro:

- Possiamo crescere quasi arbitrariamente con i numero di processori, grande scalabilità
- Si possono usare processori normali
- Ogni processore può facilmente accedere alla sua memoria senza interferenze
- Ottimo rapporto efficienza/costo

Contro:

- Chi programma deve organizzare e gestire lo scambio dei dati tra i processori
- Potrebbe essere difficile dividere i dati in modo efficiente
- Tutti gli accessi non locali avranno un costo che potrebbe non essere uniforme

Una MPP (Massively Parallel Processor) è un singolo computer con tanti processori collegati in rete, ciascuno associato alla sua memoria. La rete deve essere specializzata e deve avere un'alta velocità, ovvero bassa latenza e grande capacità di banda.

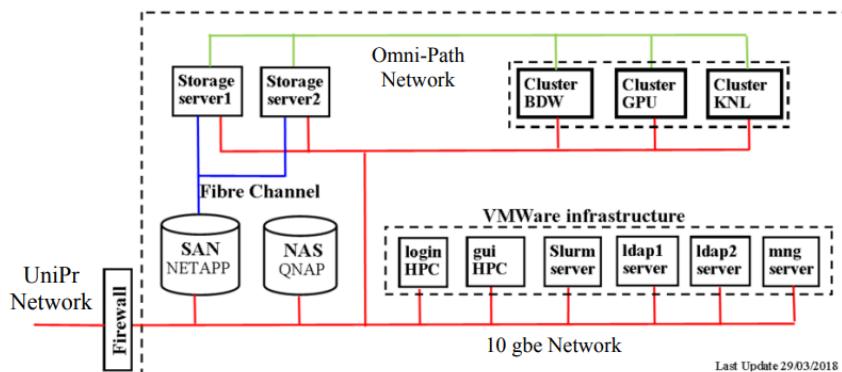
APPROCCIO MISTO

In un cluster ad alte prestazioni troviamo dei nodi collegati attraverso la rete (distribuzione), ma all'interno di un nodo troviamo una CPU che ha più unità di calcolo con una memoria locale (condivisione).

RETE

La topologia e l'efficienza della rete ha un grandissimo impatto sulle prestazioni.

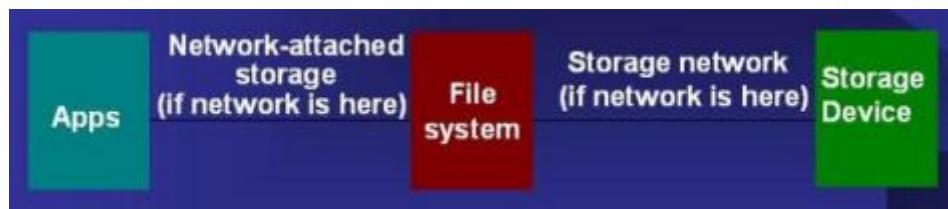
UNIPR



Ci sono 3 cluster (BDW, GPU e KNL), una rete OmniPath ad alte prestazioni che collega i nodi e li mette in connessione allo storage, collegati a server SAN e NAS, c'è poi una rete Ethernet a basse prestazioni che collega i nodi ad una serie di macchine virtuali che offrono servizi di gestione (login, code, accesso al file system, etc.).

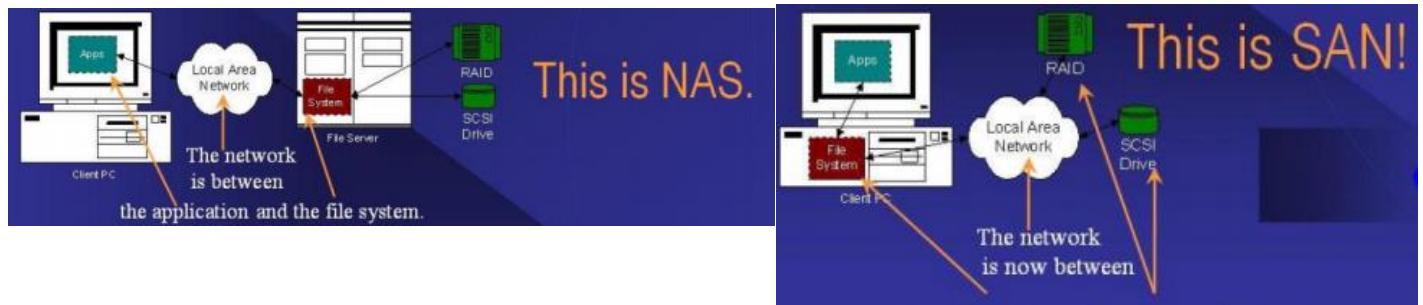
STORAGE (aggiungi info tramite i file IBM)

A seconda di come organizziamo le informazioni e dove si trova la rete tra app, file system e storage può essere SAN (Storage Area Network) o NAS (Network Attached Storage).



Se la rete si trova tra il livello utente e il file system si parla di NAS, in cui l'utente chiede via rete un certo file e dopo aver attraversato la rete, un altro server, contenente i metadati, restituirà le informazioni. Un dispositivo NAS è generalmente composto da un processore più un disco di storage, collegato poi con una rete TCP/IP a cui si accede utilizzando particolari protocolli per il file sharing. Le richieste di accesso ai dati ricevute da un NAS sono poi tradotte dal processore interno in richieste al device.

Per quanto riguarda SAN, la parte di file system e di memorizzazione sono separate. In questo caso i dati sono distribuiti all'interno della rete, ma il file system è scollegato e separato dalla rete. SAN rappresenta quindi un approccio più scalabile e flessibile, ottimizzato per le performance. Lo storage risiede su una rete dedicata, offrendo una connessione per i processori su quella rete.



MPI

Ci sono più macchine distinte interconnesse con una rete ad alte prestazioni, semplificando possiamo avere anche più processi sulla stessa macchina che comunicano tramite un protocollo (senza rete). Assumendo di avere più macchine collegate nella rete, avremo un memoria per ciascun nodo, per accedere alle informazioni sulle altre macchine, avrà bisogno di un meccanismo di comunicazione esplicito, che va quindi gestita esplicitamente (come, quando e gestione dei relativi problemi). Il metodo classico è quello del passaggio di messaggi che contengono i dati e vengono spediti tra i vari modi nella rete da un'origine ad un destinatario.

Ogni processo è responsabile per l'accesso sulla propria memoria, tramite il questo protocollo il processo decide quali dati inviare e a chi.

Il protocollo è MPI, una libreria che ci descrive un modello di scambio messaggi senza darci un'implementazione specifica né una specifica per il compilatore, è quindi una libreria di specifiche, non di implementazioni, in particolare MPI non è un linguaggio.

Obiettivi:

- Permettere di progettare un'interfaccia a livello di codice, senza preoccuparci a basso livello, delegando poi all'implementazione dell'interfaccia gli aspetti più concreti
- Permettere una comunicazione efficiente: evitando la copia, permettendo lo scambio del messaggio e la computazione contemporaneamente, permettere di far gestire la comunicazione ad un co-processore (comunicazione che si adatta all'hw), permettere implementazioni in un ambiente eterogeneo.
- Avere delle comunicazioni affidabili, senza che l'utente si debba preoccupare degli errori che saranno gestiti internamente
- Interfaccia che può essere implementata in modo indipendente dall'hw
- Interfaccia che può essere implementata in modo indipendente dal linguaggio utilizzato

Cosa è incluso nello standard

- | | |
|--|--|
| <ul style="list-style-type: none">• Point-to-point communication• Datatypes• Collective operations• Process groups• Communication contexts• Process topologies• Environmental Management and inquiry | <ul style="list-style-type: none">• The info object• Process creation and management• One-sided communication• External interfaces• Parallel file I/O• Language Bindings for Fortran, C and C++• Profiling interface |
|--|--|

Implementazioni: ce ne sono di diverse, noi usiamo Open MPI (free), ma ce ne sono anche a pagamento.

Per usare MPI si usa un compilatore apposta (*mpicc*) che si occupa di andare ad integrare tutte le librerie di MPI. Con MPI supponiamo di avere più processi che girano, per gestirli usiamo *mpirun* per lanciare un'eseguibile in multiple copie su potenzialmente diversi host.

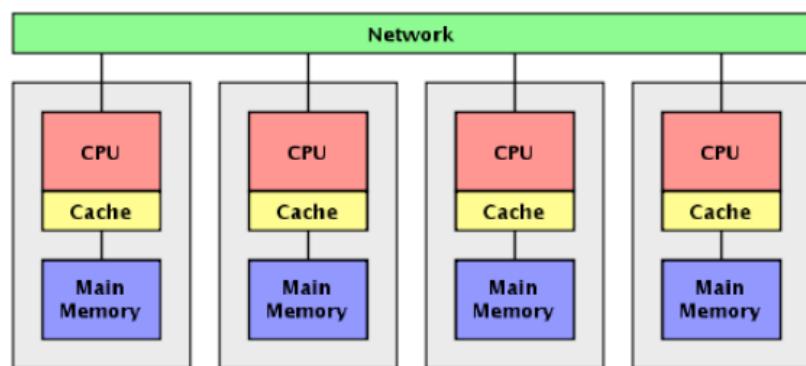
Compiling: *mpicc -o myprog myprog.c*

Running: *mpirun [-np] [-hostfile]*

In questo modo *mpirun* ha modo di avere una visione globale dei processi che girano, possiamo lanciare in parallelo più processi che vorranno collaborare durante la loro vita.

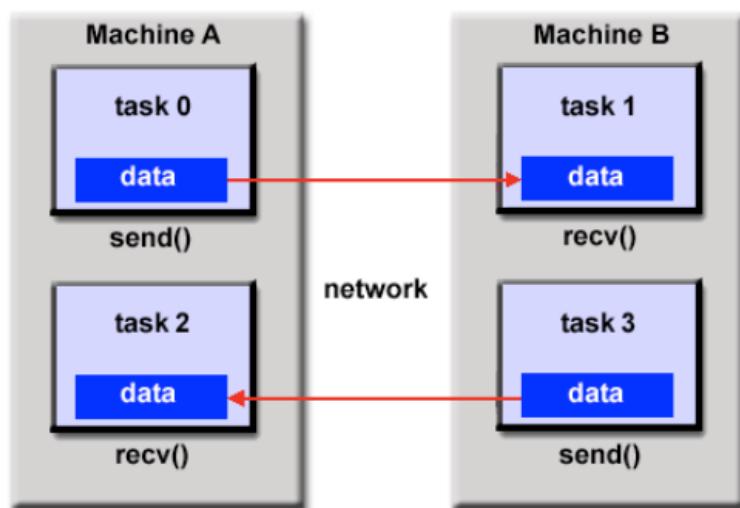
Hostfile: si va ad elencare il/gli IP delle macchine che ospitano i processi e possiamo fornire informazioni rispetto a quanti processi possono girare su un certo nodo, quanti processori ci sono a disposizione o la distribuzione dei processi sui vari processori.

Funzionamento:



Task = processo = istanza di un programma che sta girando. Si possono avere più task in esecuzione. Scrivo un solo programma che girerà in n istanze e saprà dialogare con gli altri task.

In *openmp*, avevamo un unico processore con più thread che si affidava alla memoria condivisa, con *MPI* abbiamo più processori e dobbiamo scambiare messaggi esplicitamente. La comunicazione tra più processi si basa quindi su una comunicazione esplicita (devo dichiarare sia mittente e che destinatario tramite *send()* e *recv()*).



Ogni processo in gioco è solitamente un'istanza di uno stesso programma, quindi ogni istanza del programma è eseguito su un insieme di dati differenti (data parallelism), in questo caso

si parla di **SPMD** (Single Program, Multiple Data) un singolo programma, lanciato su più istanze, ciascuna che si occupa di una parte di dati, se il livello non è di parallelismo imbarazzante, le varie istanze (identificate univocamente da un numero, chiamato *rank*, tra 0 e n-1 assegnato da MPI) dovranno scambiare messaggi tra loro. **MPMD** (Multiple Program, Multiple Data) è un'estensione che si può riportare banalmente a SPMD.

I messaggi sono composti da due parti:

- "Busta", è un header (metadati): sorgente (rank mittente), destinazione (rank destinatario), tag (ID del messaggio), communicator (in che contesto stiamo spendendo il messaggio, è l'insieme di rank che possono comunicare questo tipo di messaggio)

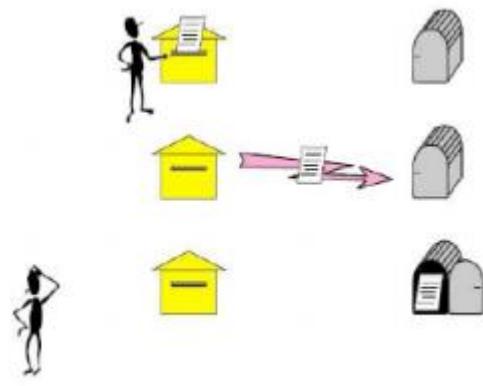
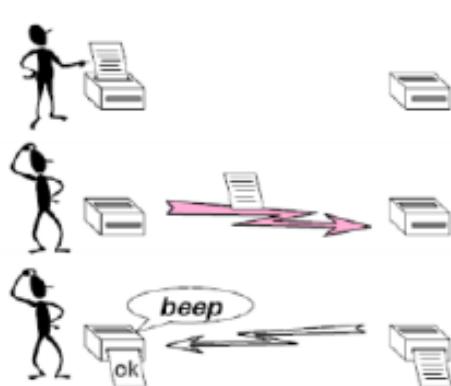
- **Corpo (dati effettivi)**: tipo di dati, lunghezza, buffer

Tipi di dati:

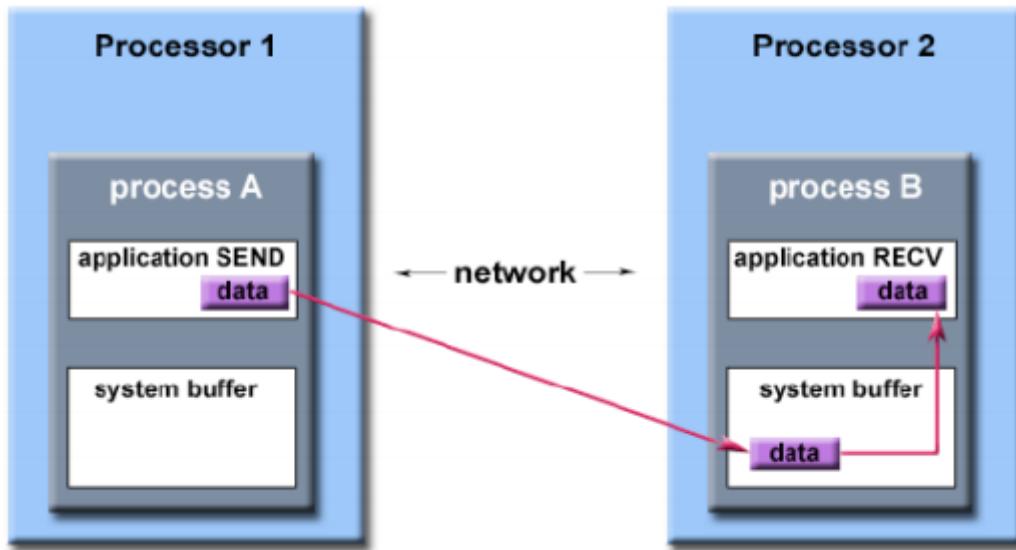
MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Tipi di comunicazione:

- **Punto a punto**: è la più semplice, coinvolge solo 2 processi, può essere **sincrona** (il mittente aspetta la risposta del ricevente per continuare) o **asincrona** (il mittente invia il messaggio e continua a fare altro).



Buffering: è implementato differentemente in ciascuna libreria MPI



Ci permette ad esempio di fare una **ricezione asincrona**, esiste sia per il mittente che per il ricevente, **è limitato** (si può riempire) e l'utente non ha il **controllo di questo buffer**.

Esempio: sender carica i dati da inviare nel buffer, continua a fare le sue attività e MPI si occupa di inviare il buffer.

Operazioni, possono essere non bloccanti (dopo la chiamata il processo può continuare) o bloccanti (dopo la chiamata il processo si blocca). Ovviamente è possibile rendere bloccanti le operazioni non bloccanti in un secondo momento per ad esempio fare un test o aspettare.

Operazioni bloccanti: send sincrona, buffered send (asincrona), send standard (asincrona), ready send, recv, sendrecv (spedisco e contemporaneamente ricevo)

Operazioni non bloccanti: come sopra, ma il mittente non si blocca, possono essere rese bloccanti in un secondo momento. Bisogna controllare che i buffer siano utilizzabili dopo la fine della comunicazione usando test o wait.

- Comunicazioni collettive (più di due processi)

Operazioni collettive utili: barriera di sincronizzazione, movimento dei dati: comunicazioni collettive (broadcast, scatter, gather), reduction: calcoli collettivi (minimo, massimo, somma, operazioni logiche come OR o AND, operazioni definite da utente).

PROGRAMMARE CON MPI

Funzioni:

error = MPI_Xxxxx(parameter, ...); MPI_Xxxxx(parameter, ...);

MPI_ è un namespace riservato a costanti e routine MPI, dopo il prefisso solo la prima lettera è maiuscola, tutte le funzioni ritornano un intero, i nomi delle costanti sono tutte maiuscole.

Header file: #include <mpi.h>

Communicator: insieme di processi che possono comunicare tra loro, ha un nome, una dimensione e ciascun processo può essere identificato unicamente. Il comunicator di default (MPI_COMM_WORLD,) include tutti i processi creati e dà la possibilità di parlare con tutti.

Rank: un processo può conoscerlo chiamando MPI_Comm_rank(MPI_Comm comm, int *rank)

Size: un processo può conoscerlo chiamando MPI_Comm_size(MPI_Comm comm, int *size). Se il communicator è quello di default, la size ci dirà il numero di processi che stanno girando contemporaneamente.

Inizializzare MPI: int MPI_Init(int *argc, char ***argv) che inizializza il communicator di default

Uscire da MPI: int MPI_Finalize()

Esempio: Hello, World!

```
#include <stdio.h>
#include "mpi.h"

#define MASTER 0

int main (int argc, char **argv)
{
    int numtasks, taskid, len;
    char hostname[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
    MPI_Get_processor_name(hostname, &len);
    printf ("Hello from task %d on %s!\n", taskid, hostname);

    if (taskid == MASTER)
        printf("MASTER: Number of MPI tasks is: %d\n", numtasks);

    MPI_Finalize();
}
```

Questo processo verrà eseguito contemporaneamente da tutti i processi, se saranno su macchine diverse, l'output sarà su stdout diversi.

La compilazione crea un unico eseguibile (parametrico), ogni comando è eseguito da ciascun processo indipendentemente, il sistema runtime controlla come l'eseguibile è lanciato sui nodi, come i processi sono creati e gestisce lo standard output/error.

PROGRAMMAZIONE: COMUNICAZIONE PUTO PUNTO

```
int MPI_Send(void *buf,  
            int count,  
            MPI_Datatype datatype,  
            int dest,  
            int tag,  
            MPI_Comm comm)
```

Dove buf è il puntatore al messaggio inviato, count è il numero di elementi nel messaggio, datatype è il tipo di elementi del messaggio, dest è il rank del destinatario, tag è un intero, comm è il communicator

```
int MPI_Recv(void *buf,  
            int count,  
            MPI_Datatype datatype,  
            int source,  
            int tag,  
            MPI_Comm comm,  
            MPI_Status *status)
```

Dove buf è il puntatore all'array dove il messaggio ricevuto deve essere salvato, count è il numero di elementi del messaggio, datatype specifica il tipo del messaggio, source specifica il rank del mittente, tag specifica il tag, comm è il communicator, status contiene le informazioni riguardo i metadati del messaggio che si andrà a ricevere.

Una comunicazione ha successo quando:

- il mittente e il ricevente specificano un rank valido e i due coincidono
- i communicator specificati da mittente e ricevente coincidono
- i tag coincidono
- i tipi di dato coincidono
- il buffer del ricevente è abbastanza capiente

Per semplificare possiamo usare le wildcard; posso usare nel receiver una costante speciale (MPI_ANY_SOURCE) in MPI_Recv quando si vuole rimanere in ascolto di un qualsiasi messaggio, senza conoscerne a priori il rank del mittente. Stessa cosa per il tag utilizzando MPI_ANY_TAG. Non esistono wildcard per il communicator.

Per ottenere il numero di elementi di uno specifico tipo di dati in un messaggio ricevuto, bisogna chiamare la seguente funzione passando come parametro lo status.

```
int MPI_Get_count(MPI_Status status, MPI_Datatype datatype, int *count)
```

Una proprietà che viene garantita dalla specifica di MPI è la preservazione dell'ordine dei messaggi con la stessa sorgente, communicator, tag e destinatario, di modo che venga garantito l'ordine temporale, se due messaggi sono spediti nelle stesse condizioni, allora l'ordine di spedizione sarà mantenuto.

Starvation e deadlock sono in mano al programmatore che deve gestirle e cercare di evitarle (due send e una receive bloccanti sullo stesso task causano starvation -> ogni send deve avere la sua receive)

Esempio: send/receive (bloccanti) di un intero

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char **argv)
{
    MPI_Status status;
    int rank, size;
    /* data to communicate */
    int data_int;
    /* initialize MPI */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (rank == 0) {
        data_int = 10;
        MPI_Send(&data_int, 1, MPI_INT, 1, 666, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Recv(&data_int, 1, MPI_INT, 0, 666, MPI_COMM_WORLD, &status);
        printf("Process 1 receives %d from process 0.\n", data_int);
    }
    MPI_Finalize();
    return 0;
}
```

Esempio: send/receive un array di float

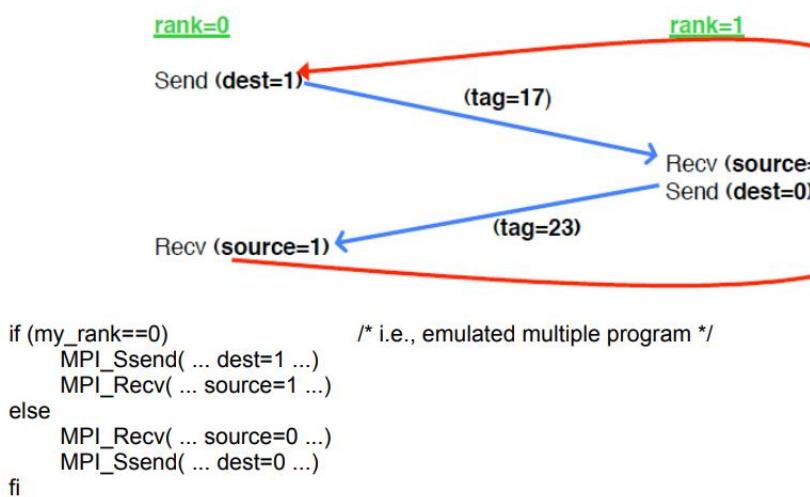
```
#include <stdio.h>
#include "mpi.h"
#define MSIZE 10

int main(int argc, char **argv)
{
    MPI_Status status;
    int rank, size;
    int i, j;
    /*data to communicate */
    float a[MSIZE];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        for (i = 0; i<MSIZE; i++)
            a[i] = (float) i;
        MPI_Send(a, MSIZE,MPI_FLOAT, 1, 666, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Recv(a, MSIZE, MPI_FLOAT, 0, 666, MPI_COMM_WORLD, &status);
        printf("Process 1 receives the following array from process 0.\n");
        for (i = 0; i<MSIZE; i++)
            printf("%6.2f\n", a[i]);
    }
    MPI_Finalize();
    return 0;
}
```

Misurare tempo di esecuzione: double MPI_Wtime(void);

Conoscere risoluzione del timer: double MPI_Wtick(void);

Esempio: PING-PONG (alternanza stretta)



Codice:

```

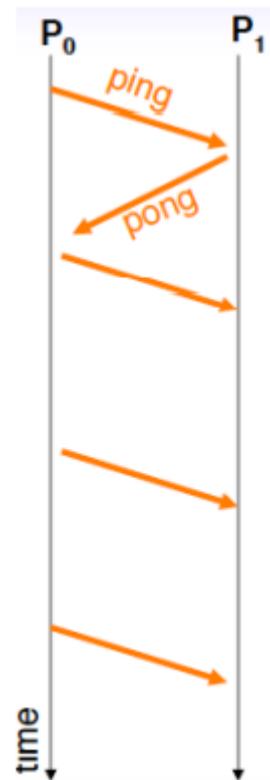
#include <stdio.h>
#include <mpi.h>
#define proc_A 0
#define proc_B 1
#define ping 100
#define pong 101
#define number_of_messages 50
#define length_of_message 1

int main(int argc, char *argv[])
{
    int my_rank;
    float buffer[length_of_message];
    int i;
    double start, finish, time;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank == proc_A)
    {
        start = MPI_Wtime();
        for (i = 1; i <= number_of_messages; i++)
        {
            MPI_Ssend(buffer, length_of_message, MPI_FLOAT, proc_B, ping,
                      MPI_COMM_WORLD);
            MPI_Recv(buffer, length_of_message, MPI_FLOAT, proc_B, pong,
                      MPI_COMM_WORLD, &status);
        }
        finish = MPI_Wtime();
        time = finish - start;
        printf("Time for one message: %f seconds.\n",
               (float)(time / (2 * number_of_messages)));
    }

    else if (my_rank == proc_B)
    {
        for (i = 1; i <= number_of_messages; i++)
        {
            MPI_Recv(buffer, length_of_message, MPI_FLOAT, proc_A, ping,
                      MPI_COMM_WORLD, &status);
            MPI_Ssend(buffer, length_of_message, MPI_FLOAT, proc_A, pong,
                      MPI_COMM_WORLD);
        }
    }

    MPI_Finalize();
}
  
```



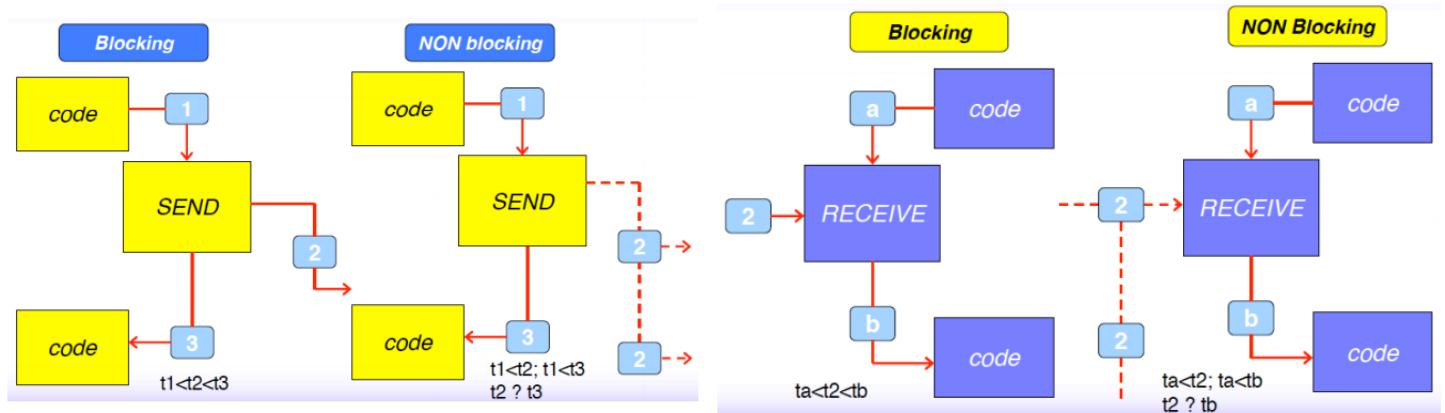
COMUNICAZIONI COMPLETATE

Una comunicazione si dice **completata localmente** quando un processo ha finito di fare tutte le attività collegate alla parte di comunicazione che gli competono, ciò significa che il processo può eseguire le comunicazioni successive alla sua chiamata. Questo perché fino a quando non si ha completato localmente la chiamata, il buffer del processo è ancora necessario e in uso.

Una comunicazione si dice **completata globalmente** se tutti i processi coinvolti nella comunicazione hanno finito, ovvero se e solo se la comunicazione è completata localmente su tutti i processi coinvolti.

La fase di completamento di una SEND dipende dalla dimensione del massaggio (che è buffered per le piccole dimensioni, sincrono per messaggi le cui dimensioni eccedono quelle del buffer). Bisogna prestare attenzione a deadlock, utilizzando nel caso anche send non bloccanti.

Comunicazioni bloccanti e non bloccanti



Varianti send:

- **sincrona:** completata quando il buffer da mandare può essere riutilizzato (dati copiati tutti in un altro buffer) e la ricezione del messaggio è iniziata
- **buffer:** completata quando il messaggio è stato completamente copiato nel buffer di trasmissione (terminato localmente)
- **standard:** completata quando l'application buffer (quello da mandare) può essere riusato, ma non sappiamo nulla della receive

Funzioni:

Blocking:

- Synchronous send → **MPI_Ssend**
- Buffered send → **MPI_Bsend**
- Standard send → **MPI_Send**
- Ready send → **MPI_Rsend**
- Receive → **MPI_Recv**

Nonblocking:

- Synchronous non blocking send → **MPI_Issend**
- Buffered non blocking send → **MPI_Ibsend**
- Standard non blocking send → **MPI_Isend**
- Ready non blocking send → **MPI_Irsend**
- Non blocking receive → **MPI_Irecv**

La **i** sta per immediato.

COMUNICAZIONI BLOCCANTI

- Synchronous send: MPI_Ssend

Questa chiamata manda il messaggio e bocca finché l'application buffer (informazioni da spedire, gestito dall'utente) può essere utilizzato di nuovo e modificato e il ricevente ha iniziato a ricevere il messaggio. Questa chiamata ci farà aspettare parecchio tempo, mittente e ricevente sono sincronizzati, tuttavia c'è rischio di deadlock (il programmatore deve programmare per evitarlo) e di andare in idle.

- Buffered send: MPI_Bsend

Blocca come quella di prima, ma finisce non appena il processo ha copiato il messaggio in un buffer speciale chiamato transfer buffer. Il costo in termini di tempo è quello della copia del buffer dal lato del mittente. Dobbiamo definire e gestire esplicitamente il buffer. Mittente e destinatario non sono sincronizzati.

- Standard send: MPI_Send

Bloccante, questa chiamata ritorna il controllo quando il messaggio è stato mandato e possiamo modificare l'application buffer.

- Ready send: MPI_Rsend

La chiamata restituisce il controllo quando si l'altro processo ha iniziato ad eseguire la receive. Le chiamate devono essere bilanciate, è abbastanza rischiosa.

- Send-receive: MPI_Sendrecv

Si crea una doppia comunicazione, tutte e due le chiamate si appoggiano allo stesso communicator, con eventualmente tag differenti. Si ha una send e una receive, entrambe bloccanti, ciascuna con il suo buffer, eventualmente con lunghezze e datatype diversi. Comodo in quelle situazioni in cui vogliamo sia spedire che ricevere.

COMUNICAZIONI NON BLOCCANTI

Tipicamente composta da 3 fasi:

1. Si definisce l'intenzione e si manda la chiamata (send o receive)
2. La chiamata rilascia subito il controllo, possiamo fare altre attività
3. Aspettiamo l'eventuale fine della comunicazione.

È molto utile, nasconde i tempi delle comunicazioni impiegando il tempo di attesa per fare attività che possono essere anticipate; riduce gli effetti di tempi di latenza della rete; non c'è il rischio di deadlock. Tuttavia le operazioni si fanno più complicate.

- Synchronous nonblocking send: MPI_Issend(buf, count, datatype, dest, tag, comm, &request_handle) e poi MPI_Wait(&request_handle, &status)

Tra le due chiamate posso eseguire del codice mentre MPI sta completando la send, poi eseguirò la wait che è bloccante. Ovviamente prima della wait devo eseguire operazioni che non dipendono dai dati dell'applicazione. Tra le due chiamate il buffer non va usato perché

non aggiornato. Se faccio le due chiamate immediatamente una dopo l'altra ottengo lo stesso effetto della Ssend. L'handle è un codice interno che si usa per riferirsi ad una send precisa. Al posto di wait (bloccante) potrei usare anche una serie di test ripetuti per tener monitorata la comunicazione senza aspettare necessariamente la fine delle attività.

- Standard nonblocking send: MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)

Si prepara a fare la send e ci ridà il controllo immediatamente. Inizializza la send, identifica un'area di memoria che verrà usata come buffer, viene ritornato un riferimento alla chiamata specifica. Il buffer utilizzato non sarà utilizzabile finché la send non sarà completata. È possibile prevedere una receive bloccante in risposta a quella chiamata, oltre a quella bloccante.

- Nonblocking receive: int MPI_Irecv(void buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request request)

Inizializza la receive, identifica la memoria che sarà utilizzata come buffer e ritorna il controllo immediatamente, senza aspettare che il messaggio sia copiato completamente. Il buffer non può essere utilizzato nel frattempo e per controllare lo stato della receive viene utilizzato l'handle (request).

- Test

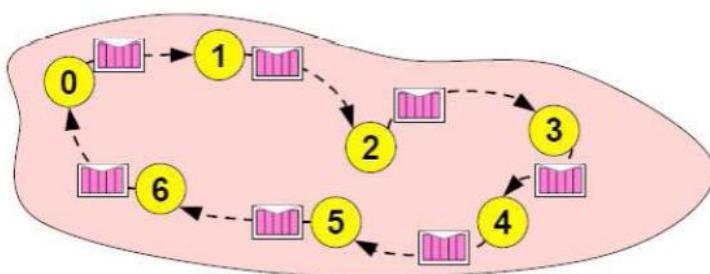
Facciamo un test se vogliamo conoscere lo stato della chiamata (vero se la comunicazione ha terminato, falso altrimenti), facciamo una wait (bloccante) che aspetta che finché il test non ritornerebbe vero, ovvero finché la comunicazione non ha terminato.

DEADLOCK

Esempio di deadlock (codice e grafico):

```
MPI_Ssend(..., right_rank, ...);
```

```
MPI_Recv(..., left_rank, ...);
```



Per evitare deadlock:

1. Cambiare ordine delle chiamate
2. Usare operazioni non bloccanti
3. Usare buffer espliciti e rilasciare il controllo appena copiato il buffer

COMUNICAZIONI NON BLOCCANTI MULTIPLE

È possibile avere una serie di comunicazioni non bloccati nello stesso tempo, quindi la `wait` e la `test` si adattano a ciò ed è possibile eseguirle per uno/tutti/alcuni messaggi tramite le chiamate `MPI_Waitany` / `MPI_Testany`; `MPI_Waitall` / `MPI_Testall`; `MPI_Waitsome` / `MPI_Testsome`.

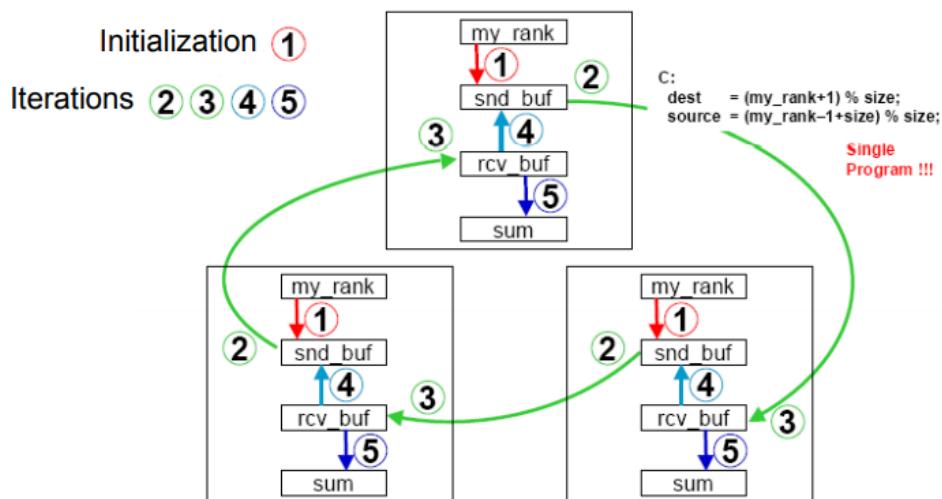
PROPAGAZIONE DEL MESSAGGIO SU UN RING

Supponiamo di avere un set di n processi organizzati come un anello.

All'inizio, ogni processo salva il suo rank in una variabile chiamata `snd_buf`.

Poi, ciascun processo per n volte: invia la `snd_buf` al suo vicino di destra e riceve una valore, lo aggiunge alla variabile di somma e lo copia sul `snd_buf`

Se si usa la `Issend` (per evitare deadlock) otteniamo questo comportamento:



Codice (dopo le varie inizializzazioni):

```
/* ... this SPMD-style neighbor computation with modulo has the same meaning as: */
/* right = my_rank + 1; */
/* if (right == size) right = 0; */
/* left = my_rank - 1; */
/* if (left == -1) left = size-1;*/
sum = 0;
snd_buf = my_rank;
for( i = 0; i < size; i++)
{
    MPI_Issend(&snd_buf, 1, MPI_INT, right, to_right, MPI_COMM_WORLD, &request);
    MPI_Recv(&recv_buf, 1, MPI_INT, left, to_right, MPI_COMM_WORLD, &status);
    MPI_Wait(&request, &status);
    sum += recv_buf;
    snd_buf = recv_buf;
}
printf ("PE%i:\tSum = %i\n", my_rank, sum);
MPI_Finalize();
```

COMUNICAZIONI COLLETTIVE

MPI offre funzioni che implementano comunicazioni che riguardano più processi, così da evitare che il programmatore debba implementarle combinando comunicazioni punto-punto. Tutti i processi devono comunicare, le comunicazioni possono essere bloccanti o no, non ci sono tag e i buffer di ricezione devono corrispondere esattamente alla grandezza del messaggio. Quelle che seguono sono bloccanti.

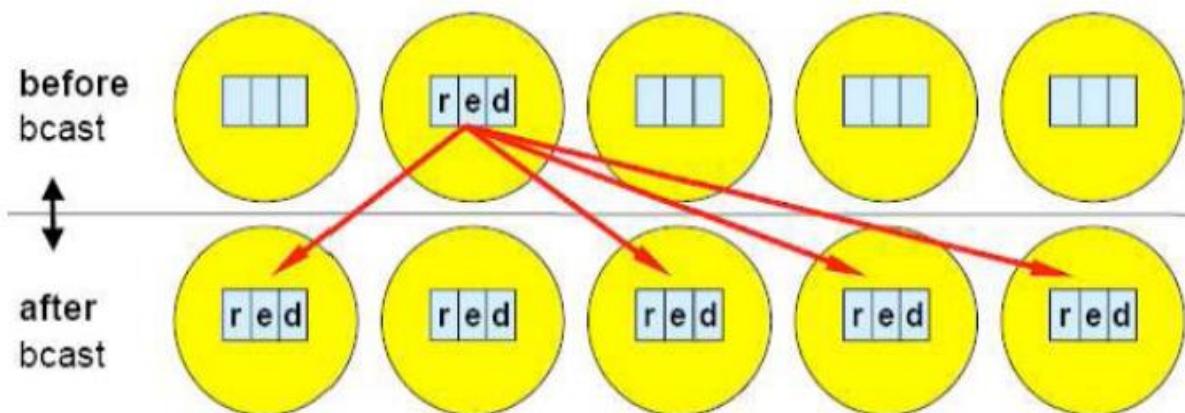
Ci sono tre classi: tutti-a-uno, uno-a-tutti, tutti-a-tutti.

- **Barriera di sincronizzazione:** int MPI_BARRIER(MPI_Comm comm)

La creiamo quando vogliamo che tutti i processi siano allo stesso livello in un certo momento.

- **Broadcast:** int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

Tutti i rank nel communicator riceveranno la stessa informazione, tranne il mittente. Molto più efficiente che creare tutte le send punto-punto. Garantisce anche una barriera di sincronizzazione



Esempio:

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    int rank;
    int buf;
    const int root = 0;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == root) {
        buf = 777;
    }

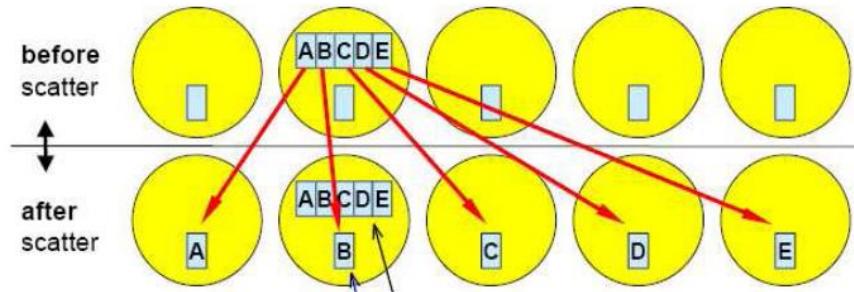
    printf("[%d]: Before Bcast, buf is %d\n", rank, buf);

    /* everyone calls bcast, data is taken from root and ends up in everyone's buf */
    MPI_Bcast(&buf, 1, MPI_INT, root, MPI_COMM_WORLD);

    printf("[%d]: After Bcast, buf is %d\n", rank, buf);

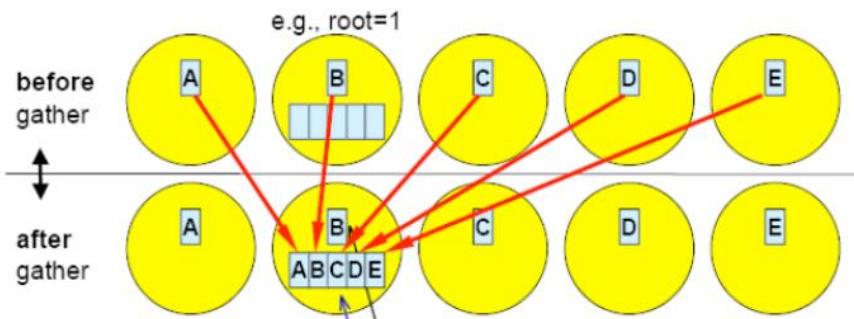
    MPI_Finalize();
    return 0;
}
```

- **Scatter**: int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)



Il messaggio è diviso in pezzi e ciascuno ne riceve una parte, anche il root.

- **Gather**: int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)



Simmetrico di gather. Un processo root ricolleziona tutti i pezzi di messaggio in un unico messaggio. Il receive buffer dev'esser grande come la somma di tutti i messaggi.

- **Riduzione**: int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

Come gather ma con un'applicazione (associativa e commutativa, ad esempio somma, prodotto, minimo, massimo, etc.) eseguita sui dati che arrivano. La reduction è utile perché ci evita di fare un ciclo sul root di tutti i valori ottenuti durante a gather (operazione che potrebbe essere inefficiente).

Esempio:

Reduction:

Example: 1-norm

```
#include <mpi.h>
#include <stdio>
#include <math.h>

void main(int argc, char* argv[]) {
    int root=0, p, myid;
    float sendbuf, recvbuf;
    MPI_Op myop;
    int commute = 0;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Op_create((MPI_user_function *)onenorm, commute, &myop);
    sendbuf = myid*((int)pow((double)-1,myid));
    MPI_BARRIER(MPI_COMM_WORLD);
    MPI_Reduce(&sendbuf, &recvbuf, 1, MPI_FLOAT, myop,
               root, MPI_COMM_WORLD);
    if (myid == root)
        printf("The operation yields %f\n", recvbuf);
    MPI_Finalize();
}
```

Comunicazione collettive non bloccanti: hanno l'handle che verrà utilizzato poi per fare wait e test.

- `int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request)`
- `int MPI_Ibcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm, MPI_Request *request)`
- `int MPI_Iscatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)`
- `int MPI_Igather(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)`

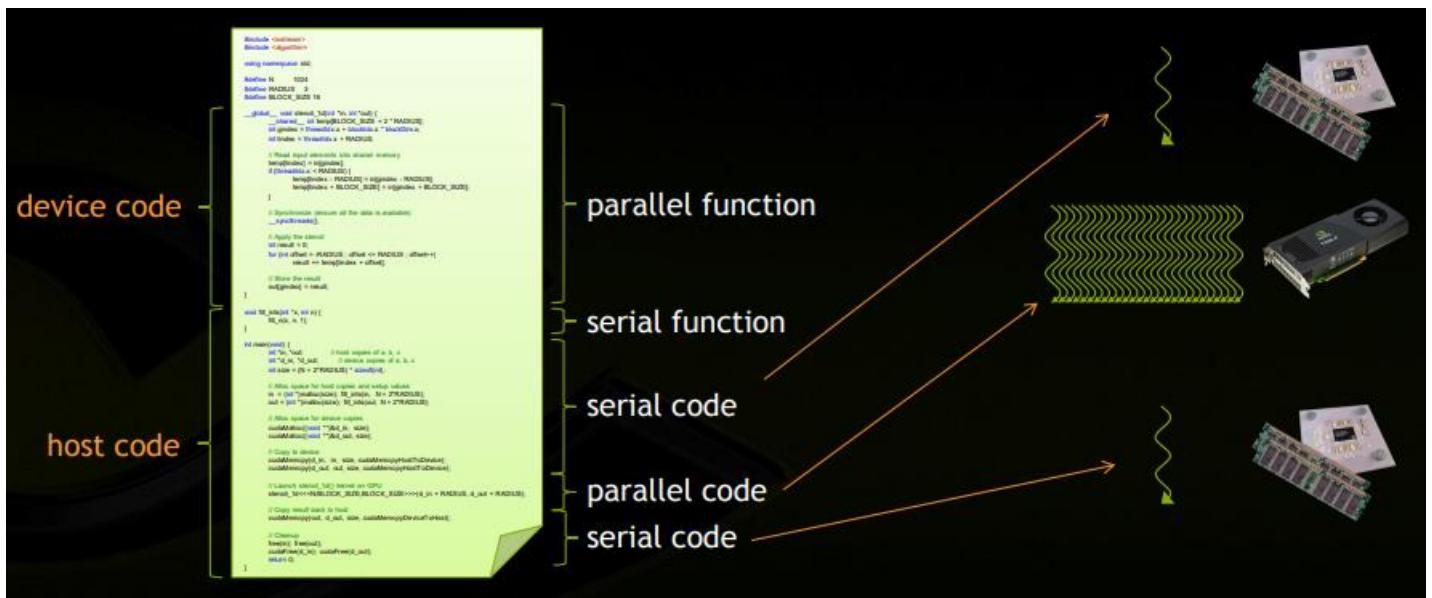
PROGRAMMAZIONE CON GPU

Le GPU nascono per rendering grafico, c'è stata una forte crescita dell'hw, sviluppando molto le pipeline di rendering grafico, per questo si sono sfruttati questi motori per fare conti generali, non legati alla grafica.

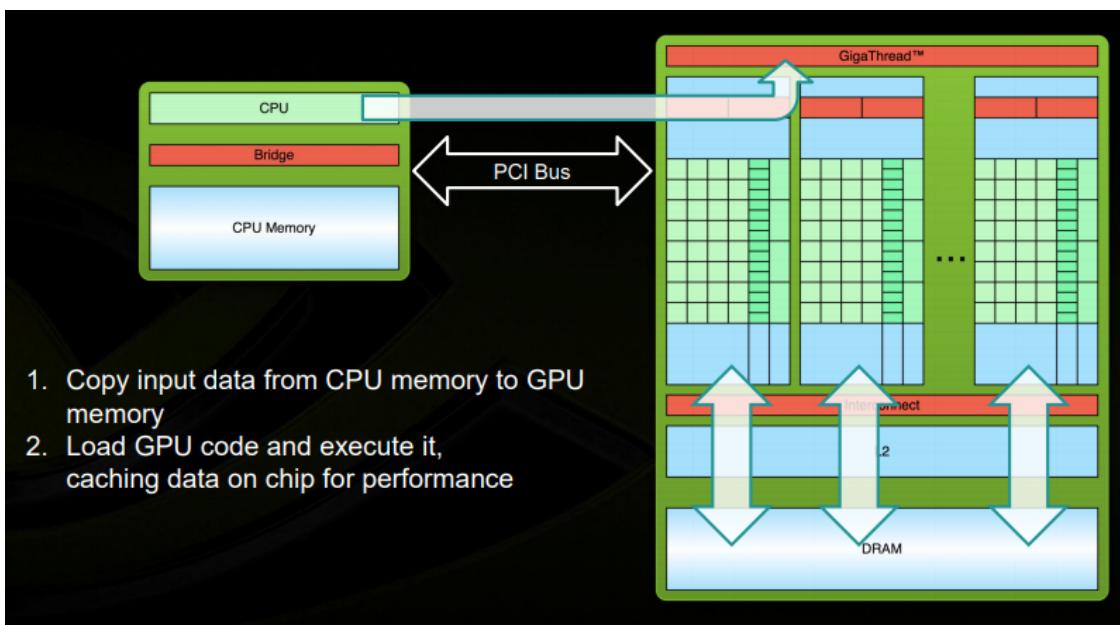
Nasce l'architettura CUDA per fare calcolo general purpose con ottime performance. CUDA è un framework per diversi linguaggi, noi vedremo quello per il C/C++.

PROGRAMMAZIONE ETEROGENEA: utilizzare due o più oggetti di natura diversa.

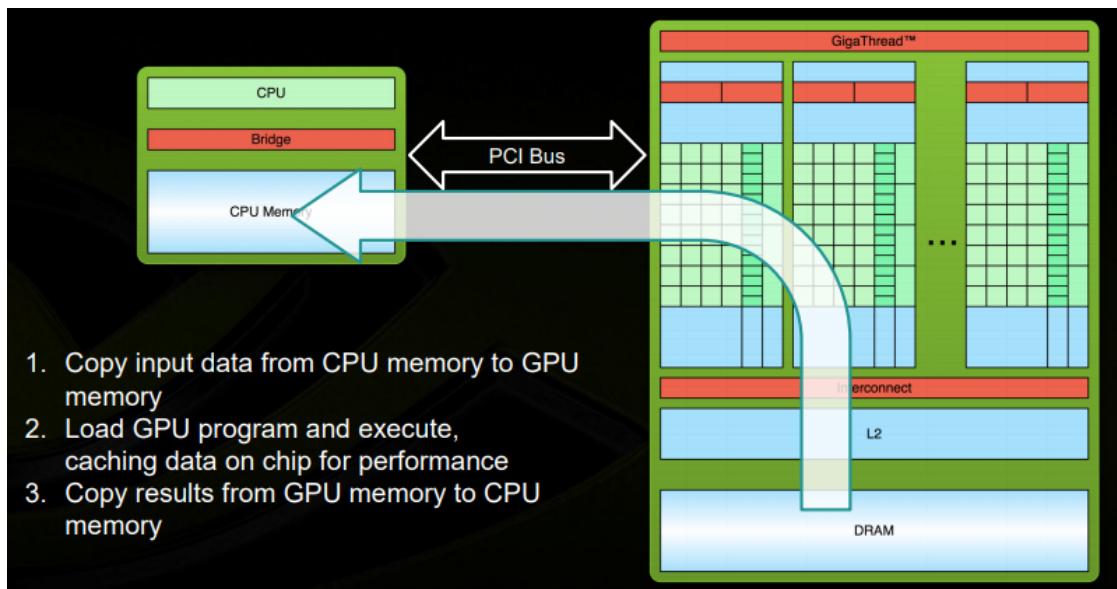
Per quanto riguarda l'hw abbiamo un CPU (host) con la sua memoria RAM e una GPU(device) con la sua memoria RAM. L'host è in relazione master-slave con il device, la GPU non è completamente indipendente, ma viene controllata dalla CPU (anche se nell'evoluzione, si sta perdendo questa divisione).



Troviamo il codice dedicato all'host (solitamente seriale) e il codice dedicato al device (solitamente parallelo). Quella sopra è una tipica gestione "a fisarmonica" in cui si alternano fasi di uso di CPU e di GPU.



La **prima fase** è quella di copiare i dati dalla memoria dell'host a quella del device, non è possibile per la GPU avere un accesso diretto alla memoria della CPU. Poi l'host farà una copia del codice che la GPU deve eseguire al dato comando della CPU (elaborazione in parallelo vera e propria).



Si fa poi una **comunicazione all'indietro** dei risultati dalla GPU alla CPU.

HELLO WORLD!

Il compilatore dell'NVIDIA comprende il C e quindi chiama all'occorrenza gcc, grazie anche alle parole chiave utilizzate, manipola invece in modo diverso il codice che è per la GPU.

```
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

- **`__global__`**: specifica che la funzione è chiamata dal codice host e indica che la funzione girerà esclusivamente sul device, anche se sarà chiamata dall'host.
 - **`mykernel<<<1,1>>>()`**: si vanno ad indentificare due interi che descrivono la geometria dei thread che vanno a lavorare il codice parallelo. Definisce il livello di parallelismo.

PARALLELISMO

Somma di due array su un terzo array. Add() gira sul device, quindi a,b,c devono puntare alla memoria del device.

```
__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

Per quanto riguarda la memoria: La GPU non è in grado da sola di allocare la memoria: le chiamate sono fatte dall'host e la memoria è gestita dall'host. La GPU non può allocarsi da sola la memoria. Sia host che device possono usare solo i riferimenti alla propria memoria, non quelli, anche se passati o visibili, dell'altro dispositivo, poiché le due memorie sono entità separate. Per usare la memoria sul device usiamo le chiamate `cudaMalloc()`, `cudaFree()`,

• `cudaMemcpy()`, quest'ultima per spostare la memoria da un device all'altro o all'interno dello stesso dispositivo, tutte le chiamate sono fatte dall'host.

Esempio codice:

```
int main(void) {
    int a, b, c;                                // host copies of a, b, c
    int *d_a, *d_b, *d_c;                      // device copies of a, b, c
    int size = sizeof(int);

    // Allocate space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Setup input values
    a = 2;
    b = 7;

    // Copy inputs to device
    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<1,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

dove `add` è

```
__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

ESECUZIONE IN PARALLELO

Vediamo i gradi di parallelismo che si possono utilizzare all'interno di una GPU.

Chiamata del kernel:

```
add<<< 1, 1 >>>();  
      ↓  
add<<< N, 1 >>>();
```

Il primo numero **N**, indica quante volte andiamo ad eseguire in parallelo il codice. Ogni lancio parallelo della chiamata viene descritto come un *blocco di lavoro*, dove ciascun blocco esegue lo stesso identico codice in parallelo. La *griglia* è il raggruppamento di tutti i blocchi. L'idea è quella di avere lo stesso programma che poi verrà differenziato in base al numero di blocco. Per identificare il numero di blocco runtime si usa la costante **blockIdx.x** che restituisce l'intero che corrisponde al blocco di lavoro.

Esempio di addizione su un vettore:

```
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

On the device, each block can execute in parallel:



```
#define N 512  
int main(void) {  
    int *a, *b, *c; // host copies of a, b, c  
    int *d_a, *d_b, *d_c; // device copies of a, b, c  
    int size = N * sizeof(int);  
  
    // Alloc space for device copies of a, b, c  
    cudaMalloc((void **)&d_a, size);  
    cudaMalloc((void **)&d_b, size);  
    cudaMalloc((void **)&d_c, size);  
  
    // Alloc space for host copies of a, b, c and setup input values  
    a = (int *)malloc(size); random_ints(a, N);  
    b = (int *)malloc(size); random_ints(b, N);  
    c = (int *)malloc(size);  
  
    // Copy inputs to device  
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);  
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);  
  
    // Launch add() kernel on GPU with N blocks  
    add<<<N,1>>>(d_a, d_b, d_c);  
  
    // Copy result back to host  
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);  
  
    // Cleanup  
    free(a); free(b); free(c);  
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);  
    return 0;  
}
```

THREAD

Un blocco può essere diviso in thread concorrenti, avendo così una doppia suddivisione del lavoro che può essere suddiviso i tanti blocchi, a loro volta suddivisi in tanti thread. Vediamo l'esempio di prima ma con un solo blocco e più thread. Per indicare il thread usiamo threadIdx.x. In questo caso utilizziamo il secondo intero tra le tre parentesi angolate.

```
__global__ void add(int *a, int *b, int *c) {
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}

#define N 512
int main(void) {
    int *a, *b, *c;                      // host copies of a, b, c
    int *d_a, *d_b, *d_c;                // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);

    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N threads
    add<<<1,N>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

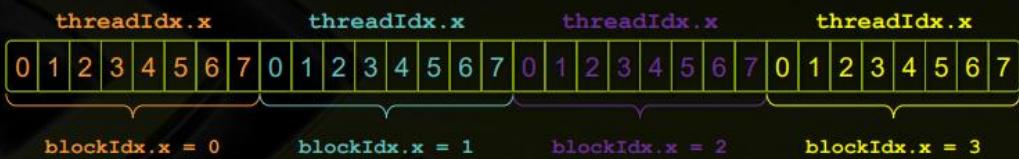
I thread possono fare cose che i blocchi non possono fare, quindi una divisione su due livelli ci permette di organizzare meglio il lavoro. Dovremo organizzare l'accesso ai dati in base al numero del thread/blocco e, in generale, riuscire a specializzare il codice in base al numero di thread/blocco.

COMBINARE BLOCCHI E THREAD

Bisogna organizzare i dati a cui ogni thread deve andare ad accedere.

No longer as simple as using `blockIdx.x` and `threadIdx.x`

- Consider indexing an array with one element per thread (8 threads/block)

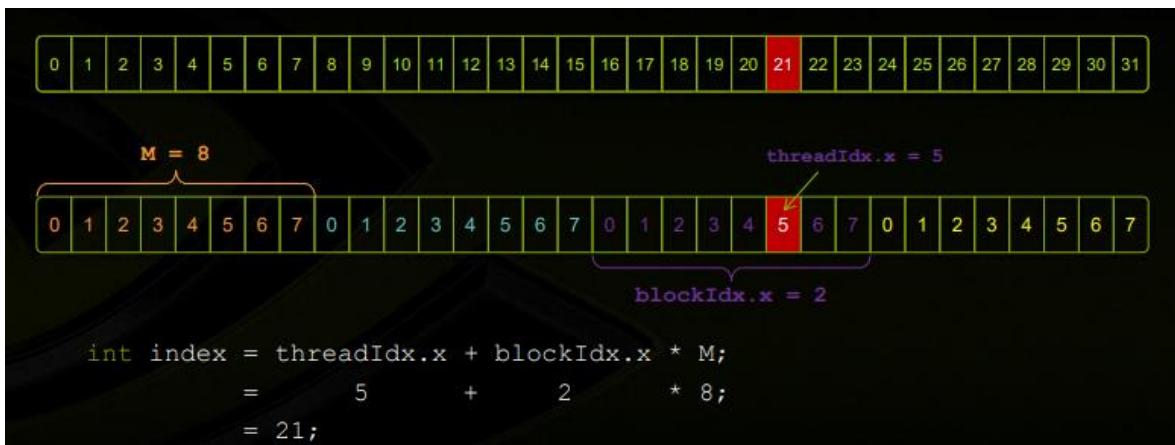


With M threads per block, a unique index for each thread is given by:

```
int index = threadIdx.x + blockIdx.x * M;
```

In questo scenario abbiamo 4 blocchi, ciascuno con 8 thread, per cui 32 thread che gireranno. Dobbiamo trovare un modo per accedere ai dati che sia facilmente gestibile, ad esempio quella mostrata nella figura in cui ogni pezzo di array è gestito da un blocco, all'interno del quale ci si muove tramite l'indice del thread, ciascun indice deve essere univoco. Nell'ultima riga della slide si ha un esempio di indicizzazione, in cui il thread indica l'indice interno al pezzo di array, mentre quando ci si sposta di blocco, avendo quel numero moltiplicato per M , ci si sposta di M posizioni, dove M è il numero di thread per blocco. Così facendo abbiamo un indicizzazione univoca e globale, in cui ogni thread ha un indice univoco.

Esempio sopra espanso con conversione in indice globale:



Possiamo usare `blockDim.x` al posto di M , che indica il numero di thread per blocco:

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

Nell'indicizzazione con thread e blocchi possiamo dire che thread rappresenti l'offset mentre il blocco * il numero di thread per blocco rappresenti la base.

Il thread tra loro è bene che lavorino in zone vicine, dato che possono esser fatti lavorare con una forte interconnessione a livello di condivisione e sincronizzazione, mentre il blocchi verranno processati in modo indipendente.

Il numero di blocchi può essere trovato dividendo il numero totale di thread per il numero di thread per blocchi (solitamente prefissata e strategica).

Esempio di add usando thread e blocchi:

```
__global__ void add(int *a, int *b, int *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}

#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int *a, *b, *c;                                // host copies of a, b, c
    int *d_a, *d_b, *d_c;                          // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);

    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Gestire array di dimensione arbitraria: se il thread non sono multipli esatti dei blocchi, si usa un nuovo blocco in cui eventualmente solo una parte dei thread vengono fatti girare, gli altri vengono lasciati a non fare nulla.

Esempio con n dimensione dell'array, se l'indice è maggiore di n, non succede nulla, semplicemente il test fallisce e il thread non fa niente. L'importante è avere abbastanza blocchi da poter coprire tutti i gli elementi dell'array. Se è multiplo, si allocano i blocchi esatti.

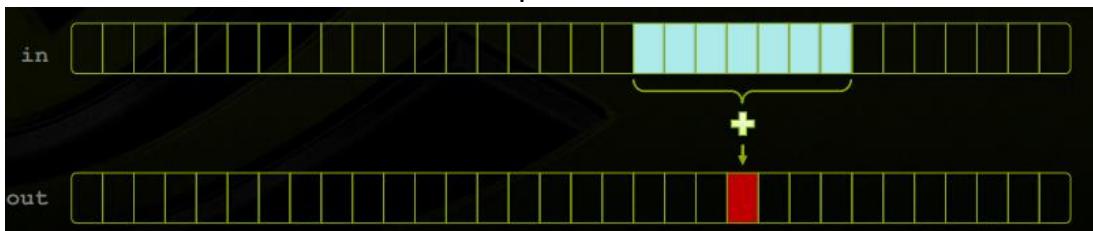
```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
}

add<<<(N + M-1) / M,M>>>(d_a, d_b, d_c, N);
```

I vantaggi di utilizzare blocchi e thread nascono direttamente dall'hw, dato che i thread hanno meccanismi per comunicare e sincronizzarsi efficientemente, mentre i blocchi non possono cooperare o interagire.

COOPERAZIONE TRA THREAD

Data una certa posizione di un array, questo produce in output la somma dei suoi n vicini (con $n = 3$) -> concetto di *stencil* che trasano. Ci sarà uno stencil per ogni thread e tutti gli stencil si sovrapporranno lungo l'array. Lo stencil (in azzurro) è la topologia dei dati su cui lavora il singolo thread a partire dalla "cella" di cui si occupa direttamente + i "vicini" che gli servono per lavorare.



Esempio con in azzurro e in rosso ciò che viene fatto dal singolo thread

Implementazione all'interno di un blocco:

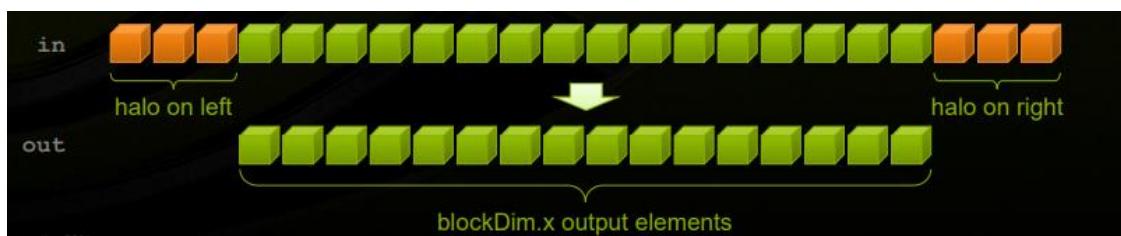
Ciascun thread processa un elemento di output.



Ogni cella dell'array in lettura sarà letto più volte (in questo caso 7, la grandezza dello stencil). Questo è uno spreco, possiamo usare quindi la memoria condivisa (componente hw per i thread), che ci permette, appunto, di condividere la memoria all'interno di un blocco, questo chip è all'interno della GPU ed è estremamente veloce. Parliamo di *shared memory* per la memoria condivisa dai thread all'interno di uno stesso blocco, solitamente 64KB, ci permette di risparmiare moltissime letture nella memoria globale (quella della RAM, dell'ordine dei GB). Essendo la memoria condivisa solo tra i thread all'interno dello stesso blocco, i vari blocchi non possono collaborare tra loro e tra le loro singole memorie condivise.

Implementazione con memoria condivisa:

Ogni thread legge un dato dalla memoria globale e lo va a memorizzare nella memoria condivisa. Quindi i 7 accessi allo stesso dato vengono fatti in memoria condivisa con un costo



molto più basso del primo accesso al dato in memoria globale e quindi con un notevole guadagno in efficienza.

Programmazione con memoria shared: la keyword shared andrà ad allocare un array che sarà mappato nella memoria shared della GPU. Dovremo quindi lavorare con due memorie, quella globale e quella condivisa. Dobbiamo quindi sapere utilizzare un'indicizzazione globale ed una per la memoria condivisa.

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}
```

Manipolazione delle due memorie da parte dei thread all'interno di un blocco:

1. Caricamento in parallelo dei dati dalla memoria globale nella memoria shared (prima riga dopo il commento)
2. If per la gestione dei thread all'inizio, quelli che hanno un indice minore del raggio (attività che verrà fatta in parallelo ma non da tutti i thread)

Notare che l'indicizzazione utilizzata da ciascun thread sono due, una per la memoria locale e una per la memoria globale.

Questo è il classico stile SIMT (Single Instruction Multiple Thread).

L'uso della memoria condivisa serve per tutte le volte in cui bisogna accedere più volte allo stesso dato.

DATA RACE: si ha se qualche thread rimane indietro, invalidando la correttezza del programma. I thread che devono fare più operazioni, potrebbero restare indietro.

```
...
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];
...
```

Per evitare la data race si usa la barriera di sincronizzazione `void __syncthreads();` prima delle operazioni sui dati (quando devono essere utilizzati e quindi corretti, tutti caricati) che sincronizza tutti i thread all'interno del blocco.

COORDINARE HOST E DEVICE

Quando noi andiamo a chiamare un kernel, l'attività è di tipo asincrono, il controllo ritorna subito al processore che può fare altre attività, mentre il kernel lavora. Tuttavia per usare i risultati del kernel, dobbiamo sincronizzarci, questo può essere fatto in diversi modi:

- `cudaMemcpy()`: bloccante, ci fa attendere che la copia sia completata, avviene solo quando tutte le chiamate precedenti sono state completate

- `cudaMemcpyAsync()`: è asincrona, non blocca la CPU

- `cudaDeviceSynchronize()`: blocca al CPU fino a quando tutti i kernel lanciati saranno completati

Gestione degli errori: si può chiedere alla GPU l'ultimo errore per poi convertirlo e stamparlo come stringa

Possiamo vedere quante GPU sono accessibili all'host, per scegliere quale usare tramite le chiamate `cudaGetDeviceCount(int *count)` che restituisce il numero delle GPU, `cudaSetDevice(int device)` che imposta quale device utilizzare, `cudaGetDevice(int *device)`, `cudaGetDeviceProperties(cudaDeviceProp *prop, int device)` che ritorna le proprietà di un device.

Più thread host possono condividere un device, ma vanno organizzati; ciascun thread può anche gestire molteplici device, anche se l'idea migliore è mantenere il rapporto 1-1.

COMPUTE CAPABILITY

Architetture ad evoluzione nel tempo, è necessario comunicarla al compilatore di modo che possa sfruttare al meglio anche l'hw della scheda. La **compute capability** di un device descrive la sua architettura: numero di registri, dimensioni della memoria, caratteristiche.

Compute Capability	Selected Features (see CUDA C Programming Guide for complete list)	Tesla models
1.0	Fundamental CUDA support	870
1.3	Double precision, improved memory accesses, atomics	10-series
2.0	Caches, fused multiply-add, 3D grids, surfaces, ECC, P2P, concurrent kernels/copies, function pointers, recursion	20-series

GUIDA ALLA PROGRAMMAZIONE

CAPITOLO 1: INTRODUZIONE

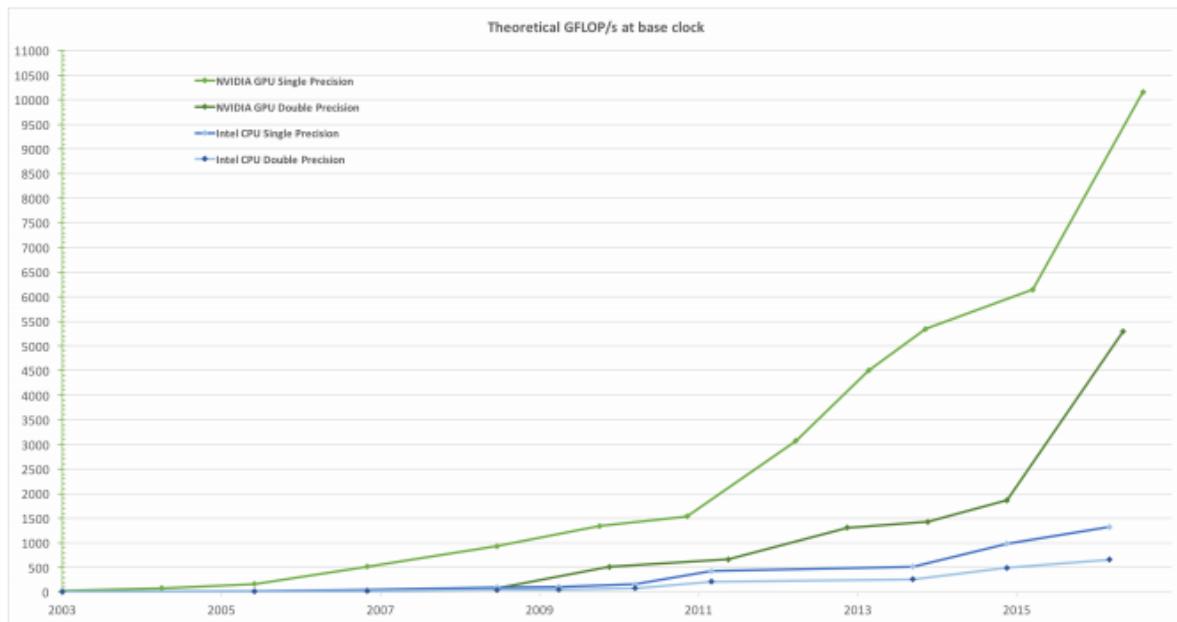


Figure 1 Floating-Point Operations per Second for the CPU and GPU

Le due linee verdi riguardano le GPU, quelle blu le CPU e misurano le operazioni in GFLOPS. Il progresso sulla potenza di calcolo di una CPU ha una crescita esponenziale, ma con base molto bassa, mentre la crescita delle GPU è estremamente più rapida, con un distacco molto forte. Da ciò si capisce che, per alcune operazioni, il contributo che può dare la GPU è molto alto.

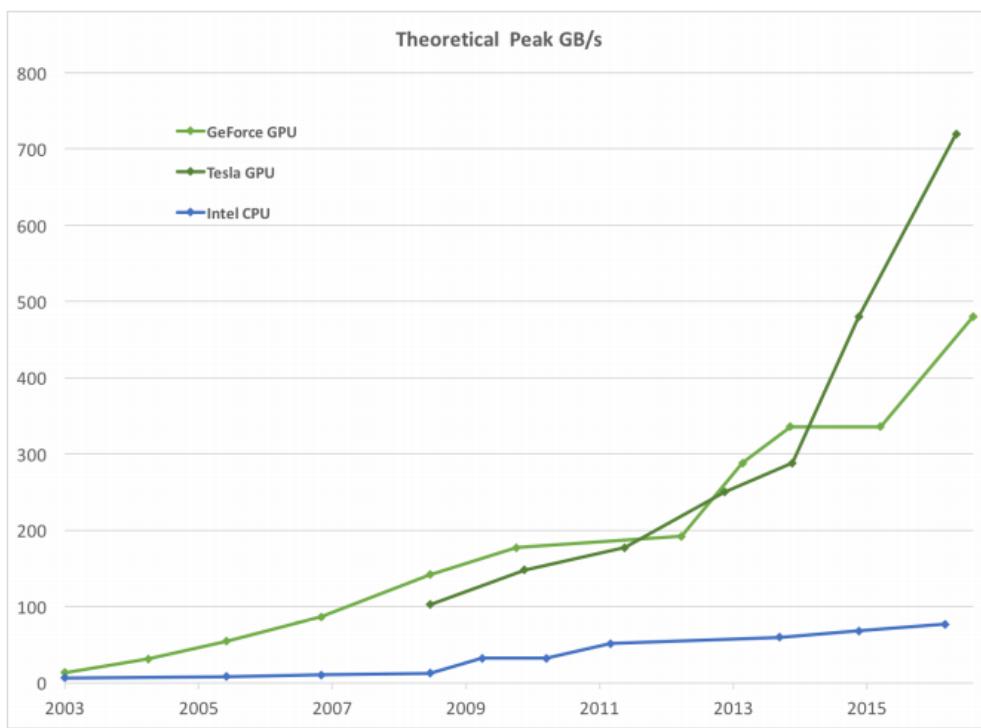


Figure 2 Memory Bandwidth for the CPU and GPU

La larghezza di banda nella memoria per i CPU è quasi assestata, per le GPU c'è un incremento molto forte.



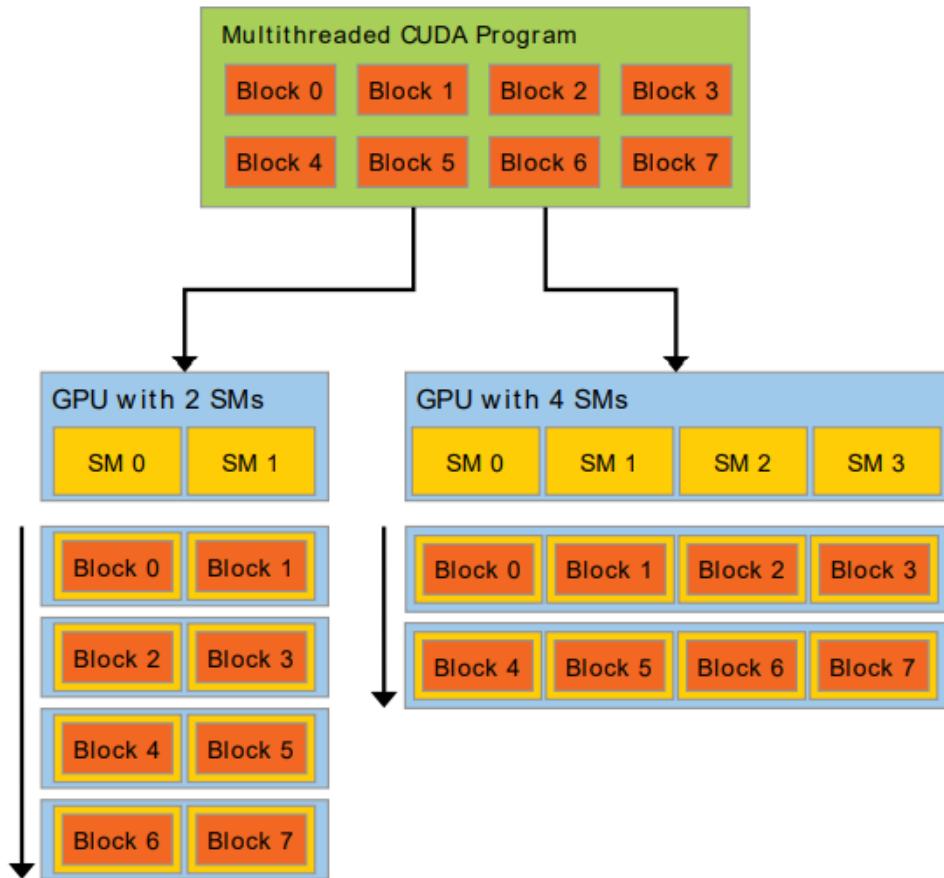
Figure 3 The GPU Devotes More Transistors to Data Processing

Questa immagine spiega perché la potenza di calcolo della GPU possa essere così ampia rispetto a quella della CPU, il motivo è l'impiego di più transistor per processare i dati. A sinistra una CPU quad core si dedica molto spazio nel chip per la parte di caching, per un recupero veloce dei dati, un'altra grossa parte per l'unità di controllo che controllerà le varie ALU, approccio di tipo MIMD. In una GPU i transistor sono per la maggior parte dedicati per realizzare delle ALU, ovvero per svolgere tanti conti in parallelo. Relativamente pochi transistor in proporzione sono dedicati alle unità di controllo e alla cache, ciò comporterà un set di istruzioni ridotte e quindi si ha un approccio di tipo SIMD. Ogni riga della GPU è un SM, nella GPU va affiancato uno scheduler di blocchi. La GPU privilegia operazioni uguali su dati diversi, con operazioni uguali per tutti i thread su dati diversi, la latenza sarà alta, ma il trasferimento sarà molto veloce. La GPU funziona molto bene se l'algoritmo si appoggia su computazione data parallel con un'alta intensità computazionale.

Avere lo stesso codice eseguito per diversi dati, richiede un'unità di controllo semplificata, questo ci permette di avere minor necessità di un controllo del flusso sofisticato. La latenza della memoria può essere nascosta mentre faccio i conti, il tempo di attesa per avere nuovi dati, sarà riempito dall'enormità di conti che dovrò fare nel frattempo. Sarà necessario creare una corrispondenza tra i vari elementi da elaborare e i vari thread, è una scelta libera che va però ad influenzare completamente le prestazioni del programma.

SCALABILITÀ

Le GPU permettono di scrivere lo stesso programma e farlo eseguire su architetture diverse, quindi il programma deve essere scalabile e indipendente dall'architettura specifica. Ogni core è in grado di eseguire un certo numero di thread in parallelo, tuttavia il kernel deve essere organizzato in modo da gestire i thread senza lasciare uno schema troppo libero, inoltre ogni core ha la sua memoria condivisa con la sua unità di controllo, per cui dovremo organizzarci di conseguenza. I nostri thread del blocco verranno lanciati su un core della scheda, i diversi blocchi del kernel non hanno nessun vincolo di relazione tra loro, sono indipendenti, quindi quello che fa solitamente la GPU è assegnare ogni blocco ad un core. Abbiamo la possibilità di prendere un blocco di thread e schedularlo, con un ordine non prevedibile, sui multiprocessori di una GPU in base al loro numero, col fine di utilizzarli tutti. Il singolo blocco eseguirà il thread in parallelo. Una volta definito il numero di blocchi, non devo più modificare il sorgente, ma questo scalerà in base al numero di SM (Stream Multiprocessor) della scheda, adattandosi. C'è un parallelo tra i blocchi e i core SIMT della GPU (gli SM).



A GPU is built around an array of Streaming Multiprocessors (SMs) (see [Hardware Implementation](#) for more details). A multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors.

Figure 5 Automatic Scalability

La scalabilità si ottiene con un certo numero di blocchi, con uno solo, questo ovviamente non avviene.

CAPITOLO 2: MODELLO DI PROGRAMMAZIONE

KERNEL: codice che andrà a controllare l'esecuzione dei blocchi sulla GPU.

ORGANIZZAZIONE DI THREAD E BLOCCHI

La GPU per adattarsi ai casi di dati 1D e 2d, fa sì che i thread supportino un'indicizzazione ha 3 dimensioni.

Esempio con indicizzazione bidimensionale:

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                      float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

In tutte le GPU c'è un limite al numero di thread per blocchi, solitamente 1024 thread all'interno di un blocco. I motivi sono hw: c'è un numero limitato di registri.

Anche i blocchi possono essere organizzati in una, due o tre dimensioni.

Esempio:

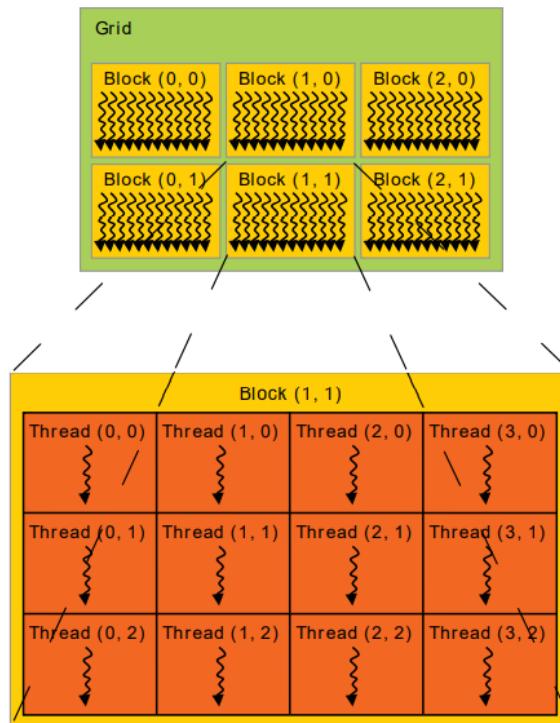


Figure 6 Grid of Thread Blocks

```

// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}

```

In questo caso ho 256 thread per blocco con blocchi a 2 dimensioni. 16x16 è una scelta comune. Dovrò organizzare i blocchi andando a partizionare n in base al numero di thread per blocco. I vari blocchi saranno eseguiti indipendentemente e non ci devono essere race-condition tra i blocchi, altrimenti dobbiamo sincronizzare o cambiare schema. All'interno di un singolo blocco c'è la collaborazione che può essere ottenuto tramite la memoria condivisa.

GERARCHIA DI MEMORIA

Ogni thread all'interno di un kernel può accedere a dati diversi in posizioni diverse. Ogni thread ha una sua memoria privata che solo lui può vedere (registri associati alle variabili del kernel) che gli altri thread non possono vedere; il blocco, che contiene un certo numero di thread, ha una memoria condivisa, a cui hanno accesso tutti i thread del blocco; tutti i thread possono accedere anche alla memoria globale, che è persistente, a differenza di quella del blocco, che sparisce quando il kernel ha terminato. Ogni blocco deve leggere e scrivere aree di memoria diverse, altrimenti si hanno risultati imprevedibili e bisogna garantire che non ci siano race condition, se ce ne sono, bisognerà sincronizzare (i thread) o sequenzializzare alcune parti. La memoria costante e la memoria texture sono memorie in sola lettura, che per le loro funzioni, hanno un sistema caching efficiente.

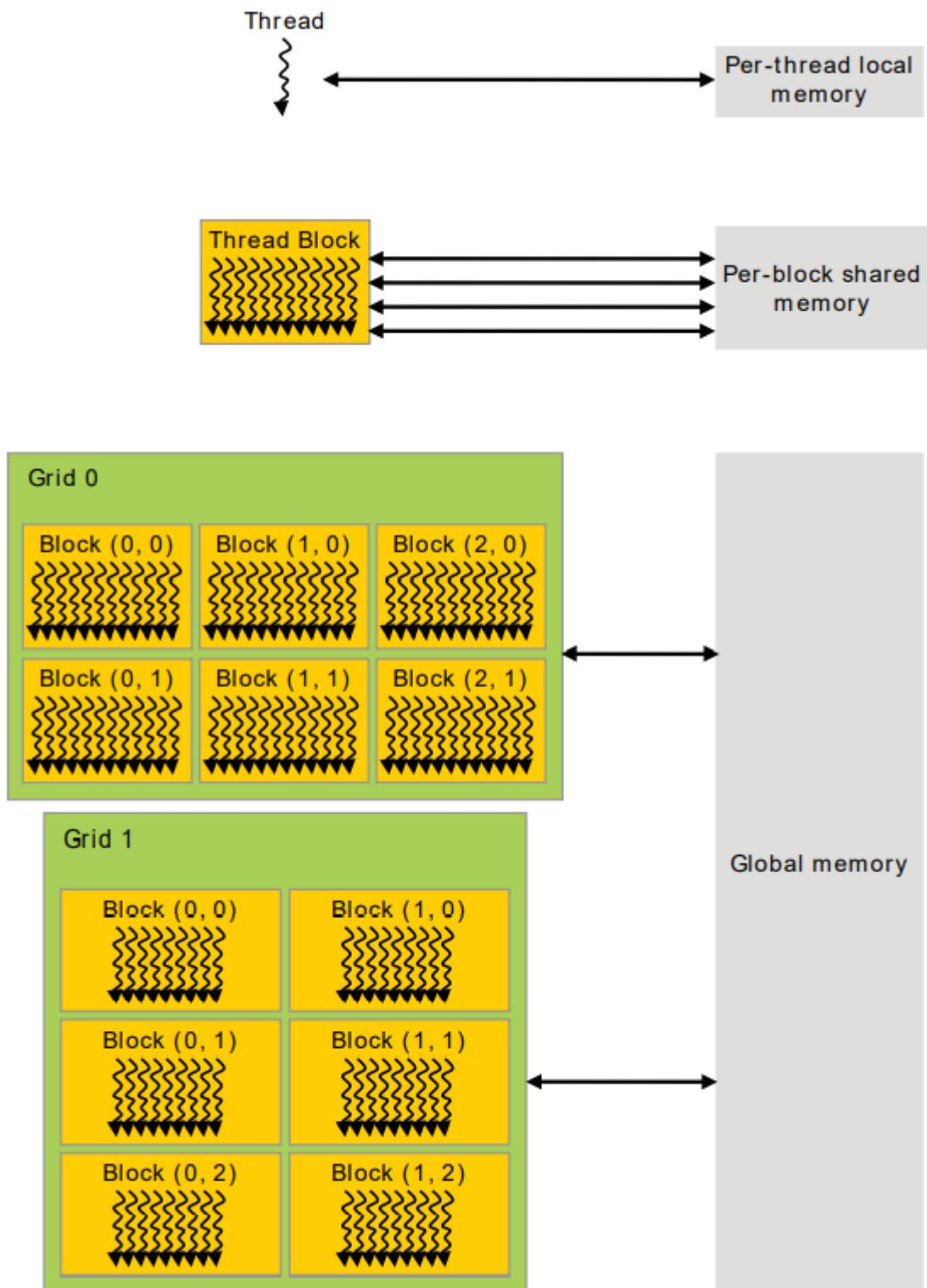


Figure 7 Memory Hierarchy

CAPITOLO 4: HW, WARP E BRANCHING

Un scheda GPU contiene uno o più processori (SM) quando lanciamo un kernel i blocchi verranno schedulati sugli SM disponibili. Tutti i thread di un blocco vanno in esecuzione sullo stesso SM, non su SM diversi.

L'architettura usata è quella SIMT (Single Instruction, Multiple-Thread). Il singolo SM che elabora un blocco può gestire e schedulare gruppi di 32 thread paralleli, chiamati warp (insieme di 32 thread paralleli, gestiti in modo trasparente dalla GPU, non da noi). I thread che compongono un warp cominciano insieme l'esecuzione e avranno lo stesso program counter (stanno eseguendo lo stesso codice parallelamente), tuttavia ogni singolo thread può lavorare con il suo specifico program counter e i suoi registri personali. È quindi possibile fare un branching, eseguendo indipendentemente qualcosa di diverso dagli altri thread del gruppo.

All'inizio si attivano 32 ALU in parallelo, con lo stesso pc per dare le stesse istruzioni a tutti i thread, tuttavia il branch permette di differenziare l'esecuzione di alcuni thread. Se due thread di un warp lanciano istruzioni diverse, il warp non potrà lavorare in parallelo. Il warp potrà quindi andare a identificare un sottoinsieme di thread, di modo da poter eseguire quei sottoinsiemi in modo parallelo. Quando il multiprocessore riceve l'ordine di eseguire diversi warp, i warp vengono schedulati all'interno del blocco. Ogni warp contiene i thread consecutivi rispetto agli id dei thread e viene sempre diviso nello stesso modo, quindi posso sempre sapere qual è il warp di un certo thread, con il primo warp contenente il thread 0. La GPU è in grado di lavorare con 32 thread in vero parallelismo per ciascun SM.

Un warp esegue contemporaneamente un'istruzione comune a tutti i thread del warp, se c'è una divergenza ovvero un'istruzione che dipende dal codice, si ha un branch, il warp dovrà eseguire ciascuna istruzione diversa in momenti diversi, raggruppando i thread che eseguono la stessa istruzione, disabilitando i thread che non fanno parte di quella istruzione, facendo ciò il tempo di esecuzione aumenta. Questa è la differenza tra MIMD e SIMT. La divergenza è divergenza di codice, del flusso di esecuzione del programma.

Posso quindi essere libero di far eseguire il codice diversi, ma aumentando il tempo di esecuzione. Il caso peggiore si ha quando si fanno molti if, avendo ogni thread del warp che fa una cosa diversa, richiedendo ciascuno un'esecuzione e quindi nessun beneficio parallelo. Dopo la divergenza sarà possibile riunire i thread, la collezione è forzata quando faccio una sincronizzazione, eliminando le differenze.

La divergenza di branch avviene solo all'interno di un warp, tutti i thread eseguono indipendentemente dagli altri warp. L'architettura SIMT è quindi simile all'architettura SIMD, la differenza è che con i thread, non c'è bisogno di specificare nulla, perché il branching viene eseguito internamente dalla GPU, che partizionerà in warp e schedulerà l'esecuzione, mentre con SIMD era il programmatore a dover gestire tutto. Le prestazioni quindi sono diverse.

I thread attivi sono quelli che stanno partecipando all'esecuzione, gli altri sono quelli inattivi. Se ho istruzioni non atomiche e un warp scrive la stessa locazione di memoria da parte di più thread, queste scritture vengono serializzate e la loro esecuzione dipende dall'architettura, si ha quindi race condition.

LABORATORIO GPU

Thread: elemento più piccolo di calcolo presente all'interno di una GPU, esegue un flusso di esecuzione e può essere lanciato in parallelo ad altri thread.

Blocchi: raggruppamento di thread, che all'interno dei blocchi possono andare in parallelismo vero. Permettono di creare un secondo livello di parallelismo per organizzare il calcolo in base alla possibilità della scheda.

Organizzazione:

I thread sono organizzati 1D, 2D o 3D nel blocco

I blocchi sono organizzati su una griglia 1D, 2D o 3D

Blocchi diversi possono essere eseguiti da processori diversi

Non c'è garanzia sull'ordine di esecuzione dei blocchi

Identificatori:

threadIdx: identifica il thread all'interno del blocco (.x, .y, .z)

blockIdx: identifica il blocco all'interno della griglia (.x, .y, .z)

blockDim: descrive quanti thread ci sono in un blocco (.x, .y, .z)

gridDim: descrive quanti blocchi ci sono nella griglia (.x, .y, .z)

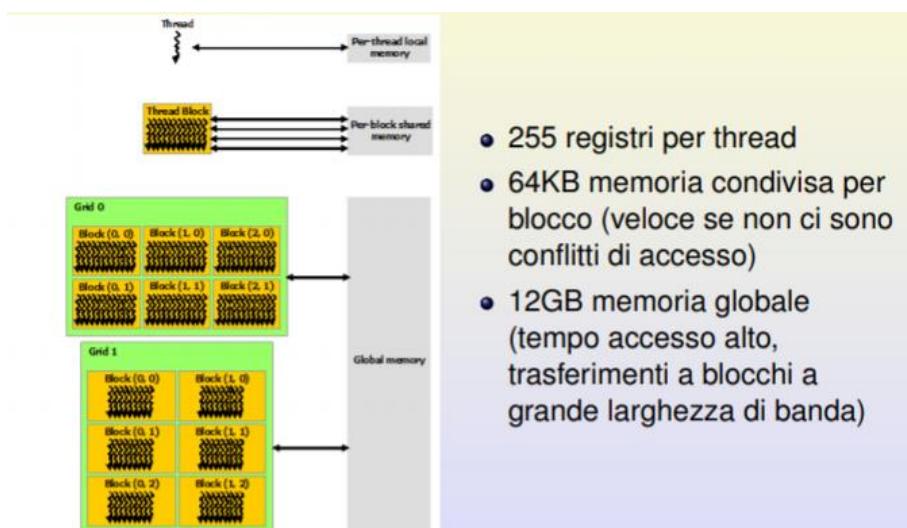
Memorie:

Le memorie presenti su una GPU sono di diverso tipo

Il loro utilizzo ha un impatto notevole sui trasferimenti dati

Necessaria una strutturazione dell'algoritmo parallelo per massimizzare prestazioni

Memoria P100 (usata oggi):

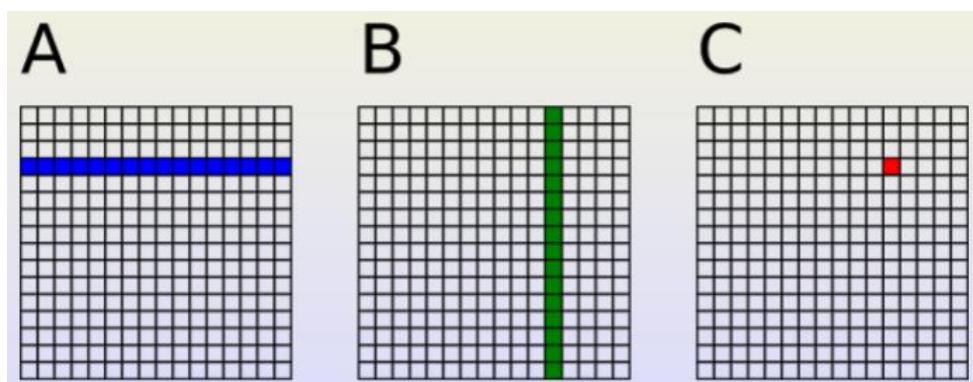


Esempio di calcolo:

1. Il programma gira su CPU (host)
2. Il programma chiede alla GPU (device) di eseguire un kernel
3. La CPU può fare altro durante il lavoro della GPU
4. È possibile lanciare più kernel in parallelo (se non ci sono dipendenze)
5. La CPU colleziona risultati dalla GPU e prosegue

ESEMPIO: PRODOTTO MATRICI

Prodotto $C = A \times B$ (A dimensione $M \times K$, B dimensione $K \times N$). Ogni cella di C richiede il prodotto scalare di due vettori. Ogni cella di A o B viene letta K volte, c'è una ridondanza negli accessi alla memoria.



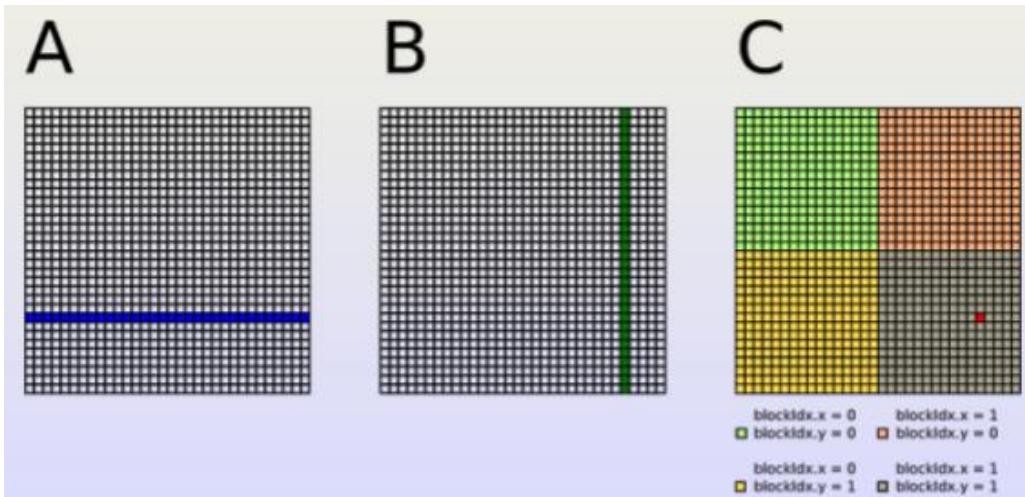
Sulla CPU:

3 for annidati: c'è spazio per la parallelizzazione. Bisogna organizzarsi per la lettura dei dati in memoria, dato che ogni cella viene letta K volte.

```
void main() {
    for i = 0 to M do
        for j = 0 to N do
            /* compute element C(i,j) */
            for k = 0 to K do
                C(i,j) = C(i,j) + A(i,k) * B(k,j)
            end
        end
    end
}
```

Parallelizzazione su GPU: idea naïve (parallelizzando le cose più banali)

Ogni thread calcola una cella di $C(i, j)$. Ogni thread carica una riga di A e una colonna di B dalla memoria globale. Il thread calcola il prodotto e scrive il risultato nella cella in memoria globale. Bisogna dividere il lavoro in blocchi da 1024 thread. Viene comodo lavorare in 2D. Ogni blocco può lavorare in modo indipendente dagli altri. Siamo andati a parallelizzare i 2 cicli for più esterni, facendo eseguire a ciascun thread il for più interno.



```
void main(){
    define A_cpu, B_cpu, C_cpu in the CPU memory
    define A_gpu, B_gpu, C_gpu in the GPU memory
    memcpy A_cpu to A_gpu
    memcpy B_cpu to B_gpu
    dim3 dimBlock(32, 32)
    dim3 dimGrid(N/dimBlock.x, M/dimBlock.y)
    matrixMul<<<dimGrid, dimBlock>>>(A_gpu, B_gpu, C_gpu, K)
    memcpy C_gpu to C_cpu
}
```

Allociamo la memoria nella CPU e nella GPU. Definiamo la dimensione del blocco (dimBlock, massimo 32 x 32) e decidiamo quanti blocchi ci saranno (dimGrid). Invochiamo un thread specificando

blocchi e thread a cui passiamo i tre puntatori nella memoria GPU e il numero K su cui verrà fatto il for da parte di ciascun thread. Quando il kernel avrà finito, dato che la memcpy è bloccante, copiamo la matrice calcolata dalla GPU alla CPU.

```
__global__ void matrixMul(A_gpu, B_gpu, C_gpu, K) {
    temp = 0
    // Row i of matrix C
    i = blockIdx.y * blockDim.y + threadIdx.y
    // Column j of matrix C
    j = blockIdx.x * blockDim.x + threadIdx.x
    for k = 0 to K-1 do
        accu = accu + A_gpu(i, k) * B_gpu(k, j)
    C_gpu(i, j) = accu
}
```

Critiche:

Troppe letture dalla memoria globale

Se i dati sono contigui e letti da thread contigui, si avranno buone prestazioni.

Tuttavia la matrice B non viene letta con celle contigue (inefficiente).

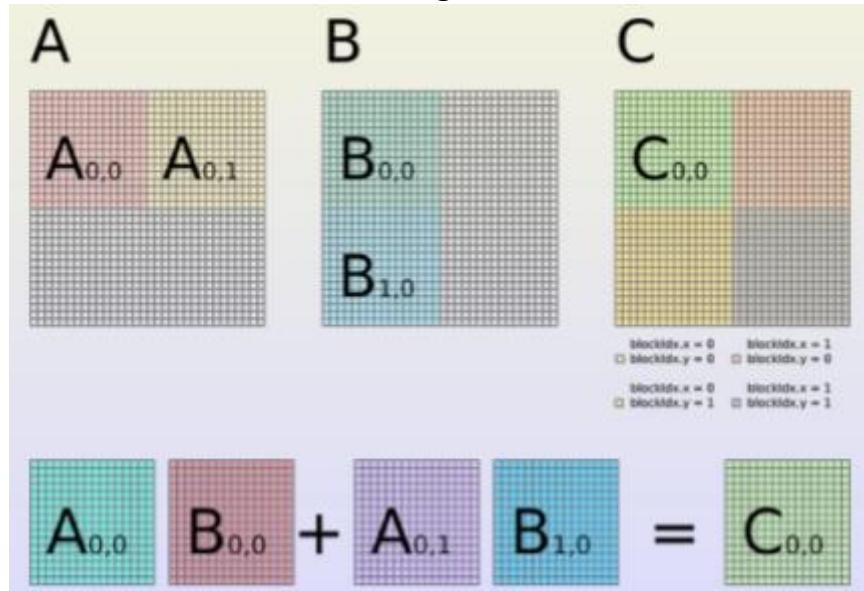
Proposta: memorizzo B trasposta (memory coalescing), ancora troppe letture dalla memoria globale.

Soluzione per la parallelizzazione su GPU con memoria condivisa:

Dovrò sfruttare la memoria condivisa del blocco, evitando di leggere tante volte gli stessi dati e organizzando il calcolo parallelo in modo diverso

Divido le matrici in blocchi (tiles) e calcolo ogni blocco di C in più fasi, invece di fare un unico for per tutta una riga e tutta la colonna, prima mi occupo di un certo blocco, dividendo il lavoro. Faccio il prodotto delle sottomatrici e poi le combino per ottenere la soluzione.

Dobbiamo minimizzare le letture in memoria globale.

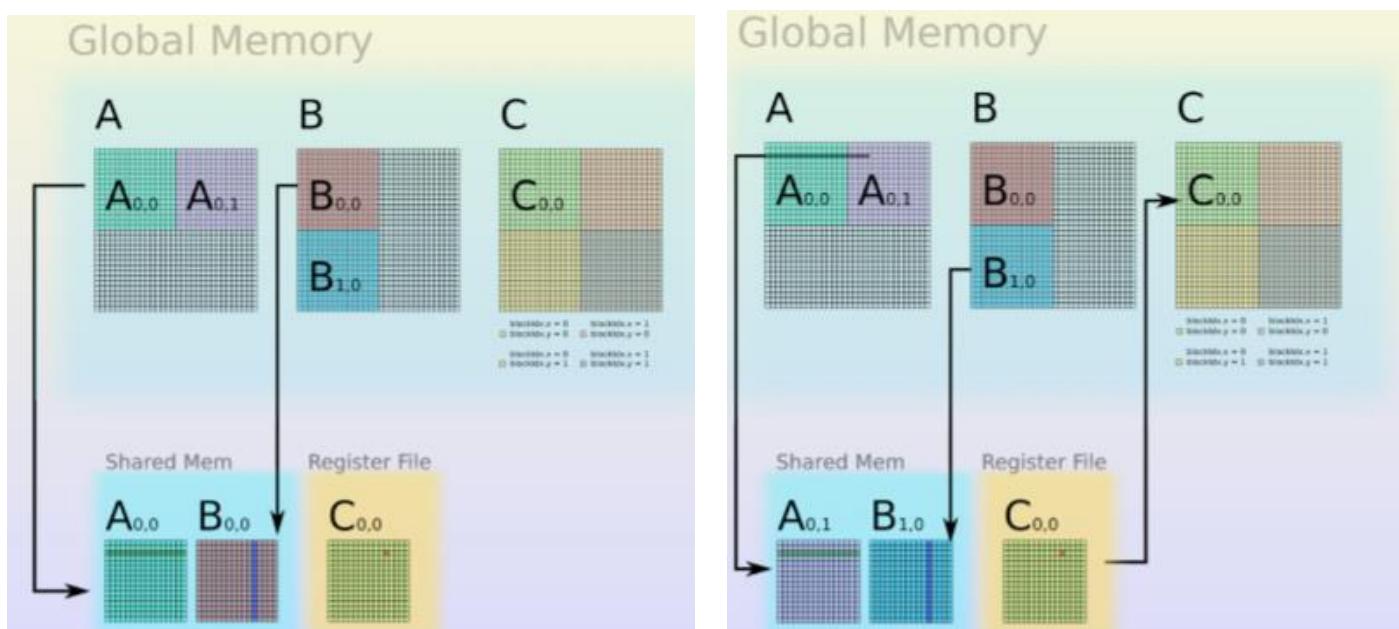


Ad ogni iterazione, ogni thread copia un blocco di A e uno di B dalla memoria globale in shared. Ogni thread calcola il prodotto e aggiorna il risultato in un registro. Al termine delle iterazioni, tutti i thread memorizzano il loro risultato (blocco di C) in memoria globale.

Ci sarà solo una copia dalla memoria globale, quella di ciascun blocco e della sua memoria shared.

Prima coppia di blocchi:

Ultima coppia di blocchi:



Esempio con pseudocodice della soluzione proposta sopra:

```
__global__ void matrixMul(A_gpu,B_gpu,C_gpu,K) {
    __shared__ float A_tile(blockDim.y, blockDim.x)
    __shared__ float B_tile(blockDim.x, blockDim.y)
    accu = 0
    for tileIdx = 0 to (K/blockDim.x - 1) do

        //carica blocchi di A e B in shared mem
        i = blockIdx.y * blockDim.y + threadIdx.y
        j = tileIdx * blockDim.x + threadIdx.x
        A_tile(threadIdx.y, threadIdx.x) = A_gpu(i,j)
        B_tile(threadIdx.x, threadIdx.y) = B_gpu(j,i)
        __sync()

    for tileIdx = 0 to (K/blockDim.x - 1) do
        //carica blocchi di A e B in shared mem
    // Prodotto scalare (accumulato)
    for k = 0 to blockDim.x do
        accu = accu + A_tile(thrIdx.y,k) * B_tile(k,thrIdx.x)
    end
    __sync()
end

    //scrive il blocco in global
    i = blockIdx.y * blockDim.y + threadIdx.y
    j = blockIdx.x * blockDim.x + threadIdx.x
    C_gpu(i,j) = accu
}
```

Le operazioni di calcolo (somme, moltiplicazioni) sono invariate. Gli accessi alla memoria globale sono stati divisi per la dimensione del blocco, se prima facevo k letture, ora farò k/32. Infatti un blocco di una matrice è letto una volta dalla globale e usato dalla shared più volte per calcolare il blocco C.

Setup cluster e lab, da slide 26:

OTTIMIZZAZIONE DELLA MEMORIA

Dobbiamo conoscere le varie gerarchie e tipologie di memoria presenti per sfruttare al meglio la potenza di calcolo e il trasferimento dei dati. Bisogna minimizzare il trasferimento di dati tra host e device. La GPU è brava a dare i dati a sé stessa e ai suoi processori, con un grosso collo di bottiglia per il trasferimento in RAM principale, rappresentato dai trasferimenti host-device. Bisogna trasferire poca informazione se in modo frequente, o una grande informazione ma alla fine del calcolo. L'idea è quindi minimizzare il trasferimento dei dati tra host e device durante il calcolo.

Pinned memory: si indica una certa quantità di memoria RAM che non viene paginata, in modo da avere un accesso diretto, per avere degli scambi a prestazioni alte nei trasferimenti tra host e device. Va usata con cautela.

Sovrapposizione tra trasferimento e calcolo di un kernel: mettiamo in parallelo diverse attività della CPU che viaggiano su stream diverse che siano asincrone e indipendenti tra loro, bisogna aver definito la memoria pinned (per essere sicuri che le pagine siano bloccate in RAM e non cambiate). Usando la copia asincrona attiviamo il trasferimento dei dati, ma abbiamo il controllo sull'host, il kernel dovrà risolvere tutte le dipendenze sui dati presenti.

La GPU implementa dei motori di copia per supportare diversi trasferimenti dei dati asincroni ed in parallelo, possiamo ottenere delle informazioni con deviceQuery.

Stream: diverse linee di esecuzione.

Sovrapposizione di computazione e trasferimento, esempio: Lancio 3 attività parallele, una di copia, una di calcolo e una di calcolo sull'host. Il kernel dovrà aspettare la fine del trasferimento, per evitare race condition sui dati. Se il kernel lavorasse indipendentemente, la copia e il kernel potrebbero lavorare in pieno parallelismo.

La copia è asincrona, ma dato che il kernel è sullo stesso stream 0, allora esso si sincronizza

```
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, 0);
kernel<<<grid, block>>>(a_d);
cpuFunction();
```



con la copia, anche se il controllo passa al kernel .

Si sincronizzano perché sono sullo stesso stream e i dati su cui lavora il kernel sono dipendenti.

Copia parallela ed esecuzione, esempio: Definisco due stream nuovi e diversi, sono in parallelismo vero perché il kernel non lavora su dati coinvolti e dipendenti dalla copia.

```
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, stream1);
kernel<<<grid, block, 0, stream2>>>(otherData_d);
```

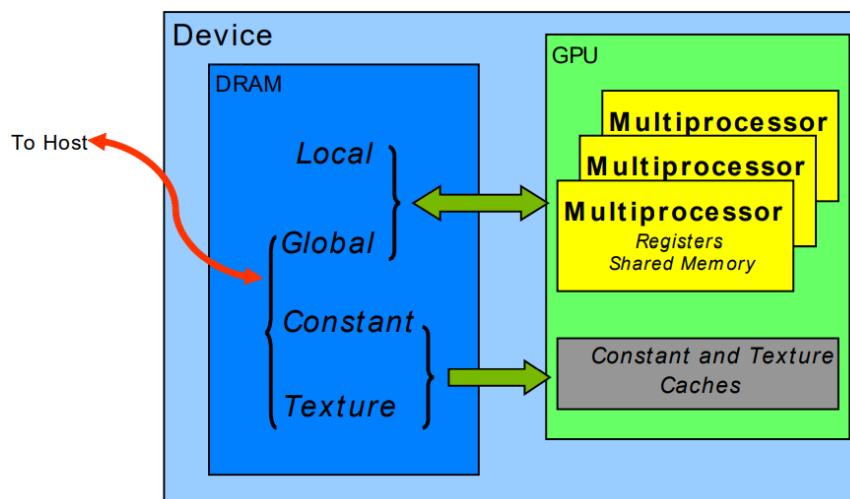


Indirizzamento virtuale unificato

Possiamo usare un sistema di indirizzamento virtuale che utilizza gli stessi riferimenti sia all'interno dell'host che del device, tramite un meccanismo che ci nasconde la locazione esplicita della memoria. Senza questo meccanismo, un'applicazione deve tener traccia dei riferimenti sia della memoria host che device, gestione esplicita.

Se usiamo il sistema virtuale, consideriamo una sola allocazione e otteniamo il puntatore da usare sulle memorie RAM (host e GPU) con `cudaPointerGetAttributes()`, nascondendo la locazione esplicita.

SPAZI DI MEMORIA GPU



Tipologie:

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes††	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation
† Cached in L1 and L2 by default on devices of compute capability 2.x; cached only in L2 by default on devices of higher compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags.					
†† Cached in L1 and L2 by default on devices of compute capability 2.x and 3.x; devices of compute capability 5.x cache locals only in L2.					

Quando scriviamo il codice del kernel, il compilatore lo compila e cerca di minimizzare il numero di registri necessari, se questi sono maggiori di quelli a disposizione, il compilatore

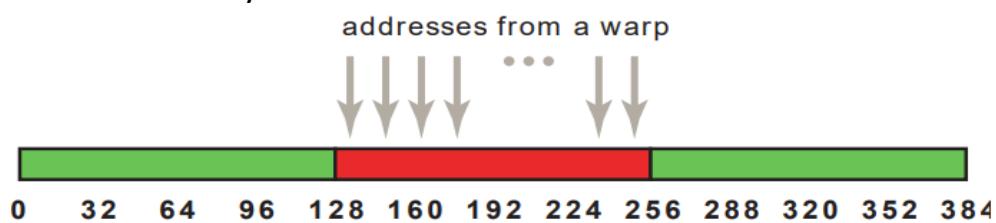
dovrà occuparsi di fare rotazione e copia dei registri nella memoria locale della GPU (*spilling*), pagando in termini di latenza che aumenta e prestazioni che diminuiscono. È buona norma andare a vedere quanti registri sono stati utilizzati e quanto spilling viene fatto.

ACCESSI ALLA MEMORIA GLOBALE

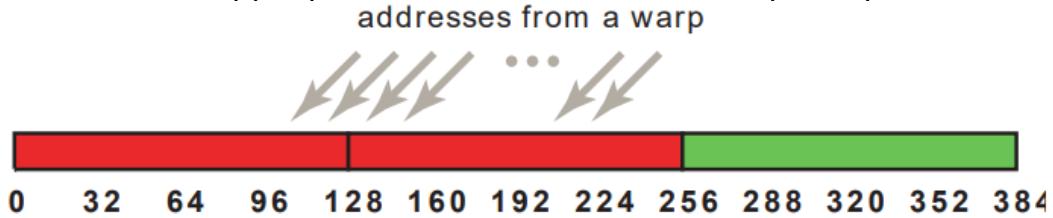
Data l'alta latenza, dobbiamo analizzare il pattern di accessi ai dati. Parliamo di **accessi coalesced**, ovvero raggruppati e ben organizzati. Le letture e scritture in memoria da parte di un thread di una singolo warp hanno un certo pattern, se questo è bene organizzato la GPU può fornire i dati ai thread dei warp in una sola transazione, anziché una transazione per ciascun thread.

Ovviamente l'organizzazione degli accessi dipende anche dalla compute capability e quindi dall'hw. L'idea è che più i thread chiedono celle in memoria che sono consecutive, più gli accessi possono essere raggruppati.

Esempio con pattern semplice: il thread k legge la k-esima parola di 4 byte (un float). Questo pattern utilizza 128 Byte bene allineati.



Esempio con accesso non bene allineato, dove l'accesso inizia a metà di un blocco di 128. Questo accesso costerà doppio perché dovrà fare due accessi per coprire i blocchi a metà.



Da ciò capiamo che l'organizzazione dei dati influisce moltissimo sulle prestazioni, sia nel pattern di accesso, che nella disposizione.

Ad esempio un array che memorizza valori a coppi è il 50% meno efficiente perché metà degli elementi nella transazione non sono usati e rappresentano spreco di banda.

Conviene allocare strutture di array piuttosto che array di strutture.

MEMORIA CONDIVISA

Solitamente viene suddivisa in banchi di memoria che possono essere letti e scritti in maniera parallela. Se ho n indirizzi consecutivi che andranno su n banchi consecutivi, posso

fare tutte le attività contemporaneamente. Se invece le richieste vanno sullo stesso banco, esse sono messe in sequenza e quindi si ha un calo delle prestazioni. Un’eccezione si ha quando tutti i thread del warp vanno a leggere la stessa locazione, in tal caso l’attività è molto efficiente.

L’organizzazione di memoria dipende dalla compute capability e quindi dall’hw

Compute Capability 2.x

On devices of compute capability 2.x, each bank has a bandwidth of 32 bits every two clock cycles, and successive 32-bit words are assigned to successive banks. The warp size is 32 threads and the number of banks is also 32, so bank conflicts can occur between any threads in the warp. See *Compute Capability 2.x* in the *CUDA C Programming Guide* for further details.

Compute Capability 3.x

On devices of compute capability 3.x, each bank has a bandwidth of 64 bits every clock cycle (*). There are two different banking modes: either successive 32-bit words (in 32-bit mode) or successive 64-bit words (64-bit mode) are assigned to successive banks. The warp size is 32 threads and the number of banks is also 32, so bank conflicts can occur

TEXTURE MEMORY E EQUAZIONE DEL CALORE (Capitolo 7 di CUDA_by_Example)

Vedremo come usare le texture per migliorare le prestazioni in certi scenari precisi. La memoria texture nasce per il rendering grafico. Possiamo utilizzare la texture per fare degli accessi sparsi in cache se l'informazione è di sola lettura.

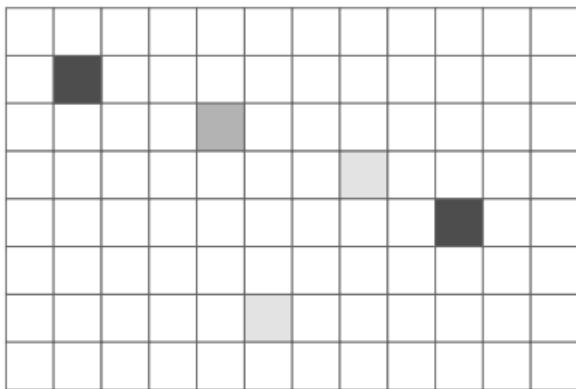


Figure 7.2 A room with "heaters" of various temperature

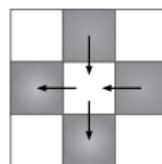


Figure 7.3 Heat dissipating from warm cells into cold cells

Punti con una temperatura prefissa e costante, vicini di questi punti che verranno modificati in base all'equazione della dissipazione del calore.

Equation 7.1

$$T_{NEW} = T_{OLD} + \sum_{NEIGHBORS} k \cdot (T_{NEIGHBOR} - T_{OLD})$$

Ad ogni iterazione della simulazione, aggiorniamo la matrice che rappresenta le varie temperature in due modi:

1. Forzando la temperatura costante di certe celle
2. Calcolare le temperature aggiornate

Possiamo usare un doppi buffer, il primo per le temperature costanti (lettura) e il secondo per quelle calcolate (scrittura)

Codice con organizzazione bidimensionale, ricordiamo che lo stencil è 2d:

```
__global__ void copy_const_kernel( float *iptr,
                                  const float *cptr ) {
    // map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    if (cptr[offset] != 0) iptr[offset] = cptr[offset];
}
```

Nella matrice abbiamo una matrice di puntatori costanti cptr che se hanno una temperatura diversa da 0, andrà aggiornata.

Kernel che aggiorna i valori calcolando i flussi:

```
__global__ void blend_kernel( float *outSrc,
                             const float *inSrc ) {
    // map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    int left = offset - 1;
    int right = offset + 1;
    if (x == 0)    left++;
    if (x == DIM-1) right--;

    int top = offset - DIM;
    int bottom = offset + DIM;
    if (y == 0)    top += DIM;
    if (y == DIM-1) bottom -= DIM;

    outSrc[offset] = inSrc[offset] + SPEED * ( inSrc[top] +
                                                inSrc[bottom] + inSrc[left] + inSrc[right] -
                                                inSrc[offset]*4 );
}
```

È importante anche conoscere i 4 vicini in questo caso, stando attenti a non fuoriuscire dalla matrice. Per ogni riga ci si chiede se si è i primi o gli ultimi.

Nell'ultima parte vengono lette le temperature, si calcola la propagazione e si va a scrivere il risultato nella matrice d'uscita. In questo caso il kernel modifica anche i valori che dovevano rimanere costanti, perciò usiamo un altro buffer e kernel che andrà a riscrivere i valori costanti. L'offset è l'indice nell'array concreto che memorizza la matrice bidimensionale ottenuta concatenando le righe.

```
#include "cuda.h"
#include "../common/book.h"
#include "../common/cpu_anim.h"

#define DIM 1024
#define PI 3.1415926535897932f
#define MAX_TEMP 1.0f
#define MIN_TEMP 0.0001f
#define SPEED 0.25f

// globals needed by the update routine
struct DataBlock {
    unsigned char *output_bitmap;
    float *dev_inSrc;
    float *dev_outSrc;
    float *dev_constSrc;
    CPUAnimBitmap *bitmap;
```

```

        cudaEvent_t      start, stop;
        float           totalTime;
        float           frames;
    };

    void anim_gpu( DataBlock *d, int ticks ) {
        HANDLE_ERROR( cudaEventRecord( d->start, 0 ) );
        dim3      blocks(DIM/16,DIM/16);
        dim3      threads(16,16);
        CPUAnimBitmap *bitmap = d->bitmap;

        for (int i=0; i<90; i++) {
            copy_const_kernel<<<blocks,threads>>>( d->dev_inSrc,
                                            d->dev_constSrc );
            blend_kernel<<<blocks,threads>>>( d->dev_outSrc,
                                            d->dev_inSrc );
            swap( d->dev_inSrc, d->dev_outSrc );
        }
        float_to_color<<<blocks,threads>>>( d->output_bitmap,
                                                d->dev_inSrc );

        HANDLE_ERROR( cudaMemcpy( bitmap->get_ptr(),
                                d->output_bitmap,
                                bitmap->image_size(),
                                cudaMemcpyDeviceToHost ) );

        HANDLE_ERROR( cudaEventRecord( d->stop, 0 ) );
        HANDLE_ERROR( cudaEventSynchronize( d->stop ) );
        float   elapsedTime;
        HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                            d->start, d->stop ) );
        d->totalTime += elapsedTime;
        ++d->frames;
        printf( "Average Time per frame: %3.1f ms\n",
                elapsedTime / (float)d->frames );
    }

    void anim_exit( DataBlock *d ) {
        cudaFree( d->dev_inSrc );
        cudaFree( d->dev_outSrc );
        cudaFree( d->dev_constSrc );

        HANDLE_ERROR( cudaEventDestroy( d->start ) );
        HANDLE_ERROR( cudaEventDestroy( d->stop ) );
    }
}

```

Il ciclo con 90 iterazioni utilizza i due kernel per fissare i punti costanti e calcolare i nuovi
59

valori, lo swap fa sì che quella che era la matrice di output diventi la matrice input per l'iterazione successiva.

L'idea è usare delle texture di float

La differenza rispetto a fare una malloc è che dobbiamo fare un collegamento (binding) tra la texture e l'area di memoria che vogliamo associare a quella texture.

```
// these exist on the GPU side
texture<float> texConstSrc;
texture<float> texIn;
texture<float> texOut;
```

Useremo un certo buffer come texture poi faremo il binding

```
HANDLE_ERROR( cudaMalloc( (void**)&data.dev_inSrc,
                           imageSize ) );
HANDLE_ERROR( cudaMalloc( (void**)&data.dev_outSrc,
                           imageSize ) );
HANDLE_ERROR( cudaMalloc( (void**)&data.dev_constSrc,
                           imageSize ) );

HANDLE_ERROR( cudaBindTexture( NULL, texConstSrc,
                               data.dev_constSrc,
                               imageSize ) );
HANDLE_ERROR( cudaBindTexture( NULL, texIn,
                               data.dev_inSrc,
                               imageSize ) );
HANDLE_ERROR( cudaBindTexture( NULL, texOut,
                               data.dev_outSrc,
                               imageSize ) );
```

Da qui in poi la GPU può andare ad accedere ai dati utilizzando delle fetch legate alle texture con `tex1Dfetch()`

Calcoliamo le posizioni nell'array ma invece di fare l'accesso nell'array, utilizzo i riferimenti delle texture e vado ad ottenere la texture tramite la cache, potenzialmente evitando l'accesso in memoria globale e quindi risparmiando tempo.

```

__global__ void blend_kernel( float *dst,
                             bool dstOut ) {
    // map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    int left = offset - 1;
    int right = offset + 1;
    if (x == 0)    left++;
    if (x == DIM-1) right--;

    int top = offset - DIM;
    int bottom = offset + DIM;
    if (y == 0)    top += DIM;
    if (y == DIM-1) bottom -= DIM;

    float t, l, c, r, b;
    if (dstOut) {
        t = tex1Dfetch(texIn,top);
        l = tex1Dfetch(texIn,left);
        c = tex1Dfetch(texIn,offset);
        r = tex1Dfetch(texIn,right);
        b = tex1Dfetch(texIn,bottom);
    }
    else {
        t = tex1Dfetch(texOut,top);
        l = tex1Dfetch(texOut,left);
        c = tex1Dfetch(texOut,offset);
        r = tex1Dfetch(texOut,right);
        b = tex1Dfetch(texOut,bottom);
    }
    dst[offset] = c + SPEED * (t + b + r + l - 4 * c);
}

```

```

__global__ void copy_const_kernel( float *iptr ) {
    // map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    float c = tex1Dfetch(texConstSrc,offset);
    if (c != 0)
        iptr[offset] = c;
}

```

Al termine del lavoro dovrò sganciare le texture con una Unbind

LABORATORIO CALORE

2 Matrici (new e old) che si aggiornano dinamicamente (new diventa old dopo aver calcolato) e producono in uscita nuovi valori.

OCCUPANCY

The CUDA Occupancy Calculator allows you to compute the multiprocessor occupancy of a GPU by a given CUDA kernel. The multiprocessor occupancy is the ratio of active warps to the maximum number of warps supported on a multiprocessor of the GPU. Each multiprocessor on the device has a set of N registers available for use by CUDA program threads. These registers are a shared resource that are allocated among the thread blocks executing on a multiprocessor.

The CUDA compiler attempts to minimize register usage to maximize the number of thread blocks that can be active in the machine simultaneously. If a program tries to launch a kernel for which the registers used per thread times the thread block size is greater than N, the launch will fail.

Grazie al [foglio Excel](#) creato da NVIDIA possiamo andare a calcolare l'occupancy della scheda, andando a modificare qualche parametro.

Per vedere ulteriori info sulla compilazione sul cluster nvcc - -ptxas-options=-v nomefile.cu

Nei punti 1 e 2 vanno inseriti i dati dell'utente, le "risposte" seguono dal punto 3 in poi e nei grafici a lato. Alla riga 51 è indicata l'occupancy.

Il primo grafico mostra quanti thread per blocco posso definire e come cambia il numero di warp che posso mandare al processore.

Quello sotto mostra la variazione dell'occupancy al variare dell'utilizzo dei registri.

L'ultimo grafico a destra mostra l'impatto al variare dell'utilizzo della memoria condivisa per ciascun blocco.

PROGRAMMAZIONE GPU SU PIU' SCHEDE ATTRAVERSO L'USO DI MPI

L'idea è quella di utilizzare più schede per dividere il lavoro e poi usare MPI per comunicare. Le memorie globali delle diverse GPU non si possono condividere tra le schede, tuttavia, se le schede condividono lo stesso bus, è possibile scambiare i dati ad alte prestazioni senza far intervenire l'host. Il codice dovrà gestire i dati tra le GPU in modo esplicito.

Opzione 1: copie esplicite attraverso l'host che gestisce tutte le attività.

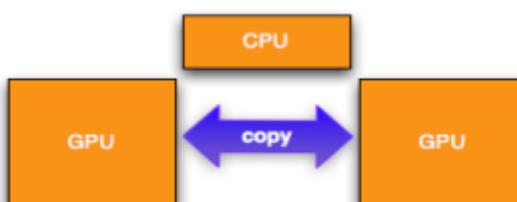
Quando abbiamo più GPU a disposizione, possiamo chiedere al sistema quale codice è stato assegnato a ciascuna GPU. Se l'host può vedere più schede, possiamo scegliere quale utilizzare con CudaSetDevice, il device 0 è quello di default e le GPU sono in ordine decrescente di performance.

Un approccio semplice in cui possiamo gestire più schede e kernel dallo stesso host:

```
// Run independent kernel on each CUDA device
int numDevs = 0;
cudaGetNumDevices(&numDevs); ...
for (int d = 0; d < numDevs; d++) {
    cudaSetDevice(d);
    kernel<<<blocks, threads>>>(args);
}
```

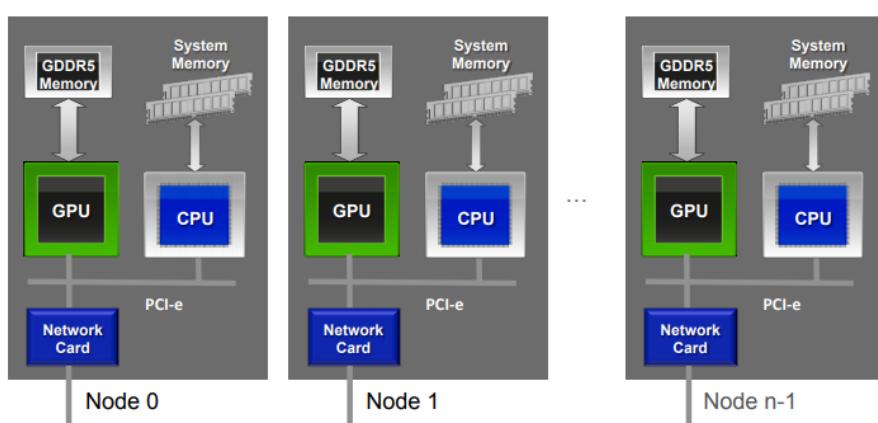
Vantaggi:

- Non dobbiamo preoccuparci di gestire più thread sull'host, perché le computazioni sono parallele sulle GPU
- Possiamo lanciare sia i kernel che le copie in parallelo sulla stessa GPU
- Possiamo scambiare dati tra le GPU usando le memorie PCIe e il supporto P2P, senza passare dall'host

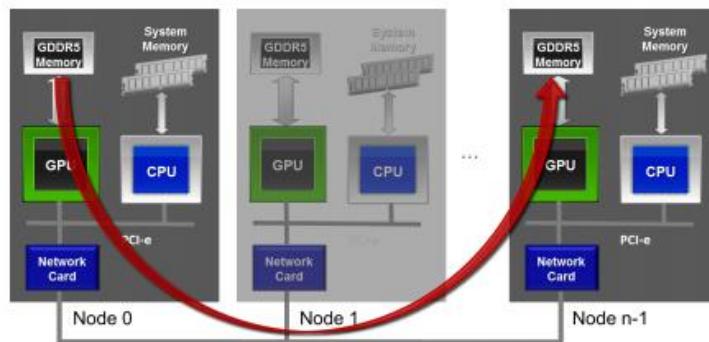


MPI + CUDA

MPI+CUDA



Trasferimento da GPU a GPU con MPI



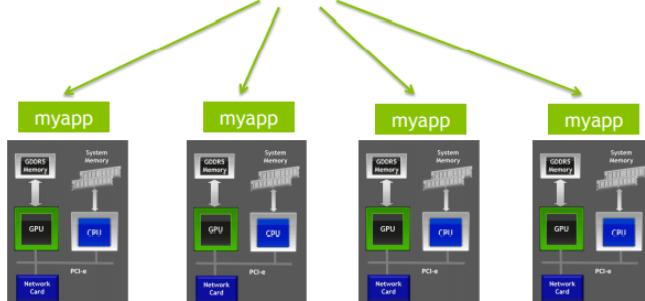
```
//MPI rank 0  
MPI_Send(s_buf_d,size,MPI_CHAR,n-1,tag,MPI_COMM_WORLD);  
  
//MPI rank n-1  
MPI_Recv(r_buf_d,size,MPI_CHAR,0,tag,MPI_COMM_WORLD,&stat);
```

MPI lavora sulla RAM del processore, quindi bisogna preparare i trasferimenti i modo più dettagliato sull'host.

COMPILAZIONE e LANCIO

Dovremo compilare con MPI, includendo le librerie per CUDA per poi lanciare con MPI

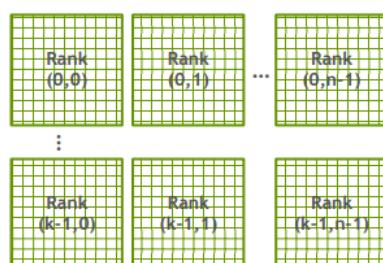
```
$ mpicc -o myapp myapp.c  
$ mpirun -np 4 ./myapp <args>
```



EQUAZIONE DEL CALORE

Se, per l'equazione del calore, dividiamo il calcolo tra più GPU, dobbiamo preoccuparci della sincronizzazione e della posizione dei dati (che, nel caso dei vicini, potrebbero essere memorizzati in un'altra GPU). Halo: bordo, dati "vicini" a quelli che uso, presenti in altre GPU, che devo recuperare tramite trasferimento, ma "facendo finta" che siano dati locali.

Esempio di decomposizione 2d con dominio $n \times k$



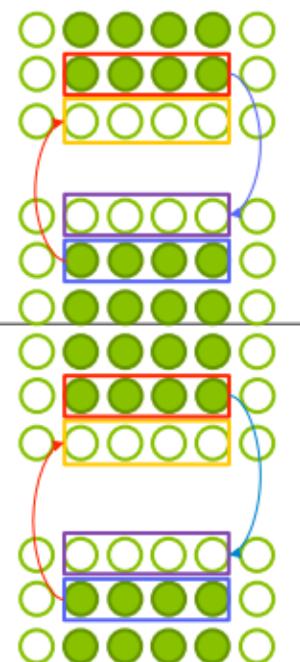
Parallelismo a tre livelli: thread, blocco e le GPU, di cui quest'ultimo gestito tramite MPI che si occuperà delle copie.

Esempio codice: Jacobi per l'aggiornamento delle righe di cima e fondo

Se lo facessimo su CPU (senza GPU):

Plain C on CPU

```
MPI_Sendrecv(Tnew+offset first row, m-2, MPI_DOUBLE, t_nb, 0,  
             Tnew+offset bottom boundary, m-2, MPI_DOUBLE, b_nb, 0,  
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



```
MPI_Sendrecv(Tnew+offset last row, m-2, MPI_DOUBLE, b_nb, 1,  
             Tnew+offset top boundary, m-2, MPI_DOUBLE, t_nb, 1,  
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Versione di MPI che non dialoga con CUDA -> copia esplicita

CUDA

```
//send to bottom and receive from top - top bottom omitted  
  
cudaMemcpy(Tnew+1, Tnew_d+1, (m-2)*sizeof(double), cudaMemcpyDeviceToHost);  
MPI_Sendrecv(Tnew+offset first row, m-2, MPI_DOUBLE, t_nb, 0,  
             Tnew+offset bottom boundary, m-2, MPI_DOUBLE, b_nb, 0,  
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);  
cudaMemcpy(Tnew_d, Tnew, (m-2)*sizeof(double), cudaMemcpyDeviceToHost);
```

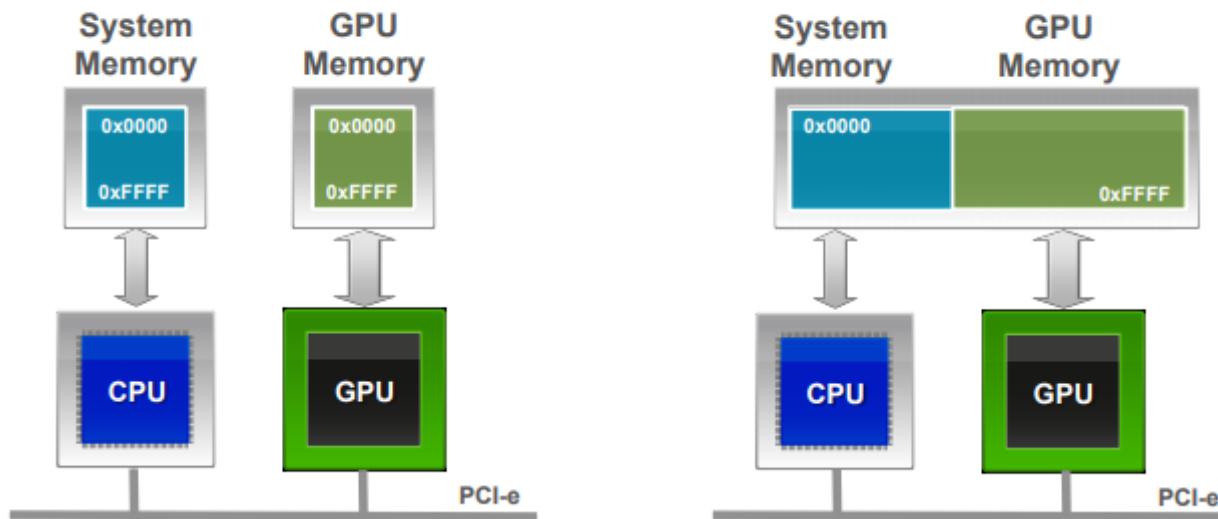
Versione di MPI che dialoga con CUDA -> MPI gestisce tutto, no copia esplicita

CUDA

Device pointer

```
MPI_Sendrecv(Tnew d+offset first row, m-2, MPI_DOUBLE, t_nb, 0,  
             Tnew d+offset bottom boundary, m-2, MPI_DOUBLE, b_nb, 0,  
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);  
MPI_Sendrecv(Tnew_d+offset last row, m-2, MPI_DOUBLE, b_nb, 1,  
             Tnew_d+offset top boundary, m-2, MPI_DOUBLE, t_nb, 1,  
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Depending on the CUDA (and MPI) version, the MPI_Sendrecv may work or fail. We may need to copy data from the GPU to the CPU to exchange them with other MPI tasks.

*No UVA : Separate Address Spaces**UVA : Single Address Space*

Un unico indirizzo per l'intera memoria GPU e CPU causa una semplificazione delle interfacce come MPI e cudaMemcpy. Supportato su device con una compute capability di 2.0 o superiore.

With UVA and CUDA-aware MPI No UVA and regular MPI

<pre>//MPI rank 0 MPI_Send(s_buf_d, size, ...);</pre> <pre>//MPI rank n-1 MPI_Recv(r_buf_d, size, ...);</pre>	<pre>//MPI rank 0 cudaMemcpy(s_buf_h, s_buf_d, size, ...); MPI_Send(s_buf_h, size, ...);</pre> <pre>//MPI rank n-1 MPI_Recv(r_buf_h, size, ...); cudaMemcpy(r_buf_d, r_buf_h, size, ...);</pre>
--	--

STREAM CUDA (capitolo 10 di Cuda by Example)

Gli stream vanno creati tutti esplicitamente, tranne quello di default.

```
// initialize the stream
cudaStream_t      stream;
HANDLE_ERROR( cudaStreamCreate( &stream ) );
```

Nel kernel, dopo aver creato lo stream, come 4 argomento possiamo passare il nome dello stream

```
kernel<<<N/256, 256, 0, stream>>>( dev_a, dev_b, dev_c );
```

Le attività nello stream sono seriali nello stream stesso, ma, se non ci sono dipendenze, paralleli rispetto ad altri stream. Se ci sono dipendenze tra dato in due stream diversi, possiamo comunque trovare momenti in cui copiare la memoria e trasferire dati.

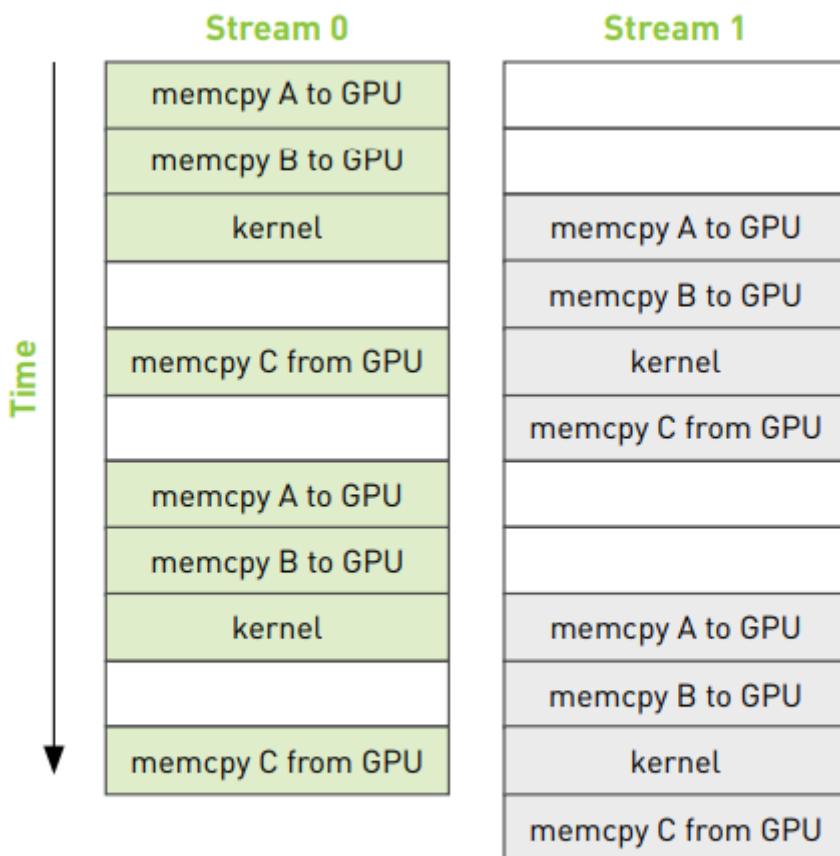


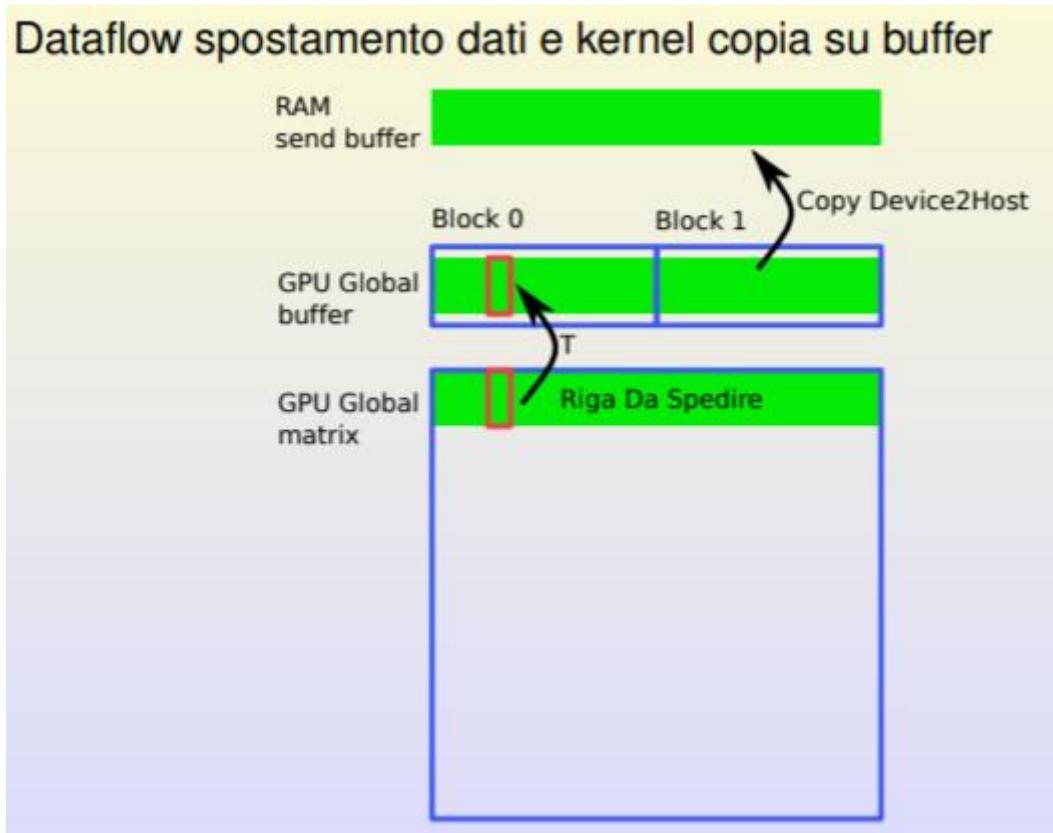
Figure 10.1 Timeline of intended application execution using two independent streams

LABORATORIO GPU + MPI

Heat transfer con una matrice $NX \times 2NY$ (due quadrati uno sopra l'altro). Progettato per 2 GPU con comunicazione MPI per scambio dati di bordo.

Trasferimento GPU -> RAM (rank 0) -> MPI -> RAM (rank 1) -> GPU

Output: ogni rank scrive un file



//AGGIUNGERE ALTRI BUFFER DI LETTURA E SCRITTURA QUANDO CI SONO 3 MATRICI UNA SOPRA L'ALTRA

ESEMPIO REALE DI INGEGNERIA CIVILE

SPH (Smoothed-Particle Hydrodynamics): Metodo numerico per integrare delle equazioni differenziali (che descrivono fenomeni fisici).

Approcci per integrare equazioni differenziali che descrivono il movimento dell'acqua in uno spazio tridimensionale, descrivono le equazioni di continuità (la massa si trasferisce, ma si conserva) e della conservazione della quantità di moto (legge di Newton in forma differenziale).

Le equazioni differenziali si integrano con metodi numerici, quelli matematici non vanno bene. L'ambito di applicazione di questi tipi di problemi è molto ampio (fluidi, previsioni metereologiche) e di alto interesse ingegneristico.

I metodi numerici utilizzati sono basati su una griglia e sono

FDM: Differenze Finite

FEM: Elementi Finiti

FVM: Volumi Finiti

I problemi di questi metodi sono: il costo, l'utilizzo di molto tempo e molte risorse che servono per la progettazione e l'analisi della griglia su cui vengono fatte le simulazioni.

L'alternativa al metodo numerico è andare in galleria del vento (approccio fisico, si può fare solo con l'aria).

I metodi a griglia hanno problemi a gestire interfacce con fluidi vari (onde + aria che si infrangono sulla spiaggia) o oggetti che si deformano e ruotano (ad esempio Turbina Pelton + acqua).

Riassumendo:

I metodi a griglia sono buoni per domini confinati e chiusi con confini che non si muovono.

I metodi a griglia non vanno bene per fare mesh generation e quando si hanno confini che si muovono.

Per superare i problemi dei metodi a griglia nascono i metodi meshless, ovvero metodi dove non c'è una griglia. I punti in cui si fanno i calcoli. Il metodo è Lagrangiano, i punti seguono i movimenti del fluido.

Lo stencil è a campana in questo caso e la cella è un punto che si sposta nello spazio.

SPH (senza programmazione parallelo) è molto lento rispetto ai metodi a griglia: 1.5 secondi della simulazione fisica con 300k particelle richiede 15 ore di esecuzione, questo è dovuto al fatto che ogni particella si interfaccia con 250 vicini. Il problema dei tempi di calcolo si adatta utilizzando metodi di calcolo HPC.

Tecniche di parallelizzazione

- OpenMP (limitato dal numero di core sul CPU corrente)
- MPI (miglior opzione per combinare le risorse di più machine connesse attraverso la rete, ma cluster HPC sono molto costosi)
- GPGPU