Università degli Studi di Cagliari

Facoltà di Ingegneria e Architettura

Corso di Laurea in Ingegneria Elettrica ed Elettronica

Calcolo Parallelo su MATLAB integrato con le GPU

Relatore:
Prof.
Giuseppe Rodriguez

Tesi di Laurea di:

Congiu Claudia

A.A. 2018-2019

SOMMARIO

1.Introduzione al Calcolo parallelo

Il Calcolo Parallelo è l'evoluzione del calcolo seriale, con questo termine ci si riferisce all'uso simultaneo di più unità di elaborazione per la risoluzione di un problema, che verrà suddiviso in "sotto-problemi", o parti discrete, ammesso che possa essere scomposto, che verranno risolti *concorrentemente*. Tali parti sono a loro volta suddivise in una serie di istruzioni. L'algoritmo parallelizzato dovrebbe essere in grado di essere risolto in meno tempo rispetto a quello seriale. Le risorse computazionali per l'applicazione del calcolo parallelo sono:

- un singolo computer con processori multipli,
- una rete di computer.

Queste vengono definite architetture parallele, le quali si differenziano da quelle sequenziali, basate sull'architettura di Von Neumann in cui tutte le istruzioni vengono eseguite dalla CPU singola.

Il parallelismo nelle architetture può avvenire a differenti livelli, come si evince dall'immagine sottostante.

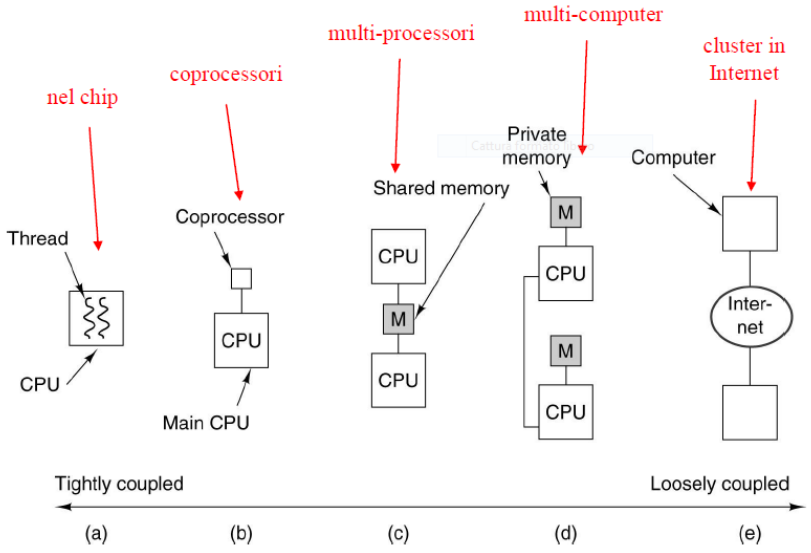


Figura 1.1 Diversi tipi di parallelismo

Al livello più basso, ossia quello del chip, le prestazioni possono essere incrementate tramite la parallelizzazione delle istruzioni, con il multithreading o addirittura l'integrazione di più CPU sullo stesso chip; a livello superiore è possibile aggiungere al sistema delle "CPU plug-in" che svolgono funzioni specializzate, questa è l'idea alla base dei cluster di computer.

Un settore nel quale risultano molto utili i coprocessori è quello dell'elaborazione grafica, per i quali le CPU non sono abbastanza performanti; per questo i computer sono dotati di GPU. La discussione dell'utilizzo di schede grafiche per il calcolo parallelo è rimandata al capitolo 3.

1.1 Classificazione delle architetture parallele: Tassonomia di Flynn

Esistono varie architetture di calcolatori paralleli, la classificazione usata ancora oggi è quella sviluppata da Flynn, ingegnere statunitense, attualmente professore emerito all'università di Stanford, nel '77. Questa si basa sui concetti di flussi di istruzioni (sequenza di istruzioni eseguita da un processore) e flussi di dati (insieme di operandi), che possono essere singoli o multipli e che sono per lo più indipendenti, quindi si hanno quattro combinazioni:

- SISD (Single Instruction Single Data): è la classica macchina sequenziale di Von Neumann, nella quale un singolo processore elabora sequenzialmente i dati con un solo flusso di istruzioni. L'esecuzione è deterministica.
- SIMD (Single Instruction Multiple Data): Un'unità di controllo singolo esegue su un singolo flusso di istruzioni, ma disponendo di varie ALU elabora simultaneamente su diversi flussi di dati. L'esecuzione in questo caso è sincrona e deterministica.
- MISD (Multiple Instruction Single Data): è una categoria anomala nella quale vengono eseguite molteplici istruzioni sullo stesso flusso di dati. Questa esecuzione è asincrona. Spesso le macchine a pipeline vengono classificate come MISD
- MIMD (Multiple Instruction Multiple Data): più processori indipendenti, facenti parte dello stesso sistema, operano sul proprio flusso di istruzioni e di dati. L'esecuzione è asincrona. La maggior parte dei calcolatori paralleli fanno parte di questa categoria.

Alla categoria MIMD appartengono i Multiprocessori, macchine a memoria condivisa, che si differenziano in base al modo di condivisione, e i Multicomputer, macchine a scambio di messaggi.

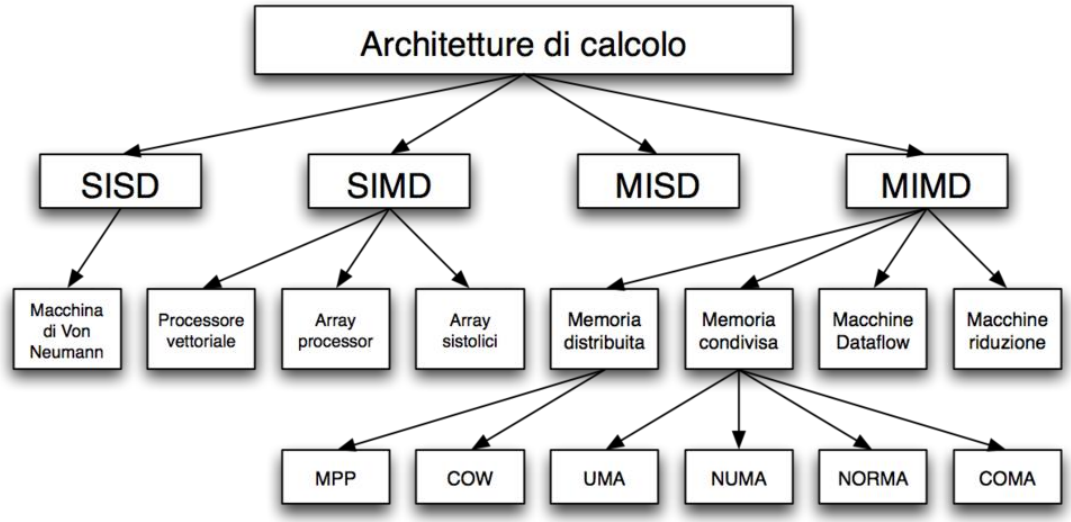


Figura 1.2 Tassonomia di Flynn ampliata

1.2 Architetture MIMD a confronto

In un sistema di calcolo parallelo, le CPU, operando sullo stesso algoritmo, devono potersi scambiare informazioni, per questo sono stati implementati i multiprocessori e i multicomputer, che si distinguono, come scritto nel paragrafo precedente, per la presenza o assenza di memoria condivisa.

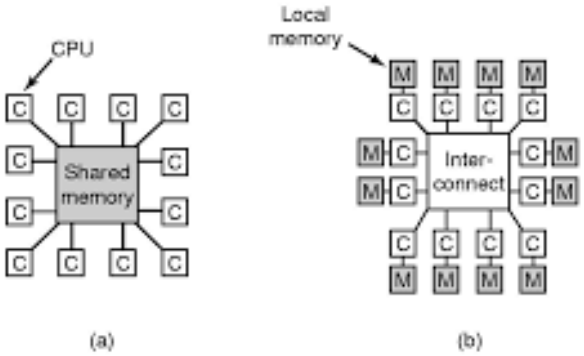


Figura1.3 a)Architetture a memoria condivisa b) Architetture a memoria distribuita

1.2.1 Multiprocessori

Un multiprocessore a memoria condivisa offre un unico spazio di indirizzamento fisico (è il caso dei chip multicore). I processori possono eseguire programmi indipendenti nei propri spazi virtuali, anche condividendo lo stesso spazio di indirizzamento fisico.

La comunicazione tra i processori avviene attraverso variabili condivise contenute in memoria, a cui tutti possono accedere tramite le semplici istruzioni LOAD e STORE.

In alcuni sistemi multiprocessore è presente una CPU dotata di funzioni speciali per l'I/O, ossia l'unica ad averne l'accesso, mentre nei sistemi SMP (Symmetric MultiProcessor) tutte le CPU hanno uguale accesso ai vari moduli.

Esiste un ulteriore suddivisione in categorie dei multiprocessori, in base al modo in cui è implementata la memoria condivisa, essendo questa divisa in più moduli:

- UMA (Uniform Memory Access): Il tempo di accesso di ogni CPU ad ogni modulo di memoria è equivalente indipendentemente dalla parola richiesta e dal processore richiedente.
- NUMA (NonUniform Memory Access): alcuni accessi sono più rapidi di altri in base al processore richiedente e al dato richiesto, perché spesso ogni CPU ha vicino un modulo di memoria.
- COMA (Cache Only Memory Access): molto simile alle NUMA, ma l'accesso si basa solo sulla memoria cache.

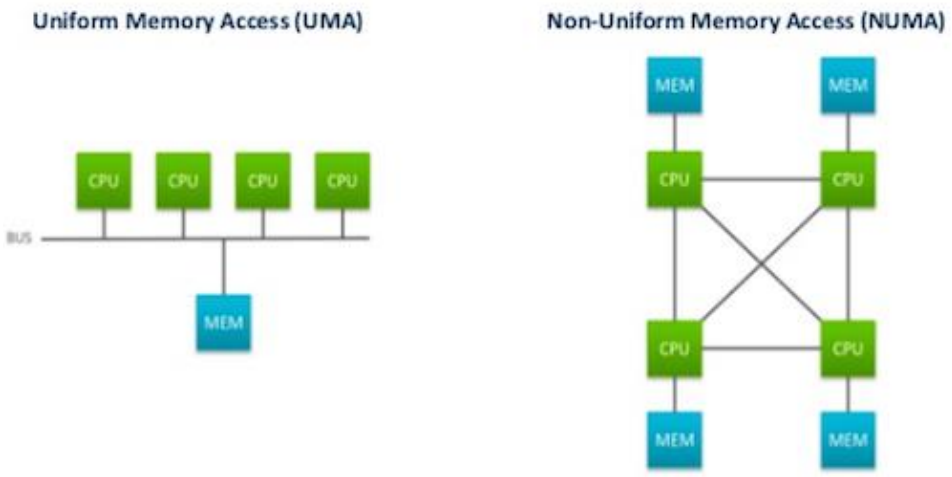


Figura 1.4 Architetture UMA e NUMA

I multiprocessori NUMA sono più problematici da programmare rispetto agli UMA, ma le architetture NUMA possono contenere un maggior numero di processori.
Data la condivisione dei dati nei processori paralleli, questi devono essere sincronizzati nel loro utilizzo, in maniera tale che uno non inizi a lavorar su un dato mentre un altro non ha finito di modificarlo. L'implementazione della sincronizzazione è basata sul lock delle variabili condivise, ossia una cella di memoria contente una variabile può essere bloccata da un solo processore per volta, mentre glia altri aspettano la fine dell'operazione.

1.2.2 Multicomputer

I multicomputer non hanno una memoria principale condivisa, ma ogni CPU prevede una memoria privata e locale, con cui nessun'altra CPU può interagire, e con cui comunica tramite i comandi LOAD e STORE. L'assenza di una memoria condivisa via hardware implica che non tutti i processi siano in grado di leggere e scrivere i dati tramite le semplici istruzioni load e store, quindi la soluzione adottata come meccanismo di comunicazione è uno scambio di messaggi (message passing) lungo la rete. Il sistema è dotato di procedure di invio e ricezione che garantiscono la coordinazione nella gestione dei dati.
Esistono due categorie di multicomputer:

- MPP (Massive Parallel Processor): costosi supercomputer costituiti da molte CPU in una rete interconnessa ad alta velocità.
- NOW (Network of Workstation) e COW (Cluster of Workstations): PC o workstation collegate da una rete d'interconnessione commerciale.

I multicomputer sono più facili da costruire dei multiprocessori, ma più difficili da programmare, perciò si sono costruiti sistemi ibridi, che abbiamo una condivisione della memoria a più livelli, con lo scopo di riuscire a ideare sistemi *scalabili*, ossia che abbiamo un comportamento funzionale al crescere del numero di CPU.

Un'altra possibilità è l'architettura DSM (Distribuited Shared Memory), nella quale si ha l'hardware di un multicomputer ma ci si serve del sistema operativo per simulare la memoria condivisa, che contiene le pagine degli indirizzi virtuali. Ogni macchina ha una propria memoria virtuale e una propria tabella delle pagine, quando la CPU effettua un'operazione all'interno di una pagina di cui non dispone, si ha una trap verso il sistema operativo, che localizza la pagina e chiede alla CPU proprietaria di spedirla lungo la rete. Dal punto di vista dell'utente, questa gestione sembra quella di un sistema a memoria virtuale.

1.3 Modelli per la Programmazione Parallela

I modelli per la programmazione parallela riflettono, generalmente, l'architettura del dispositivo, nonostante siano un'astrazione della stessa e possano essere implementati su ogni tipo di hardware [1]. Non esiste un modello migliore da utilizzare, ma la scelta dipende dalle necessità dell'utente e dal problema che si deve affrontare.
Esistono vari modelli:

- Memoria Condivisa, con o senza Thread: Il modello senza Thread è quello più semplice, utilizzato, generalmente, nei sistemi a memoria condivisa, appunto. In questo modello le task (o processi) condividono uno spazio di memoria comune da cui leggono e scrivono asincronamente. Il vantaggio di questo modello è che non c'è bisogno di specificare esplicitamente la comunicazione dei dati tra task, ogni processo ha uguale accesso alla memoria, perché non esiste un "proprietario" dei dati. Mentre è difficile gestire la località dei dati al processo, limitando quindi la performance della prestazione.Nel modello Multi-Threaded ogni processo può creare più unità di esecuzione (thread) che possono essere eseguire concorrentemente. Ognuno di questi ha accesso alla sua memoria privata ma condivide tutte le risorse associate al processo su cui opera, e comunica con gli altri thread tramite la memoria globale.Perché non si abbiamo conflitti tra i thread che operano sulla stessa locazione di memoria globale bisogna sincronizzare il processo
- Memoria Distribuita/Passaggio di Messaggi: L'insieme di processi può risiedere sulla stessa o più macchine interconnesse, ed usa la propria memoria locale. I processi si comunicano i dati attraverso uno scambio di messaggi, il che richiede cooperatività tra le operazioni, ossia un "send" deve sempre essere seguito da un receive. L'implementazione avviene attraverso un set di librerie che devono essere comprese nel codice.
- Data Parallel: Una serie di processi lavora collettivamente sullo stesso set di dati, in uno spazio di locazione globale. Ogni task lavora su una partizione differente del set. Viene utilizzata sia nelle architetture a memoria distribuita che in quelle a memoria condivisa, nella prima i processi possono accedere ai dati attraverso la

memoria globale, nella seconda la memoria globale può essere suddivisa, fisicamente o virtualmente, attraverso i processi.

- Ibridi: è una combinazione di più modelli,
- SPMD (Single Program Multiple Data) e MPMD (Multiple Program Multiple Data): sono modelli d'esecuzione, nel SPMD ogni macchina esegue lo stesso programma ma opera su dati diversi, eseguendo o meno le stesse istruzioni. Nel MPMD è prevista l'esecuzione di più programmi, che può essere eseguito da più di un processo e operare su diversi dati.

1.4 Granularità e livello di parallelismo

Il termine granularità si riferisce alla misura qualitativa del rapporto tra calcolo e computazione [1]. Il termine è quindi spesso riferito al numero e alla complessità dei processori in un sistema parallelo [28]. Un sistema a *grana fine* (fine-grain) ha un elevato numero di computazioni semplici, ognuna delle quali occupa poca memoria. In questo caso i processori devono comunicare frequentemente, dato che hanno a disposizione poca memoria. In un sistema a grana grossa (coarse-grain) invece sono presenti pochi e potenti processori, ognuno dei occupa una larga fetta di memoria, che permette a ognuno di questi di svolgere una significante mole di calcoli utilizzato solo i dati nella memoria a loro assegnata. In generale, le macchine SIMD sono costruite per essere delle macchine a grana fine dove tutti i processori lavorano in maniera serrata sul contenuto della loro memoria. Mentre le MIMD sono macchine a grana grossa che possono avere memoria condivisa o distribuita sui processori. La granularità e il livello di parallelismo comportano, quindi, anche delle differenze nel multithreading hardware delle macchine MIMD. Si possono avere [4]:

- Multithreading a grana fine: ogni istruzione passa da un thread all'altro, producendo un'esecuzione intrecciata dei vari thread, che viene realizzata passando a turno da un thread all'altro, saltando quelli che si trovano in stallo in quel momento. Perché questo sia possibile il processore deve essere capace di cambiare thread in esecuzione ad ogni ciclo di clock. Lo svantaggio principale è il rallentamento dell'esecuzione dei singoli thread.
- Multithreading a grana grossa:con questo approccio il thread in esecuzione cambia solo quando si presenta uno stallo "costoso". Non si ha, inoltre, il vincolo che il cambio di thread sia a costo zero e si rende meno probabile il rallentamento dell'esecuzione del singolo thread. Il suo svantaggio è dato dalla facilità ridotta di mascherare le perdite di thhroughtput.

Il livello di parallelismo di un programma può essere classificato a vari livelli, di cui il più alto è il parallelismo a livello di istruzione, come mostra figura 1.5

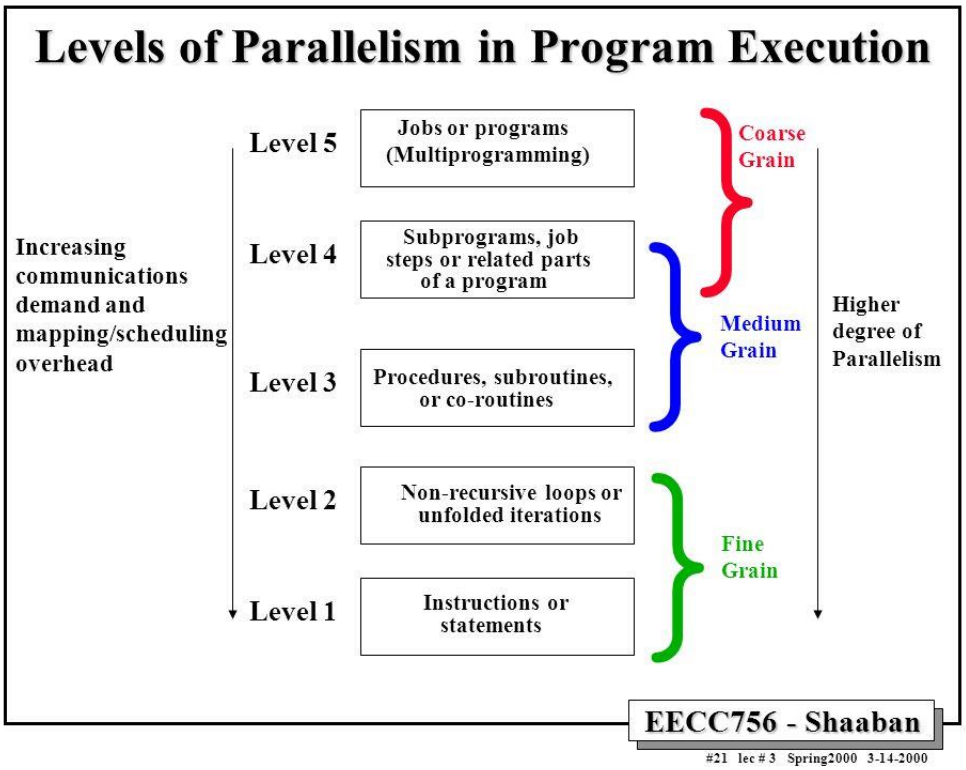


Figura 1.5 Livello di parallelismo in un programma

La maggior parte delle analisi di ILP (Istruction Level Parallelism) sono fatte esaminando le istruzioni generate dal calcolatore. Per ottenere miglioramenti nelle prestazioni, si deve sfruttare l'ILP tra diversi blocchi di base, non nel singolo blocco, nel quale le istruzioni avrebbero una forte dipendenza tra loro, che rende il parallelismo, ossia la possibilità di sovrapposizione tra istruzioni molto bassa. Il metodo più semplice e comune per aumentare l'ILP è il parallelismo a livello di ciclo, facendo in modo che ogni itertazione del ciclo possa sovrapporsi alle altre.

1.5 Valutazione del Calcolo Parallelo

Nella valutazione di algoritmi non si può prescindere dalla legge di Amdahl[1], che fornisce un fattore di guadagno della prestazione ottenibile attraverso alcune migliorie del sistema: *il miglioramento di prestazione ottenibile mediante l'uso di alcune modalità di esecuzione più veloci è limitato dalla frazione di tempo in cui tali modalità possono essere impiegate* [5].
Si definisce, innanzitutto, l'accelerazione (*speedup*) come:

[1] Da Wikipedia [8]: Gene Amdahl (Novembre 1922, Novembre 2015) è stato un ingegnere e informatico statunitense, conosciuto per le sue ricerche sui computer, condotte specialmente nella sua azienda, la Amdahl Corporation, e per aver formulato la legge di Amdahl.

$$S_p = \frac{T_1}{T_p}$$

Dove T_1 è il tempo di esecuzione su un processore e T_p è il tempo di esecuzione su *p* processori (con p numero di processori). Si ha che lo speedup, S_p misura quindi la riduzione del tempo di esecuzione rispetto all'algoritmo su un processore [2].
Da questa si ha che l'algoritmo risulta migliore quanto più lo speedup è vicino, in valore al numero di processrori, quind si puà definire lo speedup ideale come:

$$S_p^{ideale} = p$$

Nel caso reale si avrà che lo speedup è sempre minore al valore di p. Inoltre, si definisce Overhead totale la misura dell'effettiva differenza tra lo speedup effettivo e quello ideale:

$$O_h = pT_p - T_1$$

Con O_h>0.
Si nota che per ottenere una misura utile bisogna rapportare lo speedup al numero di processori utilizzati, per avere l'informazione di quanto siano state usate le risorse di calcolo. Da qui possiamo ricavare il parametro dell'efficienza, ossia, appunto un'indicazione di quanto sia stato utilizzato il parallelismo del calcolatore:

$$E_p = \frac{S_p}{p}$$

Dove l'efficienza ideale è pari a 1 mentre quella reale sempre minore di 1.
In generale, in ogni algoritmo il tempo di esecuzione sequenziale T_1 è costituito da una parte seriale e da una parte parallela (le operazioni che possono essere eseguite concorrentemente), quindi:

$$T_1 = \alpha T_1 + (1 - \alpha)T_1$$

Con α frazione sequenziale del tempo T_1 e (1- α) frazione parallela. Si può quindi scrivere il tempo di esecuzione dell'algoritmo in parallelo come:

$$T_p = \alpha T_1 + \frac{(1 - \alpha)T_1}{p}$$

Da queste possiamo scrivere la formula per lo speedup come:

$$S_p = \frac{T_1}{T_p} = \frac{T_1}{\alpha T_1 + \frac{(1 - \alpha)T_1}{p}} = \frac{1}{\alpha + \frac{(1 - \alpha)}{p}}$$

Questa è conosciuta come la legge di Amdahl, rapido metodo per determinare lo *speedup* e capire da cosa può essere limitato [6], tenendo conto della parallelizzazione dell'algoritmo e partendo dal presupposto che in ogni applicazione ci sia necessariamente una parte seriale. Da cui si evince che se l'algoritmo è puramente seriale, non si ha accelerazione, mentre quando l'algoritmo è completamente parallelizzabile si ha teoricamente un'accelerazione infinita.

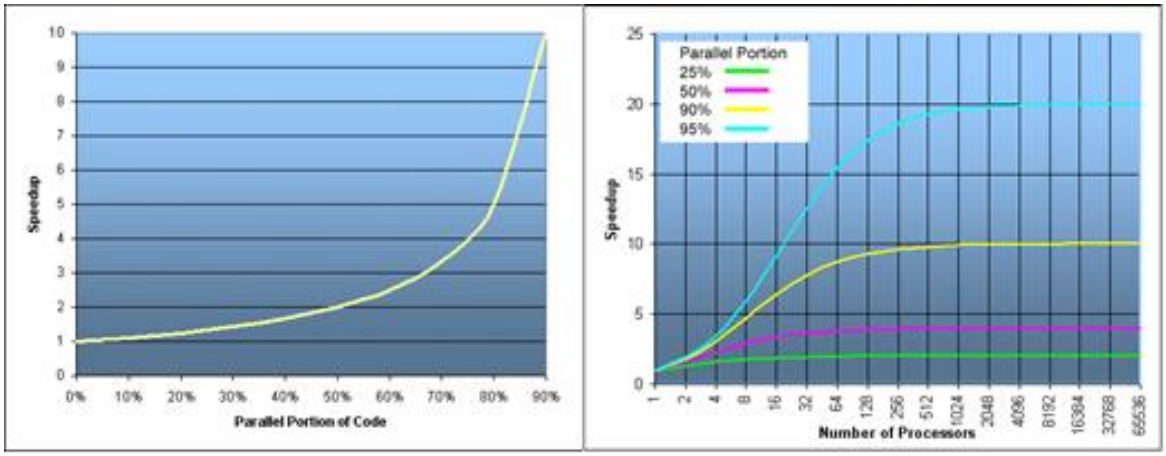


Figura 1.6 Speedup

Ma questa analisi parte dal presupposto che la parallelizzazione di un algoritmo sia fissa e non possa variare nel tempo, ed è limitata dalla scalabilità dell'algoritmo, in quanto molto spesso in base alla grandezza del problema, non si ha un significativo aumento dello *speedup* utilizzando più processori (vedere gli esempi di Murli al capitolo 4 [2]). Infatti, sperimentalmente si è dimostrato che le prestazioni migliorano "aumentando" la grandezza del problema.

Nelle applicazioni in cui la porzione seriale decresce nel tempo, come il network routing, si applica la legge di Gustafson[2] che ridefinisce la "scaled speedup", legando lo *speedup* al numero di processori utilizzati nel tempo, variando nel tempo sia p (numero di processori) che grandezza del problema. Si misura T_1 dato T_p e si definisce α' come la frazione del temp T_p relativa alle operazioni eseguite in parallelo.

$$Scaled - speedup \quad SS_p = p + (1 - p)\alpha'$$

Quando lo speed-up viene misurato scalando la dimensione del problema, α' tende a zero all'aumentare del numero di processori utilizzati [2].

[2] Da Wikipedia [9]: John Leroy Gustafson (Gennaio 1955) è un informatico statunitense conosciuto per il suo lavoro nel HPC tra cui la formulazione della legge di Gustafson.

2. Il Parallel Computing Toolbox

Matlab è l'ambiente software più semplice e performante per lavorare in campo scientifico e ingegneristico utilizzando un linguaggio di programmazione che esprime le operazioni matematiche matriciali in maniera diretta [10], dispone inoltre di numerosi toolbox sviluppati professionalmente. Tra questi, di nostro interesse è il Parallel Computing Toolbox, che permette di risolvere problemi computazionalmente complessi attraverso un codice che funzioni bene in un sistema multi-core, computer cluster e attraverso anche i processori delle GPU, attraverso il DCS (Distribuited Computing Server), permettendo all'utente di utilizzare il paradigma di programmazione più indicato per l'applicazione [11]. Il toolbox ci permette di usare la piena potenza computazionale dei multicore eseguendo le applicazioni sui workers, ossia dei processi computazionali che funzionano in completa indipendenza senza richiedere un'infrastruttura di comunicazione che li colleghi [13]. Il numero dei workers utilizzati dipende dal processore della macchina che si intende utilizzare.
Come scritto da Morel in [14] e Altman in [15], Matlab supporta quattro tipi di parallelismo: multithreaded, distributed computing, e parallelismo esplicito, che possono coesistere tra loro.

- Parallelismo multithread: un'istanza Matlab genera automaticamente flussi multipli e simultanei di informazioni. Questi vengono eseguiti da più processori che condividono la memoria dello stesso calcolatore.
- Distribuited Computing: più istanze Matlab eseguono indipendentemente più processi in computer diversi, ognuno nella propria memoria, generalmente attraverso l'uso di cluster e cloud.
- Parallelismo Esplicito: diverse istanze eseguono si diversi processori o computer, spesso con memorie separate, e simultaneamente, un singolo comando o una singola funzione. Questo tipo di parallelismo è reso possibile da nuovi costrutti tra cui parallel loops e distribuited arrays.
- GPU-delegated processing: dei grossi processi parallelizzati vengono trasferiti alla GPU che li eseguirà attraverso i suoi multipli processori. Questo aspetto verrà approfondito nel capitolo 3.

Nei prossimi paragrafi saranno analizzati i costrutti per il parallelismo esplicito, le informazioni sono per lo più tratte da [15], [12] e [13].

2.1 *Parfor Loops*

La sintassi e la semantica di un ciclo *parfor* sono come quella di un normale ciclo for, il costrutto permette a MATLAB di eseguire una serie di istruzioni, indipendentemente dal set di worker disponibile. Questi devono essere identificati e riservati tramite il comando *parpool*, che crea un pool nel cluster di default con un range specificato dall'utente tramite NumWorkers[parpool]. Ma a differenza del ciclo for convenzionale, i valori delle iterazioni in un parfor devono essere numeri consecutivi interi.

Le iterazioni vengono eseguite dagli workers in maniera indipendente e non deterministicamente. Ogni worker esegue una sola iterazione, se il loro numero è uguale a quello delle iterazioni, altrimenti, se ci sono più iterazioni che workers, alcuni di questi dovranno eseguirne più di una. Questa gestione è del tutto automatica e non gestita dall'utente, ma bisogna comunque assicurarsi che ogni worker abbia accesso alle risorse, dato il lavoro indipendente che fanno.

L'utilità del parfor risiede nei casi in cui si ha bisogno di eseguire più iterazioni dello stesso calcolo che non può essere vettorizzato, come una simulazione Monte Carlo, o quando si hanno cicli che impieghino molo tempo per l'esecuzione, per quanto non possa contenere nel set di istruzioni al suo interno un altro parfor [17]. I workers lavorano indipendentemente tra loro, generalmente eseguendo l'iterazione a loro assegnata su processori separati. Il requisito fondamentale per il buon funzionamento del costrutto è che non ci siano dipendenze dei dati tra le iterazioni del ciclo, che renderebbe il processo di comunicazione troppo complicato.

Come anticipato prima, per poter utilizzare un parfor bisogna innanzitutto creare un "pool" con il comando parpool:

```
>> parpool local
Starting parallel pool (parpool) using the 'local' profile ... connected
to 4 workers.
```

Il comando crea una "pool" di 4 workers. Quando l'esecuzione è terminata, la pool deve essere chiusa con delete (gcp).

Nel corpo di un parfor possono essere utilizzate le variabili nella tabella sottostante:

Data Class	Description
Loop	A loop index variable for arrays
Sliced	An array whose segments are manipulated on different loop iterations
Broadcast	A variable defined before the loop whose value is used inside the loop but never modified
Reduction	Accumulates a value across loop iterations, regardless of iteration order
Temporary	Variable created inside the loop, but unlike sliced or reduction variables, not used outside the loop

Le più importanti, secondo Altman, sono le *sliced variables* perché riducono la comunicazione tra client e worker, perché vengono inviate al worker solo quando deve iniziare a lavorare su un range particolare di indici. Perché una variabile sia definita "sliced" deve rispettare le seguenti condizioni (la variabile i indica il ciclo):

- Il primo livello dell'indice è indicato dalle parentesi '()' o '{}'. Le variabili A{i}.s e A(i,12).s sono *sliced*, A.q{i} no.
- Nello stesso livello di parentesi, la lista degli indici deve essere la stessa per ognuna delle occorrenze. Per esempio: h(A(i), A(h+1)) non è una *sliced*, f(A(i),A(i)).
- Nella lista degli indici di una variabile, quello che gestisce il ciclo è un espressione semplice, mentre gli altri possono essere costanti, variabili broadcast, end o vettori colonna: per esempio: A(i,i+1) o A(i+1,20:30,end) non sono *sliced*, mentre A(i,:,end) e A(i+k,j,:,3) sono *sliced*.
- Il contenuto di una variabile *sliced* non è cambiabile, ossia non sono ammessi operatori di assegnamento e cancellazione. Per esempio: A(i,:)=[] non è una variabile *sliced*.

Le variabili che non soddisfano i requisiti precedenti, e non vengono assegnate all'interno di un ciclo, sono dette broadcast e inviate a ogni worker.
Un altro tipo di variabili interessate sono quelle temporanee, le quali esistono solo nel workspace del worker che le utilizza, non vengono trasferite al client, e non obbediscono alle regole precedenti.

2.2 *Spmd*

Il costrutto *spmd* (single program multiple data) permette all'utente un controllo più diretto sul parallelismo e la distribuzione di dati tra gli workers, rispetto al parfor-loop [15]. Viene definito un blocco di codice che verrà eseguito simultaneamente su più workers che saranno riservati attraverso il costrutto *parpool*. Quando le preferenze impostate non autorizzano la creazione automatica di un pool, MATLAB creerà una pool usando gli workers presenti in quella creata con il comando parpool, mentre generalmente usa i worker di default[spmd].
"Multiple data" vuol dire che lo stesso codice viene eseguito su tutti i worker, che però possono operare su diversi dati.
La forma generale per l'istruzione spmd è

```
spmd
   istruzioni
end
```

Può essere definito anche in single-line, e si possono specificare il numero massimo e minimo di worker che possono essere utilizzati per l'esecuzione del codice, per esempio smpd(3) . Generalmente il costrutto viene usato quando è richiesta un'alta sincronizzazione

nella comunicazione tra i worker, è di rilevante importanza nei programmi con una lunga esecuzione, nei quali più worker eseguono la soluzione simultaneamente, e per i programmi che hanno un grande set di dati, i quali vengono distribuiti a worker.
A differenza che nei cicli parfor, ad ogni worker che esegue un spmd viene assegnato un valore unico attraverso labindex, che permette l'accesso specifico al worker, specificando il codice da eseguire sul determinato worker o l'accesso a dati unici. Il numero totale di worker che esegue il blocco in parallelo è dato da numlabs. La comunicazione tra worker nel corpo di un costrutto spmd è invece data da labSend e labReceive. Il costrutto spmd restituisce variabili che sono state convertite in oggetti di tipo *Composite* nel client MATLAB. Questo tipo di oggetti contiene riferimenti ai valori delle variabili contenute nei worker remoti, che possono essere recuperati attraverso l'indice dei cell-array. Finche l'oggetto composite esiste e la pool rimane aperta, i dati sui worker rimangono disponibili e possono essere utilizzati per più esecuzioni di spmd. I worker di un'istruzione spmd sono consapevoli gli uni degli altri, quindi è possibile controllare in maniera diretta il trasferimento dei dati tra essi, utilizzando una matrice codistribuita, che sarebbe il meccanismo attraverso cui MATLAB partiziona e distribuisce i dati tra i worker. Per esempio:

```
spmd (3)
    RR = rand (30, codistributor());
end
```

Il codice crea una matrice codistribuita 30x30, assegnandone ad ogni worker un segmento 30x10. Oltre questo tipo di arrey, Matlab fornisce una serie di comandi che garantiscono il controllo dell'esecuzione parallela:

- labBarrier: blocca l'esecuzione fino a quando tutti i worker ricevono questa chiamata
- labBroadcast: Spedisce dati a tutti i worker o riceve dati spediti a tutti i worker
- labProbe: Test per verificare se i messaggi sono pronti per essere ricevuti da altri lab
- labReceive: riceve dati da un altro lab
- labSend: Spedisce dati a un altro lab
- labSendReceive: Simultaneamente spedisce dati a e riceve dati da un altro lab.
- gcat: concatenazione globale di una matrice su tutti i worker

2.3 Parfeval

Nei costrutti parfor e spmd, i worker sono sessioni MATLAB *headless* (senza interfaccia grafica), non possono creare grafici o altri tipi di output grafici sul desktop. Perciò in MATLAB 8.2 è stata introdotta il costrutto parfeval, che esegue funzioni MATLAB con i worker della parallel pool senza bloccare i processi desktop. In questo modo si possono creare programmi paralleli asincroni che non blocchino l'uso della GUI:

F=parfeval(p,fcn,numout,in1, in2…)

Nella stringa precedente si richiama l'esecuzione asincrona di *fcn* su un worker della pool p, che restituisca *numout* argomenti in output con i1, i2 eccetera in input. F è un oggetto di tipo *parallel.FevalFuture* , i cui risultati vengono ottenuti tramite *fetchNext(F)*, quando ogni worker ha completato la valutazione, che continua fino alla fine se non viene interrotta dal comando cancel(F).

2.4 Matrici Distribuite e Codistribuite

L'idea alla base delle matrici distribuite è quella di fornire un'astrazione dei dati distribuiti tra tutti i worker, così che i dati possano essere visualizzati come se fossero una sola matrice nel workspace di MATLAB. Una matrice distribuita viene dichiarata e utilizzata come una normale matrice, ma i suoi elementi non esistono nel client. La differenza tra matrici distribuite e codistribuite è semplicemente una questione di prospettiva: una matrice codistribuita esistente sui worker è accessibile dal client come matrice distribuita e viceversa. Le matrici distribuite vengono create attraverso il comando *distributed*, che distribuisce una matrice esistente dal client sui worker del pool di MATLAB, e si può accedere ai suoi dati tramite dei blocchi spmd. Si può creare una matrice distribuita sui worker anche senza la matrice preesistente sul client, i comandi sono gli stessi per la dichiarazione di matrici non distribuite su MATLAB:

Distributed Method	Description
`distributed.cell(m,n,...)`	Create a distributed cell array
`eye(m,...,class,'distributed')`	Create a distributed identity matrix of given type
`distributed.spalloc(m,n,nzmax)`	Allocate space for a sparse distributed matrix
`distributed.speye(m,n)`	Create a distributed sparse identity matrix
`ones(m,n,...,'distributed')`	Create a distributed array of ones
`zeros(m,n,...,'distributed')`	Create a distributed array of zeros
`rand(m,n,...,'distributed')`	Generate a distributed array of uniformly distributed pseudo-random numbers
`randn(m,n,...,'distributed')`	Generate a distributed array of normally distributed pseudo-random numbers
`randi(m,n,...,'distributed')`	Generate a distributed array of distributed pseudo-random integer numbers
`true(m,n,...,class,'distributed')`	Create a distributed array of logical ones
`false(m,n,...,class,'distributed')`	Create a distributed array of logical zeros

Quando si distribuisce una matrice a un numero di worker, viene partizionata in segmenti e ognuno di questi è assegnato a un worker. Una matrice bidimensionale può essere partizionata per colonne, assegnandone una della matrice originale ad ogni worker, o per righe. Ogni worker ha accesso a tutti i segmenti della matrice codistribuita, nonostante l'accesso ai segmenti locali sia più veloce rispetto a quello agli accessi remoti, perché questo richiederebbe l'invio e la ricezione di dati tra worker. Si accede esplicitamente a una porzione locale di una matrice codistribuita attraverso il comando *getLocalPart*.

La forma non distribuita di una matrice può essere ripristinata con il comando gather, che prende il segmento di una matrice su un worker e lo combina in una replica della matrice su

tutti i worker, o come singola matrice su un worker. MATLAB ci mette inoltre a disposizione dei costrutti per costruire matrici distribuite che abbiano specifici valori, dimensioni e classe.

2.5 Pmode

La funzione pmode permette all'utente di lavorare interattivamente su un processo parallelo eseguito su più worker. I comandi specificati con pmode vengono eseguiti da tutti i worker contemporaneamente, ma nel proprio workspace con le proprie variabili, che possono essere trasferite tra il client MATLAB e i worker. Il comando pmode è molto simile, nell'esecuzione, al spmd quindi permette l'uso di tutte le funzioni della Tab.1.
Ovviamente ci sono delle differenze tra i due costrutti, nel pmode si ha meno controllo sull'esecuzione parallela, e questa non può coesistere con dei comandi seriali.
Quando si esce dalla sessione pmode i processi degli worker vengono chiusi efficacemente e tutti i worker e i dati vengono persi. Anche nelle sessioni pmode non è abilitata la comunicazione con la GUI, quindi per produrre dei grafici bisogna trasferire i dati al client usando un comando speciale. I comandi usati con il pmode sono:

- Pmode start [prof] [numworkers]
  Avvia la sessione pmode, specificando eventualmente il profilo PCT da utilizzare e il numero di worker.
- Pmode quit/pmode exit
  Termina la sessione pmode, cancellandola.
- Pmode client2lab clientvar workers [workervar]
  Copia la variabile *clientvar* dal client MATLAB sulla variabile *workervar* sui workers identificati da *workers*.
- Pmode lab2client workervar worker [clientvar]
  Copia la variabile *workervar* dal worker identificato da *worker*, sulla variabile *clientvar* sul client MATLAB.
- Pmode cleanup prof
  Cancella la sessione creata da pmode dell'utente sul cluster identificato da *prof*.

Il pmode può essere invocato sia come comando che come funzione:

```
>> pmode start myProfile 4
>> pmode('start', 'myProfile', 4) % alternative
```

3. GPU

Oltre che all'uso di multiprocessori e cluster, il PCT consente l'esecuzione dei calcoli direttamente da MATLAB utilizzando le schede grafiche Nvidia abilitate CUDA, attraverso uno speciale tipo di array, *gpuarray*, e le sue funzioni associate [15]. CUDA è la piattaforma di calcolo parallelo e l'architettura di elaborazione in parallelo sviluppata dalla NVIDIA che permette il netto aumento delle prestazioni di calcolo grazie alla potenza delle GPU [18], nelle applicazioni accelerate tramite le schede grafiche, la parte sequenziale del programma viene affidata alla CPU, i calcoli intensivi, invece, vengono svolti parallelamente dalla GPU [19].

Il sito della NVIDIA e MathWorks mettono a disposizione svariata documentazione e tutorial per l'uso di CUDA nelle applicazioni di Intelligenza Artificiale e Deep Learning.

La migliore architettura per GPU disponibile ora sul mercato è la Volta, utilizzata negli acceleratori GPU Tesla prodotti dalla NVIDIA. La sempre maggiore richiesta nell'ambito dei videogames e della grafica computerizzata ha fatto si che le architetture per l'elaborazione grafica si sviluppassero a velocità maggiore rispetto ai microprocessori tradizionali. La potenza delle schede grafiche NVIDIA risiede nella quantità di processori paralleli specializzati che vengono utilizzati, e tende a smentire le dinamiche della scala di Moore[3], con un tasso di crescita delle prestazioni del 50% annuo, in contrasto con quello del 10% delle CPU, frenato dalla fisica dei semiconduttori [21].

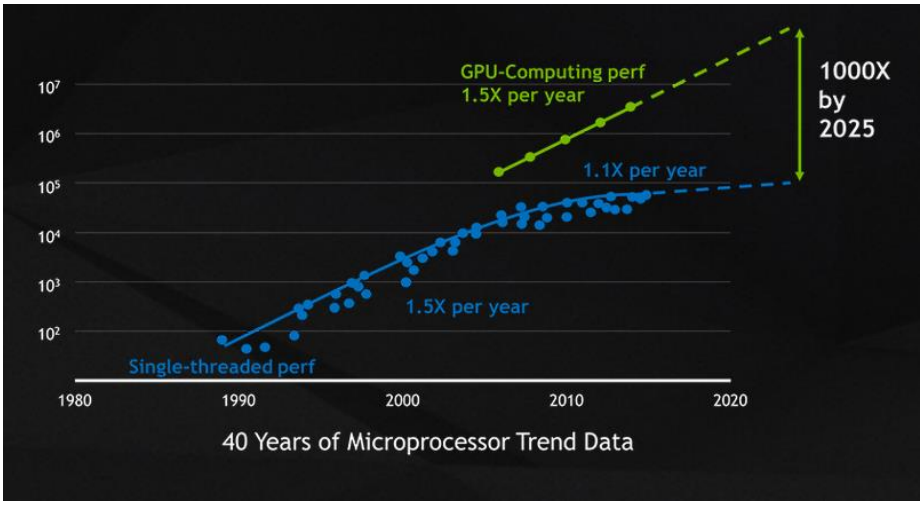


Figura 3.1 Andamento delle prestazioni computazionali

[3] La Legge di Moore, imprenditore e informatico statunitense (San Francisco, 3 gennaio 1929) [22], è un'osservazione e previsione empirica sull'avanzare dello stato dell'arte dei dispositivi a semiconduttore, che sostiene che il numero di transistor in un chip aumenti con velocità costante predicendo che questa crescita sia continui per decenni. Attualmente il numero di transistor raddoppia ogni 1 8 mesi, ossia si ha un incremento di circa il 60% ogni anno. La legge di Moore è rimasta valida per oltre tre decenni [3].

La sempre maggiore tendenza dei programmatori a utilizzare GPU invece che CPU è data, anche, da alcune differenze sostanziali tra queste, come descrive Patterson in [4].
Generalmente le GPU sono acceleratori che completano le CPU, ma che non eseguono tutti i compiti delle CPU, perciò possono dedicarsi maggiormente all'elaborazione grafica, a meno che non sia esplicitamente richiesto il suo utilizzo per l'applicazione. Questo ha portato uno sviluppo di diversi tipi di architetture. La differenza maggiore è che le GPU per superare la latenza della memoria utilizzano molti thread invece che cache multilivello come le CPU; perciò la memoria principale della GPU viene ottimizzata per massimizzare la larghezza di banda di trasferimento e vengono montati dei chip DRAM (Dynamic Random Access Memory) diversi da quelli delle CPU. Inoltre, le GPU possono contenere sia molti processori paralleli (MIMD) che molti thread, quindi ogni GPU, di per sé, è un processore multithread con un numero di thread e processori più elevato di una CPU.
Inoltre, in accordo con quanto sostiene Altman [15], l'uso della GPU nella risoluzione di un problema è molto efficace quando il problema è:

- *Massicciamente parallelo*: le computazioni possono essere suddivise in unità di lavoro indipendenti. Queste vengono create distribuendo grandi set di dati tra i vari core delle GPU, così che ogni core performi la stessa tsk utilizzando un diverso set di dati (data-level parallelism).
- *Computazionalmente intensivo*: Il tempo di computazione è significativamente maggiore rispetto a quello speso per il trasferimento dei dati alle GPU.

Il capitolo è suddiviso in tre parti: nella prima si fornirà una panoramica delle architetture per GPU, nella seconda verranno descritte le basi del CUDA, nella terza si vedranno i metodi messi a disposizione da MATLAB per velocizzare le prestazioni dei programmi.

3.1 Introduzione alle GPU – Architettura

Come precedentemente introdotto, le GPU lavorano efficacemente con i problemi che esibiscono parallelismo a livello dei dati [4], e si basano sul multithreading hardware all'interno di ogni processore SIMD a thread multipli per nascondere la latenza di memoria. Quindi una GPU è un'architettura MIMD composta da processori SIMD multithread. Per consentire la scalabilità trasparente tra modelli di GPU, con un diverso numero di processori SIMD, vengono assegnati blocchi di thread[4] ai diversi processori da un componente hardware chiamato Thread Block Scheduler. L'oggetto che viene così creato è il thread di istuzioni SIMD, ognuno di questi thread ha il proprio *program counter* e viene eseguito su un processore SIMD multithread. Lo scheduler è dotato di un controllore che informa

[4] Blocco di thread: insieme di thread concorrenti che eseguono lo stesso programma di thread e possono collaborare per calcolare il risultato[4].

l'architettura su quali siano i thread di istruzioni pronti per l'esecuzione, e di un'unità che li invia al processore SIMD multithread. Il processore SIMD deve essere costituito da unità funzionali parallele per eseguire le operazioni, poiché il thread consiste in istruzioni SIMD. Queste unità vengono chiamate cammini di elaborazione SIMD. La memoria locale, quella a disposizione sul chip, su una GPU viene condivisa tra i diversi cammini di elaborazione SIMD all'interno dello stesso processore.

Perciò, come descritto in [23], la GPU si è evoluta in un processore parallelo programmabile con una potenza computazionale che va oltre quella della CPU. L'architettura Tesla della NVIDIA, introdotta nel Novembre 2006 [5], rende possibili alte performance nelle applicazioni del calcolo parallelo scritte in linguaggio C usando il modello di programmazione CUDA (Compute Unified Device Architecture).

L'architettura Tesla è basata su un processore scalabile, la Figura 3.2 Mostra un diagramma a blocchi di una GPU GeForce 8800 con 128 streaming-processor (SP) core organizzati in 16 multiprocessori streaming (SMs) divisi in otto unità procedurali indipendenti chiamate texture/processor clusters (TPCs).

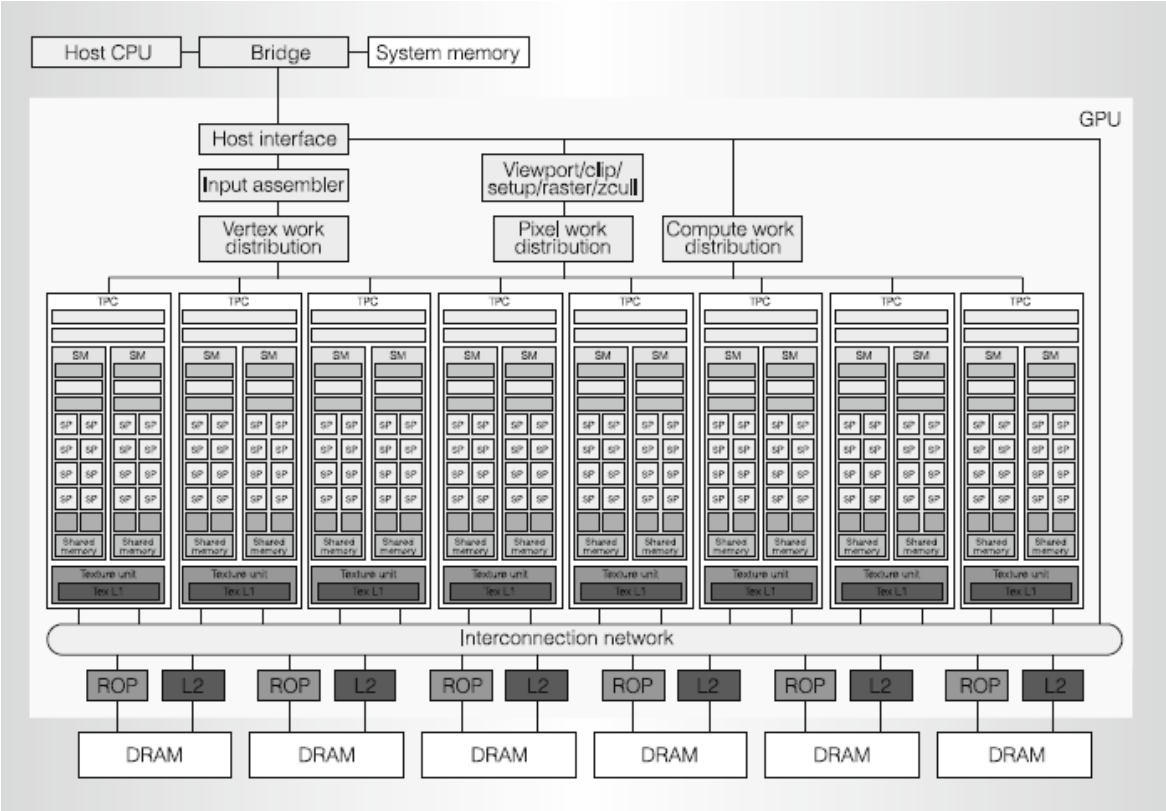


Figura 3.2 Architettura di base di una GPU unificata

Al più alto livello lo streaming processor array (SPA) della GPU performa tutti i calcoli della scheda grafica programmabile. Il sistema di memoria scalabile consiste in un controllo DRAM (Dynamic Random Access Memory) esterno e processori *fixed-function* che

[5] L'ultima architettura di GPU sviluppata dalla NVIDIA è la VOLTA, introdotta nel 2016, disponibile sul mercato dal 2017 con gli acceleratori Tesla V100 [29].

eseguono operazioni di buffer del colore e della profondità direttamente sulla memoria. La rete, inoltre, guida le richieste di lettura della memoria dagli SPA alle DRAM e scrive dati dalle DRAM alle SPA. Il PCT esegue programmi *vertex shader*, il cui risultato viene scritto su buffer on-chip. Questi buffer passano i loro risultati al blocco viewport/clip/setup/raster/zcull, che vengono poi suddivisi in pixel e distribuiti tra i TCP per essere elaborati. Gli SPA accettano ed eseguono simultaneamente elaborazioni per *stream* logici multipli. Clock multipli per GPU, processori e DRAM consentono un'ottimizzazione di potenza e prestazioni indipendente.

Questa architettura Tesla scalabile per il calcolo parallelo abilita il processore GPU all'esecuzione di applicazioni di calcolo ad alte prestazioni. Come già detto, per eseguire efficientemente un grande problema computazionale su un processore ad alta architettura parallela, bisogna scomporre il problema in sotto problemi risolvibili indipendentemente in parallelo.

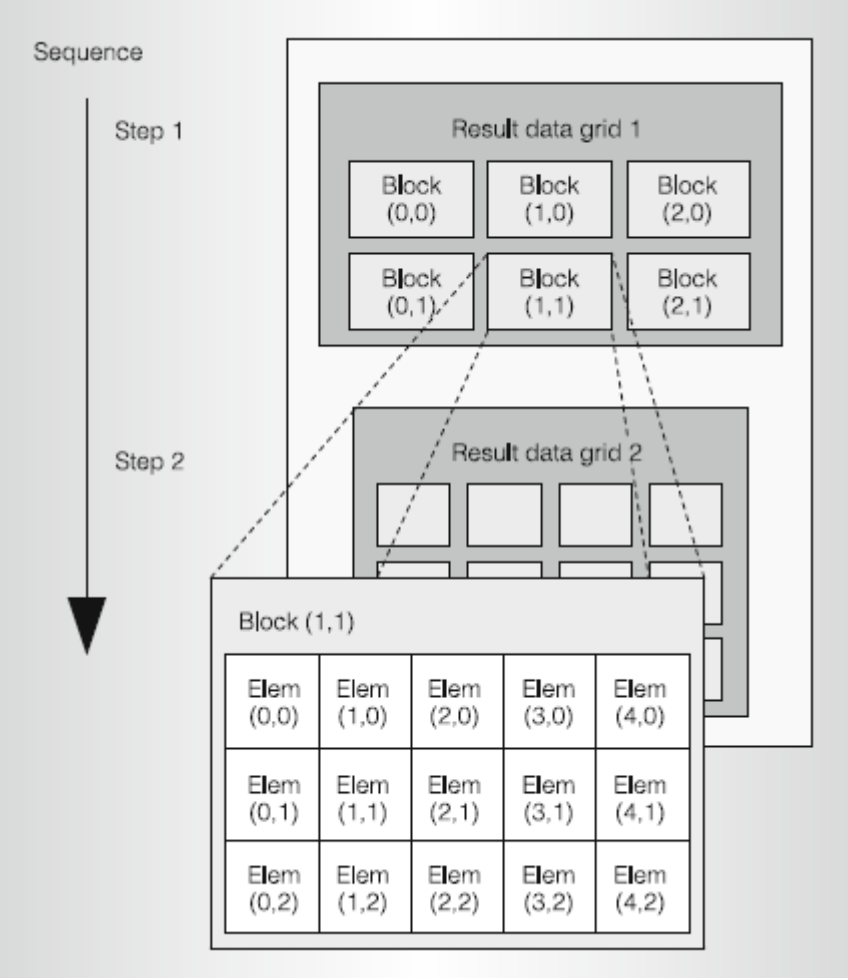


Figura 3.3 Scomposizione dei dati

In Figura 3.3 è stata decomposta una matrice in una griglia di blocchi 3x2 e ognuno dei blocchi è diviso in una matrice di 5x3 elementi. Questa scomposizione si applica bene all'architettura Tesla, gli SMs operano parallelamente sui blocchi e i thread operano parallelamente sugli elementi. Il programmatore scrive un programma che compili una

sequenza di griglie di risultati, partizionando ogni griglia in blocchi coarse-grained su cui si opera indipendentemente in parallelo. Il programma computa ogni blocco di risultati con una matrice di *fine-grained* threads, partizionando il lavoro tra i thread che operano sugli elementi risultanti [23].

Il calcolo parallelo richiede sincronizzazione e comunicazione tra i thread paralleli per arrivare efficientemente a un risultato. Per gestire il grande numero di thread che devono collaborare, l'architettura Tesla introduce il *cooperative thread array* (CTA), che viene chiamato thread block nella terminologia CUDA. Questo è un array di thread concorrenti che eseguono lo stesso programma, ed è composto da 1 a 512 thread, ognuno dei quali ha il proprio ID (TID), numerato da 0 a m, che può avere un indice di una, due o tre dimensioni. I thread di un CTA possono dividere i dati con memoria distribuita o condivisa, il lavoro viene poi assegnato attraverso il loro indice. Ogni SM esegue fino a 8 CTAs contemporaneamente, in base alla richiesta di risorse da parte del CTA. Il programmatore dichiara il numero di thread, registri e memoria condivisa richiesti dal programma CTA. Quando un SM ha abbastanza risorse disponibili, l'SMC crea il CTA i numeri TID ad ogni thread. Quindi l'SM esegue i thread CTA.

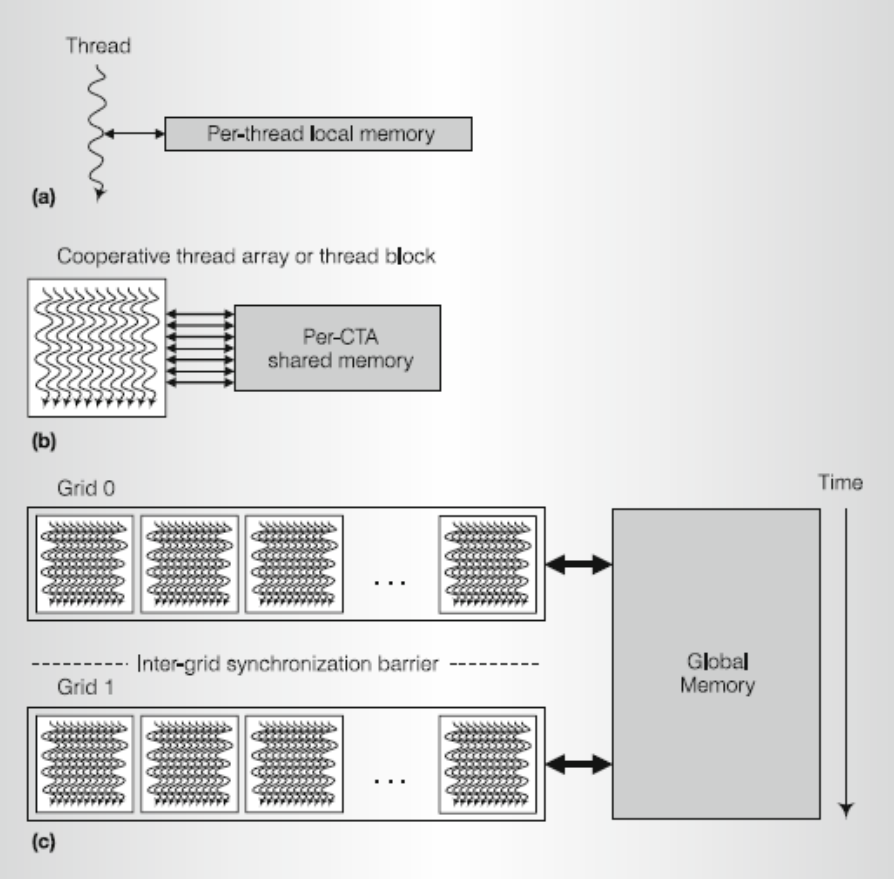


Figura 3.4 Livelli di granularità innestati

La figura 3.4 Mostra i livelli di granularità parallela in un modello di GPU per il calcolo, i tre livelli sono:

- Thread: calcola gli elementi risultanti selezionati dal il suo TID
- CTA: calcola i blocchi risultanti selezionati dal suo CTA ID

- Grid: calcola più blocchi risultanti, e griglie sequenziali calcolano le fasi di applicazioni sequenzialmente dipendenti.

Vengono inoltre mostrati i livelli di parallelismo della memoria condivisa di lettura/scrittura:

- Locale: ogni thread dispone di memoria locale privata, che verrà utilizzata da CUDA per allocare quelle variabili che non trovano spazio nei registri di thread.
- Condivisa: ogni blocco di thread dispone di una memoria condivisa visibile a tutti i thread
- Globale: tutti i thread hanno accesso alla memoria globale, nella quale le griglie comunicano e condividono grandi set di dati.

3.2 Il modello di programmazione CUDA

CUDA è l'architettura di elaborazione in parallelo di NVIDIA che permette netti aumenti delle prestazioni di computing grazie allo sfruttamento della potenza di calcolo delle GPU [18].

Sebbene le GPU siano state inizialmente progettate per un numero ristretto di applicazioni, alcuni programmatori si chiesero se potessero tradurre le loro applicazioni in una forma che consentisse loro di sfruttare le alte prestazioni delle GPU [4]. Perciò svilupparono un linguaggio di programmazione ispirato a C che consentisse di scrivere direttamente programmi per la GPU. NVIDIA decise che il tema di unificazione di queste forme di parallelismo dovesse essere il thread CUDA, usando questo livello basso di parallelismo come primitiva di programmazione, il compilatore e l'hardware possono raggruppare migliaia di thread CUDA per sfruttare i diversi tipi di parallelismo contenuti all'interno della GPU: multithreading, SIMD, MIMD e parallelismo a livello di istruzioni. Come già scritto nel capitolo precedente, questi thread vengono raggruppati a blocchi di 342 ed eseguiti contemporaneamente. L'azienda ha fatto evolvere il paradigma della "centralizzazione" su CPU a quello del "co-processing" su CPU e GPU, mettendo, inoltre, a disposizione degli utenti la piattaforma di elaborazione in parallelo CUDA®[6], che fornisce delle semplici estensioni di C e C++ che consentono di esprimere sia i dati fine-grained che quelli coarse-grained, oltre al parallelismo dei thread.

CUDA fornisce tre astrazioni fondamentali per la programmazione di GPU: una gerarchia di gruppi di thread, memorie condivise e sincronizzazioni a barriera. Il modello di programmazione scala in maniera trasparente su un numero molto elevato di processori perché un programma CUDA può essere eseguito su un qualsiasi numero di processori.

Il programma deve essere scritto come un programma sequenziale che richiama kernel[7] paralleli che possono contenere semplici funzioni o programmi completi. Il kernel deve

[6] https://developer.nvidia.com/cuda-zone

[7] Kernel: un programma o funzione per un thread, progettato per essere eseguito da molti thread.

essere eseguito parallelamente su un insieme di thread paralleli, che sono organizzati dal programmatore in una gerarchia di blocchi di thread e griglie di blocchi di thread (vedi paragrafo precedente, l'architettura delle GPU NVIDIA è strettamente legata al paradigma di programmazione CUDA). Per lanciare l'esecuzione di un kernel, il programmatore deve specificare il numero di thread per blocco e il numero di blocchi della griglia, attraverso il TID dei thread. Il testo di un kernel è quindi una funzione scritta in C per un thread sequenziale, nel quale il parallelismo è determinato dalla specificazione delle dimensioni di griglia e blocchi. L'esecuzione parallela e, quindi, la gestione, creazione, pianificazione e terminazione dei thread sono gestite automaticamente dal sistema sottostante. Infatti, le architetture Tesla effettuano questa gestione dei thread a livello hardware. Il modello CUDA offre al programmatore un livello di virtualizzazione dei thread che permette totale flessibilità nella scelta sul grado di granularità con cui parallelizzare il codice. Per questo, e per la scalabilità tra processori, CUDA richiede l'indipendenza tra i blocchi di thread, che, appunto, consente l'esecuzione dei blocchi in qualsiasi ordine su un qualsiasi numero di processori e varietà di architetture parallele.
Per quanto riguarda la gestione della memoria globale vista dai kernel tramite chiamate runtime CUDA, vengono usate le funzioni cudaMalloc() e cudaFree(). Per copiare i dati tra lo spazio allocato e la memoria di sistema della macchina si usa invece cudaMemcpy(), dato che i kernel possono essere eseguiti su un sistema fisicamente diverso.
Il modello di programmazione CUDA è simile al modello SPMD perché il parallelismo è espresso in modo esplicito e ogni kernel viene eseguito su un numero fisso di thread. Nonostante questo, un programma CUDA risulta più flessibile rispetto alle implementazioni SPMD perché ogni chiamata a kernel crea in maniera dinamica le nuove griglie e blocchi di thread richiesti dall'applicazione.
Nonostante l'estrema efficienza e semplicità di implementazione dei modelli CUDA, sono presenti alcune restrizioni. Innanzitutto, i thread e i blocchi di thread possono essere creati soltanto invocando un kernel parallelo e non dall'interno dei kernel paralleli, che insieme all'indipendenza dei blocchi di thread rende l'esecuzione del programma possibile attraverso uno scheduler (vedi 3.1). Inoltre, non sono al momento concesse le chiamate a funzioni ricorsive esplicite nei kernel CUDA, perché richiederebbe notevoli quantità di memoria da mettere a disposizione per i thread.
Per la gestione dell'architettura eterogenea di un sistema CPU-GPU, i programmi CUDA devono copiare dati e risultati tra le due memorie.
Infine, i modelli di programmazione parallela per grafica e calcolo, essendo strettamente legati all'architettura di elaborazione, hanno segnato una sostanziale differenza tra le architetture per GPU e CPU. Gli aspetti principali dei programmi per GPU che ne influenzano l'architettura sono [4]:

- Uso intensivo del parallelismo dei dati a grana fine: i programmi di shading descrivono come elaborare un singolo pixel o vertice, mentre i programmi CUDA descrivono come calcolare un singolo risultato.
- Modello di programmazione organizzato in thread: Una GPU può creare ed eseguire milioni di thread per ogni immagine, ogni thread di un programma CUDA genera un singolo risultato.
- Scalabilità: un programma aumenta automaticamente le proprie prestazioni avendo a disposizione più processori, senza apportare modifiche al codice.
- Elaborazione intensiva in virgola mobile.
- Supporto all'elaborazione di grandi quantità di dati.

3.3 MATLAB e i gpuarray

MATLAB permette di accelerare ulteriormente i programmi in parallelo utilizzando le prestazioni della GPU. Per prima cosa bisogna verificare se la scheda grafica che si vuole usare supporta il CUDA e se ha abbastanza capacità computativa. Siccome non tutte le schede grafiche sono supportate da MATLAB, bisogna usare una GPU abilitata CUDA con capacità di computazione superiore a 1.3 o superiore (si può facilmente controllare sul sito ufficiale della NVIDIA [24]). Attualmente il PCT supporta solo schede grafiche NVIDIA.

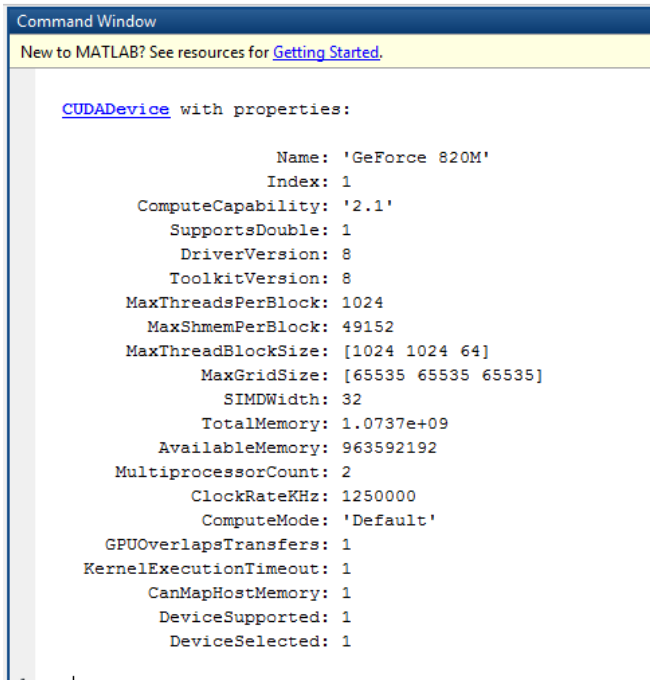


Figura 3.5 Esempio di utilizzo del comando gpuDevice

Per verificare attraverso MATLAB la capacità computativa della GPU che si vuole utilizzare, basta usare il comando gpuDevice, che ne mostra una serie di proprietà rilevanti (Figura 3.5).

I dati immagazzinati nella GPU da MATLAB sono rappresentati da un oggetto gpuarray che può essere di tutti i tipi implementati dal software: single, double, int8, int16, int32, int64, uint8, uint32, uint64 o logical. Per esempio:

```
X = rand(1000);
G = gpuarray(X);
```

Il codice trasferisce la matrice 1000x1000 'X' sulla GPU, restituendo un oggetto gpuarray. Così i dati possono essere manipolati attraverso uno dei metodi definiti per il tipo di oggetto, oppure essere passato al metodo feval[8] di un oggetto CUDA kernel. Il tipo di oggetto gpuarray è anche provvisto di una serie di metodi statici per la costruzione diretta di array sulla GPU senza il trasferimento di dati dal workspace di MATLAB, i quali non richiedono allocazioni di memoria nella CPU. Alcuni dei metodi sono elencati nella tabella sottostante [15]:

Metodo gpuarray	Descrizione
colon (a, d, b)	Genera un vettore a:d:b
eye(m,…, class, codist)	Crea una matrice identità di tipo assegnato
linspace(m, n,…., codist)	Genera un vettore distribuito linearmente
logspace(m, n,…., codist)	Genera un vettore distribuito logaritmicamente
ones(m, n,…., codist)	Crea un array di uno di tipo dato
zeros(m, n,…., codist)	Crea un array di zero di tipo dato
rand(m, n,…., codist)	Genera un array di numeri pseudo-random distribuiti uniformemente
randn(m, n,…., codist)	Genera un array di numeri pseudo-random distribuiti normalmente
true(m, n,…., class, codist)	Crea un array di uno logici
false(m, n,…., class, codist)	Crea un array di zero logici

Esempio: Costruzione di una matrice Identità sulla GPU [26]

```
II = eye(1024,'int32','gpuArray');
size(II)
```

Crea una matrice identità 1024x1024 di tipo int32 sulla GPU

Dopo che il dato è stato trasferito o inizializzato sulla GPU esistono tre approcci per eseguire calcoli GPU su MATLAB:

- Usare una delle numerose funzioni esistenti in MATLAB che supportino i dati di tipo gpuarray.
- Implementare una funzione MATLAB eseguibile sulla GPU usando arrayfun e bsxfun.
- Creare un GPU kernel che elaborerà i dati gpuarray di MATLAB.

In generale la prima opzione è la più semplice, mentre la terza la più difficile da implementare.

[8] feval(KERN, x1,..xn) è una funzione che valuta il CUDA kernel KERN con gli argomenti in ingresso x1,..,xn. I dati in ingresso possono essere dati regolari o gpuarray [25].

Il primo approccio non richiede modifiche al codice originale, tranne che per la conversione dei dati in ingresso in un tipo gpuarray, e non richiede conoscenze CUDA. Nella stessa chiamata di funzione possono essere utilizzati in ingresso sia dati di tipo gpuarray che dati sul workspace MATLAB, esiste grande set di funzioni che restituiscono un oggetto di tipo array se almeno uno degli input lo è. Il processo per implementare un programma MATLAB basato sulla CPU nella GPU è abbastanza semplice. Bisogna per prima cosa spostare i dati dallo workspace alla GPU con gpuarray, o crearlo direttamente sulla GPU. Usare lo stesso processo che si sarebbe per implementare il codice sulla CPU e poi spostare nuovamente i dati nella memoria di MATLAB con il comando gather.
La lista completa delle funzioni che supportano gpuArrey nella versione corrente di Matlab può essere visionata tramite il comando methods('gpuArray'). Oltre queste ci sono varie funzioni MATLAB che non sono metodi della classe gpuArray ma che possono lavorare con i dati di tipo gpuArray. Alcune sono elencate nella tabella sottostante.

Small Subset of the Functions That Work with *gpuArray* Data (There Are Many More)				
angle	*flipud*	*iscolumn*	*kron*	*squeeze*
blkdiag	*flipdim*	*ismatrix*	*mean*	*std*
cross	*fftshift*	*isrow*	*perms*	*rot90*
fliplr	*ifftshift*	*isvector*	*rank*	*trace*

Il secondo approccio è invece basato sull'invocazione dei metodi arrayfun and bsxfun in una funzione che possa lavorare con i tipi di dati gpuArray. La sintassi è uguale a quella per i tipi di dato classici, ma il programma viene eseguito dalla GPU. La sintassi del costrutto standard è:

```
B = arrayfun(fun,A,..)
```

Nel quale viene applicata la funzione func all'elemento A, un elemento per volta. Gli output di func vengono poi concatenati nell'array di output B [27]. L'input fun gestisce una funzione che prende in ingresso svariati argomenti, dei quali almeno uno deve essere un gpuarray e restituisce un gpuarray. Il metodo bsxfun ha la seguente sintassi:

```
C = bsxfun(fun, A, B)
```

Che applica la funzione binaria specificata dalla funzione handle fun agli array A e B (dei quali almeno uno deve essere di tipo gpuArray) e restituisce l'output gpuArray C, che sarà recuperato attraverso il comando gather.

4. Analisi di algoritmi paralleli con MATLAB

In questo capitolo verranno analizzati gli algoritmi di prodotto matrice per vettore e matrice per matrice, confrontando i tempi di esecuzione tra algoritmo seriale, algoritmo parallelo e uso dei gpuarray.
Il capitolo è suddiviso in due paragrafi. Nel primo verrà illustrata la metodologia per parallelizzare l'algoritmo del prodotto matrice per vettore, come spiega Murli in [2], nel secondo verranno trattate le simulazioni.

4.1 Parallelizzazione del prodotto matrice per vettore

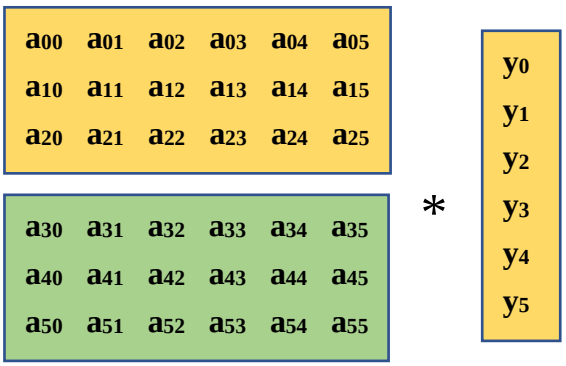
Nel progettare l'algoritmo del prodotto tra matrice e vettore, bisogna tenere conto che esistono più processori, ognuno con una propria memoria, capaci di eseguire ognuno un proprio insieme di istruzioni [2].
Il prodotto che si vuole calcolare è:

$$Ax = y \quad dove\ A \in \Re^{nxn}, x, y \in \Re^n$$

L'algoritmo sequenziale per la risoluzione è dato da:

$$y_i = \sum_{j=1}^{n} a_{i,j} \cdot x_j \quad con\ i = 1, \dots, n$$

Dove viene calcolata una componente per volta nell'ordine prestabilito.
Si nota, però, che il calcolo di ciascun prodotto può essere effettuato indipendentemente dagli altri. Si può, quindi, pensare di adattare l'algoritmo ad un ambiente MIMD, distribuendo il calcolo delle componenti di *y*.
Supponiamo che A sia una matrice quadrata 6x6, x ed y vettori di lunghezza 6 e di risolvere il problema con due processori. Si può decomporre la matrice in due blocchi di righe[9]:



[9] Per le diverse distribuzioni dei dati vedere [2].

Il primo blocco, costituito dalle prime 3 righe viene assegnato al primo processore, il secondo blocco, composto dalle rimanenti tre righe, viene assegnato al secondo processore. Il vettore x viene assegnato a entrambi i processori. I dati saranno così distribuiti localmente ai due processori:

P_0:	P_1:
$A_{loc}(0:2,:) = A(0:2,:)$ $x_{loc} = x(0:5)$ $y_{loc} = y(0:5)$	$A_{loc}(3:5,:) = A(3:5,:)$ $x_{loc} = x(0:5)$ $y_{loc} = y(0:5)$

A_{loc}, x_{loc} e y_{loc} sono le variabili locali residenti nella memoria di ciascun processore.
Ogni processore calcola il prodotto righe per colonne utilizzando i dati nella propria memoria, ossia esegue lo stesso algoritmo ma su dati diversi. Al termine della fase di calcolo il primo processore ha le prime 3 componenti del vettore y, mentre il secondo ha le ultime 3. Per fare si che entrambi i processori abbiano tutte le componenti ci deve essere uno scambio dei dati tra i due processori.

4.2 Valutazione delle prestazioni del prodotto matrice vettore e matrice per matrice.

In questo paragrafo viene analizzata l'implementazione in MATLAB degli algoritmi matrice per vettore e matrice per matrice, secondo le modalità di cui al punto precedente, usando il costrutto spmd, che risulta molto semplice in quanto non richiede la presenza di cicli for. Per confrontare le prestazioni sono stati fatti dei test al variare della dimensione delle matrici A e B e del vettore x.
Come si nota dalle Figure 4.1 e 4.2, le prestazioni ottenute utilizzando il costrutto spmd sono nettamente più basse di quello che si sarebbe aspettato. Questo perché, per operazioni così "semplici", usando il PCT si occupa del tempo per smistare i dati tra i vari worker, mentre la moltiplicazione standard effettuata da MATLAB è stata ottimizzata dagli sviluppatori. Entrambe le simulazioni sono state eseguite usando 12 worker su due macchine, una avente 24 processori, l'altra 48, si nota però che il tempo di esecuzione è migliore nel caso della macchina con 48 processori. Fig. 4.1 mostra 10 simulazioni nelle quali la dimensione della matrice varia da 1000 a 10000. La fig. 4.2 mostra 5 simulazioni utilizzando una matrice con dimensione variabile da 1000 a 5000.
Si nota, invece, dalle figure 4.3 e 4.4 che usando matrici e vettori di tipo gpuarray i miglioramenti alle prestazioni di calcolo sono assicurate dopo la prima simulazione l'operazione viene svolta in meno di 0,05 secondi.
Entrambe le simulazioni sono state eseguite sulla NVIDIA Ge Force GT 650M, con potenza di calcolo pari a 3, per l'operazione matrice per vettore vengono fatte 10 simulazioni, per quella matrice per matrice 4.

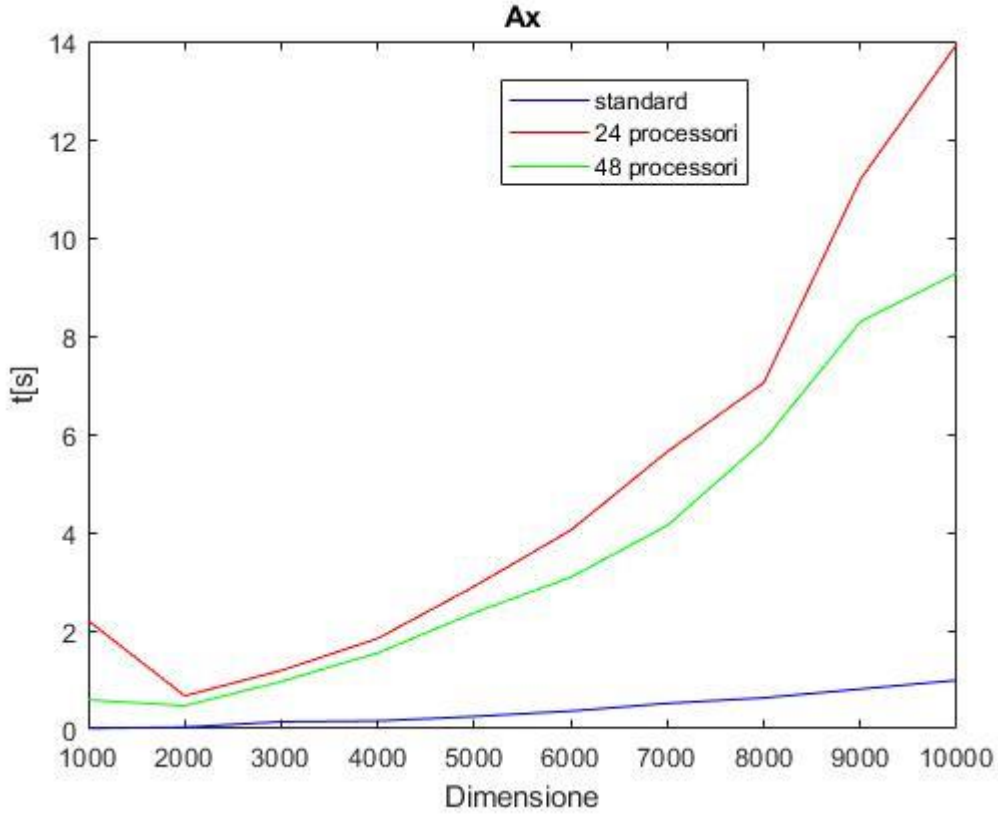


Figura 4.1 Operazione matrice per vettore.

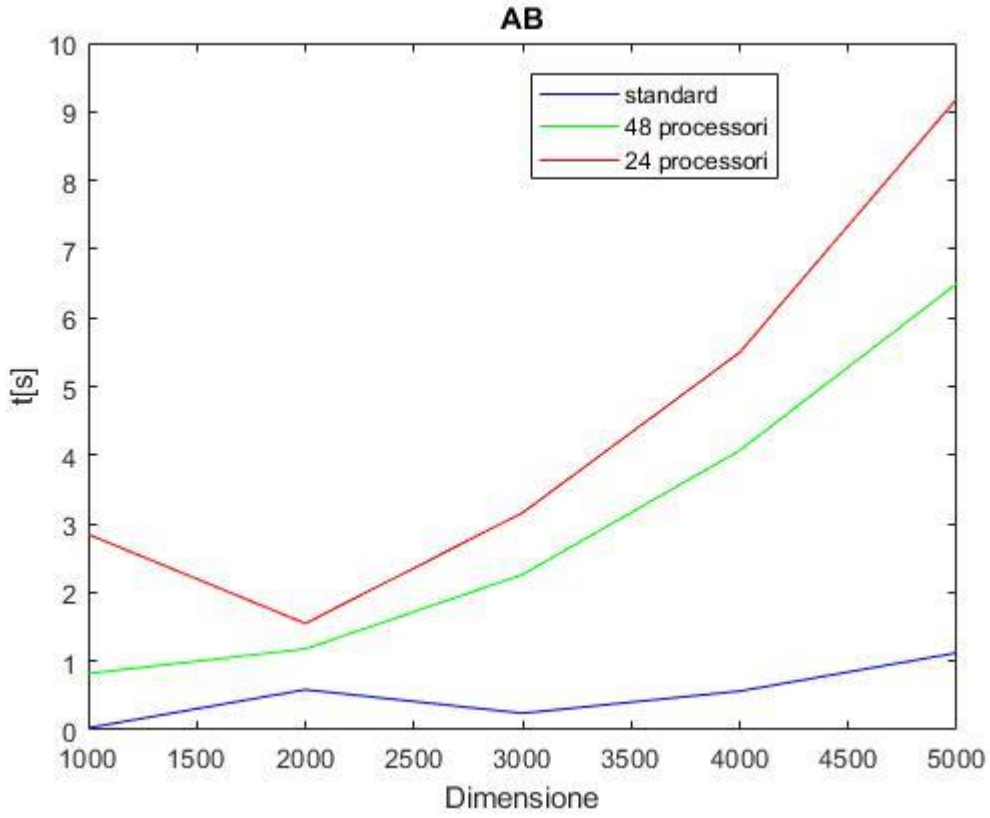


Figura 4.2 Operazione matrice per matrice.

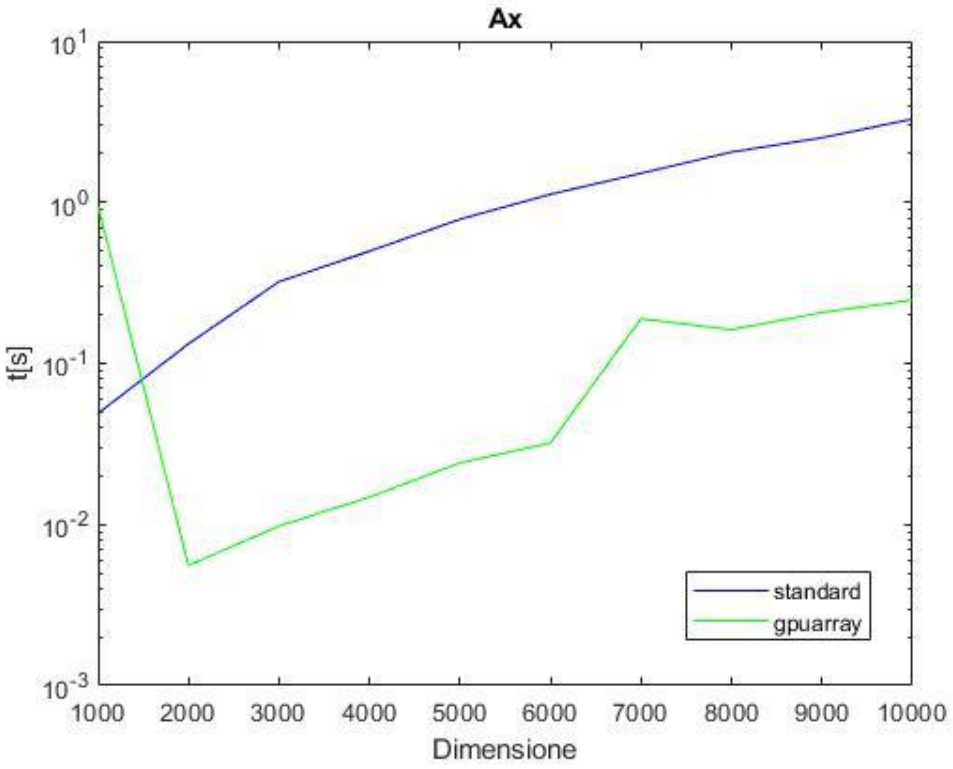


Figura 4.3 Matrice per vettore utilizzando i gpuArray

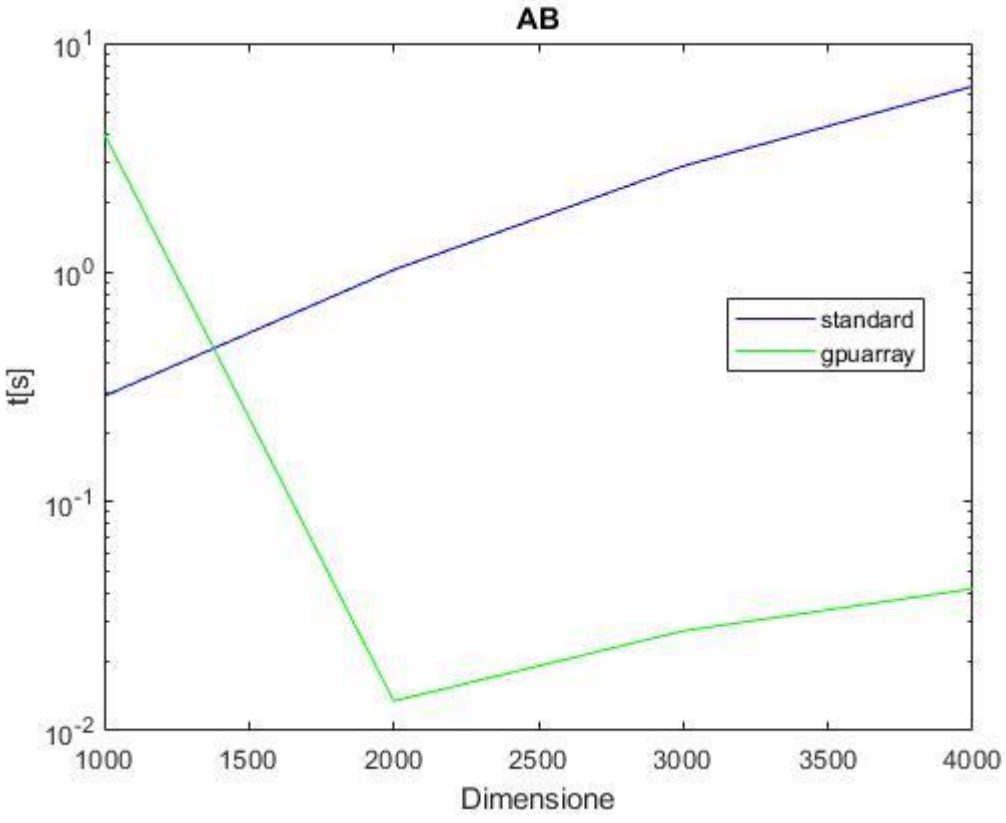


Figura 4.4 Matrice per matrice utilizzando i gpuArray

5. Conclusioni

Dalle simulazioni effettuate si può concludere che l'utilizzo del PCT per la parallelizzazione di calcoli semplici come i prodotti matriciali fa perdere efficienza all'esecuzione, in quanto MATLAB stesso è stato ottimizzato per questo tipo di operazioni, senza ricorrere all'utilizzo dei vari Toolbox. È, quindi, più conveniente lasciare al software la gestione della parallelizzazione.

Si nota invece un netto miglioramento delle prestazioni utilizzando i dati di tipo gpuArray. Questo perché si usa l'algoritmo di MATLAB per la gestione dell'operazione di moltiplicazione, ma usando la GPU per la gestione dei dati.

BIBLIOGRAFIA

[1] Blaise Barney, Lawrence Livermore National Laboratory, Introduction to Parallel Computing, https://computing.llnl.gov/tutorials/parallel_comp/#Whatis

[2] Almerico Murli, Lezioni di Calcolo Parallelo, 2006

[3] A.S. Tanenbaum, T. Austin, Architettura dei calcolatori, un approccio strutturale, sesta edizione, Pearson

[4] David A Patterson, John L. Hennesy, Struttura e progetto dei calcolatori, 2015

[5] Vito Lavecchia, Legge di Amdahl per i calcolatori elettronici, https://vitolavecchia.altervista.org/legge-di-amdahl-calcolatori-elettronici/

[6] Sandro Petrizzelli, Appunti di Calcolatori Elettronici, Capitolo 1, Principi di Progettazione dei Calcolatori, 30 giugno 2001.

[7] Silvano Iacobucci, Calcolo Parallelo e sistemi multicore

[8] Wikipedia, Gene Amdahl, https://en.wikipedia.org/wiki/Gene_Amdahl

[9] Wikipedia, John Leroy Gustafson, en.wikipedia.org/wiki/John_Gustafson_(scientist)

[10] MathWorks, https://it.mathworks.com/products/matlab.html

[11] Plotr Luszczek, Parallel Programming in Matlab, 16 giugno 2009, https://journals.sagepub.com/doi/abs/10.1177/1094342009106194

[12] Mathworks, Parallel Computing Toolbox, Documentation, https://it.mathworks.com/help/distcomp/

[13] Gaurav Sharma, Jos Martin, MATLAB®: A Language for Parallel Computing, 15 Ottobre 2008, https://link.springer.com/article/10.1007/s10766-008-0082-5.

[14] MathWorks, Cleve Moler, Parallel MATLAB: Multiple Processor and Multiple Cores, https://it.mathworks.com/company/newsletters/articles/parallel-matlab-multiple-processors-and-multiple-cores.html..html

[15] Y.M. Altman, Accelerating MATLAB® Performance: 1001 tips to speed up MATLAB programs, CRC Press, 2014.

[16] MathWorks, parpool, https://it.mathworks.com/help/distcomp/parpool.html?searchHighlight=parpool&s_tid=doc_srchtitle

[17] MathWorks, parfor, https://it.mathworks.com/help/distcomp/parfor.html?s_tid=srchtitle

[18] NVIDIA, CUDA, https://www.nvidia.it/object/cuda-parallel-computing-it.html

[19] NVIDIA, CUDA Zone, https://developer.nvidia.com/cuda-zone

[20] NVIDIA, High Performance Computing, https://www.nvidia.com/en-us/high-performance-computing/

[21] NVIDIA, CHI SIAMO, https://www.nvidia.com/it-it/about-nvidia/ai-computing/

[22] Wikipedia, Gordon Moore, https://it.wikipedia.org/wiki/Gordon_Moore

[23] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym, NVIDIA Tesla: A Unified Graphics and Computing Architecture, 2008.
[24] NVIDA, CUDA's GPU https://developer.nvidia.com/cuda-gpus#collapseOne
[25] MathWorks, feval, https://it.mathworks.com/help/distcomp/feval.html?searchHighlight=feval&s_tid=doc_srchtitle
[26] MathWorks, Create GPU Arrays Directly, https://it.mathworks.com/help/distcomp/establish-arrays-on-a-gpu.html#bspvmhe-1
[27] MathWorks, arrayfun, https://it.mathworks.com/help/matlab/ref/arrayfun.html
[28] R. Miller, Q. F. Stout, Parallel algorithms for regular architectures: meshes and pyramids, 1996
[29] NVIDIA, CHI SIAMO, Cronologia di NVIDIA, https://www.nvidia.com/it-it/about-nvidia/corporate-timeline/