



Università degli Studi di Salerno  
Dipartimento di Informatica

---

Corso di Laurea Magistrale in Informatica

Metodi Numerici per l'Informatica  
**Teoria 2025/2026**

**Autore**  
Simone D'Assisi

# Indice

<b>1</b>	<b>Introduzione al Calcolo Parallello</b>	<b>1</b>
1.1	Cenni sul Calcolo Parallello . . . . .	1
1.2	Tipi di Parallelismo . . . . .	2
1.3	Tassonomia di Flynn . . . . .	3
1.4	Principi del Calcolo Parallello . . . . .	4
1.5	Efficienza di un Algoritmo in Ambiente Parallello . . . . .	8
<b>2</b>	<b>Somma in Parallello</b>	<b>10</b>
2.1	Calcolo della Somma Totale . . . . .	11
<b>3</b>	<b>Algoritmi Paralleli per il Prodotto Matrice-Vettore</b>	<b>15</b>
3.1	Strategie per il Calcolo del Prodotto Matrice-Vettore . . . . .	15
<b>4</b>	<b>GP-GPU e l'Ambiente CUDA</b>	<b>22</b>
4.1	GP-GPU . . . . .	22
4.2	Programmare le GPU - CUDA . . . . .	23
4.3	Memorie di una GPU . . . . .	26
<b>5</b>	<b>CUDA - Configurazione Ottimale del Kernel</b>	<b>29</b>
5.1	Griglie e Blocchi in CUDA . . . . .	29
5.2	Configurazione del Kernel . . . . .	30
5.3	Scelta di blockDim . . . . .	30
5.4	CUDA Occupancy Calculator . . . . .	32
5.5	Scelta di blockDim su Architetture Fermi e Kepler . . . . .	32
<b>6</b>	<b>Shared Memory e Prodotto Scalare</b>	<b>34</b>
6.1	Approfondimento Shared Memory . . . . .	34
6.2	Prodotto scalare . . . . .	35
<b>7</b>	<b>Libreria cuBlas</b>	<b>38</b>
7.1	Da BLAS a cuBLAS . . . . .	38
<b>8</b>	<b>Sistemi di Raccomandazione</b>	<b>39</b>
8.1	Big Data . . . . .	39
8.2	Tipi di Sistemi di Raccomandazione . . . . .	40
8.3	Problemi dei Sistemi di Raccomandazione . . . . .	40
8.4	Formalizzazione Matematica del Problema della Raccomandazione . . . . .	41
8.5	Sistemi di Raccomandazione Model-Based . . . . .	43
<b>Lab 1</b>	<b>MPI</b>	<b>50</b>
Lab 1.1	Processi . . . . .	50
Lab 1.2	Contesto . . . . .	50
Lab 1.3	Funzioni di MPI . . . . .	50
<b>Lab 2</b>	<b>Topologie di Processi</b>	<b>56</b>
Lab 2.1	Tipi di Topologie . . . . .	56

Lab 2.2 Sottogriglie . . . . .	57
Lab 2.3 Funzioni . . . . .	58
<b>Lab 3 Compilazione e Valutazione di Programmi CUDA</b>	<b>59</b>
Lab 3.1 Specifiche di Compilazione . . . . .	59
Lab 3.2 Misura dei Tempi: gli Eventi . . . . .	60
Lab 3.3 Ottimizzazione Mediante Profiling . . . . .	61
<b>Lab 4 Riduzione, Sincronizzazione e Allocazione</b>	<b>62</b>
Lab 4.1 Riduzione . . . . .	62
Lab 4.2 Sincronizzazione . . . . .	63
Lab 4.3 Allocazione Statica e Dinamica . . . . .	63
<b>Lab 5 Programmare con cuBLAS</b>	<b>65</b>
Lab 5.1 Funzioni . . . . .	65
Lab 5.2 Gestione degli Errori . . . . .	66
Lab 5.3 Compilazione . . . . .	66
Lab 5.4 Gestione delle Matrici . . . . .	67

# 1 Introduzione al Calcolo Parallello

Il calcolo parallelo viene impiegato per risolvere problemi che risultano computazionalmente troppo onerosi anche per i più veloci supercomputer contemporanei. Tale approccio consente di:

- affrontare problemi di dimensioni maggiori nello stesso intervallo di tempo (*scale-up*);
- risolvere lo stesso problema in un tempo inferiore (*speed-up*);
- superare i vincoli di memoria;
- contenere i costi, utilizzando  $N$  processori economici invece di uno più costoso.

Tuttavia, il calcolo parallelo presenta alcuni limiti di natura hardware e software. È infatti necessario disporre di strutture hardware capaci di supportare la parallelizzazione (ovvero più CPU) e di software in grado di ripartire correttamente le operazioni tra i diversi processori.

## 1.1 Cenni sul Calcolo Parallello

Per diversi decenni la tecnologia informatica ha conosciuto continui progressi, che hanno rappresentato la principale forza trainante nello sviluppo delle architetture degli elaboratori. La potenza di calcolo dei computer è aumentata nel tempo grazie a diversi fattori, come l'incremento della velocità del clock, la realizzazione di chip più efficienti, l'ampliamento della capacità di memoria o l'introduzione di tecniche di ottimizzazione del calcolo, come il *pipelining* e la *vettorizzazione*. Nonostante tali progressi, l'esecuzione di calcoli particolarmente complessi può incontrare diversi limiti, come ad esempio limiti di memoria, di velocità di trasmissione dei dati, di miniaturizzazione dei componenti hardware, o di natura economica. Una possibile soluzione a queste problematiche è rappresentata dall'utilizzo simultaneo di più risorse di calcolo, approccio che può essere implementato o mediante un singolo computer dotato di più processori, o utilizzando più computer collegati in rete, oppure combinando più sistemi multiprocessore connessi tra loro.

Il calcolo parallelo può quindi essere considerato come un'evoluzione naturale del calcolo seriale (o sequenziale). In termini generali, esso consiste nell'impiego simultaneo di più unità di elaborazione (CPU) per risolvere un unico problema computazionale. Per essere eseguito su più CPU, il problema viene suddiviso in parti discrete che possono essere risolte in modo concorrente. Le istruzioni relative a ciascuna parte vengono poi eseguite contemporaneamente da processori diversi, considerando che ogni CPU può svolgere al massimo un'operazione per volta.

### 1.1.1 Confronto con il Calcolo Sequenziale

Il calcolo sequenziale rappresenta il modello computazionale tradizionale, caratterizzato dalla presenza di un singolo processore che esegue le istruzioni di un algoritmo in maniera ordinata e consecutiva. In questo approccio, un problema viene suddiviso in una sequenza di istruzioni discrete, che vengono eseguite una dopo l'altra. In ogni istante di tempo, la CPU è impegnata nell'elaborazione di una sola istruzione, completata la quale passa alla successiva. Questo modello, pur essendo semplice ed efficace per molte applicazioni, presenta

limiti evidenti quando la complessità dei problemi o la quantità di dati da elaborare cresce in modo significativo, rendendo necessario l'impiego di strategie parallele per migliorare le prestazioni computazionali.

Nel calcolo parallelo l'elaborazione non viene più eseguita da un singolo processore, ma da più unità di calcolo che operano simultaneamente. Questo può avvenire su sistemi dotati di più CPU, su processori multicore oppure su CPU con tecnologia *dual thread*, capaci di gestire più flussi di istruzioni in parallelo. Il problema da risolvere viene decomposto in componenti discrete e indipendenti, ognuna delle quali può essere elaborata in modo concorrente da processori diversi. Le istruzioni relative a tali componenti vengono quindi eseguite simultaneamente su CPU differenti, permettendo di ridurre i tempi di calcolo complessivi e di affrontare problemi di maggiore complessità rispetto al modello sequenziale.

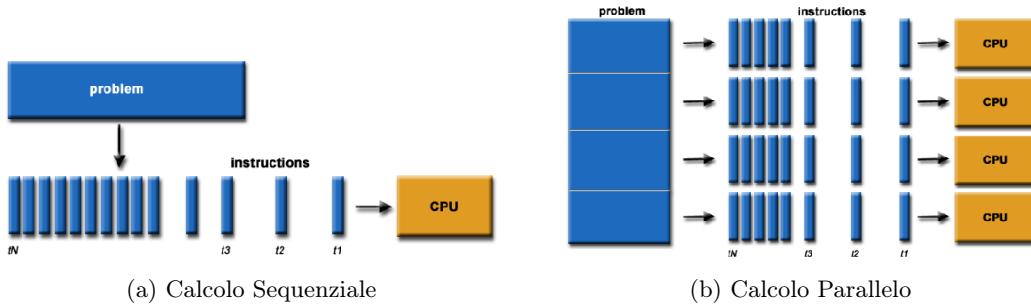


Figura 1: Confronto tra Calcolo Sequenziale e Parallelismo.

## 1.2 Tipi di Parallelismo

Il calcolo parallelo può essere realizzato secondo diversi approcci, a seconda di come vengono distribuite le operazioni e i dati tra le varie unità di elaborazione. I principali tipi di parallelismo sono tre: **Parallelismo temporale** (o *pipeline*), **Parallelismo spaziale**, e **Parallelismo asincrono**.

**Parallelismo Temporale (Pipeline).** Il parallelismo temporale, anche detto *pipelining*, può essere paragonato al funzionamento di una catena di montaggio. In questo modello, l'esecuzione di un'istruzione è suddivisa in più stadi, ognuno dei quali viene gestito da una diversa unità funzionale del processore. La presenza della pipeline consente di aumentare il numero di istruzioni eseguite contemporaneamente, incrementando così il *throughput*, ovvero il numero di istruzioni completate nell'unità di tempo. Tuttavia, la latenza — cioè il tempo necessario per completare una singola istruzione dal suo inizio al suo termine — non diminuisce. È importante che gli stadi della pipeline operino in modo sincrono: lo stadio più lento determina infatti la durata complessiva di ciascun ciclo della pipeline. Un'analogia utile è quella di una lavatrice: anche se più carichi di bucato vengono avviati in sequenza, il primo ciclo richiederà sempre lo stesso tempo per essere completato.

**Parallelismo Spaziale.** Il parallelismo spaziale consiste nell'esecuzione contemporanea della stessa operazione su dati diversi. Questo approccio è tipico dei calcolatori di tipo **SIMD** (Single Instruction, Multiple Data), in cui un'unica istruzione di controllo coordina

l’elaborazione parallela su più unità aritmetico-logiche (ALU). Tale architettura è particolarmente efficiente quando si devono eseguire le stesse operazioni su grandi insiemi di dati, come avviene nel calcolo scientifico o nel trattamento di immagini.

**Parallelismo Asincrono.** Il parallelismo asincrono, invece, si realizza quando diverse CPU eseguono istruzioni differenti su insiemi di dati diversi. Questo modello è tipico dei calcolatori di tipo **MIMD** (Multiple Instruction, Multiple Data), nei quali ciascun processore dispone di una propria unità di controllo (CU) e di una propria unità aritmetico-logica (ALU). In questo caso, i processori lavorano in modo indipendente e asincrono, permettendo una maggiore flessibilità e adattabilità nell’esecuzione di compiti complessi e diversificati.

### 1.3 Tassonomia di Flynn

La tassonomia di Flynn è una classificazione dei modelli di architettura dei calcolatori basata su come vengono gestiti flussi di istruzioni e di dati. Proposta da Michael J. Flynn nel 1966, distingue i sistemi in quattro categorie principali:

- **SISD (Single Instruction, Single Data):** Un solo flusso di istruzioni e un solo flusso di dati. È il modello dei computer sequenziali tradizionali (come le CPU classiche). Esegue un’istruzione alla volta su un solo dato per volta.
- **SIMD (Single Instruction, Multiple Data):** Un’unica istruzione è eseguita simultaneamente su più dati. Tipico delle GPU o dei processori vettoriali, ottimo per operazioni parallele su array o matrici.
- **MISD (Multiple Instruction, Single Data):** Più unità di controllo eseguono istruzioni diverse sullo stesso dato. Raro nella pratica; può apparire in sistemi di ridondanza o fault-tolerance.
- **MIMD (Multiple Instruction, Multiple Data):** Più processori eseguono istruzioni diverse su dati diversi. È il modello dei sistemi multiprocessore o multicore moderni. Ogni processore lavora in modo autonomo e può cooperare con gli altri tramite comunicazione.

<b>S I S D</b> Single Instruction Single Data	<b>S I M D</b> Single Instruction Multiple Data
<b>M I S D</b> Multiple Instructions Single Data	<b>M I M D</b> Multiple Instructions Multiple Data

Figura 2: Rappresentazione della tassonomia di Flynn, che distingue le architetture dei calcolatori in base ai flussi di istruzioni e di dati: SISD, SIMD, MISD e MIMD.

## 1.4 Principi del Calcolo Parallelo

La progettazione di algoritmi paralleli presenta numerose sfide che rendono la programmazione in parallelo più complessa rispetto a quella seriale. Tra i principali aspetti da considerare si trovano:

- la valutazione delle prestazioni di un algoritmo parallelo;
- la ricerca e lo sfruttamento della granularità;
- la preservazione della località dei dati;
- il bilanciamento del carico computazionale;
- il coordinamento e la sincronizzazione tra processori;
- la considerazione dei limiti imposti dalla **legge di Amdahl**.

### 1.4.1 Valutazione delle Prestazioni

L'utilizzo di un numero maggiore di processori non garantisce automaticamente prestazioni migliori. Un algoritmo parallelo è considerato **efficace** solo se riesce a sfruttare in modo equilibrato tutte le risorse della macchina, minimizzando gli overhead e mantenendo un buon bilanciamento del carico computazionale. Di seguito vengono riportate alcune metriche per la valutazione delle prestazioni di un sistema parallelo.

**Speed-Up.** Un parametro fondamentale per valutare l'efficacia di un algoritmo parallelo è lo **speed-up**, definito come:

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

dove:

- $T_s$  è il tempo di esecuzione seriale (in secondi);
- $T_p$  è il tempo di esecuzione parallelo utilizzando  $p$  processori (in secondi).

Idealmente, si ha  $S_p = p$ , cioè uno **speed-up lineare**. In pratica, però, tale condizione è difficilmente raggiungibile a causa degli overhead introdotti dalla comunicazione e sincronizzazione tra i processori.

In alcuni casi si può verificare uno **speed-up superlineare**, ovvero  $S_p > p$ . Questo fenomeno può manifestarsi, ad esempio, quando:

- la gestione della gerarchia di memoria è più efficiente nel sistema parallelo (ad esempio, grazie a una maggiore capacità di cache complessiva);
- il codice parallelo è ottimizzato in modo differente rispetto alla versione sequenziale;
- in problemi di ricerca su grafi, la suddivisione tra processori consente a uno di essi di trovare prima la soluzione.

È possibile distinguere tra:

- **Speed-up assoluto:**

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

- **Speed-up relativo:**

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

dove  $T_1$  è il tempo di esecuzione del codice parallelo su un solo processore.

I due valori possono differire, poiché il codice parallelo eseguito su un singolo processore può introdurre overhead non presenti nella versione strettamente sequenziale, risultando quindi più lento.

**Efficienza.** L'**efficienza** misura quanto efficacemente i processori vengono utilizzati rispetto allo *speed-up* ottenuto. È definita come:

$$E_p = \frac{S_p}{p} = \frac{T_s}{p \cdot T_p}$$

Idealmente,  $E_p = 1$ , ma nella pratica questo valore è sempre inferiore a causa dei costi di comunicazione e sincronizzazione tra i processori.

**Speed-Up Scalato e Scalabilità.** Lo **speed-up scalato** misura la capacità di un algoritmo parallelo di mantenere prestazioni efficienti quando si aumenta simultaneamente il numero di processori e la dimensione del problema. In un comportamento ideale ci si aspetterebbe:

$$SS_p = 1,$$

ovvero un'efficienza costante anche al crescere della potenza di calcolo e del carico di lavoro. L'aggettivo *scalato* indica proprio che, nella misura di questo parametro, si varia sia il numero di processori  $p$  sia la dimensione del problema  $n$ , scalando quest'ultima in modo proporzionale. Lo speed-up scalato è infatti definito come:

$$SS(p, n) = \frac{p T(1, n)}{T(p, pn)}.$$

Nel caso ideale, in cui l'algoritmo mantenga invariata l'efficienza aumentando  $p$  e trattando un problema  $p$  volte più grande, lo speed-up scalato coincide con il numero di processori:

$$SS(p, n)_{\text{ideale}} = p.$$

Assumendo inoltre che il tempo di esecuzione sequenziale su un problema di dimensione  $pn$  sia proporzionale a  $p$  volte il tempo di esecuzione su un problema di dimensione  $n$ , ossia:

$$T(1, pn) = p T(1, n),$$

la formula precedente diventa:

$$SS(p, n) = \frac{T(1, pn)}{T(p, pn)} = S_p(pn).$$

Ne segue che lo *speed-up scalato* può essere interpretato come lo *speed-up classico* applicato a un problema di dimensione  $pn$ , cioè alla versione *scalata* del problema originale. In sintesi, un algoritmo parallelo si dice **scalabile** se è in grado di mantenere un'efficienza approssimativamente costante al crescere sia del numero di processori  $p$  sia della dimensione del problema  $n$ , evitando così un degrado significativo delle prestazioni complessive.

#### 1.4.2 Granularità

La **granularità** di un algoritmo parallelo si riferisce alla dimensione dei compiti in cui il programma viene suddiviso per l'esecuzione simultanea sui vari processori. In generale, si distinguono due approcci principali:

- **Parallelismo a grana fine:** in questo caso, il programma è suddiviso in un gran numero di piccoli compiti, ciascuno assegnato a un processore. Il parallelismo a grana fine facilita il bilanciamento del carico, poiché ogni processore ha compiti di dimensioni simili da eseguire. Tuttavia, per sfruttare efficacemente tutti i processori è necessario un elevato numero di unità di calcolo, il che può aumentare significativamente l'*overhead* dovuto alla comunicazione e alla sincronizzazione tra processori.
- **Parallelismo a grana grossa:** in questo approccio, il programma è suddiviso in compiti di dimensioni maggiori, distribuiti ai vari processori. Il vantaggio principale è la riduzione del numero di comunicazioni e delle fasi di sincronizzazione. D'altro canto, possono verificarsi squilibri nel carico di lavoro tra processori o parti del programma potrebbero essere eseguite in modo sequenziale, riducendo l'efficienza complessiva dell'algoritmo parallelo.

**Overhead.** L'**overhead** rappresenta il tempo aggiuntivo richiesto per la gestione del calcolo parallelo e aumenta al crescere del numero di processi. Se la quantità di computazione parallela è consistente, l'overhead costituisce spesso la principale barriera per ottenere valori ottimali di *speed-up*. Gli elementi principali che contribuiscono all'overhead di un algoritmo parallelo includono:

- il costo di avvio (*starting*) di un processo o thread;
- il costo della comunicazione di dati condivisi tra processori;
- il costo della sincronizzazione tra processori;
- la computazione extra, cioè operazioni ridondanti necessarie per coordinare il calcolo.

In alcuni sistemi, ciascuno di questi fattori può raggiungere valori dell'ordine di millisecondi, equivalenti a milioni di operazioni in virgola mobile (*flops*).

Per ottenere un'esecuzione parallela efficiente, l'algoritmo deve essere progettato con unità di lavoro sufficientemente grandi (*granularità elevata*) da giustificare il costo dell'overhead, ma non così grandi da ridurre la quantità di lavoro parallelo disponibile. In altre parole, occorre bilanciare la dimensione dei compiti per massimizzare sia l'efficienza che il *throughput*.

### 1.4.3 Accessi in Memoria

Nelle architetture parallele, le memorie di grande capacità tendono ad essere lente, mentre le memorie veloci hanno capacità ridotte. Per questo motivo, le gerarchie di memoria cercano un compromesso tra dimensione e velocità. Nei sistemi con processori paralleli, l'insieme delle memorie locali è generalmente grande e veloce, ma l'accesso a dati remoti, che richiede comunicazione tra processori, risulta molto più lento. Per massimizzare le prestazioni, un algoritmo parallelo dovrebbe eseguire la maggior parte delle istruzioni sui dati locali, riducendo al minimo gli accessi a memoria remota.

### 1.4.4 Carico Non Bilanciato

Il **cattivo bilanciamento del carico** si verifica quando alcuni processori restano inattivi perché non hanno operazioni da eseguire. Ciò può accadere per diversi motivi:

- insufficiente parallelismo in una determinata fase del programma;
- compiti di dimensione diversa, ad esempio in problemi con struttura non omogenea.

Per ridurre questo problema, gli algoritmi paralleli devono implementare tecniche di bilanciamento del carico. Tuttavia, tali tecniche possono ridurre la località dei dati, introducendo ulteriori costi di comunicazione e sincronizzazione.

### 1.4.5 Legge di Amdahl

La **legge di Amdahl** descrive il limite massimo di miglioramento delle prestazioni che può essere ottenuto mediante il parallelismo, a causa della frazione di calcolo che deve necessariamente essere eseguita in modo sequenziale.

Sia  $\alpha$  la frazione seriale di un algoritmo che non può essere parallelizzata, ad esempio: inizializzazione dei cicli, operazioni di lettura/scrittura su memoria o overhead delle chiamate a funzioni. In questo caso, il tempo di esecuzione parallelo su  $p$  processori è dato da:

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

dove  $T_s$  è il tempo di esecuzione seriale. Se si tiene conto anche dell'*overhead* introdotto dalla comunicazione e sincronizzazione tra processori, il tempo di esecuzione parallelo diventa:

$$T_p = \alpha T_s + \frac{(1 - \alpha)T_s}{p} + T_{\text{overhead}}$$

Lo *speed-up* massimo ottenibile, senza considerare l'overhead, è quindi:

$$S_p = \frac{T_s}{T_p} = \frac{1}{\alpha + \frac{1-\alpha}{p}} \leq \frac{1}{\alpha}$$

Ad esempio, se  $\alpha = 0.1$  (10% del programma è sequenziale), il massimo speed-up teorico è 10, anche utilizzando un numero molto grande di processori. In altre parole, se la dimensione del problema  $n$  resta fissata, l'aumento del numero di processori  $p$  non permette di superare questa limitazione. Se invece il numero di processori  $p$  è fisso e la dimensione del problema

$n$  aumenta, la frazione sequenziale  $\alpha \rightarrow 0$ , e quindi lo speed-up può avvicinarsi ai valori ottimali:

$$S_p \rightarrow p \quad \text{per} \quad \alpha \rightarrow 0$$

In pratica, aumentando la dimensione del problema si possono ottenere speed-up molto buoni, ma le risorse hardware rimangono comunque limitate, quindi non è possibile scalare indefinitamente: l'ambiente di calcolo parallelo viene sfruttato in modo efficiente finché il numero di processori non supera un valore critico  $p_m$ , oltre il quale le prestazioni peggiorano. Analogamente, aumentando la dimensione del problema oltre un valore massimo  $n_{\max}$ , possono insorgere vincoli di memoria e gestione delle risorse.

## 1.5 Efficienza di un Algoritmo in Ambiente Parallelo

Per un algoritmo sequenziale, l'efficienza è tipicamente valutata tramite:

- **Complessità di tempo**  $T(n)$ : numero di operazioni eseguite dall'algoritmo;
- **Complessità di spazio**  $S(n)$ : numero di variabili utilizzate dall'algoritmo.

In questo contesto, il numero complessivo di operazioni determina anche il numero di passi temporali necessari all'esecuzione, e quindi il tempo totale di esecuzione dell'algoritmo.

Nei calcolatori paralleli, la situazione cambia: più operazioni possono essere eseguite contemporaneamente nello stesso passo temporale. Ad esempio, nell'algoritmo parallelo per il calcolo della somma di  $N$  numeri, il numero totale di operazioni non è direttamente correlato al numero di passi di esecuzione. Di conseguenza, il tempo di esecuzione non dipende unicamente dalla complessità temporale  $T(n)$  (ovvero dal numero totale di operazioni in virgola mobile), e la complessità di tempo tradizionale non è più adatta a misurare l'efficienza di un algoritmo parallelo.

Un approccio comune per valutare un algoritmo parallelo è misurare di quanto si riduce il tempo di esecuzione su  $p$  processori rispetto al tempo di esecuzione su un singolo processore. Questo concetto è formalizzato dallo **speed-up**. Tuttavia, l'utilizzo di un numero maggiore di processori **non garantisce automaticamente prestazioni ottimali**. Uno speed-up elevato non implica necessariamente che l'algoritmo parallelo sia efficace, ovvero che sfrutti tutte le risorse della macchina parallela in modo equilibrato. Per questo motivo, lo speed-up da solo non fornisce informazioni sufficienti sull'efficienza complessiva dell'algoritmo parallelo; occorre considerare anche altri fattori, come l'efficienza.

**Efficienza.** Per valutare quanto un algoritmo sfrutta effettivamente il parallelismo di un calcolatore, si utilizza il concetto di **efficienza**:

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

dove  $S_p$  è lo *speed-up* su  $p$  processori. L'efficienza ideale è pari a 1, ovvero:

$$E_p^{\text{ideale}} = 1 \quad \text{o, equivalentemente, } S_p^{\text{ideale}} = p.$$

### 1.5.1 Speed-Up Ideale

Il tempo di esecuzione sequenziale  $T_s$  può essere scomposto in due componenti:

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

dove:

- $\alpha T_s$  rappresenta la frazione di codice che deve essere eseguita in modo strettamente sequenziale;
- $(1 - \alpha)T_s$  rappresenta invece la parte parallelizzabile.

Con  $p$  processori, le operazioni parallelizzabili vengono distribuite tra i processori, per cui il tempo di esecuzione parallelo è:

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

Da questa relazione si ottiene lo *speed-up* secondo la legge di Amdahl:

$$S_p = \frac{T_s}{T_p} = \frac{1}{\alpha + \frac{1-\alpha}{p}} \leq \frac{1}{\alpha}.$$

## 2 Somma in Parallello

Consideriamo il problema del calcolo della somma di  $N$  numeri:

$$S = a_0 + a_1 + \cdots + a_{N-1}.$$

Su un calcolatore monoprocesso, la somma viene eseguita sequenzialmente. In altre parole, la somma viene calcolata eseguendo  $N - 1$  addizioni, una alla volta, secondo un ordine prestabilito:

$$S = (((a_0 + a_1) + a_2) + \cdots + a_{N-1}).$$

Questo esempio evidenzia chiaramente come, nel calcolo sequenziale, tutte le operazioni devono essere completate una dopo l'altra, senza possibilità di parallelismo.

Nel calcolo parallelo, il problema della somma di  $N$  numeri può essere suddiviso tra  $P$  processori in modo da ridurre il tempo di esecuzione complessivo. Il vettore di numeri di dimensione  $N$  viene scomposto in  $P$  sottovettori di dimensione  $N/P$ . Ogni processore calcola contemporaneamente la somma dei numeri del proprio sottovettore, generando le **somme parziali**:

$$S_0 = a_0 + \cdots + a_{N/P-1}, \quad S_1 = a_{N/P} + \cdots + a_{2N/P-1}, \quad \dots$$

Una volta calcolate le somme parziali, esse devono essere combinate in modo opportuno per ottenere la somma totale:

$$S_{\text{totale}} = S_0 + S_1 + \cdots + S_{P-1}.$$

Questo approccio permette di sfruttare il parallelismo, riducendo il numero totale di operazioni eseguite sequenzialmente e migliorando significativamente il tempo di esecuzione rispetto al modello monoprocesso.

Esempio:  $N=16, p=4$

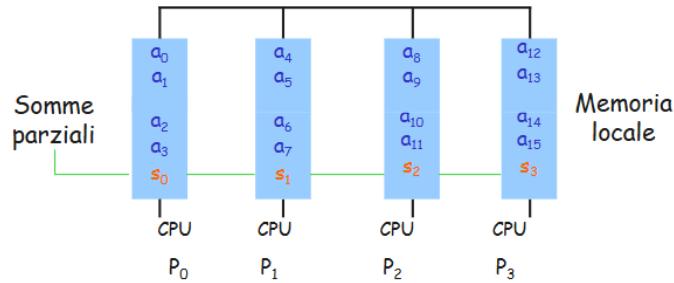


Figura 3: Gli addendi vengono distribuiti tra tutti i processori, che calcolano le somme parziali.

**Valutazione.** L'algoritmo sequenziale per la somma prevede quindi l'esecuzione di  $N - 1$  somme. Il tempo impiegato sarà quindi:

$$T_s = (N - 1)t_{\text{calc}}$$

dove  $t_{calc}$  è il tempo di esecuzione di una singola operazione floating point (come, ad esempio, una somma).

## 2.1 Calcolo della Somma Totale

### 2.1.1 Strategia I

Ogni processore calcola la propria somma parziale. Dopodiché, ad ogni passo, ciascun processore invia tale valore ad un unico processore prestabilito. Tale processore contiene la somma totale.

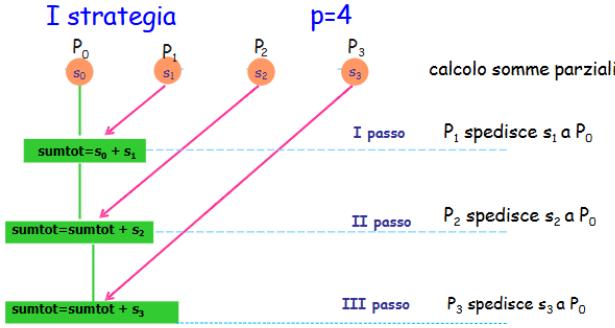


Figura 4: Con  $N = 16$  e  $p = 4$  vengono eseguiti 3 passi.

Durante l'implementazione della Strategia I, distinguiamo tre fasi distinte:

1. Distribuzione dei dati: i dati, ricevuti dal primo processore, devono essere ripartiti tra tutti i processori utilizzati. Distinguiamo due casi distinti:
  - Il numero di operazioni  $N$  (in questo caso il numero di elementi da sommare) è divisibile per il numero di processori  $p$ . In questo caso, tutti i dati vengono partizionati senza intersezione, ed ogni processore eseguirà  $nloc = \frac{N}{p}$  operazioni.
  - Il numero di operazioni  $N$  non è divisibile per il numero di processori  $p$ . In questo caso, oltre al quoziente  $nlocgen = \frac{N}{p}$  si terrà in considerazione anche  $r = resto(N, p)$ . A ciascun processore saranno assegnati  $nlocgen$  elementi, mentre ai primi  $r$  processori sarà assegnato un operando in più.
2. Calcolo delle somme parziali: ogni processore calcola la somma dei propri addendi, impiegando tempo  $\frac{N}{p}$ . Il tempo di esecuzione delle somme parziali sarà:

$$\left( \frac{N}{p} - 1 \right) t_{calc}$$

3. Scambio dei dati: il numero dei passi non dipende dal numero delle operazioni, ma dal numero dei processori  $-1$ . Ad ogni passo, la somma parziale viene inviata a  $P_0$ . Inoltre, non bisogna trascurare il tempo di comunicazione tra due processori  $t_{com}$ . Questa fase richiede quindi tempo:

$$(p - 1)t_{calc} + (p - 1)t_{com} = (p - 1)(t_{calc} + t_{com})$$

**Valutazione.** Il tempo totale richiesto dalla strategia I è quindi pari a:

$$T_p = \left( \frac{N}{p} - 1 \right) t_{calc} + (p-1)t_{calc} + (p-1)t_{com} = \left[ \frac{N}{p} - 1 + (p-1) \right] t_{calc} + (p-1)t_{com}$$

Per un valore di  $N$  molto grande, il peso della comunicazione sarà trascurabile.

**Comportamento Asintotico.** Analizzando il comportamento dello *speed-up* e dell'efficienza al variare del numero di processori  $p$  e della dimensione del problema  $n$  si ottengono le seguenti considerazioni:

- **Caso  $p \rightarrow \infty$ :** se la dimensione  $n$  del problema è fissata, all'aumentare di  $p$  si ha:

$$\lim_{p \rightarrow \infty} S_p = \frac{T_s}{T_p} = \frac{N}{\frac{N}{p} + (p-1)} = \frac{1}{\frac{1}{p} + \frac{p-1}{N}} = 0, \quad \lim_{p \rightarrow \infty} E_p = \frac{S_p}{p} = 0.$$

- **Caso  $n \rightarrow \infty$ :** se  $p$  è fissato, all'aumentare della dimensione del problema la frazione sequenziale  $\alpha$  tende a zero, e lo *speed-up* tende al valore ideale:

$$\lim_{n \rightarrow \infty} S_p = \frac{1}{\frac{1}{p} + \frac{p-1}{N}} = p, \quad \lim_{n \rightarrow \infty} E_p = \frac{S_p}{p} = 1.$$

Poiché  $N$  ha dimensioni elevate, possiamo permetterci di trascurare  $-1$ . In sintesi, fissato il numero di processori  $p$ , incrementare la dimensione del problema consente di ottenere speed-up prossimi a quello ideale, sebbene esistano limiti pratici dovuti alle risorse hardware disponibili, mentre incrementare indefinitamente  $p$  non porta miglioramenti (legge di Amdahl).

### 2.1.2 Strategia II

Ogni processore calcola la propria somma parziale. Dopodiché, ad ogni passo coppie distinte di processori comunicano contemporaneamente: in ogni coppia, un processore invia all'altro la propria somma parziale, che provvede all'aggiornamento della somma. Il risultato finale sarà presente in un unico processore prestabilito, generalmente il *root* (ossia  $p_0$ ).

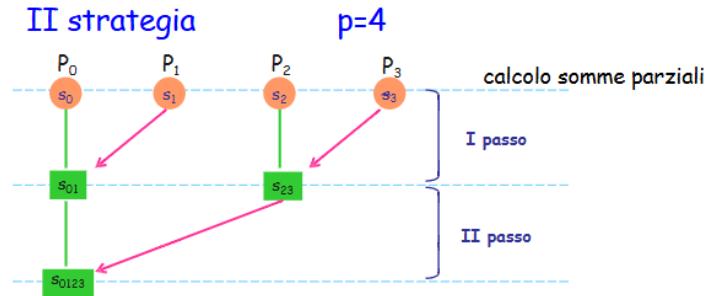


Figura 5: Esempio con  $N = 16$  e  $p = 4$ : vengono eseguiti due passi.

La principale differenza tra la Strategia I e la Strategia II risiede proprio nella comunicazione e nel numero di processori che partecipano all'esecuzione delle somme parziali. Mentre nel

primo caso tutte le somme parziali vengono inviate al processore  $p_0$ , nel secondo caso: dati DIST = distanza tra due processori, menum = ID del processore  $(0, 1, \dots, p - 1)$ ,  $k$  = passi  $(0, \dots, \log_2(p - 1))$ , valgono le seguenti regole:

- se  $\text{resto}(\text{menum}, 2^{k+1}) = 0$ , allora il processore  $\text{menum}$  riceve dal processore  $\text{menum} + \text{DIST}$  ed esegue la somma;
- se  $\text{resto}(\text{menum}, 2^{k+1}) = 2^k$ , allora il processore  $\text{menum}$  invia la propria somma parziale al processore  $\text{menum} - \text{DIST}$ .

Così facendo, vengono utilizzati più processori contemporaneamente. Il numero totale di passi eseguiti sarà pari a  $\log_2 p$  e il tempo richiesto sarà pari a:

$$(\log_2 p)t_{calc} + (\log_2 p)t_{com} = (\log_2 p)(t_{calc} + t_{com})$$

```

1 for k = 0, ..., log2(p - 1)
2   DIST = 2^k
3
4   if resto(menum, 2^(k + 1)) == 0
5     ricevo da menum + DIST
6     sommo
7   else if resto(menum, 2^(k + 1)) == 2^k
8     invio a menum - DIST

```

Codice 1: Pseudocodice della Strategia II.

**Valutazione.** Il tempo di esecuzione delle somme è lo stesso rispetto alla strategia I. Ciò che cambia è il tempo con cui vengono effettuate le comunicazioni tra i processori e i risultati parziali:

$$T_p = \left( \frac{N}{p} - 1 + \log_2 p \right) t_{calc} + \log_2 p \cdot t_{com}$$

**Comportamento Asintotico.** Analizzando il comportamento dello *speed-up* e dell'efficienza al variare del numero di processori  $p$  e della dimensione del problema  $n$  si ottengono le seguenti considerazioni:

- **Caso  $p \rightarrow \infty$ :**

$$\lim_{p \rightarrow \infty} S_p = \frac{T_s}{T_p} = \frac{N}{\frac{N}{p} + \log_2 p} = \frac{1}{\frac{1}{p} + \frac{\log_2 p}{N}} = 0, \quad \lim_{p \rightarrow \infty} E_p = \frac{S_p}{p} = 0.$$

- **Caso  $n \rightarrow \infty$ :**

$$\lim_{n \rightarrow \infty} S_p = \frac{1}{\frac{1}{p} + \frac{\log_2 p}{N}} = p, \quad \lim_{n \rightarrow \infty} E_p = \frac{S_p}{p} = 1.$$

Poiché  $N$  ha dimensioni elevate, possiamo permetterci di trascurare  $-1$ . Anche in questo caso, aumentando le dimensioni del problema  $n$  otteniamo valori di speed-up ed efficienza ideali.

### 2.1.3 Strategia III

Ogni processore calcola la propria somma parziale. Dopodiché, ad ogni passo, coppie distinte di processori comunicano contemporaneamente: in ogni coppia i processori si scambiano reciprocamente le proprie somme parziali. Il risultato finale sarà disponibile in tutti i processori.

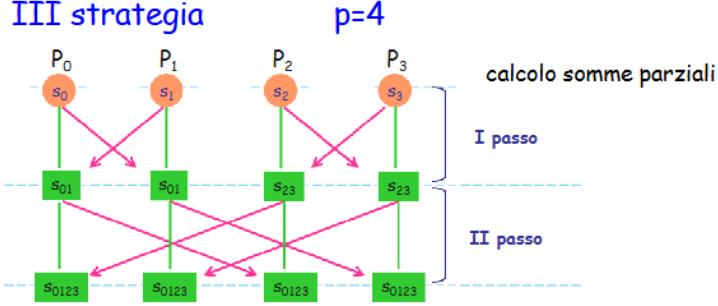


Figura 6: Esempio con  $N = 16$  e  $p = 4$ : vengono eseguiti due passi.

Il numero di passi eseguiti dalla Strategia III è uguale a quello della Strategia II. Anche in questo caso, la differenza risiede nel numero di processori effettivamente impiegati, poiché nessun processore rimane inattivo. Valgono le seguenti regole:

- se  $\text{resto}(\text{menum}, 2^{k+1}) = 0$ , allora il processore  $\text{menum}$  invia e riceve dal processore  $\text{menum} + \text{DIST}$ ;
- se  $\text{resto}(\text{menum}, 2^{k+1}) < 2^k$ , allora il processore  $\text{menum}$  invia e riceve dal processore  $\text{menum} - \text{DIST}$ .

La somma viene eseguita da ciascun processore al termine di ogni comunicazione.

```

1 for k = 0, ..., log2(p - 1)
2   DIST = 2^k
3
4   if resto(menum, 2^(k + 1)) < 2^k
5     invio e ricevo da menum + DIST
6   else
7     invio e ricevo da menum - DIST
8
9   sommo

```

Codice 2: Pseudocodice della Strategia III.

**Valutazione.** Il tempo richiesto dalla strategia III per la somma è lo stesso della strategia II.

**Comportamento Asintotico.** Per la terza strategia sono valide le stesse considerazioni effettuate per la strategia II, in quanto il tempo di esecuzione  $T_s$  è lo stesso.

### 3 Algoritmi Paralleli per il Prodotto Matrice-Vettore

Data una matrice densa  $A$  e un vettore  $x$ , l'obiettivo è progettare un algoritmo parallelo per architettura MIMD a memoria distribuita per calcolare il prodotto:

$$y = Ax, \quad A \in \mathbb{R}^{m \times n}, \quad x \in \mathbb{R}^n, \quad y \in \mathbb{R}^m$$

Per definizione, il generico elemento  $y_i$  è il prodotto scalare dell'i-esima riga di  $A$  per il vettore  $x$ .

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

L'idea di base si articola in quattro fasi:

1. Decomposizione del problema del prodotto matrice-vettore: come mostrato nella figura 7, esistono tre modi diretti di decomposizione di una matrice densa di dimensione  $m \times n$ , cioè per righe, per colonne o a blocchi;
2. Partizionamento della matrice  $A$  “in blocchi”;
3. Riformulazione dell'algoritmo sequenziale a blocchi;
4. Parallelismo dell'algoritmo a blocchi.

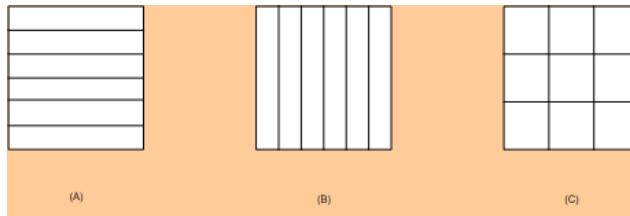


Figura 7: Metodi di decomposizione di una matrice: per righe (A), per colonne (B) o a blocchi (C).

#### 3.1 Strategie per il Calcolo del Prodotto Matrice-Vettore

##### 3.1.1 Algoritmo Seriale

L'algoritmo sequenziale più naturale prevede il calcolo del vettore  $y$  componente per componente, effettuando i prodotti scalari di ciascuna riga di  $A$  per il vettore  $x$ .

```

1 // ciclo for sulle righe
2 for i = 1 to m
3     yi = 0
4     // ciclo for sulle colonne
5     for j = 1 to n
6         yi = yi + aij * xj

```

Codice 3: Pseudocodice dell'algoritmo seriale.

L'algoritmo sequenziale richiede  $m \cdot n$  moltiplicazioni e  $m \cdot n$  addizioni. Il tempo di esecuzione è:

$$T_s = 2mn t_{calc}$$

Organizzando diversamente l'algoritmo, sono richieste  $m \cdot n$  moltiplicazioni e  $m(n - 1)$  addizioni.

### 3.1.2 Strategia I: Distribuzione per Blocchi di Righe

La matrice  $A$  viene distribuita in blocchi di righe fra i processori, mentre il vettore  $x$  è fornito interamente a tutti i processori. Il vettore prodotto finale  $y$  viene calcolato in parallelo, in blocchi distribuiti tra i processori.

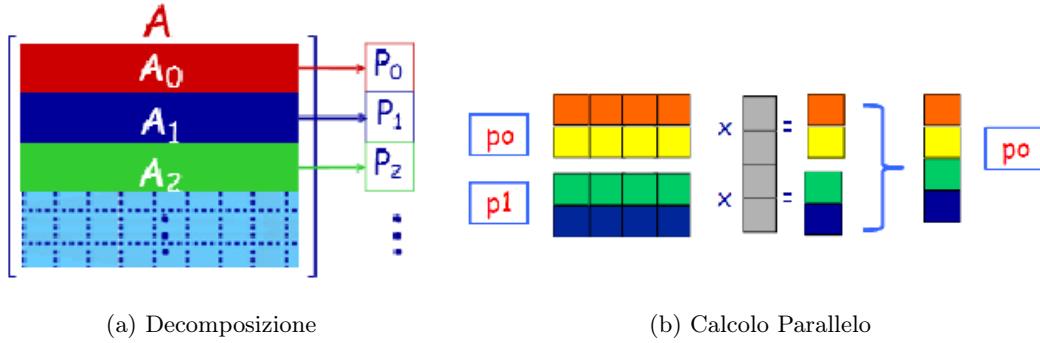


Figura 8: Strategia I di decomposizione della matrice e distribuzione tra i processori per  $m = 4, n = 4, p = 2$ .

L'algoritmo che implementa la Strategia I effettuerà le seguenti operazioni (supponendo che  $m$  sia multiplo del numero di processori  $p$ ).

1. Inizializzazione/acquisizione dei dati: il processo *root* (con identificativo  $me = 0$ ) inizializza (oppure prende in input) la matrice  $A$  ed il vettore  $x$ . Il processo *root* calcola la dimensione locale  $local\_m = \frac{m}{p}$ .
2. Distribuzione dei dati: il processo *root* distribuisce a tutti i processi:
  - $m, n$  e  $local\_m$  tramite *broadcast*;
  - La sottomatrice  $local\_A$  della matrice  $A$  (*scatter* per righe), se stesso incluso;
  - l'intero vettore  $x$  tramite *broadcast*.
3. Calcolo dei prodotti parziali: ciascun processo calcola il prodotto parziale  $local\_y = local\_A \times x$
4. Combinazione dei risultati: il processo *root* raccoglie gli elementi  $local\_y$  calcolati da ogni processo nel vettore risultato  $y$  tramite *gather*.
5. Stampa del risultato finale tramite il processo *root*.

**Valutazione.** Sia  $p$  il numero di processori, con  $p \leq m$ . Ogni processo esegue un prodotto matrice-vettore tra una matrice di dimensione  $\left(\frac{m}{p} \times n\right)$  e un vettore di lunghezza  $n$ . Quindi:

$$T_p = 2 \frac{mn}{p} t_{calc}$$

È da specificare che le fasi di comunicazione riguardano unicamente la distribuzione iniziale dei dati e la raccolta finale dei risultati, quindi non sono state considerate nella valutazione dell'algoritmo.

La Strategia I ha inoltre due varianti, delle quali la prima prevede la distribuzione ciclica delle righe di  $A$ , mentre la seconda la distribuzione ciclica a blocchi di righe di  $A$ .

**Comportamento Asintotico.** Se  $\alpha$  è la frazione di calcolo seriale:

$$T_1 = \alpha T_s + (1 - \alpha)T_s, \text{ e in generale } T_p = \alpha T_s + \frac{(1 - \alpha)}{p} T_s$$

Ma in questo caso  $\alpha = 0$  poiché ogni processo lavora su una porzione indipendente della matrice, non essendoci dipendenze tra i dati che obblighino i processi a lavorare uno dopo l'altro. Quindi:

$$S_p = \frac{T_s}{T_p} = \frac{2mn}{\frac{2mn}{p}} = p, \quad E_p = \frac{S_p}{p} = 1$$

Possiamo dedurne che in questo caso speed-up ed efficienza sono ottimali.

### 3.1.3 Strategia II: Distribuzione per Blocchi di Colonne

Se la decomposizione dei dati nel prodotto matrice-vettore avviene mediante un partizionamento in “blocchi” di colonne della matrice, il vettore prodotto finale  $y$  viene ricondotto al calcolo in parallelo dei contributi (alla somma) del vettore  $y$  distribuiti fra i processori.

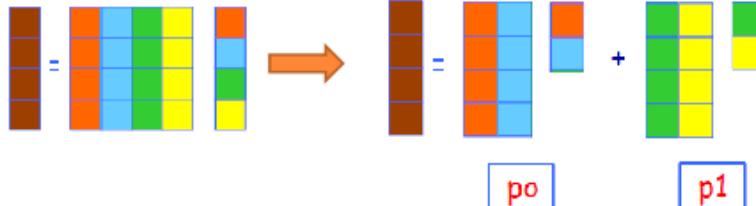


Figura 9: Strategia II di decomposizione della matrice e distribuzione tra i processori per  $m = 4, n = 4, p = 2$ .

L'algoritmo che implementa la Strategia I effettuerà le seguenti operazioni (supponendo che  $m$  sia multiplo del numero di processori  $p$ ).

1. Inizializzazione/acquisizione dei dati: il processo *root* (con identificativo  $me = 0$ ) inizializza (oppure prende in input) la matrice  $A$  ed il vettore  $x$ . Il processo *root* calcola la dimensione locale  $local\_n = \frac{n}{p}$ .
2. Distribuzione dei dati: il processo *root* distribuisce a tutti i processi:
  - $m, n$  e  $local\_n$  tramite *broadcast*;
  - La sottomatrice  $local\_A$  della matrice  $A$  (*scatter* per righe), se stesso incluso;
  - il sottovettore  $local\_x$  del vettore  $x$  tramite *scatter*.
3. Calcolo dei prodotti parziali: ciascun processo calcola il prodotto parziale  $local\_y = local\_A \times local\_x$

4. Combinazione dei risultati: Mediante un'operazione di *riduzione*, vengono sommati in parallelo tutti i vettori *local\_y*, ottenendo il vettore risultato *y*. Il vettore *y* sarà poi salvato dal processo *root* (o da tutti i processi).

5. Stampa del risultato finale tramite il processo *root*.

**Valutazione.** Ogni processo esegue un prodotto matrice-vettore tra una matrice  $\frac{m \cdot n}{p}$  e un vettore di lunghezza  $\frac{n}{p}$ , per un tempo pari a:

$$T_p = \frac{2mn}{p} t_{calc}$$

La somma in parallelo di  $p$  vettori di dimensione  $m$  può richiedere tempi diversi a seconda della strategia utilizzata:

- La Strategia I per la somma in parallelo richiede tempo  $T_s = m(p - 1)t_{calc}$ . Quindi, in totale il tempo richiesto risulterebbe essere pari a:

$$T_p = \left[ \frac{2mn}{p} + m(p - 1) \right] t_{calc}$$

Se consideriamo anche i tempi di comunicazione, avremo:

$$T_p = \left[ \frac{2mn}{p} + m(p - 1) \right] t_{calc} + m(p - 1)t_{com}$$

- La Strategia II e la Strategia III per la somma in parallelo richiedono entrambi tempo  $T_s = m(\log_2 p)t_{calc}$ . Quindi, in totale il tempo richiesto risulterebbe essere pari a:

$$T_p = \left( \frac{2mn}{p} + m \log_2 p \right) t_{calc}$$

Se consideriamo anche i tempi di comunicazione, avremo:

$$T_p = \left( \frac{2mn}{p} + m \log_2 p \right) t_{calc} + m(\log_2 p)t_{com}$$

**Comportamento Asintotico.** Ignorando le comunicazioni, utilizzando la Strategia I per la somma:

$$S_p = \frac{1}{\frac{1}{p} + \frac{p-1}{2n}}, \quad E_p = \frac{1}{1 + \frac{p^2-p}{2n}}$$

Mentre con la Strategia II o III per la somma:

$$S_p = \frac{1}{\frac{1}{p} + \frac{\log_2 p}{2n}}, \quad E_p = \frac{1}{1 + \frac{p \log_2 p}{2n}}$$

Per entrambe le strategie, si ha:

- per  $p$  fissato e  $n \rightarrow \infty$ :

$$\lim_{n \rightarrow \infty} S_p = p, \quad \lim_{n \rightarrow \infty} E_p = 1$$

- per  $n$  fissato e  $p \rightarrow \infty$ :

$$\lim_{p \rightarrow \infty} S_p = 0, \quad \lim_{p \rightarrow \infty} E_p = 0$$

### 3.1.4 Strategia III: Distribuzione per Blocchi di Righe e Colonne

Una terza modalità di decomposizione della matrice  $A$  consiste nel suddividerla in blocchi bidimensionali di righe e colonne. I processi vengono organizzati in una topologia cartesiana (Sezione Lab 2), ovvero una griglia di  $p \times q$  processi. A ciascun processo  $P_{ij}$  viene assegnata una sottomatrice  $A_{ij} = \frac{m}{p} \times \frac{n}{q}$  e il corrispondente sottovettore  $x_j = \frac{n}{q}$ , assumendo che  $m$  sia multiplo di  $p$  e  $n$  sia multiplo di  $q$ . Ogni processo calcola quindi un vettore parziale  $y_{ij} = \frac{m}{p}$ , risultato del prodotto tra la propria sottomatrice e il sottovettore ricevuto. Infine, i contributi  $y_{ij}$  vengono combinati tra i processi della stessa riga della topologia per ottenere il vettore  $y_i$ , da cui si ricostruisce il risultato completo.

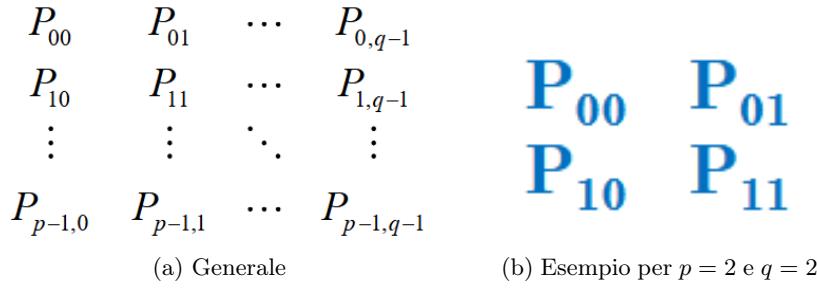


Figura 10: Topologia Cartesiana 2D di processi.

L'algoritmo che implementa la Strategia III effettuerà le seguenti operazioni:

1. Inizializzazione/acquisizione dei dati: il processo *root* (con identificativo  $(0, 0)$ ) inizializza o legge da file) la matrice  $A$  ed il vettore  $x$ .
2. Distribuzione dei dati: il processo *root*  $P_{00}$  distribuisce:
  - $m, n$  a tutti i processi tramite *broadcast*;
  - La sottomatrice *local\_A* della matrice  $A$ ;
  - Il sottovettore *local\_x*.

I dettagli della distribuzione saranno discussi in seguito;

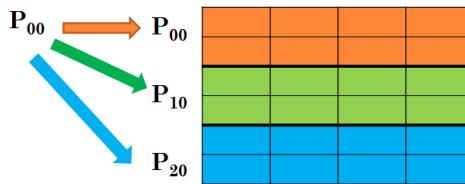
3. Calcolo dei prodotti parziali: ciascun processo calcola un vettore *local\_y* = *local\_A* × *local\_x*;
4. Combinazione dei risultati: Mediante un'operazione di **riduzione**, su ogni riga  $i$  di processi, vengono sommati in parallelo tutti i vettori  $r_{ij} = A_{ij}x_j$ ,  $j = 0, \dots, q - 1$ , ottenendo il vettore risultato  $y[i]$ , salvato da processo  $P_{i0}$  (o da tutti i processi della riga). Il processo *root*  $P_{00}$  raccoglie gli elementi  $y[i]$  calcolati da ogni processo della prima colonna nel vettore risultato  $y$  tramite *gather*;
5. Stampa del risultato finale tramite il processo *root*.

**Approfondimento sulla Distribuzione.** Durante la fase di distribuzione dei dati, è necessario distribuire ad ogni processo nella topologia sia il sottovettore  $local\_x$  di  $x$  che la sottomatrice  $local\_A$  di  $A$ . Per quanto riguarda la distribuzione del  $x$ :

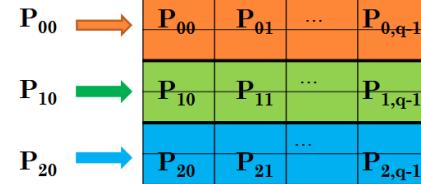
1. Il processore  $P_{00}$  distribuisce  $x$  ai processi della prima riga tramite *scatter*;
2. Ogni processo della prima riga invia copie del proprio sottovettore di  $x$  ai processori della propria colonna tramite *broadcast*.

Mentre invece, per la matrice  $A$ :

1. Il processore  $P_{00}$  distribuisce blocchi di righe di  $A$  tramite *scatter* ai processi della prima colonna. Ogni blocco contiene  $\frac{m}{p}$  righe;
2. Ogni processo della prima colonna dispone ora di una sottomatrice di dimensione  $\frac{m}{p} \times n$ . Effettua una trasposizione locale in modo da rendere contigue le colonne in memoria, quindi suddivide il risultato in  $q$  blocchi di dimensione  $\frac{m}{p} \times \frac{n}{q}$  e li distribuisce ai processi della propria riga tramite un secondo *scatter*.



(a) Passo 1



(b) Passo 2

Figura 11: Distribuzione delle sottomatrici di  $A$  ad ogni processo.

**Valutazione.** Il tempo di esecuzione della Strategia III dipende da tre operazioni principali:

- Calcolo di  $r_{ij} = A_{ij}x_j$ . Poiché  $A_{ij} = \frac{m}{p} \frac{n}{q}$  e  $x_j = \frac{n}{q}$ , allora:

$$T_p = 2 \frac{m}{p} \frac{n}{q} t_{calc}$$

- Somma in parallelo dei sottovettori  $local\_y$  di dimensione  $\frac{n}{p}$ . I tempi dipendono dalla strategia adottata per la somma, quindi avremo:

$$T_p^I = (q-1) \frac{m}{p} (t_{calc} + t_{com})$$

$$T_p^{II} = T_p^{III} = (\log_2 q) \frac{m}{p} (t_{calc} + t_{com})$$

- Gather finale per la riduzione dei risultati parziali:

$$T_p^I = \left[ 2 \frac{mn}{pq} + (q-1) \frac{m}{p} \right] t_{calc} + (q-1) \frac{m}{p} t_{com}$$

$$T_p^{II} = T_p^{III} = \left[ 2 \frac{mn}{pq} + (\log_2 q) \frac{m}{p} \right] t_{calc} + (\log_2 q) \frac{m}{p} t_{com}$$

**Comportamento Asintotico.** Ignorando le comunicazioni e ricordando che  $T_s = 2mnt_{calc}$  avremo che:

$$S_p^I = \frac{2mnt_{calc}}{\frac{2mn}{pq} + \frac{q-1}{2np}} = \frac{1}{\frac{1}{pq} + \frac{q-1}{2np}}, \quad E_p^I = \frac{1}{pq} \cdot \frac{1}{\frac{1}{pq} + \frac{q-1}{2n}} = \frac{1}{1 + \frac{q(q-1)}{2n}}$$

Una situazione analoga si verifica anche per la Strategia II e III per la somma. Per tutte le strategie, si ha:

- per  $p, q$  fissati e  $n \rightarrow \infty$ :

$$\lim_{n \rightarrow \infty} S_p = pq, \quad \lim_{n \rightarrow \infty} E_p = 1$$

Posso trascurare  $m$  perché non vi è dipendenza.

- per  $m, n$  fissati e  $q \rightarrow \infty$ :

$$\lim_{q \rightarrow \infty} S_p = 0, \quad \lim_{q \rightarrow \infty} E_p = 0$$

Posso trascurare  $p$  nel calcolo dell'efficienza perché non vi è dipendenza, mentre nello speed-up produrrà un risultato analogo.

## 4 GP-GPU e l'Ambiente CUDA

Le **Graphic Processing Unit (GPU)** sono microprocessori paralleli delle moderne schede video per computer o console. Le GPU sono nate per eseguire operazioni grafiche, ma, grazie alla loro potenza elaborativa in parallelo, le GPU sono state successivamente impiegate anche in applicazioni *General Purpose*, ovvero il loro contesto applicativo va oltre la grafica. Una GPU è composta da più multiprocessori organizzati secondo uno schema architettonale finalizzato al paradigma SIMD (Single Instruction Multiple Data): un solo processore controlla più ALU che eseguono lo stesso flusso di istruzioni su dati diversi.

**Confronto tra CPU e GPU.** Le **CPU** (Central Processing Unit) sono processori di tipo *general purpose*, progettati per eseguire una grande varietà di istruzioni e gestire compiti di natura diversa. L'esecuzione del flusso di istruzioni avviene in modo prevalentemente *sequenziale*, seguendo un controllo di flusso complesso ma molto flessibile. Le CPU sono inoltre dotate di strutture di memoria sofisticate e ad accesso ottimizzato, come le *cache* multilivello, che consentono di ridurre i tempi di accesso ai dati e migliorare le prestazioni in scenari con carichi di lavoro variabili.

Le **GPU** (Graphics Processing Unit), invece, sono specializzata in applicazioni parallele estremamente esigenti in termini di potenza di elaborazione (come il *rendering*) e dunque dedica più transistor all'elaborazione dei dati piuttosto che alla loro memorizzazione e al controllo del flusso d'esecuzione. Inoltre, come detto in precedenza, esse eseguono le operazioni in modo fortemente parallelo secondo il paradigma architettonale *SIMD*. Infine, le GPU presentano strutture di memoria più semplici e con latenza molto bassa, ottimizzate per garantire un'elevata velocità di accesso nei calcoli paralleli di tipo massivo, caratteristici delle applicazioni grafiche e di *machine learning*.

Una differenza fondamentale tra CPU e GPU riguarda le loro performance: le GPU hanno performance migliori a costi di produzione di massa. Per la legge di Moore le prestazioni delle CPU raddoppiano ogni 18 mesi; al contrario le prestazioni delle GPU raddoppiano ogni 6 mesi il che equivale al triplo della legge di Moore.



Figura 12: Differenze architetturali tra CPU e GPU.

### 4.1 GP-GPU

Il **General Purpose computation using Graphics Processing Units** (GP-GPU) si propone di impiegare le GPU per applicazioni parallele ad alte prestazioni (general purpose)

sfruttando le loro elevate capacità di elaborazione in parallelo, estendendo le loro capacità a compiti che vanno oltre la Computer Graphics. Tuttavia, non tutte le applicazioni traggono vantaggio dall'uso delle GPU: le applicazioni con un'elevata logica di controllo del processo di calcolo vengono eseguite efficientemente in modo sequenziale dalle tradizionali CPU, ma con pessime prestazioni dall'architettura parallela delle GPU (ad esempio, gestione dei database, compressione dei dati, algoritmi ricorsivi).

## 4.2 Programmare le GPU - CUDA

Mentre inizialmente la realizzazione di un'applicazione in ambiente GP-GPU consisteva nel programmare la GPU tramite le API di disegno 3D (esprimevano il problema da risolvere e i suoi dati in termini di vertici, triangoli e poligoni da manipolare), successivamente NVIDIA ha modificato la GPU rendendola interamente programmabile per le applicazioni scientifiche e aggiungendovi il supporto per linguaggi ad alto livello, come C e C++.

Il **Compute Unified Device Architecture** è un modello architettonico di tipo SIMD (**Single Instruction on Multi Thread**) progettato da NVIDIA per sfruttare le capacità dei sistemi paralleli multicore delle GPU. Grazie all'ambiente di sviluppo CUDA è possibile ricorrere a delle API apposite per la programmazione delle GPU piuttosto che alle API di disegno 3D.

### 4.2.1 Modello di Programmazione

Il modello di programmazione di CUDA considera la CPU e la GPU come due macchine distinte e separate, dette rispettivamente **host** e **device**. Un'applicazione CUDA combina:

- parti sequenziali (a carico dell'host),
- parti parallele, dette kernel (a carico del device).

Le applicazioni CUDA hanno una struttura eterogenea: ci sono parti seriali a carico dell'host e parti parallele (i kernel) a carico del device:

1. l'host ricopia i dati da elaborare sulla memoria della GPU prima di chiamare un kernel;
2. L'host chiama il kernel passandogli i parametri opportuni e configurandone l'esecuzione.
3. L'host recupera i risultati prelevandoli dalla memoria del device.

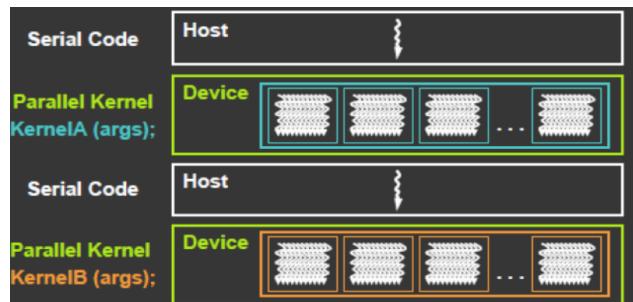


Figura 13: Modello di programmazione CUDA, che vede l'alternanza di host e device.

#### 4.2.2 Parallelismo in un'applicazione CUDA

L'unità fondamentale del parallelismo in CUDA è il **thread**: più thread eseguono lo stesso flusso di istruzioni su dati diversi carico del device (paradigma SIMT - Single Instruction Multiple Thread). I thread vengono raggruppati in blocchi (**block**), che vengono eseguiti sullo stesso multiprocessore e condividono la shared memory. A loro volta, i blocchi vengono raggruppati in una griglia (**grid**) che indica la configurazione dell'esecuzione parallela di un kernel. Ogni volta che l'host richiama un kernel deve passare i parametri e configurare il kernel:

- il programmatore definisce il numero di blocchi e di thread per blocco con cui il kernel verrà eseguito e la loro dimensionalità (transparent scalability);
- l'hardware decide autonomamente la distribuzione dei blocchi su ogni multiprocessore e ogni kernel viene "scalato" sui multiprocessori disponibili.

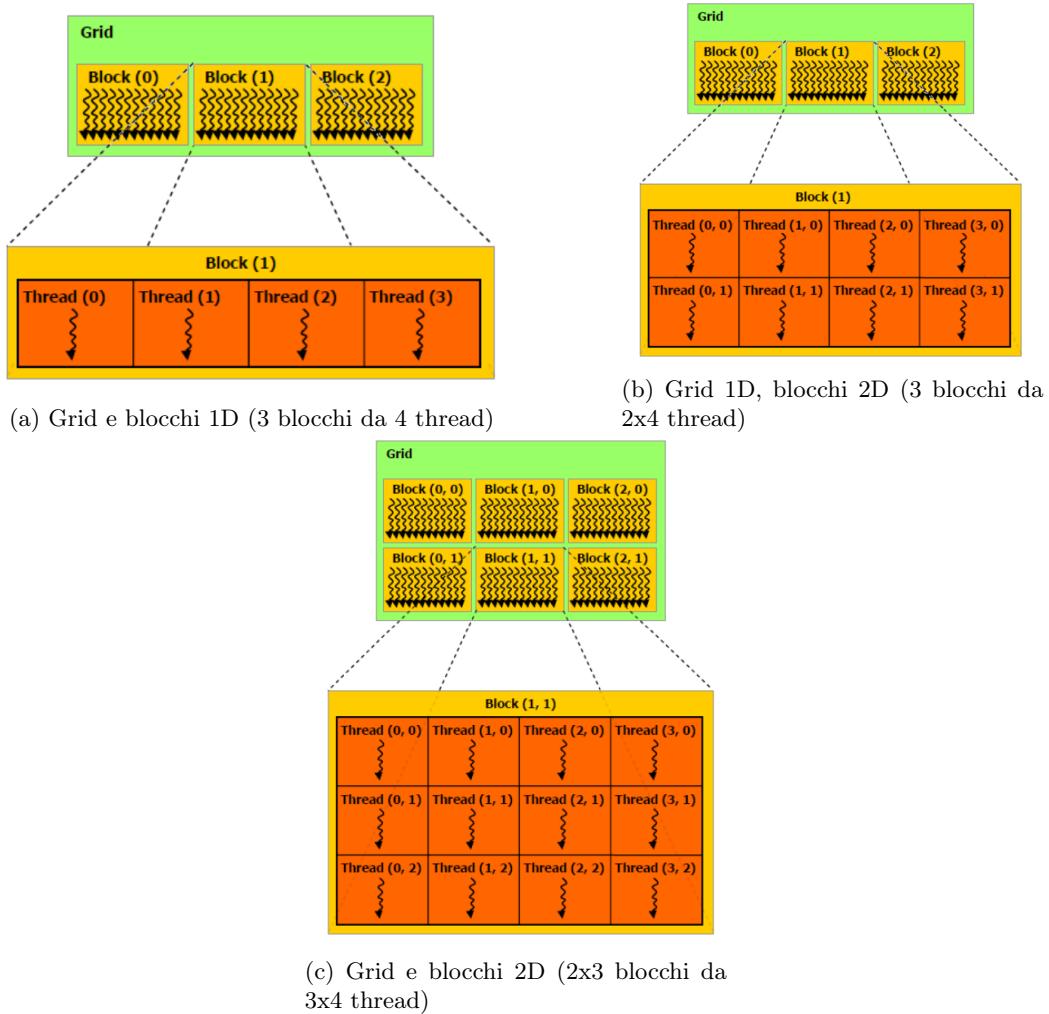


Figura 14: Multidimensionalità di grid, blocchi e thread.

**Distribuzione dei dati su thread.** I blocchi ed i thread all'interno di un blocco sono opportunamente indicizzati. Per determinare la porzione dei dati di cui si dovrà occupare un certo thread occorre determinare un'opportuna corrispondenza tra la porzione di dati e l'identificativo del thread all'interno del blocco e l'identificativo del blocco cui esso appartiene.

#### 4.2.3 Streaming Multiprocessor

I thread in CUDA sono raggruppati in blocchi. Tutti i thread appartenenti allo stesso blocco vengono eseguiti dallo stesso **Streaming Multiprocessor** (SM), un blocco architetturale fondamentale della GPU che contiene tutte le risorse necessarie per eseguire in parallelo migliaia di thread, come ad esempio la memoria condivisa (*shared memory*), utilizzata per la comunicazione tra i thread dello stesso blocco. Un nuovo blocco può essere eseguito su uno SM solo quando tutti i thread del blocco precedente hanno completato l'esecuzione. L'assegnazione dei blocchi agli SM avviene automaticamente da parte dell'hardware e non vi è alcuna garanzia sull'ordine di esecuzione: se uno SM dispone di risorse sufficienti, esso può mantenere residenti ed eseguire contemporaneamente più blocchi.

**Warp.** All'interno di uno SM, i thread vengono organizzati dall'hardware in gruppi di 32 chiamati **warp**. Ogni warp:

- condivide la stessa unità di controllo (**warp scheduler**);
- esegue la stessa istruzione in modalità SIMT (*Single Instruction, Multiple Threads*);
- è composto da thread appartenenti allo stesso blocco, disposti in ordine lineare;
- rappresenta l'unità minima di schedulazione all'interno dello SM.

In ogni ciclo di clock, uno dei warp pronti può essere selezionato per l'esecuzione; ogni SM mantiene residenti molti warp contemporaneamente e ne può schedulare più di uno in maniera indipendente. La strategia di schedulazione è detta **zero-overhead scheduling**: non è richiesto tempo aggiuntivo per cambiare warp in esecuzione, poiché essi risiedono già nello SM. Solo i warp la cui istruzione successiva ha tutti gli operandi disponibili sono candidati all'esecuzione; tra questi, lo scheduler seleziona quello da eseguire secondo una specifica politica di priorità.

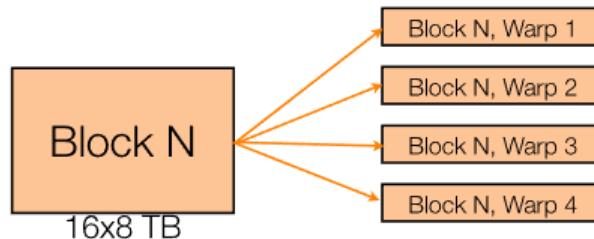


Figura 15: Struttura di un warp all'interno di un blocco.

**Divergenza nei warp.** Tutti i thread appartenenti a un warp devono eseguire la stessa istruzione nello stesso ciclo di clock; quando, a causa di rami condizionali, thread di uno stesso warp seguono percorsi di controllo differenti, si verifica una situazione di **divergenza**. L'esecuzione rimane corretta, ma il warp deve serializzare i percorsi divergenti, riducendo il parallelismo effettivo e introducendo un overhead nelle prestazioni. La divergenza è gestita dallo stesso warp scheduler, che continua ad applicare lo *zero-overhead scheduling* tra i warp residenti; la serializzazione riguarda esclusivamente i thread del warp divergente. Di seguito, un esempio di codice divergente:

```

1 // si assuma blockDim = 128
2 int sum = 0;
3
4 // i thread da 0 a 2 eseguono un ramo, quelli successivi l'altro
5 if (threadIdx.x > 2)
6     sum = sum + 2;
7 else
8     sum = sum + 5;

```

Codice 4: Esempio di programma divergente: i thread appartenenti allo stesso warp eseguono rami differenti e il warp viene serializzato.

Esempio di programma non divergente:

```

1 // si assuma blockDim = 128
2 int sum = 0;
3 int WARPSIZE = 32;
4
5 // tutti i thread appartenenti allo stesso warp valutano la condizione allo stesso modo
6 if (threadIdx.x / WARPSIZE > 2)
7     sum = sum + 3;
8 else
9     sum = sum + 5;

```

Codice 5: Esempio di programma non divergente: organizzando i thread per warp - la condizione è identica per tutti i thread del warp.

Poiché non è noto quale warp ( $0, 1, \dots$ ) venga schedulato per primo da uno SM, è necessario progettare l'algoritmo assumendo che i warp di un blocco possano essere eseguiti in ordine arbitrario, pur mantenendo coerenza interna: i thread di uno stesso warp devono seguire il medesimo percorso di controllo per evitare la divergenza.

## 4.3 Memorie di una GPU

Per ottenere prestazioni ottimali è fondamentale fare un buon uso delle memorie. Poiché CUDA lavora sia sulla CPU (host) che sulla GPU (device), Occorre tenere traccia della memoria su cui si sta operando. Nella GPU ci sono diverse memorie.

### 4.3.1 Tipi di Memorie

**Memoria globale.** La **memoria globale** (*global memory*) è la memoria principale della GPU ed è il luogo in cui risiedono i dati scambiati tra host e device tramite le operazioni `cudaMemcpy`. È accessibile da tutti i thread in lettura e scrittura e presenta dimensioni tipicamente elevate (nell'ordine dei gigabyte), ma anche una latenza molto alta, che la rende

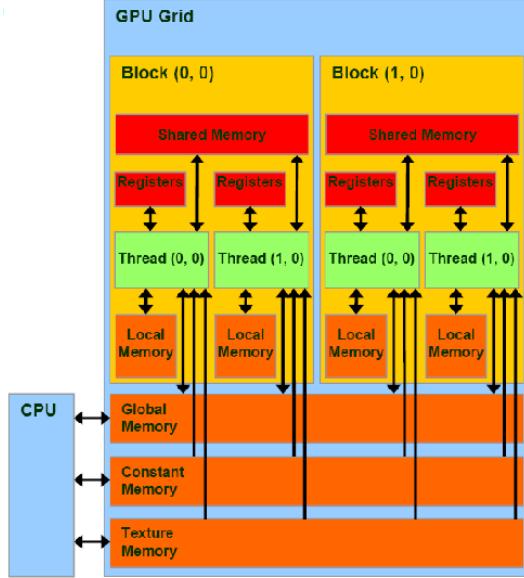


Figura 16: Schema delle memorie di una GPU.

la tipologia di memoria più lenta da utilizzare. Il contenuto della memoria globale persiste tra l'esecuzione di diversi kernel, salvo operazioni esplicite di deallocazione o sovrascrittura. Tutti gli argomenti del kernel sono nella memoria globale. Opzionalmente, le variabili memorizzate nella memoria globale, possono essere così dichiarate:

```
--device__ type variable;
```

**Memoria condivisa.** La **memoria condivisa** (*shared memory*) è una memoria on-chip<sup>1</sup> estremamente veloce, messa a disposizione dello *Streaming Multiprocessor*. Presenta dimensioni ridotte (nell'ordine di poche decine di kilobyte per SM), ma una latenza molto bassa, per cui è consigliabile sfruttarla al massimo quando possibile. È accessibile da tutti i thread appartenenti allo stesso blocco e fornisce un canale di comunicazione efficiente tra essi. Le variabili allocate in memoria condivisa esistono solo per la durata del kernel e non persistono tra diverse invocazioni. La dichiarazione di una variabile in memoria condivisa avviene tramite la parola chiave:

```
--shared__ type variable;
```

**Registri.** I **registri** (*registers*) costituiscono la memoria privata di ogni thread. Sono memorie on-chip, estremamente veloci e caratterizzate dai migliori tempi di accesso disponibili sulla GPU. Ogni thread possiede il proprio insieme di registri, non condivisibile con altri thread. Le variabili locali di un kernel vengono allocate nei registri quando possibile; se i registri non sono sufficienti, l'overflow viene gestito dalla *local memory* (situata in global memory). La dichiarazione di una variabile che risiede tipicamente in un registro avviene semplicemente come:

```
type variable;
```

---

<sup>1</sup>*On-chip* indica una memoria o una risorsa che si trova fisicamente all'interno dello stesso chip della GPU, molto vicino alle unità di calcolo.

**Memoria locale.** La **memoria locale** (*local memory*) è una memoria privata per ogni thread, utilizzata quando le variabili non possono essere allocate nei registri. Nonostante il nome, essa **non è on-chip**: è fisicamente allocata nella *global memory*, e presenta quindi una latenza elevata. La local memory viene utilizzata principalmente in due casi:

- quando si verifica un overflow dei registri disponibili;
- quando il compilatore deve gestire array o variabili locali la cui dimensione non può essere determinata a compile-time.

La dichiarazione di una variabile che potrebbe risiedere in memoria locale avviene semplicemente come:

```
type variable[DIM];
```

**Memoria costante.** La **memoria costante** (*constant memory*) è una memoria di sola lettura visibile da tutti i thread della griglia e accessibile anche dall'host. È particolarmente utile quando una stessa variabile o un piccolo insieme di dati devono essere letti frequentemente da molti thread, poiché viene mantenuta in una cache dedicata che riduce drasticamente la latenza. Sebbene sia fisicamente off-chip, risulta molto veloce in caso di accessi uniformi (tutti i thread leggono lo stesso dato), grazie al broadcast hardware. La dichiarazione di una variabile in memoria costante avviene tramite:

```
__constant__ type variable[DIM];
```

**Memoria texture.** La **memoria texture** (*texture memory*) è anch'essa una memoria di sola lettura, visibile da tutti i thread. È allocata in off-chip, ma resa efficiente grazie ad una cache specializzata, ottimizzata per la località spaziale bidimensionale. Per questa ragione è spesso impiegata per accedere a immagini o dati con struttura 2D.

#### 4.3.2 Uso Strategico delle Memorie

È consigliabile privilegiare l'uso dei registri e della memoria condivisa per ridurre al minimo l'utilizzo della memoria locale (che risiede fisicamente nella DRAM del device ed è quindi lenta). L'host può leggere e scrivere nella memoria globale, ma non può accedere alla memoria condivisa. Inoltre:

- utilizzare la memoria costante per variabili di sola lettura condivise tra tutti i thread;
- utilizzare la *shared memory* per variabili in lettura/scrittura condivise all'interno di un blocco;
- utilizzare i registri per variabili in lettura/scrittura ad uso esclusivo di un singolo thread.

È importante ricordare che il numero di blocchi che possono essere mantenuti contemporaneamente su uno SM è limitato dall'utilizzo di registri e memoria condivisa, oltre che dai vincoli architetturali sul numero massimo di blocchi e thread per SM. Una gestione efficiente di queste risorse è cruciale per ottenere buona *occupancy* e massimizzare il parallelismo.

## 5 CUDA - Configurazione Ottimale del Kernel

### 5.1 Griglie e Blocchi in CUDA

La scelta migliore del tipo di griglie e blocchi dipende dalla geometria del problema: griglie e blocchi 1D sono adatte per dati 1D, ma griglie e blocchi multidimensionali sono necessari per dati organizzati in più dimensioni e insiemi di dati più grandi rispetto alle dimensioni massime dei blocchi.

#### 5.1.1 Tipi e Strutture Predefinite

`uint3` e `dim3` sono strutture di interi senza segno, contenenti tre valori `x`, `y` e `z`.

```
1 struct uint3 {  
2     unsigned int x, y, z;  
3 }
```

In particolare, `dim3` è una struttura del tipo `uint3`, le cui componenti sono inizializzate a 1. Per specificare la grandezza di griglie e blocchi usiamo:

- `dim3 gridDim`: lunghezza delle dimensioni `x`, `y` e `z` della griglia. Il numero di blocchi nella griglia è calcolato come segue:

$$\text{Numero di Blocchi nella Griglia} = \text{gridDim}.x \times \text{gridDim}.y \times \text{gridDim}.z$$

- `dim3 blockDim`: lunghezza delle dimensioni `x`, `y` e `z` del blocco. Il numero di thread in un blocco è calcolato come segue:

$$\text{Numero di Thread in un Blocco} = \text{blockDim}.x \times \text{blockDim}.y \times \text{blockDim}.z$$

Per identificare blocchi e thread usiamo:

- `uint3 blockIdx`: otteniamo le coordinate del blocco nella griglia:

$$(\text{blockIdx}.x, \text{blockIdx}.y, \text{blockIdx}.z)$$

- `uint3 threadIdx`: otteniamo le coordinate del thread nel blocco:

$$(\text{threadIdx}.x, \text{threadIdx}.y, \text{threadIdx}.z)$$

#### 5.1.2 Distribuzione dei Dati su Thread

I blocchi ed i thread all'interno di un blocco sono opportunamente indicizzati. Per determinare la porzione dei dati di cui si dovrà occupare un certo thread occorre determinare un'opportuna corrispondenza tra la porzione di dati e l'identificativo del thread all'interno del blocco e l'identificativo del blocco cui esso appartiene.

- Geometria 1D: la scelta più naturale per la configurazione del kernel è una griglia 1D con blocchi 1D. La relazione tra l'indice dell'array monodimensionale che rappresenta i dati del problema e l'indice del thread è il seguente:

$$i = \text{blockIdx}.x \times \text{gridDim}.x + \text{threadIdx}.x$$

dove  $i$  viene anche definito indice o coordinata globale del thread;

- Geometria 2D: la scelta più naturale per la configurazione del kernel è una griglia 2D con blocchi 2D. La relazione tra le coordinate  $(i, j)$  dell'array 2D che rappresenta i dati del problema e le coordinate del thread è la seguente:

$$i = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}$$

$$j = \text{blockIdx.y} \times \text{blockDim.y} + \text{threadIdx.y}$$

dove  $i, j$  sono le coordinate globali del thread.

## 5.2 Configurazione del Kernel

Configurare un kernel significa stabilire come sono organizzate la griglia e i blocchi di thread, ovvero definire la struttura dei parametri `gridDim` e `blockDim`. In particolare, bisogna scegliere:

- se la griglia è 1D, 2D o 3D e la lunghezza di ciascuna dimensione;
- se ogni blocco è 1D, 2D o 3D e la lunghezza di ciascuna dimensione.

La configurazione ottimale di un kernel CUDA richiede di definire correttamente la geometria della griglia (`gridDim`) e dei blocchi (`blockDim`). Il processo di configurazione segue due passaggi fondamentali:

1. **Scelta del numero di dimensioni della griglia e dei blocchi** (1D, 2D o 3D), in funzione della geometria del problema da risolvere.
2. **Impostazione delle dimensioni del blocco**:

$$\text{blockDim} = \{\text{blockDim.x}, \text{blockDim.y}, \text{blockDim.z}\}$$

Le dimensioni della griglia sono ricavate come segue:

- **Problema 1D** di dimensione  $N$ :  $\text{gridDim.x} = \frac{N}{\text{blockDim.x}}$
- **Problema 2D** di dimensione  $N \times M$ :  $\text{gridDim.x} = \frac{N}{\text{blockDim.x}}, \quad \text{gridDim.y} = \frac{M}{\text{blockDim.y}}$

## 5.3 Scelta di `blockDim`

`blockDim` deve essere scelto in modo da saturare le risorse a disposizione dello **Streaming Multiprocessor** (SM) e da elevare il grado di parallelismo potenziale, ossia (in ordine di importanza):

1. Massimizzare il numero di **thread attivi per SM**;
2. Massimizzare il numero di **blocchi attivi per SM**.

Per configurazioni 1D, una scelta efficace è utilizzare un valore di `blockDim.x` che rispetti la seguente relazione:

$$\text{blockDim} = \frac{\text{massimo numero thread per SM}}{\text{massimo numero blocchi per SM}}$$

**Vincoli Strutturali nella Scelta di `blockDim`.** La configurazione di `blockDim` deve rispettare una serie di vincoli strutturali imposti dall'hardware della GPU. Tali vincoli limitano lo spazio delle possibili configurazioni e influenzano direttamente l'occupancy<sup>2</sup> dello Streaming Multiprocessor (SM). In particolare, occorre considerare:

1. **Massimo numero di thread per blocco:** limite fisico imposto dalla GPU (ad esempio 1024 thread).
2. **Massimo numero di blocchi per l'intera griglia:** vincolo sull'estensione massima di `gridDim`.
3. **Numero di registri disponibili per SM:** determina quanti thread possono essere attivi contemporaneamente.
4. **Quantità di shared memory disponibile per SM:** influisce sul numero massimo di blocchi residenti sullo stesso SM.

Da questi vincoli derivano le seguenti conseguenze operative:

- Se si superano i vincoli (1) e/o (2), il kernel *non può essere eseguito* e il programma termina con errore di configurazione.
- Se si superano i vincoli (3) e/o (4), il kernel viene comunque eseguito, ma con un numero *ridotto di blocchi attivi per SM*, comportando una minore occupancy e quindi prestazioni inferiori.

**Verificare Numero di Registri e Memoria Condivisa Usati - Esempio.** Dato un problema di dimensione  $N$ , si vuole configurare un kernel CUDA con griglia 1D e blocchi 1D su architettura **Fermi**, assumendo:

- 7 registri per thread;
- 256 bytes di shared memory per blocco.

Utilizzando la regola:

$$\text{blockDim} = \frac{\text{massimo numero di thread per SM}}{\text{massimo numero di blocchi per SM}} = \frac{1536}{8} = \boxed{192}$$

Verifica limitazioni sulla memoria per SM usando gli strumenti di Profiling (dimensione registri e memoria condivisa):

- **Registri:**  $1536 \times 7 = 10,536 < 32K \Rightarrow \text{OK}$
- **Shared memory:**  $8 \times 256 \text{ bytes} = 2048 \text{ bytes} < 48 \text{ kB} \Rightarrow \text{OK}$

Verifica numero di blocchi complessivamente usati: la griglia deve rispettare il limite massimo di  $2^{16} - 1$  blocchi in `gridDim.x`:

$$\frac{N}{192} < 2^{16} - 1 \implies N < 192 \times (2^{16} - 1) \approx 12 \times 10^6.$$

---

<sup>2</sup>L'occupancy è la frazione di thread attivi su uno Streaming Multiprocessor (SM) rispetto al numero massimo teoricamente disponibile. Indica quanto efficacemente le risorse hardware della GPU vengono utilizzate: un'occupancy elevata consente di nascondere la latenza della memoria e migliorare il throughput complessivo del kernel.

Pertanto, la configurazione è valida per problemi di dimensione fino a circa  $N \approx 1.2 \times 10^7$ .

## 5.4 CUDA Occupancy Calculator

Il *CUDA Occupancy Calculator* è uno strumento utile per verificare la bontà della configurazione scelta per il kernel. L'utente deve comunque impostare manualmente i parametri principali della configurazione (numero di thread per blocco, utilizzo di registri, shared memory per blocco, ecc.).

## 5.5 Scelta di `blockDim` su Architetture Fermi e Kepler

In questa sezione si analizza come la dimensione del blocco (`blockDim`) influisca sull'occupancy dello Streaming Multiprocessor (SM), considerando i limiti strutturali delle architetture *Fermi* e *Kepler*. L'obiettivo è determinare la dimensione ottimale di un blocco di thread, massimizzando il parallelismo potenziale e sfruttando pienamente le risorse offerte da ciascun SM. La scelta di `blockDim` deve rispettare i seguenti **vincoli strutturali**:

- Un blocco CUDA può contenere al massimo **1024 thread**.
- La griglia può contenere al massimo: **65 535** blocchi per dimensione su architettura *Fermi*, e fino a  $2^{31} - 1$  blocchi per dimensione su architettura *Kepler*.
- `blockDim` deve essere scelto in modo da saturare le risorse dello Streaming Multiprocessor (SM), massimizzare il numero di thread attivi e mantenere un elevato grado di parallelismo attraverso più blocchi residenti.

### 5.5.1 Architettura *Fermi*

In un'architettura Fermi ogni SM può gestire fino a **1536 thread**. Il numero massimo di **8 blocchi residenti** per SM. Si valutano alcuni possibili valori di `blockDim.x = blockDim.y`:

- Blocco  $8 \times 8 = 64$  thread:  $\frac{1536}{64} = 24$  blocchi necessari per saturare lo SM. Ma essendo permessi solo 8 blocchi residenti:  $64 \times 8 = 512$  thread per SM < 1536 (Occupancy bassa).
- Blocco  $16 \times 16 = 256$  thread:  $\frac{1536}{256} = 6$  blocchi residenti.  $256 \times 6 = 1536$  thread per SM (Occupancy piena dello SM).
- Blocco  $32 \times 32 = 1024$  thread:  $\frac{1536}{1024} \approx 1.5 \Rightarrow 1$  blocco per SM.  $1024 \times 1 = 1024$  thread per SM (Occupancy parziale).

La scelta ottimale per un'architettura Fermi risulta essere: `blockDim.x = blockDim.y = 16`.

### 5.5.2 Architettura *Kepler*

In un'architettura Kepler ogni SM può gestire fino a **2048 thread**. Il numero massimo di **16 blocchi residenti** per SM. Si valutano alcuni possibili valori di `blockDim.x = blockDim.y`:

- Blocco  $8 \times 8 = 64$  thread:  $\frac{2048}{64} = 32$  blocchi necessari. Ma il limite è 16 blocchi:  $64 \times 16 = 1024$  thread per SM < 2048 (Occupancy parziale).

- Blocco  $16 \times 16 = 256$  thread:  $\frac{2048}{256} = 8$  blocchi residenti.  $256 \times 8 = 2048$  thread per SM (Occupancy piena).
- Blocco  $32 \times 32 = 1024$  thread:  $\frac{2048}{1024} = 2$  blocchi.  $1024 \times 2 = 2048$  thread per SM (Occupancy piena, ma minor parallelismo potenziale - solo 2 blocchi.).

Anche per un'architettura Kepler, la scelta ottimale risulta essere:  $blockDim.x = blockDim.y = 16$ .

## 6 Shared Memory e Prodotto Scalare

### 6.1 Approfondimento Shared Memory

La **memoria condivisa** (*shared memory*) è una memoria on-chip residente su ciascuno *Streaming Multiprocessor* (SM). È estremamente veloce e fornisce un canale di comunicazione tra i thread appartenenti allo stesso blocco. Le sue principali caratteristiche sono:

- **Accesso R/W** limitato ai thread del blocco;
- **Durata limitata al kernel**: il contenuto non persiste tra l'invocazione di un kernel e quella successiva;
- **Bassa latenza**: circa 2 cicli di clock;
- **Throughput**: 4 byte per banco ogni 2 cicli di clock;
- **Dimensione tipica**: 48 KB per SM (configurabile a 16/48 KB).

La shared memory è una risorsa critica: essendo condivisa tra tutti i blocchi residenti in uno SM, una richiesta elevata di memoria condivisa da parte di un kernel riduce il numero di blocchi che possono risiedere simultaneamente nello SM, limitando il parallelismo a livello di device. L'accesso alla shared memory avviene a **livello di warp**: nel caso migliore un'unica transazione serve i 32 thread del warp; nel caso peggiore sono necessarie 32 transazioni, una per ogni thread. L'uso della shared memory segue generalmente questa sequenza:

1. caricamento dei dati dalla global memory alla shared memory;
2. eventuale sincronizzazione tramite `--syncthreads()`;
3. computazione sui dati residenti in shared memory;
4. eventuale sincronizzazione;
5. scrittura dei risultati in global memory.

#### 6.1.1 Organizzazione della Shared Memory

La shared memory è organizzata in **32 banchi** (*banks*), ciascuno di ampiezza **4 byte**. Ogni banco può contenere: 1 `int` o 1 `float`, 2 `short`, 4 `char`, 1 `half`, etc. La distribuzione dei dati avviene ciclicamente: ogni blocco di 4 byte viene assegnato a un banco successivo in ordine circolare.

**Modalità di accesso.** Gli accessi avvengono sempre per warp e possono essere di tre tipi:

- **Multicast**: se  $n$  thread dello stesso warp accedono allo stesso indirizzo, l'accesso avviene con una singola transazione.
- **Broadcast**: se tutti i thread accedono allo stesso indirizzo, una sola transazione serve l'intero warp.

- **Bank conflict:** si verifica quando due o più thread di un warp accedono a indirizzi differenti che risiedono nello stesso banco. In questo caso gli accessi vengono serviti in modo seriale, degradando le prestazioni. Di seguito un esempio di conflitto doppio:

```

1 // Il thread 0 accede ad a[0] ((banco 0)
2 // Il thread 16 accede ad a[32] ((banco 0)
3 x = a[2 * threadIdx.x + 5];
4

```

Codice 6: Conflitto doppio: due thread dello stesso warp accedono ad elementi diversi dello stesso banco.

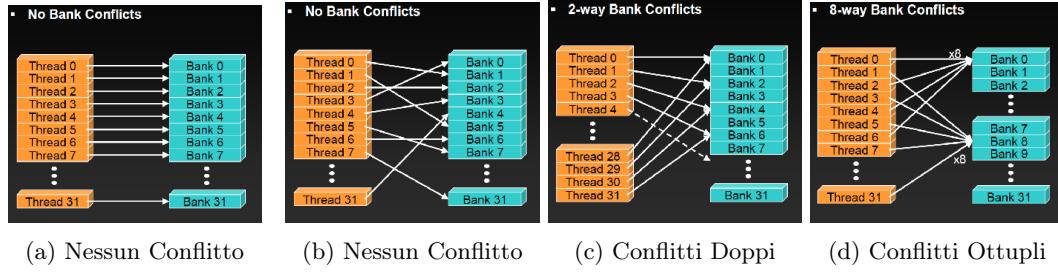


Figura 17: Negli esempi (c) e (d) è possibile osservare conflitti: thread differenti dello stesso warp tentano di accedere a dati differenti, residenti sullo stesso banco.

**Accessi e divergenza.** La divergenza del flusso di controllo e i conflitti in shared memory sono fenomeni distinti: un codice divergente non necessariamente genera conflitti di memoria. Infatti, la divergenza riguarda il *percorso di esecuzione* seguito dai thread di un warp, mentre i bank conflict dipendono esclusivamente dai *pattern di accesso* agli indirizzi in memoria. Nell'esempio seguente il codice è divergente (i thread valutano condizioni diverse), ma gli accessi alla memoria avvengono in un'unica transazione poiché tutti i thread che eseguono il ramo accedono allo stesso elemento dell'array:

```

1 if (threadIdx.x > 2)
2     x = a[1] + 5;

```

Codice 7: Il codice è divergente ma gli accessi in memoria avvengono in un'unica transazione.

## 6.2 Prodotto scalare

Il prodotto scalare tra due vettori è un'operazione che restituisce un numero (uno scalare) e si calcola sommando il prodotto delle loro componenti corrispondenti. Dati due vettori  $a$  e  $b$  di dimensione  $N$ :

$$c = ab = \sum_{i=0}^{N-1} a_i b_i$$

### 6.2.1 Strategia I (Banale)

Volendo implementare in CUDA il prodotto scalare tra due array di `float`  $a$  e  $b$ , introduciamo innanzitutto il vettore

$$v = (a_0 b_0, a_1 b_1, \dots, a_{N-1} b_{N-1}).$$

Il prodotto scalare può quindi essere riscritto come:

$$c = \sum_{i=0}^{N-1} v_i.$$

L'idea della *Strategia I* è demandare alla GPU il solo calcolo dei prodotti elemento per elemento, memorizzando il risultato nel vettore ausiliario  $v$ , e poi sommare *serialmente* tutti gli elementi di  $v$  sull'host (CPU). In questo modo:

- si sfrutta un parallelismo limitato (solo per il calcolo di  $a_i b_i$ );
- si utilizza la memoria globale della GPU (più lenta rispetto alla memoria condivisa);
- è necessario trasferire il vettore  $v$  dal device all'host per effettuare la riduzione finale.

Di conseguenza, questa strategia non è ottimale dal punto di vista delle prestazioni complessive, poiché il collo di bottiglia resta la somma seriale e il trasferimento di memoria host-device.

```

1 --global__ void dotProdGPU(float *v, const float *a, const float *b, int N) {
2     // indice globale del thread
3     int idx = threadIdx.x + blockIdx.x * blockDim.x;
4
5     if (idx < N) {
6         v[idx] = a[idx] * b[idx];
7     }
8 }
```

Codice 8: Kernel della Strategia I (banale) per il prodotto scalare.

### 6.2.2 Strategia II (Memoria Condivisa)

L'idea alla base della *Strategia II* è scomporre la sommatoria globale in più sommatorie parziali, ciascuna delle quali viene calcolata dai thread di uno stesso blocco. Ogni blocco esegue quindi una riduzione locale nella memoria condivisa (*shared memory*), producendo una somma parziale. Infine, l'host si occuperà di sommare in maniera seriale tutte le somme parziali. Consideriamo:

$$bD = \text{blockDim.x}, \quad gD = \text{gridDim.x} = \frac{N}{\text{blockDim.x}}.$$

Suddividiamo la sommatoria di lunghezza  $N$  in  $gD$  blocchi, ciascuno di lunghezza  $bD$ :

$$c = a \cdot b = \sum_{i=0}^{N-1} a_i b_i = \sum_{i=0}^{N-1} v_i = \sum_{i=0}^{bD-1} v_i + \sum_{i=bD}^{2bD-1} v_i + \cdots + \sum_{i=N-bD}^{N-1} v_i.$$

Poiché la riduzione deve avvenire nella memoria condivisa, i prodotti calcolati da ciascun blocco vengono prima copiati in un array temporaneo  $w[bD]$  allocato in *shared memory*. Ogni blocco esegue quindi la riduzione locale:

$$s_j = \sum_{i=0}^{bD-1} w_i,$$

dove  $s_j$  è la somma parziale prodotta dal blocco  $j$ . Infine, l'host combina serialmente tutte le somme parziali:

$$c = \sum_{j=0}^{gD-1} s_j.$$

Questa strategia migliora significativamente il parallelismo nella fase di riduzione, riducendo la quantità di lavoro seriale rispetto alla Strategia I. In base alla strategia adottata per la riduzione, la Strategia II presenta due varianti, presentate nella sezione Lab 4.

## 7 Libreria cuBlas

Gran parte dei problemi di calcolo scientifico richiedono la risoluzione di uno o più compiti tipici dell'algebra lineare numerica, come la soluzione di sistemi lineari, la determinazione di autovalori e autovettori oppure il calcolo della *Singular Value Decomposition* (SVD)<sup>3</sup>. Tali operazioni costituiscono spesso una parte significativa del costo computazionale totale necessario alla risoluzione del problema complessivo. Per questo motivo, un'implementazione efficiente degli algoritmi dedicati all'algebra lineare numerica riveste un ruolo fondamentale nella progettazione di soluzioni ad alte prestazioni.

Gli algoritmi di Algebra Lineare Numerica (ALN) condividono un insieme relativamente ristretto e stabile di operazioni elementari, dalle quali deriva la quasi totalità dei calcoli necessari. Le operazioni fondamentali possono essere ricondotte a tre categorie principali:

- **Prodotto scalare tra due vettori:**  $w = v_1 \cdot v_2$ , operazione che coinvolge due vettori e restituisce uno scalare;
- **Prodotto matrice–vettore:**  $w = A \cdot v$ , operazione che combina una matrice con un vettore producendo un nuovo vettore;
- **Prodotto matrice–matrice:**  $B = A_1 \cdot A_2$ , operazione che combina due matrici generando una nuova matrice.

### 7.1 Da BLAS a cuBLAS

La **Basic Linear Algebra Subprograms** (BLAS) è una delle prime librerie sviluppate (1979) per l'esecuzione di operazioni di base del calcolo matriciale, progettata per ottimizzare gli accessi alla memoria e massimizzare l'efficienza computazionale. Poiché molte librerie di livello più alto si appoggiano a BLAS, nel tempo ne sono state prodotte diverse versioni, ciascuna ottimizzata per specifiche architetture hardware.

La libreria **cuBLAS** rappresenta l'implementazione in CUDA-C dello standard BLAS. È stata introdotta con l'obiettivo di fornire al programmatore primitive di algebra lineare eseguite direttamente sulla GPU, così da sfruttare l'elevato grado di parallelismo offerto dalla scheda grafica e ottenere un notevole risparmio nei tempi di esecuzione delle operazioni.

---

<sup>3</sup>La *Singular Value Decomposition* è una delle decomposizioni più rilevanti in algebra lineare e nel machine learning. Consente di scomporre una matrice, anche non quadrata, in tre matrici con proprietà geometriche e numeriche particolarmente utili.

## 8 Sistemi di Raccomandazione

I sistemi di raccomandazione nascono con l'obiettivo di supportare gli utenti nelle scelte all'interno di insiemi di dati sempre più vasti, tipici degli scenari di *Big Data*. L'abbondanza di informazioni porta infatti al cosiddetto paradosso della scelta: di fronte a troppe alternative, l'utente può impiegare molto tempo prima di individuare ciò che cerca, non trovare affatto la risorsa desiderata oppure accontentarsi di una soluzione subottimale. Un sistema di raccomandazione riduce questa complessità orientando l'utente verso elementi potenzialmente rilevanti, sulla base delle sue preferenze esplicite o implicite. Tali sistemi possono suggerire prodotti, film, contenuti multimediali, oppure indicare utenti e pagine da seguire in un social network. In questo modo si configura una strategia *win-win*: l'utente ottiene suggerimenti personalizzati e più pertinenti, mentre la piattaforma aumenta l'engagement e la probabilità di interazioni significative.

### 8.1 Big Data

I Big Data sono un insieme di dataset grandi e complessi, che rendono difficile la loro elaborazione utilizzando i tradizionali strumenti di gestione dei database o le applicazioni di elaborazione dei dati convenzionali. Le 4V rappresentano le caratteristiche fondamentali che distinguono i Big Data dai dati tradizionali:

- **Volume:** si riferisce alla quantità enorme di dati generati e raccolti, che può raggiungere dimensioni di terabyte o petabyte.
- **Velocità (Velocity):** indica la rapidità con cui i dati vengono generati, trasmessi e devono essere elaborati per produrre valore in tempo reale o quasi.
- **Varietà (Variety):** riguarda la molteplicità di formati e tipologie di dati (strutturati, semi-strutturati, non strutturati) provenienti da fonti diverse.
- **Veridicità (Veracity):** rappresenta l'affidabilità, la qualità e la precisione dei dati, che possono contenere rumore, errori o informazioni poco attendibili.



Figura 18: Modello delle 4V (2012), basato sulla tassonomia di Laney (2001).

## 8.2 Tipi di Sistemi di Raccomandazione

Ogni sistema di raccomandazione presenta tre elementi fondamentali:

- **User:** sono gli utenti di un sistema di raccomandazione;
- **Item:** sono gli oggetti che ci vengono suggeriti dal sistema e vengono rappresentati con un set di attributi e proprietà;
- **Rating:** è la valutazione che l'utente fornisce del prodotto considerato. Può essere un numero o una valutazione binaria.

Esistono due principali classi di sistemi di raccomandazione:

- **Sistemi Content-Based (CB):** utilizzano gli attributi dei prodotti che gli utenti hanno votato in passato per calcolare la similarità con altri item. Ad esempio, se un utente ha valutato positivamente un film del genere commedia, gli verrà suggerito un altro film dello stesso genere;
- **Sistemi Collaborative-Filtering (CF):** raccomandano agli utenti prodotti apprezzati da altri utenti ritenuti simili. Analogamente si può tenere conto della similarità tra prodotti. Questi sistemi si sono molto diffusi grazie al Netflix Prize. Si dividono a loro volta in:
  - **Sistemi Model-Based:** si basano su un modello (statistico) che permette al sistema di imparare a riconoscere schemi complessi su un set di dati di training;
  - **Sistemi Memory-Based (o Instance-Based):** si basano sul concetto di similarità tra utenti, cioè si assume che ogni utente faccia parte di un gruppo di persone con interessi simili. Analogamente, possono basarsi sul concetto di similarità tra prodotti.

Tuttavia, possiamo anche includere altre classi meno diffuse o utilizzate in maniera ibrida con le precedenti:

- **Sistemi Demografici:** forniscono raccomandazioni in base al profilo demografico dell'utente;
- **Sistemi Community-Based:** forniscono raccomandazioni in base alle preferenze degli amici dell'utente. Difatti, le persone tendono ad apprezzare maggiormente le raccomandazioni di amici rispetto a quelle di utenti simili, ma sconosciuti. Inoltre tali sistemi si sono molto diffusi grazie al successo dei social-network.

## 8.3 Problemi dei Sistemi di Raccomandazione

I sistemi di raccomandazione presentano i seguenti problemi principali:

- **Cold Start Problem:** causato dall'incapacità di gestire i nuovi prodotti o utenti che vengono inseriti nel sistema. Ad esempio, nei sistemi content-based occorre avere una conoscenza dei gusti dell'utente; nei sistemi collaborative-filtering non si può stabilire la similarità di nuovi utenti e prodotti;

- **Scalabilità:** spesso è necessaria una grande potenza di calcolo per effettuare previsioni in tempo reale, a causa della grande mole di dati da analizzare;
- **Sparsità:** poiché i prodotti sono moltissimi, normalmente anche gli utenti più attivi avranno valutato solo una minuscola percentuale di essi.

## 8.4 Formalizzazione Matematica del Problema della Raccomandazione

Siano dati:

- $U$ : insieme degli utenti;
- $I$ : insieme degli item;
- $r : U \times I \rightarrow \mathbb{R}$ : funzione di rating, che associa ad ogni coppia utente-item un punteggio.

Il problema della raccomandazione può essere formalizzato come segue:

$$\forall u \in U, \text{ determinare } i_u \in I \text{ tale che } i_u = \arg \max_{i \in I} r(u, i).$$

### 8.4.1 Cross Validation

La *Cross Validation* è una strategia di valutazione che permette di stimare in modo robusto le prestazioni di un modello, riducendo il rischio di overfitting e fornendo una misura più affidabile della sua capacità di generalizzazione. L'idea alla base consiste nel suddividere il dataset in più partizioni e nel riutilizzarle ciclicamente come dati di addestramento e di test. Il processo si articola in due fasi fondamentali:

1. **Training:** l'algoritmo viene addestrato su un sottoinsieme dei dati (*training set*);
2. **Testing:** il modello viene valutato sul sottoinsieme rimanente (*test set*).

Nella pratica, si utilizza frequentemente la *k-fold Cross Validation*: il dataset viene suddiviso in  $k$  blocchi (o *fold*) di dimensione simile e si eseguono  $k$  iterazioni, ognuna delle quali utilizza un fold come test set e i restanti  $k - 1$  come training set. Al termine delle  $k$  iterazioni, le metriche ottenute vengono mediate, ottenendo una valutazione complessiva più stabile.

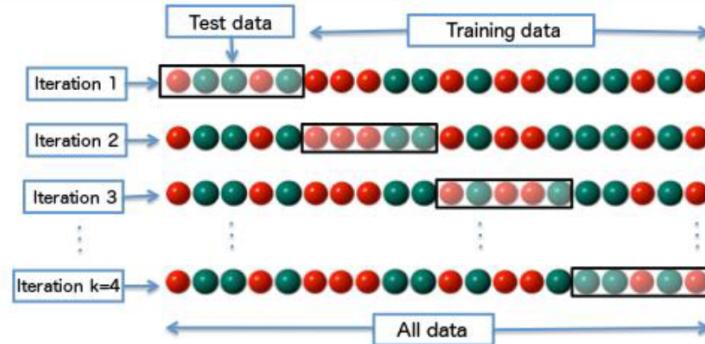


Figura 19: Esempio di k-fold Cross Validation a 4 fold.

### 8.4.2 Matrice dei Rating

La matrice dei rating  $R$  rappresenta sulle righe gli utenti e sulle colonne gli item. Il generico elemento

$$r_{kl} = r(u_k, i_l), \quad u_k \in U, i_l \in I$$

indica la valutazione che l'utente  $u_k$  ha assegnato all'item  $i_l$ . La matrice  $R$  è tipicamente molto *sparsa*, poiché la maggior parte dei rating non è nota: ogni utente valuta solo una piccola porzione degli item disponibili. Il compito del sistema di raccomandazione consiste dunque nello stimare i rating mancanti sfruttando quelli osservati, in modo da prevedere quali item possano risultare più rilevanti per ogni utente.

**Approssimazione tramite Similarità.** Esistono diversi metodi numerici per approssimare la matrice  $R$ , tra cui l'uso delle misure di **similarità**. Nei metodi *memory-based* e *model-based* si parla infatti di similarità tra utenti o tra item, calcolata sulla base delle valutazioni disponibili. Di seguito vengono illustrate alcune delle principali misure utilizzate in letteratura:

- **Distanza Euclidea:** la distanza euclidea misura la differenza tra le valutazioni fornite da due utenti. Siano  $\vec{r}_a$  e  $\vec{r}_b$  i vettori delle valutazioni degli utenti  $a$  e  $b$  sugli item in  $I$ . Poiché non tutti gli utenti valutano gli stessi item, la distanza viene calcolata solo sull'insieme degli item valutati da entrambi:

$$I_{a,b} = \{p \in I \mid r_{a,p} \text{ e } r_{b,p} \text{ sono definiti}\}.$$

La distanza euclidea è quindi definita come:

$$d(a, b) = \sqrt{\sum_{p \in I_{a,b}} (r_{a,p} - r_{b,p})^2}.$$

Notiamo che questa è una misura di *distanza*: valori più piccoli indicano utenti più simili;

- **Coseno di Similarità:** il *cosine similarity* misura il grado di somiglianza tra due vettori confrontando l'angolo tra di essi nello spazio delle valutazioni. Siano  $\vec{r}_a$  e  $\vec{r}_b$  i vettori delle valutazioni degli utenti  $a$  e  $b$  sugli item comuni  $I_{a,b}$ . La similarità è definita come:

$$\text{sim}(a, b) = \frac{\vec{r}_a \cdot \vec{r}_b}{\|\vec{r}_a\|_2 \cdot \|\vec{r}_b\|_2} = \frac{\sum_{p \in I_{a,b}} r_{a,p} r_{b,p}}{\sqrt{\sum_{p \in I_{a,b}} r_{a,p}^2} \sqrt{\sum_{p \in I_{a,b}} r_{b,p}^2}}$$

Il valore è compreso tra  $-1$  e  $1$ : valori prossimi a  $1$  indicano utenti con preferenze molto simili, mentre valori vicini a  $0$  indicano assenza di correlazione. Questa misura è molto utilizzata nei sistemi di raccomandazione poiché è robusta rispetto alle differenze di scala nelle valutazioni dei singoli utenti;

- **Correlazione di Pearson:** La *correlazione di Pearson* misura il grado di relazione lineare tra le valutazioni di due utenti, tenendo conto delle differenze individuali nei livelli medi di rating. Siano  $\vec{r}_a$  e  $\vec{r}_b$  i vettori delle valutazioni di  $a$  e  $b$  sugli item comuni

$I_{a,b}$ . Indichiamo con

$$\bar{r}_a = \frac{1}{|I_{a,b}|} \sum_{p \in I_{a,b}} r_{a,p}, \quad \bar{r}_b = \frac{1}{|I_{a,b}|} \sum_{p \in I_{a,b}} r_{b,p}$$

le rispettive medie delle valutazioni sugli item condivisi. La correlazione di Pearson è definita come:

$$sim(a, b) = \frac{\sum_{p \in I_{a,b}} (r_{a,p} - \bar{r}_a)(r_{b,p} - \bar{r}_b)}{\sqrt{\sum_{p \in I_{a,b}} (r_{a,p} - \bar{r}_a)^2} \sqrt{\sum_{p \in I_{a,b}} (r_{b,p} - \bar{r}_b)^2}}.$$

Il valore della correlazione varia tra  $-1$  e  $1$ : valori prossimi a  $1$  indicano una forte correlazione positiva nelle preferenze dei due utenti, mentre valori vicini a  $-1$  indicano una forte correlazione negativa. Grazie alla normalizzazione rispetto alla media, questa misura è particolarmente efficace quando gli utenti tendono a utilizzare scale di rating differenti.

**Approssimazione tramite K-NN.** La *k-Nearest Neighbours* è una strategia di tipo *memory-based* che consente di fornire un'approssimazione  $\hat{R}$  della matrice  $R$ , stimando i valori mancanti sulla base degli utenti più simili (approccio *User-Based*) oppure degli item più simili (approccio *Item-Based*). Si distinguono quindi due modalità:

- **User-Based.** La predizione del rating che l'utente  $k$  assegnerebbe all'item  $i$  si ottiene come:

$$\hat{r}_{ki} = r_k + \frac{\sum_{v \in N_k^n(i)} sim(k, v) (r_{vi} - r_v)}{\sum_{v \in N_k^n(i)} sim(k, v)},$$

dove:

- $N_k^n(i)$  è l'insieme dei  $n$  utenti più simili a  $k$  che hanno valutato l'item  $i$ ;
- $r_v$  è il voto medio dell'utente  $v$ ;
- $sim(k, v)$  è la similarità tra gli utenti  $k$  e  $v$ .

- **Item-Based.** La predizione del rating che l'utente  $u$  assegnerebbe all'item  $i$  si ottiene come:

$$\hat{r}_{ui} = r_i + \frac{\sum_{j \in N_i^n(u)} sim(i, j) (r_{uj} - r_j)}{\sum_{j \in N_i^n(u)} sim(i, j)},$$

dove:

- $N_i^n(u)$  è l'insieme dei  $n$  item più simili a  $i$  valutati dall'utente  $u$ ;
- $r_j$  è il voto medio dell'item  $j$ ;
- $sim(i, j)$  è la similarità tra gli item  $i$  e  $j$ .

## 8.5 Sistemi di Raccomandazione Model-Based

I sistemi di raccomandazione *model-based* costruiscono un modello statistico in grado di apprendere schemi complessi a partire da un set di dati di training. Una delle tecniche più diffuse è la **fattorizzazione di matrici**, che consiste nell'approssimare la matrice

delle valutazioni  $R$ , di dimensione  $n_U \times n_V$ , tramite il prodotto di due matrici a bassa dimensionalità:

$$\hat{R} = U \cdot V.$$

In particolare:

- $U$  è la matrice degli utenti, di dimensione  $n_U \times d$ , dove  $d$  rappresenta il numero di **fattori latenti**. Ogni riga  $u_k = (u_{k1}, u_{k2}, \dots, u_{kd})$  rappresenta il profilo latente dell'utente  $k$ -esimo;
- $V$  è la matrice degli item, di dimensione  $d \times n_V$ . Ogni colonna  $v_i = (v_{1i}, v_{2i}, \dots, v_{di})^\top$  è un vettore di fattori latenti che descrive l'item  $i$ -esimo.

L'elemento  $\hat{r}_{ki}$  della matrice ricostruita è dato dal prodotto scalare tra il vettore latente dell'utente  $k$  e quello dell'item  $i$ :

$$\hat{r}_{ik} = u_k \cdot v_i.$$

Il vantaggio principale della fattorizzazione risiede nella riduzione della dimensionalità del problema. Mentre la matrice originale delle valutazioni  $R$  contiene  $n_U \times n_V$  elementi, la sua approssimazione tramite fattorizzazione richiede soltanto la memorizzazione delle due matrici  $U$  e  $V$ , il cui numero complessivo di parametri è pari a:

$$(n_U \cdot d) + (n_V \cdot d) = (n_U + n_V) d.$$

Per valori piccoli di  $d$  - tipicamente dell'ordine di poche decine o centinaia - questo numero risulta di gran lunga inferiore rispetto ai  $n_U \times n_V$  elementi della matrice originale.

### 8.5.1 Trovare $U$ e $V$ Ideali

L'obiettivo della fattorizzazione è trovare le matrici  $U$  e  $V$  che minimizzino la distanza quadratica tra la matrice delle valutazioni osservate  $R$  e la sua approssimazione  $\hat{R} = UV$ . In particolare, si desidera risolvere il seguente problema di ottimizzazione:

$$\min_{U,V} \sum_{(k,i) \in K} (r_{ki} - \hat{r}_{ki})^2 = \sum_{(k,i) \in K} (r_{ki} - u_k \cdot v_i)^2,$$

dove  $K$  è l'insieme delle coppie  $(i, k)$  per cui è presente una valutazione nota dell'utente  $k$  sull'item  $i$ . Il problema così formulato non è lineare congiuntamente in  $U$  e  $V$ , poiché il prodotto scalare  $u_k \cdot v_i$  contiene variabili moltiplicate tra loro. Inoltre, tale formulazione può soffrire di *overfitting*, poiché il numero di parametri liberi può risultare superiore al numero di dati osservati. Per mitigare questo problema si introduce una regolarizzazione, ottenendo il seguente problema di ottimizzazione:

$$\min_{U,V} \sum_{(k,i) \in K} (r_{ki} - u_k \cdot v_i)^2 + \lambda (\|U\|_F^2 + \|V\|_F^2),$$

dove:

- $\|U\|_F^2$  è il quadrato della **norma di Frobenius**, definita come

$$\|U\|_F^2 = \sum_{i,j} u_{ij}^2,$$

e analogamente per  $\|V\|_F^2$ ;

- $\lambda > 0$  è il parametro di regolarizzazione (tipicamente compreso tra 0.01 e 0.2), che controlla la complessità del modello favorendo valori più piccoli nei fattori latenti.

Esistono due metodi che permettono di linearizzare il problema, rendendolo risolubile in maniera semplice.

**Discesa del Gradiente Stocastico.** La **discesa del gradiente** (*gradient descent*) è un metodo iterativo che aggiorna i parametri del modello in direzione opposta al gradiente della funzione di errore. Ad ogni iterazione si aggiornano  $u_k$  e  $v_i$  nel seguente modo:

$$\begin{cases} u_k = u_k + 2\eta e_{ki} v_i, \\ v_i = v_i + 2\eta e_{ki} u_k, \end{cases} \quad (k, i) \in K$$

dove:

- $e_{ki}$  è l'errore commesso sulla coppia  $(k, i)$ :  $e_{ki} = r_{ki} - \hat{r}_{ki} = r_{ki} - (u_k \cdot v_i)$ ;
- $\eta \in (0, 0.02)$  è il **learning rate**. Impedisce che la soluzione vari troppo da un'iterazione e l'altra, in modo da evitare di "perdere" il minimo.

L'obiettivo è minimizzare l'errore quadratico  $e_{ki}^2$  attraverso aggiornamenti successivi nella direzione di massima variazione della funzione. Tale direzione è il gradiente di  $e_{ki}^2$ , che è un vettore di derivate parziali:

$$\nabla e_{ki}^2 = \left( \frac{\partial e_{ki}^2}{\partial u_k}, \frac{\partial e_{ki}^2}{\partial v_i} \right).$$

dove  $\nabla$  si legge "nabla", mentre, ad esempio,  $\partial e_{ki}^2 / \partial u_k$  rappresenta la derivata di  $e_{ki}^2$ .

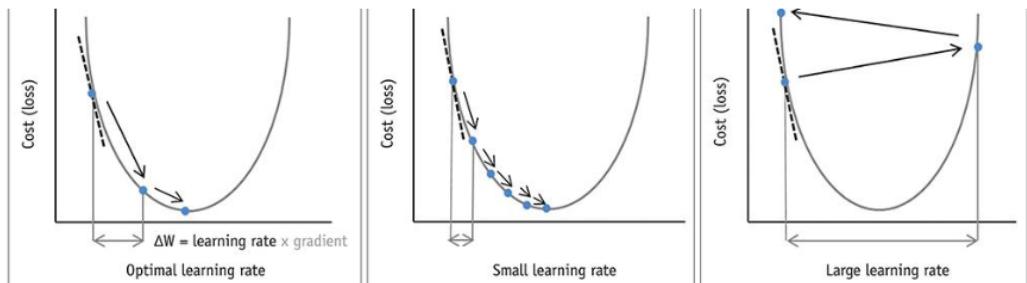


Figura 20: Esempio di Gradient Descent con diversi valori di learning rate.

**Minimi Quadrati Alternati.** Il metodo dei *minimi quadrati alternati* (ALS, *Alternating Least Squares*) è un metodo iterativo che alterna l'ottimizzazione dei fattori latenti degli utenti e degli item. Il procedimento può essere schematizzato come segue:

- Si inizializza la matrice  $V$  nel modo seguente:

$$V = \begin{pmatrix} \text{valutazioni medie dei prodotti} \\ \vdots \\ \text{piccoli valori casuali} \\ \vdots \end{pmatrix}$$

dove la prima riga è definita come  $V_{1,i} = v_i(1) = \text{media delle valutazioni dell'item } i$ .

- Fissata la matrice  $V$ , si determina la matrice  $U$  minimizzando  $\|R - \hat{R}\|^2$  (oppure la versione regolarizzata).
- Fissata la matrice  $U$ , si determina la matrice  $V$  minimizzando  $\|R - \hat{R}\|^2$  (oppure la versione regolarizzata).
- Si ripetono i punti 2 e 3 fino a raggiungere la convergenza.

### 8.5.2 Metodi Basati sulla Single Value Decomposition (SVD)

Data una matrice  $A$  reale di dimensione  $m \times n$ , esistono tre matrici  $U$ ,  $\Sigma$  e  $V$  tali che:

$$A = U\Sigma V^\top = u_1\sigma_1 v_1^\top + u_2\sigma_2 v_2^\top + \cdots + u_p\sigma_p v_p^\top, \quad p = \min(m, n)$$

dove:

- $U$  è una matrice ortogonale, cioè  $UU^\top = U^\top U = I$ , di dimensione  $m \times m$ ;
- $\Sigma$  è una matrice diagonale di dimensione  $m \times n$ . I suoi elementi diagonali, detti **valori singolari**, sono:

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p \geq 0, \quad p = \min(m, n)$$

- $V^\top$  è la trasposta di una matrice ortogonale  $V$  di dimensioni  $n \times n$ .

$m > n$	$m < n$
$A = [u_1 \ u_2 \cdots u_m] \begin{bmatrix} \sigma_1 & & & \\ & \ddots & & \\ & & \sigma_n & \\ & 0 & & \\ & \vdots & & \\ & 0 & & \end{bmatrix} \begin{bmatrix} v_1^\top \\ v_2^\top \\ \vdots \\ v_n^\top \end{bmatrix}$ $U \quad \Sigma \quad V^\top$ $m \times m \quad m \times n \quad n \times n$	$A = [u_1 \ u_2 \cdots u_m] \begin{bmatrix} \sigma_1 & & & & & \\ & \ddots & & & & \\ & & \sigma_m & & & \\ & & & 0 & & \\ & & & & \ddots & \\ & & & & & 0 \end{bmatrix} \begin{bmatrix} v_1^\top \\ v_2^\top \\ \vdots \\ v_n^\top \end{bmatrix}$ $U \quad \Sigma \quad V^\top$ $m \times m \quad m \times n \quad n \times n$

Figura 21: Due esempi di matrice  $A$  di dimensione  $m \times n$ : nel primo caso  $m > n$ , mentre nel secondo  $m < n$ .

**Proprietà.** La matrice  $A$  presenta le seguenti proprietà:

- I valori singolari sono unici;

2. Le matrici  $U$  e  $V$  non sono univocamente determinate;
3. Sia  $A = U\Sigma V^\top$  e sia  $r$  il numero dei suoi valori singolari non nulli. Allora

$$A = U\Sigma V^\top = U_r \Sigma_r V_r^\top,$$

dove  $U_r$  e  $V_r$  contengono rispettivamente le prime  $r$  colonne di  $U$  e  $V$  e  $\Sigma_r$  è la corrispondente matrice diagonale dei valori singolari non nulli. Tale matrice ha rango  $r^4$ .

Dalle proprietà 1 e 2 segue che la decomposizione a valori singolari non è unica: infatti, pur essendo i valori singolari determinati in modo univoco, le matrici ortogonali  $U$  e  $V$  possono essere scelte in molteplici modi. Per ottenere una SVD numerica si adottano quindi convenzioni e criteri computazionali che fissano univocamente tali scelte.

**SVD Troncata.** Una SVD troncata è l'approssimazione della matrice di partenza con una di rango inferiore, ottenuta utilizzando un numero minore di valori singolari.

**Theorem 8.1** (Eckart-Young). Sia  $A \in \mathbb{R}^{m \times n}$  e sia  $A = U\Sigma V^\top$  la sua decomposizione ai valori singolari, con

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r \geq \sigma_{r+1} = \cdots = \sigma_{\min(m,n)} = 0.$$

Dato un intero  $k \leq r$ , si definisca

$$A_k = U_k \Sigma_k V_k^\top,$$

dove  $\Sigma_k$  contiene i primi  $k$  valori singolari e  $U_k$ ,  $V_k$  le corrispondenti colonne di  $U$  e  $V$ . Allora  $A_k$  è la matrice di rango  $k$  più vicina ad  $A$  in norma 2 (e anche in norma di Frobenius), cioè

$$\|A - A_k\|_2 = \min_{\text{rank}(B)=k} \|A - B\|_2 = \sigma_{k+1}.$$

Il valore singolare  $\sigma_{k+1}$  rappresenta quindi l'errore minimo commesso approssimando  $A$  con la sua SVD troncata  $A_k$ .

**SVD per la Fattorizzazione della Matrice di Rating.** Utilizza la decomposizione a valori singolari per fattorizzare la matrice di rating:

$$R = USV^\top$$

Nei punti in cui  $R$  non è nota, ossia non c'è valutazione, di solito si pone  $r_{ki} = 0$ . La dimensione del problema non è diminuita, quindi si usa la SVD troncata:

$$\hat{R} = \hat{U}\hat{S}\hat{V}^\top$$

---

<sup>4</sup>Il rango di una matrice è il numero di righe o colonne linearmente indipendenti.

dove:

- $\hat{U}$ : prime  $f$  colonne di  $U$ ;
- $\hat{V}^\top$  prime  $f$  righe di  $V^\top$ ;
- $\hat{S}$  contiene gli  $f$  valori singolari più grandi.

La matrice ottenuta tramite l'applicazione della SVD troncata è molto meno complessa della matrice di rating di partenza, e ci permette di calcolare velocemente la vicinanza di un oggetto/servizio da raccomandare ad un utente. Quando utilizziamo la SVD troncata, approssimiamo la matrice di rating osservando solo le caratteristiche più importanti, ossia quelle corrispondenti ai valori singolari più grandi.

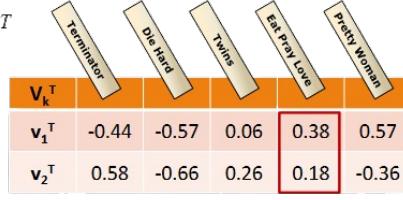
<ul style="list-style-type: none"> <li>• SVD: <math>M_k = U_k \times \Sigma_k \times V_k^\top</math></li> </ul> <table border="1" style="margin-top: 10px; border-collapse: collapse;"> <thead> <tr> <th><math>U_k</math></th> <th><math>u_1</math></th> <th><math>u_2</math></th> </tr> </thead> <tbody> <tr> <td>Alice</td> <td>0.47</td> <td>-0.30</td> </tr> <tr> <td>Bob</td> <td>-0.44</td> <td>0.23</td> </tr> <tr> <td>Mary</td> <td>0.70</td> <td>-0.06</td> </tr> <tr> <td>Sue</td> <td>0.31</td> <td>0.93</td> </tr> </tbody> </table> <p><b>Previsione:</b> <math>\hat{r}_{ui} = \bar{r}_u + U_k(Alice) \times \Sigma_k \times V_k^\top(EPL)</math>  <math>= 3 + (0.47 \times 5.63 \times 0.38 - 0.3 \times 3.23 \times 0.18)</math>  <math>= 3 + 0.84 = 3.84</math></p>	$U_k$	$u_1$	$u_2$	Alice	0.47	-0.30	Bob	-0.44	0.23	Mary	0.70	-0.06	Sue	0.31	0.93	 <table border="1" style="margin-top: 10px; border-collapse: collapse;"> <thead> <tr> <th><math>V_k^\top</math></th> <th>Terminator</th> <th>Die Hard</th> <th>Twins</th> <th>Eat Pray Love</th> <th>Pretty Woman</th> </tr> </thead> <tbody> <tr> <td><math>v_1^\top</math></td> <td>-0.44</td> <td>-0.57</td> <td>0.06</td> <td>0.38</td> <td>0.57</td> </tr> <tr> <td><math>v_2^\top</math></td> <td>0.58</td> <td>-0.66</td> <td>0.26</td> <td>0.18</td> <td>-0.36</td> </tr> </tbody> </table> <table border="1" style="margin-top: 10px; border-collapse: collapse;"> <thead> <tr> <th><math>\Sigma_k</math></th> <th>Dim1</th> <th>Dim2</th> </tr> </thead> <tbody> <tr> <td>Dim1</td> <td>5.63</td> <td>0</td> </tr> <tr> <td>Dim2</td> <td>0</td> <td>3.23</td> </tr> </tbody> </table>	$V_k^\top$	Terminator	Die Hard	Twins	Eat Pray Love	Pretty Woman	$v_1^\top$	-0.44	-0.57	0.06	0.38	0.57	$v_2^\top$	0.58	-0.66	0.26	0.18	-0.36	$\Sigma_k$	Dim1	Dim2	Dim1	5.63	0	Dim2	0	3.23
$U_k$	$u_1$	$u_2$																																									
Alice	0.47	-0.30																																									
Bob	-0.44	0.23																																									
Mary	0.70	-0.06																																									
Sue	0.31	0.93																																									
$V_k^\top$	Terminator	Die Hard	Twins	Eat Pray Love	Pretty Woman																																						
$v_1^\top$	-0.44	-0.57	0.06	0.38	0.57																																						
$v_2^\top$	0.58	-0.66	0.26	0.18	-0.36																																						
$\Sigma_k$	Dim1	Dim2																																									
Dim1	5.63	0																																									
Dim2	0	3.23																																									

Figura 22: Nell'esempio seguente, calcoliamo  $U$ ,  $S$  e  $V$ , ma conserviamo solo le due caratteristiche (fattori) principali, prendendo solo le prime due colonne di  $U$  e di  $V$ .

**Osservazioni sulla Riduzione della Dimensionalità del Problema.** Quando si fattorizza una matrice, si ottiene un'approssimazione a rango basso che permette di identificare fattori latenti e di proiettare utenti e prodotti in uno spazio  $k$ -dimensionale. La riduzione della dimensionalità influisce in vari modi sul problema originale:

- la qualità della previsione può diminuire, poiché alcune informazioni contenute nelle valutazioni originali vengono perse;
- le previsioni possono al contrario migliorare, poiché la riduzione di dimensionalità filtra il rumore e può mettere in evidenza correlazioni latenti nei dati.

La scelta del numero di dimensioni latenti (ossia del numero di valori singolari mantenuti nell'approccio SVD) è cruciale e va affinata sperimentalmente in funzione del dominio applicativo. In letteratura diversi autori suggeriscono di considerare tra 20 e 100 fattori per ottenere buone stime. In generale, la computazione della SVD completa presenta una complessità di  $O(nm^2)$  se  $m < n$ , oppure di  $O(n^2m)$  nel caso contrario. Tuttavia, il costo può ridursi sensibilmente quando si vogliono calcolare soltanto alcuni valori singolari (ad esempio i primi  $k$ ), oppure quando la matrice è sparsa.

Per la scelta di  $k$  si usa la **regola del pollice**, cioè si conserva solo l'80% o il 90% dell'energia totale del sistema:

$$\frac{\sum_{j=1}^k \sigma_j}{\sum_{j=1}^n \sigma_j} \approx 0.80 - 0.90$$

# Laboratorio

## Lab 1 MPI

In un'architettura MIMD a memoria distribuita ogni processore ha una propria memoria locale alla quale accede direttamente e può, inoltre, conoscere i dati in memoria di un altro processore o far conoscere i propri attraverso il trasferimento di dati.

*Message Passing Interface* (MPI) è un paradigma per la progettazione di algoritmi paralleli basato su processi concorrenti che coordinano la propria attività attraverso lo scambio di messaggi. La libreria MPI fornisce sia **funzioni di gestione della comunicazione**, responsabili della definizione e dell'identificazione dei gruppi di processi (task) coinvolti nella comunicazione e dell'associazione di un'identità a ciascun processo all'interno del gruppo, sia **funzioni di trasferimento dei dati**, che permettono di inviare o ricevere messaggi da e verso singoli processi o insiemi di processi.

### Lab 1.1 Processi

Il **processo** è l'unità fondamentale di un programma parallelo, costituito da una collezione di processi (task) autonomi. Ogni processo esegue un proprio algoritmo utilizzando i dati residenti nella propria memoria e, quando necessario, comunica con gli altri processi attraverso uno scambio di messaggi. In ambiente MPI un programma è visto come un insieme di componenti (o processi) concorrenti.



Figura 23: In questo esempio, il sistema lancia  $n$  processi. Ogni processo esegue una copia di `prog.exe` ed ha il suo spazio di memoria locale.

### Lab 1.2 Contesto

In MPI, l'esecuzione concorrente è organizzata in gruppi di processi, ciascuno dei quali è incluso logicamente in un **contesto** (o *communicator*). Un communicator definisce sia l'insieme dei processi che possono comunicare tra loro, sia le regole e gli attributi relativi a tale comunicazione. Due processi possono scambiarsi messaggi solo se appartengono allo stesso communicator. All'interno di un communicator, a ogni processo è assegnato un identificativo univoco, detto **rank**, rappresentato da un numero intero compreso tra 0 e  $n-1$ , dove  $n$  indica il numero totale di processi nel communicator stesso. Un programma MPI può definire molteplici communicator, e un processo può appartenere contemporaneamente a più di essi. All'avvio dell'applicazione, tutti i processi sono automaticamente inclusi in un communicator globale predefinito, chiamato `MPI_COMM_WORLD`.

### Lab 1.3 Funzioni di MPI

MPI è una libreria che mette a disposizione un insieme esteso di primitive per la gestione del parallelismo basato su scambio di messaggi. In particolare, le funzioni possono essere suddivise nelle seguenti categorie principali:

- **Funzioni di gestione dell'ambiente:** includono routine di base che si occupano di inizializzazione, finalizzazione e raccolta informazioni sull'ambiente di esecuzione;
- **Funzioni di comunicazione:** possono essere di tipo *uno a uno* o *collettiva*;
- **Funzioni per la sincronizzazione:** includono operazioni di barriera e calcolo del tempo.

Tutte le funzioni della libreria MPI iniziano con il prefisso `MPI_`. Per utilizzare MPI in un programma C è necessario includere l'header `<mpi.h>`.

Tutte le routine MPI restituiscono un indicatore di errore `error = MPI_Function(parametri,...)`. Il valore ritornato indica l'esito della chiamata. Tra i principali codici di stato:

- `MPI_SUCCESS`: l'operazione è stata completata correttamente;
- `MPI_ERR_ARG`: parametro o argomento non valido;
- `MPI_ERR_RANK`: rank non valido nel communicator specificato;
- `MPI_ERR_COMM`: communicator non valido;
- `MPI_ERR_COUNT`: numero di elementi non corretto;
- `MPI_ERR_TYPE`: datatype non coerente o non valido.

#### Lab 1.3.1 Routine di Base

In questa sezione vengono introdotte le routine fondamentali che costituiscono la struttura di un programma MPI. Tali funzioni consentono di inizializzare e terminare l'ambiente di esecuzione parallela, e di ottenere informazioni sul contesto di comunicazione e sull'identità del processo:

- **`MPI_Init(&argc, &argv)`:** questa routine deve essere chiamata prima di ogni altra routine MPI. Inizializza l'ambiente di esecuzione di MPI, definisce l'insieme dei processi attivati (contesto) e inizializza il contesto `MPI_COMM_WORLD`. Argc e argv sono gli argomenti del main.
- **`MPI_Comm_rank(comm, &me)`:** routine che assegna ad ogni processo di un dato contesto un proprio numero identificativo.
  - Input: `MPI_Comm comm`, il contesto a cui appartiene il processo;
  - Output: identificativo assegnato al processo, salvato nella variabile `int me`.
- **`MPI_Comm_size(comm, &nproc)`:** routine che restituisce ad ogni processo il numero totale di processi del contesto.
  - Input: `MPI_Comm comm`, il contesto a cui appartiene il processo;
  - Output: numero di processi del contesto, salvato nella variabile `int nproc`.
- **`MPI_Finalize()`:** routine che determina la fine del programma MPI. Dopo questa routine non è possibile chiamare nessun'altra routine MPI.

```

1 #include <stdio.h>
2 #include <mpi.h>
3
4 main(int argc, char *argv[]){
5     int me, nproc;
6     //ulteriori dichiarazioni di variabili
7
8     MPI_Init(&argc,&argv);
9     MPI_Comm_size (MPI_COMM_WORLD,&nproc);
10    MPI_Comm_rank (MPI_COMM_WORLD, &me);
11
12    /* corpo del programma */
13
14    MPI_Finalize();
15    return 0;
16 }
```

Codice 9: Esempio di utilizzo delle Routine di Base di MPI.

### Lab 1.3.2 Funzioni di Comunicazione

Un messaggio è essenzialmente un blocco di dati da trasferire tra i processi. La comunicazione di un messaggio può coinvolgere due o più processori: per comunicazioni che coinvolgono solo due processori si considerano funzioni MPI per comunicazioni uno a uno, mentre per comunicazioni che coinvolgono più processori si considerano funzioni MPI per comunicazioni collettive.

**Comunicazioni uno a uno.** Include le seguenti due routine:

- **MPI\_Send(msg, count, datatype, dest, tag, comm):** routine che si occupa dell'invio di un messaggio a un secondo processore. Quando termina non è detto che il messaggio sia stato effettivamente consegnato. Avrà i seguenti parametri:
  - **void \*msg:** indirizzo del primo elemento da spedire;
  - **int count:** numero di elementi da spedire;
  - **MPI\_Datatype datatype:** tipo degli elementi da spedire, come ad esempio: MPI\_INT per int, MPI\_FLOAT per i float, MPI\_DOUBLE per i double e MPI\_CHAR per i char;
  - **int dest:** identificativo del destinatario del messaggio;
  - **int tag:** identificativo del messaggio;
  - **MPI\_Comm comm:** contesto a cui appartengono i processi.
- **MPI\_Recv(msg, count, datatype, source, tag, comm, &status):** routine che si occupa della ricezione di un messaggio. Termina quando il messaggio è stato effettivamente ricevuto. Avrà i seguenti parametri:
  - **void \*msg:** indirizzo del primo elemento da ricevere;
  - **int count:** numero di elementi da ricevere. Devono essere consecutivi in memoria;

- `MPI_Datatype datatype`: tipo degli elementi da ricevere;
  - `int source`: identificativo del mittente del messaggio. Può assumere il valore `MPI_ANY_SOURCE` per ricevere da qualsiasi mittente;
  - `int tag`: identificativo del messaggio. Può assumere il valore `MPI_ANY_TAG` per ricevere qualsiasi messaggio;
  - `MPI_Comm comm`: contesto a cui appartengono i processi;
  - `MPI_Status status`: valore che permette di conoscere il mittente (se nascosto in `MPI_ANY_SOURCE`), il tag e la dimensione esatta di un messaggio.
- **`MPI_Sendrecv(sendmsg, sendcount, sendtype, dest, sendtag, recvmsg, recvcount, recvtype, source, recvtag, comm, &status)`**: questa routine si occupa di spedizione e ricezione con un'unica chiamata. Il messaggio da spedire (`sendmsg`) e da ricevere (`recvmsg`) devono essere distinti.

La modalità di spedizione bloccante di un messaggio può essere:

- **sincrona**: il processo mittente sospende l'esecuzione finché il messaggio spedito non viene recepito dal destinatario;
- **bufferizzata**: il processo mittente deposita il messaggio in un buffer (memoria tamponcino) e prosegue l'esecuzione;

Il funzionamento di default dipende dalla libreria utilizzata. Esistono altre funzioni di spedizione/ricezione, come Send/Recv sincrone, buffered, non bloccanti etc.

**Comunicazioni Collettive.** Includono le seguenti routine principali:

- **`MPI_Bcast(msg, count, datatype, root, comm)`**: esegue una comunicazione di tipo **Broadcast**, ovvero un processo `root` invia lo stesso messaggio a tutti gli altri processi del communicator `comm`.
- **`MPI_Scatter(sendmsg, sendcount, sendtype, recvmsg, recvcount, recvtype, root, comm)`**: esegue una comunicazione di tipo **Scatter**, in cui il processo `root` distribuisce porzioni differenti del proprio buffer a ciascun processo del communicator.
- **`MPI_Gather(sendmsg, sendcount, sendtype, recvmsg, recvcount, recvtype, root, comm)`**: esegue una comunicazione di tipo **Gather**, in cui ogni processo invia un proprio messaggio al processo `root`, che li raccoglie in un array ordinato secondo i rank dei mittenti. Anche `root` contribuisce con il proprio dato.
- **`MPI_Allgather(sendmsg, sendcount, sendtype, recvmsg, recvcount, recvtype, comm)`**: realizza un'operazione di tipo **Allgather**, in cui ogni processo invia un proprio messaggio a tutti gli altri processi del communicator. Non è previsto alcun parametro `root`, poiché tutti i processi svolgono ruolo sia di mittenti che di destinatari. I dati raccolti vengono memorizzati in ordine crescente di rank dei processi mittenti.

**Operazioni Collettive.** Queste routine permettono di combinare valori provenienti da tutti i processi del communicator applicando un’operazione di *riduzione* (ad esempio somma o massimo). Il risultato dell’operazione può essere memorizzato su un unico processo **root** mediante la funzione **MPI\_Reduce**, oppure distribuito a tutti i processi mediante **MPI\_Allreduce**.

- **MPI\_Reduce(sendmsg, recvmsg, count, type, op, root, comm)**: raccoglie i valori inviati da tutti i processi, applica l’operazione **op** e memorizza il risultato nel processo **root**.
- **MPI\_Allreduce(sendmsg, recvmsg, count, type, op, comm)**: funzione analoga a **MPI\_Reduce**, ma il risultato della riduzione viene reso disponibile a ogni processo del communicator.

Il parametro **op** indica l’operazione da applicare ai valori e può assumere uno tra i seguenti valori predefiniti: **MPI\_MAX** (massimo), **MPI\_MIN** (minimo), **MPI\_SUM** (somma), **MPI\_PROD** (prodotto), **MPI\_MAXLOC** (max con posizione), **MPI\_MINLOC** (min con posizione).

### Lab 1.3.3 Funzioni di Sincronizzazione

Talvolta occorre sincronizzare tutti i processi prima che essi proseguano la computazione, ad esempio quando si devono leggere dei dati di input.

- **MPI\_Barrier(comm)**: blocca ogni processo del contesto finché tutti i processi del contesto **comm** non hanno richiamato **MPI\_Barrier**.
- **MPI\_Wtime()**: restituisce il tempo attuale, in secondi, come double. Le operazioni di I/O solito non vengono incluse ai fini del tempo di calcolo del processore.

```

1 double start_time, end_time, max_time;
2
3 // tempo di inizio del calcolo del tempo
4 start_time = MPI_Wtime();
5
6 // tempo di fine del calcolo del tempo
7 end_time = MPI_Wtime();
8
9 // tempo impiegato per l'esecuzione del programma
10 max_time = end_time - start_time();

```

Codice 10: Snippet per il calcolo del tempo di esecuzione di un programma.

### Lab 1.3.4 Shift

L’operazione di *shift* consiste nello spostamento dei bit di una rappresentazione binaria verso sinistra o verso destra, inserendo opportuni valori nelle posizioni che rimangono libere. Essa è equivalente ad una moltiplicazione o divisione per la base della rappresentazione numerica: in base 10 lo shift equivale a moltiplicare (a sinistra) o dividere (a destra) per 10, mentre in base 2 equivale a moltiplicare o dividere per 2. In particolare, nello shift in base 2 valgono le seguenti regole:

- **Shift a sinistra (left shift)**: equivale ad una moltiplicazione per due (o potenze di due), poiché ogni bit viene spostato verso posizioni di peso maggiore. Viene inserito uno **0** nella posizione del *least significant bit* (LSB).
  - **Shift a destra (right shift)**: equivale ad una divisione intera per due (o potenze di due), con spostamento verso posizioni di peso minore. Viene inserito uno **0** nella posizione del *most significant bit* (MSB). Può verificarsi **troncamento**, poiché il bit LSB viene eliminato durante lo spostamento.

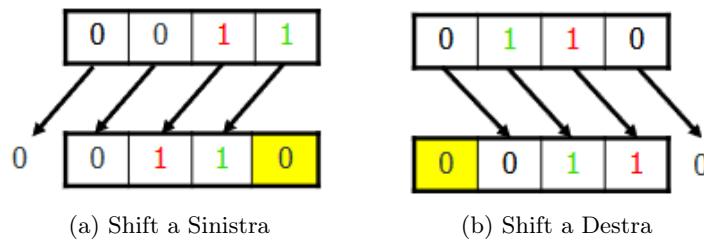


Figura 24: Esempi di Shift.

```
// sia n potenza di 2
q=0;
while(n!=1){
    // shift di un bit a destra
    n=n>>1;
    q++;
}
```

Codice 11: Snippet per il calcolo dello shift in base due. Utile per il calcolo di un logaritmo in base due di un intero.

## Lab 2 Topologie di Processi

Prima di cominciare, risulta importante chiarire la differenza tra contesto e topologia:

- **Contesto (Communicator)**: identifica un insieme di processi che possono comunicare tra loro, definendo un ambiente di comunicazione isolato dagli altri;
- **Topologia**: impone una struttura ai processi di un contesto, al fine di guidare schemi di comunicazione specifici. La disposizione è puramente *virtuale*, senza necessariamente riflettere quella fisica dell'hardware.

La topologia di comunicazione rappresenta lo schema logico secondo cui avvengono gli scambi di messaggi tra processi. L'uso di topologie virtuali consente di semplificare la progettazione delle comunicazioni, ridurre la probabilità di errori e sfruttare più facilmente pattern ricorrenti quando il flusso di comunicazione segue una struttura definita.

Se consideriamo il caso del prodotto matrice/vettore, risulta utile disporre i processori in una griglia bidimensionale, cioè lungo le righe e le colonne della matrice, così da semplificare le comunicazioni. Il dominio computazionale viene quindi suddiviso in blocchi. Questo tipo di suddivisione è vantaggioso perché:

- rispecchia la geometria del problema, rendendo più naturale l'assegnazione dei dati ai processi;
- permette una migliore granularità nella distribuzione del lavoro;
- riduce la quantità di dati da comunicare tra i processi.

D'altro canto, la gestione iniziale della griglia richiede una maggiore attenzione nella definizione delle comunicazioni e nell'organizzazione dei dati.

### Lab 2.1 Tipi di Topologie

L'utilizzo di una topologia per la progettazione di un algoritmo in ambiente MIMD è spesso legata alla geometria "intrinseca" del problema in esame.

**Topologia Lineare.** Utilizzata di default da MPI: viene assegnato a ciascun processo presente in un determinato contesto un identificativo da 0 a  $n - 1$ .



Figura 25: Esempio di Topologia Lineare.

**Topologia Cartesiana.** Nella topologia cartesiana ogni processo è identificato da coordinate cartesiane ed è connesso ai vicini tramite una griglia virtuale. Ai bordi può essere presente o meno periodicità. Si distinguono i seguenti tipi:

- **Topologia Cartesiana Monodimensionale (1D)**: un esempio è la *circular shift*, che realizza una topologia ad anello;

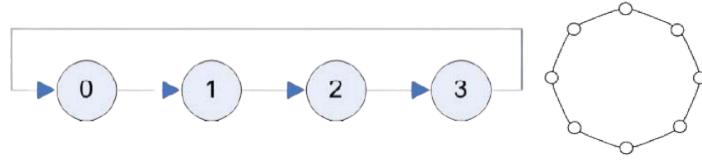


Figura 26: Esempio di Topologia Cartesiana 1D (ad Anello).

- **Topologia Cartesiana Bidimensionale (2D)**: ad ogni processo è associata una coppia di indici, che rappresentano le coordinate nello spazio 2D. In una Topologia Cartesiana si verifica **periodicità** quando l'ultimo processo di una riga comunica con il primo e viceversa, e lo stesso avviene sulle colonne.

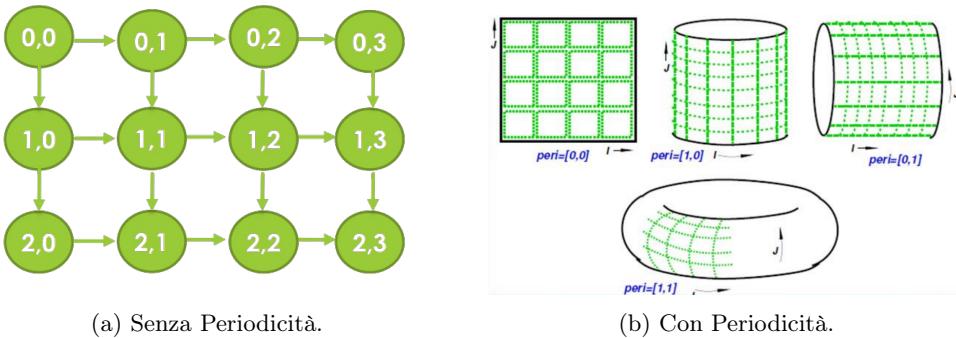


Figura 27: Esempio di Topologia Cartesiana 2D senza e con periodicità sulle righe, sulle colonne e su entrambe.

## Lab 2.2 Sottogriglie

Spesso si vuole operare su una parte di una topologia: in questo caso si divide la topologia in sottotopologie che formano griglie di dimensioni minori. Ogni sottogriglia genera un nuovo contesto in cui eseguire operazioni collettive. Il numero di sottotopologie è il prodotto del numero di processi relativi alle dimensioni eliminate. Le sottogriglie hanno dimensione pari

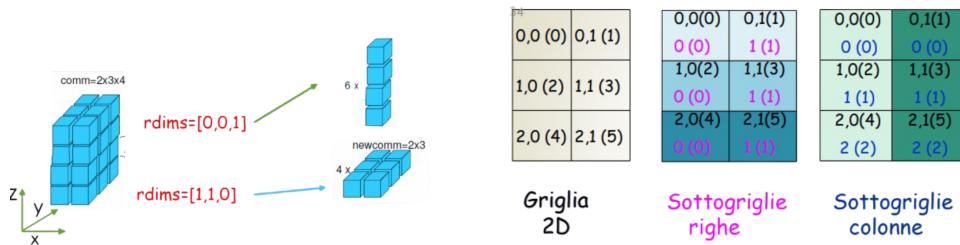


Figura 28: Esempio di Sottotopologia.

alla dimensione della griglia di origine, lungo la direzione considerata. Ogni sottogriglia è un contesto, che eredita le proprietà della griglia cartesiana; in particolare è ancora una griglia cartesiana, quindi ogni processo ha sia id che coordinate. In più acquisisce un nuovo identificativo in base alla sua posizione nella sottogriglia.

## Lab 2.3 Funzioni

**Topologia Cartesiana.** Di seguito vengono presentate le funzioni principali per la creazione e gestione delle topologie cartesiane in MPI:

- **MPI\_Cart\_Create(comm\_old, ndims, dims, periods, reorder, comm\_grid):** crea una topologia cartesiana associata al communicator `comm_old`. La nuova topologia viene definita specificando il numero di dimensioni, la loro estensione e l'eventuale periodicità. I parametri sono:
  - `comm_old`: communicator di input;
  - `ndims`: numero di dimensioni della griglia cartesiana;
  - `dims`: array di lunghezza `ndims`. Il valore `dims[i]` indica il numero di processi lungo la  $i$ -esima dimensione;
  - `periods`: array di lunghezza `ndims`. Se `periods[i] = 1`, la griglia è periodica lungo la  $i$ -esima dimensione, altrimenti (0) non lo è. Se imponiamo la periodicità, ogni riferimento prima della prima o dopo l'ultima componente di ogni riga/colonna si riavvolgerà ciclicamente. Se non c'è periodicità, ogni riferimento all'infuori del range definito, produrrà un identificativo negativo (`MPI_PROC_NULL`, pari a -1);
  - `reorder`: se diverso da zero, MPI può riordinare i rank dei processi per ottimizzare il mapping sulla macchina;
  - `comm_grid`: communicator risultante, strutturato con topologia cartesiana.
- **MPI\_Cart\_coords(MPI\_Comm comm\_grid, int menum\_grid, int dim, int \*coordinate):** operazione collettiva che restituisce a ciascun processo di `comm_grid` con identificativo `menum_grid`, le sue coordinate all'interno della griglia predefinita. `*coordinate` è un array di dimensione `dim`, i cui elementi rappresentano le coordinate del processo all'interno della griglia, output.
- **MPI\_Cart\_rank(MPI\_Comm comm\_grid, int icoords, int rank):** operazione collettiva che restituisce a ciascun processo di `comm_grid`, date le coordinate `icoords`, l'identificativo `rank` associato all'interno della griglia predefinita.

**Sottogriglia.** Di seguito verrà proposta la principale funzione per le sottogrigli:

- **MPI\_Cart\_sub(MPI\_Comm comm\_grid, int \*rdims, MPI\_Comm \*new\_griglia):** funzione per la creazione di una sottogriglia, dove:
  - `comm_grid`: contesto della topologia;
  - `*rdims`: vettore di dimensione `dim` (dimensione della topologia). Se `rdims[i]=1`, la  $i$ -ma dimensione varia; se `rdims[i]=0`, la  $i$ -ma dimensione è fissata.
  - `new_griglia`: nuovo contesto della sottogriglia.

## Lab 3 Compilazione e Valutazione di Programmi CUDA

Ogni file sorgente contenente estensioni CUDA deve essere compilato con un compilatore CUDA compliant: `nvcc` per CUDA C (NVIDIA). Il compilatore processa il sorgente separando codice device dall'host, di cui:

- il codice host viene rediretto a un compilatore standard di default (ad esempio `gcc`);
- il codice device viene tradotto in **PTX**.

A partire dal PTX prodotto è possibile produrre codice oggetto binario (**cubin**) specializzato per una particolare architettura GPU, oppure un eseguibile che include PTX e/o codice binario (**cubin**). Quando si specifica un'architettura virtuale, `nvcc` posticipa la fase di assemblaggio del codice PTX all'esecuzione dell'applicazione, quando l'architettura reale è nota. Ad esempio, il comando seguente genera un codice binario che funziona perfettamente se lanciato su architetture con compute capability 5.0 o successive.

```
1 nvcc x.cu --gpu-architecture=compute_50 --gpu-code=compute_50
```

Codice 12: Snippet per la generazione di codice binario.

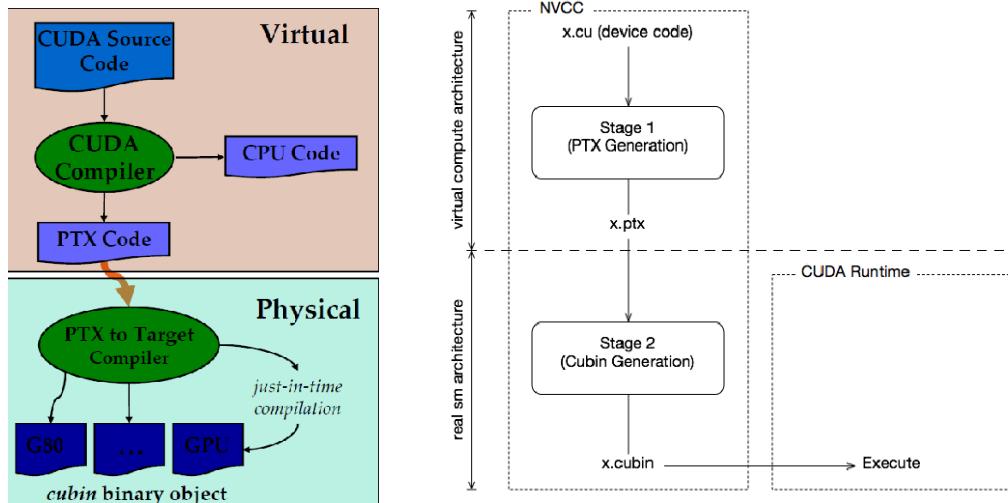


Figura 29: Panoramica del processo di compilazione CUDA: il codice sorgente .cu viene prima tradotto in PTX, un'architettura virtuale indipendente dall'hardware. Successivamente, il PTX viene compilato in un file .cubin specifico per l'architettura reale della GPU tramite una fase di compilazione a bersaglio (*PTX-to-Target*) che può avvenire staticamente o tramite *just-in-time* al momento dell'esecuzione. Le immagini mostrano sia la distinzione tra livello virtuale e fisico, sia la suddivisione in due stadi effettuata da `nvcc`.

### Lab 3.1 Specifiche di Compilazione

Al compilatore va sempre specificata:

- l'**architettura virtuale** con cui generare il PTX code, dove l'architettura virtuale è un'indicazione della compute capability<sup>5</sup> che deve avere l'architettura reale su cui

<sup>5</sup><https://developer.nvidia.com/cuda-gpus> per conoscere la compute capability della propria GPU.

andrà eseguito il codice. Richiedere un'architettura virtuale meno performante, lascia più scelta per l'architettura reale su cui potrà essere eseguito il codice;

- **l'architettura reale** per creare il codice oggetto (cubin), che dovrebbe essere scelta quale la migliore possibile. Ovviamente ciò è possibile solo se è nota l'architettura fisica su cui sarà eseguito il codice.

Per chi usa Google Colab, quando si specifica l'architettura reale o virtuale, indicarne una con compute capability 3.0 o successiva.

```

1 // Forma estesa per la compilazione
2 nvcc -arch=compute_30 -code=sm_30, sm_30
3
4 // Forma abbreviata corrispondente per la compilazione
5 nvcc -arch=sm_30
6
7 // Forma per Google Colab per la compilazione (doppio trattino)
8 nvcc --arch=sm_30
9
10 // Versione di CUDA
11 nvcc -version // oppure
12 nvcc -V
13
14 // Informazioni diagnostiche (NVIDIA System Management Interface)
15 nvidia-smi

```

Codice 13: Comandi utili.

**Altre Informazioni.** In compilazione, si può specificare **una sola** architettura virtuale e **una lista** di architetture reali.

## Lab 3.2 Misura dei Tempi: gli Eventi

Gli eventi sono una sorta dimarcatori che possono essere utilizzati nel codice per:

- misurare il tempo trascorso (*elapsed*) durante l'esecuzione di chiamate CUDA (precisione a livello di ciclo di clock);
- bloccare la CPU fino a quando le chiamate CUDA precedenti l'evento non siano state completate.

```

1 cudaEvent_t start, stop;
2 cudaEventCreate(&start);
3 cudaEventCreate(&stop);
4
5 // ...
6
7 cudaEventRecord(start);
8 kernel<<<grid, block>>>(...);
9 cudaEventRecord(stop);
10 // assicura che tutti i thread siano arrivati all'evento stop prima di registrare il tempo
11 cudaEventSynchronize(stop);
12
13 float elapsed; // tempo tra i due eventi in millisecondi

```

```

14 cudaEventElapsedTime(&elapsed,start, stop);
15
16 // ...
17
18 cudaEventDestroy(start);
19 cudaEventDestroy(stop);

```

Codice 14: Codice per il calcolo del tempo attraverso gli eventi.

### Lab 3.3 Ottimizzazione Mediante Profiling

Lo sviluppo di applicazioni HPC implica due passi principali:

1. Sviluppare codice corretto;
2. Migliorare il codice per elevare le performance.

*Profiling* significa analizzare le prestazioni del programma misurando la complessità in spazio e/o in tempo dell'algoritmo e la frequenza e la durata delle chiamate a funzione.

In generale, in CUDA un'implementazione naïve non dà alte prestazioni, ma i tool di profiling aiutano a trovare colli di bottiglia da rimuovere. Ad esempio `nvprof ./programma`, disponibile da CUDA 5.0 in poi, permette di valutare le singole chiamate a CUDA in termini di tempo. Per le versioni di CUDA precedenti alla 5.0, è necessario impostare `% export COMPUTE_PROFILE=1` e poi eseguire il programma normalmente. Verrà creato il file `cuda_profile_0.log`, contenente le informazioni sui tempi di trasferimento dati in microsecondi.

```

1 # CUDA_PROFILE_LOG_VERSION 2.0
2 # CUDA_DEVICE 0 Tesla C2050 / C2070
3 # TIMESTAMPFACTOR fffff545f316c5a0
4 method,gputime,cputime,occupancy
5 method=[ memcpyHtoD ] gputime=[ 159.744 ] cputime=[ 1180.000 ]
6 method=[ memcpyHtoD ] gputime=[ 159.680 ] cputime=[ 1078.000 ]
7 method=[ memset32_aligned1D ] gputime=[ 8.224 ] cputime=[ 20.000 ] occupancy=[ 1.000 ]
8 method=[ _Z27prodottoArrayCompPerCompGPUPiS_S_i ] gputime=[ 17.408 ]cputime=[ 26.000 ]
    occupancy=[ 0.667 ]
9 method=[ memcpyDtoH ] gputime=[ 127.584 ] cputime=[ 863.000 ]

```

Codice 15: Esempio di file di log.

## Lab 4 Riduzione, Sincronizzazione e Allocazione

### Lab 4.1 Riduzione

Nel calcolo parallelo capita frequentemente di dover *ridurre* un insieme di valori, calcolati in parallelo, a un singolo risultato finale. Le operazioni di riduzione più comuni sono: somma (ADD), prodotto (MUL), ricerca del minimo (MIN) e del massimo (MAX). In Figura 30 è mostrato un confronto tra riduzione sequenziale e riduzione parallela. Se volessimo applicare

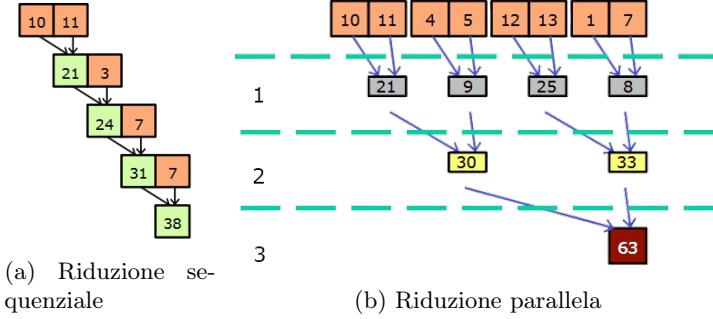


Figura 30: Confronto tra riduzione sequenziale e parallela. La riduzione sequenziale ha complessità  $O(N)$ , mentre la riduzione parallela ha complessità  $O(\log_2 N)$ .

la stessa strategia di riduzione vista nella *Strategia II della Somma in Parallello* (Sezione 2.1), otterremmo però prestazioni molto deludenti: non solo introdurremmo codice altamente divergente, ma produrremmo anche conflitti tra i thread nell'accesso alla memoria condivisa (*bank conflicts*).

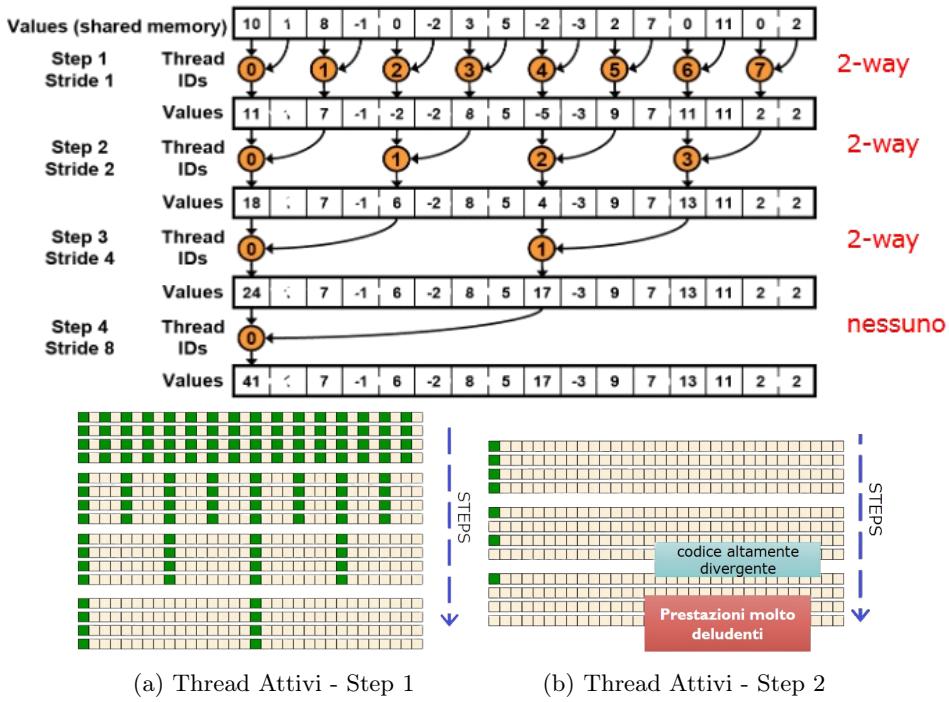


Figura 31: Utilizzando la strategia II di riduzione parallela, ad ogni passo si verificano 2-way bank conflict. Inoltre il codice prodotto risulta altamente divergente.

Una gestione ottimale dei thread prevede che vengano attivati sempre **thread consecutivi**. In questo modo: tutti i thread attivi del blocco eseguono le stesse istruzioni, evitando la divergenza; ogni thread accede a una cella distinta della memoria condivisa, evitando conflitti di accesso (*bank conflicts*); la riduzione procede in modo regolare, dimezzando il numero di thread attivi a ogni passo.

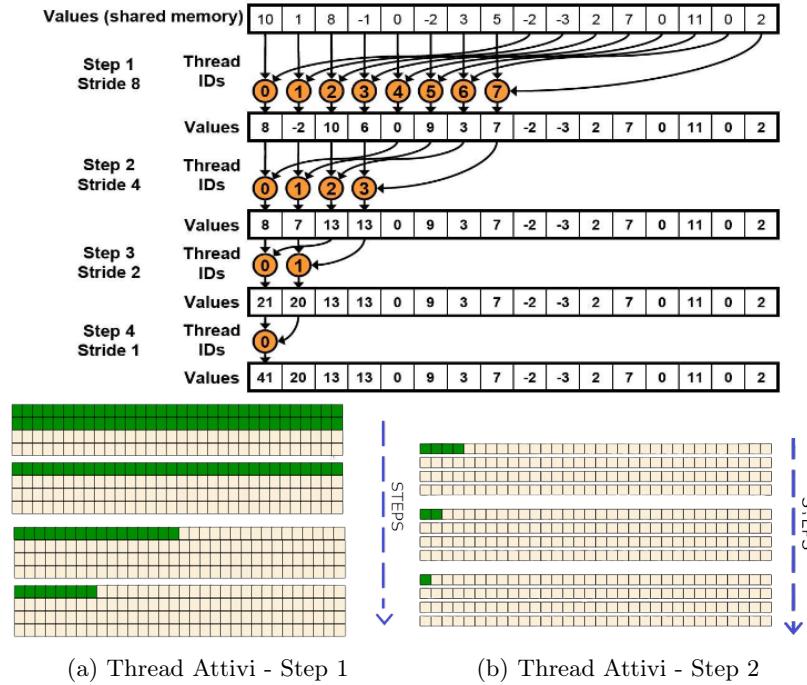


Figura 32: Riduzione parallela ottimizzata: i thread attivi sono consecutivi, il codice non presenta divergenze e non si verificano conflitti di memoria.

## Lab 4.2 Sincronizzazione

La sincronizzazione tra i thread di un blocco avviene attraverso l'istruzione `__syncthreads()`. Questa istruzione garantisce che ogni thread del blocco abbia completato le istruzioni precedenti `__syncthreads()` prima che l'hardware proceda nell'esecuzione.

## Lab 4.3 Allocazione Statica e Dinamica

La memoria condivisa (*shared memory*) è una porzione di memoria veloce, situata all'interno dello Streaming Multiprocessor (SM), accessibile in lettura e scrittura da tutti i thread di uno stesso blocco. CUDA permette due modalità di allocazione della memoria condivisa all'interno di un kernel:

- **allocazione statica:** la dimensione dell'array in `shared memory` è nota e fissata a tempo di compilazione;
- **allocazione dinamica:** la dimensione dell'array è determinata a runtime e specificata al momento del lancio del kernel.

**Allocazione Statica.** Quando la dimensione dell'array è nota a priori, è possibile dichiarare la memoria condivisa direttamente all'interno del kernel:

```
__shared__ int s[30];
```

In questo caso, CUDA riserva esattamente lo spazio indicato per ogni blocco che verrà lanciato. L'allocazione statica è semplice e veloce, ma non flessibile: il numero di elementi dev'essere conosciuto durante la compilazione.

**Allocazione Dinamica.** Quando la dimensione necessaria dipende dai parametri del problema, è preferibile utilizzare la memoria condivisa dinamica. In questo caso la dichiarazione nel kernel è:

```
extern __shared__ int M[];
```

L'effettiva quantità di memoria da riservare viene passata come terzo parametro del lancio del kernel:

```
kernel<<grid, block, sizeInBytes>>(...);
```

La memoria condivisa dinamica offre maggiore flessibilità ed è particolarmente utile in algoritmi come le riduzioni, gli scan, i prodotti scalari ottimizzati e in generale quando è richiesto un buffer di dimensione variabile per blocco.

```
1 __global__ void staticKernel(...) {
2     __shared__ int s[30]; // Allocazione statica
3     ...
4 }
5
6 __global__ void dynamicKernel(...) {
7     extern __shared__ int M[]; // Allocazione dinamica
8     ...
9 }
10
11 int main () {
12     ...
13     sizeM = m * sizeof(int);
14     dynamicKernel<<<grid, block, sizeM>>>(...); // Allocazione dinamica
15     ...
16 }
```

Codice 16: Esempio di allocazione statica e dinamica.

## Lab 5 Programmare con cuBLAS

La programmazione con cuBLAS segue cinque step principali:

1. Creazione delle strutture dati in GPU;
2. Copia dati di input dalla CPU alla GPU con `cublasSetVector()` e `cublasSetMatrix()`;
3. Elaborazione dei dati caricati sul device attraverso le funzioni di cuBLAS. Queste si dividono in BLAS di livello 1, 2, 3, per singola e doppia precisione e per numeri complessi;
4. Aggiornamento dei dati su CPU attraverso le `primitivecublasGetVector()` e `cublasGetMatrix()`;
5. Deallocazione dello spazio in GPU.

### Lab 5.1 Funzioni

- **`cublasStatus_t cublasSetVector(int n, int elemSize, const void *x, int incx, void *y, int incy)`:** copia n elementi da un vettore x nello spazio di memoria dell'host a un vettore y nello spazio di memoria della GPU. La funzione presenta i seguenti parametri:
  - n: il numero degli elementi da copiare dall'array x;
  - elemSize: il numero di byte di ogni elemento dei vettori;
  - x: puntatore all'array allocato sull'host;
  - incx: la spaziatura di memorizzazione tra elementi consecutivi nell'array x;
  - y: puntatore all'array allocato sulla GPU;
  - incy: la spaziatura di memorizzazione tra elementi consecutivi nell'array y.
- **`cublasStatus_t cublasGetVector(int n, int elemSize, const void *x,int incx, void *y, int incy)`:** copia n elementi da un vettore x nello spazio di memoria della GPU a un vettore y nello spazio di memoria dell'host. La funzione presenta i seguenti parametri:
  - n: il numero degli elementi da copiare dall'array x;
  - elemSize: il numero di byte di ogni elemento dei vettori;
  - x: puntatore all'array allocato sulla GPU;
  - incx: la spaziatura di memorizzazione tra elementi consecutivi nell'array x;
  - y: puntatore all'array allocato sull'host;
  - incy: la spaziatura di memorizzazione tra elementi consecutivi nell'array y.

Per ogni programma è necessario avviare le cuBLAS attraverso `cublasCreate(&handle)` prima di utilizzare qualsiasi operazione cuBLAS (in modo da creare un handle specifico della libreria per la gestione delle informazioni e relativo contesto in cui essa opera), il contesto creato deve essere poi passato a tutte le successive chiamate di funzione di libreria e dovrebbe essere infine distrutto con `cublasDestroy(handle)`.

## Lab 5.2 Gestione degli Errori

CuBLAS inoltre è dotato di un sistema per recuperare e comprendere gli errori che avvengono in GPU durante l'esecuzione delle operazioni. Ogni funzione cuBLAS, infatti restituisce un oggetto status di tipo `cublasStatus_t` contenente le informazioni sui possibili errori. Di conseguenza è buona norma controllare lo stato dell'operazione prima di andare avanti:

```
1 if (cudaStat!=CUBLAS_STATUS_SUCCESS) {
2     printf ("CUBLAS error/n");
3     return EXIT_FAILURE;
4 }
```

Valore	Significato
CUBLAS_STATUS_SUCCESS	L'operazione è stata completata con successo.
CUBLAS_STATUS_NOT_INITIALIZED	La libreria cuBLAS non è stata inizializzata. Ciò è solitamente causato dall'assenza di una precedente chiamata <code>cublasCreate()</code> , da un errore dell'API Runtime CUDA richiamato dalla routine cuBLAS o da un errore nella configurazione dell'hardware.
CUBLAS_STATUS_ALLOC_FAILED	L'allocazione delle risorse è fallita nella libreria cuBLAS, tipicamente a causa di un errore in <code>cudaMalloc()</code> .
CUBLAS_STATUS_INVALID_VALUE	Alla funzione è stato passato un parametro non valido o non supportato (ad esempio una dimensione negativa).
CUBLAS_STATUS_ARCH_MISMATCH	La funzione richiede una caratteristica non supportata dall'architettura della GPU; spesso dovuto a capacità di calcolo inferiori a 5.0.
CUBLAS_STATUS_MAPPING_ERROR	Un accesso allo spazio di memoria della GPU non è riuscito.
CUBLAS_STATUS_EXECUTION_FAILED	L'esecuzione del programma GPU è fallita. Spesso è causato da un errore di lancio del kernel sulla GPU.
CUBLAS_STATUS_INTERNAL_ERROR	Un'operazione interna di cuBLAS non è riuscita. È comunemente associato a un errore in <code>cudaMemcpyAsync()</code> .
CUBLAS_STATUS_NOT_SUPPORTED	La funzione richiesta non è supportata.
CUBLAS_STATUS_LICENSE_ERROR	La funzionalità richiesta richiede una licenza; è stato rilevato un errore durante il controllo della licenza corrente.

Tabella 1: Codici di errore di cuBLAS.

## Lab 5.3 Compilazione

Per compilare un codice CUDA-C che usa la libreria cuBLAS, utilizzando il compilatore nvcc basta inserire un collegamento `-lcublas` per la libreria insieme alla compilazione del programma.

```

1 // Compilare
2 nvcc nomefile.cu -lcublas -o nomeprogramma
3 // Eseguire
4 ./nomeprogramma

```

## Lab 5.4 Gestione delle Matrici

La libreria cuBLAS è stata progettata per risultare pienamente compatibile con il linguaggio Fortran, il quale adotta una memorizzazione delle matrici in *column-major order*. Di conseguenza, anche cuBLAS utilizza lo stesso schema di memorizzazione, disponendo gli elementi in memoria per colonne.

In C/C++, invece, la memorizzazione standard degli array bidimensionali avviene in *row-major order*, quindi gli elementi di una stessa riga sono contigui in memoria. Per poter utilizzare correttamente le funzioni di cuBLAS, è quindi necessario rappresentare le matrici in formato column-major, rendendo contigui in memoria gli elementi appartenenti alla stessa colonna. Per facilitare la gestione degli indici, il manuale di cuBLAS propone la seguente macro, utile per accedere all'elemento nella riga *i* e colonna *j* di una matrice memorizzata per colonne<sup>6</sup>:

```

1 #define IDX2C(i,j,ld) (((j)*(ld)) + (i)) // ld = numero di righe

```

Qualora non si voglia rinunciare alla semantica degli array multidimensionali tipica del C, è possibile ricorrere a una strategia alternativa: chiamare le funzioni cuBLAS utilizzando le versioni che operano su matrici trasposte. In questo modo, la matrice può rimanere memorizzata in row-major, mentre la logica delle operazioni viene adattata tramite l'uso dei flag di trasposizione (CUBLAS\_OP\_T, CUBLAS\_OP\_N), lasciando invariata la rappresentazione interna dei dati.

- **cublasStatus\_t cublasSetMatrix(int rows, int cols, int elemSize, const void \*A, int lda, void \*B, int ldb):** copia un riquadro di elementi da una matrice A nello spazio di memoria dell'host a una matrice B nello spazio di memoria della GPU.

La funzione presenta i seguenti parametri:

- rows: numero di righe da copiare dalla matrice A;
- cols: numero di colonne da copiare dalla matrice A;
- elemSize: il numero di byte di ogni elemento della matrice;
- A: puntatore alla matrice allocata sull'host;
- lda: numero di righe della matrice allocata per A anche se ne viene utilizzata solo una sottomatrice;
- B: puntatore alla matrice allocata sulla GPU;
- ldb: numero di righe della matrice allocata per B anche se ne viene utilizzata solo una sottomatrice.

- **cublasStatus\_t cublasGetMatrix(int rows, int cols, int elemSize, const void \*A, int lda, void \*B, int ldb):** copia un riquadro di elementi da una matrice A

---

<sup>6</sup>La macro è utilizzabile solo quando la matrice è già disponibile in formato lineare, dunque non è impiegabile come sostituto diretto della semantica degli array bidimensionali in C/C++.

nello spazio di memoria della GPU a una matrice B nello spazio di memoria dell'host.  
La funzione presenta i seguenti parametri:

- rows: numero di righe da copiare dalla matrice A;
  - cols: numero di colonne da copiare dalla matrice A;
  - elemSize: il numero di byte di ogni elemento della matrice;
  - A: puntatore alla matrice allocata sulla GPU;
  - lda: numero di righe della matrice allocata per A anche se ne viene utilizzata solo una sottomatrice;
  - B: puntatore alla matrice allocata sull'host;
  - ldb: numero di righe della matrice allocata per B anche se ne viene utilizzata solo una sottomatrice.
- **cublasStatus\_t cublasSgemv(cublasHandle\_t handle, cublasOperation\_t trans, int m, int n, const f loat \*alpha, const f loat \*A, int lda, constf loat \*x, int incx, const f loat \*beta, f loat \*y, int incy):** esegue la seguente operazione:

$$y = \alpha \cdot op(A) \cdot x + \beta \cdot y$$

con  $\alpha, \beta$  scalari, A matrice e y vettore. La funzione presenta i seguenti parametri:

- handle: l'handle al contesto della libreria cuBLAS;
- trans: operazione sulla matrice A: con CUBLAS\_OP\_N allora  $op(A) = A$  mentre con CUBLAS\_OP\_T allora  $op(A) = A^T$ ;
- m: numero di righe della matrice A;
- n: numero di colonne della matrice A;
- alpha: scalare utilizzato per la moltiplicazione;
- A: puntatore alla matrice allocata sulla GPU;
- lda: *leading dimension* della matrice A (numero di righe nel caso di memorizzazione per colonne);
- x: puntatore al vettore allocato sulla GPU;
- incx: lo spazio tra due elementi consecutivi nell'array x;
- beta: scalare utilizzato per la moltiplicazione;
- y: puntatore al vettore risultato allocato sulla GPU;
- incy: lo spazio tra due elementi consecutivi nell'array y.