# Prodotto tra due matrici in parallelo

Valerio Brauzi
*Tor Vergata*
Rome, Italy
0333768

***Abstract*—Il presente documento intende descrivere la struttura di un sistema applicativo costruito al fine di calcolare in maniera parallela il prodotto tra due matrici come segue:**

$$C \leftarrow C + AB$$

**dove $A$ è una matrice $m \times k$ e $B$ è una matrice $k \times n$, motivando le scelte architetturali prese nello sviluppo dello stesso.**

## I. Introduzione

Il progetto presentato si focalizza sulla realizzazione di un nucleo di calcolo per il prodotto tra due matrici, con l'obiettivo di calcolare l'espressione $C \leftarrow C + AB$, dove $A$ è una matrice di dimensioni $m \times k$ e $B$ una matrice di dimensioni $k \times n$. Il progetto considera due principali configurazioni per le matrici di input:

1) Matrici quadrate con $m = n = k$.
2) Matrici rettangolari con $m, n \gg k$, dove $k$ assume valori tipici della tassellatura dei dati ($k = 32, 64, 128, 156$).

In entrambe le configurazioni, i valori di $m$ e $n$ saranno incrementati per ottenere una curva delle prestazioni,in termini di FLOPS misurate utilizzando la formula:

$$\text{FLOPS} = \frac{2 \cdot mnk}{T}$$

dove $T$ rappresenta il tempo di esecuzione.

Il codice è sviluppato in due versioni: una con parallelizzazione **MPI** e l'altra con parallelizzazione **CUDA**. Si noti come la versione MPI sarà progettata per avere un'interfaccia e una distribuzione dei dati simili a quelle utilizzate in ScaLAPACK.

Le prestazioni del codice sono state valutate sul server didattico del dipartimento. Per il collaudo e le misure di performance, è stata predisposta la generazione di matrici di prova, utilizzando generatori di numeri pseudocasuali. Ciascun prodotto viene svolto in parallelo e, ove la dimensione del problema lo permette, verificate con un classico prodotto lineare.

### A. Legenda

Di seguito si descrivono i simboli utilizzati nella successiva documentazione, al fine di non appesantire la trattazione.

| Simbolo | Descrizione |
|---|---|
| $m$ | Numero di righe della matrice $A$ e della matrice $C$ |
| $n$ | Numero di colonne della matrice $A$ e numero di righe della matrice $B$ |
| $k$ | Numero di colonne della matrice $B$ e della matrice $C$ |
| $B_r$ | Numero di righe del blocco ScaLAPACK |
| $B_c$ | Numero di colonne del blocco ScaLAPACK |
| $p$ | Numero di processi utilizzati |

TABLE I
DESCRIZIONE DEI SIMBOLI UTILIZZATI NEL PROGETTO

## II. MPI

### A. Overview

Il primo framework utilizzato per parallelizzare l'operazione descritta poco sopra è **MPI**, allo scopo di permettere l'esecuzione del codice su $p$ processi differenti. Per ottimizzare le prestazioni del programma, è essenziale che i $p$ processi suddividano il lavoro in modo equo, lavorando su porzioni delle matrici con dimensioni il più possibile simili. In

particolare, ai fini del progetto, è richiesto che la distribuzione dei dati e l'interfaccia utilizzata siano analoghe a quelle impiegate in **ScaLAPACK**.

### *B. ScaLAPACK*

Relativamente alla suddivisione dei dati, nella versione più generale di ScaLAPACK è previsto che una delle matrici venga suddivisa secondo uno schema **block cyclic**: nello specifico caso, è la matrice C ad essere suddivisa secondo tale tecnica.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 2 | 3 | 2 | 3 | 2 | 3 | 2 | 3 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 2 | 3 | 2 | 3 | 2 | 3 | 2 | 3 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 2 | 3 | 2 | 3 | 2 | 3 | 2 | 3 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 2 | 3 | 2 | 3 | 2 | 3 | 2 | 3 |

Fig. 1. Matrice 8x8 con blocchi 2x2

Nella figuta 1 è rappresentato un esempio di matrice C di dimensioni $8\times8$ suddivisa in blocchi $2\times2$, la cui topologia è evidenziata in giallo. In una configurazione con $p$ = 4, ciascun processo (identificati da 0 a 3) è responsabile di una sola cella per blocco: in questo modo si perviene ad una suddivisione del carico di lavoro quanto più equa possibile.
Si noti che per come è stata strutturata l'applicazione la matrice può essere anche rettangolare, così come il blocco: ogni caso "particolare" è gestito con una logica analoga a quella descritta.

### *C. Generazione Matrici*

La generazione delle matrici di input è stata implementata in un programma ausiliario (matrix_generator.ccp) e avviene pseudo-randomicamente mediante la funzione **udist(rng)**, che simula l'estrazione di un valore posta una distribuzione normale standard (si noti come rng sfrutti l'algoritmo del Mersenne Twister e venga inizializzato per gli esperimenti con un seed = 42, aumentato di uno per la generazione della matrice B). Le matrici così generate avranno tutti i valori compresi tra 0.0 e 100.0.

### *D. Trasporre la matrice B*

In entrambe le versioni del software (Cuda e MPI), è etato deciso di **trasporre la matrice B** per motivi di efficienza. Come è ben noto, il prodotto tra matrici prevede che l'i-esima riga della prima sia moltiplicata all'i-esima colonna della seconda in maniera iterativa: mentre gli elementi presenti su una stessa riga possono essere acceduti in maniera sequenziale (sono difatti memorizzati in locazioni di memoria contigue), gli elementi su una stessa colonna non godono di questa garanzia, dunque l'accesso risulta ben più lento e per moltiplicazioni tra matrici di grandi dimensioni si sperimenta un delay significativo.
Se invece, prima di eseguire la moltiplicazione, si traspone la matrice B, si può eseguire un prodotto "riga per riga", che ci assicura solo accessi a celle di memoria contigue.

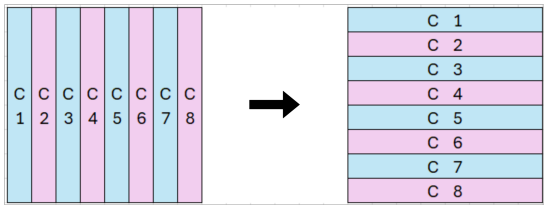


Fig. 2. Trasposizione Matrice B; Cx = colonna x

### *E. Descrizione dell'applicativo MPI*

Di seguito si descrive il flusso di lavoro per svolgere il prodotto tra matrici sfruttando il framework MPI.
Dopo aver inizializzato le variabili utili ed aver creato i processi MPI, vengono create le matrici A e B solo nel processo con rank 0, per poi essere diffuse in tutti gli altri processi tramite comunicazione broadcast.

Dopo aver trasposto la matrice B, è necessario che per ogni processo venga calcolato l'**array delle coordinate** degli elementi della matrice C che devono essere dall i-esimo processo. A tale scopo sono state progettate due soluzioni:

1) se ci sono meno processi che elementi nel blocco, è necessario che almeno un processo gestisca almeno due elementi all'interno dello stesso blocco: posta questa premessa ogni elemento della matrice C viene mappato con l'id del processo che dovrà gestirlo, ottenendo una matrice come quella mostrata di seguito. Con questa matrice "mappata" è possibile, per ogni processo, calcolare la dimensione dell'array di coordinate, nonchè l'array stesso.
   Si noti che senza l'operazione di mapping, in questo caso non sarebbe possibile calcolare direttamente la dimensione dell'array.
2) se invece il numero dei processi è uguale alla dimensione del blocco, è possibile calcolare direttamente la dimensione dell'array di coordinate, e dunque lo stesso.
3) se ci sono più processi che elementi nel blocco, il programma non può essere eseguito.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 2 | 0 | 2 | 0 | 2 | 0 | 2 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 2 | 0 | 2 | 0 | 2 | 0 | 2 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 2 | 0 | 2 | 0 | 2 | 0 | 2 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 2 | 0 | 2 | 0 | 2 | 0 | 2 | 0 |

Fig. 3. Esempio con 3 processi e blocco 2x2

Dopo questa fase viene effettuato il prodotto tra matrici: ora che l'i-esimo processo conosce quali elementi della matrice C deve riempire, **il problema viene ridotto a svolgere alcuni prodotti vettoriali** (in caso di una matrice quadrata e di $p = B_r \times B_c$, sarebbero $\frac{m \times k}{p}$ per ciascun processo).
Supponiamo che il processo $i$ debba gestire l'elemento di coordinate (x,y): il processo moltiplicherà il vettore riga x della matrice A e il vettore riga y della matrice B (che si ricorda essere stata precedentemente trasposta). Questa operazione deve essere ripetuta per ogni entry dell'array delle coordinate, e ciascun risultato viene scritto in un nuovo array, che chiameremo **array risultati**: utilizzando ancora l'array delle coordinate e quello appena ricavato ciascun processo costruisce una **matrice temporanea** tale per cui gli elementi gestiti dal processo $i$ vengono portati al valore corrispondente, mentre tutti gli altri elementi sono inizializzati tutti a 0. Si perviene dunque ad una matrice di questo tipo:

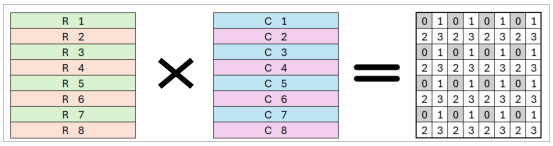


Fig. 4. Esempio di prodotto

In questo esempio, il processo $p = 0$ sarà responsabile di calcolare il prodotto tra i vettori $R1 \times C1$ e salvarne il risultato nell'elemento di $C$ che ha coordinate (0,0).
In ultima istanza, tutti i processi inviano al processo con rank = 0 la propria matrice temporanea, il quale procede a sommarle man mano che vengono ricevute, fino a pervenire alla **matrice C finale**. Al termine di queste operazioni lo spazio allocato viene liberato ed i processi vengono terminati.

### *F. Risultati*

I risultati sono disponibili in formato csv al link riportato. Si noti come l'andamento dei FLOPS,'se fissato il numero di processi è costante, mentre aumenta All'aumentare del numero di processi utilizzati, il tempo di esecuzione per le moltiplicazioni di matrici quadrate $8192 \times 8192$ diminuisce sensibilmente solo quando i processi riempiono pienamente

il blocco ScaLAPACK (come chiaramente c'era da aspettarsi). Di seguito si riportano due grafici, rispettivamente relativi all'andamento del tempo di esecuzione (espresso in secondi) e ai FLOPS per il prodotto di matrici $8192 \times 8192$ al variare del numero di processi MPI coinvolti nelle operazioni.

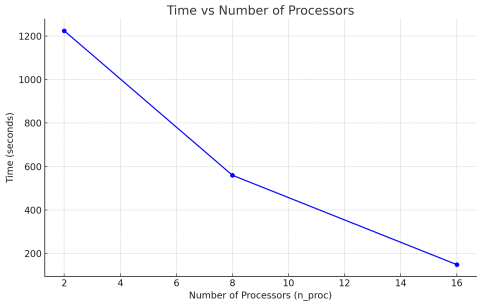


Fig. 5. time vs n-proc

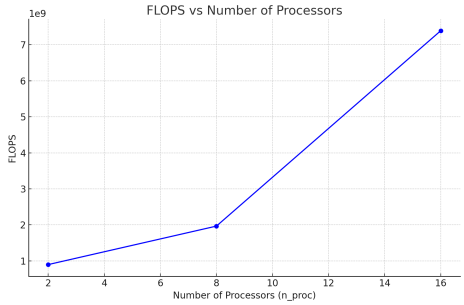


Fig. 6. FLOPS vs n-proc

Si noti come il risultato del prodotto tra matrici, per matrici di dimensioni minori di 2048 elementi, viene sempre controllato tramite un calcolo a singolo processo della stessa operazione: in caso le due matrici C non combaciassero, le configurazioni di A e B sarebbero riportate in un file denominato **errors.csv**, che chiaramente è vuoto.

# III. CUDA

## A. Overview

Il secondo framework scelto per parallelizzare il prodotto tra due matrici

$$C \leftarrow C + AB$$

è **CUDA**, sviluppato da NVIDIA, che consente agli sviluppatori di utilizzare le GPU per eseguire calcoli generali tramite funzioni chiamate **kernel**. La parallelizzazione dell'operazione avviene sfruttando i thread che vengono attivati in GPU: tali thread sono organizzati in **blocchi**, tali per cui ciascun blocco esegue su una stessa unità di processamento. In ultimo, tutti i blocchi sono organizzati in **griglie**, tali per cui ciascun kernel è eseguito come una griglia di blocchi di threads. Nell'applicazione sviluppata, i blocchi in cui viene suddivisa la matrice per la parallelizzazione corrispondono ai blocchi di thread CUDA: ciascun thread del blocco CUDA opererà solo su alcuni valori del blocco della matrice.
La dimesione del blocco è stata scelta secondo un ragionamento basato sul funzionamento harware indotto da CUDA: la macchina eseguente organizza i thread in **warp**, gruppi di 32 thread indicizzati in grado di eseguire la stessa sequenza di istruzioni contemporaneamente. Dato che i thread utilizzano le stesse risorse hardware anche se in un warp ce ne sono meno di 32 (di fatto non scalano), **la scelta più corretta è quella di utilizzare anche a livello software blocchi di dimensione multipla di 32**: nella fattispecie, l'applicativo utilizza blocchi 32x32. Si noti come la generazione delle matrici segua lo stesso procedimento descritto nella sottosezione C della sezione MPI. Inoltre si noti come, in questo caso, non è stato ritenuto necessario trasporre la matrice B.

## B. Descrizione dell'applicativo CUDA

Il software è stato sviluppato seguendo il paradigma di programmazione descritto nella documentazione di cuda, disponibile a questo link e qui riportato:

- Copiare i dati di input dalla memoria host alla memoria del dispositivo, noto anche come trasferimento host-to-device.

- Caricare il programma GPU ed eseguirlo, memorizzando nella cache i dati on-chip per migliorare le prestazioni.
- Copiare i risultati dalla memoria del dispositivo alla memoria host, noto anche come trasferimento device-to-host.

Per prima cosa viene allocato lo spazio per ospitare le matrici A,B e C sull'host tramite la funzione *cudaMallocHost*, per poi generare con il generatore pseudo-random le matrici A e B. Successivamente, tramite la funzione *cudaMalloc*, viene allocato lo stesso spazio sul device che eseguirà le operazioni e tramite una *cudaMemcpy* ci vengono copiate le matrici A e B.
Ora vengono impostate le dimensioni della griglia e dei blocchi tramite le funzioni *dimGrid* e *dimBlock*: occorre notare come le dimensioni della griglia (bidimensionale) siano calcolate secondo le formule

$$\texttt{grid\_rows} = \left\lceil \frac{m + \texttt{BLOCK\_SIZE} - 1}{\texttt{BLOCK\_SIZE}} \right\rceil \quad (1)$$

$$\texttt{grid\_cols} = \left\lceil \frac{k + \texttt{BLOCK\_SIZE} - 1}{\texttt{BLOCK\_SIZE}} \right\rceil \quad (2)$$

Questa in letteratura è descritta come una best practise per impostare le suddette dimensioni, **garantendo che tutti gli elementi della matrice siano coperti dai blocchi di thread**, anche quando la dimensione della matrice non è un multiplo esatto di BLOCK_SIZE.
Con queste premesse è possibile lanciare i kernel per il calcolo del prodotto matriciale: nell'applicazione vengono distinti i casi matrice quadrata e matrice rettangolare:

1) Per le matrici quadrate viene lanciato il kernel apposito: vengono dichiarate due matrici (della dimensione di un blocco) *tile_a* e *tile_b* nella memoria condivisa. Ogni blocco di thread condivide questi dati, riducendo gli accessi alla memoria globale e migliorando quindi le prestazioni. **Ogni thread** carica un singolo elemento di a in tile_a e un singolo elemento di b in tile_b, quindi **collettivamente**, tutti i thread in un blocco caricano un intero blocco di elementi da a e b, rispettivamente. Ogni thread esegue un prodotto scalare tra la riga di tile_a e la colonna di tile_b; il risultato parziale viene accumulato e quando tutti i thread hanno terminato si perviene al risultato desiderato. Questo ragionamento viene ripetuto per ogni blocco della matrice.
2) Per le matrici rettangolari si è preferito non utilizzare la memoria condivisa, in quanto le dimensioni possibilmente fortemente variabili rendono difficile il calcolo degli indici. Dopo aver calcolato le coordinate iniziali del thread esso esegue un ciclo su tutti gli elementi della riga di A e della colonna di B a lui destinate, calcolando il prodotto scalare in maniera iterativa e scrivendolo, alla fine, nell'elemento corrispettivo della matrice C.

Alla luce di quanto riportato, è lecito aspetarsi che i prodotti di matrici quadrate siano più efficienti rispetto a quelli tra matrici rettangolari.
Ove possibile in termini di tempo, il risultato dei calcoli della GPU viene confrontato con quelli eseguiti dulla CPU dell'host in maniera sequenziale, per verificarne la coerenza.
In ultimo ne viene calcolato il tempo di esecuzione e i FLOPS come specificato nella sezione riguardante MPI.

### C. Risultati

I risulati sono disponibili a questo link. Si noti come il tempo è espresso in millisecondi. Si può apprezzare come nella moltiplicazione di matrici quadrate di dimensione $8192 \times 8192$ i FLOPS arrivino all'ordine di 1.13924920177 GFLOPS e il tempo di esecuzione sia di 965.119507 ms, rispetto ai 148.820297 secondi della versione MPI con 16 processi. Provando ad eseguire il programma per effettuare la moltiplicaizione tra matrici $8192 \times 81923$ e $8193 \times 8192$ (utilizzando quindi la funzione di moltiplicazione per matrici rettangolari, che dunque non utilizza shared memory) possiamo notare come sia sensibilmente più lenta: il tempo di esecuzione arriva a 1310.077515 ms mentre i FLOPS a 839477105.22: questo dipende sicuramente da una più difficile suddivisione della matrice tra i vari thread, ma anche dal fatto che l'accesso a memoria condivisa utilizzato nella moltiplicazione

di matrici quadrate è più efficiente di quello a memoria globale.