

# Parallel Matrix Multiplication

Matteo Conti, Luca Falasca

Universita' degli Studi di Roma Tor Vergata

# Roadmap

## 1 Introduzione

- Descrizione del problema
- Obiettivi
- Metriche di valutazione
- Raccolta dei dati

## 2 MPI

- Distribuzione del carico
- Riduzione del risultato
- Implementazione del prodotto

## 3 CUDA

- 1 versione
- 2 versione
- 3 versione
- Configurazione dei parametri

## 4 MPI+CUDA

## 5 Analisi delle prestazioni

- MPI
- CUDA
- MPI+CUDA

# Introduzione - Descrizione del problema

Il progetto verte sull'implementazione di un nucleo di calcolo per effettuare il prodotto tra due matrici dense, definito come:

## Definition

$$C = C + A \cdot B \quad (1)$$

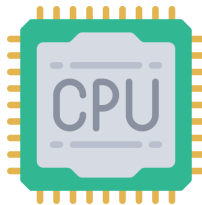
dove  $A$ ,  $B$  e  $C$  sono matrici di dimensioni  $M \times K$ ,  $K \times N$  ed  $M \times N$  rispettivamente, in particolare verranno considerate:

- Matrici quadrate
- Matrici rettangolari con  $M, N \gg K$  con  $K = \{32, 64, 128, 156\}$

# Introduzione - Obiettivi

Verranno analizzate le prestazioni di tre differenti implementazioni del prodotto, in particolare:

- MPI, utilizzando il paradigma SIMD per la parallelizzazione su CPU
- CUDA, sfruttando le potenzialità delle GPU per l'accelerazione computazionale
- MPI+CUDA, cercando di combinare i vantaggi delle due precedenti versioni



# Introduzione - Metriche di valutazione

Per valutare le prestazioni delle soluzioni sviluppate la metrica utilizzata sono i FLOPS, i quali sono definiti come:

## Definition

$$FLOPS = \frac{2MNK}{exec\_time} \quad (2)$$



# Introduzione - Raccolta dei dati

I dati raccolti sono stati ottenuti eseguendo i vari nuclei di calcolo sul server di dipartimento il quale presenta le seguenti specifiche:

- CPU: 2 x Intel Xeon Silver 4210
- Memory: 64 GiB of RAM
- GPU: Nvidia Quadro RTX 5000
- CUDA version: 12.3
- MPI version: 4.1



# Roadmap

## 1 Introduzione

- Descrizione del problema
- Obiettivi
- Metriche di valutazione
- Raccolta dei dati

## 2 MPI

- Distribuzione del carico
- Riduzione del risultato
- Implementazione del prodotto

## 3 CUDA

- 1 versione
- 2 versione
- 3 versione
- Configurazione dei parametri

## 4 MPI+CUDA

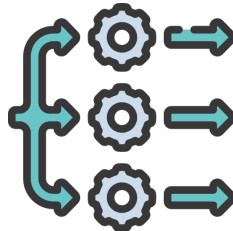
## 5 Analisi delle prestazioni

- MPI
- CUDA
- MPI+CUDA

# MPI

In questa sezione verranno presentate la versione dell'implementazione del prodotto tra matrici utilizzando le funzionalità offerte da MPI per la parallelizzazione su CPU, in particolare verranno affrontati tre aspetti:

- Distribuzione del carico
- Riduzione del risultato
- Implementazione del prodotto





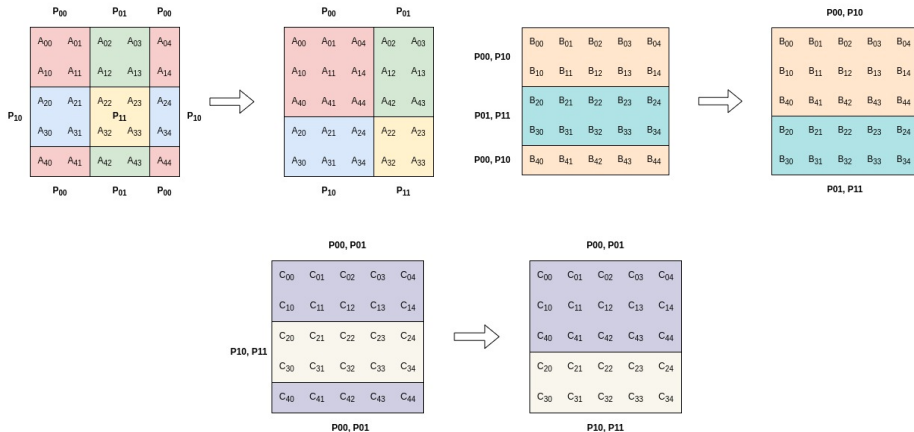
# MPI - Distribuzione del carico (1)

Per la distribuzione del carico è stato utilizzato un approccio analogo a quello utilizzato dalla libreria ScaLAPACK, cioè la block cyclic distribution, questa tecnica permette di distribuire la matrice di partenza in modo da bilanciare il carico tra i processi in modo abbastanza ragionevole. La tecnica si basa su tre idee:

- I processi vengono visti come una griglia  $P_r \times P_c$
- La matrice viene divisa in blocchi di dimensione  $B_r \times B_c$
- La griglia dei processi viene fatta scorrere in modo tumbling sui blocchi assegnando tali blocchi ai processi

# MPI - Distribuzione del carico (2)

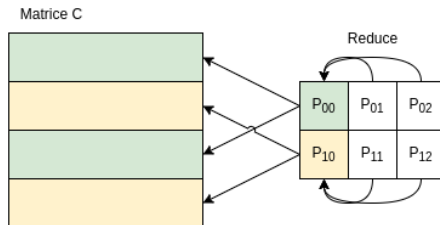
Le matrici A, B e C vengono distribuite come segue:



# MPI - Riduzione del risultato

Dato che ogni riga di processi nella process grid partecipa alle stesse K righe del risultato, si è scelto di non effettuare la riduzione su un solo processo bensì di definire per ogni riga un *row leader*.

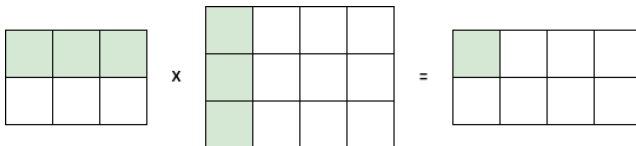
I leader effettuano la riduzione dei risultati parziali di tutti i processi nella loro riga e successivamente scriveranno su file il risultato senza interferire tra loro, in quanto scriveranno in punti diversi.



# MPI - Implementazione del prodotto

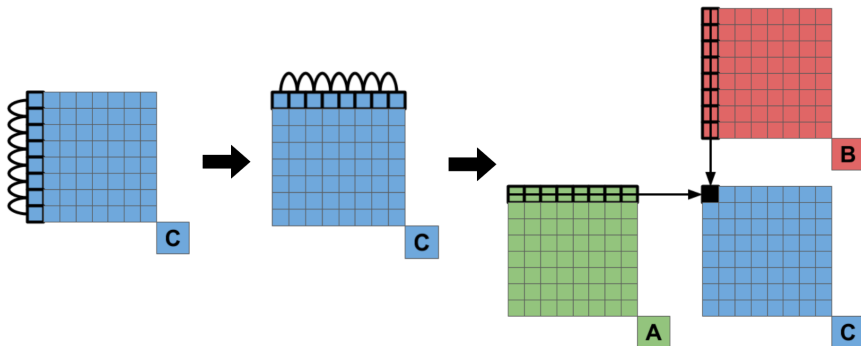
L'effettiva implementazione del prodotto è stata realizzata in due versioni:

- *Naive*, implementazione semplice che non tiene conto di ottimizzazioni
- *Column blocked*, implementazione più complessa che ottimizza l'utilizzo della cache



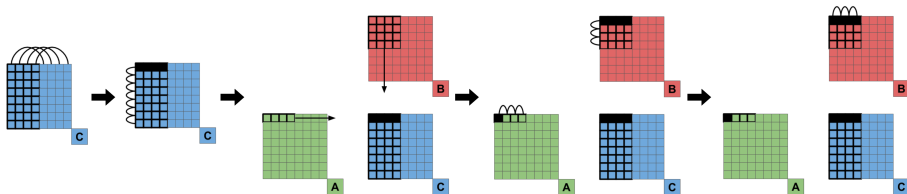
# MPI - Implementazione del prodotto - Naive

Questa implementazione è composta da soli tre cicli che permettono di scorrere le matrici e costruire il risultato.



# MPI - Implementazione del prodotto - Column blocked

Qui viene tenuto conto del fatto che quando si accede un elemento della matrice esso viene caricato in cache insieme ai 15 elementi successivi, è possibile quindi riorganizzare il processamento per sfruttare fenomeno.



# Roadmap

## 1 Introduzione

- Descrizione del problema
- Obiettivi
- Metriche di valutazione
- Raccolta dei dati

## 2 MPI

- Distribuzione del carico
- Riduzione del risultato
- Implementazione del prodotto

## 3 CUDA

- 1 versione
- 2 versione
- 3 versione
- Configurazione dei parametri

## 4 MPI+CUDA

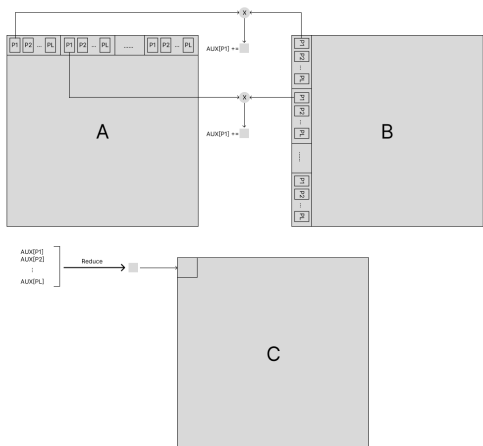
## 5 Analisi delle prestazioni

- MPI
- CUDA
- MPI+CUDA

# CUDA



# CUDA - 1 versione



- 1 Divisione della riga di A tra i thread
- 2 Divisione della colonna di B tra i thread
- 3 Calcolo del prodotto per ogni thread
- 4 Memorizzazione dei risultati parziali in shared memory
- 5 Reduce dei risultati parziali
- 6 Scrittura sulla matrice C
- 7 Ripeti per ogni colonna di B
- 8 Ripeti per ogni riga di A

# CUDA - 1 versione

Nella versione 1 tra il calcolo di una colonna e l'altra, i thread devono sincronizzarsi per poi fare l'operazione di reduce dei risultati parziali.

Questo perchè il vettore in shared memory riesce a contenere solo i risultati della riga di A per 1 colonna di B.

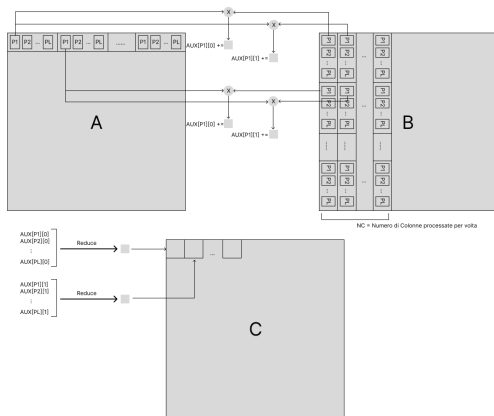
## Soluzione:

Utilizzare una matrice di shared memory che mantiene i risultati parziali di più colonne per volta.

$$aux = \begin{bmatrix} pr_{col_0,t_0} & pr_{col_0,t_2} & \cdots & pr_{col_0,t_{BD}} \\ pr_{col_1,t_0} & pr_{col_1,t_2} & \cdots & pr_{col_1,t_{BD}} \\ \cdots & \cdots & \cdots & \cdots \\ pr_{col_Q,t_0} & pr_{col_Q,t_2} & \cdots & pr_{col_Q,t_{BD}} \end{bmatrix}$$

# CUDA - 2 versione

- 1 Divisione della riga di A tra i thread
- 2 Divisione delle colonne di B tra i thread
- 3 Calcolo del prodotto per ogni thread per tutto il gruppo di colonne
- 4 Memorizzazione dei risultati parziali in shared memory
- 5 Reduce dei risultati parziali
- 6 Scrittura sulla matrice C
- 7 Ripeti per ogni gruppo di colonne di B
- 8 Ripeti per ogni riga di A



# CUDA - 2 versione

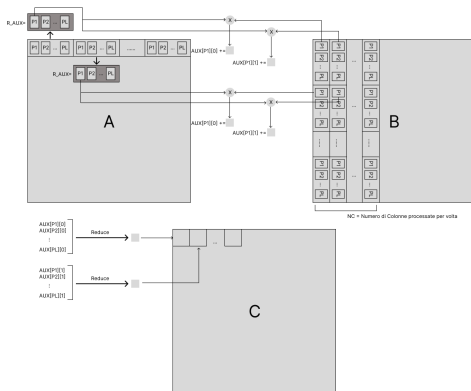
Nella versione 2 quando si processa il gruppo di colonne, poiché si calcola il risultato parziale di una colonna per volta, per ognuna di esse bisogna riaccedere alla riga della matrice  $A$  che è in memoria globale.

## Soluzioni:

- Anziché processare una colonna per volta si processano i primi  $BD$  elementi delle  $Q$  colonne del gruppo corrente.
  - ▶ Questo permette di accedere una sola volta alla riga di  $A$ .
- Mantenere in shared memory parte della colonna  $A$  necessario per il calcolo.
  - ▶ Questo permette di accedere alla memoria globale una sola volta.
  - ▶ Ogni thread accede al proprio elemento della riga di  $A$  necessario per il calcolo dalla memoria shared.

# CUDA - 3 versione

- 1 Divisione della riga di A tra i thread
- 2 Divisione delle colonne di B tra i thread
- 3 Memorizzazione della riga parziale di A in shared memory
- 4 Calcolo del prodotto per ogni thread per il blocco di righe del gruppo di colonne
  - ▶ ripeti per ogni blocco di righe del gruppo di colonne
- 5 Memorizzazione dei risultati parziali in shared memory
- 6 Reduce dei risultati parziali
- 7 Scrittura sulla matrice C



# CUDA - Configurazione dei parametri - Thread

# CUDA - Configurazione dei parametri - Shared memory

# CUDA - Configurazione dei parametri - Bank conflict



# Roadmap

## 1 Introduzione

- Descrizione del problema
- Obiettivi
- Metriche di valutazione
- Raccolta dei dati

## 2 MPI

- Distribuzione del carico
- Riduzione del risultato
- Implementazione del prodotto

## 3 CUDA

- 1 versione
- 2 versione
- 3 versione
- Configurazione dei parametri

## 4 MPI+CUDA

## 5 Analisi delle prestazioni

- MPI
- CUDA
- MPI+CUDA

# MPI+CUDA

# Roadmap

## 1 Introduzione

- Descrizione del problema
- Obiettivi
- Metriche di valutazione
- Raccolta dei dati

## 2 MPI

- Distribuzione del carico
- Riduzione del risultato
- Implementazione del prodotto

## 3 CUDA

- 1 versione
- 2 versione
- 3 versione
- Configurazione dei parametri

## 4 MPI+CUDA

## 5 Analisi delle prestazioni

- MPI
- CUDA
- MPI+CUDA

# Analisi delle prestazioni - MPI

# Analisi delle prestazioni - MPI - Matrici quadrate

# Analisi delle prestazioni - MPI - Matrici rettangolari

# Analisi delle prestazioni - CUDA

# Analisi delle prestazioni - CUDA - Matrici quadrate



# Analisi delle prestazioni - CUDA - Matrici rettangolari

# Analisi delle prestazioni - MPI+CUDA

# Analisi delle prestazioni - MPI+CUDA - Matrici quadrate

# Analisi delle prestazioni - MPI+CUDA - MPI+CUDA

# Grazie per l'attenzione!

- Tutto il codice che implementa il progetto è disponibile al seguente repository:  
<https://github.com/LucaFalasca/ParallelMatrixMultiplication>
- contattaci a:
  - ▶ [matteo.conti@students.uniroma2.eu](mailto:matteo.conti@students.uniroma2.eu)
  - ▶ [luca.falasca@students.uniroma2.eu](mailto:luca.falasca@students.uniroma2.eu)