Parallel Matrix Multiplication

Matteo Conti, Luca Falasca

Universita' degli Studi di Roma Tor Vergata

Roadmap

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

$$C \leftarrow C + A \cdot B \tag{1}$$

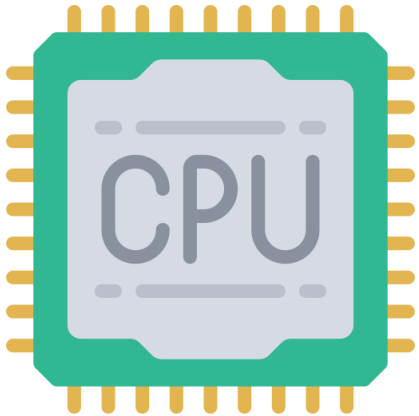
dove A, B e C sono matrici di dimensioni $M \times K$, $K \times N$ ed $M \times N$ rispettivamente, in particolare verranno considerate:

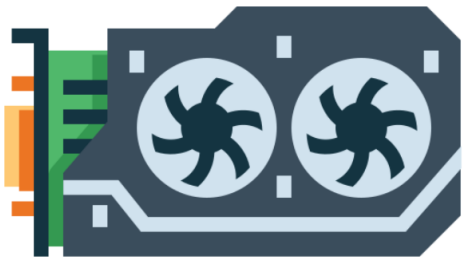
- Matrici quadrate
- Matrici rettangolari con $M, N >> K$ con $K = \{32, 64, 128, 156\}$

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



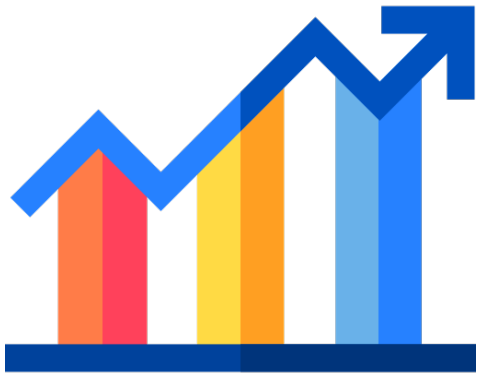


Introduzione - Metriche di valutazione

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

Definition

$$FLOPS = \frac{2MNK}{exec\_time} \tag{2}$$



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:

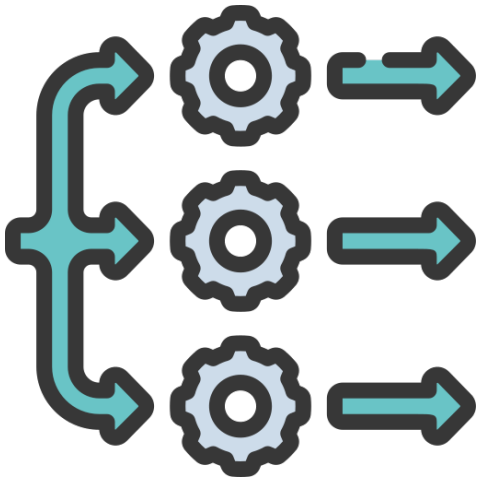
- CPU: 2 x Intel Xeon Silver 4210
- Memory: 64 GiB of RAM
- GPU: Nvidia Quadro RTX 5000
- CUDA version: 12.3
- MPI version: 4.1

Roadmap

MPI

In questa sezione l'implementazione MPI del prodotto tra matrici, in particolare verranno affrontati principalmente tre problemi:

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



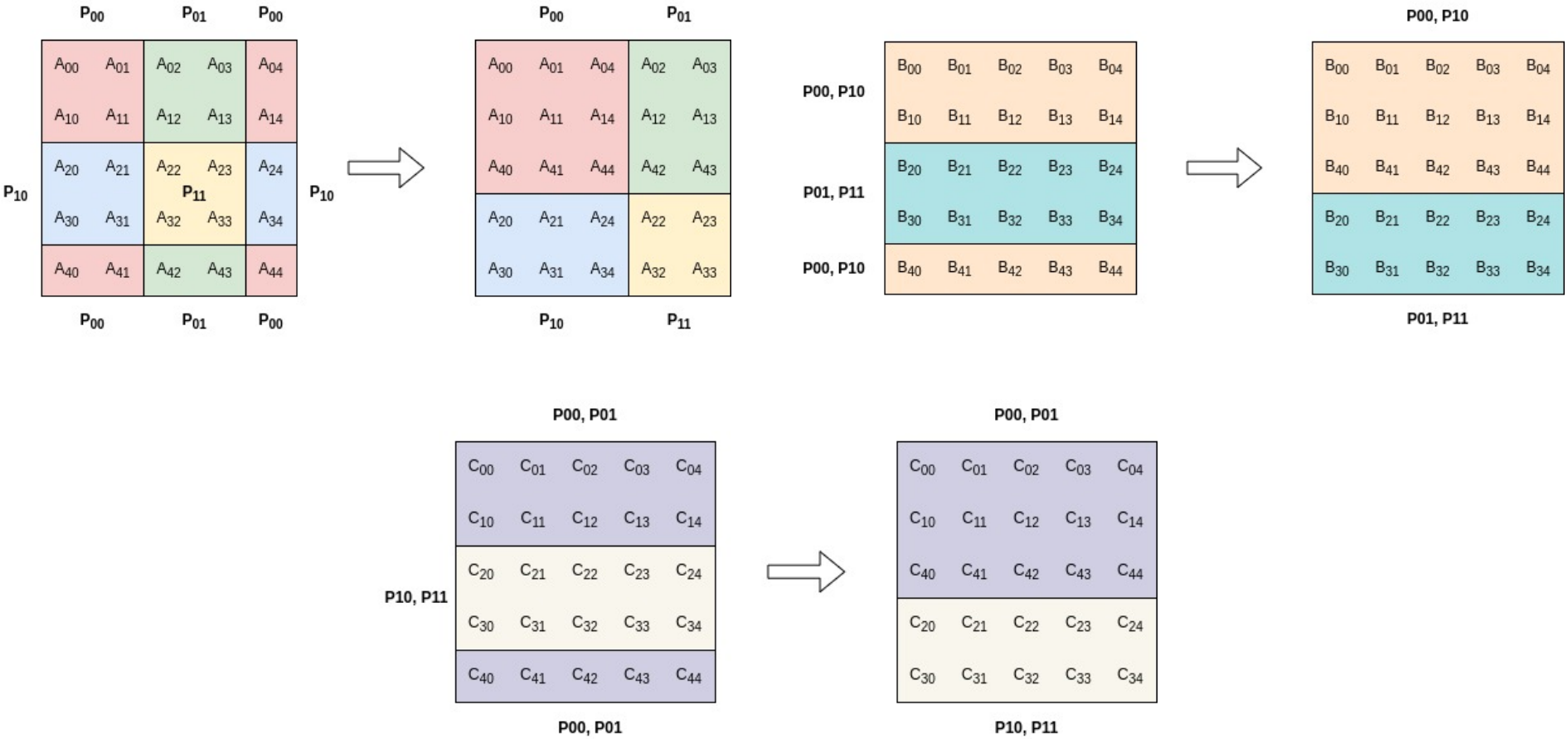
MPI - Distribuzione del carico (1)

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)

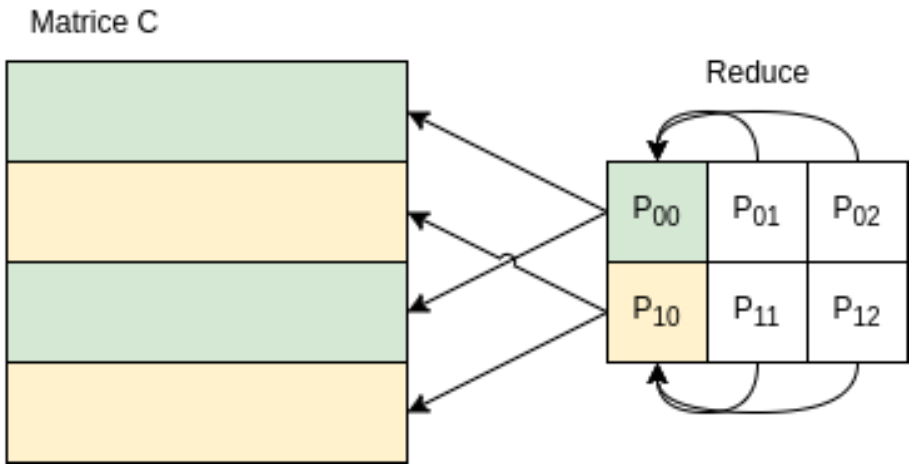
Le matrici A, B e C vengono distribuite come segue:



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*.

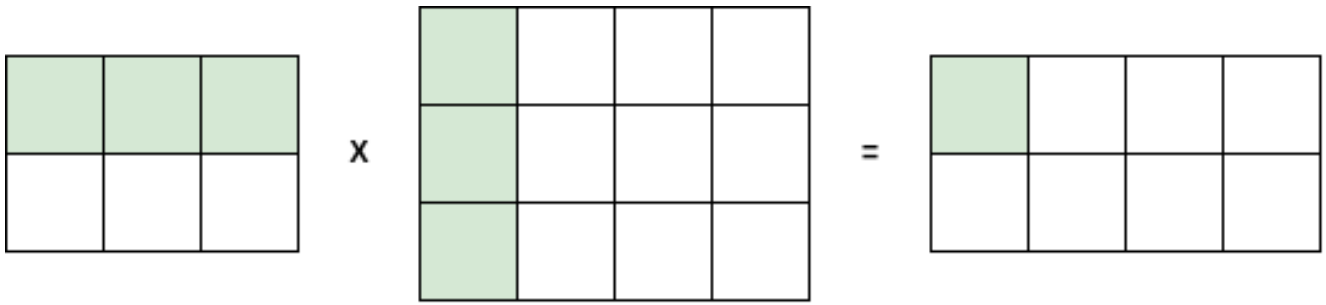
I leader effettuano la riduzione dei risultati parziali di tutti i processi nella loro riga e successivamente scriveranno su file il risultato senza interferire tra loro, in quanto scriveranno in punti diversi.



MPI - Implementazione del prodotto

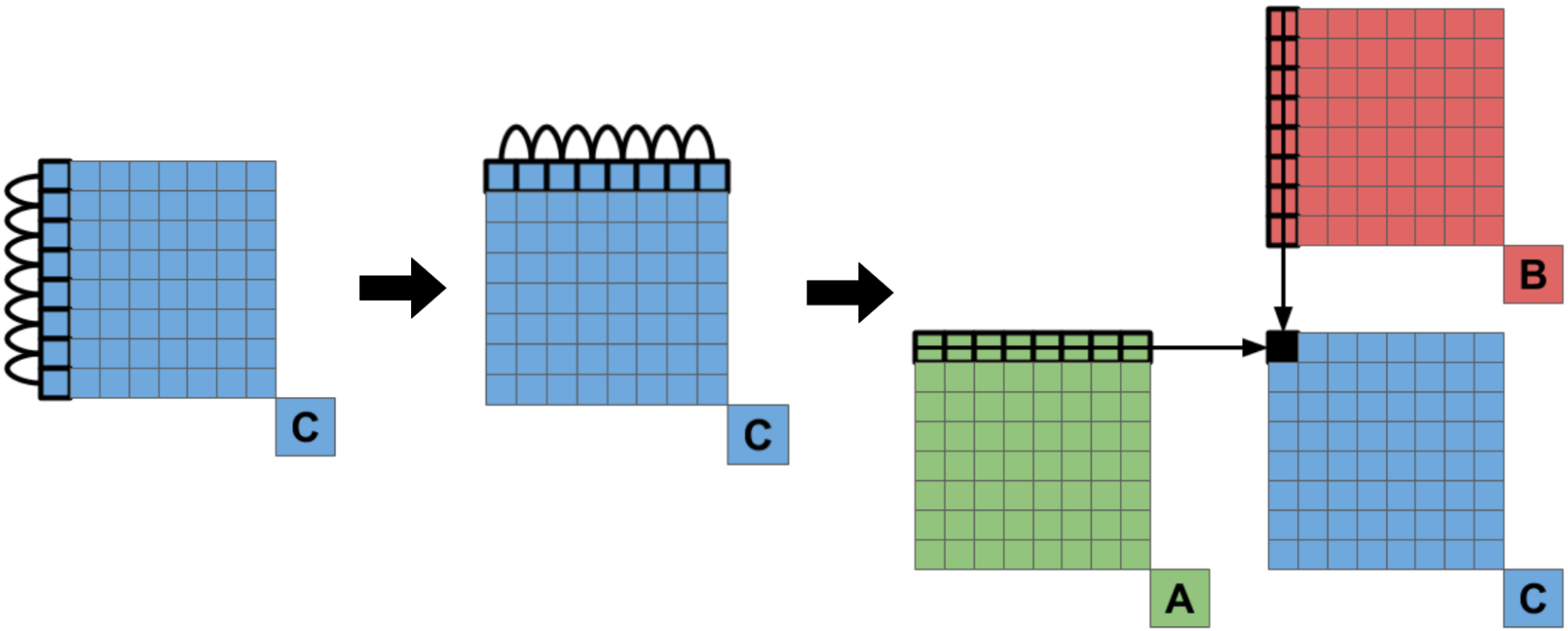
L'effettiva implementazione del prodotto è stata realizzata in due versioni:

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



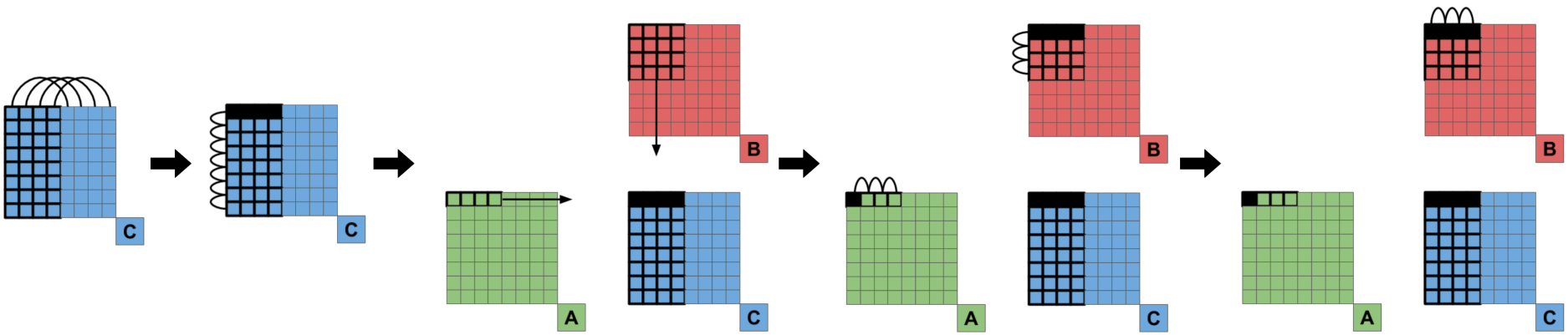
MPI - Implementazione del prodotto - Naive

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

CUDA

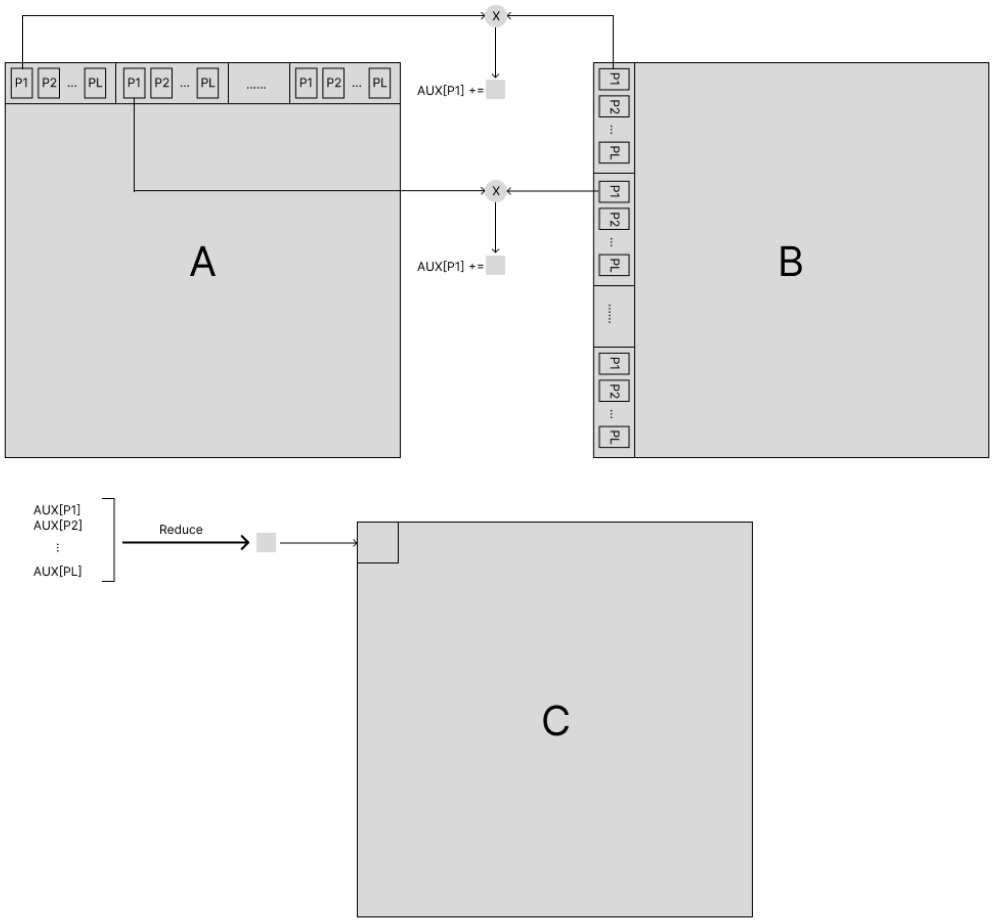
In questa sezione si parlerà dell'implementazione in cuda del prodotto matrice matrice.

Verranno presentate 3 versioni del codice che mostrano degli upgrade basandosi su uno sfruttamento migliore delle risorse a disposizione.

Verranno inoltre affrontati gli aspetti di:

- Gestione del numero di thread
- Gestione della shared memory
- Gestione dei conflitti tra bank e accessi coalizzati

CUDA - Versione 1 (1)



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 - Versione 1 (2)

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.

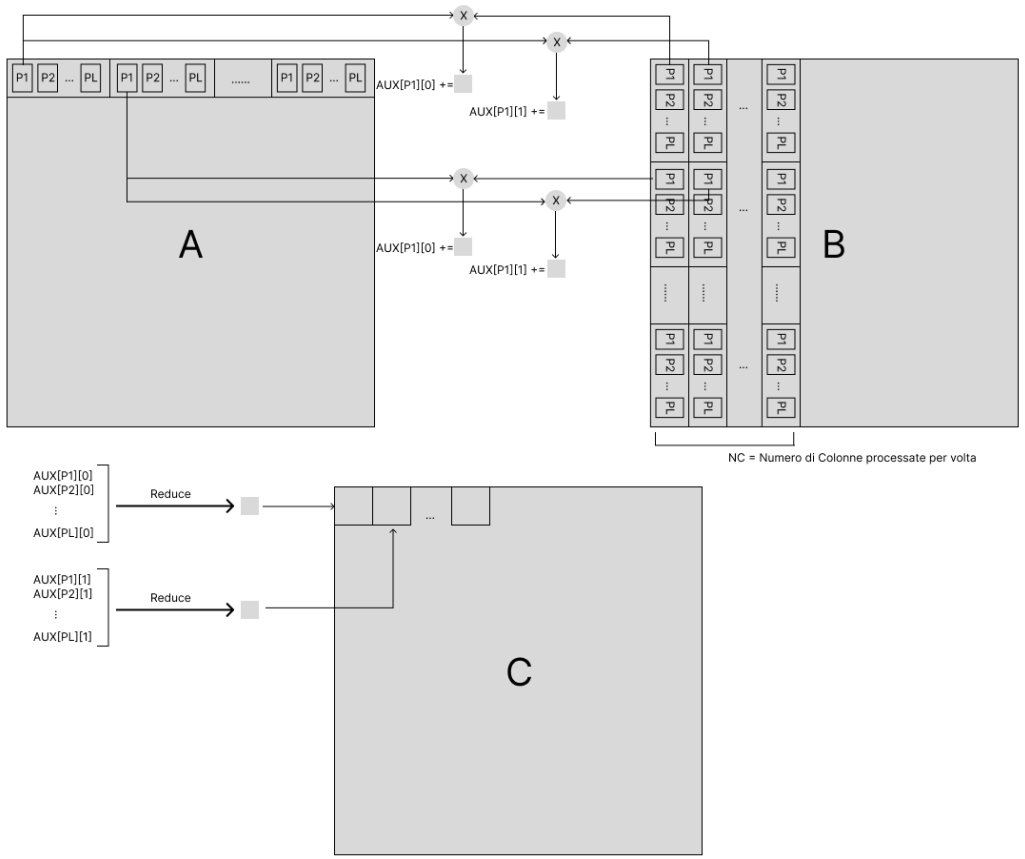
Questo perchè il vettore in shared memory riesce a contenere solo i risultati della riga di A per 1 colonna di B.

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

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

CUDA - Versione 2 (1)

1. Divisione della riga di A tra i thread
2. Divisione delle colonna 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 - Versione 2 (2)

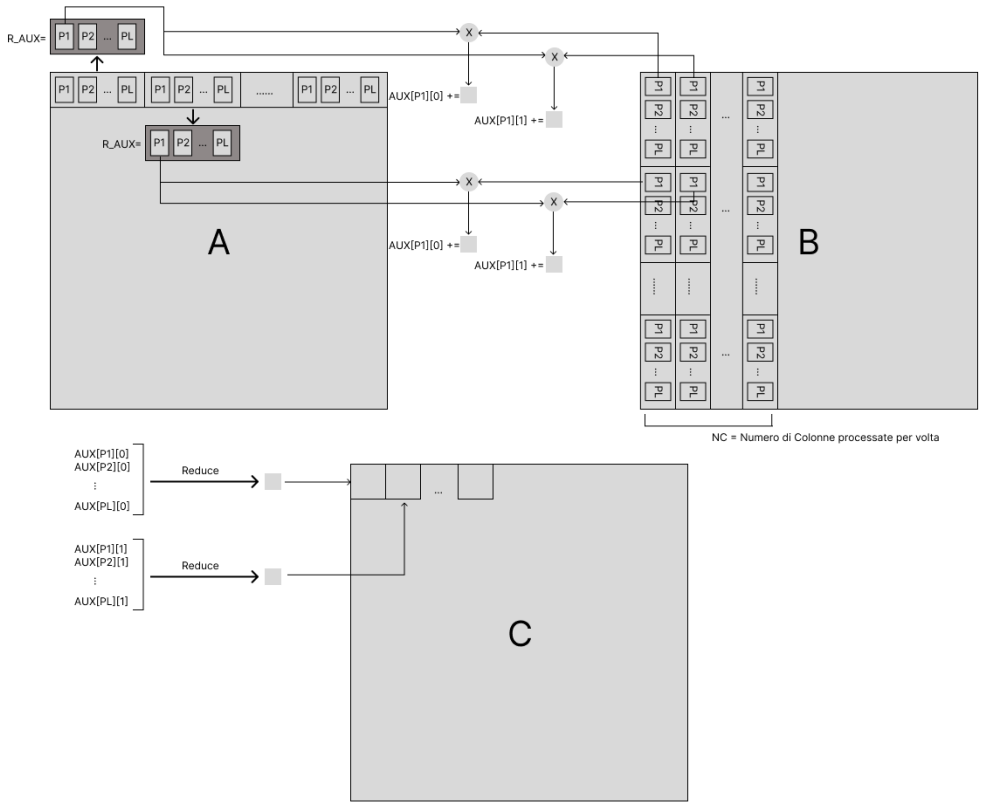
Nella versione 2 quando si processa il gruppo di colonne, poiché si calcola il risultato parziale di una colonna per volta, per ognuna di esse bisogna riaccedere alla riga della matrice A che è in memoria globale.

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.

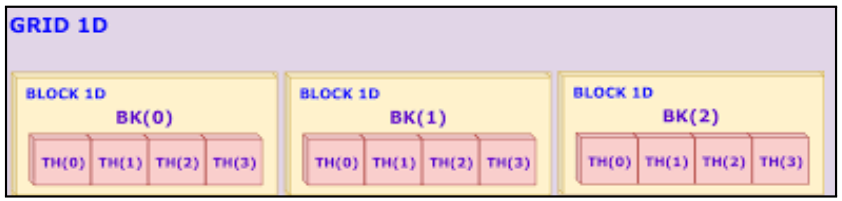
CUDA - Versione 3

1. Divisione della riga di A tra i thread
2. Divisione delle colonna 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

- Griglia unidimensionale di dimensione pari al numero di righe della matrice A
  - Dovuto al fatto che ogni blocco è responsabile di una singola riga della matrice A
- Numero di thread per ogni blocco (block size) pari a 256.
  - Difficile stabilire un valore ottimale
  - Si è rivelato il più efficiente in termini di prestazioni
  - Multiplo di 32, dimensione del warp



CUDA - Configurazione dei parametri - Shared memory

- 1 versione:
  - $size = BD \cdot 4$
- 2 versione:
  - $size = BD \cdot NC \cdot 4$
- 3 versione:
  - $size = (BD \cdot NC + BD) \cdot 4 = 4(BD \cdot (NC + 1))$

Avrebbe senso scegliere un valore di NC tale da massimizzare la shared memory disponibile, tuttavia tale approccio non porta a prestazioni migliori.

Quindi è stato utilizzato un approccio empirico per la scelta di NC, scegliendo il valore con prestazioni maggiori, ovvero $NC = 28$.

CUDA - Configurazione dei parametri - Bank conflit

Verrà analizzata solo la versione 3 perché tutti gli altri casi sono sottoinsiemi di questo.

2 possibili utilizzi della shared memory:

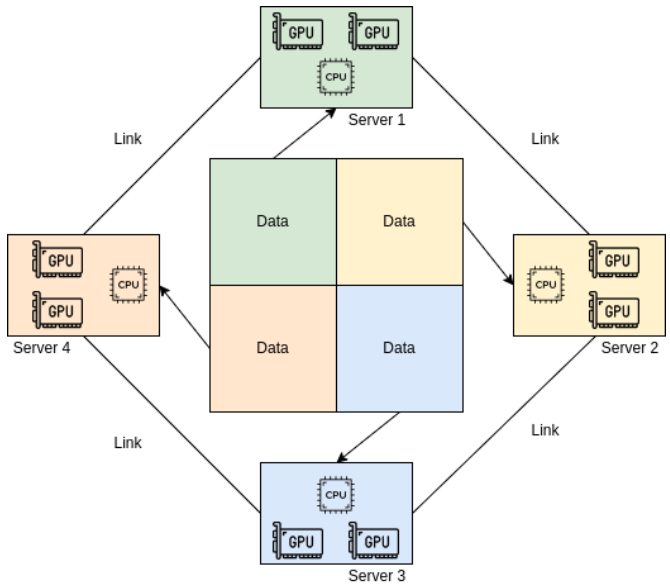
- Memorizzare la riga della matrice A
  - Si memorizza un pezzo per volta di matrice A pari al numero di thread
  - Ogni warp accede a 32 elementi contigui per volta
    - pattern di accesso lineare con stride di una word da 32 bit
- Memorizzare il vettore dei risultati parziali
  - Matrice aux row-order major
  - Ogni warp accede a 32 elementi contigui per volta
    - pattern di accesso lineare con stride di una word da 32 bit

Roadmap

MPI+CUDA

Con questa soluzione si è cercato di sfruttare i vantaggi di MPI e CUDA e mitigarne gli svantaggi, in particolare:

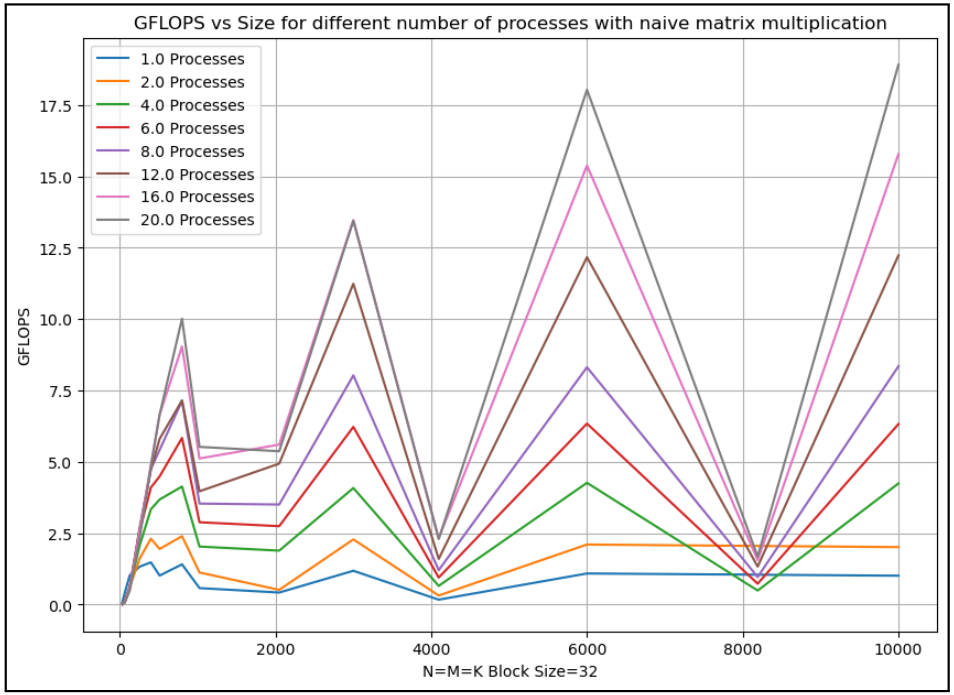
- La taglia del problema non è più limitata dalla singola macchina
- Si sfruttano le GPU dell'intero cluster

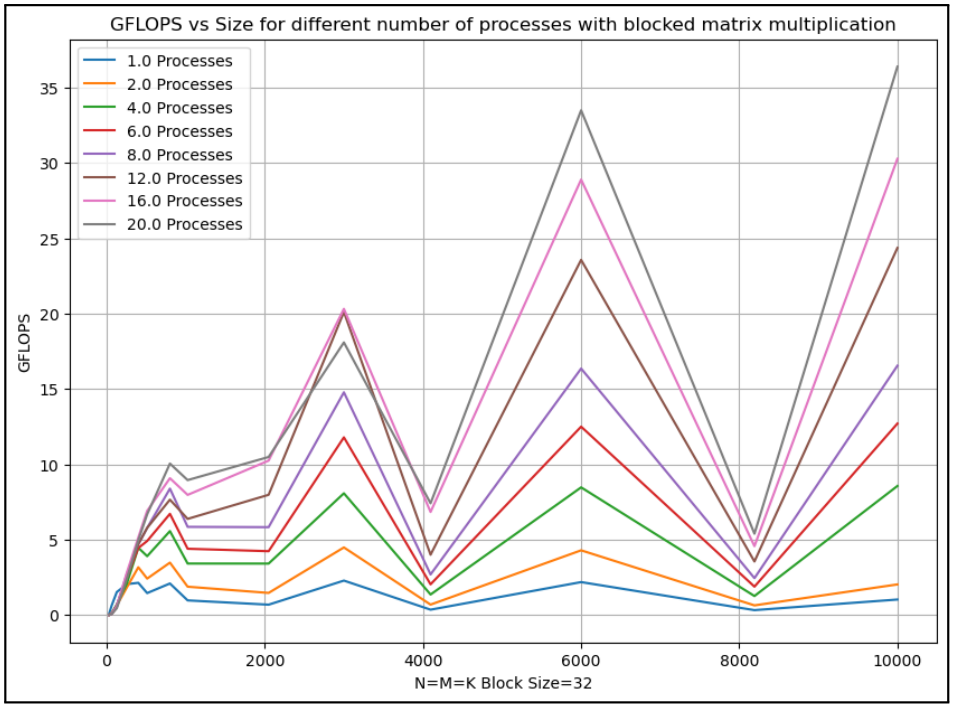


Con questa soluzione si distribuisce (idealmente) il problema all'intero cluster tramite la block cyclic distribution implementata in MPI ed ogni server effettua il prodotto tra le sottmatrici assegnate utilizzando il nucleo di calcolo implementato in CUDA sulle proprie GPU.

Roadmap

Analisi delle prestazioni - MPI - Matrici quadrate (1)

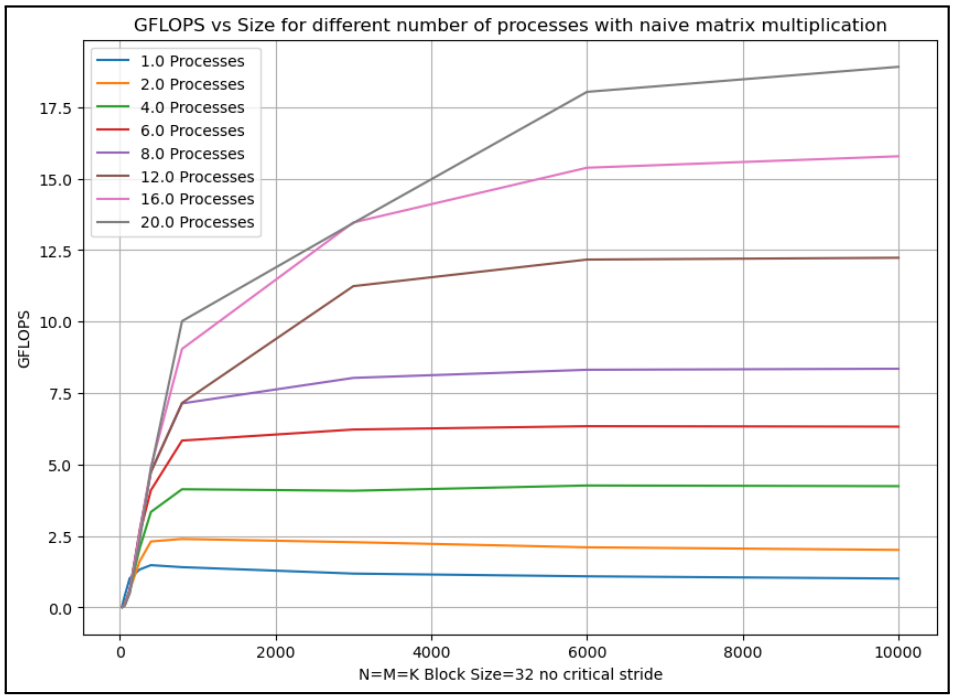


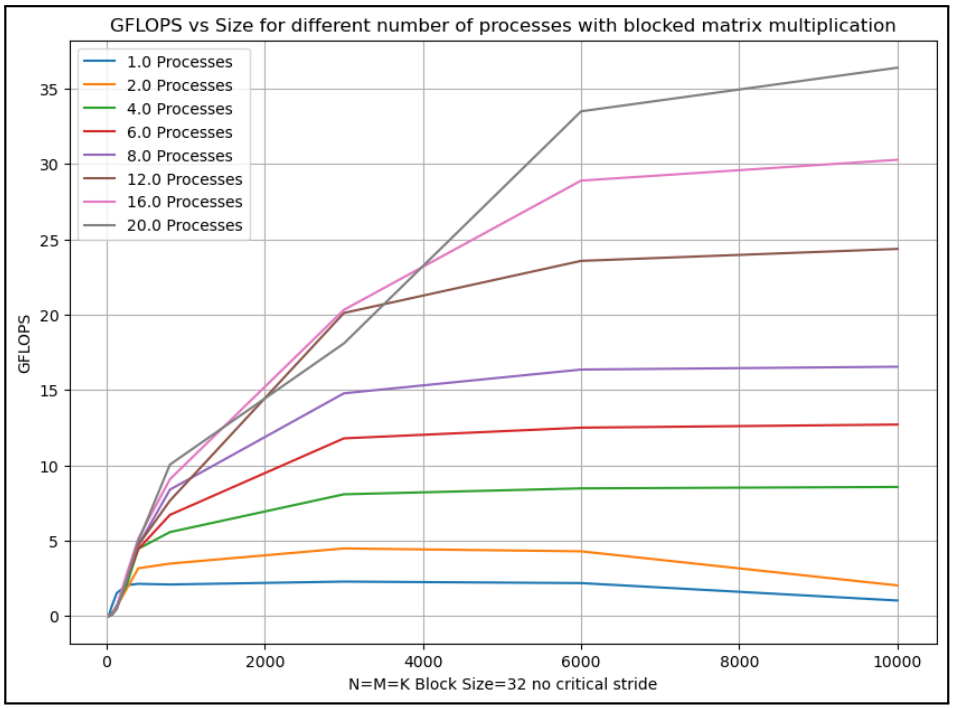


Come è possibile vedere ci sono dei drastici cali di performance, questi avvengono quando le dimensioni delle matrici sono potenze di 2, questo è causato da problemi di cache dovuti al fatto che la taglia delle matrici è multiplo del $critical\_stride = \frac{cache\_size}{associativity-ways}$.

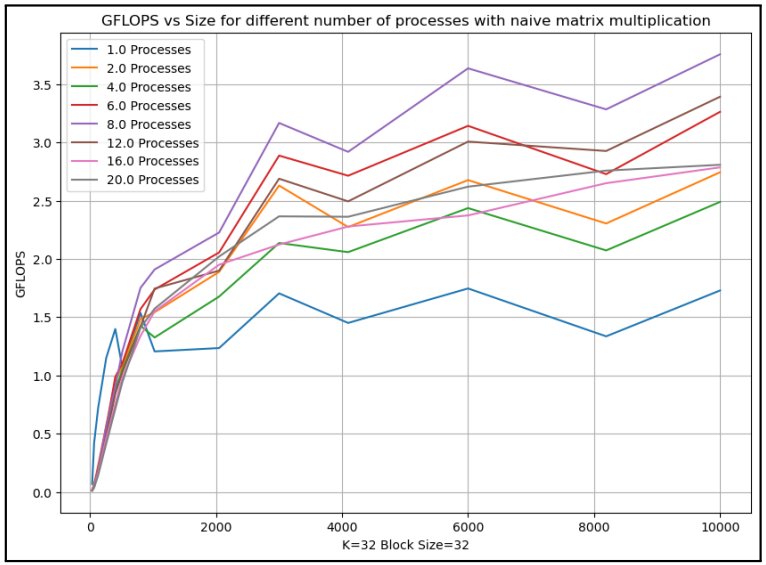
Analisi delle prestazioni - MPI - Matrici quadrate (2)

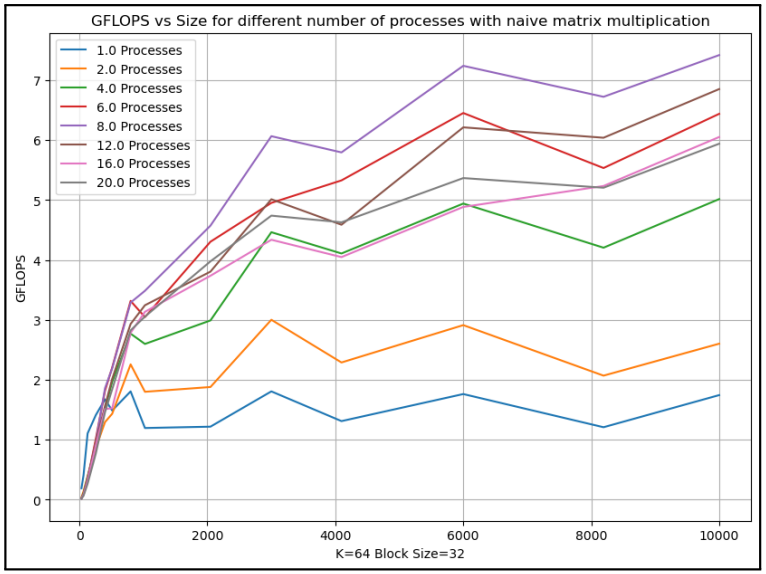
Di seguito per apprezzare meglio l'andamento delle performance vengono riportati grafici in cui non è presente il problema del critical stride.

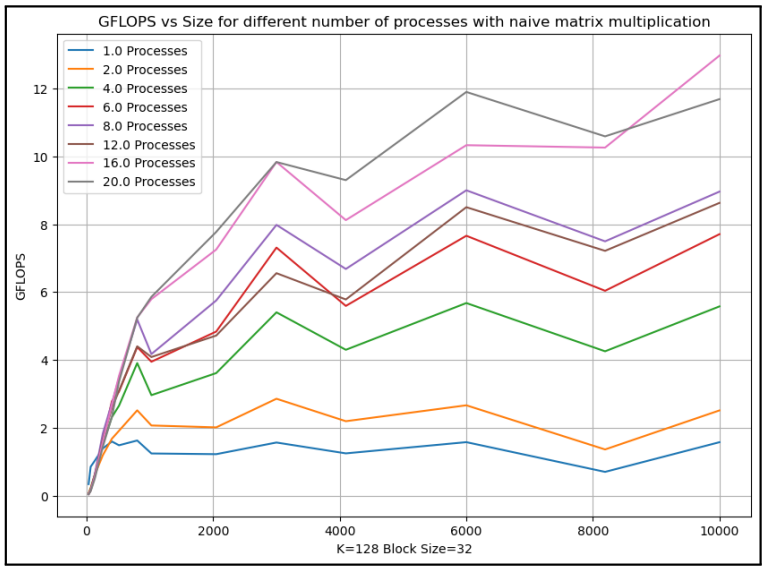


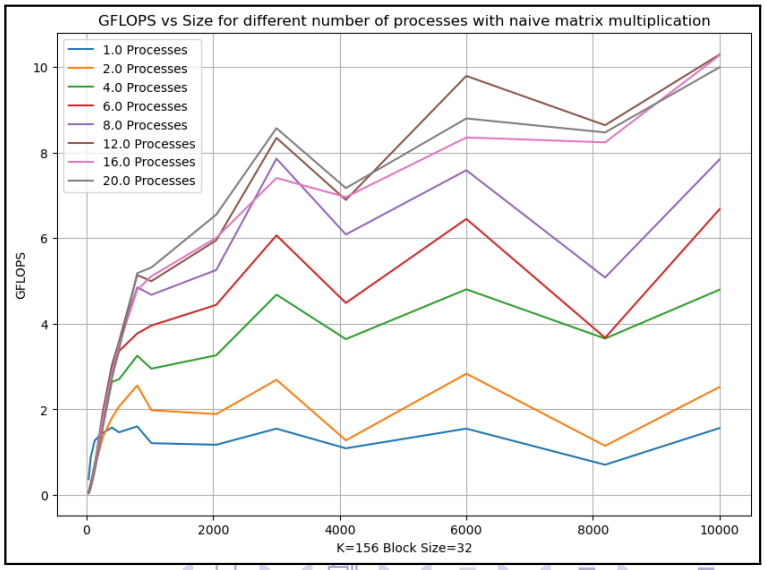


Analisi delle prestazioni - MPI - Matrici rettangolari (1)

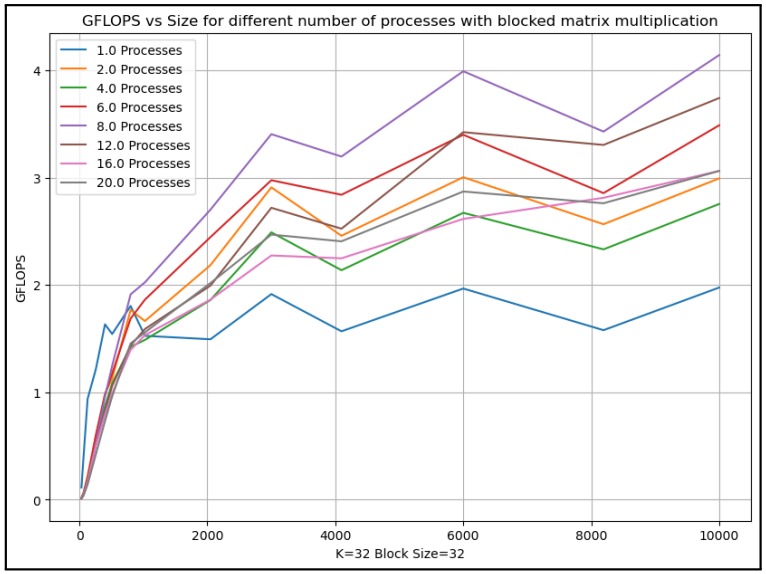


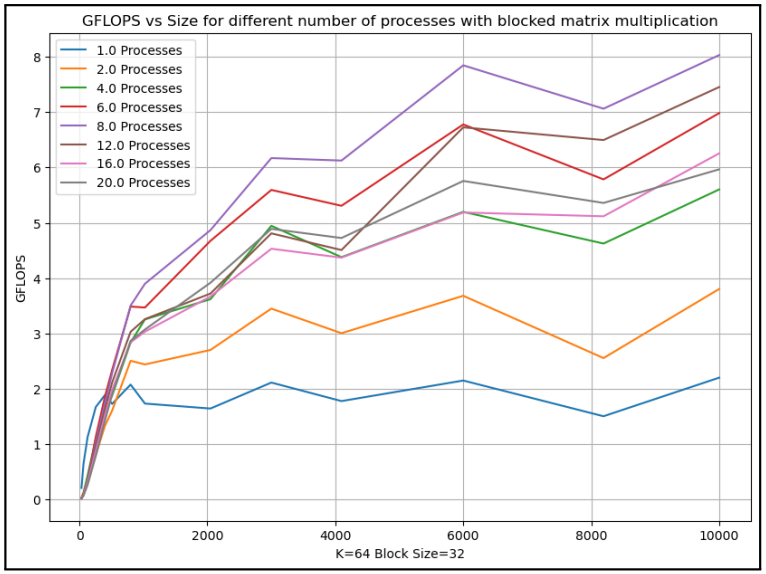


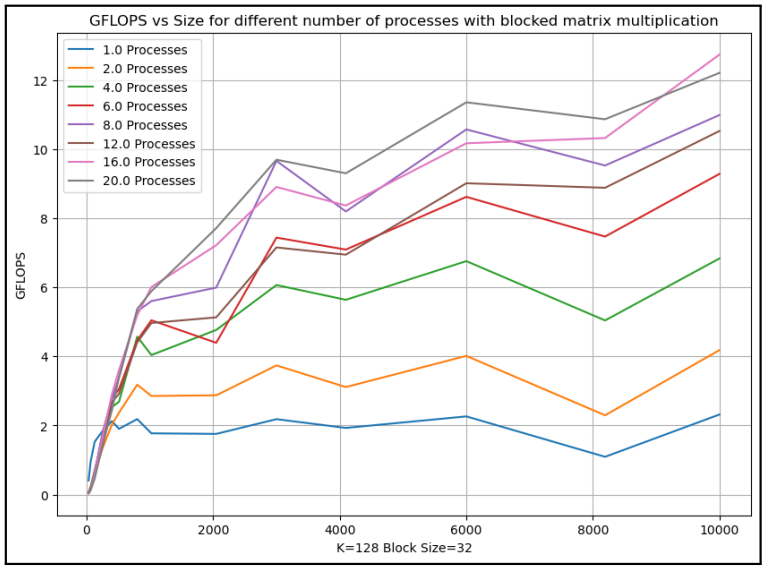


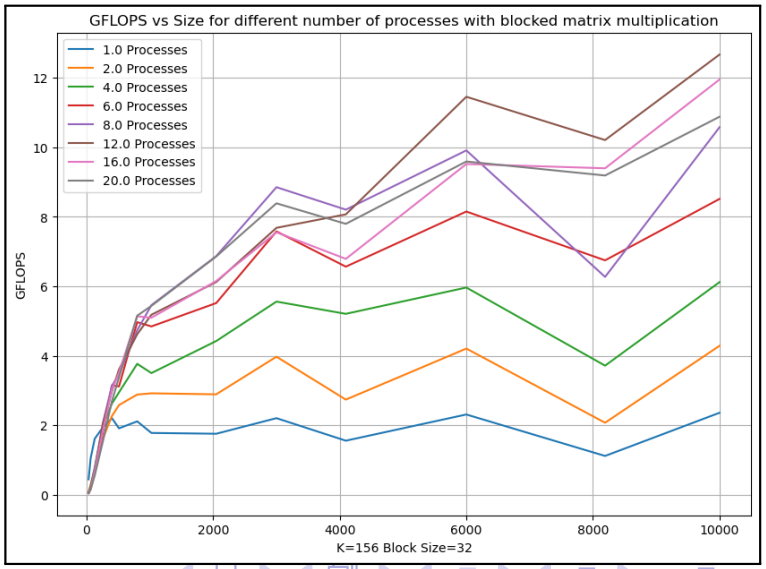


Analisi delle prestazioni - MPI - Matrici rettangolari (2)



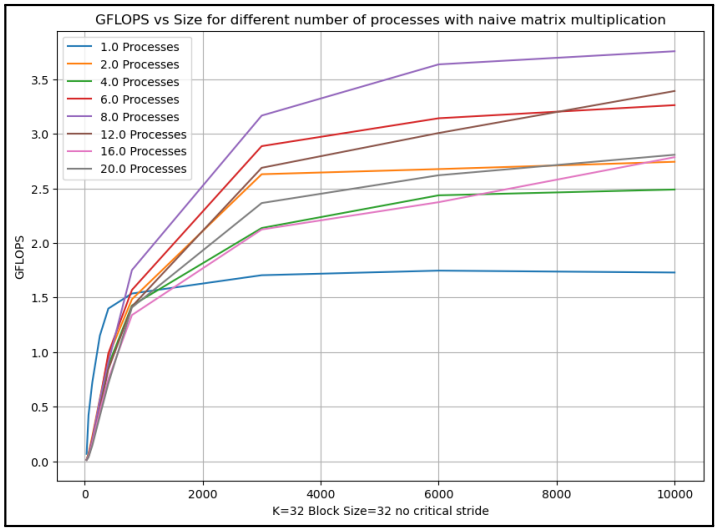


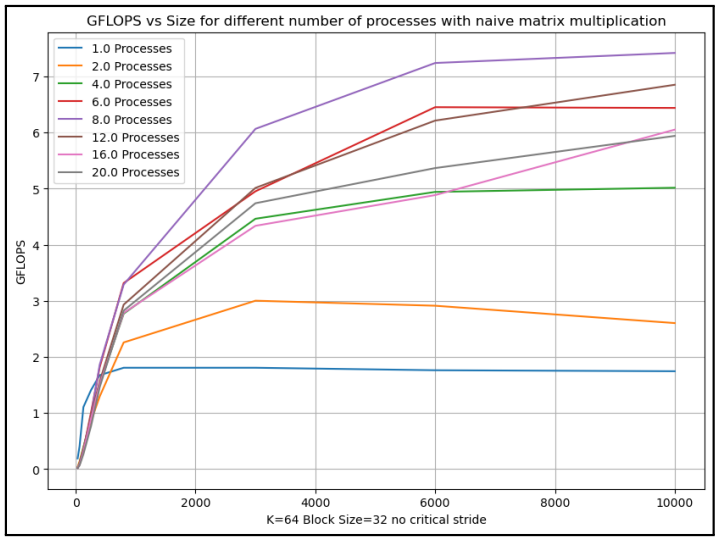


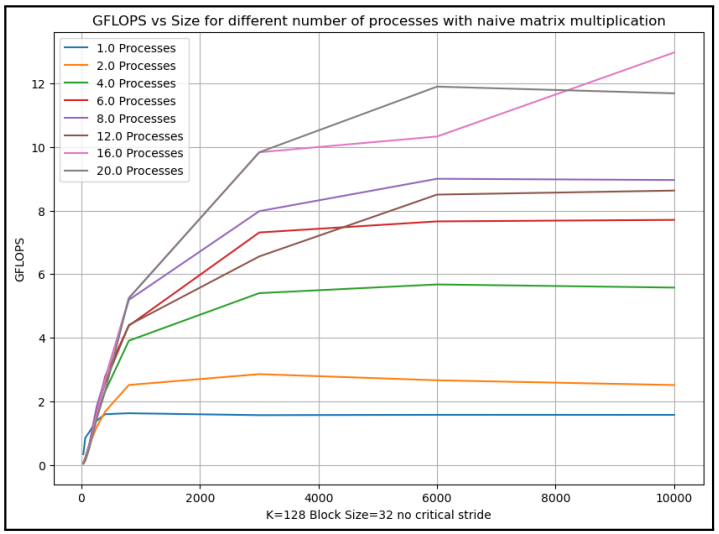


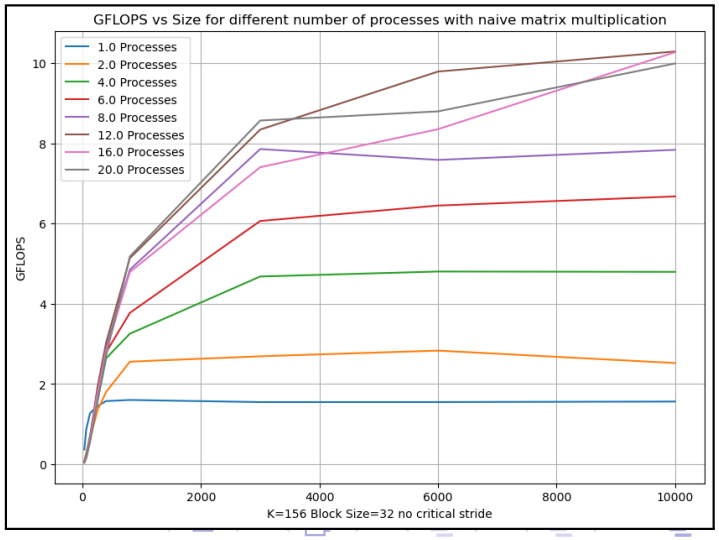
Analisi delle prestazioni - MPI - Matrici rettangolari (3)

Di seguito vengono riportati grafici non affetti dal critical stride.

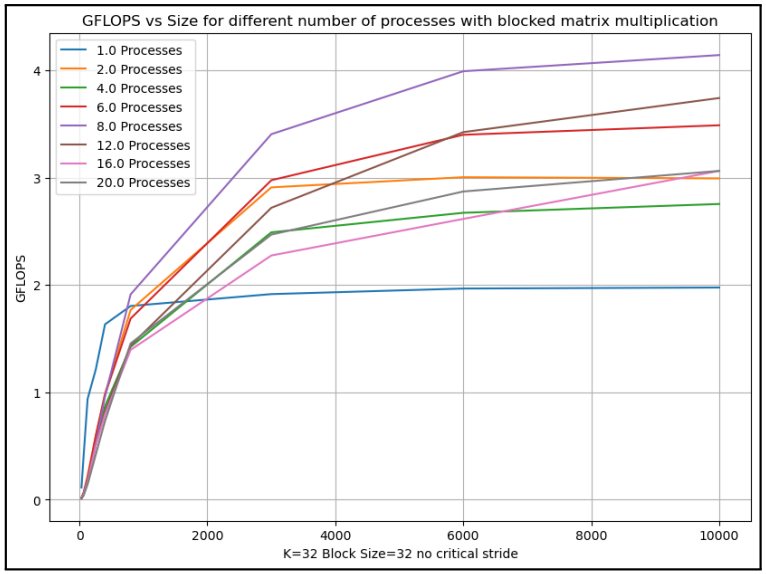


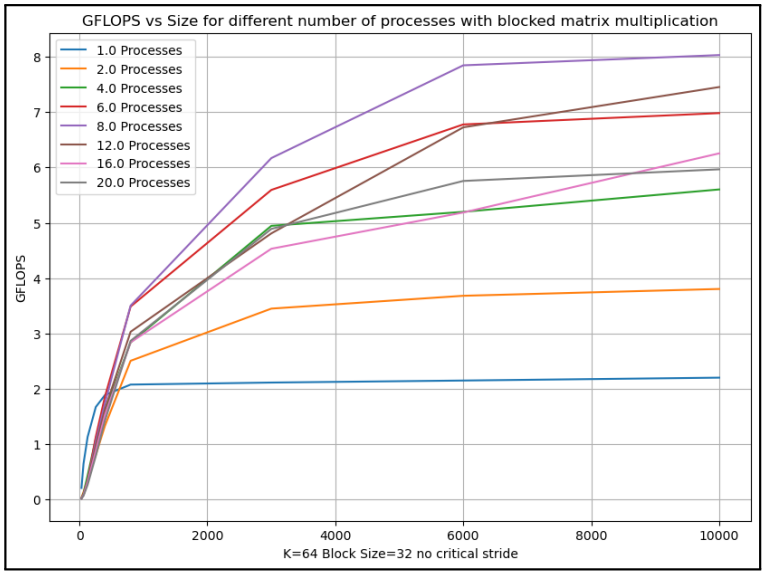


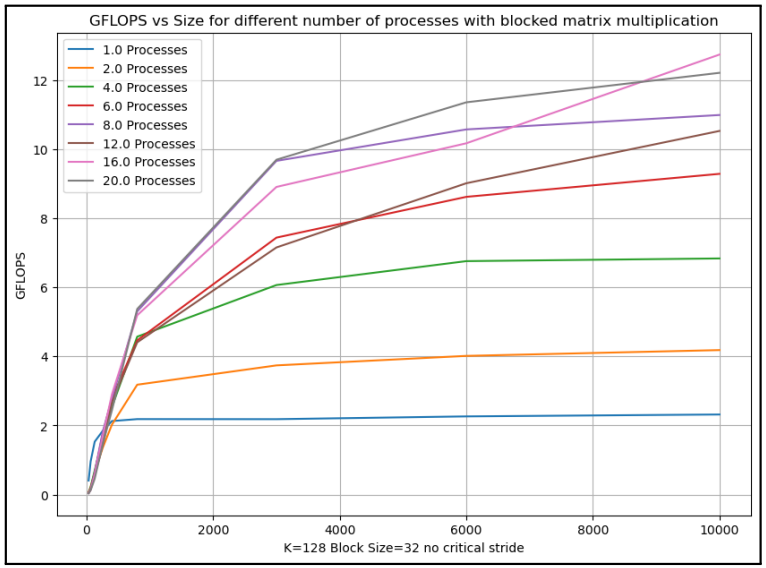


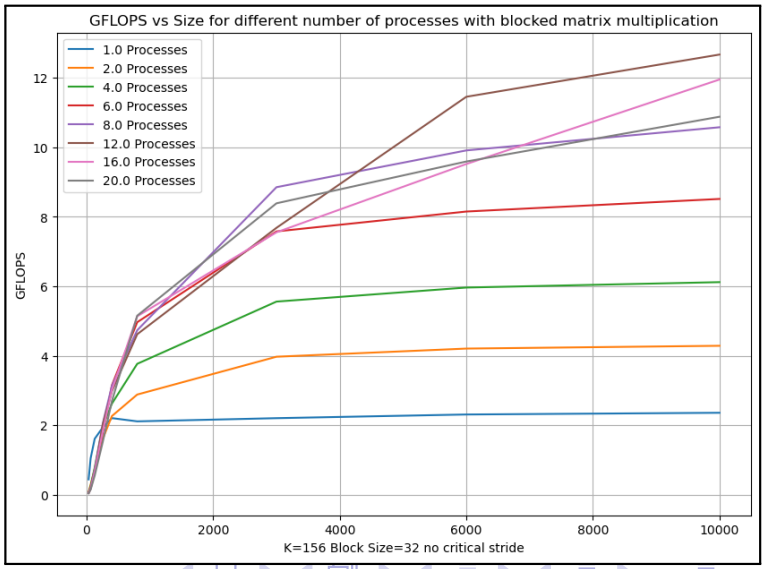


Analisi delle prestazioni - MPI - Matrici rettangolari (4)

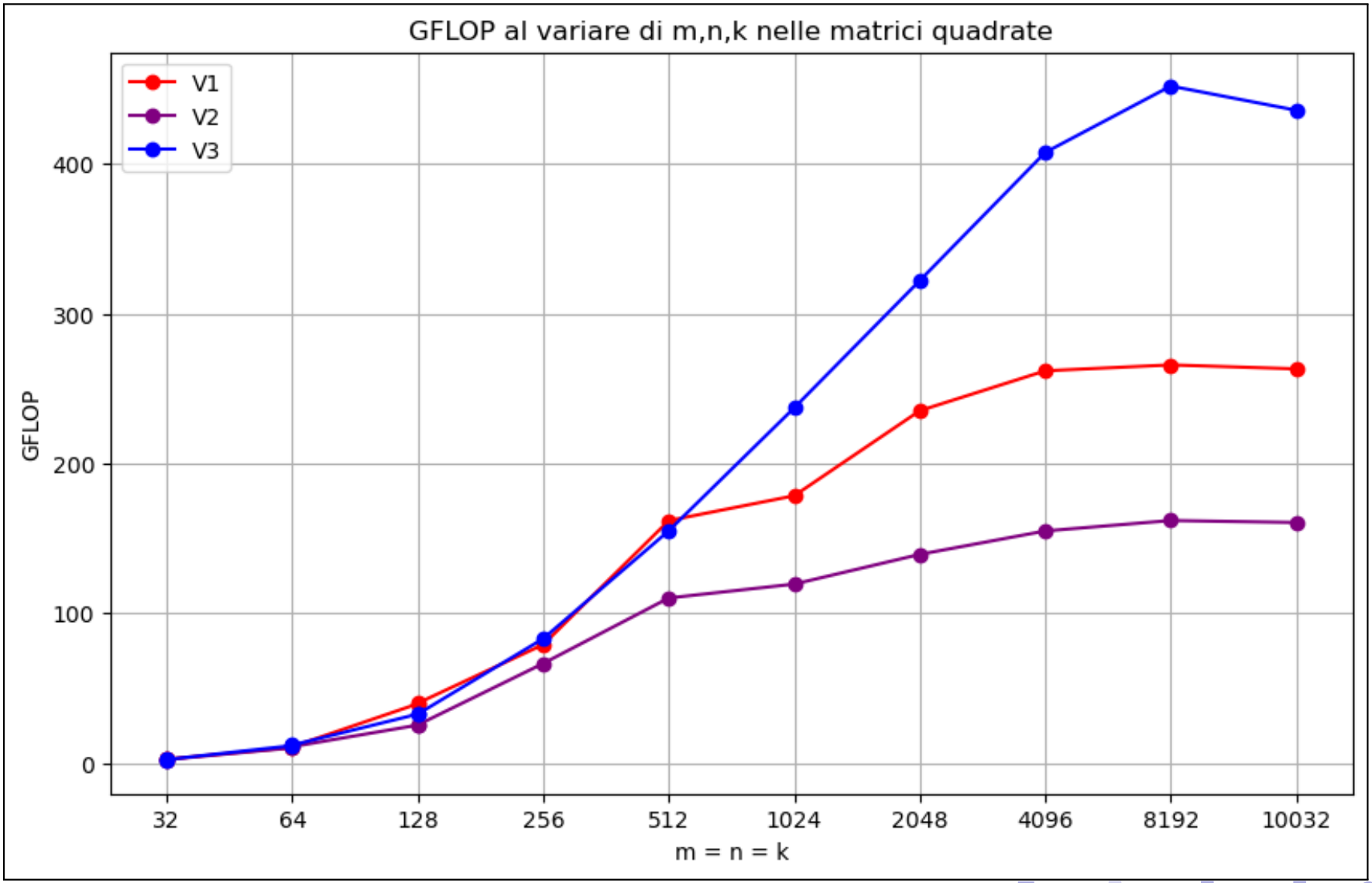








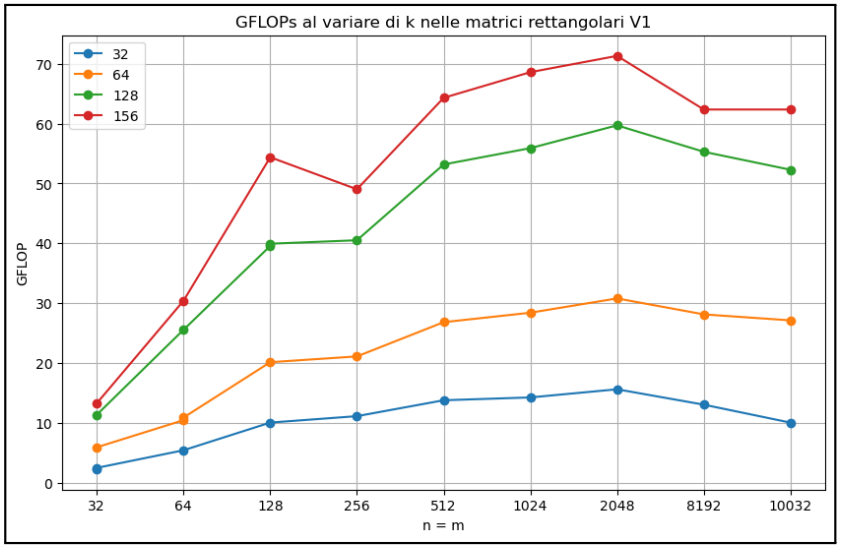
Analisi delle prestazioni - CUDA - Matrici quadrate

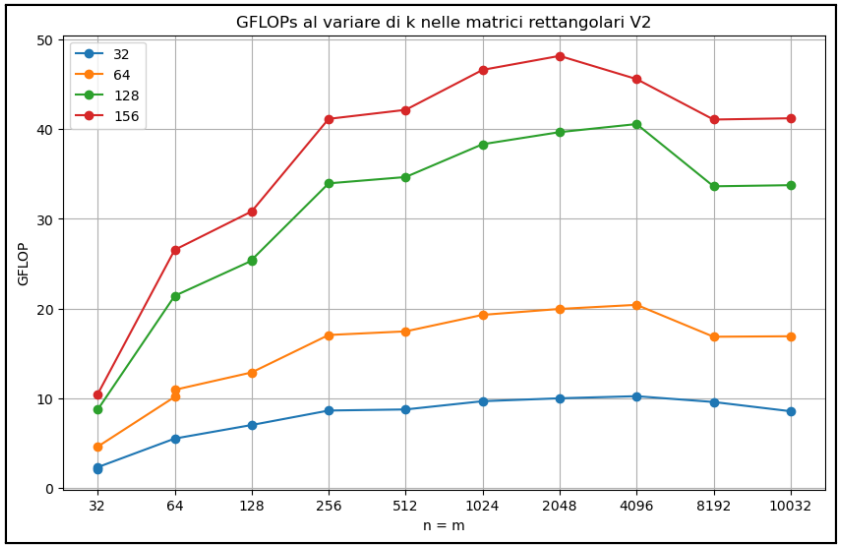


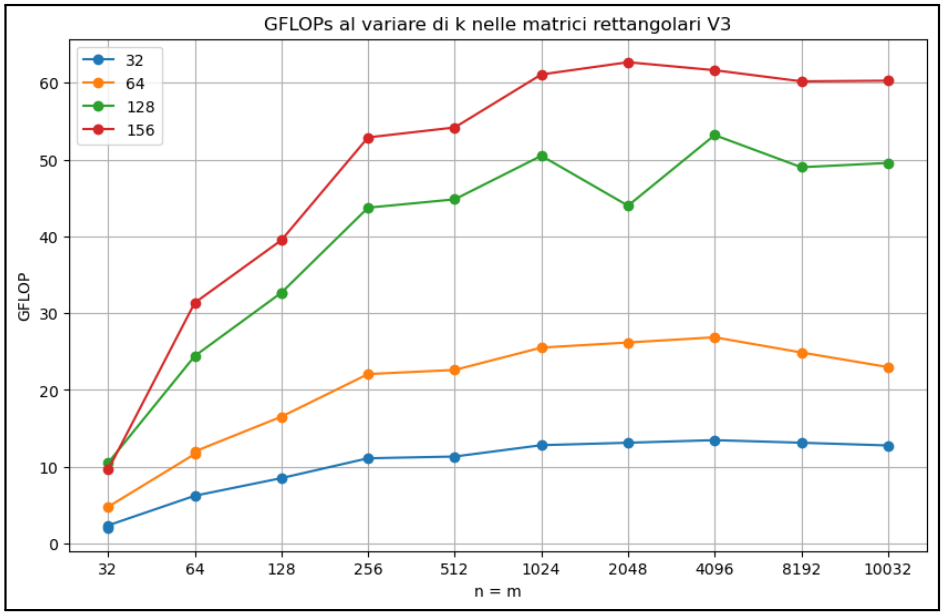
Analisi delle prestazioni - CUDA - Matrici quadrate

- V3 prestazioni migliori
  - dovuto al fatto che essa sfrutta al meglio la shared memory e riduce il numero di accessi alla memoria globale
- V2 prestazioni peggiori di V1
  - Complessità aggiuntiva dovuta alla necessità di gestire un numero di colonne diverso da 1
  - Potrebbe aver portato a ridurre i benefici introdotti e quindi ad ottenere delle prestazioni minori

Analisi delle prestazioni - CUDA - Matrici rettangolari







Analisi delle prestazioni - CUDA - Matrici rettangolari

- Si può notare un calo considerevole delle prestazioni rispetto alla versione quadrata.
  - ogni blocco lavora su una singola riga della colonna A alla volta, dividendola per il numero di thread in quel blocco
  - ogni riga della matrice A è grande k, il quale è minore del numero di thread
  - molti thread per ogni blocco che non fanno nulla, impatto dell'overhead di averli creati
  - prestazioni peggiorano sempre più al diminuire del valore di k proprio perché il numero di thread inutili aumenta

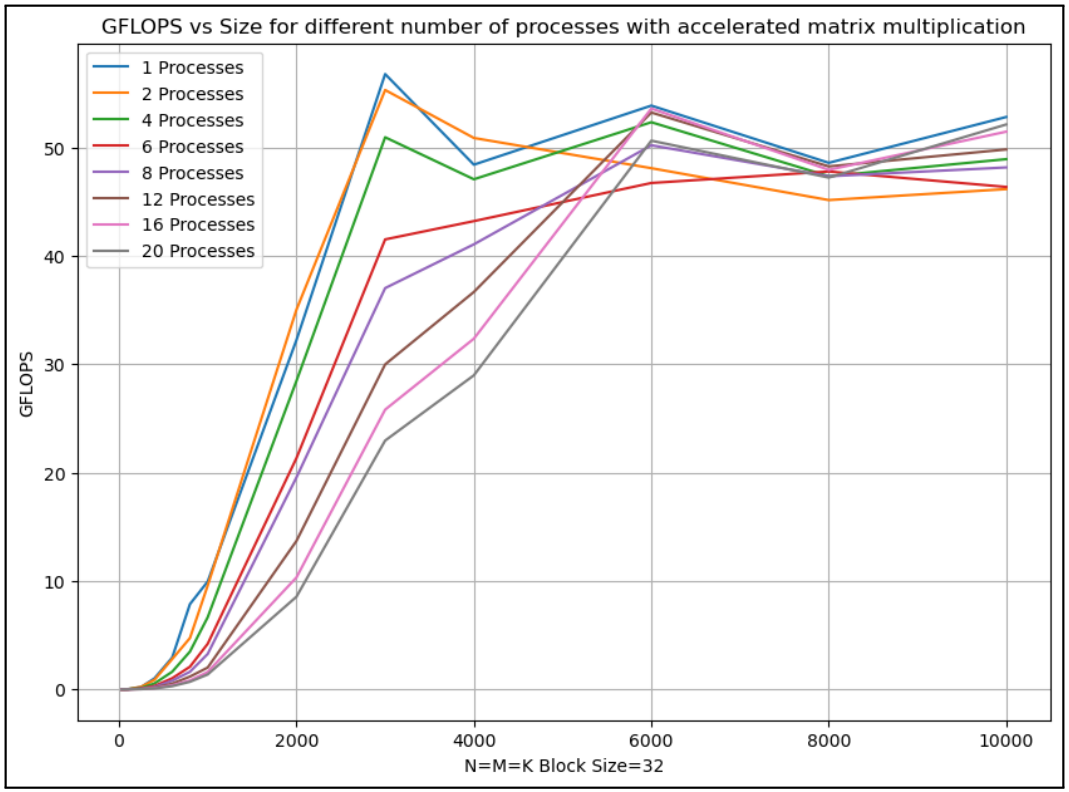
Analisi delle prestazioni - MPI+CUDA

Limiti intrinseci:

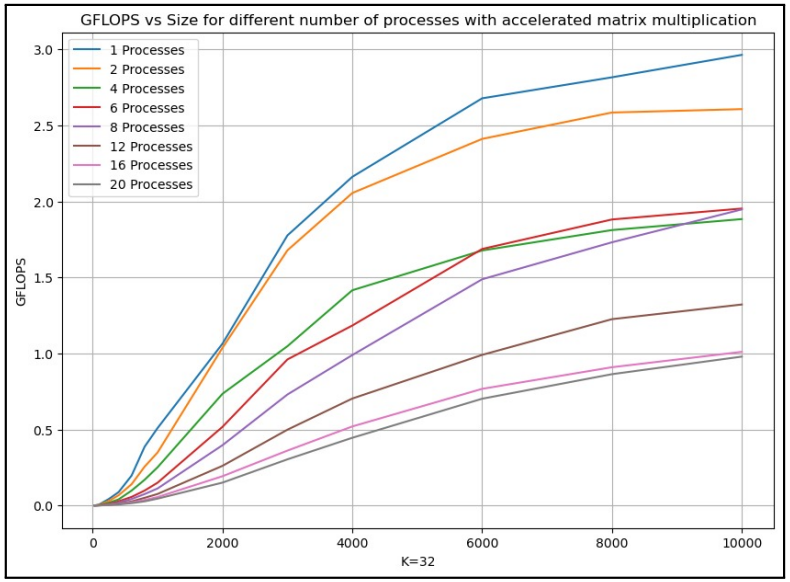
- Nel server dove sono stati fatti gli esperimenti è presente una sola GPU
  - limitazione notevole dato che il vantaggio di utilizzare la soluzione MPI+CUDA è proprio quello di poter utilizzare le GPU di più server
  - quando i processi andranno a tentare di eseguire concorrentemente il nucleo di calcolo su CPU, essi verranno serializzati
  - Sfruttamento limitato del potenziale della soluzione
- Le prestazioni calcolate contengono anche il conteggio del trasferimento della memoria RAM alla memoria globale della GPU
  - non confrontabili direttamente con le prestazioni ottenute con la soluzione solo CUDA

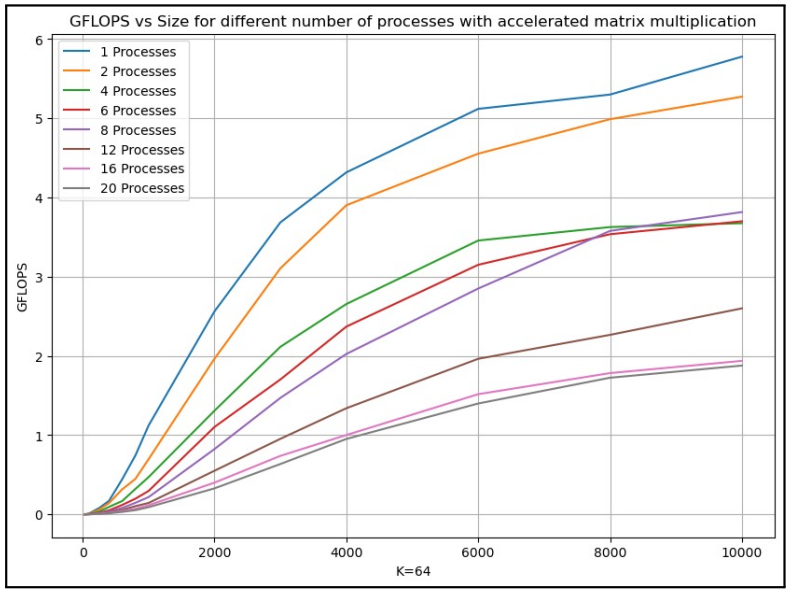
Analisi delle prestazioni - MPI+CUDA - Matrici quadrate

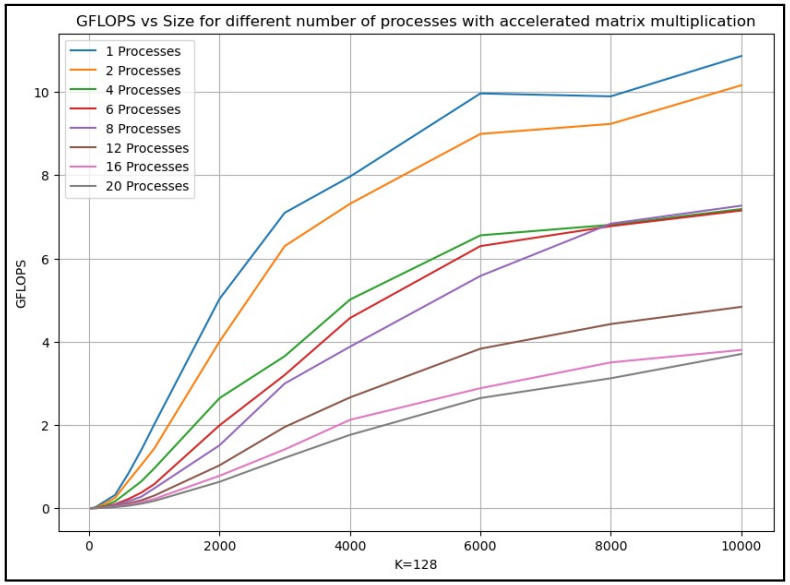
- - Processi $\Rightarrow$ + Performance
- Overhead memoria con matrici grandi

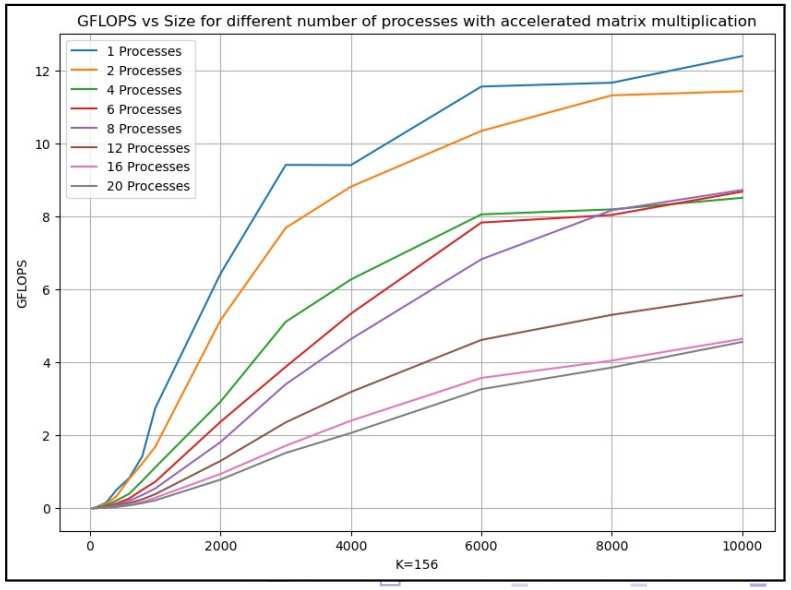


Analisi delle prestazioni - MPI+CUDA - Matrici rettangolari









Analisi delle prestazioni - MPI+CUDA - Matrici rettangolari

- Trend in linea al codice CUDA
  - Al diminuire di k le prestazioni peggiorano
- Al contrario del caso quadrato non c'è un punto in cui c'è un calo delle prestazioni dovuto al trasferimento di memoria
  - Matrici rettangolari sono più piccole $\Rightarrow$ tempo di trasferimento è minore
  - non si arriva al punto di annullare la controparte di calcolo

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
  - luca.falasca@students.uniroma2.eu