Prodotto matrice-matrice parallelo

Matteo Fanfarillo
Facoltà di Ingegneria Informatica
Università degli Studi di Roma - Tor Vergata
Roma, Italia
matteo.fanfarillo@students.uniroma2.eu
0316179

Matteo Federico
Facoltà di Ingegneria Informatica
Università degli Studi di Roma - Tor Vergata
Roma, Italia
matteo.federico.98@students.uniroma2.eu
0321569

***Sommario*—Il presente documento si propone lo scopo di illustrare tutte le fasi di sviluppo del progetto finale assegnato nel corso di Sistemi di Calcolo Parallelo e Applicazioni della facoltà di Ingegneria Informatica dell'Università di Roma - Tor vergata. Lo scopo del progetto è quello di sviluppare un nucleo di calcolo per il prodotto tra due matrici dense, che sia quindi in grado di calcolare in modo parallelo $C = C + A * B$ con $A \in \mathcal{M}(m,k)$, $B \in \mathcal{M}(k,n)$, $C \in \mathcal{M}(m,n)$**
Per lo sviluppo del progetto si sono scelti i framework MPI e CUDA.

***Index Terms*—Calcolo Parallelo, CUDA, MPI, Prodotto Matriciale.**

I. Introduzione del progetto

Il prodotto tra matrici è un'operazione fondamentale nell'ambito dell'elaborazione numerica e dell'algebra lineare. Consiste nell'ottenere ogni elemento della matrice risultante come una combinazione lineare delle righe della prima matrice e delle colonne della seconda. Il prodotto tra matrici ha numerose applicazioni in vari campi, tra cui:

- grafica computerizzata;
- machine learning e data science;
- simulazioni fisiche e ingegneristiche.

Il calcolo del prodotto tra matrici può richiedere un notevole tempo di esecuzione: infatti, coinvolge un numero significativo di operazioni aritmetiche e rappresenta un carico computazionale elevato, specialmente quando si lavora con matrici di grandi dimensioni. Per comprendere meglio l'aspetto computazionale di questa operazione, consideriamo la complessità computazionale del prodotto tra due matrici di dimensioni A e B. Se A è una matrice di dimensioni $m \times k$ e B è una matrice di dimensioni $k \times n$, allora il prodotto tra loro produce una matrice C di dimensioni $m \times n$. Il calcolo di ciascun elemento della matrice C richiede k moltiplicazioni e $k-1$ addizioni. Quindi, il numero totale di operazioni aritmetiche richieste per calcolare tutti gli elementi della matrice C è dato da:

$$OperazioniTotali = m \times n \times (k + k - 1) \tag{1}$$

Questo mostra chiaramente come il numero di operazioni cresca in modo quadratico rispetto alle dimensioni delle matrici coinvolte.

II. Metriche

Nel nostro progetto siamo andati a valutare le performance dei vari nuclei di calcolo utilizzando tre metriche principali:

A. FLOPS

I FLOPS (FLoating point Operations Per Second) rappresentano la metrica di riferimento fondamentale per misurare le prestazioni dei programmi. Sia nel caso di MPI che ne caso di CUDA, vengono calcolati mediante la seguente uguaglianza:

$$GFLOPS = \frac{2mnk}{T} \tag{2}$$

dove T rappresenta il tempo di esecuzione del programma in nanosecondi.

B. Speed Up

Lo Speed Up rappresenta una metrica fondamentale per capire quanto il programma parallelo stia migliorando rispetto a quello seriale. Di fatto, questa metrica è valutata mediante la seguente uguaglianza:

$$SpeedUp = \frac{T_s}{T_p} \tag{3}$$

Dove T_s e T_p rappresentano rispettivamente i tempi di esecuzione per risolvere un dato problema tramite un programma seriale e uno parallelo.

Quindi, per costruzione della metrica, più il programma parallelo è veloce, più il tempo di esecuzione sarà basso (ovviamente avendo un'istanza di problema fissata) e, di conseguenza, lo Speed Up sarà alto.

Tuttavia, questa metrica non sempre può essere valutata poiché richiede di conoscere il tempo di esecuzione dell'algoritmo seriale, il che potrebbe essere difficile da ottenere su istanze di problemi troppo grandi.

C. Errore relativo

L'ultima metrica presa in considerazione, che differisce concettualmente dalle altre, è l'errore relativo. Di fatto, rispetto alle altre due metriche sopra citate, questa non ci permette di vedere quanto il nucleo è veloce, ma ci fa ben capire se i calcoli che sta eseguendo sono corretti, ed è perció fondamentale per avere un programma che sia realmente utilizzabile. Per fare un controesempio, un algoritmo dummy che non fa nulla e ritorna sempre la stessa matrice potrebbe

essere l'algoritmo più veloce al mondo ma non è utilizzabile e utile ai nostri scopi, dato che fornirebbe risultati diversi da quelli attesi. L'errore relativo si calcola con la seguente equazione:

$$\text{Errore Relativo} = \frac{||C_s - C_p||}{||C_s||} \tag{4}$$

dove C_s e C_p rappresentano rispettivamente i due risultati prodotti dalla computazione seriale e parallela.

Come per lo Speed Up, anche questa metrica non è stata calcolata sempre poiché richide di conoscere il valore prodotto dal programma seriale che non è sempre calcolato.

III. MPI

A. Introduzione

Il primo framework utilizzato per parallelizzare il prodotto $C = C + A * B$ è MPI, il quale permette di far eseguire il codice a p processi differenti. Al fine di ottimizzare le prestazioni del programma, è necessario che i p processi si suddividano il lavoro in modo equo, ovvero lavorino su porzioni delle matrici che abbiano le dimensioni più vicine possibili tra loro. In particolare, ai fini del progetto, è richiesto che la distribuzione dei dati e l'interfaccia utilizzata siano simili a quelle usate in ScaLAPACK [1].

Per quanto riguarda la suddivisione dei dati, nella versione più generale di ScaLAPACK è previsto che una delle matrici venga suddivisa secondo uno schema **block cyclic**, come riportato in "Fig. reffig1".

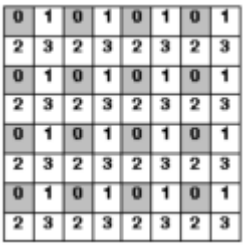

Figura 1. Suddivisione della matrice secondo uno schema block cyclic

Nell'esempio particolare ritratto in "Fig.1" si hanno $p = 4$ processi che si suddividono i dati. La matrice è organizzata in blocchi, dove ciascun blocco è a sua volta diviso in 4 sottoblocchi, ognuno relativo a un processo. I sottoblocchi hanno una topologia regolare ottenuta in modo che sia la più quadrata possibile: nel nostro esempio, si hanno due righe e due colonne. Ulteriori variabili da fissare sono le dimensioni di ciascun sottoblocco in termini di numero di righe e numero di colonne della matrice che compongono il sottoblocco stesso. In definitiva, abbiamo le seguenti variabili in gioco:

- p : numero di processi generati per eseguire il programma in modo parallelo.
- m : numero di righe della matrice C e numero di righe della matrice A.
- n : numero di colonne della matrice C e numero di colonne della matrice B.
- k : numero di colonne della matrice A e numero di righe della matrice B.
- P_r : numero di righe di processi all'interno di un blocco ScaLAPACK.
- P_c : numero di colonne di processi all'interno di un blocco ScaLAPACK.
- MB : numero di righe della matrice comprese in un sottoblocco ScaLAPACK.
- NB : numero di colonne della matrice comprese in un sottoblocco ScaLAPACK (caso in cui la matrice suddivisa secondo lo schema ScaLAPACK sia C).
- KB : numero di colonne della matrice comprese in un sottoblocco ScaLAPACK (caso in cui la matrice suddivisa secondo lo schema ScaLAPACK sia A).

In realtà, per un semplice prodotto tra matrici, non è necessario ricorrere a una suddivisione di tipo block cyclic, che invece viene impiegata in casi più complessi: sarebbe sufficiente avere una semplice suddivisione a blocchi, dove a ciascun processo viene assegnato un unico blocco contiguo della matrice, con MB righe e NB colonne (oppure MB righe e KB colonne). Tuttavia, procederemo comunque con la suddivisione di tipo block cyclic per scopi didattici.

Per quanto riguarda il programma parallelo MPI, proponiamo due possibili soluzioni e ne confronteremo le performance per stabilire quale variante sia migliore:

- Variante basata sulla suddivisione di C: è la matrice C a essere suddivisa secondo uno schema ScaLAPACK.
- Variante basata sulla suddivisione di A: è la matrice A a essere suddivisa secondo uno schema ScaLAPACK.

Prima di analizzare le due soluzioni nel dettaglio, descriveremo come è stato scelto di generare le matrici A, B e C di input.

B. Generazione delle matrici di input

La generazione delle matrici di input è stata implementata in un programma ausiliario (MPI-matrix-gen.c) e avviene pseudo-randomicamente mediante la funzione rand() del linguaggio C con un seed passato tramite argomento del programma (per i nostri esperimenti abbiamo usato un seme fissato con valore 123456789). Qui le matrici vengono memorizzate in appositi file all'interno di una directory passata come parametro, che verranno poi letti dal programma principale per effettuare la computazione.

Le matrici vengono scritte in 3 file differenti ed ogni file come "header" ha i primi 8 byte che indicano la dimensione della matrice, poi la matrice viene memorizzata per righe come se fosse un unico array.

Questa soluzione è risultata semplice da implementare ma, in realtà, non è la migliore dal punto di vista delle prestazioni: in alternativa, si può memorizzare soltanto il seed con cui le matrici sono state create per poi riutilizzare lo stesso seed per generare le stesse matrici sia per l'esecuzione seriale, sia per l'esecuzione parallela.

Per entrambe le versioni del programma, vengono gestiti contestualmente sia il caso $p = 1$, sia il caso $p > 1$, dove:

- Caso $p = 1$: corrisponde all'esecuzione seriale del programma, che viene avviata prima di tutte le esecuzioni parallele. Qui le matrici di input vengono lette interamente da file dall'unico processo in esecuzione. Dopodiché, il programma seriale effettua il prodotto e memorizza su uleriori file le informazioni utili riguardanti la computazione, come la matrice C di output, il tempo di esecuzione e il numero di FLOPS.
- Caso $p > 1$: corrisponde all'esecuzione parallela del programma. Qui le matrici di input vengono lette da file in modo tale che ciascun processo acquisisca soltanto quelle componenti di sua competenza. Dopodiché, viene effettuato il prodotto, viene confrontato l'output dell'esecuzione parallela con quello della precedente esecuzione seriale e vengono registrate le informazioni utili rigurardanti la computazione, come il tempo di esecuzione, il numero di GFLOPS e gli errori assoluto e relativo.

C. Variante basata sulla suddivisione di C

Questa prima variante è nata con la seguente idea: meno i processi devono comunicare fra loro (e quindi attendersi l'uno con l'altro), meno si avrà overhead dovuto all'attesa dello scambio di messaggi e la sincronizzazione dei processi. Pertanto, la domanda che ci siamo posti è la seguente:
"Come si può permettere che ogni processo da quando inizia l'esecuzione eviti il più possibile di comunicare con gli altri processi?"
Fortunatamente, a questa domanda, per il problema di "moltiplicazione fra matrici", si ha una risposta facile e veloce:
"Basta che ogni processo si occupi di effettuare un prodotto scalare tra due vettori".
Infatti, quello che facciamo è suddividere mediante uno schema block cyclic la matrice C in modo tale che ogni processo dovrà prendersi solo i vettori riga della matrice A e i vettori colonna della matrice B inerenti alla computazione dei sottoblocchi C_{ij} a lui affidati; in realtà, qui i sottoblocchi sono dei singoli elementi di C, in modo tale da avere la distribuzione di carico di lavoro più uniforme possibile ($MB = NB = 1$).

In "Fig.2" è riportato uno schema visivo della suddivisione delle matrici, assumendo $p = 4$. Si noti che, per quanto concerne le matrici A, B, il colore giallo (relativo al processo 0) è sovrapposto al colore blu (relativo al processo 1), mentre il colore verde (relativo al processo 2) è sovrapposto al colore rosso (relativo al processo 3).

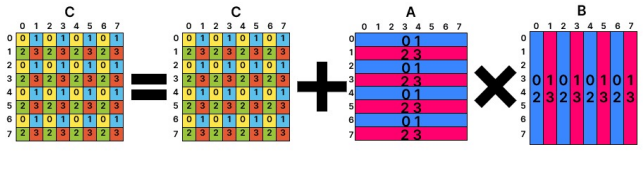

Figura 2. Suddivisione del lavoro in base alla suddivisione della matrice C

1) Riduzione del problema: Questa strategia ha quindi portato alla divisione del problema del prodotto matrice-matrice in tanti sotto-problemi il cui scopo è quello di calcolare il prodotto scalare tra due vettori, ovvero una riga di A e una colonna di B.

Ogni processo quindi conosce:

- chi è lui (ovvero qual è il suo rank);
- quanti processi stanno lavorando sull'istanza corrente del problema;
- la directory dove sono salvate le matrici su cui si deve andare a lavorare.

In primis, ogni processo deve calcolare la dimensione della griglia dei processi (cercando di ottenere una griglia il più quadrata possibile) e poi, a partire dal proprio rank, stabilire quali sono le sue coordinate all'interno della griglia. In seguito, tramite queste informazioni, il processo può iniziare a prelevare dai file i dati che a lui interessano (dato che i processi devono solo leggere dai file non serve alcuna sincronizzazione tra di loro).

In particolare, dato che a ogni elemento della matrice C è associata una riga di A e una colonna di B, vengono prelevate dai file, e l'operazione viene ripetuta per ciascun elemento; in seguito, si effettua il prodotto tra i vettori.

Questo comportamento fissa il massimo grado di parallelismo alla massima dimensione della matrice, ossia, determinata l'istanza di un problema identificato dalle dimensioni M, N, K, il massimo numero di processi che saranno effettivamente operativi può essere $M * N$, che corrisponde al caso in cui ogni processo deve occuparsi di un singolo componente della matrice C.

Questo provoca una scissione sui casi applicativi, infatti così facendo il programma potrebbe essere svantaggiato nel caso in cui la matrice abbia valori M e N molto più piccoli della dimensione K.

2) Occupazione della memoria: Il programma, per poter funzionare, assume che le matrici suddivise tra i processi non vadano a superare la massima capacità di memoria del nodo.

Ma quanta memoria deve avere ogni processo? È facile vedere che ogni processo dovrà conoscere degli elementi sparsi di C, delle righe di A e delle colonne di B; nello specifico, la memoria che ogni processo andrà ad utilizzare sarà:

$$(m_{rank} * n_{rank} + m_{rank} * k + n_{rank} * k) * sizeof(float) \quad (5)$$

dove $m_{rank} = \lfloor (m/p) \rfloor$ e $n_{rank} = \lfloor (m/p) \rfloor$.

Il primo termine dell'addizione sta a indicare lo spazio che ogni processo utilizzerà per tenere traccia della matrice C, il secondo sarà quello utilizzato per le righe della matrice A e il terzo per le colonne della matrice B. Con l'aumentare del numero di processi, diminuisce la memoria che ogni processo deve andare a utilizzare. Nonostante ciò, la dimensione minima sarà sempre vincolata da K. Infatti, se il numero di processi arrivasse a permettere l'assegnazione di un processo per ogni elemento di C, si avrebbe che ogni processo debba avere in memoria spazio sufficiente per $(2 * k + 1) * sizeof(float)$ byte.

3) Prestazioni: I grafici in "Fig.3" e "Fig.4" mostrano il comportamento dell'algoritmo quando viene sottoposto a matrici quadrate di dimensioni dell'ordine delle migliaia di

righe. In particolare, si può vedere l'andamento dei GFLOPS: se fissiamo il numero di processi è costante, mentre aumenta all'aumentare del numero di processi utilizzati.

Lo Speed Up non ha un comportamento altrettanto positivo ma è comunque buono, visto che si ha uno Speed Up sempre costante e che tende all'ottimo, ovvero pari al numero di processi utilizzati.

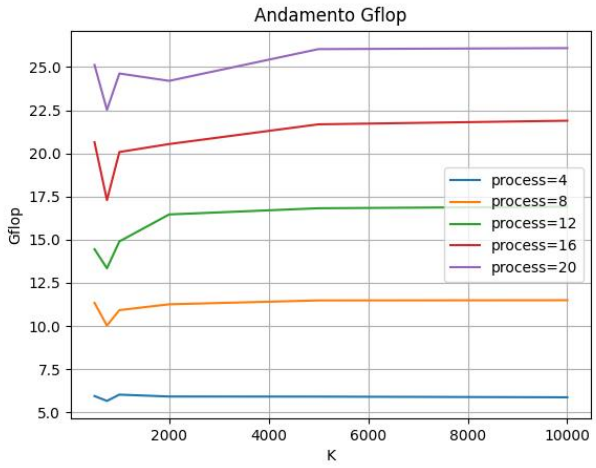

Figura 3. Grafico dell'andamento dei GFLOPS per matrici quadrate

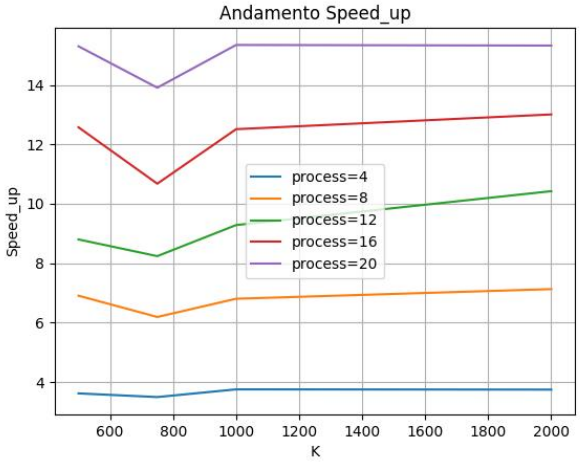

Figura 4. Grafico dell'andamento dello Speed Up per matrici quadrate

Nei grafici in figura "Fig.5" si mostrano invece i risultati prodotti dall'algoritmo in caso di matrici rettangolari, dove la dimensione k è notevolmente più piccola delle altre due dimensioni: in particolare, i valori su cui si è fatta variare la dimensione di k sono 32, 64, 128, mentre le altre due dimensioni sono state fissate a 15000.

Questi casi fanno esplodere l'algoritmo che dà il massimo delle performance arrivando a prestazioni di picco di poco sotto i 100 GFLOPS.

Questi casi così fortunati sono dovuti alla perfetta dimensione delle righe che permettono al compilatore GCC di ottimizzare tramite la tecnica del loop-unroll e tramite l'utilizzo dei registri vettoriali.

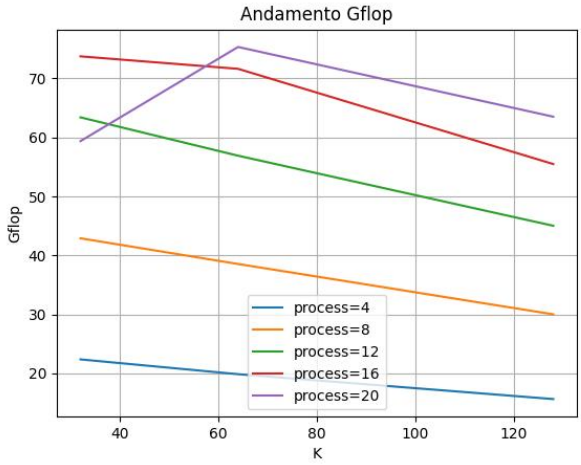

Figura 5. Grafico dell'andamento dei GFLOPS per matrici rettangolari

Il dato più rilevante di questo algoritmo è che il suo errore massimo e, di conseguenza, l'errore assoluto sono pari a 0. Ciò è dovuto al fatto che ogni elemento della matrice C viene computato come se fosse in un programma seriale. Infatti, l'ordine delle operazioni di somme e prodotti per gli elementi della matrice C è lo stesso tra la computazione seriale e parallela.

D. Variante basata sulla suddivisione di A

Questa seconda variante prevede che la suddivisione in ScaLAPACK venga effettuata sulla matrice A, con conseguente suddivisione in righe sia della matrice B, sia della matrice C. Qui non è stato ritenuto necessario fissare $MB = KB = 1$ ma, per semplicità, si è comunque assunto di avere sottoblocchi quadrati (i.e. $MB = KB$), dove la dimensione di tali sottoblocchi viene fatta variare al fine di stabilire empiricamente qual è il valore di MB, KB che permette di raggiungere le prestazioni migliori.

A differenza della variante introdotta precedentemente, questa versione prevede una maggiore comunicazione tra i processi. Ciò è evidente nel momento in cui si nota che più processi insistono sulle medesime righe della matrice C.

Infatti, essendo che ogni elemento della matrice A va a collegarsi a un intera riga di C, i processi che gestiscono un determinato elemento produrrano una semi riga di C, questa semi riga non è finale ma deve subire una fase di "reduce": ovvero una fase in cui i processi che gestiscono elementi della matrice A collegati ad una stessa riga di C, si devono mettere d'accordo al fine di effettuare una somma di tutte le righe parziali prodotte dalla fase precedente. Il lavoro di reduce viene delegato al processo con indice di riga 0. In "Fig.6" è riportato uno schema visivo della suddivisione delle matrici, assumendo $p = 6$.

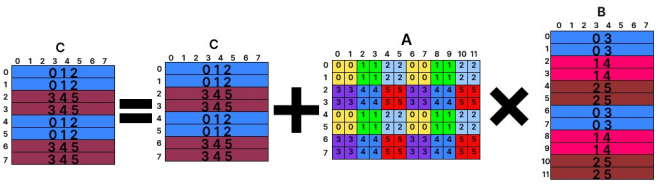

Figura 6. Suddivisione del lavoro in base alla suddivisione della matrice A

E. Errore relativo

Essendo che l'operazione di aggregazione viene eseguita in maniera casuale, ovvero la prima riga che arriva al processo 0 sarà la prima ad essere sommata, l'ordine con cui vengono fatte le somme non è detto sia sempre lo stesso: questo può provocare nei risultati dei valori che possono differire e quindi avere un errore che però dovrebbe essere contenuto. Il grafico in ' 'Fig.7" indica per l'appunto l'andamento dell'errore relativo che ottiene questa versione.

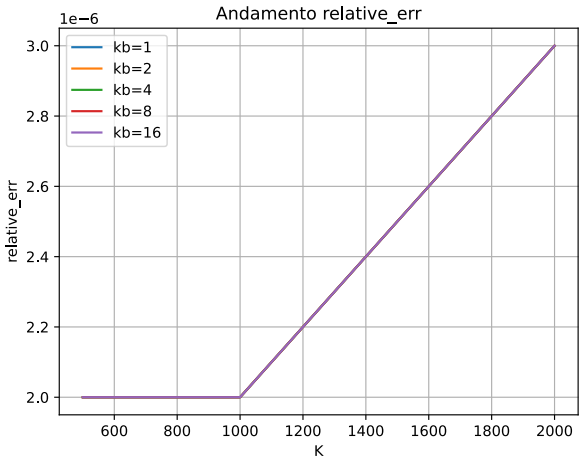

Figura 7. Suddivisione del lavoro in base alla suddivisione della matrice A

F. Confronto tra le due varianti

I grafici in figura "Fig.8" e "Fig.9" mostrano l'andamento dei GFLOPS per i due nuclei di calcolo quando sono sottoposti a due tipi di carichi di lavoro, ovvero matrici quadrate e matrici rettangolari strette e lunghe. Si può vedere come la divisione in ScaLAPACK della matrice C dia performance migliori, tenendo anche conto che la seconda variante ottiene un errore relativo (seppur piccolo) dovuto alla comunicazione tra i processi, mentre la prima variante ha un errore relativo nullo.

Un aspetto strano in questo confronto è lo Speed Up nel caso di matrici rettangolari: la seconda variante vede uno Speed Up molto elevato con un comportamento nettamente migliore rispetto alla prima variante, come riportato in figura "Fig.10".

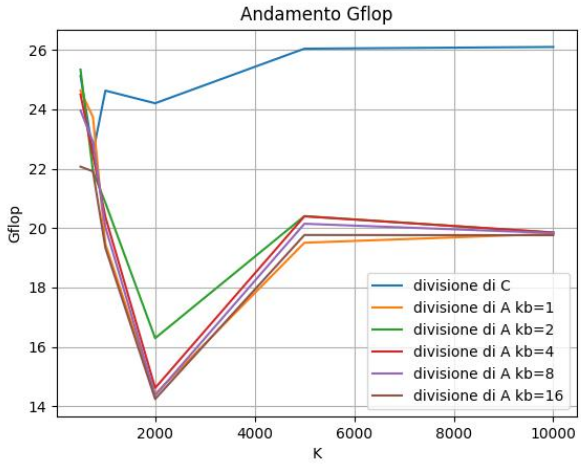

Figura 8. Confronto dello Speed Up in caso di matrici quadrate

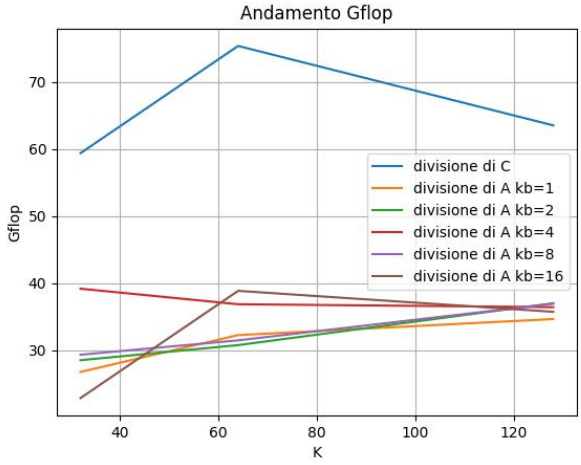

Figura 9. Confronto dello Speed Up in caso di matrici rettangolari

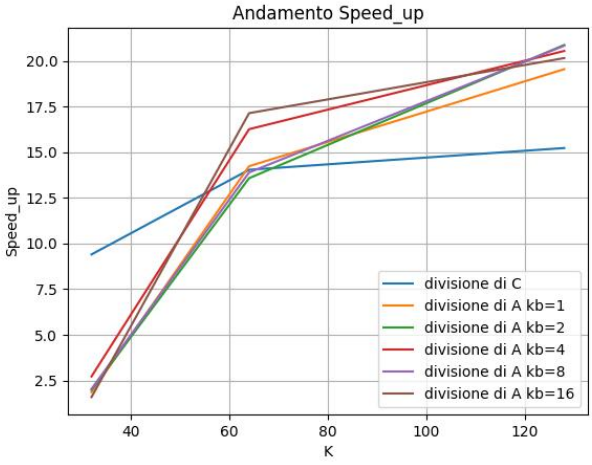

Figura 10. Confronto dello Speed Up in caso di matrici rettangolari

IV. CUDA

A. Introduzione

Il secondo framework utilizzato per parallelizzare il prodotto $C = C + A * B$ è CUDA, il quale permette di far eseguire il codice sulla GPU. Anche qui, per ottimizzare le prestazioni del programma bisogna suddividere equamente le matrici tra i vari thread che vengono attivati in GPU per l'esecuzione del programma.

Ricordiamo che i thread sono organizzati in **warp**, che sono gruppi di 32 thread indicizzati in grado di eseguire la stessa sequenza di istruzioni contemporaneamente, al più sfruttando locazioni di memoria diverse. Dal punto di vista del programmatore, i thread risultano suddivisi in **blocchi**, ovvero gruppi di thread che vengono mandati in esecuzione su una stessa unità di processamento; tali blocchi possono essere organizzati in una dimensione (dove i thread hanno un unico indice x all'interno del blocco), in due dimensioni (dove i thread hanno gli indici x, y) oppure in tre dimensioni (dove i thread hanno gli indici x, y, z). I blocchi, a loro volta, sono disposti in **griglie**, e anch'esse possono avere una topologia unidimensionale, bidimensionale o tridimensionale. A valle di questa descrizione, è chiaro che la dimensione di un blocco debba essere un multiplo della dimensione di un warp (32 thread), in modo tale che all'interno di un blocco vi siano tot warp in grado di eseguire la stessa sequenza di istruzioni eventualmente su zone di memorie differenti. Di conseguenza, da un punto di vista prestazionale, risulta molto furbo assegnare ai 32 thread di uno stesso warp (e, di conseguenza, ai thread di uno stesso blocco) dei dati memorizzati in aree di memoria contigue: infatti, in tal modo, i dati possono essere manipolati con un unico movimento dati in cache. Per assicurarsi che ciò avvenga, se uno stesso thread deve iterare su più aree di memoria, è bene che tali aree di memoria non siano contigue tra loro, bensì abbiano una distanza pari alla dimensione del blocco a cui il thread appartiene.

Un'altra cosa importante da tener presente è che le funzioni mandate in esecuzione in GPU (dette **kernel**) possono usare la **global memory** (memoria globale) e la **shared memory** (memoria condivisa). La prima richiede un tempo maggiore per essere acceduta, mentre la seconda, se viene sovrascritta, richiede una sincronizzazione esplicita tra i thread per poter garantire la consistenza dei dati. A causa di questo compromesso, non possiamo stabilire a priori se convenga o meno utilizzare la shared memory, ed è per questo motivo che, ai fini del progetto, proponiamo ben cinque soluzioni diverse per parallelizzare il prodotto $C = C + A * B$ in CUDA, di cui due senza uso della shared memory e tre con l'uso della shared memory. Prima di analizzare queste cinque varianti, vediamo come è avvenuta la generazione delle matrici in input.

B. Generazione delle matrici in input

In tutte e cinque le versioni, il programma si articola in due parti principali:

- Una prima parte in cui le matrici vengono generate pseudo-randomicamente, si effettua il prodotto seriale in CPU e si inizializzano opportune variabili per tenere traccia delle informazioni relative all'esecuzione seriale, come il risultato corretto del prodotto (matrice C di output), il tempo di esecuzione e il numero di GFLOPS.
- Una seconda parte in cui si riutilizzano le stesse matrici di input generate precedentemente, si invoca il kernel per eseguire il prodotto in GPU, si confronta il risultato ottenuto con quello derivante dall'esecuzione seriale e si registrano le informazioni più rilevanti, come il tempo di esecuzione, il numero di GFLOPS e gli errori assoluto e relativo.

C. Prima variante senza shared memory

L'idea alla base di questa prima variante è sfruttare il funzionamento dei warp e la località dei dati in modo estremamente semplice per migliorare le performance del prodotto tra matrici: di fatto, si tratta di una versione di partenza implementata molto velocemente che rappresenta la base da cui mettere in piedi ragionamenti e raffinamenti successivi.

Tale soluzione prevede una griglia unidimensionale composta da blocchi di thread bidimensionali. Ciascun blocco è relativo a YBD righe della matrice C, dove YBD è pari alla dimensione y dei blocchi di thread. Per suddividere al meglio i dati tra i thread, si impone la cardinalità di ciascun blocco pari al suo massimo ammissibile, che è 1024 thread, per cui le dimensioni XBD, YBD (rispettivamente la dimensione x e la dimensione y), verranno fatte variare in modo tale che la condizione *XBD*YBD=1024* sia sempre verificata. In tal modo, il thread con indici (threadIdx.x, threadIdx.y) all'interno del blocco di indice blockIdx.x lavorerà esclusivamente sulla riga di indice *threadIdx.y+blockIdx.x*blockDim.y* della matrice C, prendendosi un componente ogni XBD di tale riga (quelli tali per cui *columnIdx%XBD=threadIdx.x*). Corrispondentemente, il thread dovrà leggere da:

- l'intera riga di indice *threadIdx.y+blockIdx.x*blockDim.y* della matrice A;
- le intere colonne della matrice B tali per cui *columnIdx%XBD=threadIdx.x*.

D. Seconda variante senza shared memory

Questa seconda variante rappresenta un tentativo di ottimizzare ulteriormente l'utilizzo della memoria: la suddivisione delle tre matrici avviene esattamente come nella versione precedentemente introdotta. L'unica differenza risiede in come viene memorizzata la matrice B, ovvero:
[Primi 32 componenti della prima riga; primi 32 componenti della seconda riga;...; primi 32 componenti dell'ultima riga; secondi 32 componenti della prima riga; ...]

E. Prima variante con shared memory

Questa variante ha lo scopo di confrontare le prestazioni del programma nella sua versione originale con le prestazioni nel caso in cui si utilizzi la shared memory per leggere i componenti della matrice A di interesse. Questo permette di accedere in global memory con lo scopo di leggere la matrice

A una volta sola e, quindi, di velocizzare eventuali accessi ad A successivi.

Questa soluzione può risultare sensibilmente vantaggiosa nel momento in cui ciascun thread debba effettuare numerosi accessi in lettura ad A; se invece gli accessi ad A sono particolarmente pochi, si rischia di non osservare alcun cambiamento delle prestazioni.

F. Seconda variante con shared memory

Questa variante segue un approccio totalmente diverso rispetto alle precedenti. Stavolta, la griglia è bidimensionale e i blocchi di thread sono unidimensionali. L'idea è assegnare un blocco di thread a ciascun componente della matrice C. Gli n blocchi assegnati alla r-esima riga di C prenderanno in carico la r-esima riga di A: ciascun thread di un blocco leggerà in input un componente ogni blockDim.x di tale riga (quelli tali per cui *columnIdx%blockDim.x=threadIdx.x*). D'altra parte, gli m blocchi assegnati alla c-esima colonna di C prenderanno in carico la c-esima colonna di B: ciascun thread di un blocco leggerà in input un componente ogni blockDim.x di tale colonna (quelli tali per cui *rowIdx%blockDim.x=threadIdx.x*).

La shared memory entra in gioco poiché tutti i thread di uno stesso blocco contribuiscono a fornire il risultato di output del medesimo componente di C: di conseguenza, all'interno del kernel è necessario definire un'area di memoria condivisa con un numero di entry pari a *blockDim.x*, ciascuna delle quali ospiterà il risultato parziale prodotto dal corrispondente thread del blocco. Una volta che tutti i thread hanno terminato la loro computazione, si esegue un'operazione di **reduce** (riduzione) per sommare tutti i contributi parziali memorizzati nella shared memory.

G. Terza variante con shared memory

L'ultimo tentantivo che abbiamo fatto è quello di unire le idee precedenti insieme: vengono definiti dei blocchi di thread su uno spazio bidimensionale (blocchi di dimensione 32*32) e ogni thread del blocco è responsabile di un singolo elemento della matrice C, per cui la nostra griglia, anch'essa bidimensionale, ha una dimensione esattamente pari alle volte che un blocco di thread entra nella matrice C.

Ogni thread appartenente allo stesso blocco sarà responsabile anche di due zone (anch'esse bidimensionali) di un'area della memoria condivisa. Tali zone sono una sottomatrice di A e una sottomatrice di B, entrambe di dimensioni 32×32.

Dunque ci chiediamo: qual è l'idea vincente? È quella di far fare a ogni blocco di thread T iteazioni, dove ogni iterazione è composta da una fase di inserimento dei dati all'interno delle due aree di memoria condivisa e una seconda fase dove ogni thread computa un contributo parziale per il valore del prodotto vettoriale tra la riga e la colonna di interesse. Quindi ogni thread porta con sé in shared memory un elemento di A e di B come negli algoritmi classici visti fino ad ora, ma si troverà ulteriori 31 elemnti di A e 31 elementi di B portati in shared memory dagli altri thread del suo blocco; questo permette un notevole risparmio di tempo poiché, quando il thread dovrà effettuare un accesso, lo farà direttamente in memoria condivisa e, quindi, avrà tempi di accesso notevolmente ridotti. Questa cosa è stata riscontrata soprattutto dai dati prestazionali che sono emersi dalle esecuzioni. In definitiva, il presente algoritmo è, a livello di GFLOPS, il più performante.

L'immagine in "Fig.11" indica su quali dati delle matrici i thread del blocco evidenziato in grigio più scuro vanno a lavorare; inoltre, i numeri sulle componenti delle matrici A e B indicano a quale numero di iterazione viene caricato ciascun blocco in memoria condivisa per essere eleborato. Ovviamente è un caso esemplificativo in cui le matrici A, B, C hanno la stessa dimensione, che è anche multipla della dimensione dei blocchi, e la griglia dei blocchi ha dimensione 4×4.

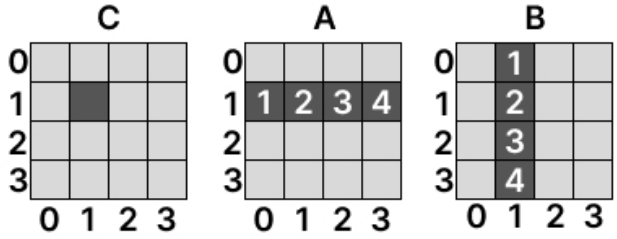

Figura 11. Confronto dello Speed Up in caso di matrici rettangolari

H. Prestazioni

Tutti i test sono stati effettuati sul server di dipartimento avente una scheda video NVIDIA "Quadro RTX 5000" e un processore Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz.

Dai grafici si può vedere come la versione finale di CUDA, utilizzando la shared memory, va nettamente meglio di quelle senza la shared memory. In particolare, i grafici in "Fig.12" rappresentano l'andamento dei GFLOPS su matrici quadrate e rettangolari con dimensione che varia da 500 a 15000; come si può ammirare, il programma con shared memory arriva a toccare 1.3 TFLOPS (TeraFLOPS), e raggiunge queste prestazioni già con istanze di problemi dove le matrici hanno 1000 righe.

Nei grafici in "Fig.13", invece, vengono mostrati i risultati di FLOPS ottenuti su matrici rettangolari: tali risultati sono analoghi a quelli descritti precedentemente.

Una cosa più anomala che invece è stata riscontrata per quanto riguarda i primi test di CUDA è che vi erano delle istanze di problema dove gli algoritmi avevano degli spike di prestazioni. Questi esempi sono rappresentati dai grafici in "Fig.14".
Di fatto, ancora una volta risulta migliore la versione finale del codice che utilizza la shared memory, ma quello senza la shared memory ha un notevole e insolito aumento di prestazioni poiché passa dalla sua media di 300 GFLOPS a toccare anche 480 GFLOPS.

V. Divisione del lavoro nel progetto

- Implementazione del programma MPI, variante basata sulla suddivisione di C: Matteo Federico.

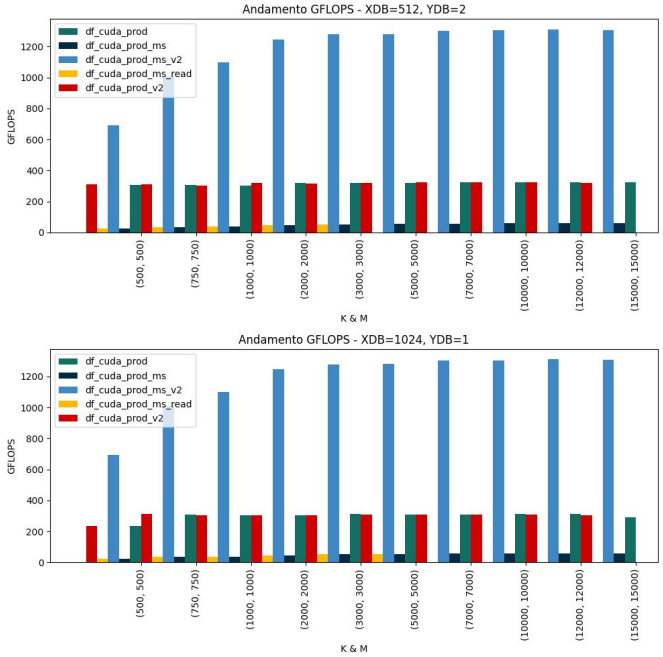

Figura 12. Confronto dei GFLOPS in caso di matrici quadrate

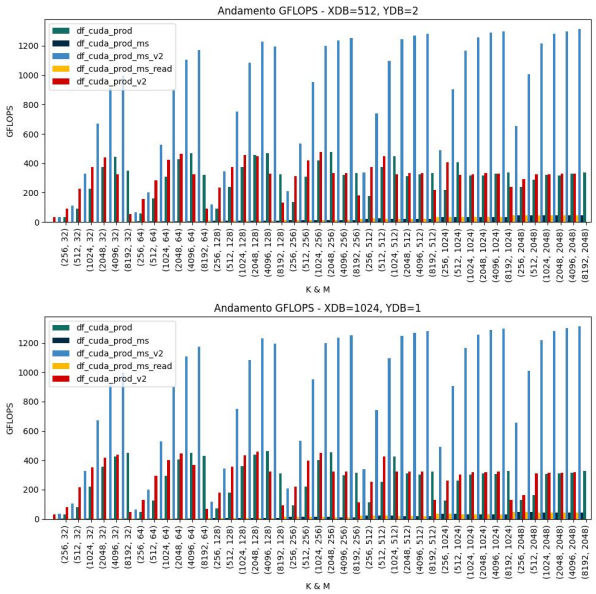

Figura 14. Confronto dei GFLOPS in caso di matrici speciali

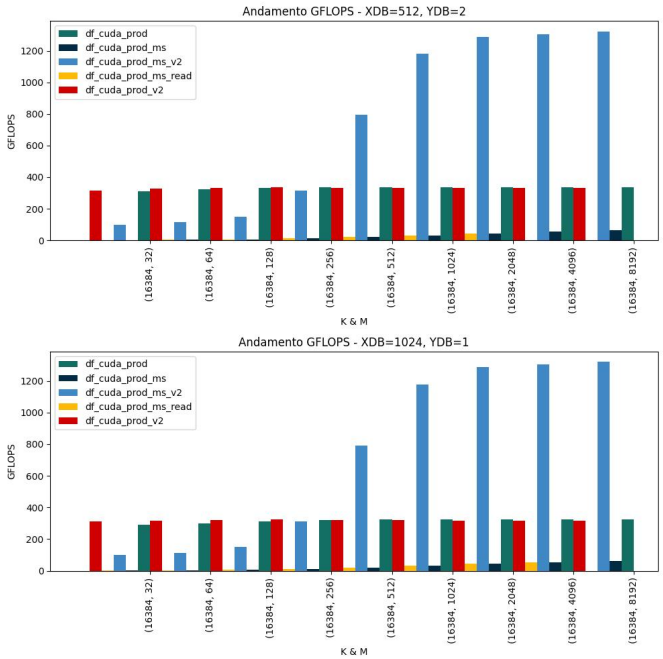

Figura 13. Confronto dei GFLOPS in caso di matrici rettangolari

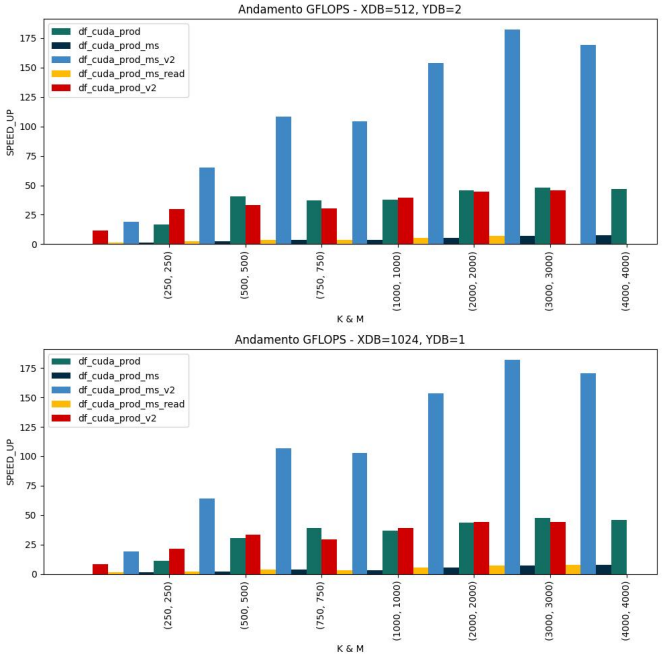

Figura 15. Confronto dello Speed Upfra la computazione in caso di matrici rettangolari

- Implementazione del programma MPI, variante basata sulla suddivisione di A: Matteo Fanfarillo.
- Implementazione del programma CUDA, prima variante senza shared memory: Matteo Fanfarillo.
- Implementazione del programma CUDA, seconda variante senza shared memory: Matteo Federico.
- Implementazione del programma CUDA, prima variante con shared memory: Matteo Fanfarillo.
- Implementazione del programma CUDA, seconda variante con shared memory: Matteo Fanfarillo.
- Implementazione del programma CUDA, terza variante con shared memory: Matteo Federico.
- Raccolta delle statistiche di performance e dei grafici: Matteo Federico.
- Stesura del presente documento: Matteo Fanfarillo e Matteo Federico.

RIFERIMENTI BIBLIOGRAFICI

[1] National Science Foundation, "ScaLAPACK - Scalable Linear Algebra PACKage" https://www.netlib.org/scalapack/, 2010.