Sistemi di Calcolo Parallelo e Applicazioni
Report di Progetto

Autore: Simone Nicosanti
Matricola: 0334319
Email: simone.nicosanti.01@uniroma2.eu

Università degli Studi di Roma Tor Vergata

June 22, 2024

1 Introduzione

1.1 Problema

Il progetto verte sulla realizzazione di un nucleo di calcolo per il prodotto tra due matrici, che sia quindi in grado di calcolare

$$C \leftarrow C + AB$$

dove:

- A è una matrice $m * k$
- B è una matrice $k * n$

Per l'esecuzione dei test sono stati considerati due casi principali:

- Matrici quadrate con $m = n = k$;
- Matrici rettangolari con $m, n >> k$; in questo caso k è stato considerato pari a dei valori tipici della tassellatura dei dati ($k = 32, 64, 128, 156$).

1.2 Metriche

Sono stati considerati due tipi di metriche per i risultati sperimentali:

- Prestazionali
  - FLOPS
  - Speed-Up
- Di errore
  - Errore Relativo

I *FLOPS* sono stati misurati come:

$$FLOPS = \frac{2 * m * n * k}{T}$$

Dove:

- m, n, k sono le dimensioni del problema
- T è il tempo di esecuzione dell'implementazione in secondi

Lo *Speed-Up* è stato misurato come:

$$Speedup = \frac{T_{Sec}}{T_{Par}}$$

Dove:

- T_{Sec} è il tempo di esecuzione sequenziale
- T_{Par} è il tempo di esecuzione parallelo

L'*errore relativo* è stato invece misurato come:

$$RelativeError = \frac{\|C_{Seq} - C_{Par}\|}{\|C_{Seq}\|}$$

Dove:

- C_{Seq} è il risultato calcolato usando l'algoritmo sequenziale
- C_{Par} è il risultato calcolato usando l'algoritmo parallelo
- La norma può essere una qualsiasi norma definita per matrici: nel nostro caso è stata usata la norma infinito

Si noti che le metriche di *Speed-Up* ed *Errore Relativo* possono essere sempre calcolate, ma il loro calcolo richiede la risoluzione del problema in modo sequenziale che, per problemi di grandi dimensioni, può necessitare molto tempo.

1.3 Matrici di Input

Le matrici di input sono state generate in-memory in modo randomico usando la funzione `rand()` di C e un seed impostato a 987654. Una volta generate le tre matrici di input, della matrice C vengono create due versioni, una per l'implementazione sequenziale e una per quella parallela: questo è dovuto al fatto che, per come sono stati progettati i nuclei di calcolo, il risultato dell'elaborazione viene sovrascritto sulla matrice C stessa, quindi non potremmo usare la stessa matrice C di input per entrambi i test. Nel caso MPI le matrici di input vengono generate da un processo root (processo 0 di default), mentre nel caso CUDA vengono generate dall'host e poi copiate nel device.

Per quanto riguarda matrici quadrate, le dimensioni delle matrici di input variano con un ciclo avente i seguenti parametri:

- Dimensione iniziale 250
- Dimensione finale 10001
- Incremento 250

Per quanto riguarda invece le matrici rettangolari, le dimensioni m ed n vengono tenute fisse, mentre la dimensione k viene fatta variare come indicato nella parte introduttiva.

1.4 Esecuzione dei Test

Per ogni possibile configurazione di dimensione, vengono eseguiti tre test di cui poi viene presa la media del tempo di esecuzione. Il risultato del singolo test viene scritto su un file *.csv*.

Per ogni configurazione viene eseguito sempre il test parallelo, mentre il test sequenziale viene eseguito solo fino ad una certa dimensione k per evitare tempi eccessivamente lunghi (nel nostro caso il limite è stato posto a 5001).

2 Sequenziale

L'implementazione sequenziale si basa su:

- Tile Product
- Riordinamento degli indici dei cicli

Entrambe le scelte sono state prese al fine di aumentare il numero di cache hit.

3 MPI

3.1 Suddivisione del Lavoro

Consideriamo i processi distribuiti su una griglia cartesiana bidimensionale.

Per quanto riguarda la distribuzione dei dati nell'implementazione MPI è stata presa in considerazione la distribuzione *ScaLAPACK* dei dati vale a dire una *Block Cycling Distribution* (*BCD*) come quella mostrata in figura 1. In questa distribuzione abbiamo che la matrice viene divisa sia lungo le righe sia lungo le colonne ottenendo dei blocchi che sono poi ridistribuiti ai processi. In una distribuzione di questo tipo quindi, i parametri che dobbiamo considerare sono:

- Le dimensioni della matrice *m, k, n*
- Il numero totale di processi P
  - Il numero di processi sulle righe della griglia cartesiana P_r
  - Il numero di processi sulle colonne della griglia cartesiana P_c
- Le dimensioni del sottoblocco *blockRows, blockCols*

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

Figure 1: Block Cyclic Distribution

Possiamo considerare due alternative per quanto riguarda la suddivisioni delle matrici:

- Distribuire la matrice A secondo una *BCD* e distribuire la matrice B e C per gruppi di righe
- Distribuire la matrice A per gruppi di righe, la matrice B per gruppi di colonne e la matrice C con una *BCD*.

Considerando una *BCD* della matrice A (figura 2), i passi di computazione per il calcolo sarebbero i seguenti:

1. Ogni processo riceve blocchi A_{it} della matrice A
2. Un processo ad indice di colonna j nella griglia cartesiana riceve un blocco di righe della matrice B_t della matrice B

3. Un processo ad indice di colonna $j = 0$ nella griglia cartesiana riceve blocchi di righe C_i della matrice C
4. Il singolo processo calcola il prodotto $A_{it} * B_t = C_i'$ per i blocchi che ha ricevuto ottenendo un sotto gruppo di righe della matrice C
5. I processi che si trovano sulla stessa riga della griglia cartesiana fanno una reduce sul processo ad indice di colonna $j = 0$ sommando $C_i = C_i + \sum_{j=0}^{processGrid[1]} C_i'$
6. Viene fatta una gather sul processo root dei gruppi di righe della matrice C ottenuti come risultato della reduce

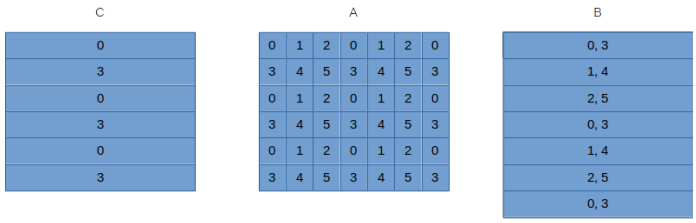

Figure 2: Distribuzione BCD della matrice A

Considerando una *BCD* della matrice C (figura 3), i passi di computazione per il calcolo sarebbero i seguenti:

1. Ogni processo riceve dei gruppi di righe A_i della matrice A
2. Ogni processo riceve dei gruppi di colonne B_j della matrice B
3. Ogni processo riceve dei blocchi C_{ij} della matrice C
4. Il singolo processo calcola $C_{ij} = C_{ij} + A_i * B_j$ per ogni coppia (A_i, B_j) ricevuta
5. Viene fatta una gather sul processo root sui risultati ricomponendo il risultato C

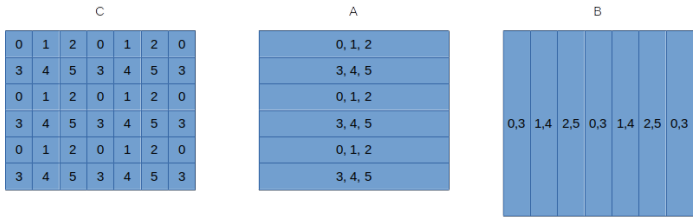

Figure 3: Distribuzione BCD della matrice C

Da questa analisi sommaria dei due procedimenti di calcolo, possiamo notare come:

- La prima alternativa richieda un numero di punti di sincronizzazione maggiore rispetto alla seconda in quanto richiede, oltre alle `send-recv` per la distribuzione, anche delle operazioni di reduce
- La seconda alternativa ha una struttura più semplice e lineare che ricorda maggiormente il classico prodotto riga-colonna e richiede meno punti di sincronizzazione perché ha solo le coppie `send-recv` per la distrivuzione e le coppie `send-recv` di ricomposizione del risultato,

Per questi motivi si è deciso di realizzare un'implementazione seguendo la seconda alternativa.

3.2 Implementazione

3.2.1 Uso dei tipi MPI

La distribuzione delle matrici è stata realizzata usando i tipi MPI; questo permette di:

- Avere maggiore riutilizzo del codice
- Delegare i dettagli di ottimizzazione ad MPI
- Evitare l'uso di molti cicli annidati per realizzare la distribuzione sia seguendo una BCD sia distribuendo per gruppi di righe o colonne

Nello specifico, quando è necessario inviare o ricevere una matrice, vengono creati un massimo di nove tipi MPI per gestire tutti quanti i possibili casi di suddivisione della matrice. Consideriamo la suddivisione delle colonne tra i diversi processi e sia

$$timesGridInCols = \frac{colsNum}{P_c * blockCols}$$

il numero di volte in cui la griglia di processi entra nelle colonne; possiamo allora avere tre casi:

1. Un processo prende un numero di blocchi pari a $timesGridInCols$ e in cui tutti i blocchi hanno dimensione pari a quella del blocco
2. Un processo prende un numero di blocchi pari a $timesGridInCols$ e in cui tutti i blocchi tranne l'ultimo hanno dimensione pari a quella del blocco, mentre l'ultimo ha valore pari a

$$colsNum\%(P_c * blockCols)\%blockCols = finalCols\%blockCols$$

3. Un processo prende un numero di blocchi pari a $timesGridInCols - 1$ e in cui tutti i blocchi hanno dimensione pari a quella del blocco

In figura 4 con lo stesso colore sono rappresentate le parti di riga appartenenti allo stesso tipo: notiamo che a prescindere dal valore di P_c, le possibilità che abbiamo sono sempre le stesse; quest'ultimo aspetto si può capire meglio considerando la figura 5, dove si può notare bene come preso il processo che prende la parte finale della riga, e quindi il tipo 2, tutti i processi prima di lui nella riga prendono il tipo 1, mentre i processi dopo prendono il tipo 3.

Figure 4: MPI - Tipi per suddividere le righe

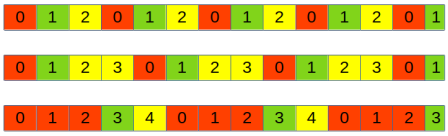

Figure 5: MPI - Suddivisione dei tipi tra processi

Lo stesso ragionamento fatto sulle colonne deve essere fatto anche sulle righe, portandoci quindi a combinare i diversi casi, ottenendo quindi una configurazione simile a quella in figura 6: anche qui riquadri con lo stesso colore afferiscono allo stesso tipo MPI e qui si vede chiaramente come siano necessari al massimo 9 tipi.

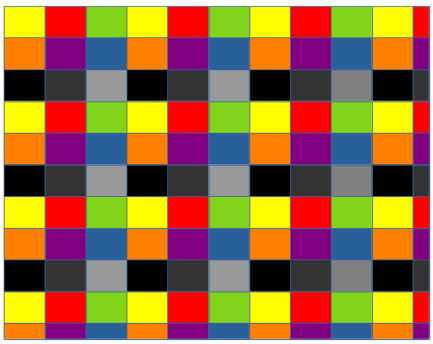

Figure 6: MPI - Tipi per suddividere la matrice

Questi tipi sono stati realizzati usando prima degli `MPI_type_indexed` per dividere le colonne nei tre casi spiegati sopra, per poi dividere le righe usando degli `MPI_hindexed_block` partendo da ognuno degli `MPI_type_indexed`. Ricordiamo che:

- Gli `indexed` ci permettono di definire delle organizzazioni in memoria in

cui ogni blocco ha dimensioni diverse e spiazzamenti non regolari che è ciò che serve a noi per dividere le colonne

- Gli `hindexed_block` ci permettono di definire delle organizzazioni in memoria in cui le dimensioni dei blocchi sono le stesse ma gli spiazzamenti sono irregolari; inoltre essendo di tipo `h` lo spiazzamento può essere definito in byte piuttosto che come multiplo del tipo di base. Questo è ciò che ci serve per dividere le righe in quanto possiamo vedere blocchi di `indexed` di diverse dimensioni ma a distanza omogenea (come lo sono in questo caso) come `indexed` singoli (quindi di stessa dimensione) a distanza non omogenea e, in particolare, a distanza 1 quando si tratta di `indexed` nello stesso blocco e a distanza $P_r * blockRows$ quando si trovano in blocchi diversi. La scelta della versione `h` è stata dettata dal fatto che lo spiazzamento non è multiplo del tipo di base (in questo caso, uno degli `indexed` definiti precedentemente), bensì del numero di colonne della matrice che stiamo ripartendo.

Creare questi dati richiede di definire degli array che ci permettano di indicare spiazzamenti e dimensioni dei vari blocchi di dati; ciò comporta quindi un uso di memoria sullo stack la cui quantità è:

- Per la creazione degli `indexed` proporzionale a $timesGridInCols$ perché per ogni blocco devo indicare il suo spiazzamento e il numero di elementi che contiene
- Per la creazione degli `hindexed_block` proporzionale a $blockRows * timesGridInRows$

Questo può essere considerato come un caso di tradeoff spazio-tempo: per risparmiare cicli di `send-recv` o di `scatter` abbiamo bisogno di memoria sia per descrivere l'organizzazione dei dati in memoria sia per mantenere i nove tipi MPI di cui viene fatto il commit.

La creazione dei tipi è stata fatta in maniera tale da supportare, usando i dovuti parametri, diversi casi di distribuzione:

- Per blocchi con BCD
- Per gruppi di righe in modo ciclico
- Per gruppi di colonne in modo ciclico

I parametri in questione sono nello specifico la dimensione del blocco di righe e colonne: ad esempio, specificando una dimensione del blocco di colonne pari al numero di colonne stesso, si ottengono soltanto tre tipi validi che dividono la matrice per gruppi di righe (viceversa facendo lo stesso per le righe).

3.2.2 Scatter

Una volta creati i tipi la distribuzione della matrice è molto semplice perché, dato un processo, è sufficiente trovare:

- Il tipo derivato ad esso associato
- Il punto di partenza dei dati spettanti al processo nella matrice di input

Queste due informazioni si possono trovare mantenendosi coerenti con la definizione della matrice dei tipi, con una indicizzazione opportuna dettata dalla posizione del processo nella griglia dei processi, e con un'attenzione al modo in cui la matrice viene distribuita (se per BCD oppure per gruppi di righe o colonne). Quest'ultimo punto è quello a cui bisogna dare un'attenzione particolare: supponiamo di avere sei processi distribuiti su una griglia $2 * 3$ e di distribuire la matrice per gruppi di righe; in questo caso tutti i processi sulla stessa riga della griglia devono ricevere gli stessi dati (come nel caso della matrice A in figura 3) quindi tutti i processi sulla stessa riga si comportano come il processo 0 e il loro indice di colonna viene considerato 0; generalizzando i processi che si trovano sulla riga i prendono lo stesso tipo del processo ad indice di colonna $j = 0$ che si trova sulla stessa riga (il ragionamento è analogo se distribuiamo per gruppi di colonne).

Notiamo ancora che l'uso dei tipi MPI permette di eseguire l'invio dei dati ad un processo usando una singola operazione di `send`. Per quanto riguarda la `recv` corrispondente, ricordiamo prima di tutto che la definizione di tipi MPI ci permette di descrivere l'organizzazione con cui i dati compaiono in memoria; quando riceviamo i dati non vogliamo che i dati siano salvati in memoria con la stessa organizzazione con cui comparivano nel processo mittente, bensì che siano ricevuti in modo continuo: per farlo quindi possiamo fare la `recv` di un numero di elementi pari al numero di elementi della matrice che il processo deve ricevere, anche questo calcolabile partendo dall'indice del processo nella griglia dei processi.

3.2.3 Prodotto

Una volta ricevuti i dati da elaborare, il singolo processo esegue il tile product sequenziale.

3.2.4 Gather

Fatto il prodotto tra le sotto matrici, i processi eseguono una `send` inviando i dati secondo un pattern contiguo, mentre il processo Root, responsabile di ricostruire e sovrascrivere il risultato nella matrice C, li riceve usando i tipi di dato di cui sopra: questo assicura che, facendo attenzione al tipo di dato MPI che si indicizza nella matrice dei tipi e alla posizione della matrice C che si specifica nella `recv`, i dati verranno sovrascritti nella posizione corretta di C.

3.3 Risultati Empirici

I test sono stati eseguiti considerando come dimensioni per la BCD:

- Blocco di riga di dimensione 1

- Blocco di colonna di dimensione 1

Questo permette la divisione del lavoro più equa possibile tra i processi.

I tempi sono stati presi non considerando il tempo impiegato per la distribuzione dei dati e per la ricostruzione del risultato finale, né per la creazione di comunicatori atti alla corretta esecuzione dell'implementazione: l'unica cosa considerata è il tempo che impiegano i processi ad eseguire il prodotto delle sottomatrici che hanno ricevuto.

3.3.1 Caso Quadrato

Le prestazioni ottenute con MPI nel caso quadrato sono mostrate nelle figure 7 e 8.

Come possiamo vedere gli andamenti sono quelli che ci aspettavamo: a partire da una certa dimensione in poi, i GFLOPS e lo SpeedUp aumentano con il numero di processi e si mantengono più o meno costanti a parità di numero di processi. La costanza a parità di numero di processi è dovuta al fatto che, a partire da una certa dimensione in poi, il problema satura le CPU e quindi la dimensione del problema diventa ininfluente sulle prestazioni.

Notiamo che in alcuni casi gli andamenti dei picchi e dei crolli nelle curve sono molto simili, cosa che succede, ad esempio, per le coppie (20, 16) e per le coppie (4, 8), così come possiamo notare una certa regolarità negli alti e bassi della stessa curva: ciò può essere attribuito al tiling fatto nel prodotto dai singoli processi.

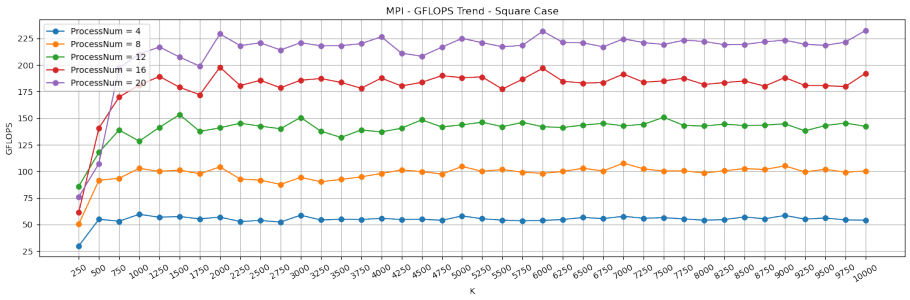

Figure 7: MPI - GFLOPS Caso Quadrato

Per quanto riguarda lo speed up possiamo notare come in molti casi ci si avvicini allo speed up ottimo, ovvero pari al numero di processi utilizzati.

3.3.2 Caso Rettangolare

Le prestazioni ottenute con MPI nel caso rettangolare sono mostrate nelle figure 9 e 10.

Anche nel caso rettangolare i risultati sono quelli che ci aspettavamo e sono molto simili, in termini di andamento e comportamento, a quelli del caso quadrato.

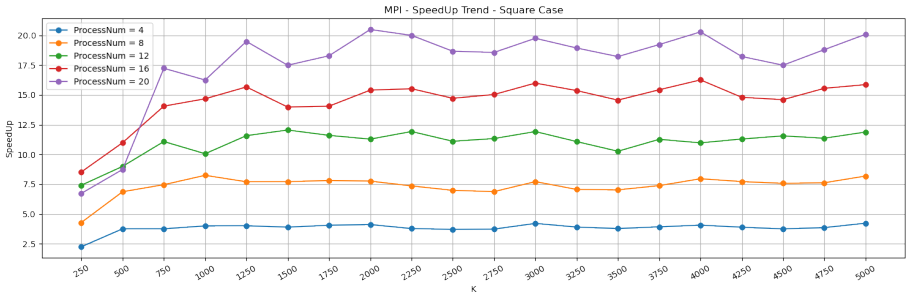

Figure 8: MPI - SpeedUp Caso Quadrato

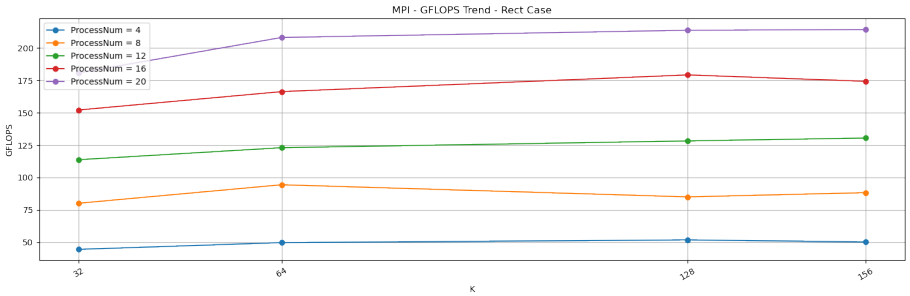

Figure 9: MPI - GFLOPS Caso Rettangolare

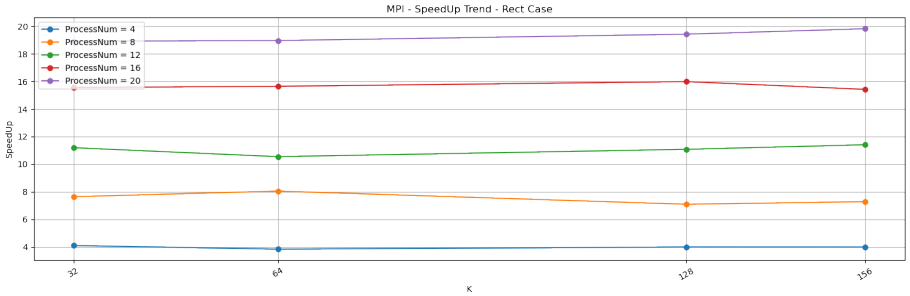

Figure 10: MPI - SpeedUp Caso Rettangolare

3.3.3 Errore Relativo

In tutti i casi considerati, l'errore relativo prodotto dal nucleo di calcolo è stato nullo. Notiamo inoltre che non ci sono casi in cui un elemento della matrice C è calcolato facendo operazioni, come la somma, in ordini variabili, quindi anche i risultati prodotti da esecuzioni diverse a parità di input daranno lo stesso risultato.

4 CUDA

4.1 Aspetti preliminari

Sono state implementate diverse versioni del prodotto tra matrici, ognuna delle quali rappresenta un diverso grado di raffinamento dell'implementazione dell'algoritmo su GPU.

Anche in questo caso si è cercato di avere un approccio orientato a libreria: i dettagli dell'uso di CUDA sono nascosti al chiamante che deve passare al metodo principale le matrici allocate nell'host e le relative dimensioni, mentre la libreria si occupa di copiarle sulla memoria globale della GPU per poi far partire una delle versioni dell'algoritmo, specificabile come parametro alla libreria stessa. La copia sulla memoria globale è fatta usando:

1. `cudaHostRegister`
2. `cudaMallocPitch`
3. `cudaMemcpy2D`

Al termine del calcolo con GPU la matrice C viene copiata nella memoria dell'host dalla memoria globale usando la `cudaMemcpy2D` per poi deallocare le matrici sul device.

4.2 Implementazione

4.2.1 Kernel 0

Il kernel 0 è un'implementazione naive del prodotto tra matrici realizzato su GPU. Le operazioni vengono eseguite:

- Direttamente in memoria globale
- Senza particolare attenzione per il coalesce nell'accesso in memoria
- Usando un accumulatore in un registro locale

La configurazione di lancio è mostrata nel codice 1: lanciando il kernel in questo modo abbiamo che il singolo thread nel blocco è responsabile dell'esecuzione di un solo prodotto riga-colonna, i cui indici sono calcolati partendo dagli indici del thread all'interno del blocco, e quindi della computazione di un solo elemento all'interno della matrice C (figura 11).

Come già detto, in questo kernel non abbiamo il coalesce per gli accessi in memoria globale e questo può essere facilmente illustrato con un esempio: sappiamo che, in un blocco bidimensionale, la dimensione che cresce prima è la x e secondariamente la y; consideriamo due thread successivi all'interno di un warp, ad esempio $(0,0)$ e $(1,0)$; allora questi due thread, supponendo per semplicità che si trovino nel blocco di coordinate $(0,0)$, accederanno al variare di k a :

$$(0,0) \to \begin{cases} A[0,k] \\ B[k,0] \end{cases} \qquad (1,0) \to \begin{cases} A[1,k] \\ B[k,0] \end{cases}$$

```
dim3 blockDim(BLOCK_SIZE, BLOCK_SIZE) ;
dim3 gridDim((m - 1) / BLOCK_SIZE + 1, (n - 1) / BLOCK_SIZE + 1) ;

// Kernel Launch...

int rowIdx = threadIdx.x + blockIdx.x * blockDim.x ;
int colIdx = threadIdx.y + blockIdx.y * blockDim.y ;

//Other code...

... += A[INDEX(rowIdx, kIdx, pitchA)] * B[INDEX(kIdx, colIdx, pitchB)]
```

Listing 1: Kernel 0 - Indicizzazione ed accesso

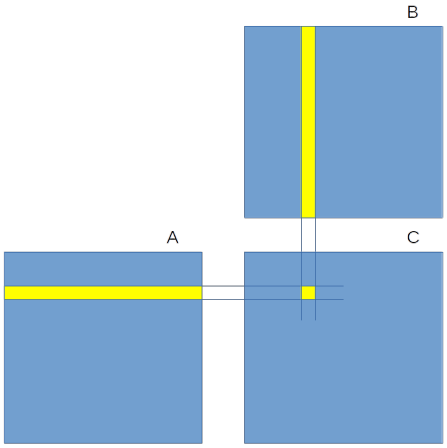

Figure 11: Kernel 0 - Product

Ovvero accedono allo stesso elemento di B, ma ad elementi su righe diverse di A: questo chiaramente comporta una perdita di performance significativa. In figura 12 sono mostrati gli accessi fatti da thread nello stesso warp a parità di indice k.

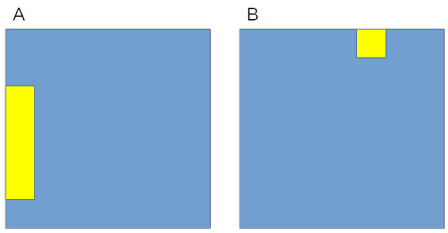

Figure 12: Kernel 0 - Pattern Accesso Memoria Globale

4.2.2 Kernel 1

Il kernel 1 implementa il prodotto nel seguente modo:

- Direttamente in memoria globale
- Sfruttando il coalesce nell'accesso in memoria
- Usando un accumulatore in un registro locale

La differenza sostanziale di questo kernel rispetto al precedente risiede nel modo in cui viene creata la griglia di blocchi: in questo caso infatti viene creata come mostrato nel codice 2 e cioè invertendo le dimensioni m ed n.

```
dim3 blockDim(BLOCK_SIZE, BLOCK_SIZE) ;
dim3 gridDim((n - 1) / BLOCK_SIZE + 1, (m - 1) / BLOCK_SIZE + 1) ;

// Other Code ...

int rowIdx = threadIdx.y + blockIdx.y * blockDim.y ;
int colIdx = threadIdx.x + blockIdx.x * blockDim.x ;
```

Listing 2: Kernel 1 - Griglia ed indicizzazione

Vediamo come questa differenza nel lancio del kernel cambia il pattern di accesso in memoria considerando l'esempio precedente e la nuova indicizzazione

mostrata nel codice 2; in questo caso abbiamo:

$$(0,0) \rightarrow \begin{cases} A[0,k] \\ B[k,0] \end{cases} \qquad (1,0) \rightarrow \begin{cases} A[0,k] \\ B[k,1] \end{cases}$$

Quindi non solo accedono alla stesso elemento di A, ma accedono anche ad elementi contigui nella matrice B. In figura 13 sono mostrati gli accessi fatti nello stesso warp a parità di indice k.

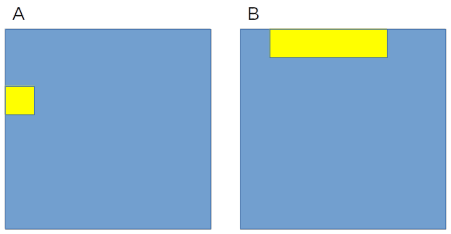

Figure 13: Kernel 1 - Pattern Accesso Memoria Globale

4.2.3 Kernel 2

Il kernel 2 implementa il prodotto nel seguente modo:

- Caricando una porzione dei dati all'interno della shared memory
- Sfruttando il coalesce nell'accesso in memoria globale
- Usando un accumulatore in un registro locale
- Un thread calcola un singolo elemento della matrice C

Una rappresentazione grafica del modo in cui opera il kernel è mostrata in figura 14:

- Il singolo blocco di thread si occupa del prodotto tra le parti marcate in rosso all'interno delle matrici A e B.
- Una porzione di A e B viene caricata all'interno della shared memory, in sottomatrici chiamate subA e subB ed indicate in verde all'interno della figura; avendo la shared memory una dimensione limitata e non potendo quindi caricare un'intera riga di A (che potrebbe essere anche molto grande), iteriamo, per eseguire il caricamento, lungo la dimensione k.
- Ad ogni passo di iterazione del ciclo che scorre lungo la dimensione k, il singolo thread si occupa del prodotto tra una riga di subA ed una colonna di subB (marcate in giallo all'interno della figura)

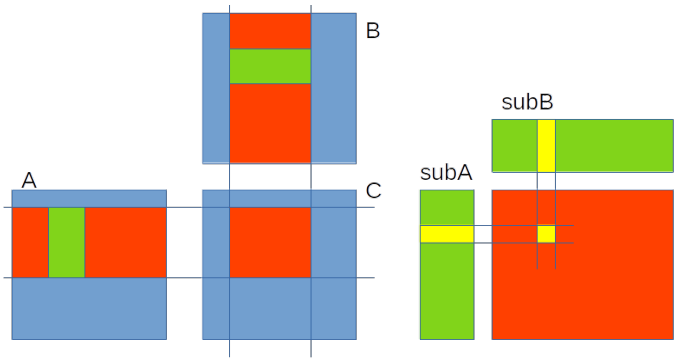

Figure 14: Kernel 2

- Al termine, l'accumulatore viene portato nella matrice C globale

Il lancio del kernel non subisce variazioni rispetto al kernel 1.

L'uso della shared memory permette di ridurre i tempi di accesso ai dati in quanto ha una latenza minore rispetto alla memoria globale ma, di contro, richiede meccanismi di sincronizzazione afficnhé i dati possano essere letti e scritti in modo coerente.

4.2.4 Kernel 3

Il kernel 3 implemta il prodotto nel seguente modo:

- Caricando una porzione dei dati all'interno della shared memory
- Sfruttando il coalesce nell'accesso in memoria
- Un thread calcola una sotto matrice di C attraverso un metodo di tiling: l'accumulatore in questo caso non è un solo valore, ma una matrice di elementi
- Caricamento di una colonna della tile di A e di una riga della tile di B all'interno di registri per evitare accessi multipli alla shared memory

Una rappresentazione grafica del modo in cui opera il kernel è mostrata in figura 15:

- Ogni blocco è responsabile del prodotto tra un blocco di A con MB righe e un blocco di B con NB colonne (rappresentati in rosso nella figura)
- Ogni blocco elabora il prodotto scorrendo lungo la dimensione k con un ciclo di incremento KB (rappresentato in verde in figura)
- Il blocco di A è diviso in tile di dimensione TA, mentre il blocco di B è diviso in tile di dimensione TB e ogni thread è responsabile di moltiplicare una tile di B per una tile di A (in giallo nella figura): questo porta ad ottenere una tile di C di dimensione $TA * TB$

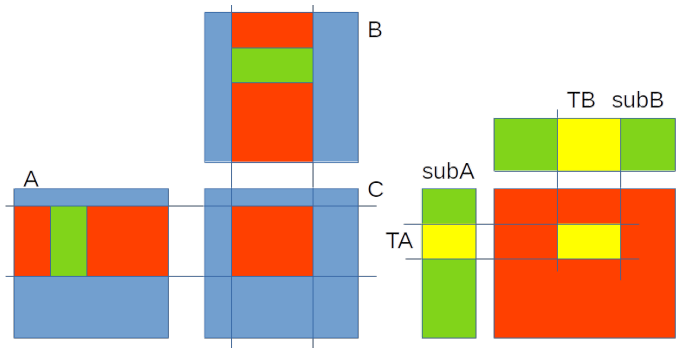

Figure 15: Kernel 3

Essendo implementato in questo modo l'algoritmo, abbiamo che la griglia e il blocco sono come indicate nel codice 3: per come è lanciata la griglia e per come sono fatti i blocchi abbiamo che le tiles di cui il thread deve occuparsi sono date proprio dai suoi indici x ed y all'interno del blocco.

```
dim3 blockDim((N_BLOCK_SIZE / B_TILE_SIZE), (M_BLOCK_SIZE /
↪  A_TILE_SIZE)) ;
dim3 gridDim(((n - 1) / N_BLOCK_SIZE) + 1, ((m - 1) /
↪  M_BLOCK_SIZE) + 1) ;

// Kernel Launch...

int thrTileBIdx = threadIdx.x ;
int thrTileAIdx = threadIdx.y ;
```

Listing 3: Kernel 3 - Griglia ed indicizzazione

Un'altra accortezza aggiunta in questo kernel è l'uso di due cache mantenute all'interno dei registri: all'interno di queste cache vengono inserite, iterativamente, una colonna di subA ed una riga di subB (esse sono dunque di dimensione TA e TB); con i dati così caricati viene fatto il maggior numero possibile di operazioni, in modo da diminuire il numero di accessi alla shared memory dato che essa presenta una latenza maggiore rispetto ai registri.

4.2.5 Kernel 4

Il kernel 4 implemta il prodotto nel seguente modo:

- Caricando una porzione dei dati all'interno della shared memory

- Sfruttando il coalesce nell'accesso in memoria
- Un thread calcola una sotto matrice di C attraverso un metodo di tiling: l'accumulatore in questo caso non è un solo valore, ma una matrice di elementi
- Caricamento di una colonna della tile di A e di una riga della tile di B all'interno di registri per evitare accessi multipli alla shared memory
- I dati nella subA sono caricati trasposti nella shared memory: questo permette di accedere ad elementi contigui nella shared memory quando è il momento di salvare i dati nella cache di registri menzionata nel kernel 3

Da un punto di vista implementativo l'unica cosa che cambia rispetto al caso 3 è solo l'indicizzazione dei dati mantenuti in subA.

4.2.6 Caricamento in Shared Memory

Il caricamento in shared memory è un punto cruciale per tutti i kernel a partire dalla versione 2. Facciamo quindi una breve panoramica su come questo caricamento viene eseguito. Data una matrice, dobbiamo considerare le dimensione del sotto blocco per eseguire il caricamento; consideriamo ad esempio la matrice A (lo stesso discorso vale analogo per B, ma modificando gli indici): il suo caricamento viene eseguito dal codice 4.

```
int startLoadSubRowA = threadIdx.y ;
int startLoadRowA = MB * blockIdx.y ;
int rowsPerBlock = min(MB, m - MB * blockIdx.y) ;

    for (int kSubA = threadIdx.x ; kSubA < min(k - kDispl, KB) ;
    ↪  kSubA += blockDim.x) {
        for (int loadRowIdx = startLoadSubRowA ; loadRowIdx <
        ↪  rowsPerBlock ; loadRowIdx += blockDim.y) {
            subA[INDEX(loadRowIdx, kSubA, colsSubA)] =
            ↪  A[INDEX(startLoadRowA + loadRowIdx, kDispl +
            ↪  kSubA, pitchA)] ;
        }
    }
```

Listing 4: Caricamento sotto matrice A

In particolare abbiamo che:

- Un thread è responsabile del caricamento di uno o più elementi all'interno della subA, a seconda di come sono relazionate tra loro le dimensioni MB, KB e la dimensione del blocco sulla x
- Il primo ciclo serve a gestire il caricamento dei dati lungo la dimensione k: nel caso in cui il KB fosse più grande della blockDim.x sarebbero necessari più cicli di caricamento per caricare lungo la dimensione k
- Il secondo ciclo serve a gestire il caricamento dei dati lungo la dimensione m: nel caso in cui MB fosse più grande della blockDim.y sarebbero necessari più cicli di caricamento (questo è il caso del kernel 3, in cui subA ha dimensione MB, ma blockDim.y è più piccolo perché lanciato come $\frac{MB}{TA}$).

In figura 16 viene mostrato schematicamente quali sono i dati caricati dal thread in posizione $(0, 0)$ di un generico blocco: i dati di dui un thread è responsabile sono evidenziati in giallo, mentre due blocchi gialli sono separati da blockDim.x o blockDim.y elementi

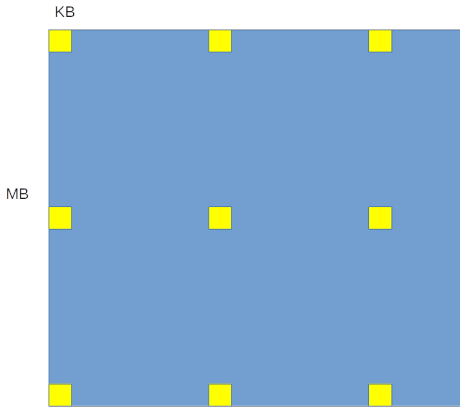

Figure 16: Shared Memory - Caricamento

4.3 Risultati Empirici

I tempi sono stati presi non considerando il tempo impiegato per la copia delle matrici di input ed output sul e dal device:: l'unica cosa considerata è il tempo impiegato dal kernel nel produrre il risultato,

4.3.1 Caso Quadrato

Le prestazioni ottenute con CUDA nel caso quadrato sono mostrate in figura 17.

Prima di tutto notiamo, come ci aspettavamo, il kernel 0 ci restituisce i risultati meno efficienti: il mancato uso del coalesce ha quindi un impatto molto forte sulle prestazioni del kernel.

Un primo salto di prestazioni si ottiene passando dal kernel 0 al kernel 1 con un miglioramento di circa 14 volte: il solo accesso con coalesce alla memoria globale quindi è sufficiente ad incrementare le prestazioni in modo significativo.

Il passaggio dal kernel 1 al kernel 3 invece permette di raddoppiare ulteriormente le performance del nucleo di calcolo: questo è dovuto all'aumento dell'intensità aritmetica del nucleo di calcolo nel passaggio da una versione all'altra. Notiamo anche che nel caso del kernel 3 abbiamo un tempo maggiore di assestamento nelle prestazioni: questo può essere sia dovuto a delle anomalie nel corso delle misurazioni, sia alla particolare scelta degli iperparametri.

Il passaggio dal kernel 3 al kernel 4 non sembra invece dare un miglioramento significativo delle performance ma, al contrario, in alcuni punti sembra dare performance leggermente peggiori.

Il risultato più anomalo in questo contesto è dato dal kernel 2: l'uso della memoria condivisa infatti ci farebbe supporre in automatico delle performance migliori, cosa che però, in questo caso, non sembra trovare riscontro. Il rallentamento potrebbe essere causato dall'introduzione delle `__syncthreads` necessarie a far funzionare l'algoritmo: infatti, sebbene la memoria globale abbia una latenza di accesso maggiore, l'uso di memoria condivisa richiede sincronizzazione dei thread affinché questa possa essere scritta e letta in modo coerente.

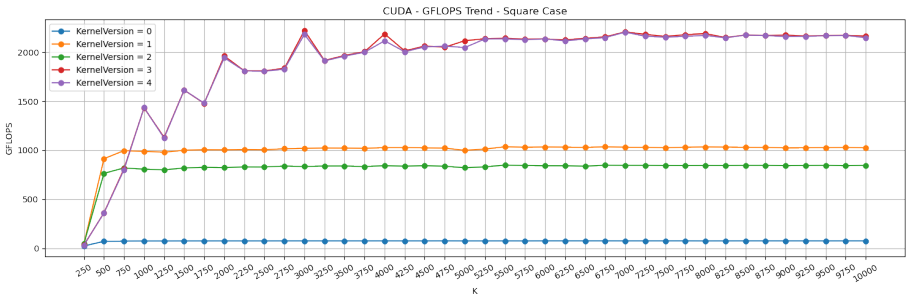

Figure 17: CUDA - GFLOPS Caso Quadrato

4.3.2 Caso Rettangolare

Le prestazioni ottenute con CUDA nel caso quadrato sono mostrate in figura 18.

Il caso rettangolare mostra risultati molto simili a quelli del caso rettangolare: non c'è infatti nulla di particolare da segnalare rispetto al caso precedente.

4.3.3 Errore Relativo

In tutti i casi considerati, l'errore relativo prodotto dal nucleo di calcolo è stato nullo. Notiamo inoltre che non ci sono casi in cui un elemento della matrice C

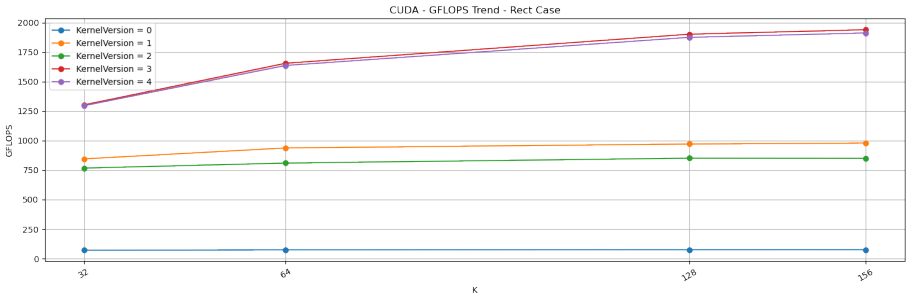

Figure 18: CUDA - GFLOPS Caso Rettangolare

è calcolato facendo operazioni, come la somma, in ordine diverso, quindi anche i risultati prodotti da esecuzioni diverse a parità di input daranno lo stesso risultato.