Università degli Studi di Napoli
"Parthenope"
Dipartimento di Scienze e Tecnologie
Corso di Laurea Magistrale in
Informatica Applicata

Corso di High Performance Computing

Accelerazione del prodotto matriciale con l'algoritmo di Strassen

Dario Musella
0120000240

Indice

1 Analisi e definizione del problema **3**
1.1 Analisi dell'algoritmo 4
1.2 Algoritmo di Strassen 5

2 Descrizione dell'algoritmo in ambiente GPU-CUDA **8**
2.1 Studio dell'algoritmo in ambiente CPU sequenziale 9
2.1.1 Rifiniture e ulteriori ottimizzazioni 10
2.2 Implementazione in ambiente GPU-CUDA 12
2.2.1 Gestione della memoria 13
2.2.2 Strategia di parallelizzazione 14

3 Configurazione e risoluzione istanze **15**

4 Routine implementate **16**

5 Analisi delle prestazioni **18**
5.0.1 Algoritmo ibrido di Strassen con algoritmo naive 18
5.0.2 Algoritmo ibrido di Strassen con procedura di libreria cuBLAS 19
5.1 Algoritmo di Strassen su CPU sequenziale 22
5.2 Algoritmo ibrido di Strassen in ambiente GPU-CUDA 25
5.3 Algoritmo naive del prodotto matriciale con uso della shared memory 27
5.4 Interfaccia per l'algoritmo di Strassen in ambiente GPU-CUDA 28

Capitolo 1

Analisi e definizione del problema

In questa relazione viene affrontata la problematica dell'algoritmo del prodotto tra due matrici.

Sia data una matrice $\mathbf{A}$ di dimensione $m \times n$ e sia data una matrice $\mathbf{B}$ di dimensione $n \times p$, si ricava la matrice $\mathbf{C} = \mathbf{A} \times \mathbf{B}$ di dimensione $m \times p$ mediante il prodotto tra le n righe della matrice $\mathbf{A}$ e le n colonne della matrice $\mathbf{B}$

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

dove il generico elemento c_{ij} è dato dalla seguente somma [1]

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \qquad \forall i \in \{1, 2, \ldots, m\},\ \forall j \in \{1, 2, \ldots, p\}$$

Questo algoritmo è implementabile mediante un codice del tipo

Algoritmo 1 Prodotto matriciale naive

```
for i ← 1 to m do
    for j ← 1 to p do
        c_ij ← 0
        for k ← 1 to n do
            c_ij ← c_ij + a_ik · b_kj
        end for
    end for
end for
```

1.1 Analisi dell'algoritmo

Considerando l'algoritmo sopra citato, con due matrici di dimensioni rispettivamente $m \times n$ e $n \times p$, si evince che nel processo vengono eseguiti un numero di prodotti pari a $m \times n \times p$, determinando una complessità di tempo dell'algoritmo pari a $O(mnp)$. Ponendo $m = p$ e $m = n$, si può riscrivere la definizione di complessità temporale dell'algoritmo come $O(n^3)$, giustificata anche dalla presenza di tre blocchi iterativi innestati.

L'algoritmo, per quanto intuitivo possa risultare dall'implementazione sopra illustrata, ha una crescita di tempo cubica all'aumentare delle dimensioni di input.

Per rafforzare tale affermazione, si può introdurre una differente implementazione dell'algoritmo in questione: una strutturazione in un approccio del tipo *divide et impera*.

Siano date due matrici quadrate $\mathbf{A}$ e $\mathbf{B}$ dell'ordine di 2^n, è possibile decomporre le due matrici in quattro sottomatrici dell'ordine di 2^{n-1}

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{pmatrix}, \ \mathbf{B} = \begin{pmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{pmatrix}$$

ed è possibile costruire la matrice $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$

$$\mathbf{C} = \begin{pmatrix} \mathbf{C}_{11} & \mathbf{C}_{12} \\ \mathbf{C}_{21} & \mathbf{C}_{22} \end{pmatrix}$$

dove le quattro sottomatrici possono essere ottenute con delle somme di prodotti

$$\begin{pmatrix} \mathbf{C}_{11} & \mathbf{C}_{12} \\ \mathbf{C}_{21} & \mathbf{C}_{22} \end{pmatrix} = \begin{pmatrix} \mathbf{A}_{11} \cdot \mathbf{B}_{11} + \mathbf{A}_{12} \cdot \mathbf{B}_{21} & \mathbf{A}_{1,1} \cdot \mathbf{B}_{1,2} + \mathbf{A}_{1,2} \cdot \mathbf{B}_{2,2} \\ \mathbf{A}_{2,1} \cdot \mathbf{B}_{1,1} + \mathbf{A}_{2,2} \cdot \mathbf{B}_{2,1} & \mathbf{A}_{2,1} \cdot \mathbf{B}_{1,2} + \mathbf{A}_{2,2} \cdot \mathbf{B}_{2,2} \end{pmatrix}$$

Complessivamente si presentano 8 moltiplicazioni da calcolare su matrici di dimensione 2^{n-1}, ovvero la metà della dimensione originaria. I fattori ottenuti sono successivamente membri di un'addizione, che nel caso di due matrici vengono eseguite in un tempo $O(n^2)$.

Una possibile implementazione dell'algoritmo appena descritto può essere la seguente. Si ipotizzano matrici quadrate per favorire una maggiore leggibilità.

Algoritmo 2 Prodotto matriciale naive divide et impera

function MATMUL($\mathbf{A}, \mathbf{B}$)
  $\mathbf{C} \leftarrow \emptyset$ ▷ $\mathbf{C}$ è una matrice $n \times n$
  if $n = 1$ **then**
    $c_{11} \leftarrow a_{11} \cdot b_{11}$
  else ▷ partiziona matrici $\mathbf{A}$ e $\mathbf{B}$ in quattro porzioni $\frac{n}{2} \times \frac{n}{2}$
    $\mathbf{C}_{11} \leftarrow$ MATMUL($\mathbf{A}_{11}$, $\mathbf{B}_{11}$) + MATMUL($\mathbf{A}_{12}$, $\mathbf{B}_{21}$)
    $\mathbf{C}_{12} \leftarrow$ MATMUL($\mathbf{A}_{11}$, $\mathbf{B}_{12}$) + MATMUL($\mathbf{A}_{12}$, $\mathbf{B}_{22}$)
    $\mathbf{C}_{21} \leftarrow$ MATMUL($\mathbf{A}_{21}$, $\mathbf{B}_{11}$) + MATMUL($\mathbf{A}_{22}$, $\mathbf{B}_{21}$)
    $\mathbf{C}_{22} \leftarrow$ MATMUL($\mathbf{A}_{21}$, $\mathbf{B}_{12}$) + MATMUL($\mathbf{A}_{22}$, $\mathbf{B}_{22}$)
  end if
  return C
end function

Il caso base dell'algoritmo consiste in un prodotto numerico, in seguito alla decomposizione ricorsiva delle matrici di dimensioni superiori. Poichè l'algoritmo è di tipo divide et impera, si ha la relazione di ricorrenza [2]

$$T(n) = 8T\left(\frac{n}{2}\right) + O(n^2)$$

e per il primo caso del metodo dell'esperto si ha

$$T(n) = O(n^{\log_2 8}) = O(n^3)$$

Questo conclude che l'ordine di complessità dell'algoritmo del prodotto tra due matrici è *logaritmicamente* proporzionale al numero di prodotti computati per ottenere le matrici decomposte $\mathbf{C}_{ij}$.

Sebbene l'implementazione con i tre costrutti iterativi si presti bene alla parallelizzazione in ambienti paralleli, come MIMD-SM per le CPU e SIMT per ambienti GPGPU, questa relazione vuole presentare una ulteriore miglioria all'algoritmo naive per il prodotto di due matrici.

L'accelerazione è possibile ricorrendo ad un metodo alternativo di calcolo, ovvero l'algoritmo di Strassen.

1.2 Algoritmo di Strassen

L'algoritmo proposto da V. Strassen si contrappone alla logica dell'algoritmo precedentemente citato, proponendo una implementazione controintuitiva, ma che presenta un ordine di complessità inferiore. [3]

L'algoritmo in questione è del tipo *divide et impera* e prevede il calcolo di sette matrici ausiliarie

$$\begin{aligned}
\mathbf{M}_1 &= (\mathbf{A}_{11} + \mathbf{A}_{22})(\mathbf{B}_{11} + \mathbf{B}_{22}) \\
\mathbf{M}_2 &= (\mathbf{A}_{21} + \mathbf{A}_{22})\mathbf{B}_{11} \\
\mathbf{M}_3 &= \mathbf{A}_{11}(\mathbf{B}_{12} - \mathbf{B}_{22}) \\
\mathbf{M}_4 &= \mathbf{A}_{22}(\mathbf{B}_{21} - \mathbf{B}_{11}) \\
\mathbf{M}_5 &= (\mathbf{A}_{11} + \mathbf{A}_{12})\mathbf{B}_{22} \\
\mathbf{M}_6 &= (\mathbf{A}_{21} - \mathbf{A}_{11})(\mathbf{B}_{11} + \mathbf{B}_{12}) \\
\mathbf{M}_7 &= (\mathbf{A}_{12} - \mathbf{A}_{22})(\mathbf{B}_{21} + \mathbf{B}_{22})
\end{aligned}$$

riducendo così il numero di prodotti da otto a sette e, di conseguenza, la complessità di tempo. Si ricombina quindi nella matrice risultato

$$\begin{pmatrix} \mathbf{C}_{11} & \mathbf{C}_{12} \\ \mathbf{C}_{21} & \mathbf{C}_{22} \end{pmatrix} = \begin{pmatrix} \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 & \mathbf{M}_3 + \mathbf{M}_5 \\ \mathbf{M}_2 + \mathbf{M}_4 & \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6 \end{pmatrix}$$

Questo processo di decomposizione viene ripetuto fintantoché le sottomatrici non hanno dimensioni 1×1, diventando così dei singoli numeri

$$\begin{aligned}
M_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\
M_2 &= (a_{21} + a_{22})b_{11} \\
M_3 &= a_{11}(b_{12} - b_{22}) \\
M_4 &= a_{22}(b_{21} - b_{11}) \\
M_5 &= (a_{11} + a_{12})b_{22} \\
M_6 &= (a_{21} - a_{11})(b_{11} + b_{12}) \\
M_7 &= (a_{12} - a_{22})(b_{21} + b_{22})
\end{aligned}$$

dove in questo caso banale le due matrici di input sono

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \qquad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

Si può dire quindi che il caso banale è dato da due matrici $\mathbf{A}$ e $\mathbf{B}$ di ordine $n = 1$, mentre con $n > 1$ si ha un caso non banale (ricorsivo), per cui avviene la decomposizione.

Di seguito viene illustrato l'algoritmo di Strassen in pseudocodice

Algoritmo 3 Algoritmo di Strassen

function STRASSEN($\mathbf{A}, \mathbf{B}$)
  $\mathbf{C} \leftarrow \emptyset$ ▷ $\mathbf{C}$ è una matrice $n \times n$
  if $n = 1$ **then**
    $c_{11} \leftarrow a_{11} \cdot b_{11}$
  else ▷ partiziona matrici $\mathbf{A}$ e $\mathbf{B}$ in quattro porzioni $\frac{n}{2} \times \frac{n}{2}$
    $\mathbf{M}_1 \leftarrow$ STRASSEN$(\mathbf{A}_{11} + \mathbf{A}_{22}, \mathbf{B}_{11} + \mathbf{B}_{22})$
    $\mathbf{M}_2 \leftarrow$ STRASSEN$(\mathbf{A}_{21} + \mathbf{A}_{22}, \mathbf{B}_{11})$
    $\mathbf{M}_3 \leftarrow$ STRASSEN$(\mathbf{A}_{11}, \mathbf{B}_{12} - \mathbf{B}_{22})$
    $\mathbf{M}_4 \leftarrow$ STRASSEN$(\mathbf{A}_{22}, \mathbf{B}_{21} - \mathbf{B}_{11})$
    $\mathbf{M}_5 \leftarrow$ STRASSEN$(\mathbf{A}_{11} + \mathbf{A}_{12}, \mathbf{B}_{22})$
    $\mathbf{M}_6 \leftarrow$ STRASSEN$(\mathbf{A}_{21} - \mathbf{A}_{11}, \mathbf{B}_{11} + \mathbf{B}_{12})$
    $\mathbf{M}_7 \leftarrow$ STRASSEN$(\mathbf{A}_{12} - \mathbf{A}_{21}, \mathbf{B}_{21} + \mathbf{B}_{22})$
    $\mathbf{C}_{11} \leftarrow \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7$
    $\mathbf{C}_{12} \leftarrow \mathbf{M}_3 + \mathbf{M}_5$
    $\mathbf{C}_{21} \leftarrow \mathbf{M}_2 + \mathbf{M}_4$
    $\mathbf{C}_{22} \leftarrow \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6$
  end if
  return C
end function

Questo algoritmo calcola il prodotto di due matrici ricorrendo a 7 chiamate ricorsive in sottomatrici di dimensioni $\frac{n}{2} \times \frac{n}{2}$, con l'aggiunta di 18 addizioni (e sottrazioni). La complessità temporale dell'algoritmo del prodotto matriciale di Strassen può essere ricavato studiando la relazione di ricorrenza [4]

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$$

per il teorema dell'esperto si ottiene che la complessità di tempo dell'algoritmo è

$$T(n) = O(n^{\log_2 7}) \approx O(n^{2,8})$$

In conclusione, i due algoritmi illustrati presentano grosse differenze

	Algoritmo naive	Algoritmo di Strassen
Complessità	$O(n^3)$	$O(n^{2,8})$
Approccio	Iterativo (e divide et impera)	Divide et impera
Applicazione	Qualsiasi matrice	Matrici quadrate

Capitolo 2

Descrizione dell'algoritmo in ambiente GPU-CUDA

Benché l'algoritmo di Strassen goda di una complessità computazionale di tempo inferiore ad un ordine cubico, in un contesto pratico (GPGPU in questo caso) risulta necessario porre l'accento su altre considerazioni. La riduzione del numero di moltiplicazioni da effettuare è, difatti, possibile grazie a delle matrici ausiliarie che gravano in modo considerevole sul quantitativo di spazio richiesto, dal momento che le sette matrici $\mathbf{M}_1, ..., \mathbf{M}_7$ hanno le stesse dimensioni delle matrici di input.

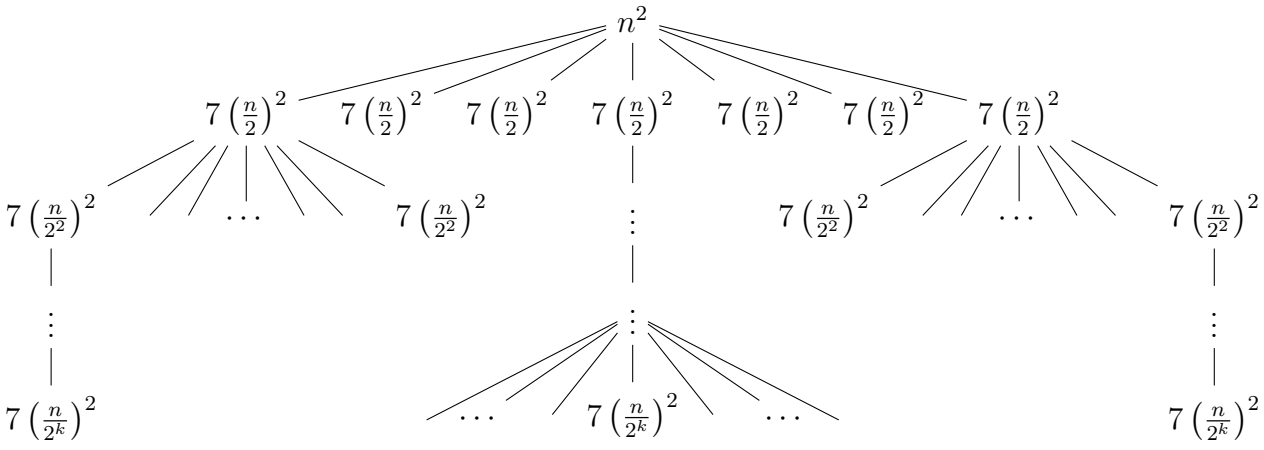

Figura 2.1: Albero di ricorsione dell'algoritmo di Strassen.

Questa peculiarità penalizza notevolmente l'algoritmo in questione, soprattutto se si considera che l'algoritmo standard può essere eseguito sul posto.

Risulta, pertanto, necessario ricorrere ad empirismi al fine di poter valutare l'effettivo guadagno prestazionale ricorrendo all'algoritmo di Strassen.

Gli algoritmi che verrano successivamente descritti saranno riportati in appendice.

2.1 Studio dell'algoritmo in ambiente CPU sequenziale

In questa sezione viene effettuata una prima trattazione dell'algoritmo di Strassen, per poi confrontarlo con l'algoritmo naive per il prodotto tra due matrici.

Innanzitutto salta all'occhio la necessità di creare e distruggere dinamicamente le matrici temporanee nei sottoproblemi generati mediante chiamate a `malloc()`, `calloc()` e conseguenti `free()` che aumentano l'overhead generale dell'algoritmo, oltre al considerevole numero dei sottoproblemi stessi generati (come visto nella figura 2.1).

L'algoritmo implementato in linguaggio C non presenta grosse differenze rispetto allo pseudocodice numero 3 precedentemente trattato: il costrutto iterativo gestisce i casi base e ricorsivo del problema. Nel caso base si ha una trattazione con matrici di dimensione 2×2 che, con una opportuna indicizzazione permette un calcolo puntuale degli elementi.

Sebbene la complessità temporale è inferiore all'ordine cubico, l'algoritmo richiede maggiore spazio per generare i sottoproblemi nello stack e allocare le matrici nell'heap. Di seguito sono riportati i tempi di esecuzione degli algoritmi del prodotto matriciale, ovvero l'algoritmo di Strassen e l'algoritmo naive.

Ordine	**Naive**	**Strassen**
256	0,1006855	0,2553297
512	0,83907	1,389018667
1024	9,215430667	8,239617167
2048	182,7594968	50,9861302
4096	1680,595993	341,1824423

Tabella 2.1: Tempi di esecuzione in ambiente CPU sequenziale, in secondi.

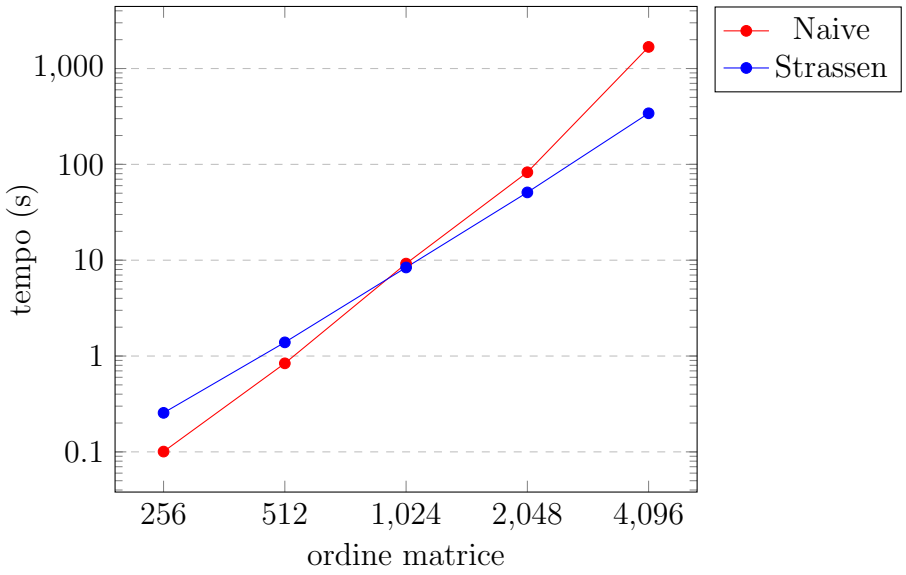

Figura 2.2: Grafico dei tempi in ambiente CPU sequenziale.

Dai risultati si evince che l'algoritmo standard per il prodotto matriciale offre delle prestazioni migliori fintantoché le dimensioni degli input non superano la grandezza di 512×512, mentre l'algoritmo di Strassen vince il confronto se le matrici hanno più di 512×512 elementi.

La curva dell'algoritmo naive parte da un punto più basso, ma la complessità di tempo cubica ne causa una crescita più rapida rispetto all'algoritmo di Strassen, che seppur avendo dei tempi maggiori per via di un overhead nettamente superiore (che è visibile su matrici più piccole) ha una crescita più lenta. Questo rende l'algoritmo di Strassen una scelta migliore tanto quanto più grandi sono le dimensioni delle matrici.

2.1.1 Rifiniture e ulteriori ottimizzazioni

A valle della ricerca empirica appena descritta, si può ora lavorare sull'algoritmo, focalizzando l'attenzione su dettagli che possano ottimizzarlo.

In sintesi, la complessità temporale inferiore dell'algoritmo di Strassen gode di benefici a partire da un determinato punto n_0, che nei test svolti in ambiente CPU sequenziale, si potrebbe porre $n_0 = 1024$. Se si considera n_0 l'ordine delle matrici per cui l'algoritmo di Strassen gode di migliori performance, si può stabilire un criterio dove si determina quale algoritmo risulta maggiormente efficace.

La proposta di un algoritmo ibrido pone l'accento sul controllo preliminare delle dimensioni delle matrici, in particolare se l'ordine $n \geqslant n_0$ allora si

godono i benefici della decomposizione di Strassen, in caso contrario, invece, non conviene "sacrificare" spazio computazionale, in quanto l'overhead incide in modo non trascurabile sulle prestazioni.

Algoritmo 4 Algoritmo ibrido del prodotto matriciale

function STRASSEN($\mathbf{A}, \mathbf{B}$)
  $\mathbf{C} \leftarrow \emptyset$ ▷ $\mathbf{C}$ è una matrice $n \times n$
  if $n < n_0$ **then**
    $\mathbf{C} \leftarrow$ MATMUL($\mathbf{A}$, $\mathbf{B}$)
  else ▷ partiziona matrici $\mathbf{A}$ e $\mathbf{B}$ in quattro porzioni $\frac{n}{2} \times \frac{n}{2}$
    $\mathbf{M}_1 \leftarrow$ STRASSEN($\mathbf{A}_{11} + \mathbf{A}_{22}$, $\mathbf{B}_{11} + \mathbf{B}_{22}$)
    $\mathbf{M}_2 \leftarrow$ STRASSEN($\mathbf{A}_{21} + \mathbf{A}_{22}$, $\mathbf{B}_{11}$)
    $\mathbf{M}_3 \leftarrow$ STRASSEN($\mathbf{A}_{11}$, $\mathbf{B}_{12} - \mathbf{B}_{22}$)
    $\mathbf{M}_4 \leftarrow$ STRASSEN($\mathbf{A}_{22}$, $\mathbf{B}_{21} - \mathbf{B}_{11}$)
    $\mathbf{M}_5 \leftarrow$ STRASSEN($\mathbf{A}_{11} + \mathbf{A}_{12}$, $\mathbf{B}_{22}$)
    $\mathbf{M}_6 \leftarrow$ STRASSEN($\mathbf{A}_{21} - \mathbf{A}_{11}$, $\mathbf{B}_{11} + \mathbf{B}_{12}$)
    $\mathbf{M}_7 \leftarrow$ STRASSEN($\mathbf{A}_{12} - \mathbf{A}_{21}$, $\mathbf{B}_{21} + \mathbf{B}_{22}$)
    $\mathbf{C}_{11} \leftarrow \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7$
    $\mathbf{C}_{12} \leftarrow \mathbf{M}_3 + \mathbf{M}_5$
    $\mathbf{C}_{21} \leftarrow \mathbf{M}_2 + \mathbf{M}_4$
    $\mathbf{C}_{22} \leftarrow \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6$
  end if
  return C
end function

La bozza di codice appena illustrata sintetizza l'idea di un algoritmo ibrido che stabilisce quale algoritmo utilizzare per risolvere quel determinato sottoproblema: si ha una soglia n_0 per cui non conviene più creare ulteriori sottoproblemi, bensì conviene risolverlo direttamente sul posto. Si ha un problema iniziale dove l'ordine delle matrici è $n \geqslant n_0$, e decomponendo si arriva ad un punto dove n decresce fino a che $n < n_0$. In questo modo si possono combinare i benefici del meccanismo divide et impera proposto da Strassen per la risoluzione di grossi problemi, e i benefici dell'algoritmo naive per la risoluzione di problemi piccoli, in quanto presenta poco overhead.

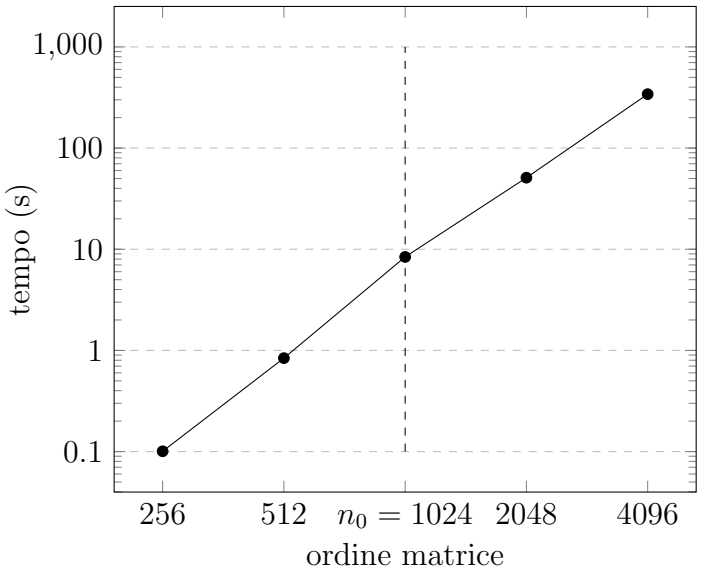

Figura 2.3: Grafico dei tempi dell'algoritmo ibrido in ambiente CPU sequenziale. Il punto n_0 indica la dimensione per cui avviene la risoluzione mediante l'algoritmo di Strassen.

In questo modo si è ottenuto un algoritmo che risulta ottimale per ogni input. Le successive trattazioni e discussioni verteranno esclusivamente sull'implementazione ottimale appena illustrata, pertanto con il termine generico algoritmo si intenderà, d'ora in poi, unicamente questa versione.

2.2 Implementazione in ambiente GPU-CUDA

L'operazione di porting in ambiente GPGPU richiede di prestare attenzione alla strategia di parallelizzazione dell'algoritmo, nonché della struttura dello stesso.

L'algoritmo risolve il problema con l'approccio divide et impera, il che lo rende a primo impatto poco parallelizzabile. L'architettura GPU è intrinsecamente portata a risolvere agevolmente problemi con una struttura iterativa (soprattutto sul posto), presenta uno stack nettamente più piccolo rispetto ad un'architettura CPU e, di conseguenza, non risulta essere la scelta ottimale per algoritmi ricorsivi.

Al fine di ottenere un algoritmo in ambiente GPU-CUDA altrettanto performante è necessario risolvere innanzitutto i problemi derivanti dalla struttura ricorsiva dell'algoritmo, quindi successivamente discutere della strategia di parallelizzazione.

2.2.1 Gestione della memoria

L'algoritmo proposto in ambiente CPU sequenziale alloca e dealloca dinamicamente le matrici temporanee mediante chiamate a `malloc()` e `free()`. In ambiente CUDA queste chiamate rallentano in modo considerevole il programma, pertanto si può ragionare a priori sulle matrici temporanee richieste per la computazione.

Considerando l'albero di ricorsione (figura 2.1) si può osservare che ad ogni livello le dimensioni delle matrici si dimezzano, ottenendo quattro quadranti della stessa, dove ogni sottoproblema risolve uno dei quattro quadranti. Dal momento che la somma delle dimensioni dei quattro quadranti corrisponde alle dimensioni dell'input originario, si può dedurre che ad ogni livello dell'albero le dimensioni delle matrici temporanee $\mathbf{M}_1, \ldots, \mathbf{M}_7$ diminuiscono di dimensione solo localmente.

Si può, quindi, gestire la memoria sulla falsariga della programmazione dinamica [5], allocando un numero di matrici pari (o superiore) al numero di livelli dell'albero di ricorsione, ovvero fare in modo che `M1, ..., M7` siano sette array di puntatori a matrici, dove ognuno è associato ad un determinato livello dell'albero.

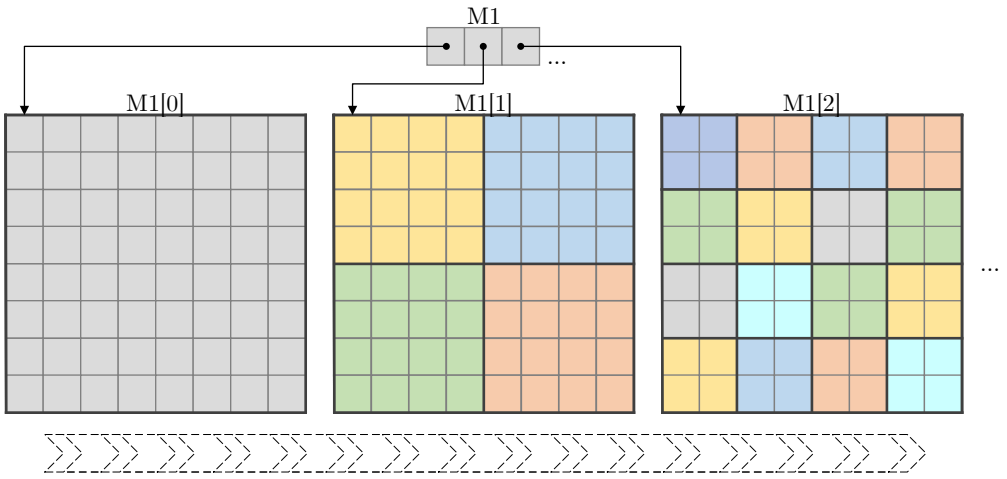

Figura 2.4: Esempio di allocazione delle matrici $\mathbf{M}_1$ temporanee per ogni livello dell'albero. La prima matrice $\mathbf{M}_1$ (di indice zero) corrisponde al primo livello dell'albero, dove la dimensione è $n \times n$, la seconda matrice $\mathbf{M}_1$ corrisponde al secondo livello dell'albero, partizionato in quattro quadranti di dimensione $\frac{n}{2} \times \frac{n}{2}$, dove ognuno è associato a un sottoproblema, e così via.

2.2.2 Strategia di parallelizzazione

Siccome l'algoritmo ha una struttura intrinsecamente ricorsiva, è necessario analizzare passo per passo gli step che lo compongono. La parallelizzazione degli step è stata gestita mediante la libreria cuBLAS.

Considerando il blocco **else** dell'algoritmo 4 si ha che le sette chiamate ricorsive sono precedute da addizioni e sottrazioni matriciali, oltre ad altre operazioni dello stesso tipo per la collezione dei risultati nelle sottomatrici $\mathbf{C}_{11}, \mathbf{C}_{12}, \mathbf{C}_{21}, \mathbf{C}_{22}$. Queste operazioni possono essere opportunamente parallelizzate.

Considerando, inoltre, il blocco **if** del medesimo algoritmo si osserva che la risoluzione delle sottomatrici di ordine $n < n_0$, ovvero, il caso base, è delegata all'algoritmo standard del prodotto matriciale, che è iterativo e fortemente parallelizzabile.

Per quanto concerne il caso base dell'algoritmo è possibile ricorrere ad un qualsiasi algoritmo fortemente parallelizzabile per poter accelerare ulteriormente il programma complessivo. In questo elaborato vengono proposte due alternative: un kernel CUDA implementato a mano e la procedura fornita dalla libreria cuBLAS.

Il kernel richiede la configurazione a monte del numero di blocchi da destinare e il numero di thread per blocco. Siccome il problema prevede in input matrici quadrate e di ordine 2^n è consigliato impostare tali valori in modo che siano potenze di due, così da evitare sbilanciamenti sul carico di lavoro da parte dei thread. Questo kernel offre una routine parallelizzata e ottimizzata dal punto di vista architetturale, in quanto ricorre alla memoria condivisa per la suddivisione degli input per la computazione.

Un'altra idea è, come già anticipato, ricorrere alla libreria cuBLAS, in quanto fornisce la funzione `cublasSgemm()`[1] fortemente ottimizzata per l'ambiente GPU-CUDA. Benché sia possibile risolvere il problema direttamente con la routine di libreria, l'incapsulamento all'interno della decomposizione di Strassen ha dato luogo a scenari interessanti che verranno successivamente discussi.

[1] Più genericamente il parco funzioni `cublas<t>gemm()`, ma nei test si è optato per il tipo di dato float.

Capitolo 3

Configurazione e risoluzione istanze

Il programma presenta un'interfaccia testuale piuttosto semplice. Una volta compilato il programma con

```
nvcc main.cu -lcublas -o main
```

si ottiene l'eseguibile che da riga di comando prevede in input:

- ordine n delle matrici quadrate
- soglia n_0 per caso base
- flag per impostare l'algoritmo per il caso base

  0 ricorre al kernel implementato ad hoc

  1 richiama la routine `cublasSgemm()` [6]

Una volta creata l'istanza, dove vengono generate le matrici $\mathbf{A}$ e $\mathbf{B}$ di dimensioni $n \times n$ con valori reali[1], il programma calcolerà la matrice risultato $\mathbf{C}$ in primo tempo con l'algoritmo naive, quindi la ricalcola con l'algoritmo accelerato. Nelle fasi di calcolo dei due rispettivi algoritmi vengono calcolati i tempi in millisecondi impiegati per il calcolo effettivo. Al termine della fase di calcolo, il programma verifica se la matrice risultato ricavata con i due differenti algoritmi coincidono.

In output il programma restituisce la matrice $\mathbf{C} = \mathbf{A} \times \mathbf{B}$ e mostra i risultati su schermo[2]

[1] Poiché `rand()` restituisce valori interi, si tratta di un tipo reale solo dal punto di vista della gestione della memoria. In particolare cuBLAS richiede necessariamente un tipo di dato a virgola mobile.

[2] Se l'input è sufficientemente piccolo. In fase di test l'output a schermo della matrice $\mathbf{C}$ è servito per verificarne la correttezza.

Capitolo 4

Routine implementate

Il progetto è composto principalmente da due procedure per il calcolo del prodotto matriciale.

In particolare è stata implementata una procedura `matmul()` dell'algoritmo naive per il prodotto matriciale, che fa uso della memoria condivisa. L'intestazione della procedura prevede come parametri le tre matrici `A`, `B` e `C`, oltre all'ordine n. L'algoritmo prevede innanzitutto una copia degli elementi dalla memoria globale alla memoria condivisa per ogni blocco, per poi sincronizzare tutti i thread in barriera. Terminata la fase di copia per tutti i thread, questi calcoleranno il prodotto matriciale facendo sì che ogni thread in parallelo esegua dei prodotti riga per colonna. Una ulteriore barriera previene race condition nella fase finale di collezione dei risultati.

La procedura `strassen()` risolve il problema col meccanismo divide et impera. In input richiede le tre matrici `A`, `B` e `C`, l'handle creato da cuBLAS, l'ordine n, e in particolare la soglia n_0 e un valore d che indica la profondità nell'albero di ricorsione, usato come indice dei puntatori alle matrici temporanee $\mathbf{M}_1, \ldots, \mathbf{M}_7$. La routine viene eseguita dal `main()` del programma con l'impostazione $d = 0$. In primo luogo verifica se l'istanza da risolvere rientra nel caso base o nel caso ricorsivo, quindi effettua l'opportuna selezione.

Nel blocco `if` della selezione viene affrontato il caso banale del problema. In base a come è stato settato il flag da shell avviene la risoluzione mediante il kernel `matmul()` oppure con la routine di libreria `cublasSgemm()`. Nel caso in cui il problema venga risolto con `matmul()`, vengono allocate un numero di griglie bidimensionali di thread con l'espressione

$$\left(\left\lfloor \frac{n+k-1}{k} \right\rfloor, \left\lfloor \frac{n+k-1}{k} \right\rfloor \right)$$

dove k è una costante fissata al preprocessore, e deve essere tale che sia potenza di 2. L'espressione garantisce che con k potenza di 2 si ha un numero

di griglie sempre pari ad una potenza di 2. Ogni griglia ha un numero di thread pari a (k, k).

Nel blocco `else` viene risolto il caso non banale del problema, andando innanzitutto a decomporre la matrice in quattro quadranti, quindi vengono ricavate le matrici temporanee $\mathbf{M}_1, \ldots, \mathbf{M}_7$ mediante addizioni e sottrazioni matriciali. Queste operazioni sono svolte in parallelo mediante la routine di cuBLAS `cublasSgeam()` [7] che calcola l'espressione $\mathbf{C} = \alpha\mathbf{A} + \beta\mathbf{B}$, dove α e β sono degli scalari fissati a 0, 1 o -1 a seconda dell'operazione da svolgere. Queste operazioni sono "incapsulate" attraverso procedure che astraggono alcuni dettagli di cuBLAS irrilevanti ai fini del progetto, oltre al favorirne una maggiore lettura.

La procedura `addmat()` effettua una addizione matriciale impostando $\alpha = 1$ e $\beta = 1$, mentre la sottrazione matriciale viene eseguita da `submat()` che richiama la stessa routine di cuBLAS settando $\alpha = 1$ e $\beta = -1$. La procedura `matcpy()` sfrutta l'operazione fatta da `cublasSgeam()` per copiare il quadrante di riferimento dalla matrice di input alla propria matrice temporanea di dimensione $\frac{n}{2} \times \frac{n}{2}$, impostando opportunamente la *leading dimension* per quella data operazione.

Una volta effettuate queste operazioni, le matrici temporanee vengono risolte mediante la chiamata ricorisva `strassen()`. In particolare le chiamate ricorsive prevedono l'incremento della profondità d, in quanto l'algoritmo sta "percorrendo" l'albero di ricorsione verticalmente.

Le matrici di input su GPU `A_dev` e `B_dev` sono costruiti nel medesimo modo delle matrici temporanee (figura 2.4), in quanto ad ogni livello dell'albero differiscono le istanze dei sottoproblemi.

L'interfaccia del programma si occupa di allocare e deallocare la memoria sia sull'host che sul device, oltre ad inizializzare l'handle di cuBLAS. L'allocazione è anticipata da un controllo sull'ordine delle matrici fornito in input: se il valore fornito non è una potenza di 2, allora viene sostituito con la potenza di 2 prossima più vicina.

Capitolo 5

Analisi delle prestazioni

Di seguito sono riportati i tempi di esecuzione in millisecondi dell'algoritmo proposto in varie configurazioni e istanze.

Le tabelle dei tempi sono seguite da grafici comparativi.

5.0.1 Algoritmo ibrido di Strassen con algoritmo naive

Sotto sono riportati i tempi di esecuzione dell'algoritmo accelerato ibrido, con risoluzione del caso base con l'algoritmo naive che ricorre alla memoria condivisa. Questi tempi sono confrontati con istanze risolte esclusivamente con l'algoritmo naive.

Il valore soglia n_0 è stato impostato a 512 per tutte le dimensioni di seguito riportate.

Dimensioni	**Naive**	**Strassen**
1024	14,45792	13,77344
2048	114,66022	96,90343
4096	908,87927	686,35437
8192	7190,91992	4869,87354

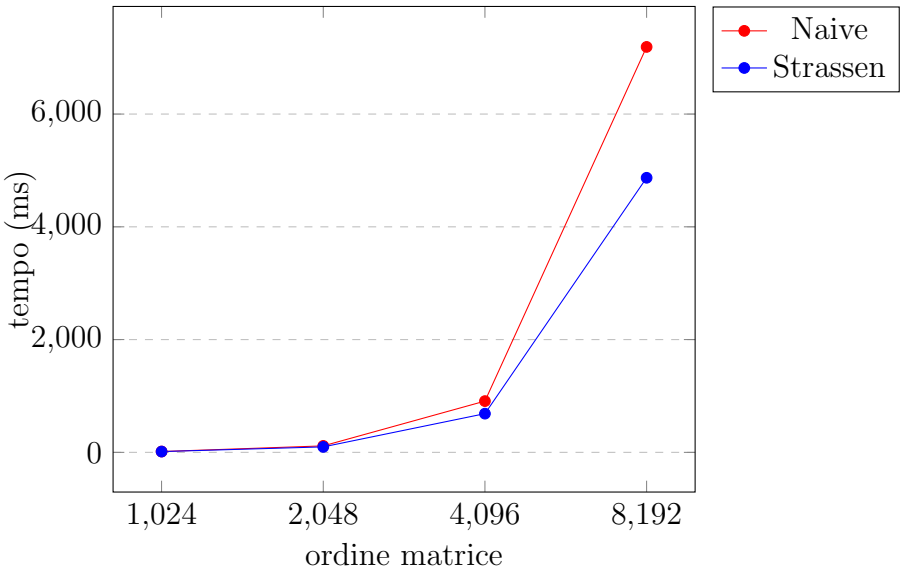

Figura 5.1: Grafico dei tempi dell'algoritmo ibrido di Strassen con l'algoritmo naive.

5.0.2 Algoritmo ibrido di Strassen con procedura di libreria cuBLAS

In questo caso vengono riportati i tempi di esecuzione dell'algoritmo accelerato ibrido, con risoluzione del caso base con la routine della libreria cuBLAS `cublasSgemm()`. Questi tempi sono confrontati con istanze risolte esclusivamente con la procedura presente in cuBLAS.

Il valore soglia n_0 è stato impostato a 1024 per tutte le dimensioni di seguito riportate.

Dimensioni	**cuBLAS**	**Strassen**
2048	16,31264	16,99622
4096	128,79140	127,44643
8192	1023,52594	928,07965

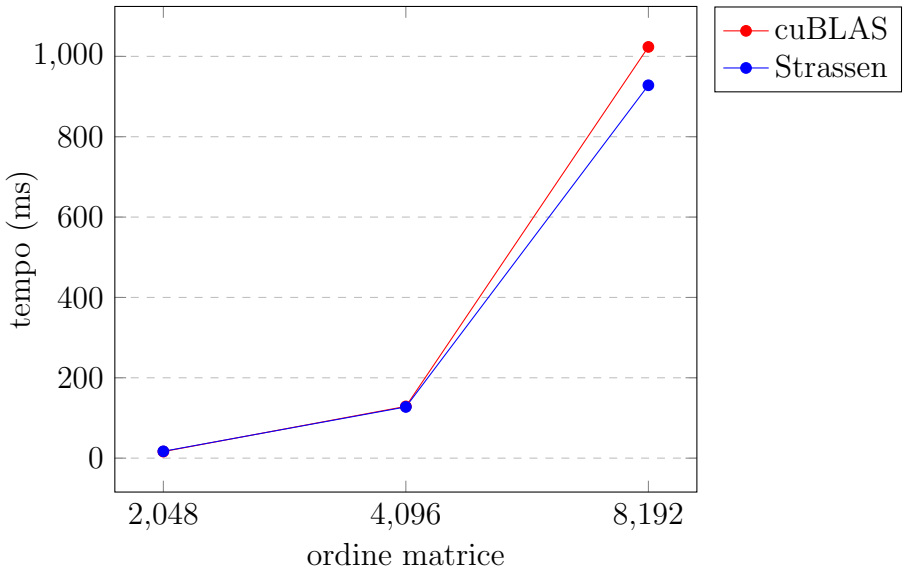

Figura 5.2: Grafico dei tempi dell'algoritmo ibrido di Strassen con la routine di cuBLAS.

In conclusione, si evince che esiste un valore soglia n_0 tale che per ogni $n > n_0$, l'algoritmo ibrido di Strassen risolve il problema in un tempo inferiore agli altri algoritmi usati per il confronto.

Bibliografia

[1] *Matrix Multiplication*. indirizzo: https://mathworld.wolfram.com/MatrixMultiplication.html (visitato il 21/01/2022).

[2] T. Cormen, C. Leiserson, R. Rivest e C. Stein, «Introduzione agli algoritmi e strutture dati,» in 2010, cap. 4. Ricorrenze, pp. 61–63.

[3] V. Strassen, «Gaussian Elimination is not Optimal,» *Numerische Mathematik*, pp. 354–356, 1969.

[4] G. Miller e D. Witmer, *Lecture 1: Introduction and Strassen's Algorithm*, online, 2016. indirizzo: http://www.cs.cmu.edu/afs/cs/academic/class/15750-s17/ScribeNotes/lecture1.pdf.

[5] A. Monti e I. Finocchi, «Tecnica Programmazione dinamica,» 2006.

[6] *cuBLAS Toolkit Documentation, `cublassgemm()`*. indirizzo: https://docs.nvidia.com/cuda/cublas/index.html#cublas-lt-t-gt-gemm (visitato il 21/01/2022).

[7] *cuBLAS Toolkit Documentation, `cublassgeam()`*. indirizzo: https://docs.nvidia.com/cuda/cublas/index.html#cublas-lt-t-gt-geam (visitato il 21/01/2022).

Appendice

Di seguito vengono riportati i codici sorgenti del progetto sviluppato.

5.1 Algoritmo di Strassen su CPU sequenziale

```
1  void strassen(int *C, const int *A, const int *B, const int n) {
2      if (n == 2) { /* caso base */
3          int M1 = (A[0] + A[n + 1]) * (B[0] + B[n + 1]);
4          int M2 = (A[n] + A[n + 1]) * B[0];
5          int M3 = A[0] * (B[1] - B[n + 1]);
6          int M4 = A[n + 1] * (B[n] - B[0]);
7          int M5 = (A[0] + A[1]) * B[n + 1];
8          int M6 = (A[n] - A[0]) * (B[0] + B[1]);
9          int M7 = (A[1] - A[n + 1]) * (B[n] + B[n + 1]);
10
11         C[0] = M1 + M4 - M5 + M7;
12         C[1] = M3 + M5;
13         C[n] = M2 + M4;
14         C[n + 1] = M1 + M3 - M2 + M6;
15     }
16     else { /* caso ricorsivo */
17         /* DIVIDE */
18         int m = n/2;
19
20         /* matrici temporanee */
21         int *a = (int *) malloc(m * m * sizeof(int));
22         int *b = (int *) malloc(m * m * sizeof(int));
23
24         /* M1 = (A[0][0] + A[1][1]) * (B[0][0] + B[1][1]) */
25         int *M1 = (int *) malloc(m * m * sizeof(int));
```

```
26          addmat(a, &A[0], &A[m * (n + 1)], m, n);
27          addmat(b, &B[0], &B[m * (n + 1)], m, n);
28          strassen(M1, a, b, m);
29
30          /* M2 = (A[1][0] + A[1][1]) * B[0][0] */
31          int *M2 = (int *) malloc(m * m * sizeof(int));
32          addmat(a, &A[m * n], &A[m * (n + 1)], m, n);
33          matcpy(b, &B[0], m, m, n, n);
34          strassen(M2, a, b, m);
35
36          /* M3 = A[0][0] * (B[0][1] - B[1][1]) */
37          int *M3 = (int *) malloc(m * m * sizeof(int));
38          matcpy(a, &A[0], m, m, n, n);
39          submat(b, &B[m], &B[m * (n + 1)], m, n);
40          strassen(M3, a, b, m);
41
42          /* M4 = A[1][1] * (B[1][0] - B[0][0]) */
43          int *M4 = (int *) malloc(m * m * sizeof(int));
44          matcpy(a, &A[m * (n + 1)], m, m, n, n);
45          submat(b, &B[m * n], &B[0], m, n);
46          strassen(M4, a, b, m);
47
48          /* M5 = (A[0][0] + A[0][1]) * B[1][1] */
49          int *M5 = (int *) malloc(m * m * sizeof(int));
50          addmat(a, &A[0], &A[m], m, n);
51          matcpy(b, &B[m * (n + 1)], m, m, n, n);
52          strassen(M5, a, b, m);
53
54          /* M6 = (A[1][0] - A[0][0]) * (B[0][0] + B[0][1]) */
55          int *M6 = (int *) malloc(m * m * sizeof(int));
56          submat(a, &A[m * n], &A[0], m, n);
57          addmat(b, &B[0], &B[m], m, n);
58          strassen(M6, a, b, m);
59
60          /* M7 = (A[0][1] - A[1][1]) * (B[1][0] + B[1][1]) */
61          int *M7 = (int *) malloc(m * m * sizeof(int));
62          submat(a, &A[m], &A[m * (n + 1)], m, n);
63          addmat(b, &B[m * n], &B[m * (n + 1)], m, n);
64          strassen(M7, a, b, m);
65
66          /* IMPERA */
```

```
67          /* C00 = M1 + M4 - M5 + M7 */
68          int *C00 = (int *) malloc(m * m * sizeof(int));
69          submat(a, M7, M5, m, m);
70          addmat(b, M4, a, m, m);
71          addmat(C00, M1, b, m, m);
72          free(M7);
73
74          /* C01 = M3 + M5 */
75          int *C01 = (int *) malloc(m * m * sizeof(int));
76          addmat(C01, M3, M5, m, m);
77          free(M5);
78
79          /* C10 = M2 + M4 */
80          int *C10 = (int *) malloc(m * m * sizeof(int));
81          addmat(C10, M2, M4, m, m);
82          free(M4);
83
84          /* C11 = M1 + M3 - M2 + M6 */
85          int *C11 = (int *) malloc(m * m * sizeof(int));
86          submat(a, M6, M2, m, m);
87          addmat(b, M3, a, m, m);
88          addmat(C11, M1, b, m, m);
89          free(M6); free(M3); free(M2); free(M1);
90          free(a); free(b);
91
92          /* COMBINA */
93          int i, j;
94          for (i = 0; i < m; ++i)
95              for (j = 0; j < m; ++j) {
96                  C[i * n + j] = C00[i * m + j];
97                  C[i * n + j + m] = C01[i * m + j];
98                  C[(i + m) * n + j] = C10[i * m + j];
99                  C[(i + m) * n + j + m] = C11[i * m + j];
100             }
101
102         free(C00); free(C01); free(C10); free(C11);
103     }
104 }
```

5.2 Algoritmo ibrido di Strassen in ambiente GPU-CUDA

```
1  void strassen(cublasHandle_t handle, float *C, const float *A,
   ↪  const float *B, const int n, const int d, const int
   ↪  threshold) {
2      const int m = n / 2;
3
4      if (n <= threshold) {/* caso base */
5          if (cublas)
6              cublasMatmul(handle, C, A, B, n);
7          else {
8              dim3 blocks((n + BLOCK_DIM - 1) / BLOCK_DIM, (n +
   ↪  BLOCK_DIM - 1) / BLOCK_DIM);
9              dim3 threads(BLOCK_DIM, BLOCK_DIM);
10             matmul<<<blocks, threads>>>(C, A, B, n);
11         }
12     }
13     else {/* caso ricorsivo */
14         /* M1 = (A[0][0] + A[1][1]) * (B[0][0] + B[1][1]) */
15         addmat(handle, A_dev[d + 1], &A[0], &A[m * n + m], m,
   ↪  m, n, n, m);
16         addmat(handle, B_dev[d + 1], &B[0], &B[m * n + m], m,
   ↪  m, n, n, m);
17         strassen(handle, M1[d + 1], A_dev[d + 1], B_dev[d + 1],
   ↪  m, d + 1, threshold);
18
19         /* M2 = (A[1][0] + A[1][1]) * B[0][0] */
20         addmat(handle, A_dev[d + 1], &A[m * n], &A[m * n + m],
   ↪  m, m, n, n, m);
21         matcpy(handle, B_dev[d + 1], &B[0], m, m, n, m);
22         strassen(handle, M2[d + 1], A_dev[d + 1], B_dev[d + 1],
   ↪  m, d + 1, threshold);
23
24         /* M3 = A[0][0] * (B[0][1] - B[1][1]) */
25         matcpy(handle, A_dev[d + 1], &A[0], m, m, n, m);
26         submat(handle, B_dev[d + 1], &B[m], &B[m * n + m], m,
   ↪  m, n, n, m);
27         strassen(handle, M3[d + 1], A_dev[d + 1], B_dev[d + 1],
   ↪  m, d + 1, threshold);
```

```
28
29          /* M4 = A[1][1] * (B[1][0] - B[0][0]) */
30          matcpy(handle, A_dev[d + 1], &A[m * n + m], m, m, n,
    ↪   m);
31          submat(handle, B_dev[d + 1], &B[m * n], &B[0], m, m, n,
    ↪   n, m);
32          strassen(handle, M4[d + 1], A_dev[d + 1], B_dev[d + 1],
    ↪   m, d + 1, threshold);
33
34          /* M5 = (A[0][0] + A[0][1]) * B[1][1] */
35          addmat(handle, A_dev[d + 1], &A[0], &A[m], m, m, n, n,
    ↪   m);
36          matcpy(handle, B_dev[d + 1], &B[m * n + m], m, m, n,
    ↪   m);
37          strassen(handle, M5[d + 1], A_dev[d + 1], B_dev[d + 1],
    ↪   m, d + 1, threshold);
38
39          /* M6 = (A[1][0] - A[0][0]) * (B[0][0] + B[0][1]) */
40          submat(handle, A_dev[d + 1], &A[m * n], &A[0], m, m, n,
    ↪   n, m);
41          addmat(handle, B_dev[d + 1], &B[0], &B[m], m, m, n, n,
    ↪   m);
42          strassen(handle, M6[d + 1], A_dev[d + 1], B_dev[d + 1],
    ↪   m, d + 1, threshold);
43
44          /* M7 = (A[0][1] - A[1][1]) * (B[1][0] + B[1][1]) */
45          submat(handle, A_dev[d + 1], &A[m], &A[m * n + m], m,
    ↪   m, n, n, m);
46          addmat(handle, B_dev[d + 1], &B[m * n], &B[m * n + m],
    ↪   m, m, n, n, m);
47          strassen(handle, M7[d + 1], A_dev[d + 1], B_dev[d + 1],
    ↪   m, d + 1, threshold);
48
49          /* IMPERA (E COMBINA) */
50          /* C00 = M1 + M4 - M5 + M7 */
51          matcpy(handle, &C[0], M1[d + 1], m, m, m, n);
52          addmat(handle, &C[0], &C[0], M4[d + 1], m, m, n, m, n);
53          submat(handle, &C[0], &C[0], M5[d + 1], m, m, n, m, n);
54          addmat(handle, &C[0], &C[0], M7[d + 1], m, m, n, m, n);
55
56          /* C01 = M3 + M5 */
```

```
57          matcpy(handle, &C[m], M3[d + 1], m, m, m, n);
58          addmat(handle, &C[m], &C[m], M5[d + 1], m, m, n, m, n);
59
60          /* C10 = M2 + M4 */
61          matcpy(handle, &C[m * n], M2[d + 1], m, m, m, n);
62          addmat(handle, &C[m * n], &C[m * n], M4[d + 1], m, m,
    ↪   n, m, n);
63
64          /* C11 = M1 + M3 - M2 + M6 */
65          matcpy(handle, &C[m * n + m], M1[d + 1], m, m, m, n);
66          submat(handle, &C[m * n + m], &C[m * n + m], M2[d + 1],
    ↪   m, m, n, m, n);
67          addmat(handle, &C[m * n + m], &C[m * n + m], M3[d + 1],
    ↪   m, m, n, m, n);
68          addmat(handle, &C[m * n + m], &C[m * n + m], M6[d + 1],
    ↪   m, m, n, m, n);
69      }
70  }
```

5.3 Algoritmo naive del prodotto matriciale con uso della shared memory

```
1   __global__ void matmul(float *C, const float *A, const float
    ↪   *B, const int n) {
2       const int i = blockIdx.y * blockDim.y + threadIdx.y;
3       const int j = blockIdx.x * blockDim.x + threadIdx.x;
4       const int nn = gridDim.x;
5
6       __shared__ float sharedA[BLOCK_DIM][BLOCK_DIM],
    ↪   sharedB[BLOCK_DIM][BLOCK_DIM];
7
8       if (i < n && j < n) {
9           float sum = 0;
10          for (int k = 0; k < nn; ++k) {
11              sharedA[threadIdx.y][threadIdx.x] = A[i * n + k *
    ↪   BLOCK_DIM + threadIdx.x];
12              sharedB[threadIdx.y][threadIdx.x] = B[(k *
    ↪   BLOCK_DIM + threadIdx.y) * n + j];
13              __syncthreads();
```

```
14
15              for (int l = 0; l < BLOCK_DIM; ++l)
16                  sum += sharedA[threadIdx.y][l] *
    ↪ sharedB[l][threadIdx.x];
17              __syncthreads();
18          }
19
20          C[i * n + j] = sum;
21      }
22  }
```

5.4 Interfaccia per l'algoritmo di Strassen in ambiente GPU-CUDA

```
1   int main(int argc, char** argv) {
2       if (argc != 4) {
3           fprintf(stderr, "Usage: %s <matrices size> <threshold>
    ↪ <cublas>\n", argv[0]);
4           fputs("cublas:\t0: use Strassen with naive algorithm
    ↪ with SM\n\t1: use Strassen with cuBLAS' algorithm\n",
    ↪ stderr);
5           exit(0);
6       }
7
8       int n = atoi(argv[1]);
9       int threshold = atoi(argv[2]);
10      cublas = (bool) atoi(argv[3]);
11
12      if (!isPowerOfTwo(n)) {
13          /* rectangular matrices */
14          fputs("WARNING: matrices size is not power of two\n",
    ↪ stdout);
15          n = pow(2, ceil(log2((float) n)));
16          printf("dimension of matrices = %d\n", n);
17      }
18
19      dim3 blocks((n + BLOCK_DIM - 1) / BLOCK_DIM, (n + BLOCK_DIM
    ↪ - 1) / BLOCK_DIM);
20      dim3 threads(BLOCK_DIM, BLOCK_DIM);
```

```
21
22      if (n < threshold && threshold < BLOCK_DIM) {
23          fputs("ERROR: matrix size is less than threshold\n",
    ↪   stderr);
24          exit(-2);
25      }
26
27      float *A_host = (float *) calloc(n * n, sizeof(float));
28      float *B_host = (float *) calloc(n * n, sizeof(float));
29      float *C_host = (float *) calloc(n * n, sizeof(float));
30      float *tmp = (float *) calloc(n * n, sizeof(float));
31
32      srand(time(NULL));
33      initm(A_host, n, n);
34      initm(B_host, n, n);
35
36      printm(A_host, n, n, "A");
37      printm(B_host, n, n, "B");
38
39      /* matrici temporanee */
40      int depth, dim = n;
41      for (depth = 0; depth < DEPTH && dim > 0; ++depth, dim /=
    ↪   2) {
42          cudaMalloc((float **) &A_dev[depth], dim * dim *
    ↪   sizeof(float));
43          cudaMalloc((float **) &B_dev[depth], dim * dim *
    ↪   sizeof(float));
44
45          if (depth == 0)
46              cudaMalloc((float **) &C_dev, dim * dim *
    ↪   sizeof(float));
47          else {
48              cudaMalloc((float **) &M1[depth], dim * dim *
    ↪   sizeof(float));
49              cudaMalloc((float **) &M2[depth], dim * dim *
    ↪   sizeof(float));
50              cudaMalloc((float **) &M3[depth], dim * dim *
    ↪   sizeof(float));
51              cudaMalloc((float **) &M4[depth], dim * dim *
    ↪   sizeof(float));
```

```
52              cudaMalloc((float **) &M5[depth], dim * dim *
    ↪ sizeof(float));
53              cudaMalloc((float **) &M6[depth], dim * dim *
    ↪ sizeof(float));
54              cudaMalloc((float **) &M7[depth], dim * dim *
    ↪ sizeof(float));
55          }
56      }
57
58      cudaMemcpy(A_dev[0], A_host, n * n * sizeof(float),
    ↪ cudaMemcpyHostToDevice);
59      cudaMemcpy(B_dev[0], B_host, n * n * sizeof(float),
    ↪ cudaMemcpyHostToDevice);
60
61      cublasHandle_t handle;
62      cublasCreate(&handle);
63
64      Timer t;
65
66      t.start();
67      matmul<<<blocks, threads>>>(C_dev, A_dev[0], B_dev[0], n);
68      cudaDeviceSynchronize();
69      t.stop();
70
71      printf("time for naive: %.5fms\n", t.get());
72      cudaMemcpy(tmp, C_dev, n * n * sizeof(float),
    ↪ cudaMemcpyDeviceToHost);
73
74      t.start();
75      strassen(handle, C_dev, A_dev[0], B_dev[0], n, 0,
    ↪ threshold);
76      t.stop();
77
78      printf("time for strassen: %.5fms\n", t.get());
79      cudaMemcpy(C_host, C_dev, n * n * sizeof(float),
    ↪ cudaMemcpyDeviceToHost);
80
81      if (cublas) {
82          t.start();
83          cublasMatmul(handle, C_dev, A_dev[0], B_dev[0], n);
84          t.stop();
```

```
85
86          printf("time for cublas: %.5fms\n", t.get());
87          cudaMemcpy(C_host, C_dev, n * n * sizeof(float),
    ↪   cudaMemcpyDeviceToHost);
88      }
89
90      cublasDestroy(handle);
91
92      int i;
93      for (i = 0; i < depth; ++i) {
94          cudaFree(A_dev[i]); cudaFree(B_dev[i]);
95
96          if (i == 0)
97              cudaFree(C_dev);
98          else {
99              cudaFree(M1[i]);
100             cudaFree(M2[i]);
101             cudaFree(M3[i]);
102             cudaFree(M4[i]);
103             cudaFree(M5[i]);
104             cudaFree(M6[i]);
105             cudaFree(M7[i]);
106         }
107     }
108
109     printm(C_host, n, n, "C");
110     printf(matcmp(tmp, C_host, n) ? "Strassen OK\n" : "Strassen
    ↪   not working\n");
111
112     cudaDeviceReset();
113
114     free(A_host); free(B_host); free(C_host); free(tmp);
115
116     return 0;
117 }
```