

Accelerazione del prodotto matriciale con l'algoritmo di Strassen

Corso di High Performance Computing

Dario Musella

Università degli Studi di Napoli “Parthenope”

Anno Accademico 2021-2022



Introduzione

Il progetto sviluppato vuole proporre una strategia per l'accelerazione del prodotto tra due matrici in ambiente GPU-CUDA. Partendo dall'algoritmo standard si effettuano delle considerazioni, mattone dopo mattone, al fine di ottenere un algoritmo ottimizzato.

$$\begin{bmatrix} \square & \square & \cdots & \square \\ \square & \square & \cdots & \square \\ \vdots & \vdots & \times & \vdots \\ \square & \square & \cdots & \square \end{bmatrix}$$



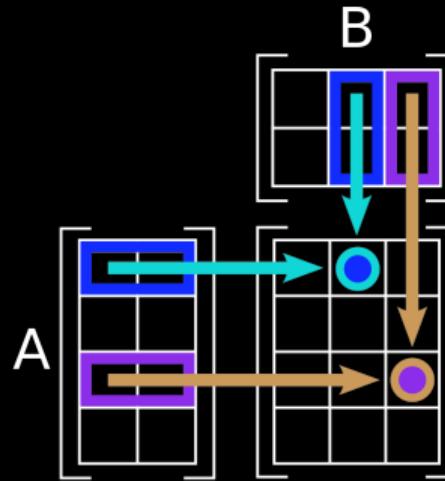
Indice

- ① Analisi e definizione del problema
- ② Descrizione dell'algoritmo in ambiente GPU-CUDA
- ③ Analisi delle prestazioni



Moltiplicazione di matrici

Sia data una matrice **A** di dimensione $m \times n$ e sia data una matrice **B** di dimensione $n \times p$, si ricava la matrice **C** = **A** × **B** di dimensione $m \times p$.



Moltiplicazione di matrici

Il generico elemento c_{ij} è dato dalla seguente somma

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$

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

$$\forall i \in \{1, 2, \dots, m\}, \forall j \in \{1, 2, \dots, p\}$$



Algoritmo

Algoritmo 1 Prodotto matriciale naïve

```

for  $i \leftarrow 1$  to  $m$  do
  for  $j \leftarrow 1$  to  $p$  do
     $c_{ij} \leftarrow 0$ 
    for  $k \leftarrow 1$  to  $n$  do
       $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$ 
    end for
  end for
end for
  
```

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

Complessità di tempo $O(mnp) = O(n^3)$



Analisi dell'algoritmo

Esempio con approccio divide et impera

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

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

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

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

Le quattro sottomatrici possono essere
ottenute con delle somme di prodotti

$$\mathbf{C}_{11} = \mathbf{A}_{11} \cdot \mathbf{B}_{11} + \mathbf{A}_{12} \cdot \mathbf{B}_{21}$$

$$\mathbf{C}_{12} = \mathbf{A}_{11} \cdot \mathbf{B}_{12} + \mathbf{A}_{12} \cdot \mathbf{B}_{22}$$

$$\mathbf{C}_{21} = \mathbf{A}_{21} \cdot \mathbf{B}_{11} + \mathbf{A}_{22} \cdot \mathbf{B}_{21}$$

$$\mathbf{C}_{22} = \mathbf{A}_{21} \cdot \mathbf{B}_{12} + \mathbf{A}_{22} \cdot \mathbf{B}_{22}$$



Analisi dell'algoritmo

Calcolo complessità di tempo

Algoritmo 2 Prodotto matriciale naïve divide et impera

```

function MATMUL(A, B)
    C  $\leftarrow \emptyset$                                  $\triangleright \mathbf{C}$  è una matrice  $n \times n$ 
    if  $n = 1$  then
         $c_{11} \leftarrow a_{11} \cdot b_{11}$ 
    else       $\triangleright$  partiziona A e B in quattro porzioni  $\frac{n}{2} \times \frac{n}{2}$ 
        C11  $\leftarrow$  MATMUL(A11, B11) + MATMUL(A12, B21)
        C12  $\leftarrow$  MATMUL(A11, B12) + MATMUL(A12, B22)
        C21  $\leftarrow$  MATMUL(A21, B11) + MATMUL(A22, B21)
        C22  $\leftarrow$  MATMUL(A21, B12) + MATMUL(A22, B22)
    end if
    return C
end function

```

Relazione di ricorrenza:

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

per il primo caso del metodo dell'esperto si ha

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



Algoritmo di Strassen

Definizione

Algoritmo divide et impera che effettua la moltiplicazione di matrici. Esso calcola sette matrici ausiliarie

$$\mathbf{M}_1 = (\mathbf{A}_{11} + \mathbf{A}_{22})(\mathbf{B}_{11} + \mathbf{B}_{22})$$

$$\mathbf{M}_2 = (\mathbf{A}_{21} + \mathbf{A}_{22})\mathbf{B}_{11}$$

$$\mathbf{M}_3 = \mathbf{A}_{11}(\mathbf{B}_{12} - \mathbf{B}_{22})$$

$$\mathbf{M}_4 = \mathbf{A}_{22}(\mathbf{B}_{21} - \mathbf{B}_{11})$$

$$\mathbf{M}_5 = (\mathbf{A}_{11} + \mathbf{A}_{12})\mathbf{B}_{22}$$

$$\mathbf{M}_6 = (\mathbf{A}_{21} - \mathbf{A}_{11})(\mathbf{B}_{11} + \mathbf{B}_{12})$$

$$\mathbf{M}_7 = (\mathbf{A}_{12} - \mathbf{A}_{22})(\mathbf{B}_{21} + \mathbf{B}_{22})$$



Algoritmo di Strassen

Definizione

Rispetto a prima, si ricombinano sette matrici invece di otto

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

In questo modo si riduce la complessità computazionale di tempo della moltiplicazione matriciale.



Algoritmo di Strassen

Algoritmo 3 Algoritmo di Strassen

```

function STRASSEN(A, B)
    C  $\leftarrow \emptyset$                                  $\triangleright \mathbf{C}$  è una matrice  $n \times n$ 
    if  $n = 1$  then
         $c_{11} \leftarrow a_{11} \cdot b_{11}$ 
    else       $\triangleright$  partiziona A e B in quattro porzioni  $\frac{n}{2} \times \frac{n}{2}$ 
        M1  $\leftarrow$  STRASSEN(A11 + A22, B11 + B22)
        M2  $\leftarrow$  STRASSEN(A21 + A22, B11)
        M3  $\leftarrow$  STRASSEN(A11, B12 - B22)
        M4  $\leftarrow$  STRASSEN(A22, B21 - B11)
        M5  $\leftarrow$  STRASSEN(A11 + A12, B22)
        M6  $\leftarrow$  STRASSEN(A21 - A11, B11 + B12)
        M7  $\leftarrow$  STRASSEN(A12 - A21, B21 + B22)
        C11  $\leftarrow$  M1 + M4 - M5 + M7
        C12  $\leftarrow$  M3 + M5
        C21  $\leftarrow$  M2 + M4
        C22  $\leftarrow$  M1 - M2 + M3 + M6
    end if
    return C
end function

```

Relazione di ricorrenza:

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

per il primo caso del metodo dell'esperto si ha

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



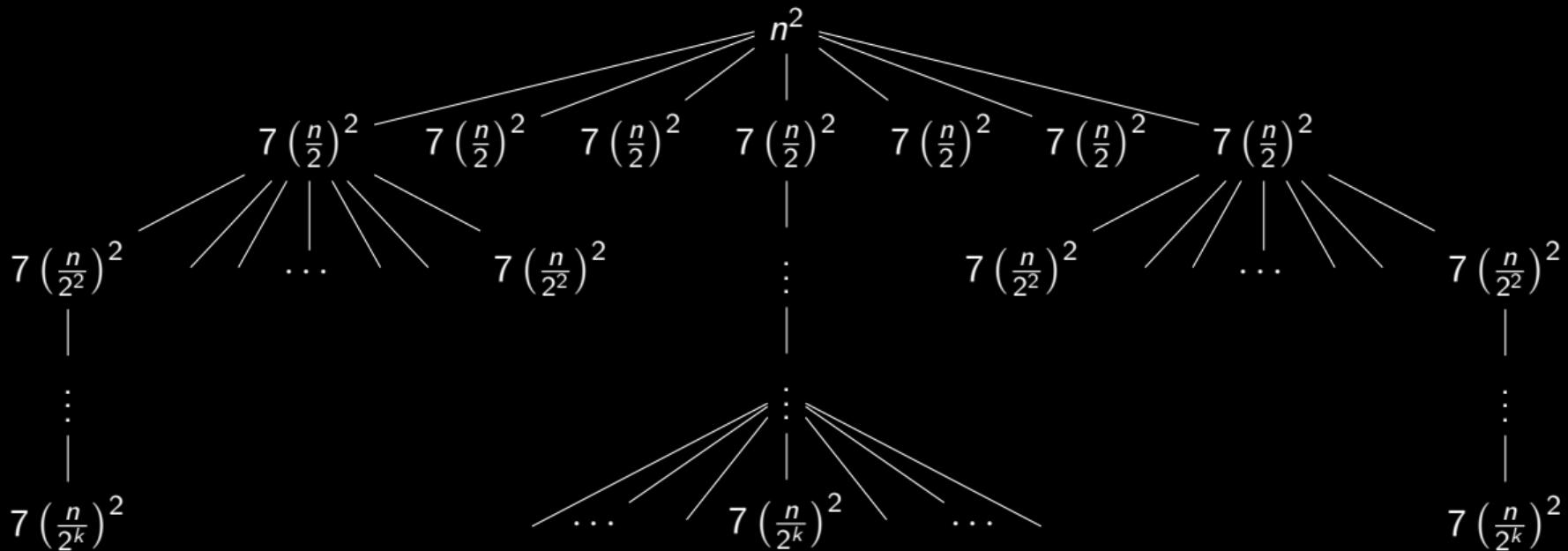
Confronto

	Algoritmo naïve	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



Algoritmo di Strassen

Albero di ricorsione



Studio dell'algoritmo in ambiente CPU sequenziale

```
void strassen(int *C, const int *A, const int *B, const int n) {  
    if (n == 2) { /* caso base */  
        int M1 = (A[0] + A[n + 1]) * (B[0] + B[n + 1]);  
        int M2 = (A[n] + A[n + 1]) * B[0];  
        int M3 = A[0] * (B[1] - B[n + 1]);  
        int M4 = A[n + 1] * (B[n] - B[0]);  
        int M5 = (A[0] + A[1]) * B[n + 1];  
        int M6 = (A[n] - A[0]) * (B[0] + B[1]);  
        int M7 = (A[1] - A[n + 1]) * (B[n] + B[n + 1]);  
  
        C[0] = M1 + M4 - M5 + M7;  
        C[1] = M3 + M5;  
        C[n] = M2 + M4;  
        C[n + 1] = M1 + M3 - M2 + M6;  
    }  
    else { /* caso ricorsivo */  
        ...  
    }  
}
```



Studio dell'algoritmo in ambiente CPU sequenziale

```

else { /* caso ricorsivo */
    /* DIVIDE */
    int m = n/2;
    /* matrici temporanee */
    int *a = (int *) malloc(m * m * sizeof(int));
    int *b = (int *) malloc(m * m * sizeof(int));

    int *M1 = (int *) malloc(m * m * sizeof(int));
    addmat(a, &A[0], &A[m * (n + 1)], m, n);
    addmat(b, &B[0], &B[m * (n + 1)], m, n);
    strassen(M1, a, b, m);

    int *M2 = (int *) malloc(m * m * sizeof(int));
    addmat(a, &A[m * n], &A[m * (n + 1)], m, n);
    matcpy(b, &B[0], m, m, n, n);
    strassen(M2, a, b, m);

    int *M3 = (int *) malloc(m * m * sizeof(int));
    matcpy(a, &A[0], m, m, n, n);
    submat(b, &B[m], &B[m * (n + 1)], m, n);
    strassen(M3, a, b, m);

    int *M4 = (int *) malloc(m * m * sizeof(int));
    matcpy(a, &A[m * (n + 1)], m, m, n, n);
    submat(b, &B[m * n], &B[0], m, n);
    strassen(M4, a, b, m);

    int *M5 = (int *) malloc(m * m * sizeof(int));
    addmat(a, &A[0], &A[m], m, n);
    matcpy(b, &B[m * (n + 1)], m, m, n, n);
    strassen(M5, a, b, m);

    int *M6 = (int *) malloc(m * m * sizeof(int));
    submat(a, &A[m * n], &A[0], m, n);
    addmat(b, &B[0], &B[m], m, n);
    strassen(M6, a, b, m);

    int *M7 = (int *) malloc(m * m * sizeof(int));
    submat(a, &A[m], &A[m * (n + 1)], m, n);
    addmat(b, &B[m * n], &B[m * (n + 1)], m, n);
    strassen(M7, a, b, m);
}

```

Studio dell'algoritmo in ambiente CPU sequenziale

```

/* IMPERA */
int *C00 = (int *) malloc(m * m * sizeof(int));
submat(a, M7, M5, m, m);
addmat(b, M4, a, m, m);
addmat(C00, M1, b, m, m);
free(M7);

int *C01 = (int *) malloc(m * m * sizeof(int));
addmat(C01, M3, M5, m, m);
free(M5);

int *C10 = (int *) malloc(m * m * sizeof(int));
addmat(C10, M2, M4, m, m);
free(M4);

int *C11 = (int *) malloc(m * m * sizeof(int));
submat(a, M6, M2, m, m);
addmat(b, M3, a, m, m);
addmat(C11, M1, b, m, m);
free(M6); free(M3); free(M2); free(M1);

```

/ COMBINA */*

```

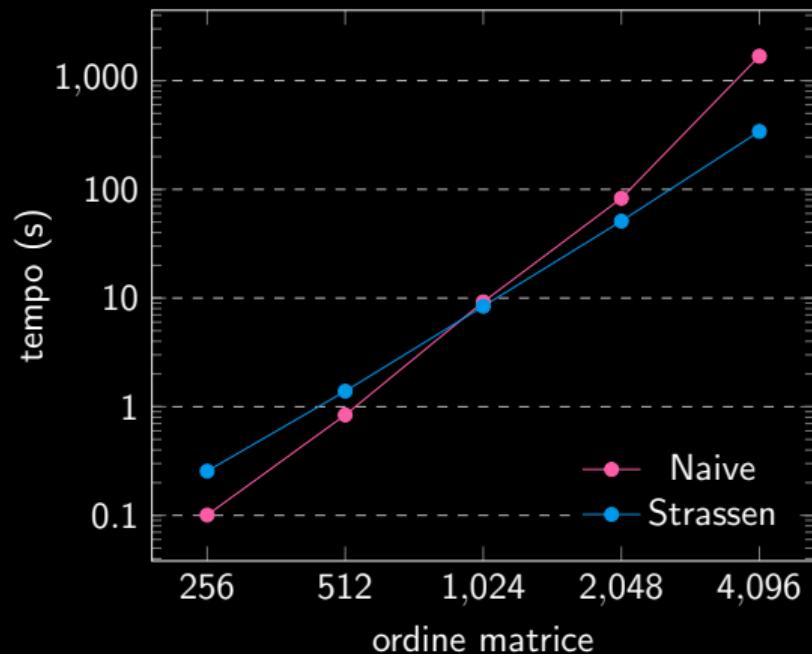
int i, j;
for (i = 0; i < m; ++i)
    for (j = 0; j < m; ++j) {
        C[i * n + j] = C00[i * m + j];
        C[i * n + j + m] = C01[i * m + j];
        C[(i + m) * n + j] = C10[i * m + j];
        C[(i + m) * n + j + m] = C11[i * m + j];
    }
}
free(C00); free(C01); free(C10); free(C11);
}

```

Studio dell'algoritmo in ambiente CPU sequenziale

Tempi di esecuzione

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



Rifiniture e ottimizzazioni

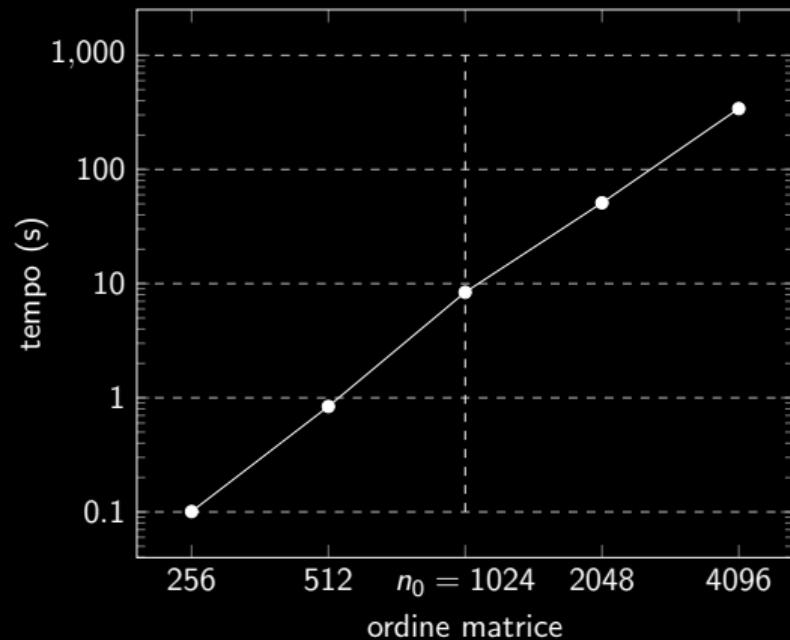
L'idea è unire i due algoritmi!

Algoritmo 4 Algoritmo ibrido del prodotto matriciale

```

function STRASSEN(A, B)
    C  $\leftarrow \emptyset$                                  $\triangleright \mathbf{C}$  è una matrice  $n \times n$ 
    if  $n < n_0$  then
        C  $\leftarrow$  MATMUL(A, B)
    else       $\triangleright$  partiziona A e B in quattro porzioni  $\frac{n}{2} \times \frac{n}{2}$ 
        M1  $\leftarrow$  STRASSEN(A11 + A22, B11 + B22)
        M2  $\leftarrow$  STRASSEN(A21 + A22, B11)
        M3  $\leftarrow$  STRASSEN(A11, B12 - B22)
        M4  $\leftarrow$  STRASSEN(A22, B21 - B11)
        M5  $\leftarrow$  STRASSEN(A11 + A12, B22)
        M6  $\leftarrow$  STRASSEN(A21 - A11, B11 + B12)
        M7  $\leftarrow$  STRASSEN(A12 - A21, B21 + B22)
        C11  $\leftarrow$  M1 + M4 - M5 + M7
        C12  $\leftarrow$  M3 + M5
        C21  $\leftarrow$  M2 + M4
        C22  $\leftarrow$  M1 - M2 + M3 + M6
    end if
    return C
end function

```



Parallelizzazione in ambiente GPU-CUDA

Problematiche da considerare

- Struttura ricorsiva del codice
- Allocazione dinamica delle matrici
- Cosa si può effettivamente parallelizzare?



Parallelizzazione in ambiente GPU-CUDA

Analisi struttura ricorsiva del codice

Algoritmo 5 Algoritmo parallelizzabile ibrido del prodotto matriciale

```

function STRASSEN(A, B)
    C  $\leftarrow \emptyset$                                  $\triangleright \mathbf{C}$  è una matrice  $n \times n$ 
    if  $n < n_0$  then
        C  $\leftarrow$  MATMUL(A, B)                 $\triangleright$  parallelizzabile
    else                                          $\triangleright$  partiziona A e B in quattro porzioni  $\frac{n}{2} \times \frac{n}{2}$ 
        M1  $\leftarrow$  STRASSEN(A11 + A22, B11 + B22)       $\triangleright$  parametri parallelizzabili
        M2  $\leftarrow$  STRASSEN(A21 + A22, B11)           $\triangleright$  parametri parallelizzabili
        M3  $\leftarrow$  STRASSEN(A11, B12 - B22)           $\triangleright$  parametri parallelizzabili
        M4  $\leftarrow$  STRASSEN(A22, B21 - B11)           $\triangleright$  parametri parallelizzabili
        M5  $\leftarrow$  STRASSEN(A11 + A12, B22)           $\triangleright$  parametri parallelizzabili
        M6  $\leftarrow$  STRASSEN(A21 - A11, B11 + B12)       $\triangleright$  parametri parallelizzabili
        M7  $\leftarrow$  STRASSEN(A12 - A21, B21 + B22)       $\triangleright$  parametri parallelizzabili
        C11  $\leftarrow$  M1 + M4 - M5 + M7            $\triangleright$  parallelizzabile
        C12  $\leftarrow$  M3 + M5                          $\triangleright$  parallelizzabile
        C21  $\leftarrow$  M2 + M4                          $\triangleright$  parallelizzabile
        C22  $\leftarrow$  M1 - M2 + M3 + M6            $\triangleright$  parallelizzabile
    end if
    return C
end function

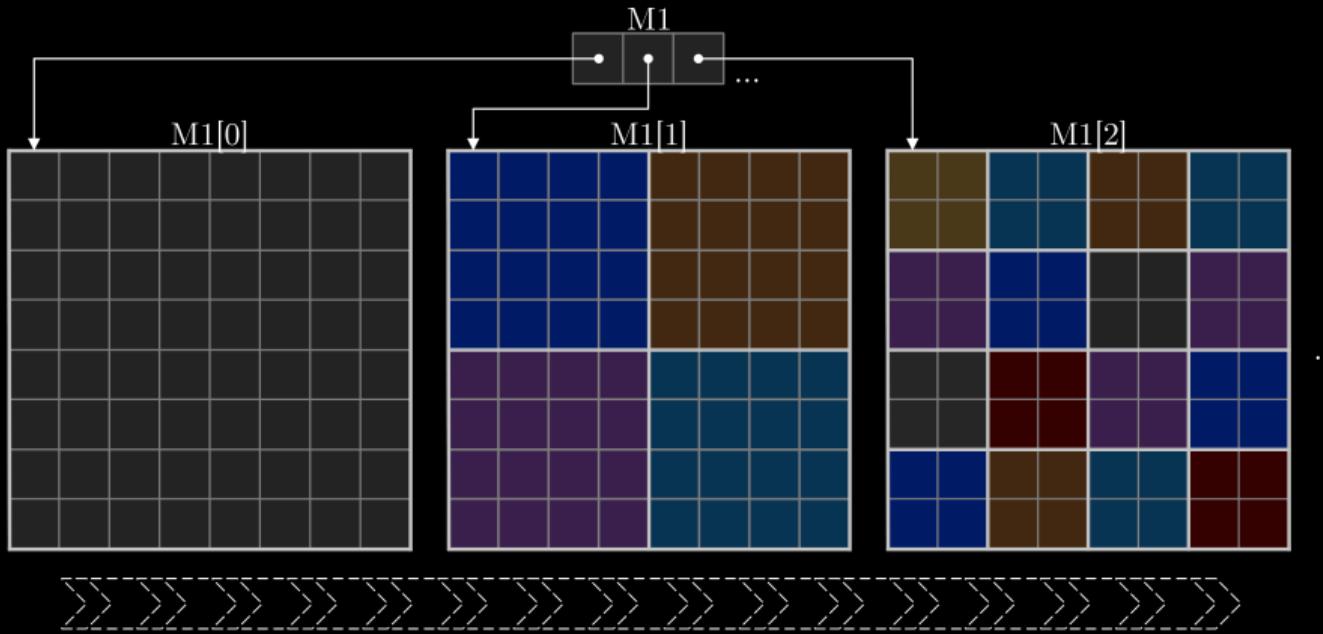
```



Parallelizzazione in ambiente GPU-CUDA

Allocazione dinamica delle matrici

Si può gestire la memoria di $\mathbf{M}_1, \dots, \mathbf{M}_7$ sulla falsariga della programmazione dinamica.



Parallelizzazione in ambiente GPU-CUDA

Quali operazioni sono parallelizzabili?

- Addizioni e sottrazioni tra matrici sono operazioni parallelizzabili
 - Buona parte dell'algoritmo è composto da addizioni e sottrazioni
 - Si può fare uso di librerie che gestiscono il parallelismo di queste operazioni, come cuBLAS
- Il caso base dell'algoritmo viene risolto con un algoritmo altrettanto parallelizzabile
 - Ad esempio ricorrere all'algoritmo naive per il prodotto matriciale o alla routine di cuBLAS



Parallelizzazione in ambiente GPU-CUDA

Parallelizzazione delle operazioni

```
void addmat(cublasHandle_t handle, float *C, const float *A, const float *B, const int m, const int
→ n, const int lda, const int ldb, const int ldc) {
    const float plus = 1.0;
    cublasSgem(handle, CUBLAS_OP_N, CUBLAS_OP_N, m, n, &plus, A, lda, &plus, B, ldb, C, ldc);
}

void submat(cublasHandle_t handle, float *C, const float *A, const float *B, const int m, const int
→ n, const int lda, const int ldb, const int ldc) {
    const float minus = -1.0;
    const float plus = 1.0;
    cublasSgem(handle, CUBLAS_OP_N, CUBLAS_OP_N, m, n, &plus, A, lda, &minus, B, ldb, C, ldc);
}

void matcpy(cublasHandle_t handle, float *B, const float *A, const int m, const int n, const int lda,
→ const int ldb) {
    const float zero = 0.0;
    const float plus = 1.0;
    cublasSgem(handle, CUBLAS_OP_N, CUBLAS_OP_N, m, n, &plus, A, lda, &zero, B, ldb, B, ldb);
}
```

Algoritmo in ambiente GPU-CUDA

```
void strassen(cublasHandle_t h, float *C, const float *A, const float *B, const int n,
← const int d, const int threshold) {
    const int m = n/2;

    if (n <= threshold) {/* caso base */
        if (cublas)
            cublasMatmul(handle, C, A, B, n);
        else {
            dim3 blocks((n + BLOCK_DIM - 1) / BLOCK_DIM, (n + BLOCK_DIM - 1) / BLOCK_DIM);
            dim3 threads(BLOCK_DIM, BLOCK_DIM);
            matmul<<<blocks, threads>>>(C, A, B, n);
        }
    }
    else { /* caso ricorsivo */
        ...
    }
}
```



Algoritmo in ambiente GPU-CUDA

```

else /* caso ricorsivo */
/* DIVIDE */
addmat(h, A_dev[d+1], &A[0], &A[m*n+m], ...);
addmat(h, B_dev[d+1], &B[0], &B[m*n+m], ...);
strassen(h, M1[d+1], A_dev[d+1], B_dev[d+1],
↪ m, d+1, threshold);

addmat(h, A_dev[d+1], &A[m*n], &A[m*n+m],
↪ ...);
matcpy(h, B_dev[d+1], &B[0], ...);
strassen(h, M2[d+1], A_dev[d+1], B_dev[d+1],
↪ m, d+1, threshold);

matcpy(h, A_dev[d+1], &A[0], ...);
submat(h, B_dev[d+1], &B[m], &B[m*n+m], ...);
strassen(h, M3[d+1], A_dev[d+1], B_dev[d+1],
↪ m, d+1, threshold);

```

```

matcpy(h, A_dev[d+1], &A[m*n+m], ...);
submat(h, B_dev[d+1], &B[m*n], &B[0], ...);
strassen(h, M4[d+1], A_dev[d+1], B_dev[d+1], m,
↪ d+1, threshold);

addmat(h, A_dev[d+1], &A[0], &A[m], ...);
matcpy(h, B_dev[d+1], &B[m*n+m], ...);
strassen(h, M5[d+1], A_dev[d+1], B_dev[d+1], m,
↪ d+1, threshold);

submat(h, A_dev[d+1], &A[m*n], &A[0], ...);
addmat(h, B_dev[d+1], &B[0], &B[m], ...);
strassen(h, M6[d+1], A_dev[d+1], B_dev[d+1], m,
↪ d+1, threshold);

submat(h, A_dev[d+1], &A[m], &A[m*n+m], ...);
addmat(h, B_dev[d+1], &B[m*n], &B[m*n+m], ...);
strassen(h, M7[d+1], A_dev[d+1], B_dev[d+1], m,
↪ d+1, threshold);

```

Algoritmo in ambiente GPU-CUDA

```

/* IMPERA (& COMBINA) */
/* CO0 = M1 + M4 - M5 + M7 */
matcpy(handle, &C[0], M1[d + 1], ...);
addmat(handle, &C[0], &C[0], M4[d + 1], ...);
submat(handle, &C[0], &C[0], M5[d + 1], ...);
addmat(handle, &C[0], &C[0], M7[d + 1], ...);

/* C01 = M3 + M5 */
matcpy(handle, &C[m], M3[d + 1], ...);
addmat(handle, &C[m], &C[m], M5[d + 1], ...);

/* C10 = M2 + M4 */
matcpy(handle, &C[m * n], M2[d + 1], ...);
addmat(handle, &C[m * n], &C[m * n], M4[d + 1], ...);

/* C11 = M1 + M3 - M2 + M6 */
matcpy(handle, &C[m * n + m], M1[d + 1], ...);
submat(handle, &C[m * n + m], &C[m * n + m], M2[d + 1], ...);
addmat(handle, &C[m * n + m], &C[m * n + m], M3[d + 1], ...);
addmat(handle, &C[m * n + m], &C[m * n + m], M6[d + 1], ...);
}

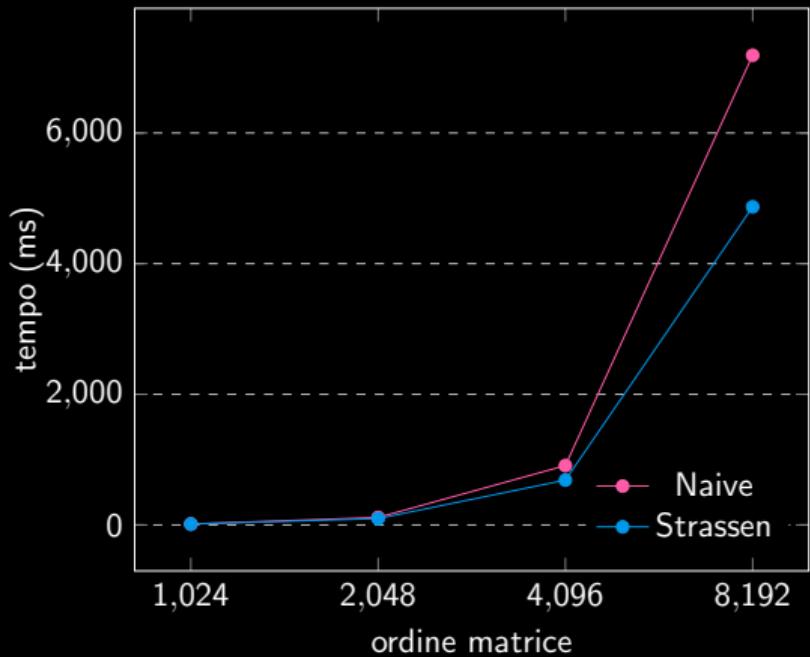
}

```



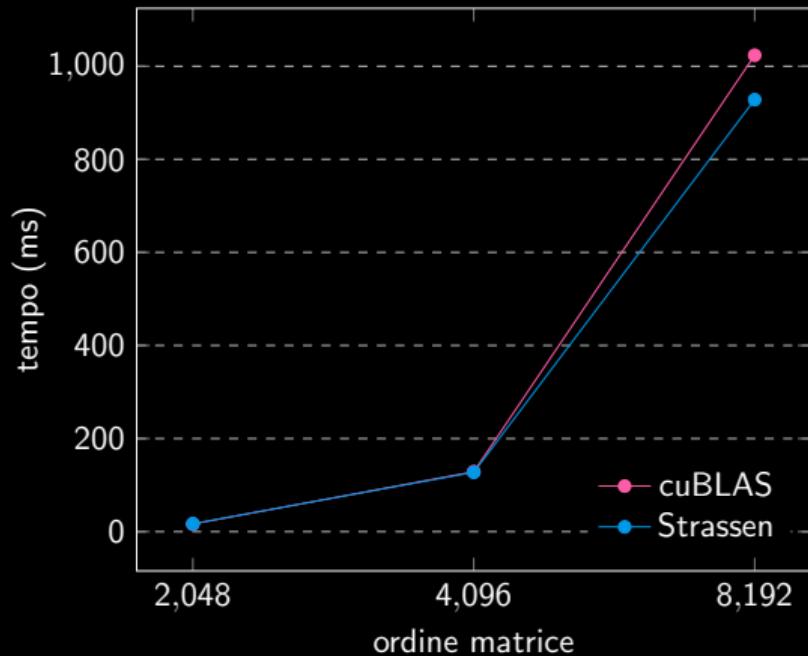
Algoritmo ibrido di Strassen con algoritmo naive

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



Algoritmo ibrido di Strassen con algoritmo cuBLAS

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



Conclusioni

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.

Dai test svolti, qualsiasi algoritmo (non ricorsivo) capace di risolvere il caso base del problema permette di ottenere benefici prestazionali ulteriori se "trapiantati" nell'algoritmo di Strassen.



GRAZIE PER L'ATTENZIONE

