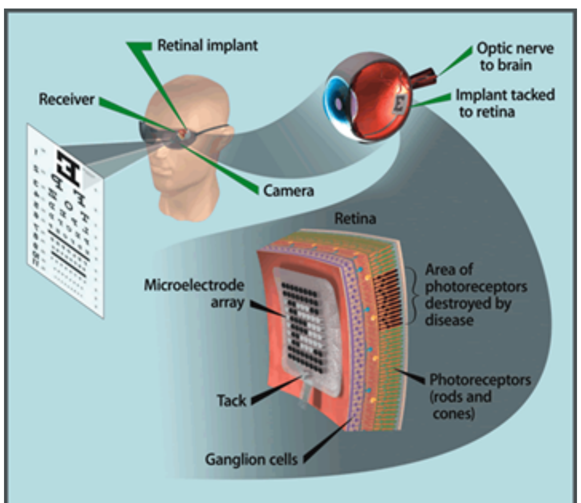
Análisis de Algoritmos

Algoritmo de Strassen

Tania Patiño
Víctor Peña
Javier Sagastuy
Ernesto Valdés

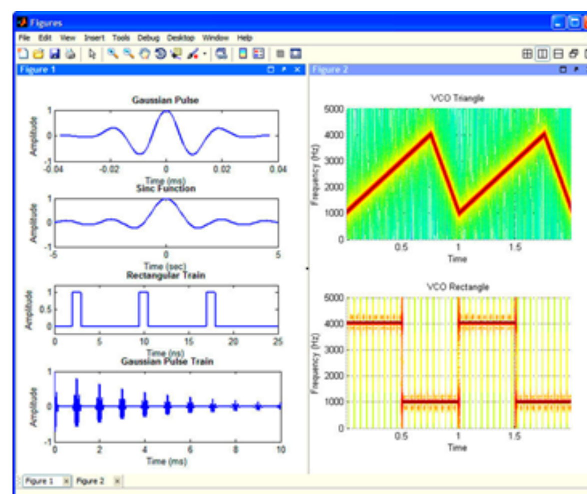
Motivación

Sabemos que la multiplicación de matrices, es una operación habitual en numerosos algoritmos de diferentes áreas de la ingeniería y la computación. Por ejemplo, en el área de **visión artificial** cada imagen es representada por una matriz. En el área de **procesamiento de señales** se aplica a la manipulación matemática de una señal de información para modificarla o mejorarla en algún sentido.



visión artificial

procesamiento de señales



Motivación

La multiplicación de matrices consiste de un alto número de operaciones aritméticas, es por esto que con el objetivo de optimizar estas operaciones y observar los tiempos de ejecución en los experimentos fue que comparamos los **tiempos de ejecución** de:

multiplicación de matrices

vs.

multiplicación de matrices aplicando el algoritmo de Strassen.

Multiplicación de Matrices *O* (n^3)

Procedimiento en python:

```
def prod(A,B):
        (m,n) = A.shape
        (n,k) = B.shape
        C = np.zeros(shape=(m,k))
        for i in range(m):
                for j in range(k):
                        for l in range(n):
                                C[i,j] += A[i,l]*B[l,j]
        return C
```

Algoritmo de Strassen

El algoritmo de Volker Strassen, permite calcular multiplicaciones de matrices de gran tamaño, llevando a cabo un menor número de operaciones numéricas que el producto usual de matrices.

El funcionamiento consiste de la división de una matriz cuadrada de 2^n x 2^n elementos en cuatro submatrices de 2^{m-1} x 2^{m-1}

Entonces se aplica el mismo algoritmo recursivamente a las submatrices hasta obtener matrices de 1x1. Se reconstruye el resultado por bloques.

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix}$$

Algoritmo de Strassen

Llevando a cabo operaciones con estas dos matrices tenemos el mismo número de operaciones que con la multiplicación estándar: **8 operaciones.**

$$
\begin{aligned}
\mathbf{C}_{1,1} &= \mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1} \\
\mathbf{C}_{1,2} &= \mathbf{A}_{1,1}\mathbf{B}_{1,2} + \mathbf{A}_{1,2}\mathbf{B}_{2,2} \\
\mathbf{C}_{2,1} &= \mathbf{A}_{2,1}\mathbf{B}_{1,1} + \mathbf{A}_{2,2}\mathbf{B}_{2,1} \\
\mathbf{C}_{2,2} &= \mathbf{A}_{2,1}\mathbf{B}_{1,2} + \mathbf{A}_{2,2}\mathbf{B}_{2,2}
\end{aligned}
$$

Algoritmo de Strassen

El algoritmo funciona como sigue:

Calculamos recursivamente las siguientes siete matrices.

$$
\begin{aligned}
\mathbf{M}_1 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2}) \\
\mathbf{M}_2 &:= (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1} \\
\mathbf{M}_3 &:= \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2}) \\
\mathbf{M}_4 &:= \mathbf{A}_{2,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1}) \\
\mathbf{M}_5 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\mathbf{B}_{2,2} \\
\mathbf{M}_6 &:= (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2}) \\
\mathbf{M}_7 &:= (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2})
\end{aligned}
$$

Eliminamos una multiplicación y ahora tenemos **7 multiplicaciones** en total expresadas en M_k

Algoritmo de Strassen

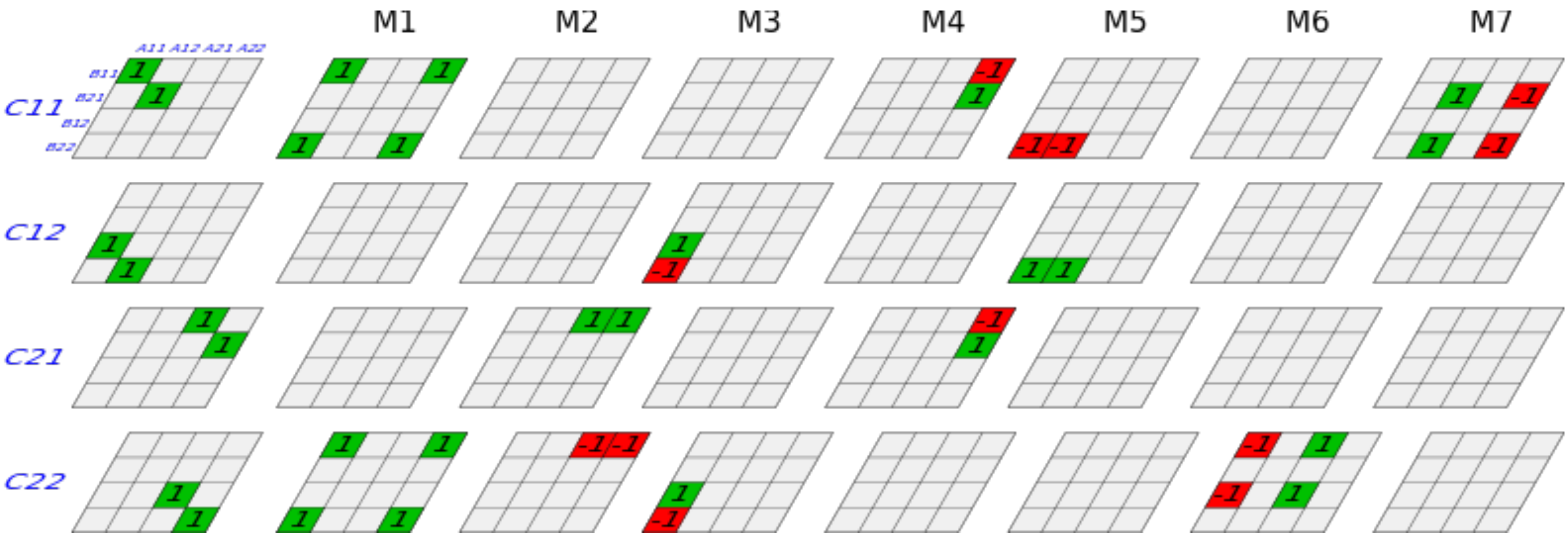
Finalmente estás multiplicaciones se expresan en cuatro bloques como sigue:

$$\begin{aligned}
\mathbf{C}_{1,1} &= \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 \\
\mathbf{C}_{1,2} &= \mathbf{M}_3 + \mathbf{M}_5 \\
\mathbf{C}_{2,1} &= \mathbf{M}_2 + \mathbf{M}_4 \\
\mathbf{C}_{2,2} &= \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6
\end{aligned}$$

$$\mathbf{C} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix}$$

Vamos a describir a continuación el análisis, complejidad e implementación del algoritmo Strassen.

Una manera de verlo



Análisis

Sea m la longitud de uno de los lados de una de las matrices de entrada. El problema se divide en cuatro submatrices de $\frac{m}{2}$ de longitud por lado. Así, con la notación del teorema maestro,

$$b = 2$$

En cada iteración se calculan 7 productos de forma recursiva con las matrices de tamaño reducido. Esto implica

$$a = 7$$

Finalmente, la división del problema en subproblemas requiere sólo de obtener ocho submatrices y realizar varias sumas de matrices de tamaño reducido lo cual es

$$\theta\left(\frac{n^2}{4}\right) = \theta(n^2)$$

(Ésta es también la complejidad asintótica de combinar soluciones parciales)

Complejidad

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

Así, al aplicar el teorema maestro, tenemos que

$$n^{\log_2 7} = n^r$$

con $r = \log_2 7 = 2.80735$. Puesto que claramente

$$\theta(n^2)\text{"} < \text{"}\theta(n^{r-\varepsilon}) \quad \forall\, 0 < \varepsilon < r - 2 = 0.80735$$

el teorema maestro nos permite concluir que

$$T(n) = \theta\left(n^{\log_b a}\right) = \theta\left(n^r\right) = \theta\left(n^{2.80735}\right)$$

Implementaciones

Se realizaron tres implementaciones:

- Python
- Java
- Matlab

Implementación Python

```
import numpy as np
import math
import optparse
import time

def strassen(A,B):
    (ma,na) = A.shape
    (mb,nb) = B.shape
    if na != mb:
        print 'Dimensions dont agree'
    else:
        n = max(ma,na,nb)
        m = math.ceil(math.log(n,2))
        A0 = np.zeros(shape=(2**m, 2**m))
        B0 = np.zeros(shape=(2**m, 2**m))
        A0[0:ma,0:na] = A
        B0[0:mb,0:nb] = B
        C0 = strassenR(A0,B0)
        return C0[0:ma,0:nb]

def strassenR(A,B):
    l = len(A)
    if l == 1:
        return A.dot(B)
    A11 = A[0:l/2,0:l/2]
    A12 = A[0:l/2,l/2:l]
    A21 = A[l/2:l,0:l/2]
    A22 = A[l/2:l,l/2:l]
    B11 = B[0:l/2,0:l/2]
    B12 = B[0:l/2,l/2:l]
    B21 = B[l/2:l,0:l/2]
    B22 = B[l/2:l,l/2:l]
```

```

    M1 = strassenR(A11+A22, B11+B22)
    M2 = strassenR(A21+A22, B11)
    M3 = strassenR(A11, B12-B22)
    M4 = strassenR(A22, B21-B11)
    M5 = strassenR(A11+A12, B22)
    M6 = strassenR(A21-A11, B11+B12)
    M7 = strassenR(A12-A22, B21+B22)

    C11 = M1 + M4 - M5 + M7
    C12 = M3 + M5
    C21 = M2 + M4
    C22 = M1 - M2 + M3 + M6

    C = np.zeros(shape=(l,l))
    C[0:l/2,0:l/2] = C11
    C[0:l/2,l/2:l] = C12
    C[l/2:l,0:l/2] = C21
    C[l/2:l,l/2:l] = C22
    return C

def prod(A,B):
    (m,n) = A.shape
    (n,k) = B.shape
    C = np.zeros(shape=(m,k))
    for i in range(m):
        for j in range(k):
            C[i,j] = sum(A[i,:]*B[:,j])
    return C
```

Implementación Java

Las matrices se implementaron utilizando vectores bidimensionales.

```java
    public Matrix strassen(Matrix A, Matrix B) {
        int rows, columns, max;
        double log, twoPow;

        Matrix A0, B0, C0;

        rows = B.rows;
        columns = A.columns;
        C0 = null;

        if (rows != columns) {
            System.out.println("Las matrices no son multiplicables");
            System.exit(-1);
        } else {
            max = Math.max(A.columns, Math.max(A.rows, B.columns));
            log = Math.log(max) / Math.log(2);
            twoPow = Math.ceil(log);
            twoPow = Math.pow(2, twoPow);

            A0 = new Matrix(A, (int) twoPow);
            B0 = new Matrix(B, (int) twoPow);
            C0 = strassenR(A0, B0);
        }
        return C0.sub(0, A.rows-1, 0, A.columns -1);
    }
```

Implementación Java

```java
private Matrix strassenR(Matrix A, Matrix B) {
    int l;
    Matrix A11, A12, A21, A22, B11, B12, B21, B22, M1, M2, M3, M4, M5, M6, M7;
    Matrix C11,C12,C21,C22,C;

    l = A.matrixLength(A);
    if (l == 1) {
        return Matrix.multiply(A, B);
    }
    A11 = A.sub(0, (l / 2) - 1, 0, (l / 2) - 1);
    A12 = A.sub(0, (l / 2) - 1, (l / 2), l - 1);
    A21 = A.sub((l / 2), l - 1, 0, (l / 2) - 1);
    A22 = A.sub((l / 2), l - 1, (l / 2), l - 1);

    B11 = B.sub(0, (l / 2) - 1, 0, (l / 2) - 1);
    B12 = B.sub(0, (l / 2) - 1, (l / 2), l - 1);
    B21 = B.sub((l / 2), l - 1, 0, (l / 2) - 1);
    B22 = B.sub((l / 2), l - 1, (l / 2), l - 1);

    M1 = strassenR(Matrix.add(A11, A22), Matrix.add(B11, B22));
    M2 = strassenR(Matrix.add(A21, A22), B11);
    M3 = strassenR(A11, Matrix.subtract(B12, B22));
    M4 = strassenR(A22, Matrix.subtract(B21, B11));
    M5 = strassenR(Matrix.add(A11, A12), B22);
    M6 = strassenR(Matrix.subtract(A21, A11), Matrix.add(B11, B12));
    M7 = strassenR(Matrix.subtract(A12, A22), Matrix.add(B21, B22));

    C11 = Matrix.add(Matrix.subtract((Matrix.add(M1, M4)), M5),M7);
    C12 = Matrix.add(M3, M5);
    C21 = Matrix.add(M2,M4);
    C22 = Matrix.add(Matrix.add(Matrix.subtract(M1, M2), M3),M6);

    C = joinMatrix(C11,C12,C21,C22);
```

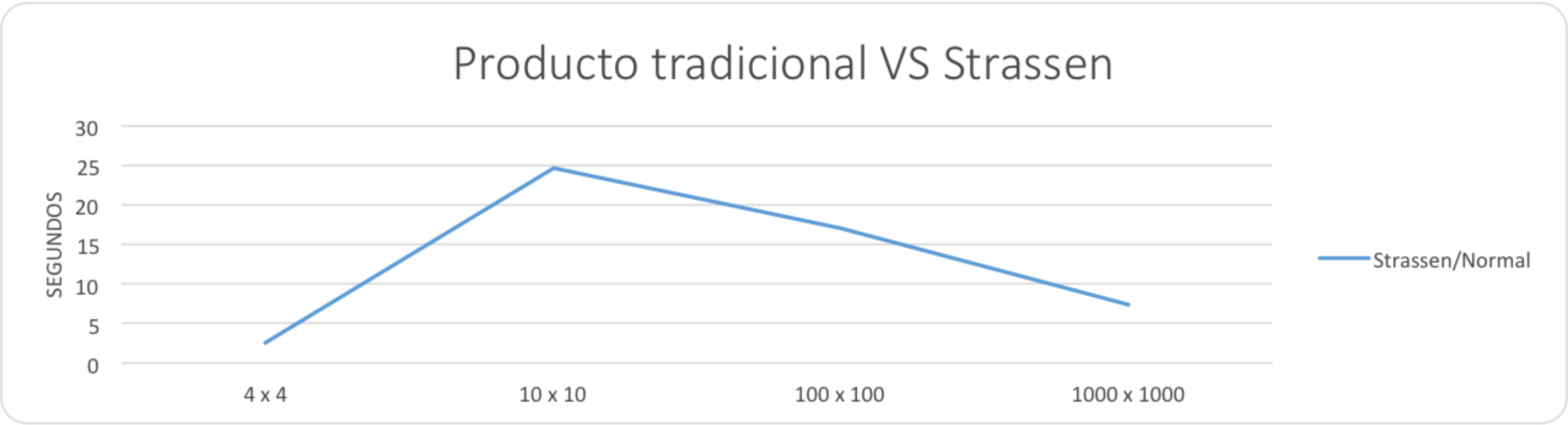
Resultados

Se comparó el producto de matrices tradicional y el algoritmo de Strassen:

Caso Python

Dimensión matriz	Producto tradicional	Strassen
4 x 4	0.000890016555786 seg	0.0022599697113 seg
10 x 10	0.000956058502197 seg	0.0236649513245 seg
100 x 100	0.441963911057 seg	7.53841590881 seg
1000 x 1000	352.728410006 seg	2596.56640506 seg
4000 x 4000		

Resultados

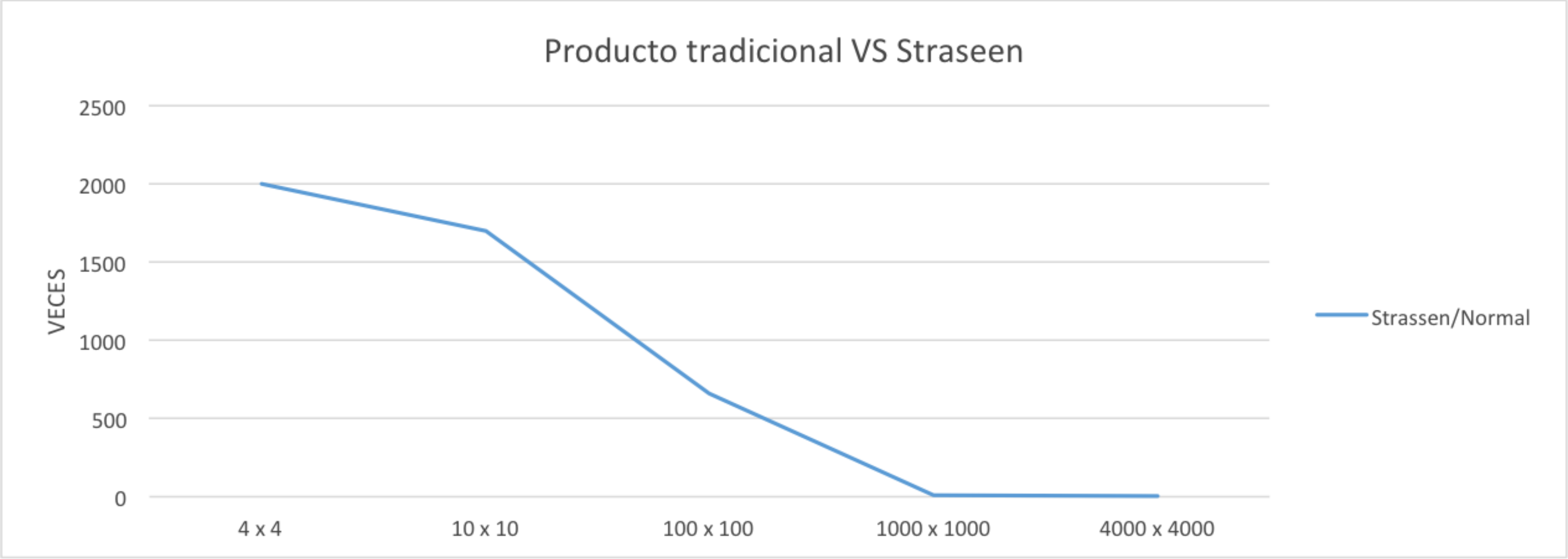

Producto tradicional VS Strassen
SEGUNDOS
30
25
20
15
10
5
0
4 x 4
10 x 10
100 x 100
1000 x 1000
Strassen/Normal

Resultados

Caso Java

Dimensión matriz	Producto tradicional	Strassen
4 x 4	0 seg	0.002 seg
10 x 10	0 seg	0.017 seg
100 x 100	0.002 seg	1.324 seg
1000 x 1000	20.429 seg	252.179 seg
4000 x 4000	3180.524 seg	13802.893 seg

Resultados



Producto tradicional VS Straseen
2500
2000
1500
1000
500
0
VECES
4 x 4
10 x 10
100 x 100
1000 x 1000
4000 x 4000
Strassen/Normal

Análisis de los resultados

A partir de la ejecución con diversas matrices se concluye lo siguiente:

1. El producto tradicional resulta mejor para matrices pequeñas.

2. Al aumentar las dimensiones Strassen comienza a tener tiempos de ejecución más razonables.

3. Existe una matriz de dimensión nxm en la que Strassen resulta mejor.

4. La utilización de memoria es proporcional a las dimensiones de la matriz.