Proyecto Final

## Paralelización de multiplicación de matrices

Arturo Alam Téllez Villagómez

A01205569

Tecnológico de Monterrey Campus Querétaro

A01205569@itesm.mx

# Resumen

El siguiente documento tiene como propósito presentar los resultados de la investigación final enfocada a la paralelización del algoritmo para la multiplicación de matrices en diferentes tecnologías cuyo fin es la división de tareas para mejorar la eficiencia de los programas, con el propósito de evaluar el desempeño de cada una para determinar cuál de ellas logra el mejor desempeño en la materia de multiprocesadores.

# Introducción

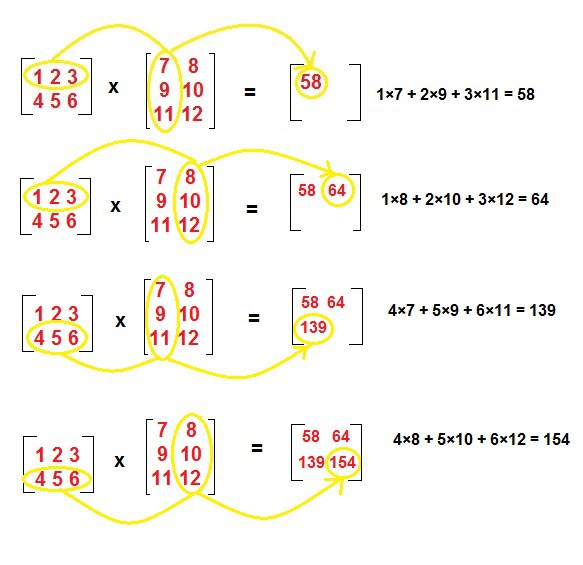
El trabajo de investigación fue elaborado de manera individual por el autor tomando como problema el algoritmo para la multiplicación de matrices de tamaño M x N por N x P, debido a la complejidad que este representa, existía la alternativa de realizar la multiplicación de matrices de tamaño N x N por N x N, sin embargo, esto no representaba un reto tal como el primero.

# Desarrollo

¿En qué consiste la multiplicación de matrices?

Primero es importante denotar que para que se pueda llevar a cabo un procedimiento de este tipo, existen restricciones, como es presentado en el párrafo anterior, el tipo de arreglos bidimensionales que serán utilizados para el experimento, serán de dimensiones diferentes, representadas por la restricción MxN y NxP que significa que la primera matriz deberá tener el mismo número de columnas, que el número de renglones de la matriz B y cuyo resultado será una matriz de tamaño MxP.

La definición de una multiplicación de matrices puede ser descrita como una multiplicación renglón-columna. Para una explicación gráfica, se puede apreciar la siguiente imagen:



Se puede apreciar que para cada posición i,j de la matriz resultante, es la sumatoria de los productos de los elementos del renglón i de la matriz A por los elementos en la columna j de la matriz B.

Partiendo de esta solución se busco implementar este procedimiento utilizando las siguientes tecnologías:

* *Threads* en Java.
* Fork/Join *framework* en Java.
* OpenMP en C.
* CUDA
* TBB

Para cada una de ellas, también se llevó a cabo el desarrollo del algoritmo de manera secuencial como punto de referencia para la comparación del desempeño en procesamiento.

Cabe destacar que para evaluar los resultados en las mismas condiciones se hizo uso del servidor proporcionado por el profesor de la materia de multiprocesadores Pedro Pérez, en donde se encontraban todas las librerías necesarias para compilar y ejecutar en cada una de las tecnologías, además de que se utilizaron matrices de tamaños 3000x700 y 700x2000 en la ejecución de todos los programas.

Threads en Java

En la elaboración del siguiente algoritmo, se realizó llevó a cabo la división de la matriz A en segmentos iguales dados por el numero de threads disponibles de acuerdo a las especificaciones del sistema, en el caso del server proporcionado se hizo la división entre 8 y cada uno procesaba una sección diferente de la matriz para llevar a cabo las operaciones necesarias.

Fork/Join *framework* en Java

La siguiente implementación realizó tareas similares a las de Threads en Java, sin embargo, la división de estas tareas fue llevada a cabo por la librería de Fork-Join Pool en que se estableció un threshold de 100 y al llegar un thread a un tamaño como el antes descrito, entonces puede llevar a cabo las operaciones necesarias.

OpenMP en C

OpenMP fue un poco más complicado al menos desde la perspectiva del autor ya que hay que tener en consideración qué variables serán compartidas por los threads y cuales serán reservadas, sin embargo se llevó a cabo la implementación siguiendo un proceso similar al realizado en el de Threads de Java

CUDA

Después de realizar una ardua investigación respecto a la implementación de este algoritmo, se hizo uso de las herramientas proporcionadas por la librería de CUDA, en que se puede hacer una subdivisión de las matrices utilizando los dimBlocks teniendo un tamaño de 16 x 16 que es el estándar y un dimGrid acorde a las dimensiones de la matriz resultante. En el procedimiento ejecutado en el device (el GPU) lo primero es capturar el threadID para saber qué localidad de la matriz es la que vamos a evaluar, a partir de las dimensiones proporcionadas con el dimBlock y el dimGrid, con ello podremos saber el renglón de la matriz A y la columna de la matriz B que se multiplicarán, también puede ocurrir que estos excedan las limitantes dadas por las dimensiones de las matrices y es por ello que existe una condición que se asegura que los threadID que se evaluarán cumplan con las restricciones, a continuación se efectúa la multiplicación iterando por cada elemento de la dimensión que tienen en comón (columnasA = renglonesB) y se guarda el resultado en la localidad correspondiente de la matriz resultado.

TBB

Como extra y parte del ejercicio de repasar todas las tecnologías vistas en clase, también se llevó a cabo la implementación del algoritmo en la tecnología TBB de Intel.

Los resultados se encuentran en el archivo de Excel adjunto.

# Conclusiones

Nos encontramos en una etapa tecnológica donde no basta saber programar, también hay que saber hacerlo en diferentes paradigmas y tecnologías, durante el curso se puede apreciar que existen numerosas tecnologías que nos permiten romper las barreras de lo secuencial para dar paso al procesamiento paralelo, algunas de ellas tienen propósitos específicos y se desempeñan mejor en algunas tareas que en otras es por ello que en algoritmos como el presentado en este artículo, hubo alguna implementación que no consiguió un speedup significativo, sin embargo para otro tipo de tareas podría incluso superar los de otras tecnologías y viceversa. Agradezco el contenido y la introducción a estas tecnologías que me ayudarán a desempeñarme de mejor manera durante mi vida profesional y destacar ante un ecosistema sumamente competido.

# Agradecimientos

Se agradece el compromiso y pasión proporcionada por el profesor Pedro Pérez en la impartición de la materia, al ingresar al curso sentía un tanto de incertidumbre por el contenido dado que es un tópico de suma importancia para un ingeniero en sistemas computacionales de hoy en día, como lo hemos visto, la ley de Moore deja de cumplirse y con ello debemos prepararnos para ser más eficientes con la tecnología con la que contamos, es por ello que se agradece tener un instructor con vasto conocimiento y la disposición de compartirlo.

# Referencias

# Apéndices

/\*

\* Arturo Alam Tellez Villagomez

\* A01205569

\* Matrix Multiplication

\* Multiprocessors

\* Final Project

\*/

#include <stdio.h>

#include "cuda\_runtime.h"

#include <stdlib.h>

#include <time.h>

#include "../utils/cheader.h"

#define BLOCK\_SIZE 16

\_\_global\_\_ void matMult(int \*a, int \*b, int \*c, int filasA, int columnasB, int k) {

// This somehow does the job

float temp = 0;

int i;

int row = blockIdx.y \* blockDim.y + threadIdx.y;

int col = blockIdx.x \* blockDim.x + threadIdx.x;

// If the tid is greater than the size of the result matrix, return

if(row > filasA || col > columnasB) return;

for (i = 0; i < k; ++i)

temp += (a[row \* k + i]) \* (b[i \* columnasB + col]);

c[row \* columnasB + col] = temp;

}

int main(){

// Declare variables

int \*a, \*b, \*c;

int \*dev\_a;

int \*dev\_b;

int \*dev\_c;

double ms;

// Columns and rows to use

int filasA;

int columnasA;

int filasB;

int columnasB;

// Get values for rows and columns

printf ("Rows of matrix A: ");

scanf ("%i",&filasA);

printf ("Columns of matrix A: ");

scanf ("%i",&columnasA);

filasB = columnasA;

printf ("Columns of matrix B: ");

scanf ("%i",&columnasB);

/\*

filasA = 5;

columnasA = 5;

filasB = columnasA;

columnasB = 5;

\*/

// Define sizes to store in both host and device

int sizeA = filasA \* columnasA \* sizeof(int);

int sizeB = filasB \* columnasB \* sizeof(int);

int sizeC = filasA \* columnasB \* sizeof(int);

// define matrixes

a = (int\*)malloc(sizeA);

b = (int\*)malloc(sizeB);

c = (int\*)malloc(sizeC);

// Fill matrix A

for(int i = 0; i < filasA; i++)

for(int j = 0; j < columnasA; j++)

a[i\*columnasA + j] = (int)((rand() % 100) + 1);

//Fill matrix B

for(int i = 0; i < filasB; i++)

for(int j = 0; j < columnasB; j++)

b[i\*columnasB + j] = (int)((rand() % 100) + 1);

// Allocate values in device

cudaMalloc((void\*\*) &dev\_a, sizeA);

cudaMalloc((void\*\*) &dev\_b, sizeB);

cudaMalloc((void\*\*) &dev\_c, sizeC);

// Copy the values from matrix A and B in host to the GPU

cudaMemcpy(dev\_a, a, sizeA, cudaMemcpyHostToDevice);

cudaMemcpy(dev\_b, b, sizeB, cudaMemcpyHostToDevice);

// Define the grid that is going to have onw thread per matrix location of the result

dim3 dimBlock(BLOCK\_SIZE, BLOCK\_SIZE);

dim3 dimGrid((columnasB + dimBlock.x - 1) / dimBlock.x, (filasA + dimBlock.y - 1) / dimBlock.y);

// Call function in GPU

ms = 0;

start\_timer();

matMult<<<dimGrid, dimBlock>>>(dev\_a, dev\_b, dev\_c, filasA, columnasB, columnasA);

ms = stop\_timer();

// Get the value from GPU to Host so we can print it

cudaMemcpy(c, dev\_c, sizeC, cudaMemcpyDeviceToHost);

//Print the three matrixes

// printf("Matrix A \n");

// for(int i = 0; i < filasA; i++){

// for(int j = 0; j < columnasA; j++){

// printf("%i ", a[i\*columnasA + j]);

// }

// printf("\n");

// }

// printf("\n");

// printf("Matrix B \n");

// for(int i = 0; i < filasB; i++){

// for(int j = 0; j < columnasB; j++){

// printf("%i ", b[i\*columnasB + j]);

// }

// printf("\n");

// }

// printf("\n");

// printf("Matrix C (Result) \n");

// for(int i = 0; i < filasA; i++){

// for(int j = 0; j < columnasB; j++){

// printf("%i \t", c[i\*columnasB + j]);

// }

// printf("\n");

// }

// printf("\n");

printf("Time = %lf\n", ms);

cudaFree(dev\_a);

cudaFree(dev\_b);

cudaFree(dev\_c);

free(a);

free(b);

free(c);

}

/\*

\* Arturo Alam Tellez Villagomez

\* A01205569

\* Matrix Multiplication

\* Multiprocessors

\* Final Project

\*/

import java.util.Random;

import java.util.Scanner;

class SeqMatMult

{

// Main

public static void main(String args[])

{

// Variable declaration

int filasA, columnasA, filasB, columnasB,i,j,k,sum;

long startTime, stopTime;

double acum = 0;

Scanner in = new Scanner(System.in);

// Ask for matrix dimensions

// Matrix A

System.out.print("Rows of matrix A: ");

filasA = in.nextInt();

// Matrix A and Bs K (similar value)

System.out.print("Columns of matrix A = rows matrix B: ");

columnasA = in.nextInt();

filasB = columnasA;

// Matrix B

System.out.print("Columns of matrix B: ");

columnasB = in.nextInt();

// Allocate space for matrices

int matA[][] = new int[filasA][columnasA];

int matB[][] = new int[filasB][columnasB];

int matC[][] = new int[filasA][columnasB];

// Fill with random values

Utils.fillMatrix(matA);

Utils.fillMatrix(matB);

// Start Timer

startTime = System.currentTimeMillis();

for ( i= 0 ; i < filasA ; i++ ){

for ( j= 0 ; j <columnasB;j++){

sum=0;

for ( k= 0 ; k <filasB;k++ ){

sum +=matA[i][k]\*matB[k][j] ;

}

matC[i][j]=sum;

}

}

// Stop Timer

stopTime = System.currentTimeMillis();

acum += (stopTime - startTime);

// Utils.displayMatrix(matA, 'A');

// Utils.displayMatrix(matB, 'B');

// Utils.displayMatrix(matC, 'C');

// Print result

System.out.printf("Time = %f\n", (acum/1000));

System.out.println();

}

}

/\*

\* Arturo Alam Tellez Villagomez

\* A01205569

\* Matrix Multiplication

\* Multiprocessors

\* Final Project

\*/

import java.util.concurrent.ForkJoinPool;

import java.util.Scanner;

class ForkJoinMatMult {

// Get maximum number of available threads

    private static final int MAXTHREADS = Runtime.getRuntime().availableProcessors();

    public static void main(String args[]) {

        ForkJoinPool pool;

        long startTime, stopTime;

        double acum = 0;

        int filasA, columnasA, filasB, columnasB,i,j,k,sum, size;

Scanner in = new Scanner(System.in);

// Ask for matrix dimensions

// Matrix A

System.out.print("Rows of matrix A: ");

filasA = in.nextInt();

// Matrix A and Bs K (similar value)

System.out.print("Columns of matrix A = rows matrix B: ");

columnasA = in.nextInt();

filasB = columnasA;

// Matrix B

System.out.print("Columns of matrix B: ");

columnasB = in.nextInt();

// Allocate space for matrices

int matA[][] = new int[filasA][columnasA];

int matB[][] = new int[filasB][columnasB];

int matC[][] = new int[filasA][columnasB];

size = filasA;

// Fill with random values

Utils.fillMatrix(matA);

Utils.fillMatrix(matB);

        acum = 0;

startTime = System.currentTimeMillis();

pool = new ForkJoinPool(MAXTHREADS);

pool.invoke(new MultiplicatorFJ(matA, matB, matC, 0, matA.length));

stopTime = System.currentTimeMillis();

acum += (stopTime - startTime);

        // Utils.displayMatrix(matA, 'A');

// Utils.displayMatrix(matB, 'B');

// Utils.displayMatrix(matC, 'C');

        System.out.printf("Time = %.5f\n", (acum / 1000));

    }

}

/\*

\* Arturo Alam Tellez Villagomez

\* A01205569

\* Matrix Multiplication

\* Multiprocessors

\* Final Project

\*/

import java.util.Random;

import java.util.Scanner;

class ThreadMatMult

{

private static final int MAXTHREADS = Runtime.getRuntime().availableProcessors();

// Main

public static void main(String args[])

{

// Variable declaration

int filasA, columnasA, filasB, columnasB,i,j,k,sum;

int block;

long startTime, stopTime;

double acum = 0;

Multiplicator[] threads;

Scanner in = new Scanner(System.in);

System.out.println(MAXTHREADS);

// Ask for matrix dimensions

// Matrix A

System.out.print("Rows of matrix A: ");

filasA = in.nextInt();

// Matrix A and Bs K (similar value)

System.out.print("Columns of matrix A = rows matrix B: ");

columnasA = in.nextInt();

filasB = columnasA;

// Matrix B

System.out.print("Columns of matrix B: ");

columnasB = in.nextInt();

// Allocate space for matrices

int matA[][] = new int[filasA][columnasA];

int matB[][] = new int[filasB][columnasB];

int matC[][] = new int[filasA][columnasB];

threads = new Multiplicator[MAXTHREADS];

block = matA.length / MAXTHREADS;

// Fill with random values

Utils.fillMatrix(matA);

Utils.fillMatrix(matB);

// Create threads that go through the matrix

for (i = 0; i < threads.length; i++) {

if (i != threads.length - 1) {

threads[i] = new Multiplicator(matA, matB, matC, (i \* block), ((i + 1) \* block));

} else {

threads[i] = new Multiplicator(matA, matB, matC, (i \* block), matA.length);

}

}

// Start Timer

startTime = System.currentTimeMillis();

// Run threads

for (i = 0; i < threads.length; i++) {

threads[i].start();

}

for (i = 0; i < threads.length; i++) {

try {

threads[i].join();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

// Stop Timer

stopTime = System.currentTimeMillis();

acum += (stopTime - startTime);

// Utils.displayMatrix(matA, 'A');

// Utils.displayMatrix(matB, 'B');

// Utils.displayMatrix(matC, 'C');

// Print result

System.out.printf("Time = %f\n", acum/1000);

System.out.println();

}

}

/\*

\* Arturo Alam Tellez Villagomez

\* A01205569

\* Matrix Multiplication

\* Multiprocessors

\* Final Project

\*/

#include <stdio.h>

#include <stdlib.h>

#include "../utils/cheader.h"

void matrix\_multiplication(int \*\*a, int \*\*b, int \*\*c, int filasA, int columnasB, int n)

{

int i, j, k, sum = 0;

#pragma parallel for shared(a, b, c, filasA) private(j, k, sum)

for (i = 0; i < filasA; i++){

for (j = 0; j < columnasB; j++){

sum = 0;

for (k = 0; k < n; k++)

{

sum += a[i][k] \* b[k][j];

}

c[i][j] = sum;

}

}

}

int main(int argc, char \*argv[])

{

int i, j, k, sum;

int filasA;

int columnasA;

int filasB;

int columnasB;

int \*\*a, \*\*b, \*\*c;

double ms;

srand(time(0));

printf("Rows of matrix A: ");

scanf("%i", &filasA);

printf("Columns of matrix A: ");

scanf("%i", &filasB);

columnasA = filasB;

printf("Columns of matrix B: ");

scanf("%i", &columnasB);

// printf("%i %i %i", filasA, columnasA, columnasB);

// Allocate matrix A

a = (int \*\*)malloc(filasA \* sizeof(int \*));

for (i = 0; i < filasA; i++)

a[i] = (int \*)malloc(columnasA \* sizeof(int));

// Allocate matrix B

b = (int \*\*)malloc(filasB \* sizeof(int \*));

i = 0;

for (i = 0; i < filasB; i++)

b[i] = (int \*)malloc(columnasB \* sizeof(int));

// Allocate matrix C

c = (int \*\*)malloc(filasA \* sizeof(int \*));

i = 0;

for (i = 0; i < filasA; i++)

c[i] = (int \*)malloc(columnasB \* sizeof(int));

// Fill A with random values

for (i = 0; i < filasA; i++)

for (j = 0; j < columnasA; j++)

a[i][j] = (rand() % 100) + 1;

// Display A

int m, n;

// printf("\nMat A\n");

// for (m = 0; m < filasA; m++)

// {

// for (n = 0; n < columnasA; n++)

// {

// printf("%i\t", a[m][n]);

// }

// printf("\n");

// }

for (i = 0; i < filasB; i++)

for (j = 0; j < columnasB; j++)

b[i][j] = (rand() % 100) + 1;

// printf("\nMat B\n");

// for (m = 0; m < filasB; m++)

// {

// for (n = 0; n < columnasB; n++)

// {

// printf("%i\t", b[m][n]);

// }

// printf("\n");

// }

// ms = 0;

start\_timer();

matrix\_multiplication(a, b, c, filasA, columnasB, columnasA);

ms = stop\_timer();

// printf("\nMat Res\n");

// for (m = 0; m < filasA; m++) {

// for (n = 0; n < columnasB; n++){

// printf("%i\t", c[m][n]);

// }

// printf("\n");

// }

printf("\nTime = %lf\n", ms);

free(a);

free(b);

free(c);

return 0;

}

/\*

\* Arturo Alam Tellez Villagomez

\* A01205569

\* Matrix Multiplication

\* Multiprocessors

\* Final Project

\*/

#include <stdio.h>

#include <stdlib.h>

#include "../utils/cheader.h"

int main(int argc, char const \*argv[])

{

int i, j, k, sum;

int filasA;

int columnasA;

int filasB;

int columnasB;

int \*\*a, \*\*b, \*\*c;

double ms;

srand(time(0));

printf("Rows of matrix A: ");

scanf("%i", &filasA);

printf("Columns of matrix A: ");

scanf("%i", &filasB);

columnasA = filasB;

printf("Columns of matrix B: ");

scanf("%i", &columnasB);

// printf("%i %i %i", filasA, columnasA, columnasB);

// Allocate matrix A

a = (int \*\*)malloc(filasA \* sizeof(int \*));

for (i = 0; i < filasA; i++)

a[i] = (int \*)malloc(columnasA \* sizeof(int));

// Allocate matrix B

b = (int \*\*)malloc(filasB \* sizeof(int \*));

i = 0;

for (i = 0; i < filasB; i++)

b[i] = (int \*)malloc(columnasB \* sizeof(int));

// Allocate matrix C

c = (int \*\*)malloc(filasA \* sizeof(int \*));

i = 0;

for (i = 0; i < filasA; i++)

c[i] = (int \*)malloc(columnasB \* sizeof(int));

// Fill A with random values

for (i = 0; i < filasA; i++)

for (j = 0; j < columnasA; j++)

a[i][j] = (rand() % 100) + 1;

// Display A

int m, n;

// printf("\nMat A\n");

// for (m = 0; m < filasA; m++) {

// for (n = 0; n < columnasA; n++){

// printf("%i\t", a[m][n]);

// }

// printf("\n");

// }

for (i = 0; i < filasB; i++)

for (j = 0; j < columnasB; j++)

b[i][j] = (rand() % 100) + 1;

// printf("\nMat B\n");

// for (m = 0; m < filasB; m++) {

// for (n = 0; n < columnasB; n++){

// printf("%i\t", b[m][n]);

// }

// printf("\n");

// }

ms = 0;

sum = 0;

start\_timer();

for (i = 0; i < filasA; i++)

{

for (j = 0; j < columnasB; j++)

{

for (k = 0; k < filasB; k++)

{

sum += a[i][k] \* b[k][j];

}

c[i][j] = sum;

sum = 0;

}

}

ms = stop\_timer();

// printf("\nMat Res\n");

// for (m = 0; m < filasA; m++) {

// for (n = 0; n < columnasB; n++){

// printf("%i\t", c[m][n]);

// }

// printf("\n");

// }

printf("\nTime = %lf\n", ms/1000);

free(a);

free(b);

free(c);

return 0;

}