Tarea - Multiplicacion de matrices

Alumno: Ángel Alonso Galarza Chávez
Profesor: Dr. Cuauhtemoc Mancillas López
Curso: Programación Avanzada

Multiplicacion de matrices en C y Python

La multiplicación de matrices es una operación fundamental en álgebra lineal y se utiliza en una amplia variedad de campos, desde ciencias de la computación hasta física. En términos simples, es una forma de combinar dos o más matrices para obtener una nueva matriz.

Los intrínsecos AVX son funciones proporcionadas por los compiladores que permiten a los programadores acceder directamente a instrucciones específicas del procesador. Estas instrucciones, parte de las Extensiones Vectoriales Avanzadas (AVX, por sus siglas en inglés), están diseñadas para acelerar el procesamiento de datos al realizar múltiples operaciones simultáneamente en múltiples elementos de datos.

Los compiladores ofrecen una variedad de intrínsecos AVX que cubren operaciones aritméticas, lógicas, de comparación, movimiento de datos y vectorización. Al utilizar estos intrínsecos, puedes escribir código altamente optimizado que aprovecha al máximo las capacidades de hardware de tu procesador.

El código presentado utiliza la biblioteca Intel Intrinsics, específicamente las instrucciones AVX, para llevar a cabo una multiplicación de matrices a nivel de vector. Se emplean tipos de datos de 256 bits (__m256) para cargar simultáneamente ocho valores de punto flotante de simple precisión desde la memoria a los registros SIMD. Estos registros son procesados en paralelo dentro de tres bucles anidados, donde cada iteración realiza la multiplicación punto a punto de vectores correspondientes y acumula los resultados en un registro acumulador. Finalmente, los valores acumulados se almacenan en la matriz resultante.

```c
#include <immintrin.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
  float diff;
  const int row_A = 100;
  const int col_A = 500;
  const int col_B = 500;
  const int num_trails = 10;

  float A[row_A][col_A];
  float B[col_A][col_B];
  float C[row_A][col_B];
  float scratchpad[8];
  clock_t t1, t2;

  for (int i = 0; i < row_A; i++) {
    for (int j = 0; j < col_A; j++) {
      A[i][j] = rand() % 10;
    }
  }
  for (int i = 0; i < col_A; i++) {
    for (int j = 0; j < col_B; j++) {
      B[i][j] = rand() % 10;
    }
  }

  __m256 ymm0, ymm1, ymm2;

  t1 = clock();
  const int col_A_reduced = col_A - col_A % 8;
  for (int r = 0; r < num_trails; r++) {
    for (int i = 0; i < row_A; i++) {
      for (int j = 0; j < col_B; j++) {
        float res = 0;
        for (int k = 0; k < col_A_reduced; k += 8) {
```

```
          // Cargar 8 elementos de A y B usando AVX
          ymm0 = _mm256_loadu_ps(&A[i][k]);
          ymm1 = _mm256_loadu_ps(&B[k][j]);

          // Multiplicar los valores cargados
          ymm2 = _mm256_mul_ps(ymm0, ymm1);

          // Almacenar el resultado en un scratchpad
          _mm256_storeu_ps(scratchpad, ymm2);

          // Sumar los valores del scratchpad
          for (int x = 0; x < 8; x++) {
            res += scratchpad[x];
          }
        }
        // Procesar los elementos restantes de forma escalar
        for (int k = col_A_reduced; k < col_A; k++) {
          res += A[i][k] * B[k][j];
        }
        C[i][j] = res;
      }
    }
  }
  t2 = clock();
  diff = (((float)t2 - (float)t1) / CLOCKS_PER_SEC) * 1000;
  printf("Tiempo de ejecucion con intrinsics AVX: %f ms\n", diff);

  return 0;
}
```

A continuación se presenta una implementación en C de la multiplicación de matrices, utilizando las mismas dimensiones que en el código anterior. Sin embargo, en esta versión no se emplea la biblioteca Intrinsics, recurriendo a un enfoque más tradicional.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void multiplicarMatrices(int filasA, int columnasA, int filasB, int columnasB,
                         int matrizA[filasA][columnasA],
                         int matrizB[filasB][columnasB],
                         int resultado[filasA][columnasB]) {

  if (columnasA != filasB) {
    printf("No se pueden multiplicar las matrices.\n");
    return;
  }

  for (int i = 0; i < filasA; i++) {
    for (int j = 0; j < columnasB; j++) {
      resultado[i][j] = 0;
      for (int k = 0; k < columnasA; k++) {
        resultado[i][j] += matrizA[i][k] * matrizB[k][j];
      }
    }
  }
}

int main() {

  int filasA = 100;
  int columnasA = 500;
  int filasB = 500;
  int columnasB = 100;

  // Verifica si las matrices se pueden multiplicar antes de declararlas
  if (columnasA != filasB) {
    printf("No se pueden multiplicar las matrices. El n mero de columnas de la "
           "primera matriz debe ser igual al n mero de filas de la segunda "
           "matriz.\n");
    return 1;
  }

```

```
  int matrizA[filasA][columnasA], matrizB[filasB][columnasB],
      resultado[filasA][columnasB];

  srand(time(NULL));
  for (int i = 0; i < filasA; i++) {
    for (int j = 0; j < columnasA; j++) {
      matrizA[i][j] = rand() % 10;
    }
  }

  for (int i = 0; i < filasB; i++) {
    for (int j = 0; j < columnasB; j++) {
      matrizB[i][j] = rand() % 10;
    }
  }

  clock_t inicio = clock();
  multiplicarMatrices(filasA, columnasA, filasB, columnasB, matrizA, matrizB,
                      resultado);

  clock_t fin = clock();

  float tiempo_total = (((float)fin - (float)inicio) / CLOCKS_PER_SEC) * 1000;
  printf("Tiempo de ejecuci n: %f segundos\n", tiempo_total);

  return 0;
}
```

Por ultimo, se muestra el código con la implementación en pichona de la multiplicación de las matrices.

```
    import numpy as np
import time

def multiplicar_matrices(A, B):
    filas_A, columnas_A = A.shape
    filas_B, columnas_B = B.shape

    if columnas_A != filas_B:
        raise ValueError("Las matrices no se pueden multiplicar")

    C = np.zeros((filas_A, columnas_B))
    for i in range(filas_A):
        for j in range(columnas_B):
            for k in range(columnas_A):
                C[i][j] += A[i][k] * B[k][j]

    return C


start_time = time.time()
A = np.random.rand(100, 500)
B = np.random.rand(500, 100)

Resultado = multiplicar_matrices(A, B)
end_time = time.time()
print('tiempo: ', str(end_time - start_time))
```

Ejecución

Los tiempos de ejecución de las implementaciones en C y Python se presentan en la figura siguiente.

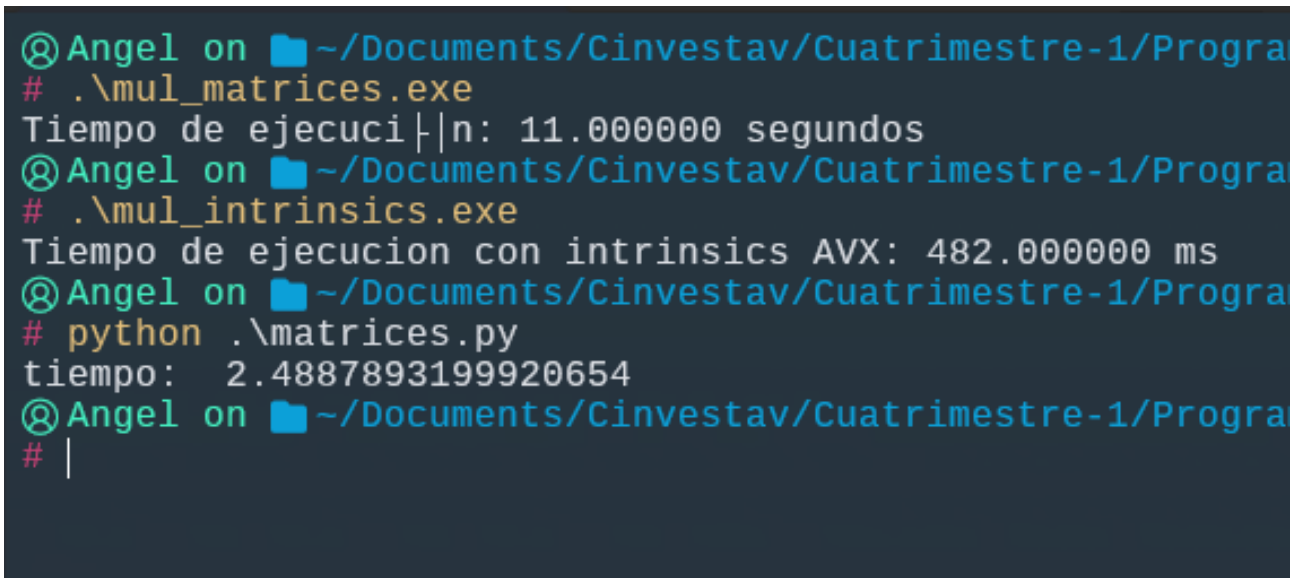

Figure 1: Tiempos de ejecucion de los codigos

Los tiempos de ejecución obtenidos revelan que la implementación con Intrinsics supera significativamente a las demás, con un tiempo de 482 milisegundos. La implementación en C sin Intrinsics es la más lenta, requiriendo 11 segundos para completar la tarea. Sorprendentemente, Python obtiene un tiempo de ejecución intermedio de 2.48 segundos, lo que sugiere que la optimización de la biblioteca estándar de Python para ciertas operaciones numéricas puede compensar en parte la diferencia de rendimiento con respecto a C.