Práctica Nro. 1
Optimización de algoritmos secuenciales

Hardware usado para compilación:

- AMD Ryzen 5 5500U
- 8 Gb RAM DDR4
- M.2 256 GB
- OS: Fedora 41
- Versión del gcc: 14.2.1
- Mientras se carga la laptop

1. El algoritmo fib.c resuelve la serie de Fibonacci, para un número N dado, utilizando dos métodos: recursivo e iterativo. Analice los tiempos de ejecución de ambos métodos ¿Cuál es más rápido? ¿Por qué?
Nota: ejecute con N=1 ... 50. N=25

- Tiempo con fibonacci iterativo: 0.0000001907
  Este algoritmo tiene **complejidad lineal** O(N), ya que solo usa un bucle que itera hasta N.
- Tiempo fibonacci recursivo: 0.0007534027
  Este tiene **complejidad exponencial** $O(2^N)$, ya que cada llamada recursiva genera dos nuevas llamadas. A medida que N crece, el número de operaciones aumenta rápidamente.

2. El algoritmo funcion.c resuelve, para un x dado, la siguiente sumatoria:

$$\sum_{i=0}^{100.000.000} 2^* \frac{x^3+3x^2+3x+2}{x^2+1}\text{-i}$$

El algoritmo compara dos alternativas de solución. ¿Cuál de las dos formas es más rápida? ¿Por qué?
Primera prueba:

- Primera alternativa:
  1. Total: 0.5377
  2. Promedio: 0.0000000054
- Segunda alternativa (calculada cada vez):
  1. Total: 0.5360
  2. Promedio: 0.0000000054

Segunda prueba:

- Primera alternativa:
  1. Total: 0.2737
  2. Promedio: 0.0000000027
- Segunda alternativa (calculada cada vez):
  1. Total: 0.2739
  2. Promedio: 0.0000000027

Al ejecutar ambas soluciones las primeras veces nos dio como resultado que la primera solución se ejecutó en menor tiempo que la segunda. Esto es un tanto curioso ya que al ver ambos algoritmos se evidencia que el primero calcula en cada iteración el resultado de la

cuenta, mientras que el segundo guarda el resultado en una variable y simplemente la utiliza. La primera función es entonces más rápida porque evita recalcular la misma expresión en cada iteración, reduciendo el número de operaciones costosas.

3. Investigue en la documentación del compilador o a través de Internet qué opciones de optimización ofrece el compilador gcc (flag O). Compile y ejecute el algoritmo matrices.c, el cual resuelve una multiplicación de matrices de NxN. Explore los diferentes niveles de optimización para distintos tamaños de matrices. ¿Qué optimizaciones aplica el compilador? ¿Cuál es la ganancia respecto a la versión sin optimización del compilador? ¿Cuál es la ganancia entre los distintos niveles?

La flag O (-O) para el compilador significa control de optimización. Hay varias opciones, del 0 al 3 y luego S y fast, cada una aplica distintas técnicas para mejorar el rendimiento del código.

`O0` (Sin optimización, por defecto)

- No aplica optimizaciones.
- Facilita la depuración porque conserva la estructura original del código.
- Pruebas con matrices de:
    1. 512x512: 1.60 s
    2. 1024x1024: 39.24 s
    3. 2048x2048: 497.06 s

`O1` (Optimización básica)

- Eliminar código redundante.
- Optimiza el acceso a variables en registros.
- Puede reducir el tamaño del binario.
- Pruebas con matrices de:
    1. 512 x 512: 0.78 s
    2. 1024 x 1024: 7.58 s
    3. 2048 x 2048: 115.57 s

`O2` (Optimización agresiva sin afectar la estabilidad)

- Todas las optimizaciones de `-O1`.
- Eliminación más agresiva de código muerto.
- Mejor manejo de bucles (desenrollado y vectorización básica).
- Reordenamiento de instrucciones para mejorar el uso del procesador.
- Pruebas con matrices de:
    1. 512 x 512: 0.76 s
    2. 1024 x 1024: 8.49 s
    3. 2048 x 2048: 113.31 s

`O3` (Optimización máxima para velocidad)

- Todas las optimizaciones de `-O2`.
- Expansión agresiva de bucles y funciones en línea.
- Mayor vectorización de código.
- Puede aumentar el tamaño del binario.
- Pruebas con matrices de:
    1. 512 x 512: 0.795 s
    2. 1024 x 1024: 7.93 s
    3. 2048 x 2048: 107.65 s

`Ofast` (Máxima optimización posible, sin respetar el estándar IEEE)

- Incluye `-O3` + optimizaciones específicas que pueden alterar la precisión de los cálculos de punto flotante.
- Desactiva comprobaciones de redondeo y orden de operaciones.
- Pruebas con matrices de:
    1. 512 x 512: 0.796 s
    2. 1024 x 1024: 7.97 s
    3. 2048 x 2048: 258:48 s

`Os` (Optimización para reducir tamaño del ejecutable)

- Similar a `-O2`, pero evita optimizaciones que aumenten el tamaño del binario
- Pruebas con matrices de:
    1. 512 x 512: 0.78 s
    2. 1024 x 1024: 7.52 s
    3. 2048 x 2048: 142.72 s

4. Dada la ecuación cuadrática: $x^2 - 4.0000000\ x + 3.9999999 = 0$, sus raíces son $r_1 = 2.000316228$ y $r_2 = 1.999683772$ (empleando 10 dígitos para la parte decimal).

- Optimización usada -O3

a. El algoritmo quadratic1.c computa las raíces de esta ecuación empleando los tipos de datos float y double. Compile y ejecute el código. ¿Qué diferencia nota en el resultado?
Soluciones Float: 2.00000 2.00000
Soluciones Double: 2.00032 1.99968

Las soluciones en doubles deberían ser más cercanas a los resultados reales ya que tiene más bits para precisión.

b. El algoritmo quadratic2.c computa las raíces de esta ecuación, pero en forma repetida. Compile y ejecute el código variando la constante TIMES. ¿Qué diferencia nota en la ejecución?

- Times = 100
    1. Tiempo requerido solución Double: 0.9251 s
    2. Tiempo requerido solución Float: 1.1300 s
- Times = 200
    1. Tiempo requerido solución Double: 1.8603 s
    2. Tiempo requerido solución Float: 2.2499 s
- Times = 300
    1. Tiempo requerido solución Double: 2.7615 s
    2. Tiempo requerido solución Float: 3.3757 s

A medida que aumenta la cantidad de iteraciones la diferencia entre los tiempo de la solución de Double y Float se va alargando exponencialmente

c. El algoritmo quadratic3.c computa las raíces de esta ecuación, pero en forma repetida. Compile y ejecute el código variando la constante TIMES. ¿Qué diferencia nota en la ejecución? ¿Qué diferencias puede observar en el código con respecto a quadratic2.c? Nota: agregue el flag -lm al momento de compilar. Pruebe con el nivel de optimización que mejor resultado le haya dado en el ejercicio anterior.

- Times = 100
    1. Tiempo requerido solución Double: 0.9419 s
    2. Tiempo requerido solución Float: 0.6561 s
- Times = 200
    1. TIempo requerido solución Double: 1.8565 s
    2. Tiempo requerido solución Float: 1.3046 s
- Times = 300
    1. Tiempo requerido solución Double: 2.8471 s
    2. Tiempo requerido solución Float: 1.9643 s

Con respecto a la anterior prueba, en esta el que se aleja es el Float siendo más rápido que el cómputo de Double, esto se debe a que al especificar el tipo de dato numérico el compilador no debe hacer ninguna conversión de Double a Float, por ende tarda menos

5. Analice el algoritmo matrices.c. ¿Dónde cree que se producen demoras? ¿Cómo podría optimizarse el código? Al menos, considere los siguientes aspectos:
o Explotación de localidad de datos a través de reorganización interna de matrices A, B o C (según corresponda).
Las matrices se recorren por filas, pero la memoria de las matrices en C es almacenada en formato por columnas (column-major order) debido a cómo se manejan los arreglos multidimensionales.
Reorganizar los bucles de acceso a las matrices para que primero se accedan las columnas de las matrices A y B (lo que está en memoria contiguo) en lugar de filas.

Reemplazar el orden de los bucles de la multiplicación de matrices para recorrer las matrices de manera más cache-friendly.

o El uso de Setters y getters es una buena práctica en la programación orientada a objetos. ¿Tiene sentido usarlos en este caso? ¿Cuál es su impacto en el rendimiento?
En general se usan getters y setters para mantener abstracción y encapsulación. aquí no tienen mucho sentido. No obstante, el setter realiza un if que habría que hacer igualmente. En términos de rendimiento, la mejora puede ser mínima si no se usan ya que implican un llamado extra a una función que habría que ir a buscar en memoria para luego finalmente acceder al dato.

o ¿Hay expresiones en el cómputo que pueden refactorizarse para no ser computadas en forma repetida?
En primer lugar tanto lo de setValor como lo de getValor podrían ahorrarse si hiciéramos que el orden sea siempre por filas o por columnas.
Por otro lado, al momento de multiplicar las matrices se va cambiando innecesariamente el valor de c en cada iteración cuando es más sencillo simplemente hacer la cuenta sobre una variable temporal y luego setearle el valor.

o En lugar de ir acumulando directamente sobre la posición C[i,j] de la matriz resultado (línea 72), pruebe usar una variable local individual y al finalizar el bucle más interno, asigne su valor a C[i,j]. ¿Esta modificación impacta en el rendimiento? ¿Por qué?

Combine las mejoras que haya encontrado para obtener una solución optimizada y compare los tiempos con la solución original para diferentes tamaños de matrices.
Antes de la optimización:

Multiplicación de matrices de 1024x1024. Tiempo en segundos 27.757642

Luego de la optimización:

Multiplicación de matrices de 1024x1024. Tiempo en segundos 3.385127

6. Analice y describa brevemente cómo funciona el algoritmo mmblk.c que resuelve la multiplicación de matrices cuadradas de NxN utilizando una técnica de multiplicación por bloques. Luego, ejecute el algoritmo utilizando distintos tamaños de matrices y distintos tamaños de bloque (pruebe con valores que sean potencia de 2; p.e. N={512,1024,2048} y TB={16,32,64,128}). Finalmente, compare los tiempos con respecto a la multiplicación de matrices optimizada del ejercicio anterior. Según el tamaño de las matrices y del bloque elegido, responda: ¿Cuál es más rápido? ¿Por qué? ¿Cuál sería el tamaño de bloque óptimo para un determinado tamaño de matriz? ¿De qué depende el tamaño de bloque óptimo para un sistema?

La multiplicación de matrices en lugar de hacerse de manera tradicional (en un solo bucle triple) se divide en bloques más pequeños.
Se elige un tamaño de bloque bs, y el algoritmo divide las matrices A, B y C en bloques de tamaño bs \times bs.
Multiplicación por bloques:
El ciclo principal de la multiplicación se realiza sobre bloques de la siguiente manera:
Se recorren las matrices en pasos de tamaño bs.

Para cada bloque de matrices A y B, se realiza la multiplicación de submatrices en la función blkmul(), y se acumulan los resultados en la submatriz correspondiente de la matriz C.
El cálculo de la multiplicación de matrices en bloques se realiza en la función blkmul(), que toma bloques de A, B y los multiplica para actualizar el bloque correspondiente de C.

MMBLK-SEC;1024;16;4.477357;0.479632
MMBLK-SEC;1024;32;4.477258;0.479643
MMBLK-SEC;1024;64;4.554997;0.471457

MMBLK-SEC;2048;16;36.238972;0.474072
MMBLK-SEC;2048;32;35.129910;0.489038
MMBLK-SEC;2048;64;36.655484;0.468685

MMBLK-SEC;512;16;0.558915;0.480279
MMBLK-SEC;512;32;0.577662;0.464693
MMBLK-SEC;512;64;0.747403;0.359158

7. Analice el algoritmo triangular.c que resuelve la multiplicación de una matriz cuadrada por una matriz triangular inferior, ambas de NxN. ¿Cómo se podría optimizar el código? ¿Se pueden evitar operaciones? ¿Se puede reducir la memoria reservada? Implemente una solución optimizada y compare los tiempos probando con diferentes tamaños de matrices.
Memoria reservada:

- La matriz T sigue reservando espacio para toda la matriz N×N \times N×N, pero solo la parte triangular inferior es utilizada durante la multiplicación.
- La matriz C también está completamente reservada porque el resultado de la multiplicación es una matriz de la misma dimensión.

Evitar operaciones innecesarias:

- En el ciclo interior (el que recorre k), solo se multiplica hasta el índice j porque los elementos de la matriz triangular superior de T son cero. Esto reduce la cantidad de multiplicaciones y, por lo tanto, mejora el rendimiento.

Tiempo de ejecución:

- La optimización de la multiplicación de matrices triangulars elimina operaciones de multiplicación redundantes, lo que mejora el rendimiento de manera significativa, especialmente cuando N es grande.

Tiempo en segundos: 2.324387 (optimizada)
Tiempo en segundos 4.864949 (sin optimizar) c: