UNIVERSIDAD NACIONAL DE MOQUEGUA

Facultad de Ingeniería y Arquitectura

Escuela Profesional de Ingeniería de Sistemas e Informática

Paralelizacion de pixeles

Informe laboratorio N° 4

Estudiante:
Mayerling Pancca Ccalla

Profesores:
Honorio Apaza Alanoca

10 de julio de 2025

Índice

1. Introducción

1.1. Motivación y Contexto

En la actualidad, el procesamiento de imágenes digitales es una de las áreas más relevantes dentro de la computación, especialmente en aplicaciones como la visión por computadora, el reconocimiento de objetos y la animación. Las imágenes se representan comúnmente mediante matrices, donde cada elemento de la matriz (llamado píxel) contiene valores de color, comúnmente en formato RGB (rojo, verde, azul). Manipular estas matrices de manera eficiente es crucial para realizar tareas de procesamiento de imágenes como el filtro, la transformación y el movimiento de objetos dentro de una imagen.
Una de las técnicas más utilizadas en el procesamiento de imágenes es el uso de **kernels**. Un kernel es una pequeña matriz que se mueve sobre la imagen original, aplicando una operación matemática a los píxeles de la imagen para generar una nueva imagen. Este proceso se conoce como *convolución* y es esencial para realizar transformaciones como el desenfoque, la detección de bordes o, como en este caso, el movimiento de un objeto dentro de una imagen.
El uso de **memoria compartida** y **hilos** mejora considerablemente el rendimiento del procesamiento de imágenes, ya que permite la ejecución paralela de múltiples operaciones en diferentes partes de la imagen.

1.2. Objetivo general

El objetivo de este informe es explorar el uso de **kernels** en el procesamiento de imágenes, específicamente para mover imágenes de objetos (en este caso, un perro) dentro de un fondo, utilizando matrices en Python. Se busca aprender cómo los kernels manipulan las matrices de píxeles para simular el movimiento y cómo el uso de hilos y memoria compartida optimiza el rendimiento de este tipo de operaciones.

1.3. Objetivos específicos

A continuación, se presentan los objetivos específicos que guiarán este informe:

- **Comprender cómo los kernels se aplican a las matrices de píxeles para realizar operaciones sobre imágenes.** Los kernels se utilizan para realizar diversas transformaciones en las imágenes, como la detección de bordes, la mejora del contraste, o el desplazamiento de objetos.
- **Implementar el movimiento de una imagen usando kernels y matrices en Python.** Aplicar un kernel sobre una matriz de píxeles para mover la imagen de un objeto, simulando su desplazamiento a lo largo de un fondo.

- **Optimizar el proceso de manipulación de imágenes utilizando hilos en Python.** Utilizar la librería `threading` para ejecutar el procesamiento de la imagen en paralelo, mejorando la eficiencia del programa.

1.4. Justificación

El uso de kernels en el procesamiento de imágenes es fundamental para realizar transformaciones complejas de manera eficiente. Al tratarse de matrices pequeñas que operan sobre matrices más grandes, los kernels permiten manipular los píxeles de una imagen sin necesidad de modificar toda la matriz en cada iteración, lo que mejora la eficiencia computacional.
En este proyecto, se utiliza un kernel para simular el movimiento de un objeto (el perro) sobre un fondo. Este proceso se logra desplazando los píxeles de la imagen a lo largo de la matriz del fondo, creando la ilusión de movimiento.
El uso de **hilos** y **memoria compartida** optimiza este proceso al permitir que las operaciones de movimiento se realicen de manera concurrente, distribuyendo la carga de trabajo entre múltiples hilos. De esta forma, el programa puede procesar varias partes de la imagen al mismo tiempo, mejorando la velocidad de ejecución.
Este estudio no solo proporciona una comprensión más profunda de los principios del paralelismo y la programación concurrente, sino que también permite entender cómo manipular y transformar imágenes de manera eficiente utilizando matrices y kernels.

1.5. Explicación de la Implementación

En este proyecto, se utilizó la librería `matplotlib` de Python para realizar la visualización de la imagen y simular el movimiento de un perro sobre un fondo utilizando **kernels** y **matrices**.

- **Representación de la Imagen como Matriz:** Cada imagen, tanto el fondo como el perro, se representa como una matriz de píxeles. Cada píxel tiene tres valores que corresponden a los colores en el espacio RGB. Por ejemplo, un píxel podría tener el valor `[255, 0, 0]`, lo que indica que el píxel es de color rojo.

- **Uso de Kernels:** El kernel utilizado en este proyecto es una representación de la imagen del perro en una pequeña matriz. El kernel se mueve sobre la matriz del fondo, desplazando los píxeles de la imagen del perro a lo largo del fondo en función de los valores de los índices del kernel y la cantidad de pasos definidos en el programa.

- **Simulación del Movimiento:** El movimiento se simula desplazando cada píxel de la imagen del perro en la matriz del fondo. A cada paso, los píxeles del perro se colocan en una nueva posición dentro de la matriz, creando la ilusión de que el perro se está moviendo. Este proceso se repite en ciclos, cada uno representando un pequeño paso en el desplazamiento del perro.

- **Paralelización con Hilos:** La librería `threading` de Python se utiliza para paralelizar el procesamiento de la imagen. Al dividir la imagen en varias partes y procesarlas simultáneamente, se mejora la eficiencia del programa y se acelera el tiempo de ejecución.

2. Marco teórico

2.1. Antecedentes

La simulación del movimiento de objetos es crucial en áreas como la computación científica, la ingeniería y los sistemas interactivos. Tradicionalmente, se han utilizado métodos como las ecuaciones de Newton y técnicas modernas que aplican procesamiento de imágenes y algoritmos paralelos. En este contexto, los kernels son fundamentales para modelar el movimiento de objetos, ya que permiten modificar píxeles mediante operaciones de convolución. Además, el uso de algoritmos paralelos y técnicas como la paralelización con hilos y GPU han acelerado significativamente este tipo de simulaciones.

2.2. Marco conceptual

Este proyecto se basa en varios conceptos clave que permiten simular el movimiento de objetos con kernels y procesamiento paralelo:

2.2.1. Simulación de Movimiento

La simulación de movimiento busca representar el desplazamiento de un objeto en un espacio. En computación, esto se logra moviendo objetos a través de matrices de datos. Estas simulaciones son útiles en animación, robótica y física computacional.

2.2.2. Kernels

Un kernel es una matriz pequeña que se aplica sobre los datos para modificar sus valores. En simulación de movimiento, se utiliza para actualizar la posición de los objetos en una matriz de píxeles mediante una operación llamada convolución.

2.2.3. Matrices RGB

Las matrices RGB contienen valores de píxeles en tres canales: rojo, verde y azul. Cada componente tiene un valor entre 0 y 255, lo que permite representar una variedad de colores. En este proyecto, se utiliza para modelar el movimiento de objetos visualmente.

2.2.4. Algoritmos Paralelos

Los algoritmos paralelos permiten distribuir el procesamiento entre varios núcleos, acelerando la simulación, especialmente cuando se manejan grandes volúmenes de datos.

2.2.5. Convolución

La convolución es una operación matemática donde un kernel se aplica a una matriz para modificar sus valores. Se usa para efectos visuales como el desenfoque o el movimiento de objetos.

2.2.6. Programación Concurrente

La programación concurrente divide un trabajo en varios hilos que se ejecutan simultáneamente, mejorando la eficiencia en tareas como la simulación de movimiento. En Python, la librería `threading` permite gestionar estos hilos.

2.2.7. Memoria Compartida y Paralelización con GPU

El uso de memoria compartida y GPUs permite que múltiples hilos trabajen sobre los mismos datos de manera simultánea, acelerando el procesamiento de simulaciones complejas.

2.3. Subtemas del Marco Conceptual

1. **Simulación de Movimiento con Kernels:** Utilización de kernels para modificar las posiciones de objetos en una matriz bidimensional a través de convoluciones.
2. **Paralelización de Algoritmos de Simulación:** Mejoras en la eficiencia mediante el uso de múltiples hilos y paralelización con GPUs.
3. **Optimización en Simulaciones Paralelas:** Estrategias para dividir tareas, sincronizar hilos y reducir latencias en simulaciones paralelas.

3. Metodología

Este proyecto siguió un enfoque práctico basado en la implementación directa de la simulación del movimiento de un objeto usando kernels en matrices RGB. A continuación, se describe el proceso seguido.

3.1. Uso de kernels en matrices binarias

El primer paso fue aprender a usar un kernel para mover una matriz de valores binarios en un fondo binario. Se trabajó con una matriz de 1s y 0s, donde los 1s representaban un objeto y los 0s el fondo vacío. El objetivo fue aplicar el kernel para mover el objeto modificando las posiciones de los valores, simulando el desplazamiento en un espacio bidimensional.

3.2. Preparación de matrices RGB

Una vez comprendido el uso de kernels en matrices binarias, se pasaron a matrices RGB para representar objetos más complejos. Se utilizaron tres elementos: un gato, un fondo y una pelota, cada uno representado con una matriz RGB que contiene tres valores (rojo, verde y azul) por celda, asegurando una visualización realista.

3.3. Simulación del movimiento del gato

Con las matrices RGB listas, se procedió a mover el gato sobre el fondo. Se aplicó el kernel para desplazar al gato por la matriz del fondo, asegurando un movimiento fluido a lo largo de una trayectoria definida.

3.4. Simulación del gato siguiendo la pelota

Luego, se agregó la pelota y se implementó la lógica para que el gato la siguiera. Se aplicaron las mismas técnicas de movimiento, pero con un enfoque adicional para que el gato se moviera hacia la pelota mientras esta se desplazaba.

3.5. Paralelización del proceso

Para mejorar el rendimiento, se paralelizó la simulación usando múltiples hilos en Python. Esto permitió dividir las tareas de movimiento entre varios hilos, acelerando el proceso y aumentando la eficiencia, especialmente al trabajar con matrices grandes y más objetos.

3.6. Comprobación de resultados

Finalmente, se verificó que el movimiento del gato fuera preciso y fluido, tanto en la versión secuencial como en la paralelizada. Se compararon los tiempos de ejecución, confirmando

que la paralelización mejoró el rendimiento, y se comprobó que el gato siempre siguiera correctamente a la pelota.

4. Propuesta

Este proyecto tiene como objetivo simular el movimiento de un gato en un entorno bidimensional con un fondo de campo y una pelota, utilizando kernels para manipular matrices RGB. Para lograrlo, se sigue una metodología que combina teoría de kernels y programación paralela para obtener una simulación fluida y eficiente.

4.1. Uso de kernels en matrices binarias

El primer paso fue aprender a aplicar un kernel en matrices binarias, donde los valores 1 representaban un objeto y los valores 0 el fondo. Este ejercicio fue esencial para entender cómo mover un objeto dentro de una matriz utilizando la técnica de convolución. Este conocimiento inicial sentó las bases para aplicar kernels en matrices RGB más complejas.

4.1.1. Codificacion:

```
import numpy as np
import matplotlib.pyplot as plt

def mostrar_imagen(fig,ax,imagen, titulo="", cmap=None):
    ax.clear()  # Limpia el eje anterior
    im=ax.imshow(imagen, cmap=cmap)
    ax.set_title(titulo)
    fig.canvas.draw()
    plt.pause(0.05)


"""def avanzar_kernel(fondo,kernel, pasos):
    plt.ion()  # Modo interactivo
    fig, ax = plt.subplots(figsize=(7,7))
    kernel_ach=kernel.shape[1]
    kernel_alt=kernel.shape[0]
    for paso in range(pasos):
        fond=fondo.copy()
        for i in range(kernel_alt):
            for j in range(kernel_ach):
                fond[i][j+paso]*=kernel[i][j]
        mostrar_imagen(fig,ax,fond, "Imagen Binaria (0: Negro, 1: Blanco
   )",cmap='binary')"""

#avanzar_kernel(fondo,kernel,40)
"""def mostrar_imagen_unica(imagen, titulo="", cmap=None):

    plt.figure(figsize=(60, 40)) # Ajusta el tama o de la figura
    plt.imshow(imagen, cmap=cmap)
    plt.title(titulo)
```

```
    #plt.colorbar(label="Intensidad del P xel") # Muestra la barra de color
    plt.axis('off') # Oculta los ejes para una visualizaci n m s limpia
    plt.show()"""

def avanzar_kernel(fondo,kernel,paso,fig,ax):
    kernel_ach=kernel.shape[1]
    kernel_alt=kernel.shape[0]
    #print(kernel_ach,kernel_alt)
    #mov=kernel_ach-1
    fond=fondo.copy()
    avance_pixel=5
    for i in range(kernel_alt):
        for j in range(kernel_ach):
            if not np.all(kernel[i][j]==[255, 255, 255]):
                fond[i + 400][j+paso*avance_pixel]=kernel[i][j]
                #fond[i+400][j+paso]=kernel[i][j]
                #kernel_ach-1-j
            """if not np.array_equal(kernel[i][j], [255, 255, 255]):
                # Mover el p xel a la nueva posici n en el fondo
                if 680 + i < fondo.shape[0] and kernel_ach - 1 - j + paso < fondo.shape[1]:
                    fond[680 + i][kernel_ach - 1 - j + paso] = kernel[i][j]"""
    mostrar_imagen(fig,ax,fond, "Imagen RGb", cmap=None)
    plt.pause(0.01)


def avance():
    plt.ion()
    pasos=140
    fig, ax = plt.subplots(figsize=(40,20))
    #avanzar_kernel(fondo,gato0,0,fig,ax)
    for i in range(0,pasos,10):
        avanzar_kernel(fondop,perro0,i,fig,ax)
        avanzar_kernel(fondop,perro1,i,fig,ax)
        avanzar_kernel(fondop,perro2,i+6,fig,ax)
        avanzar_kernel(fondop,perro3,i+8,fig,ax)
    #avanzar_kernel(fondo,gato3,i,fig,ax)
    plt.pause(0.01)
    plt.show()
avance()
#mostrar_imagen_unica(perro0,"perro")
```

4.1.2. Desarrollo y resultado

Este ejercicio busca aplicar la programación con hilos dividiendo la suma de los primeros 1,000,000 de números naturales entre dos hilos. Cada hilo calcula una mitad y luego se

combinan los resultados usando sincronización para evitar condiciones de carrera. Permite reforzar el uso de threading y mecanismos como Lock para asegurar la correcta manipulación de variables compartidas.

5. Conclusiones

1. **Confirmación de la efectividad de los algoritmos** Todos los algoritmos implementados han demostrado ser efectivos y precisos al ejecutarse con las matrices de prueba proporcionadas. Ejercicios como la suma de diagonales, la rotación y la transposición de matrices se validaron con éxito al comparar los resultados con cálculos manuales. En todos los casos, los resultados fueron consistentes, lo que confirma la correcta implementación de los algoritmos.

2. **Necesidad de optimización en algunos casos** Aunque los algoritmos funcionan correctamente, algunos de ellos, como la transposición de matrices, podrían beneficiarse de una optimización en cuanto a espacio. Implementar una transposición *in-place* en matrices cuadradas reduciría significativamente el consumo de memoria. Esta mejora sería especialmente útil cuando se manejan matrices grandes, como las de 1000x1000 o más, donde el uso de memoria se convierte en un factor crítico.

3. **Validación de la complejidad de los algoritmos** Los análisis de complejidad confirmaron que los algoritmos se comportan dentro de lo esperado. En cuanto a la complejidad temporal, la mayoría de los algoritmos tienen una complejidad de $\mathbf{O(n^2)}$, lo que es coherente con operaciones que requieren recorrer todos los elementos de una matriz. Por otro lado, la complejidad espacial generalmente se mantiene en $\mathbf{O(1)}$, a excepción de los algoritmos que requieren espacio adicional para almacenar resultados, como la rotación de matrices, lo cual es justificado debido al uso de estructuras auxiliares.

6. Resultados

A lo largo de las pruebas y la implementación de los algoritmos, se lograron los siguientes resultados:

- **Funcionamiento correcto en ejercicios básicos:** Las matrices de tamaño 3x3 fueron procesadas correctamente en todos los casos, con resultados que coincidieron exactamente con los cálculos manuales. Esto incluye operaciones como la suma de diagonales, rotación y transposición.
- **Optimización de memoria:** Aunque los algoritmos funcionan bien, algunos podrían beneficiarse de mejoras de eficiencia en el uso de memoria, especialmente cuando se trabaja con matrices grandes. Esto sería particularmente útil en algoritmos como la transposición de matrices, que podrían optimizarse mediante una implementación in-place.
- **Rendimiento en matrices grandes:** Las pruebas realizadas con matrices más grandes (hasta 1000x1000) mostraron que los algoritmos mantienen un rendimiento adecuado, con tiempos de ejecución dentro de los límites razonables. Las métricas de complejidad temporal y espacial se alinearon con las expectativas.
- **Eficiencia de la concurrencia:** Al incorporar el uso de hilos para paralelizar ciertas operaciones, se observó una mejora en el rendimiento en casos de operaciones de I/O o procesamiento repetitivo, aunque el impacto no fue significativo en operaciones simples debido a las limitaciones del Global Interpreter Lock (GIL) en Python.