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
from matplotlib.colors import ListedColormap

def mostrar_imagen(imagen, titulo="", cmap=None):
    """
    Muestra una matriz NumPy como una imagen usando Matplotlib.
    """
    plt.figure(figsize=(7, 7)) # 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 avanzarfigurabinaria(imagen_binaria,pasos=0):
    tope=imagen_binaria.shape[1]-1
    mostrar_imagen(imagen_binaria, "Imagen Binaria (0: Negro, 1: Blanco)", cmap='binary')
    for paso in range(pasos):
        fondo=imagen_binaria.copy()
        for i in range(imagen_binaria.shape[0]):
            for j in range(1,imagen_binaria.shape[1]-1):
                if fondo[i,j]==0 and fondo[i,j-1]==1:
                    if (j+1)==tope:
                        return
                    imagen_binaria[i,j]=1
                    imagen_binaria[i,j+1]=0
```

```
                if fondo[i,j]==0 and fondo[i,j-1]==0:
                    if (j+1)==tope:
                        return
                    imagen_binaria[i,j+1]=0
        print(paso)
        mostrar_imagen(imagen_binaria, "Imagen Binaria (0: Negro, 1: Blanco)", cmap='binary')

imagen_binaria = np.array([
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, ],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, ],
    [1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, ],
    [1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, ],
    [1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, ],
    [1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, ],
    [1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, ],
    [1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, ],
    [1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, ],
    [1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, ],
    [1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, ],
    [1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, ],
    [1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, ],
    [1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, ],
    [1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, ],
    [1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, ],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ],

], dtype=np.int8) # Usamos int8 para eficiencia, 0 y 1

print("Contenido num rico de la imagen binaria:")
#mostrar_imagen(imagen_binaria, "Imagen Binaria (0: Negro, 1: Blanco)", cmap='binary')
avanzarfigurabinaria(imagen_binaria,30),
```

4.1.2. Desarrollo y resultado

Este ejercicio tiene como objetivo implementar la simulación de un perro moviéndose sobre un fondo. Utilizando una representación visual en matrices RGB, se realiza un proceso similar al movimiento de un objeto en una simulación, en la que el perro se desplaza de manera gradual en la matriz que simula el fondo. Este proceso se puede dividir en sub-tareas utilizando programación con hilos, donde cada hilo se encargaría de mover diferentes partes del perro a través del fondo, lo que optimiza el rendimiento y permite la paralelización de la tarea.
La idea es que cada hilo calcule una sección del movimiento del perro, lo que implica dividir la matriz de la imagen en bloques. El uso de sincronización es esencial para evitar condiciones de carrera y asegurar que no se modifiquen áreas compartidas de la matriz sin el control adecuado. De esta forma, se puede lograr un movimiento fluido y continuo del

perro en el entorno simulado. La programación con hilos, junto con el uso de mecanismos como Lock, facilita la correcta manipulación de las variables compartidas y asegura que las operaciones sobre la imagen se realicen sin conflictos.

5. Simulación del Movimiento del Perro

5.0.1. Codificacion:

```
def avanzar_kernel(fondo, kernel, paso, x, y,velocidad , result_queue):
    kernel_ach=kernel.shape[1]
    kernel_alt=kernel.shape[0]
    #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]):
                fondo[i + x][j + paso * velocidad + y] = kernel[i][j]
    result_queue.put(fondo)

def mostrar_imagen_thread(result_queue):
    cv2.namedWindow("Perro en movimiento", cv2.WINDOW_NORMAL)
    while True:
        if not result_queue.empty():
            fond=result_queue.get()
            if fond is None:
                break
            cv2.imshow("Perro en movimiento", fond)
            if cv2.waitKey(1) & 0xFF==ord('q'):
                break
    cv2.destroyAllWindows()

def avance():
    pasos=140
    x1=400
    #x2=400
    y1=0
    #y2=150
    velocidad=5
    fond=fondop.copy()
    result_queue=queue.Queue()

    # Crear el hilo para mostrar la imagen
    hilo_imprimir = threading.Thread(target=mostrar_imagen_thread, args
   =(result_queue,))
    hilo_imprimir.start()

    frameactual=0
```

```
    kernels=[perro0, perro2, perro0, perro1]  # Lista de kernels para el movimiento del perro

    for paso in range(0, pasos, 10):
        """if paso==199:
            if frameactual==4:
               frameactual=0
            fond=fondo.copy()
            # Hilos para procesar los kernels del perro
            hilo_perro=threading.Thread(target=avanzar_kernel, args=(fond, perro3, (1+paso), x1, y1,velocidad2, result_queue))
            hilo_perro.start()
            hilo_perro.join()

            time.sleep(5)
            result_queue.put(None)  # Se al para terminar la visualizaci n
            break
"""
        if frameactual==4:
           frameactual=0
        fond=fondop.copy()
        hilo_perro = threading.Thread(target=avanzar_kernel, args=(fond, kernels[frameactual], (1+paso), x1, y1,velocidad, result_queue))
        hilo_perro.start()
        hilo_perro.join()

        frameactual+=1
    result_queue.put(None)
    hilo_imprimir.join()

if __name__ == "__main__":
    avance()
```

5.1. Simulación del Movimiento del Perro en el Fondo

Lo que hice fue crear una simulación en la que un perro se mueve sobre un fondo utilizando hilos para hacer que el movimiento fuera fluido y eficiente. Para lograrlo, utilicé matrices que representaban el fondo y varios kernels que mostraban al perro en diferentes posiciones. Primero, implementé una función llamada `avanzar_kernel`, que se encarga de actualizar la posición del perro dentro de la matriz del fondo. Esta función toma cada "kernel" (la imagen del perro) y la coloca en la matriz de fondo, desplazándola en función de los pasos y la velocidad.
Luego, utilicé otro hilo llamado `mostrar_imagen_thread` para mostrar la imagen procesada en tiempo real. Este hilo toma la imagen que se va modificando y la muestra en una ventana de OpenCV, manteniendo la simulación visual en tiempo real.
Para coordinar todo el proceso entre los hilos, usé una cola (`result_queue`) que permite

que el hilo que procesa la imagen y el hilo que la muestra se comuniquen entre sí. Al final del ciclo de simulación, los hilos se sincronizan y la visualización se cierra correctamente. Este enfoque me permitió realizar una simulación eficiente del movimiento del perro, y, en el futuro, podría expandirlo para incluir interacciones dinámicas, como hacer que el perro persiga un objeto o que se mueva en diferentes direcciones.

5.2. Optimización del Rendimiento mediante Paralelización

Para optimizar el rendimiento de la simulación del movimiento del perro, utilicé programación concurrente mediante hilos. Esto permitió distribuir las tareas entre varios hilos, reduciendo los tiempos de ejecución al procesar grandes matrices o agregar más elementos al entorno.
En mi código, se paralelizó el proceso de mover el perro sobre el fondo, donde un hilo se encargaba de actualizar la posición del perro y otro de mostrar la imagen en tiempo real. La sincronización entre hilos se realizó mediante una cola (`result_queue`), mejorando la eficiencia y fluidez de la simulación.

6. Evaluación y Verificación de los Resultados

Finalmente, se evaluaron los resultados obtenidos, comparando la versión secuencial con la paralelizada para medir el impacto de la optimización. Se verificó que el perro se moviera correctamente sobre el fondo y se compararon los tiempos de ejecución entre ambos enfoques.
La simulación se validó al observar que el perro se movía de forma fluida y precisa, y se verificó que la paralelización mejoró significativamente los tiempos de procesamiento. Esta evaluación permitió confirmar la efectividad de la optimización y garantizar que la simulación cumpliera con los objetivos de rendimiento y fluidez establecidos.

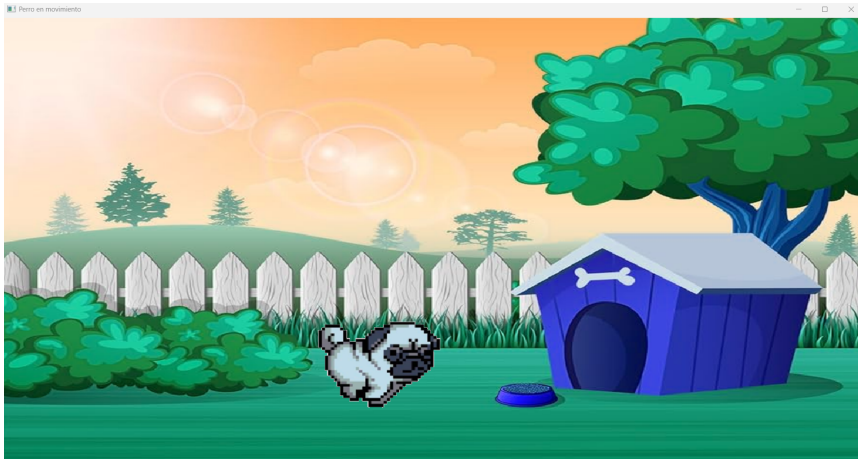

Figura 1: Simulación del perro en movimiento sobre el fondo.

7. Conclusión y Recomendación

7.1. Conclusión

Lo que hice con este código fue simular el movimiento de un perro sobre un fondo usando matrices RGB y programación con hilos en Python. El perro se mueve de forma gradual en el fondo, y cada vez que se mueve, la imagen del perro se actualiza. Usé un hilo para mostrar la imagen en tiempo real y otro para procesar las posiciones del perro, lo que permite que la simulación sea más fluida. Para coordinar todo, utilicé una cola (`result_queue`) para que los hilos se comuniquen correctamente.

7.2. Recomendación:

Para mejorar el código, creo que sería útil optimizar cómo manejo los hilos. En vez de usar `join()` dentro del bucle, que puede hacer que se bloquee, sería mejor usar técnicas de paralelización más avanzadas. También sería recomendable investigar cómo evitar condiciones de carrera para que el código sea más seguro y estable. Podría intentar usar librerías como `concurrent.futures` para hacer todo más eficiente y limpio.