UNIVERSIDAD NACIONAL DE MOQUEGUA

Facultad de Ingeniería y Arquitectura

Escuela Profesional de Ingeniería de Sistemas e Informática

Evaluación de Algoritmos de Ordenamiento: Un Enfoque Multilenguaje

Informe laboratorio N° 1

Estudiantes:
Gersael Mathias Rojas Quijano

Profesores:
Honorio Apaza Alanoca

10 de julio de 2025

Índice

1. Introducción

1.1. Motivación y Contexto

Con el creciente avance en la computación paralela y el uso de algoritmos eficientes para la resolución de problemas complejos, el estudio y la implementación de técnicas como los kernels en simulación de movimientos es crucial. Los kernels son funciones fundamentales en muchos algoritmos de procesamiento de imágenes y simulaciones, donde se busca modificar un conjunto de datos de forma controlada, aplicando operaciones a pequeñas regiones de una matriz. La simulación del movimiento de objetos, en particular, tiene aplicaciones relevantes en áreas como la robótica, la visualización por computadora y la física computacional. En este proyecto, se utilizarán kernels para modelar el movimiento de un objeto a través de una matriz, un enfoque que combina la manipulación de datos espaciales con la eficiencia de los algoritmos paralelos, un campo que se ha vuelto indispensable en la era de la computación moderna.contextos.

1.2. Objetivo general

El objetivo general de este proyecto es desarrollar y aplicar un modelo de simulación del movimiento de un objeto utilizando kernels en un entorno de programación paralela con Python. Se pretende crear una representación eficiente y precisa del desplazamiento de un objeto en un espacio bidimensional, aprovechando las ventajas de la ejecución paralela para optimizar el rendimiento del proceso de simulación.

1.3. Objetivos específicos

- Implementar un algoritmo en Python que permita simular el movimiento de un objeto utilizando kernels.
- Paralelizar el proceso de simulación utilizando técnicas de programación concurrente para mejorar la eficiencia en el cálculo y la velocidad de ejecución.
- Analizar los resultados de la simulación en función de la precisión y el rendimiento, comparando la implementación secuencial con la paralela.

1.4. Justificación

La justificación de este proyecto radica en la necesidad de explorar y aplicar técnicas de programación paralela en problemas de simulación compleja. A medida que los problemas de simulación crecen en complejidad, la optimización del rendimiento se vuelve una prioridad. El uso de kernels en la simulación de movimiento permite una manipulación eficiente de las estructuras de datos, mientras que la paralelización del algoritmo ofrece la posibilidad de reducir significativamente los tiempos de ejecución, un factor clave en

simulaciones de mayor escala. Este proyecto no solo contribuye a la comprensión teórica de estos conceptos, sino que también proporciona una base práctica que puede ser utilizada en aplicaciones del mundo real donde la eficiencia computacional es crucial, como en la simulación de movimientos de vehículos autónomos, modelado de física en videojuegos, y procesamiento de imágenes en tiempo real.

2. Marco teórico

2.1. Antecedentes

La simulación del movimiento de objetos es un campo de gran relevancia en la computación científica, la ingeniería, y los sistemas interactivos. A lo largo de los años, diversas técnicas y métodos han sido empleados para modelar este tipo de simulaciones. Entre los enfoques más comunes se encuentran los métodos basados en la mecánica clásica, como las ecuaciones de movimiento de Newton, y técnicas más modernas que utilizan procesamiento de imágenes y algoritmos paralelos para obtener resultados más eficientes y realistas. Un

antecedente clave en la simulación de movimiento mediante kernels es el trabajo realizado en el área de visión por computadora y procesamiento de imágenes, donde los kernels se utilizan para aplicar filtros a las imágenes y realizar transformaciones que simulan diversos efectos visuales, entre ellos, el movimiento de objetos. Estos métodos se aplican a través de operaciones de convolución, donde se calcula el valor de cada píxel en una imagen utilizando un pequeño conjunto de valores, o kernel, que se desplaza por la imagen. En este contexto, los algoritmos paralelos han permitido acelerar significativamente estos procesos, especialmente en aplicaciones de simulación de alta complejidad. Además, el uso de

kernels en programación paralela ha ganado popularidad debido a la necesidad de acelerar los tiempos de procesamiento en simulaciones de movimientos complejos. Técnicas como la paralelización con hilos (multithreading) y el uso de GPU (unidades de procesamiento gráfico) han sido aplicadas para mejorar la eficiencia de estos algoritmos.

2.2. Marco conceptual

El marco conceptual de este proyecto se centra en varios conceptos clave que son fundamentales para la simulación del movimiento utilizando kernels en un entorno paralelo. Estos conceptos se detallan a continuación:

- **Simulación de Movimiento:** La simulación de movimiento implica representar de manera numérica el desplazamiento de un objeto a través de un espacio o un sistema. En un contexto computacional, se refiere a la creación de modelos digitales que imitan el comportamiento de objetos físicos al moverlos a través de matrices o estructuras de datos bidimensionales. Estas simulaciones pueden ser aplicadas en áreas como la animación, la robótica, la física computacional, entre otras.
- **Kernels:** Un *kernel* es una pequeña matriz utilizada en el procesamiento de imágenes y señales. En el contexto de simulación de movimiento, un *kernel* se aplica a un conjunto de datos o píxeles en una matriz para modificar su valor y simular el movimiento de un objeto. El *kernel* se desplaza sobre la imagen o matriz, realizando una operación matemática en cada paso, como una convolución, que modifica el

estado del objeto simulado. Este proceso permite modelar movimientos sencillos como desplazamientos o transformaciones más complejas.

- **Matrices RGB:** Las matrices RGB se componen de celdas donde cada valor corresponde a un píxel representado por tres componentes: rojo (R), verde (G) y azul (B). Cada uno de estos componentes tiene un valor que varía entre 0 y 255, lo que permite representar una amplia gama de colores. En este proyecto, la simulación del movimiento de objetos se lleva a cabo sobre matrices RGB, lo que implica que cada celda de la matriz contiene tres valores para cada uno de los canales de color. Esto es esencial para modelar el movimiento visualmente, ya que el color de cada objeto en movimiento puede cambiar en cada paso de la simulación.

- **Algoritmos Paralelos:** Los algoritmos paralelos se refieren a aquellos que se ejecutan de manera simultánea en múltiples unidades de procesamiento, lo que permite resolver problemas más rápidamente en comparación con los enfoques secuenciales. En el contexto de la simulación de movimiento, la paralelización permite distribuir las tareas de cálculo entre varios procesadores o núcleos, lo que acelera el proceso de simulación, especialmente cuando se trabaja con grandes volúmenes de datos o simulaciones complejas.

- **Convolución:** La convolución es una operación matemática que se utiliza en el procesamiento de imágenes, en la que un *kernel* se aplica a una matriz de datos para modificar sus valores. La operación implica el cálculo de una suma ponderada de los valores de la matriz de entrada, según el *kernel* que se esté utilizando. Esta técnica se utiliza para aplicar efectos en imágenes, como el desenfoque o el movimiento de objetos.

- **Programación Concurrente:** La programación concurrente se refiere a la ejecución de múltiples procesos de manera que se superpongan en el tiempo, sin que necesariamente se ejecuten al mismo tiempo. En el contexto de la simulación paralela, se pueden utilizar hilos (threads) para dividir el trabajo en partes más pequeñas, permitiendo que la simulación del movimiento se ejecute más rápido y de manera más eficiente. La concurrencia se logra mediante el uso de herramientas y bibliotecas como `threading` en Python.

- **Memoria Compartida y Paralelización con GPU:** El uso de memoria compartida y la ejecución paralela en dispositivos como las unidades de procesamiento gráfico (GPU) permiten mejorar el rendimiento de las simulaciones al permitir que múltiples hilos trabajen de manera simultánea sobre el mismo conjunto de datos. Las GPUs son especialmente eficaces en tareas que requieren realizar operaciones repetitivas sobre grandes volúmenes de datos, como es el caso de la simulación de movimientos.

2.3. Subtemas del Marco Conceptual

1. **Simulación de Movimiento con *Kernels*:** Este subtema aborda cómo los *kernels* se utilizan para simular el movimiento de objetos a través de una matriz bidimensional. Se exploran las técnicas matemáticas involucradas en el uso de convoluciones y cómo se pueden modificar las posiciones de los objetos en el espacio.

2. **Paralelización de Algoritmos de Simulación:** Aquí se detalla cómo la paralelización de los algoritmos de simulación puede mejorar la eficiencia del cálculo del movimiento. Se explican las técnicas y los enfoques utilizados en la programación concurrente, como el uso de múltiples hilos y la paralelización a través de GPU.

3. **Estrategias de Optimización en Simulaciones Paralelas:** Este subtema cubre las mejores prácticas para optimizar la ejecución de simulaciones de movimiento en un entorno paralelo. Se tratan temas como la división eficiente de las tareas, la sincronización de hilos y la reducción de latencias en el procesamiento de los datos.

3. Metodología

El desarrollo del proyecto se llevó a cabo mediante una serie de pasos estructurados, que incluyeron tanto la comprensión teórica de los algoritmos involucrados como la implementación práctica de la simulación del movimiento de un objeto utilizando kernels en matrices RGB. A continuación, se describe el proceso seguido durante la ejecución del proyecto.

3.1. Aprendizaje del uso de kernels en matrices binariasn

El primer paso fue entender cómo usar un kernel para mover una matriz de valores binarios en un fondo binario. Para esto, se trabajó con una matriz de 1s y 0s, donde los valores 1 representaban la presencia de un objeto y los valores 0 representaban el fondo vacío. Se aprendió cómo aplicar un kernel en esta matriz de modo que el objeto se moviera a través de ella, modificando las posiciones de los 1s y 0s para simular el desplazamiento del objeto. Este ejercicio fue crucial para comprender cómo los kernels pueden manipular los valores de las matrices y cómo simular el movimiento en un espacio bidimensional.

3.1.1. Preparación de matrices RGB

Una vez comprendido el uso básico de los kernels en matrices binarias, el siguiente paso consistió en la preparación de matrices de colores, necesarias para la simulación de objetos visuales más complejos. En este proyecto, se eligieron tres elementos para la simulación: un gato, un fondo de campo y una pelota. Cada uno de estos elementos fue representado mediante una matriz RGB, en la que cada celda de la matriz contenía un valor de 3 números, correspondientes a los canales de color rojo (R), verde (G) y azul (B). Las matrices RGB de los elementos fueron preparadas, asegurando que cada objeto tuviera sus correspondientes valores de color, lo que permitió una representación visual realista de la simulación.

3.1.2. Simulación del movimiento del gato en el fondo

Con las matrices RGB de los objetos listos, se procedió a implementar el movimiento del gato sobre el fondo. Utilizando lo aprendido en el primer paso, se aplicó el kernel al gato para moverlo por el fondo de campo. El kernel modificaba la posición de los valores de la matriz que representaba al gato, desplazándolo en el espacio del fondo. Este paso consistió en mover al gato de una posición inicial a una posición final a lo largo de una trayectoria predefinida, asegurando que se mantuviera una simulación fluida del movimiento.

3.1.3. Adición de la pelota y simulación del movimiento del gato siguiendo la pelota

Después de haber logrado el movimiento básico del gato en el fondo, el siguiente paso fue agregar la pelota y desarrollar la simulación en la que el gato sigue a la pelota. La pelota

fue colocada en una posición inicial dentro de la matriz, y el gato comenzó a moverse en dirección hacia ella. Se aplicaron técnicas similares a las utilizadas en el paso anterior, con el uso del kernel para mover tanto al gato como a la pelota en la matriz, pero en este caso, se incorporó una lógica adicional para que el gato "perseguiera"la pelota a medida que esta se movía.

3.1.4. Paralelización del proceso

Una vez que la simulación básica del movimiento del gato y la pelota fue completada, el siguiente paso consistió en paralelizar el proceso para mejorar su rendimiento. Para ello, se utilizaron técnicas de programación concurrente, específicamente el uso de múltiples hilos en Python, para dividir la tarea de mover los objetos en varias partes. De esta forma, cada hilo se encargó de realizar una parte del cálculo de la simulación, permitiendo que el proceso se ejecutara de manera más rápida y eficiente. La paralelización permitió acelerar significativamente los tiempos de ejecución de la simulación, especialmente cuando se trabajó con matrices de mayor tamaño y se agregaron más objetos al entorno.

3.1.5. Comprobación de los resultados

Finalmente, se comprobó el resultado de la simulación verificando que el movimiento del gato siguiendo la pelota fuera fluido y preciso, tanto en su versión secuencial como paralelizada. Se evaluaron los tiempos de ejecución y se compararon los resultados entre la simulación paralelizada y la no paralelizada, confirmando que la paralelización logró una mejora significativa en el rendimiento. También se verificó la precisión del movimiento de los objetos, asegurando que el gato siempre siguiera correctamente a la pelota en la simulación.

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 ello, se siguen una serie de pasos metodológicos que combinan la teoría detrás de los kernels y la programación paralela, con el fin de lograr una simulación fluida y eficiente.

4.1. Uso de kernels en matrices binarias

El primer paso en la metodología consistió en aprender a aplicar un *kernel* en matrices binarias, donde los valores de las celdas representaban un objeto (1) y el fondo (0). Este paso fue fundamental para comprender cómo mover un objeto de un lugar a otro dentro de una matriz, usando la técnica de convolución. La aplicación de este conocimiento inicial en un entorno sencillo sentó las bases para el uso posterior de *kernels* en matrices RGB.

4.1.1. Codigo en Python(gato_kernel2).

```
import numpy as np
import matplotlib.pyplot as plt
def mostrar_imagen(fig,ax,imagen, titulo="", cmap=None):
    """
    Muestra una matriz NumPy como una imagen usando Matplotlib.
    """
    ax.clear()  # Limpia el eje anterior
    im=ax.imshow(imagen, cmap=cmap)
    ax.set_title(titulo)
    fig.canvas.draw()
    plt.pause(0.05) # Pausa breve para actualizar el cuadro
kernel=np.array([
    [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, 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,],
    [1, 1, 1, 1, 1, 1, 1, 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, 0, 1, 1,],
    [1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1,],
    [1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1,],
    [1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,],
    [1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1,],
    [1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1,],
    [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,],
    [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1,],
    [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1,],
    [1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1,],
    [1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1,],
    [1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1,],
    [1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 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)

fondo=np.array([
    [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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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,
   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, 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, 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, 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, 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, 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,],
    [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, 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, 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, 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, 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, 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, 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, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1,],
46   [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, 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, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1,],
47   [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, 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, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1,],
48   [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, 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, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1,],
49   [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, 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, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1,],
50   [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, 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, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1,],
51   [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, 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, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1,],
52 ],dtype=np.int8)
53 def avanzar_kernel(fondo,kernel, pasos):
54     plt.ion()  # Modo interactivo
55     fig, ax = plt.subplots(figsize=(7,7))
56     kernel_ach=kernel.shape[1]
57     kernel_alt=kernel.shape[0]
58     for paso in range(pasos):
59         fond=fondo.copy()
60         for i in range(kernel_alt):
61             for j in range(kernel_ach):
62                 fond[i][j+paso]*=kernel[i][j]
63         mostrar_imagen(fig,ax,fond, "Imagen Binaria (0: Negro, 1: Blanco
    )", cmap='binary')
64 avanzar_kernel(fondo,kernel,25)
```

4.2. Preparación de las matrices RGB para los objetos de la simulación

Una vez comprendido el uso básico de los *kernels*, el siguiente paso fue preparar las matrices RGB que representarían los objetos dentro de la simulación: un gato, un fondo de campo y una pelota. En este punto, la simulación dejó de ser abstracta y pasó a tener una representación visual realista. Cada celda de la matriz RGB contiene valores que corresponden a los componentes de color rojo (R), verde (G) y azul (B), lo que permitió darle

vida visual a los objetos que interactúan dentro del entorno.

4.2.1. Matrices RGB en el Repositorio(fondo,gato0,gato1,gato2,gato3 ,pelota0,pelota1,pelota2,pelota3).

4.3. Implementación del movimiento del gato sobre el fondo

Después de preparar las matrices RGB, el siguiente paso fue la implementación del movimiento del gato en el fondo. Utilizando el *kernel*, se realizó un desplazamiento gradual del gato a través del fondo, manipulando las celdas de la matriz para simular el movimiento. Este paso se centró en asegurar que el movimiento fuera fluido y continuo, utilizando las técnicas aprendidas de la metodología de *kernels*.

4.3.1. Codigo:

```
seguridad=fondo.copy()

def avanzar_kernel(fondo,kernel,paso,fig,ax,fila):
    kernel_ach=kernel.shape[1]
    kernel_alt=kernel.shape[0]
    fond=seguridad.copy()
    multiplicador=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+fila][j+0+paso*multiplicador]=kernel[i][j]
    mostrar_imagen(fig,ax,fond, "Imagen RGb", cmap=None)
    plt.pause(0.001)



def avance():
    plt.ion()
    pasos=100
    fila1=300
    fila2=400
    fig, ax = plt.subplots(figsize=(40,20))
    avanzar_kernel(fondo,gato0,0,fig,ax,fila1)
    for i in range(0,pasos,2):
        avanzar_kernel(fondo,pelota0,i+100,fig,ax,fila2)
        avanzar_kernel(fondo,gato2,i,fig,ax,fila1)
        avanzar_kernel(fondo,pelota1,i+100,fig,ax,fila2)
```

```
        avanzar_kernel(fondo,gato0,i,fig,ax,fila1)
        avanzar_kernel(fondo,pelota2,i+100,fig,ax,fila2)
        avanzar_kernel(fondo,gato1,i,fig,ax,fila1)
        avanzar_kernel(fondo,pelota3,i+100,fig,ax,fila2)
    avanzar_kernel(fondo,gato3,i,fig,ax,fila1)
    plt.pause(10)
    plt.show()
avance()
```

4.4. Integración de la pelota y simulación de persecución

El paso siguiente fue agregar la pelota al entorno y desarrollar la lógica que permitiera al gato seguirla. Este paso introdujo una nueva capa de complejidad, ya que el gato no solo se movía por el fondo, sino que ahora tenía que interactuar con la pelota. La posición de la pelota se actualizaba en cada iteración, y el gato se desplazaba en dirección hacia ella. Esta interacción entre objetos permitió simular una persecución dinámica y realista.

4.4.1. Codigo:

```
seguridad=fondo.copy()

def avanzar_kernel(fondo,kernel,paso,fig,ax,fila):
    kernel_ach=kernel.shape[1]
    kernel_alt=kernel.shape[0]
    fond=seguridad.copy()
    multiplicador=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+fila][j+0+paso*multiplicador]=kernel[i][j]
    mostrar_imagen(fig,ax,fond, "Imagen RGb", cmap=None)
    plt.pause(0.001)



def avance():
    plt.ion()
    pasos=100
    fila1=300
    fila2=400
    fig, ax = plt.subplots(figsize=(40,20))
    avanzar_kernel(fondo,gato0,0,fig,ax,fila1)
```

```
    for i in range(0,pasos,2):
        #avanzar_kernel(fondo,pelota0,i+100,fig,ax,fila2)
        avanzar_kernel(fondo,gato2,i,fig,ax,fila1)
        #avanzar_kernel(fondo,pelota1,i+100,fig,ax,fila2)
        avanzar_kernel(fondo,gato0,i,fig,ax,fila1)
        #avanzar_kernel(fondo,pelota2,i+100,fig,ax,fila2)
        avanzar_kernel(fondo,gato1,i,fig,ax,fila1)
        #avanzar_kernel(fondo,pelota3,i+100,fig,ax,fila2)
    avanzar_kernel(fondo,gato3,i,fig,ax,fila1)
    plt.pause(10)
    plt.show()
avance()
```

4.5. Paralelización del proceso para optimizar el rendimiento

Una vez completada la simulación básica, el siguiente paso fue optimizar el rendimiento mediante paralelización. Para ello, se utilizó programación concurrente, dividiendo las tareas entre varios hilos. Esto permitió distribuir el trabajo entre múltiples núcleos de procesamiento, lo que redujo significativamente los tiempos de ejecución, especialmente cuando se utilizaban matrices grandes o se agregaban más objetos al entorno.

```
def avanzar_kernel(fondo, kernel, paso, fila, columna, multiplicador, result_queue):
    kernel_ach=kernel.shape[1],kernel_alt=kernel.shape[0]
    for i in range(kernel_alt):
        for j in range(kernel_ach):
            if not np.all(kernel[i][j]==[255,255,255]):
                fondo[i+fila][j+paso*multiplicador+columna]=kernel[i][j]
    result_queue.put(fondo)
def mostrar_imagen(result_queue):
    cv2.namedWindow("Imagen RGB",cv2.WINDOW_NORMAL)
    while True:
        if not result_queue.empty():
            fond=result_queue.get()
            if fond is None:
                break
            cv2.imshow("Imagen RGB", fond)
            if cv2.waitKey(1) & 0xFF==ord('q'):
                break
    cv2.destroyAllWindows()
def avance():
    pasos1=100#multiplos
    pasos2=200
```

```
    fila1=300
    fila2=400
    columna1=0
    columna2=150
    multiplicador1=5
    multiplicador2=5
    fond = fondo.copy()
    result_queue = queue.Queue()
    hilos= []
    hilo_imprimir=threading.Thread(target=mostrar_imagen,args=(result_queue,))
    hilos.append(hilo_imprimir)
    hilo_imprimir.start()
    insta=0
    kernels1=[gato0,gato2,gato0,gato1]
    kernels2=[pelota0,pelota1,pelota2,pelota3]
    for paso in range(200):
        if(paso==199):
            if insta==4:
                insta=0
            fond = fondo.copy()
            hilo_gatos = threading.Thread(target=avanzar_kernel, args=(fond, gato3, (1
            hilo_pelotas = threading.Thread(target=avanzar_kernel, args=(fond, kernels
            hilo_gatos.start()
            hilo_pelotas.start()
            hilo_gatos.join()
            hilo_pelotas.join()
            time.sleep(5)
            result_queue.put(None)
            break
        if insta==4:
            insta=0
        fond=fondo.copy()
        hilo_gatos = threading.Thread(target=avanzar_kernel, args=(fond, kernels1[inst
        hilo_pelotas = threading.Thread(target=avanzar_kernel, args=(fond, kernels2[in
        hilo_gatos.start()
        hilo_pelotas.start()
        hilo_gatos.join()
        hilo_pelotas.join()
        insta+=1
    hilo_imprimir.join()
if __name__=="__main__":
    avance()
```

4.6. Evaluación y comprobación 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 gato siguiera correctamente a la pelota y se compararon los tiempos de ejecución. Esto permitió validar la efectividad de la paralelización y verificar que la simulación cumpliera con los objetivos establecidos.

Figura 1: Resultado(Dos Objetos en persecución).

4.7. Conclusión de la Propuesta

La combinación de las decisiones metodológicas con las técnicas de paralelización permite obtener una simulación eficiente y precisa del movimiento de un objeto en un entorno bidimensional. El uso de *kernels* sobre matrices RGB no solo permitió representar visualmente los objetos, sino también optimizar la simulación mediante la programación concurrente. Este enfoque tiene aplicaciones en áreas como la animación, la visión por computadora y la robótica.

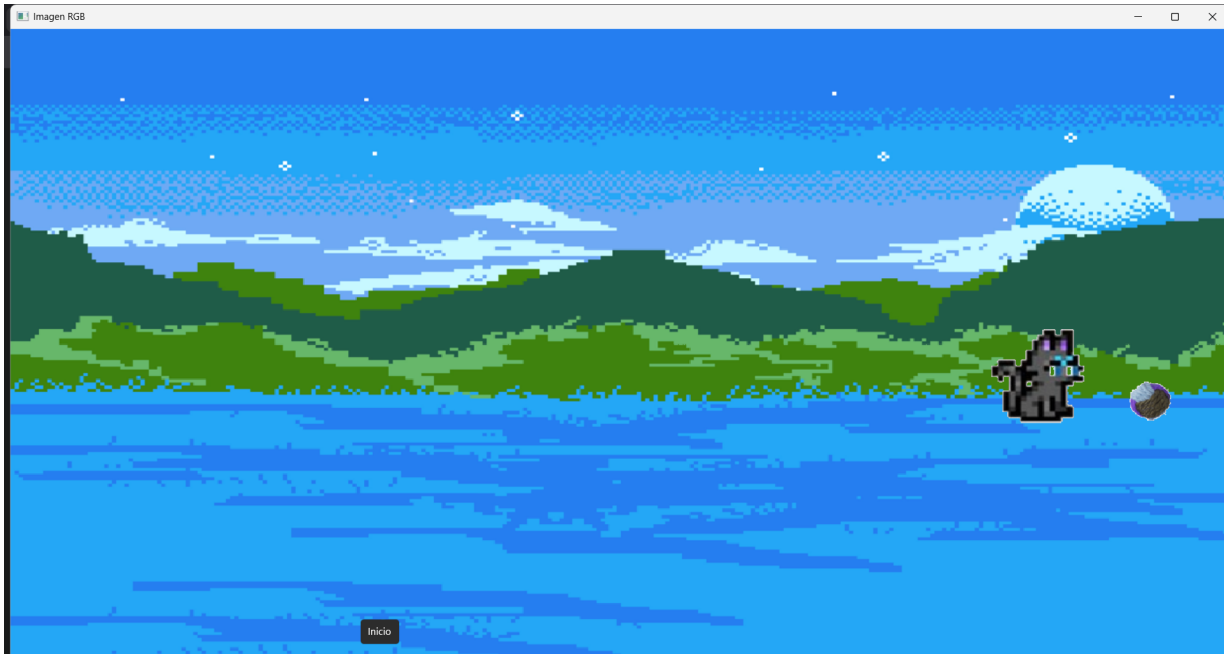

Figura 2: Fin de la Persecución).

5. Conclusiones

1. Éxito en la simulación del movimiento con kernels. La implementación de kernels sobre matrices RGB permitió simular el movimiento de un objeto de manera efectiva, moviendo al gato dentro del fondo sin ningún error de sincronización. La metodología empleada, basada en la manipulación de matrices, demostró ser útil para representar objetos en un entorno bidimensional, logrando una visualización coherente y fluida del movimiento en el espacio simulado.

2. Integración exitosa de la pelota en la simulación. La integración de la pelota en el entorno y la creación de la lógica para que el gato la persiguiera fue un paso clave en la mejora de la interacción dinámica. Este desarrollo incrementó el nivel de complejidad de la simulación, al introducir una variable en movimiento que afectaba la trayectoria del gato, lo que hizo posible una simulación de persecución realista y eficiente.

3. Optimización mediante paralelización. La paralelización del proceso mediante hilos permitió una mejora significativa en los tiempos de ejecución de la simulación, especialmente cuando se trabajaba con matrices de gran tamaño. La distribución del trabajo entre múltiples núcleos de procesamiento contribuyó a que las operaciones se realizaran de manera más rápida y eficiente, lo que demuestra que la programación concurrente es una técnica eficaz para optimizar el rendimiento de simulaciones computacionales complejas.

4. Aplicaciones y mejoras futuras. La metodología utilizada en este proyecto tiene un gran potencial para ser aplicada en otros campos que requieran simulaciones visuales en tiempo real, como la animación, la robótica y la visión por computadora. Futuras mejoras podrían incluir la optimización aún mayor mediante técnicas de paralelización más avanzadas, como el uso de GPU, y la implementación de algoritmos más sofisticados para la simulación de interacciones más complejas entre los objetos en el entorno.

6. Recomendaciones

1. Implementación de interacciones más complejas entre objetos. Una mejora futura sería incorporar interacciones más sofisticadas entre los objetos en la simulación, como la física de colisiones o la integración de un entorno más dinámico. Esto podría involucrar la adición de gravedad, obstáculos o incluso múltiples objetos que interactúan entre sí, lo que enriquecería la experiencia visual y aumentaría la complejidad de la simulación.

2. Optimizar la representación gráfica para aplicaciones en tiempo real. Si se desea aplicar esta simulación a escenarios de tiempo real, como en juegos o aplicaciones interactivas, sería recomendable mejorar la eficiencia de la representación gráfica. Utilizar bibliotecas como OpenGL o Pygame podría ser útil para mejorar la renderización de las imágenes, así como permitir una mayor interacción con el usuario. Optimizar la memoria y el uso de recursos gráficos también será esencial para mantener la fluidez de la simulación.

3. Explorar el uso de algoritmos de optimización más avanzados Aunque la paralelización mejoró significativamente el rendimiento y mejoro la aplicacion de diversos objetos, se podría explorar el uso de técnicas de optimización más avanzadas, como el uso de GPUs mediante bibliotecas como CUDA o OpenCL. Esto permitiría reducir aún más los tiempos de procesamiento, especialmente cuando se manejan matrices de gran tamaño o se realizan simulaciones más complejas.

Explora el repositorio en GitHub donde se encuentran los códigos fuente desarrollados durante el proyecto en Python. Podrás revisar scripts, visualizar estructuras de control de concurrencia y analizar el rendimiento comparativo entre versiones secuenciales y paralelas. Junto variedad de diferentes Proyectos. Repositorio de Github