Maestría en Cómputo Estadístico (2025), 1–4
Curso - Análisis de Imágenes con Deep Learning

Tarea 3

Aplicación de Filtros de Convolución en Imágenes Usando CuPy

Gustavo Hernández Angeles[1]
[1]CIMAT, A.C., Unidad Monterrey, N.L., México

Abstract

This guide is for authors who are preparing papers for the *Publications of the Astronomical Society of Australia* journal using the LaTeX document preparation system and the CUP PAS style file.

1. Introducción

En esta tarea, se implementan filtros de convolución en imágenes utilizando la biblioteca CuPy Nishino and Loomis (2017), que permite realizar operaciones de álgebra lineal en GPU a través de Python. Los filtros de convolución son herramientas fundamentales en el procesamiento de imágenes, ya que permiten resaltar características específicas como bordes, desenfoques y detalles finos. Se aplican varios filtros comunes, como el filtro de Laplace, el filtro de promedio y también se implementa un filtro sin convolución: el filtro de la derivada. El objetivo es comparar los resultados obtenidos con estos filtros y analizar su efectividad cuando son ejecutados en una GPU con respecto a su implementación en CPU.

1.1. Filtro de Laplace

El filtro de Laplace es un operador de segunda derivada que se utiliza para detectar bordes en imágenes. Se define mediante la siguiente máscara de convolución:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Este filtro resalta las áreas de la imagen donde hay cambios abruptos en la intensidad, lo que es útil para identificar bordes y contornos.

1.2. Filtro de Promedio

El filtro de promedio, también conocido como filtro de suavizado, se utiliza para reducir el ruido en una imagen. La máscara de convolución para este filtro es:

$$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Author for correspondence:
Cite this article:

Este filtro reemplaza cada píxel con el promedio de sus vecinos, lo que suaviza las variaciones rápidas en la imagen.

1.3. Filtro de la Derivada

El filtro de la derivada es un operador que calcula la tasa de cambio de la intensidad de la imagen. A diferencia de los filtros de convolución, este filtro no utiliza una máscara fija, sino que calcula la derivada en cada punto de la imagen y obtiene su magnitud. La derivada se puede aproximar utilizando diferencias finitas, como se muestra a continuación:

$$\frac{\partial I}{\partial x} \approx I(x,y) - I(x-1,y)$$

$$\frac{\partial I}{\partial y} \approx I(x,y) - I(x,y-1)$$

Este filtro es útil para detectar cambios rápidos en la intensidad de la imagen, similar al filtro de Laplace, pero con un enfoque diferente.

2. Método

La aplicación de los filtros de convolución se realiza iterativamente sobre cada píxel de la imagen. Para cada píxel, se extrae una submatriz de la imagen que corresponde al tamaño del filtro y se realiza la operación de convolución, la cual consiste en multiplicar elemento por elemento la submatriz con la máscara del filtro y sumar los resultados. Este proceso se repite para todos los píxeles de la imagen, asegurándose de manejar adecuadamente los bordes de la imagen. Por ejemplo, digamos que estamos sobre un pixel en la posición (i, j) de la imagen y queremos aplicar un filtro de 3×3. Extraemos la submatriz centrada en (i, j):

$$\begin{bmatrix} I(i-1,j-1) & I(i-1,j) & I(i-1,j+1) \\ I(i,j-1) & I(i,j) & I(i,j+1) \\ I(i+1,j-1) & I(i+1,j) & I(i+1,j+1) \end{bmatrix}$$

Luego, aplicamos la máscara del filtro, por ejemplo, el filtro de Laplace. La operación de convolución se realiza como sigue:

$$\begin{aligned}\text{Resultado}(i,j) = &(1 \cdot I(i-1,j)) + (1 \cdot I(i,j-1)) \\ &+ (-4 \cdot I(i,j)) + (1 \cdot I(i,j+1)) \\ &+ (1 \cdot I(i+1,j))\end{aligned}$$

Este valor se asigna al píxel (i, j) en la imagen resultante. Este proceso se repite para todos los píxeles de la imagen.

Las GPU's son especialmente adecuadas para este tipo de operaciones debido a su capacidad para manejar múltiples operaciones en paralelo. Al utilizar `CuPy`, se puede aprovechar la potencia de la GPU para acelerar significativamente el proceso de convolución en comparación con una implementación en CPU.

3. Detalles de la Implementación

En esta sección se describen las dos implementaciones principales desarrolladas para esta tarea, ambas utilizando GPU con CuPy: una función genérica para aplicar kernels de convolución arbitrarios y una función especializada para calcular la magnitud del gradiente.

3.1. Función para Aplicar Kernels de Convolución

La función `apply_kernel` implementa la convolución discreta en GPU utilizando CuPy y kernels CUDA personalizados. Esta función acepta como parámetros la ruta de una imagen y un kernel arbitrario representado como un array de NumPy. El proceso de implementación sigue los siguientes pasos:

1. **Carga y transferencia a GPU:** La imagen se carga utilizando la biblioteca PIL y se convierte a escala de grises. Posteriormente, tanto la imagen como el kernel se transfieren a la memoria de la GPU utilizando `cp.asarray`, convirtiéndolos a tipo `float32` para operaciones de punto flotante de precisión simple.
2. **Definición del kernel CUDA:** Se define un kernel CUDA mediante `cp.RawKernel` que implementa la operación de convolución. El kernel opera de la siguiente manera:
   - Cada thread procesa un píxel único, determinado por los índices de thread y bloque.
   - Para cada píxel (x, y), se calcula la convolución con el kernel:

     $$\text{Result}(x,y) = \sum_{k_y=0}^{n-1} \sum_{k_x=0}^{n-1} I(x+k_x-h, y+k_y-h) \cdot K(k_y,k_x)$$

     donde n es el tamaño del kernel, $h = \lfloor n/2 \rfloor$ es el radio del kernel y K representa el kernel.
   - Se ignoran los píxeles fuera de los límites de la imagen para manejar los bordes adecuadamente.
3. **Ejecución y post-procesamiento:** El kernel se lanza con los parámetros de la imagen, el kernel de convolución y las dimensiones necesarias. Al finalizar, se aplica `cp.nan_to_num` para asegurar valores finitos, se transfiere el resultado de vuelta a CPU usando `cp.asnumpy`, y se normaliza al rango $[0, 255]$ con conversión a `uint8`.

Esta implementación es general y puede aplicar cualquier kernel de convolución (Laplace, promedio, etc.).

3.2. Función para Calcular la Magnitud del Gradiente

La función `apply_magnitud` implementa el cálculo de la magnitud del gradiente utilizando un kernel CUDA personalizado, ejecutado en GPU mediante CuPy. Esta implementación aprovecha el paralelismo masivo de las GPU para acelerar significativamente el procesamiento. El proceso se describe a continuación:

1. **Carga y transferencia a GPU:** La imagen se carga en escala de grises y se convierte a un array de NumPy. Posteriormente, se transfiere a la memoria de la GPU utilizando `cp.asarray`, convirtiéndola a tipo `float32`.
2. **Definición del kernel CUDA:** Se define un kernel CUDA mediante `cp.RawKernel` que implementa el cálculo del gradiente. El kernel opera de la siguiente manera:
   - Cada thread procesa un píxel único, determinado por su índice de thread y bloque.
   - Para píxeles interiores (excluyendo la primera fila y columna), se calculan las derivadas parciales.
   - Se calcula la magnitud del gradiente.
   - Los píxeles en los bordes ($x = 0$ o $y = 0$) se establecen en cero para evitar accesos fuera de los límites.
3. **Ejecución y post-procesamiento:** El kernel se lanza con la configuración especificada. Al finalizar, se aplica `cp.nan_to_num` para asegurar valores finitos, se transfiere el resultado de vuelta a CPU usando `cp.asnumpy`, y se normaliza al rango $[0, 255]$ con conversión a `uint8`.

3.3. Comparación de Rendimiento

Se ejecutaron ambas implementaciones sobre la misma imagen de prueba de resolución 400×400 píxeles, repitiendo el proceso 10 veces para calcular el tiempo promedio de ejecución y reducir la variabilidad en las mediciones.

Para la implementación en GPU, se utilizaron `CUDA Events`, que son marcadores de tiempo nativos de CUDA que permiten medir con alta precisión el tiempo transcurrido en operaciones GPU. Para la implementación en CPU, se utilizó el módulo `time` de Python estándar.

Finalmente, se calcula el *speedup* como la relación entre el tiempo de ejecución en CPU y el tiempo en GPU:

$$\text{Speedup} = \frac{t_{\text{CPU}}}{t_{\text{GPU}}}$$

Este valor representa cuántas veces más rápida es la implementación en GPU comparada con la CPU. Un speedup mayor a 1 indica que la GPU es más rápida, mientras que valores cercanos o menores a 1 indicarían que no hay ventaja significativa (o incluso una desventaja) en usar GPU para esta tarea particular.

4. Resultados

4.1. Imagen de Prueba

Para probar las funciones implementadas, se utiliza una imagen de prueba (ver Figura 1). Esta imagen se carga y se procesa con cada uno de los filtros definidos y los resultados se comparan visualmente y en términos de rendimiento. Esta imagen tiene una resolución de 400×400 píxeles y contiene una variedad de detalles y texturas que permiten evaluar la efectividad de los filtros aplicados.

Figure 1. Imagen de prueba utilizada para aplicar los filtros de convolución y el filtro de la derivada.

4.2. Filtro de Laplace

En la figura 2 se muestra el resultado de aplicar el filtro de Laplace a la imagen de prueba. Este filtro resalta los bordes y contornos presentes en la imagen, facilitando la identificación de características importantes.

Figure 2. Resultado del filtro de Laplace aplicado a la imagen de prueba.

4.3. Filtro de Promedio

En la figura 3 se presenta el resultado del filtro de promedio aplicado a la imagen de prueba. Este filtro suaviza la imagen, reduciendo el ruido y las variaciones rápidas en la intensidad. En este caso, se observa un difuminado general de los detalles finos en la imagen original (ver Figura 1), por el efecto de promediado.

Figure 3. Resultado del filtro de promedio aplicado a la imagen de prueba.

4.4. Filtro de la Derivada

En la figura 4 se muestra el resultado del filtro de la derivada aplicado a la imagen de prueba. Este filtro destaca las áreas de cambio rápido en la intensidad, lo que muestra su utilidad para detectar bordes en imagenes. A comparación con el filtro de Laplace, ambos filtros resaltan bordes, pero se perciben más suavizados en el caso del filtro de Laplace. Además, el filtro de la derivada muestra ser más sensible al ruido, por lo que puede contener artefactos no deseados.

Figure 4. Resultado del filtro de la derivada aplicado a la imagen de prueba.

4.5. Comparación de Rendimiento

Para evaluar el rendimiento de las implementaciones en GPU, se midió el tiempo de ejecución para cada filtro aplicado a la imagen de prueba. Los resultados se resumen en la Tabla 1.

Table 1. Tiempos de ejecución para cada filtro aplicado a la imagen de prueba, comparando GPU y CPU.

Filtro	T. GPU (ms)	T. CPU (ms)	SpeedUp
Filtro de Laplace	5.43	4415	812.76
Filtro de Promedio	7.38	1451	196.53
Filtro de la Derivada	6.35	183.4	28.90

Se muestra que el filtro de Laplace obtiene la mayor aceleración, seguido por el filtro de promedio y finalmente el filtro de la derivada. Aún cuando se implementó la repetición de la ejecución 10 veces para obtener un promedio, se observó variabilidad en los tiempos medidos. Sin embargo, la tendencia general indica que la implementación en GPU ofrece una mejora significativa en la eficiencia con respecto a la CPU.

5. Conclusiones

Se concluye que la implementación de filtros de convolución en GPU ofrece una mejora significativa en el rendimiento en comparación con la CPU. Esto se debe a la naturaleza paralela de las operaciones en GPU, que permite procesar múltiples píxeles simultáneamente. Esto mismo aplica para el filtro de la derivada, que aunque no utiliza convolución, se beneficia igualmente del paralelismo de la GPU.

Además, los resultados visuales obtenidos con los diferentes filtros son consistentes con las expectativas teóricas. El filtro de Laplace es efectivo para resaltar bordes, el filtro de promedio suaviza la imagen y reduce el ruido, y el filtro de la derivada destaca cambios rápidos en la intensidad. La elección del filtro adecuado dependerá del objetivo específico del análisis de imágenes, y con ayuda de la GPU, estos procesos pueden realizarse de manera eficiente.

References

ROYUD Nishino and Shohei Hido Crissman Loomis. Cupy: A numpy-compatible library for nvidia gpu calculations. *31st confernce on neural information processing systems*, 151(7), 2017.