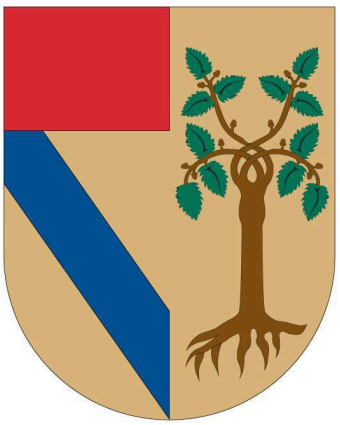
**COM152 Programación Avanzada En Paralelo: Examen Final**



UNIVERSIDAD
PANAMERICANA

# Paralelización del Algoritmo:

# “Filtro Gaussiano”


Alumnos: Pablo Raschid Llamas Aun (0238166)

Ángel Martínez Rodríguez (0235347)

Alberto Morales Vizcarra (0230866)

Edgar Velázquez Mercado (0217557)

Profesor: Dr. Germán Alonso Pinedo Díaz


Fecha de Entrega: Viernes 31 de mayo del 2024

Palabras: 1589

# 1. ALGORITMO FILTRO GAUSSIANO

El algoritmo del filtro gaussiano es una técnica utilizada en procesamiento de imágenes y visión por computadora para suavizar o desenfocar imágenes. Este filtro es conocido por sus propiedades que permiten una reducción de ruido en una imagen al tiempo que se preservan los bordes de los objetos en la misma. A continuación, se explica brevemente cómo funciona.

## 1.1. CONCEPTO BÁSICO

El filtro gaussiano aplica una transformación a cada píxel de la imagen usando una función gaussiana, que es una distribución de probabilidad normal en dos dimensiones. Esta función asigna un peso a cada píxel en función de su distancia al píxel central, con los píxeles más cercanos al centro recibiendo mayores pesos.

## 1.2. FUNCIÓN GAUSSIANA

La función gaussiana en dos dimensiones se define como:

$$G(x,y) = \frac{1}{2\pi\sigma^2} exp(-\frac{x^2+y^2}{2\sigma^2})$$

Donde:

- $(x, y)$ son las coordenadas del píxel respecto al centro del filtro.
- $\sigma$ es la desviación estándar de la distribución gaussiana, que controla la extensión del suavizado. Un $\sigma$ mayor produce un desenfoque más intenso.

## 1.3. APLICACIÓN DEL FILTRO

1. **Construcción del Kernel Gaussiano:** Se crea una matriz (kernel) basada en la función gaussiana, con el tamaño del kernel generalmente determinado por $\sigma$.
2. **Convolución:** Se realiza una operación de convolución de la imagen original con el kernel gaussiano. Esto implica superponer el kernel sobre cada píxel de la imagen, multiplicar los valores del kernel por los valores de los píxeles correspondientes y sumar estos productos para obtener el nuevo valor del píxel.

### 1.4. VENTAJAS

1. **Reducción de Ruido:** Suaviza la imagen reduciendo el ruido de alta frecuencia.
2. **Preservación de Bordes:** Menos agresivo que otros filtros de suavizado, lo que ayuda a mantener los bordes de los objetos más definidos.

# 2. NVIDIA CUDA C/C++

NVIDIA CUDA (Compute Unified Device Architecture) es una plataforma de computación paralela y un modelo de programación creado por NVIDIA que permite utilizar la potencia de procesamiento de las GPUs (Unidades de Procesamiento Gráfico) para realizar cálculos complejos de manera eficiente y rápida. CUDA proporciona extensiones al lenguaje de programación C/C++ para escribir código que se ejecute en la GPU.

### 2.1. CONCEPTOS BÁSICOS

- **Hilos y Bloques:** En CUDA, los cálculos se dividen en pequeños hilos. Los hilos se agrupan en bloques, y múltiples bloques forman una grid.
- **Memoria Compartida:** Los hilos dentro del mismo bloque pueden compartir datos a través de la memoria compartida, lo que permite un acceso rápido a los datos que necesitan ser reutilizados.
- **Kernels:** Las funciones que se ejecutan en la GPU se llaman kernels. Un kernel se lanza con una configuración de grid y bloques que determina el paralelismo.

### 2.2. BENEFICIOS DE UTILIZAR CUDA

- **Aceleración:** Las GPUs contienen miles de núcleos que pueden manejar múltiples hilos simultáneamente, proporcionando una aceleración significativa en el procesamiento de imágenes.
- **Escalabilidad:** El modelo de programación CUDA permite escalar fácilmente los cálculos a GPUs más potentes o múltiples GPUs.
- **Eficiencia Energética:** Las GPUs son más eficientes energéticamente para tareas de procesamiento paralelo en comparación con las CPUs tradicionales.

### 2.3. CONSIDERACIONES

- **Optimización de Memoria:** Es crucial optimizar el uso de la memoria compartida y global en la GPU para obtener el mejor rendimiento.

- **Sincronización de Hilos:** Asegurarse de que los hilos estén correctamente sincronizados cuando acceden a memoria compartida para evitar condiciones de carrera.
- **Ajuste de Parámetros:** Ajustar la configuración de grid y bloques según la arquitectura de la GPU utilizada.

# 3. PARALELIZAIÓN DEL ALGORITMO

La paralelización del filtro gaussiano es un proceso que permite acelerar su ejecución distribuyendo el trabajo entre múltiples unidades de procesamiento. Esto es especialmente útil para imágenes grandes o en aplicaciones que requieren procesamiento en tiempo real. Aquí se describe cómo se puede paralelizar el filtro gaussiano.

## 3.1. PASOS PARA LA PARALELIZACIÓN

1. División de la Imagen:
    - La imagen se divide en bloques o secciones más pequeñas. Cada bloque puede ser procesado independientemente por diferentes unidades de procesamiento (threads o núcleos de CPU/GPU).
2. Aplicación del Filtro en Paralelo:
    - Cada bloque se asigna a un thread o núcleo que aplica el filtro gaussiano a su sección de la imagen. Este proceso incluye la convolución de la sección de la imagen con el kernel gaussiano.
3. Manejo de Bordes:
    - Para asegurar que los bordes de los bloques sean procesados correctamente, es necesario considerar una superposición (overlap) entre los bloques. Esta superposición debe tener al menos el tamaño del radio del kernel gaussiano para garantizar que los píxeles en los bordes se suavicen adecuadamente.
4. Recomposición de la Imagen:
    - Una vez que todos los bloques han sido procesados, se recomponen para formar la imagen final suavizada.

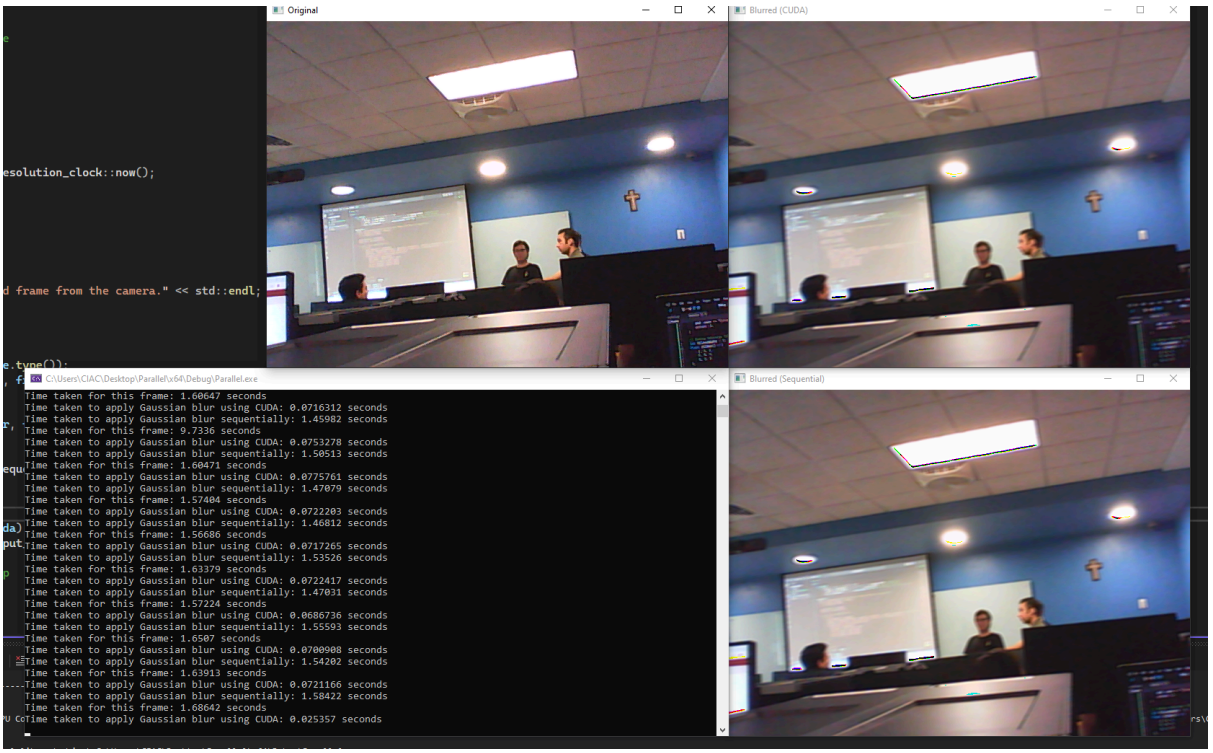
## 3.2. IMPLEMENTACIÓN EN CUDA C/C++

En esta sección, se explicará paso a paso cómo se implementó el filtro gaussiano paralelizado utilizando CUDA en C/C++, y cómo comparar su rendimiento con una implementación no paralelizada utilizando la librería ***'chrono'*** para medir el tiempo de ejecución. También se utilizará OpenCV para capturar los fotogramas de la cámara y aplicar el filtro en tiempo real.

1. **Captura de Video con OpenCV:**
    - Utilizamos ***'cv::VideoCapture'*** para capturar los frames de la cámara en tiempo real.
2. **Definición del Kernel Gaussiano:**
    - El kernel gaussiano se puede definir manualmente o utilizando funciones especializadas de OpenCV como ***'cv::getGaussianKernel'***.
3. **Kernel CUDA para el Desenfoque Gaussiano**
    - Define un kernel CUDA que aplica el filtro gaussiano a una imagen utilizando múltiples hilos de GPU para procesar cada píxel de la imagen de manera paralela.
4. **Aplicación del Desenfoque Gaussiano Utilizando CUDA**
    - Gestiona la memoria en la GPU, copia los datos de la imagen y el filtro a la GPU, lanza el kernel CUDA para realizar el desenfoque y luego copia el resultado de vuelta a la CPU.
5. **Aplicación del Desenfoque Gaussiano Secuencial**
    - Aplica el filtro gaussiano a la imagen de forma secuencial en la CPU, iterando sobre cada píxel y calculando el valor del desenfoque utilizando un enfoque de convolución.
6. **Comparación de Desempeño**
    - Se mide el tiempo de ejecución tanto para la versión paralelizada en CUDA como para la versión secuencial utilizando ***'std::chrono'***.

## 3.3. RESULTADOS DE LA EJECUCIÓN

A continuación se presentan los resultados de la ejecución del código de C/C++ que implementa el algoritmo del filtro gaussiano en secuencial y paralelizado con CUDA, respectivamente. En la venta de salida se puede la diferencia en tiempos de ejecución.




## 3.4. POSIBLES OPTIMIZACIONES

Para mejorar aún más el rendimiento del algoritmo de desenfoque gaussiano paralelizado en CUDA, se pueden considerar varias optimizaciones. A continuación se describen algunas de las técnicas más efectivas.

1. **Uso de Memoria Pinned (PIN Memory)**

   La memoria Pinned, o memoria de página bloqueada, permite transferencias de datos más rápidas entre la CPU y la GPU. Esto se debe a que la memoria Pinned no puede ser paginada a disco por el sistema operativo, lo que garantiza que las transferencias DMA (acceso directo a memoria) sean más rápidas y eficientes.

   - **Implementación:** Asignar memoria Pinned en el host para las imágenes de entrada y salida, y utilizar esta memoria para las transferencias entre la CPU y la GPU.

2. **Uso de Memoria Compartida (Shared Memory)**

   La memoria compartida en CUDA es una memoria muy rápida que está disponible para todos los hilos dentro del mismo bloque. Usar memoria compartida para almacenar el filtro gaussiano y los datos de la imagen puede reducir significativamente el número de accesos a la memoria global, mejorando así el rendimiento.

   - **Implementación:** Cargar las porciones de la imagen y el filtro en la memoria compartida antes de realizar las operaciones de convolución. Cada hilo puede colaborar para cargar estos datos de manera eficiente.

3. **Optimización de Tamaño de Bloque y Grid**

   El tamaño de bloque y la configuración de la grilla (grid) pueden tener un gran impacto en el rendimiento del kernel CUDA. Es importante encontrar un equilibrio adecuado entre la ocupación de la GPU y el uso eficiente de los recursos disponibles.

   - **Implementación:** Experimentar con diferentes tamaños de bloques y configuraciones de grid para encontrar la combinación que maximice la ocupación y el rendimiento de la GPU.

# 4. CONCLUSIÓN

La implementación del filtro gaussiano mediante la paralelización utilizando la plataforma CUDA de NVIDIA demuestra mejoras significativas en el procesamiento de imágenes. Este filtro, fundamental para suavizar y reducir el ruido en imágenes, preservando los bordes, aplica una función gaussiana a cada píxel, asignando pesos según la proximidad al centro del filtro. CUDA, al extender el lenguaje C/C++, permite ejecutar cálculos paralelos de manera eficiente, aprovechando la capacidad de las GPUs para acelerar significativamente tareas de procesamiento de imágenes en comparación con métodos secuenciales.

En el proceso de paralelización del filtro gaussiano, se dividió la imagen en bloques, procesados en paralelo, lo que redujo drásticamente el tiempo de ejecución. La implementación incluyó la captura de video en tiempo real con OpenCV, la definición y aplicación del kernel CUDA, y una comparación de rendimiento con una versión secuencial utilizando la librería chrono. Los resultados mostraron una mejora notable en el tiempo de ejecución con la versión paralelizada en CUDA.

Adicionalmente, se identificaron varias optimizaciones posibles para mejorar aún más el rendimiento, como el uso de memoria Pinned para transferencias de datos más rápidas entre la CPU y la GPU, y el empleo de memoria compartida para reducir accesos a la memoria global. Ajustar el tamaño de los bloques y la configuración de la grilla puede maximizar la eficiencia y ocupación de la GPU. En resumen, la implementación del filtro gaussiano con CUDA no solo ilustra las ventajas de la computación paralela, sino que también destaca el potencial para optimizaciones adicionales que pueden llevar a un rendimiento aún mejor, haciendo más eficientes las aplicaciones que requieren procesamiento de imágenes en tiempo real.