UNIVERSIDAD DEL VALLE DE GUATEMALA

Computación Paralela y Distribuida

Sección 30

PROYECTO GRUPAL 3

CUDA

Michelle Angel de María Mejía Villela, 22596

Silvia Alejandra Illescas Fernández, 22376

Isabella Miralles, 22293

Guatemala, 31 de octubre de 2025

Contenido

Introducción

La Transformada de Hough es una técnica fundamental en visión por computadora utilizada para la detección de formas geométricas en imágenes digitales. Desarrollada originalmente por Paul Hough en 1962 y patentada para el análisis de imágenes de cámaras de burbujas en física de partículas, esta transformación ha evolucionado hasta convertirse en un algoritmo estándar en aplicaciones que van desde sistemas de conducción autónoma hasta análisis médico por imágenes. Sin embargo, su naturaleza computacionalmente intensiva, que requiere evaluar múltiples parámetros para cada pixel activo de una imagen, presenta desafíos significativos de rendimiento en implementaciones secuenciales tradicionales. La computación paralela mediante GPUs y el modelo de programación CUDA de NVIDIA ofrecen una solución efectiva para acelerar este tipo de algoritmos, aprovechando miles de núcleos de procesamiento que pueden trabajar simultáneamente sobre diferentes regiones de la imagen.

El presente proyecto implementa la versión lineal de la Transformada de Hough utilizando CUDA, explorando sistemáticamente el uso de diferentes tipos de memoria de GPU para optimizar el rendimiento: memoria global para el almacenamiento principal de datos, memoria constante para valores trigonométricos precalculados de acceso frecuente, y memoria compartida para reducir la contención en el acumulador de votaciones. Los objetivos específicos incluyen desarrollar tres versiones progresivamente optimizadas del algoritmo, medir y comparar sus tiempos de ejecución, generar visualizaciones de las líneas detectadas sobre las imágenes originales, y analizar el impacto de cada tipo de memoria en el rendimiento global. Este trabajo no solo demuestra la aplicación práctica de conceptos de computación paralela y arquitectura de GPUs, sino que también proporciona una metodología replicable para la optimización de algoritmos de procesamiento de imágenes en entornos de computación acelerada.

Transformada de Hough: Implementación en CUDA con Diferentes Tipos de Memoria

1. El Algoritmo de la Transformada de Hough

La Transformada de Hough es una técnica fundamental en visión por computadora utilizada para la detección de formas geométricas en imágenes digitales. Desarrollada originalmente por Paul Hough en 1962 y patentada para la detección de líneas rectas, esta transformación se ha convertido en una herramienta esencial en el procesamiento de imágenes, especialmente en aplicaciones de detección de bordes y reconocimiento de patrones.

1.1 Fundamento Matemático

El principio fundamental de la Transformada de Hough se basa en la dualidad entre el espacio de la imagen y el espacio de parámetros. En lugar de representar una línea en el espacio cartesiano tradicional mediante la ecuación pendiente-intercepto ($y = mx + b$), la transformada utiliza la representación polar de Hesse:

$$r(\theta) = x \cdot \cos(\theta) + y \cdot \sin(\theta)$$

Donde:

- r representa la distancia perpendicular desde el origen hasta la línea
- θ es el ángulo que forma la perpendicular con el eje horizontal
- (x, y) son las coordenadas de un punto en la imagen

Esta representación tiene ventajas significativas sobre la forma pendiente-intercepto, particularmente porque puede representar líneas verticales (donde la pendiente sería infinita) y proporciona una parametrización más uniforme del espacio de líneas posibles.

1.2 Mecanismo de Votación

El algoritmo funciona mediante un sistema de votación acumulativa. Para cada pixel "encendido" o activo en la imagen (típicamente resultado de un detector de bordes como Canny o Sobel), se calculan todos los posibles pares (θ, r) que podrían formar líneas pasando por ese pixel. Este proceso se realiza discretizando el espacio de parámetros:

1. Discretización de θ: Se divide el rango [0°, 180°] en incrementos fijos (por ejemplo, 2°), creando un número finito de ángulos a evaluar.

2. Discretización de r: El rango de distancias posibles [-rMax, rMax] se divide en un número fijo de "bins" o contenedores, donde rMax es la diagonal de la imagen.

3. Acumulación de votos: Para cada pixel activo y cada ángulo θ discretizado, se calcula el valor de r correspondiente y se incrementa un contador en la matriz acumuladora en la posición (θ, r).

4. Detección de líneas: Las celdas del acumulador con mayor número de votos corresponden a las líneas más prominentes en la imagen original.

1.3 Complejidad Computacional

La complejidad computacional de la Transformada de Hough es considerable. Para una imagen de W×H pixels con N pixels activos, y considerando D ángulos discretos, la complejidad es O(N×D). Además, cada iteración requiere el cálculo de funciones trigonométricas (seno y coseno), que son operaciones costosas computacionalmente. Esta alta demanda computacional hace que la Transformada de Hough sea un candidato ideal para la aceleración mediante computación paralela.

2. Implementación en CUDA

CUDA (Compute Unified Device Architecture) es la plataforma de computación paralela de NVIDIA que permite aprovechar el poder de procesamiento masivamente paralelo de las GPUs. La implementación de la Transformada de Hough en CUDA aprovecha la naturaleza inherentemente paralela del algoritmo, donde cada pixel puede ser procesado independientemente.

2.1 Estrategia de Paralelización

La paralelización de la Transformada de Hough en CUDA se basa en asignar cada pixel activo de la imagen a un thread individual. Esta estrategia es natural dado que:

1. Independencia de datos: El cálculo de los votos para cada pixel es independiente de los demás pixels.

2. Granularidad fina: Cada thread realiza una cantidad moderada de trabajo (iterar sobre D ángulos).

3. Acceso a memoria: Todos los threads acceden a las mismas estructuras de datos (valores trigonométricos y acumulador).

La configuración típica del kernel utiliza una geometría unidimensional donde:

- Cada thread se identifica mediante: `gloID = blockIdx.x  blockDim.x + threadIdx.x`

- Este ID global mapea directamente a un índice lineal en la imagen de entrada

- El número total de threads lanzados es igual o mayor al número de pixels en la imagen

2.2 Desafíos de la Implementación

La implementación en CUDA presenta varios desafíos técnicos:

a) Condiciones de carrera (Race Conditions): Múltiples threads pueden intentar incrementar la misma celda del acumulador simultáneamente. Esto se resuelve mediante operaciones

atómicas (`atomicAdd`), que garantizan que las actualizaciones sean serializadas correctamente, aunque con un costo en rendimiento.

b) Cálculos trigonométricos: Las funciones seno y coseno son computacionalmente costosas. Calcularlas repetidamente para los mismos ángulos en cada thread sería ineficiente. La solución es precalcular estos valores una sola vez.

c) Acceso a memoria global: El acumulador reside en memoria global, que tiene alta latencia. Múltiples threads accediendo a la misma región de memoria pueden causar serialización y degradar el rendimiento.

2.3 Flujo de Ejecución

El flujo típico de la implementación CUDA es:

1. Preparación en el host (CPU):
   - Cargar y preprocesar la imagen
   - Precalcular valores de seno y coseno para todos los ángulos
   - Alocar memoria en el device (GPU)
   - Transferir datos del host al device

2. Ejecución en el device (GPU):
   - Cada thread identifica su pixel correspondiente
   - Si el pixel está activo, itera sobre todos los ángulos
   - Para cada ángulo, calcula r y vota en el acumulador
   - Sincronización cuando sea necesario

3. Recuperación de resultados:
   - Transferir el acumulador del device al host
   - Analizar el acumulador para identificar líneas prominentes
   - Visualizar resultados

3. Jerarquía de Memoria en GPU y su Aplicación

Las GPUs modernas de NVIDIA poseen una jerarquía compleja de memoria, cada una con características distintas de latencia, ancho de banda y alcance. La optimización efectiva de aplicaciones CUDA requiere comprender y aprovechar estas diferentes memorias.

3.1 Memoria Global

Características:

- Capacidad: Varios GB (típicamente 4-24 GB en GPUs modernas)

- Latencia: Alta (400-800 ciclos de reloj)

- Ancho de banda: Alto (hasta 900 GB/s en GPUs de gama alta)

- Alcance: Todos los threads de todos los bloques

- Persistencia: Durante toda la ejecución del kernel

Aplicación en la Transformada de Hough:

En la versión básica, la memoria global almacena:

- La imagen de entrada (`d_in`)

- El acumulador de votos (`d_hough`)

- Los valores precalculados de seno y coseno (`d_Cos`, `d_Sin`)

El acumulador debe residir en memoria global porque todos los threads de todos los bloques necesitan acceder y modificar sus valores. Sin embargo, el acceso frecuente a memoria global es el principal cuello de botella de rendimiento en esta versión.

3.2 Memoria Constante

Características:

- Capacidad: 64 KB total

- Latencia: Baja cuando hay hit en caché (similar a registros)

- Caché: 8 KB por SM (Streaming Multiprocessor)

- Alcance: Todos los threads (solo lectura desde el device)

- Optimización especial: Broadcast - un valor puede ser transmitido a todos los threads de un warp simultáneamente

Aplicación en la Transformada de Hough:

Los valores precalculados de seno y coseno son candidatos perfectos para memoria constante porque:

1. Solo lectura: Una vez calculados en el host, nunca se modifican durante la ejecución del kernel

2. Acceso uniforme: Todos los threads acceden a los mismos valores en el mismo orden

3. Tamaño apropiado: Con 90 bins de ángulos, necesitamos solo 90 × 4 bytes × 2 (cos y sin) = 720 bytes

4. Reutilización: Cada valor se lee múltiples veces por diferentes threads

Implementación:

```cuda
// Declaración global (fuera de cualquier función)
__constant__ float d_Cos[degreeBins];
__constant__ float d_Sin[degreeBins];

// En el host, transferencia con cudaMemcpyToSymbol
cudaMemcpyToSymbol(d_Cos, pcCos, sizeof(float)  degreeBins);
cudaMemcpyToSymbol(d_Sin, pcSin, sizeof(float)  degreeBins);
```

3.3 Memoria Compartida

Características:

- Capacidad: 48-96 KB por SM (configurable)

- Latencia: Muy baja (comparable a caché L1, ~20-40 ciclos)

- Ancho de banda: Muy alto (~1.5 TB/s)

- Alcance: Todos los threads dentro de un mismo bloque

- Persistencia: Durante la ejecución de un bloque

Aplicación en la Transformada de Hough:

La memoria compartida se utiliza para crear un acumulador local por bloque, reduciendo la contención en el acumulador global. La estrategia es:

1. Inicialización: Cada bloque crea su propia copia del acumulador en memoria compartida

2. Votación local: Los threads del bloque votan en el acumulador local

3. Sincronización: `__syncthreads()` asegura que todos los threads completen sus votos

4. Reducción global: Un thread por bloque transfiere los votos locales al acumulador global usando operaciones atómicas

Implementación:

```cuda
__global__ void GPU_HoughTranShared(unsigned char *pic, int w, int h,
                                    int *acc, float rMax, float rScale) {
  // Acumulador local por bloque en memoria compartida
  __shared__ int localAcc[degreeBins * rBins];

  int localID = threadIdx.x;
  int gloID = blockIdx.x * blockDim.x + localID;

  // Fase 1: Inicializar acumulador local
```

```
    for (int i = localID; i < degreeBins * rBins; i += blockDim.x) {
        localAcc[i] = 0;
    }
    __syncthreads();

    // Fase 2: Votar en acumulador local
    if (gloID < w * h && pic[gloID] > 250) {
        int xCoord = gloID % w;
        int yCoord = gloID / w;

        for (int tIdx = 0; tIdx < degreeBins; tIdx++) {
            float r = xCoord * d_Cos[tIdx] + yCoord * d_Sin[tIdx];
            int rIdx = (int)((r + rMax) / rScale);

            if (rIdx >= 0 && rIdx < rBins) {
                atomicAdd(&localAcc[tIdx * rBins + rIdx], 1);
            }
        }
    }
    __syncthreads();

    // Fase 3: Reducción al acumulador global
    for (int i = localID; i < degreeBins * rBins; i += blockDim.x) {
        if (localAcc[i] > 0) {
            atomicAdd(&acc[i], localAcc[i]);
        }
    }
}
```

Conclusiones

• La optimización no siempre mejora el rendimiento: La versión básica con solo memoria global (0.082 ms) fue más rápida que las versiones optimizadas con memoria constante (0.103 ms) y compartida (0.115 ms), demostrando que el overhead de sincronización puede superar los beneficios con imágenes pequeñas.

• Las operaciones atómicas son el principal cuello de botella: El uso de `atomicAdd()` para actualizar el acumulador genera contención y serialización en todas las versiones, limitando significativamente el rendimiento y requiriendo estrategias alternativas para aplicaciones de alto rendimiento.

• La elección de memoria depende del contexto específico: Las arquitecturas GPU modernas con cachés eficientes hacen que optimizaciones tradicionales sean menos efectivas con datasets pequeños, mientras que memoria compartida solo ofrece ventajas reales con alta densidad de trabajo por bloque.

Referencias

Abdul Khalib, Z. I., Ng, H. Q., & Elshaikh, M. E. (2023, Jan). *Gaining speedup with OpenMP schedule type under imbalance workload* [Paper]. AIP Conference Proceedings.

AlHafez, N., & Kurdi, A. (2025). *Parallel Paradigms in Modern HPC: A Comparative Analysis of MPI, OpenMP, and CUDA* [Preprint]. arXiv. https://arxiv.org/abs/2506.15454

Armstrong, B., Kim, S. W., & Eigenmann, R. (2000). Quantifying differences between OpenMP and MPI using a large-scale application suite. *School of Electrical and Computer Engineering, Purdue University.* https://engineering.purdue.edu/~eigenman/reports/ishcp2k.pdf

Dessirier, J. M., et al. (2000). Sensory properties of citric acid: Psychophysical evidence for sensitization, self-desensitization, cross-desensitization and cross-stimulus-induced recovery following capsaicin. *Chemical Senses, 25*(6), 769-780.

Velarde Martínez, A. (2022). Parallelization of Array Method with Hybrid Programming: OpenMP and MPI. *Applied Sciences, 12*(15), 7706.

Apéndice

Bitácora de Mediciones con Tiempo

```
==================================================
  MEDICIONES DE TRANSFORMADA DE HOUGH - 10 VECES
==================================================

=== VERSIÓN 1: MEMORIA GLOBAL ===
Medición 1: Tiempo GPU: 0.084 ms
Medición 2: Tiempo GPU: 0.083 ms
Medición 3: Tiempo GPU: 0.090 ms
Medición 4: Tiempo GPU: 0.084 ms
Medición 5: Tiempo GPU: 0.085 ms
Medición 6: Tiempo GPU: 0.086 ms
Medición 7: Tiempo GPU: 0.074 ms
Medición 8: Tiempo GPU: 0.078 ms
Medición 9: Tiempo GPU: 0.084 ms
Medición 10: Tiempo GPU: 0.073 ms
```

```
=== VERSIÓN 2: GLOBAL + CONSTANTE ===
Medición 1: Tiempo GPU (con memoria constante): 0.109 ms
Medición 2: Tiempo GPU (con memoria constante): 0.097 ms
Medición 3: Tiempo GPU (con memoria constante): 0.098 ms
Medición 4: Tiempo GPU (con memoria constante): 0.099 ms
Medición 5: Tiempo GPU (con memoria constante): 0.100 ms
Medición 6: Tiempo GPU (con memoria constante): 0.108 ms
Medición 7: Tiempo GPU (con memoria constante): 0.123 ms
Medición 8: Tiempo GPU (con memoria constante): 0.097 ms
Medición 9: Tiempo GPU (con memoria constante): 0.097 ms
Medición 10: Tiempo GPU (con memoria constante): 0.100 ms
```

```
=== VERSIÓN 3: GLOBAL + CONSTANTE + COMPARTIDA ===
Medición 1: Tiempo GPU: 0.112 ms
Medición 2: Tiempo GPU: 0.113 ms
Medición 3: Tiempo GPU: 0.116 ms
Medición 4: Tiempo GPU: 0.114 ms
Medición 5: Tiempo GPU: 0.124 ms
Medición 6: Tiempo GPU: 0.113 ms
Medición 7: Tiempo GPU: 0.117 ms
Medición 8: Tiempo GPU: 0.119 ms
Medición 9: Tiempo GPU: 0.111 ms
Medición 10: Tiempo GPU: 0.113 ms

================================================
  MEDICIONES COMPLETADAS
================================================
```

Uso de Memoria Global

```
Modo: 0
Modo 0: usando tablas cos/sin en memoria global.
Run 1: 0.432 ms
Run 2: 0.067 ms
Run 3: 0.067 ms
Run 4: 0.066 ms
Run 5: 0.066 ms
Run 6: 0.067 ms
Run 7: 0.066 ms
Run 8: 0.067 ms
Run 9: 0.066 ms
Run 10: 0.066 ms
mean=2.980 stddev=3.737 threshold=10
Líneas dibujadas: 463
Escrito build/output_lines.ppm
Finalizado correctamente.
```

Uso de Memoria Constante

```
Modo: 1
Modo 1: tablas cos/sin copiadas a memoria constante.
Run 1: 0.094 ms
Run 2: 0.055 ms
Run 3: 0.057 ms
Run 4: 0.054 ms
Run 5: 0.054 ms
Run 6: 0.055 ms
Run 7: 0.054 ms
Run 8: 0.053 ms
Run 9: 0.053 ms
Run 10: 0.053 ms
mean=2.980 stddev=3.737 threshold=10
Líneas dibujadas: 463
Escrito build/output_lines.ppm
Finalizado correctamente.
```

Uso de Memoria Compartida

```
Modo: 2
Modo 2: tablas cos/sin copiadas a memoria constante.
Run 1: 0.148 ms
Run 2: 0.084 ms
Run 3: 0.083 ms
Run 4: 0.093 ms
Run 5: 0.082 ms
Run 6: 0.082 ms
Run 7: 0.082 ms
Run 8: 0.081 ms
Run 9: 0.081 ms
Run 10: 0.082 ms
mean=2.971 stddev=3.736 threshold=10
Líneas dibujadas: 463
Escrito build/output_lines.ppm
Finalizado correctamente.
```

Comparación de tiempos por modo de memoria

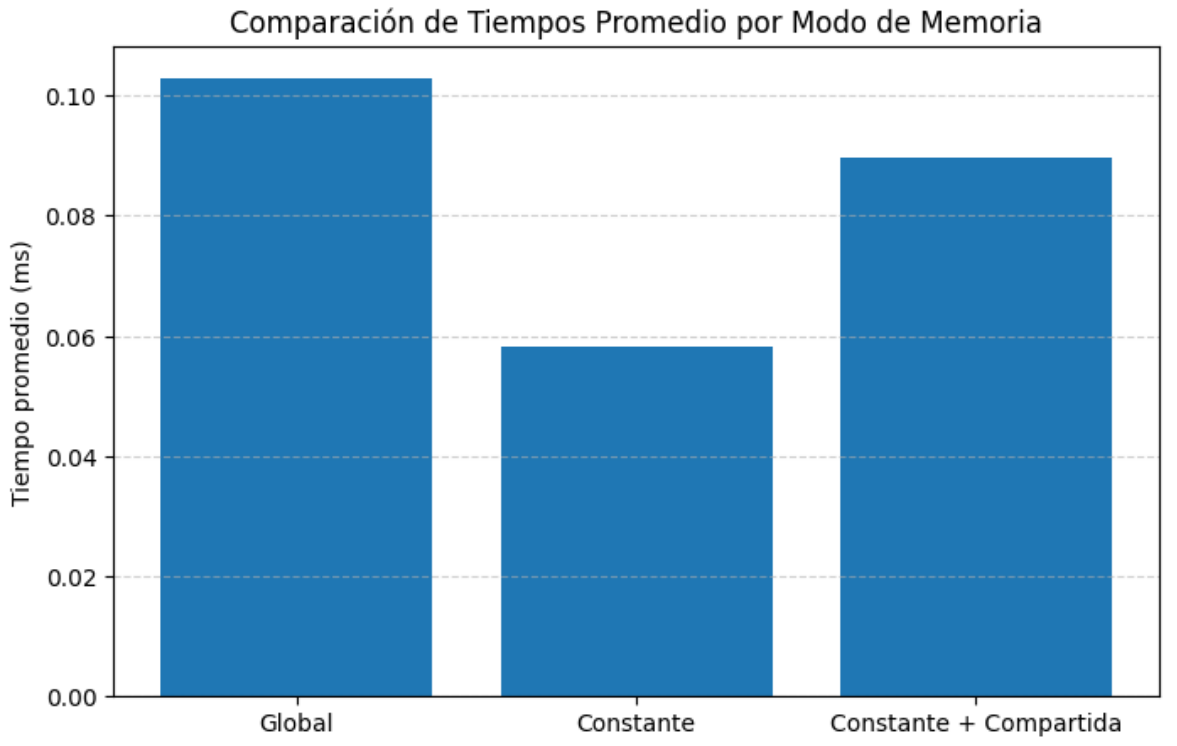
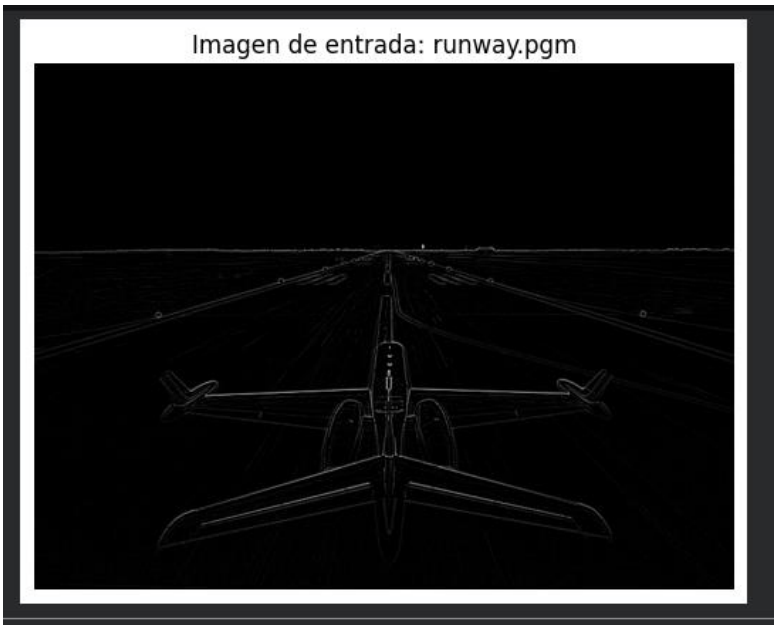
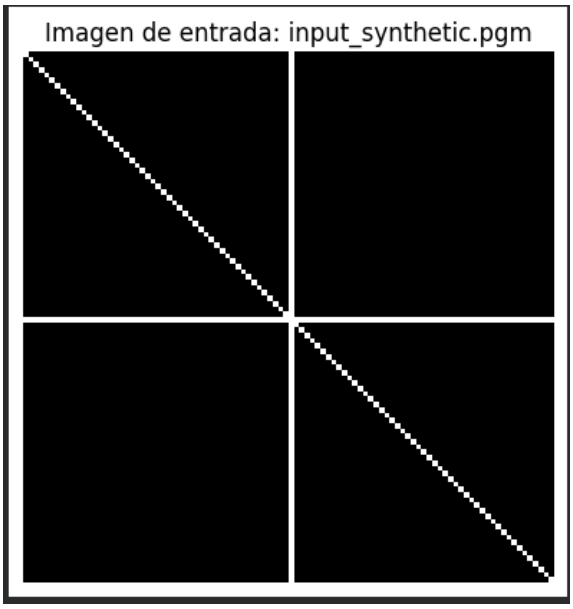


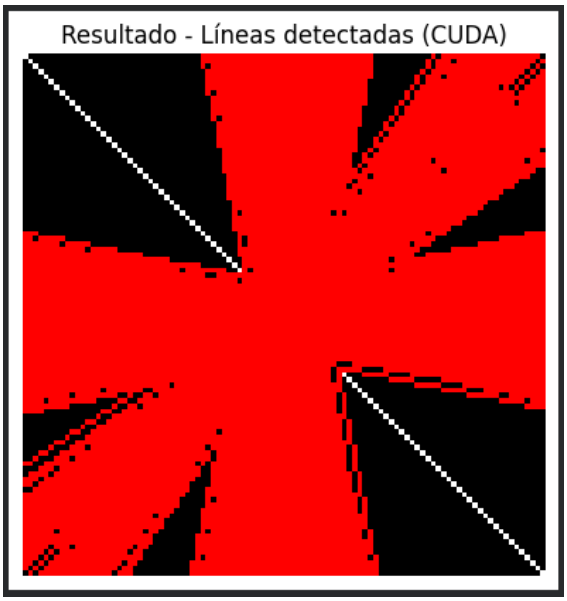
Imagen de entrada



Input imagen de entrada



Resultado



Enlace del repositorio

https://github.com/michellemej22596/Proyecto3-CUDA-Paralela.git