

Universidad de Alcalá de Henares

Escuela Politécnica Superior

# **Paradigmas Avanzados de Programación**

PL1 - Manipulación de Imágenes

**Autores:**

Luciana Paola Diaz

**Fecha:**

28 de marzo de 2025

# Índice

<b>1. Introducción</b>	<b>2</b>
<b>2. Desarrollo</b>	<b>2</b>
2.1. Fase 01. Conversión a blanco y negro . . . . .	2
2.2. Filtro pixelado cristalizado . . . . .	4
2.3. Fase 03. Identificación de colores (Rojo, Verde y Azul) . . . . .	6
2.4. Fase 04. Halo sobre color objetivo en blanco y negro . . . . .	8
2.5. Fase 05. Reducción paralela de imagen RGB y generación de hash ASCII .	10
<b>3. Conclusión</b>	<b>12</b>

# 1. Introducción

El presente documento describe el desarrollo e implementación de un conjunto de cinco fases de procesamiento de imágenes en formato BMP mediante el uso de programación paralela con CUDA. El objetivo general del proyecto es demostrar el uso eficiente de la GPU para acelerar operaciones gráficas intensivas, como la conversión a escala de grises, aplicación de filtros de pixelado, identificación de colores específicos, delineado con halo y generación de un hash visual a partir de la imagen.

Para lograr una implementación robusta, se ha seguido una arquitectura modular que permite lanzar cada fase de forma independiente desde un menú gráfico interactivo. Todas las fases han sido diseñadas para adaptarse dinámicamente a las características del hardware CUDA, cumpliendo con los requerimientos indicados en el enunciado oficial de la práctica PECL1.

Cada fase se ha construido utilizando kernels CUDA optimizados y técnicas como memoria compartida, uso de memoria constante, conteos atómicos y estrategias de reducción jerárquica. Además, se ha ofrecido flexibilidad al usuario permitiendo la configuración de parámetros como el tamaño del filtro, los umbrales de color o el tamaño del halo desde la terminal.

A continuación, se describe el funcionamiento de cada fase, incluyendo su estructura técnica, los cálculos realizados y los resultados obtenidos sobre diferentes imágenes de prueba.

## 2. Desarrollo

### 2.1. Fase 01. Conversión a blanco y negro

El objetivo principal de esta fase es convertir una imagen en color a escala de grises, generando una salida en blanco y negro. Para ello, se hace uso del kernel `--global__ GrayscaleKernel()` que ejecuta la transformación de cada píxel RGB al formato de grises de manera paralela en la GPU.

En primer lugar, se leen las propiedades de la GPU mediante la función `cudaGetDeviceProperties()`, concretamente el número máximo de hilos por bloque (`maxThreadsPerBlock`). A partir de este valor, se calcula dinámicamente el tamaño de bloque como una cuadrícula cuadrada de lado `sqrt(maxThreadsPerBlock)`, garantizando así que el lanzamiento del kernel sea portable a cualquier GPU.

La configuración del grid también se calcula de forma dinámica usando:

$$\text{gridSize.x} = \left\lceil \frac{\text{ancho}}{\text{blockSize.x}} \right\rceil, \quad \text{gridSize.y} = \left\lceil \frac{\text{alto}}{\text{blockSize.y}} \right\rceil$$

De este modo, se asegura que toda la imagen queda cubierta por los hilos lanzados, sin dejar píxeles sin procesar.

Cada hilo calcula su coordenada global (`x, y`) y, si se encuentra dentro de los límites de la imagen, accede al píxel correspondiente y lo convierte a gris mediante la siguiente fórmula, basada en la percepción visual humana:

$$\text{gray} = 0,299 \cdot R + 0,587 \cdot G + 0,114 \cdot B$$

Este valor se asigna a los tres canales del píxel, produciendo así un tono uniforme y generando una imagen en blanco y negro.

El flujo completo de la fase incluye:

- Lectura de la imagen BMP mediante `ReadImage()`.
- Asignación de memoria en el dispositivo con `cudaMalloc()`.
- Copia de los datos al dispositivo con `cudaMemcpy()`.
- Ejecución del kernel con la configuración dinámica mencionada.
- Copia del resultado final al host.
- Escritura de la imagen de salida usando `WriteImage()`.

```
===== MENÚ =====
Imagen cargada: input.bmp (1280x962)
1. Filtro Blanco y Negro
2. Filtro Pixelado
3. Identificación de Colores
4. Delineado con Halo
5. Pseudo-Hash
6. Cambiar imagen base
0. Salir
=====
Seleccione una opción: 1
Proceso completado. Imagen guardada como 'outputBlackAndWhite.bmp'.
```

Figura 1: Menú implementando la Fase 1



Figura 2: Imagen convertida a blanco y negro en la Fase 1

## 2.2. Fase 02. Filtro pixelado cristalizado

En esta fase, se aplica un efecto de pixelado sobre la imagen utilizando un enfoque llamado *filtro cristalizado*. A diferencia de un filtro de promedio, este efecto reemplaza cada región de la imagen por el color del píxel central del bloque.

El filtro se ejecuta sobre bloques de tamaño definido por el parámetro `filterSize`, que representa el tamaño del vecindario cuadrado. Cada bloque de hilos cubre una región de `filterSize × filterSize` píxeles, y en lugar de calcular el promedio, todos los píxeles de esa región son reemplazados por el color del píxel central del bloque.

El kernel `__global__ PixelateCrystalKernel()` hace uso de memoria compartida para cargar localmente los píxeles del bloque, lo cual mejora el rendimiento al reducir accesos a memoria global. Solo el hilo  $(0,0)$  del bloque escribe los resultados de salida, copiando el color del centro del bloque a toda la región cubierta.

Antes de lanzar el kernel, el programa obtiene las propiedades del dispositivo mediante `cudaGetDeviceProperties()`, particularmente la cantidad máxima de memoria compartida por bloque. Si el valor de `filterSize` solicitado supera esta capacidad, se reduce automáticamente hasta ajustarse, garantizando portabilidad y eficiencia.

La configuración del grid y bloque también se determina dinámicamente con:

$$\text{gridSize.x} = \left\lceil \frac{\text{ancho}}{\text{filterSize}} \right\rceil, \quad \text{gridSize.y} = \left\lceil \frac{\text{alto}}{\text{filterSize}} \right\rceil$$

Esto asegura que cada región de la imagen sea procesada correctamente, incluso si la imagen no es divisible exactamente por el tamaño del filtro.

En el flujo de ejecución desde el host se incluyen los siguientes pasos:

- Lectura de la imagen BMP original.
- Asignación dinámica de memoria en el dispositivo.
- Ajuste dinámico del tamaño de bloque según `filterSize` y la memoria compartida disponible.
- Ejecución del kernel con la configuración calculada.
- Copia del resultado final de vuelta al host.
- Guardado de la imagen pixelada en un nuevo archivo BMP.

Este efecto produce una imagen con aspecto artístico o de censura pixelada, cuya intensidad depende del valor de `filterSize`: a mayor valor, más notorio será el efecto de bloque.

```
===== MENÚ =====
Imagen cargada: input.bmp (1280x962)
1. Filtro Blanco y Negro
2. Filtro Pixelado
3. Identificación de Colores
4. Delineado con Halo
5. Pseudo-Hash
6. Cambiar imagen base
0. Salir
=====
Seleccione una opción: 2
¿Desea aplicar el filtro pixelado sobre la imagen (1) en color o (2) en blanco y negro?: 1
Introduce el tamaño del filtro de pixelado (por ejemplo, 15 o 30): 5
Proceso completado. Imagen guardada como 'outputPixelateCrystal.bmp'.
```

Figura 3: Menú implementando la fase 2 en color



Figura 4: Pixelado con rango 5



Figura 5: Pixelado con rango 15



Figura 6: Pixelado con rango 30

Se observa que al aumentar el rango, el efecto cristalizado es más evidente, ya que bloques mayores implican más píxeles siendo sustituidos por un único color.

También es posible aplicar el filtro sobre una imagen previamente convertida a escala de grises, generando un efecto en blanco y negro.

```
===== MENÚ =====
Imagen cargada: fast.bmp (1200x794)
1. Filtro Blanco y Negro
2. Filtro Pixelado
3. Identificación de Colores
4. Delineado con Halo
5. Pseudo-Hash
6. Cambiar imagen base
0. Salir
=====
Seleccione una opción: 2
¿Desea aplicar el filtro pixelado sobre la imagen (1) en color o (2) en blanco y negro?: 2
Introduce el tamaño del filtro de pixelado (por ejemplo, 15 o 30): 15
Proceso completado. Imagen guardada como 'outputPixelateCrystal.bmp'.
```

Figura 7: Menú implementando la fase 2 en blanco y negro



Figura 8: Pixelado con rango 15, blanco y negro



Figura 9: Pixelado con rango 15, blanco y negro optimizado

### 2.3. Fase 03. Identificación de colores (Rojo, Verde y Azul)

El objetivo de esta fase es identificar los píxeles que pertenecen a zonas con color rojo, verde o azul, según unos umbrales definidos. Estos píxeles se conservan en imágenes separadas, mientras que los píxeles que no cumplen con el umbral son coloreados de blanco, destacando visualmente solo las regiones detectadas.

Para esta tarea, se define una estructura `ColorRange` que contiene seis enteros: los valores mínimo y máximo permitidos para cada canal RGB. Estos valores están definidos en memoria constante en la GPU y permiten evaluar si un píxel pertenece o no a un determinado color.

Por defecto, se utilizan los siguientes rangos:

Color	R (mín,máx)	G (mín,máx)	B (mín,máx)
Rojo	100,255	0,150	0,150
Verde	30,150	50,255	0,75
Azul	0,200	0,250	100,255

Cuadro 1: Rangos por defecto utilizados para el filtrado de color

No obstante, el usuario puede modificar estos umbrales desde la terminal cuando se ejecuta la fase. El programa pregunta por cada parámetro si se desea usar valores personalizados o continuar con los predefinidos.

El kernel `__global__ ClassifyAndFilterColors()` es el encargado de realizar esta detección. Cada hilo accede a un píxel mediante sus coordenadas globales (`x, y`), y compara los valores RGB del píxel con los rangos definidos en memoria constante. Si el píxel entra dentro del rango de un color, se copia a su imagen correspondiente (rojo, verde o azul). Si no, el píxel en esa imagen se marca como blanco ((255, 255, 255)). Esta operación se realiza de forma paralela, asignando un hilo por píxel.

Además, se usan contadores atómicos para registrar cuántos píxeles han sido clasificados como rojo, verde o azul. Estos valores se muestran por pantalla tras la ejecución de la fase.

Desde el `host`, se configuran dinámicamente los parámetros de ejecución. Se consulta el número máximo de hilos por bloque con `cudaGetDeviceProperties()`, y a partir de eso se calcula un `blockSize` cuadrado (`sqrt(maxThreadsPerBlock)`), ajustando el `gridSize` para cubrir toda la imagen:

$$\text{gridSize.x} = \left\lceil \frac{\text{ancho}}{\text{blockSize.x}} \right\rceil, \quad \text{gridSize.y} = \left\lceil \frac{\text{alto}}{\text{blockSize.y}} \right\rceil$$

Una vez finalizado el procesamiento en GPU, se copian los resultados de vuelta al host y se generan tres imágenes BMP:

- red\_filtered.bmp
- green\_filtered.bmp
- blue\_filtered.bmp

```
===== MENÚ =====
Imagen cargada: input.bmp (1280x962)
1. Filtro Blanco y Negro
2. Filtro Pixelado
3. Identificación de Colores
4. Delineado con Halo
5. Pseudo-Hash
6. Cambiar imagen base
0. Salir
=====
Seleccione una opción: 3
¿Desea introducir umbrales personalizados para los colores? (s/n): n

Resumen de resultados:
Rojo: 278939 píxeles
Verde: 143139 píxeles
Azul: 261245 píxeles
Proceso completado. Imágenes guardadas como 'red_filtered.bmp', 'green_filtered.bmp' y 'blue_filtered.bmp'.
```

Figura 10: Menú de selección y personalización de umbrales

### Resultados por defecto



Figura 11: Detección de color rojo



Figura 12: Detección de color verde



Figura 13: Detección de color azul

## Resultados con umbrales personalizados

```
===== MENÚ =====
Imagen cargada: input.bmp (1280x962)
1. Filtro Blanco y Negro
2. Filtro Pixelado
3. Identificación de Colores
4. Delineado con Halo
5. Pseudo-Hash
6. Cambiar imagen base
0. Salir
=====
Seleccione una opción: 3
¿Desea introducir umbrales personalizados para los colores? (s/n): s
--- Umbrales para ROJO ---
r_min: 0
r_max: 255
g_min: 0
g_max: 50
b_min: 0
b_max: 50
--- Umbrales para VERDE ---
r_min: 0
r_max: 50
g_min: 0
g_max: 255
b_min: 0
b_max: 50
--- Umbrales para AZUL ---
r_min: 0
r_max: 50
g_min: 0
g_max: 50
b_min: 0
b_max: 255

Resumen de resultados:
Rojo: 271365 píxeles
Verde: 114862 píxeles
Azul: 111895 píxeles
Proceso completado. Imágenes guardadas como 'red_filtered.bmp', 'green_filtered.bmp' y 'blue_filtered.bmp'.
```

Figura 14: Menú al usar umbrales personalizados



Figura 15: Rojo personalizado

Figura 16: Verde personalizado

Figura 17: Azul personalizado

Figura 18: Imágenes filtradas con umbrales personalizados

## 2.4. Fase 04. Halo sobre color objetivo en blanco y negro

El propósito de esta fase es aplicar un efecto visual que destaque una región de color específico (rojo, verde o azul) sobre una imagen en escala de grises. Los píxeles que pertenecen al color seleccionado se mantienen en su color original, mientras que el resto de la imagen se convierte a blanco y negro. Además, se aplica un halo negro alrededor de cada píxel detectado del color objetivo, generando un contorno oscuro que realza aún más la zona destacada.

Para lograrlo, se implementa un kernel CUDA llamado `_global_ FilterAndOutlineKernel()`, el cual utiliza memoria compartida para acceder eficientemente a los píxeles vecinos y reducir accesos a memoria global. El kernel opera sobre bloques bidimensionales de hilos, cada uno encargado de una región de la imagen. La dimensión de cada bloque se ajusta dinámicamente en función de la cantidad de memoria compartida disponible en la GPU, asegurando compatibilidad con el hardware.

La región de trabajo de cada bloque incluye un margen adicional definido por el parámetro `haloSize`, que determina el grosor del contorno negro a dibujar. Por tanto, se reserva una porción de memoria compartida mayor que el bloque en sí, para permitir que cada hilo acceda a su vecindario expandido. Esta zona es cargada desde la imagen original a memoria compartida por todos los hilos, y se sincroniza con `_syncthreads()` para garantizar que todos los datos estén disponibles antes de procesar.

Cada hilo primero convierte el píxel que le corresponde a escala de grises mediante la fórmula:

$$\text{gray} = 0,299 \cdot R + 0,587 \cdot G + 0,114 \cdot B$$

A continuación, se evalúa si el píxel cumple con el rango RGB correspondiente al color objetivo, almacenado en la constante `c_targetColorRange`. Si el píxel pertenece al color seleccionado, se conserva en color y se incrementa un contador atómico de detección. Posteriormente, se recorren sus vecinos dentro del rango del halo, y si alguno de ellos no cumple con el umbral del color, se colorea de negro, formando el contorno o halo.

El tamaño del halo es configurable por el usuario en la terminal, permitiendo mayor o menor énfasis en el borde. El kernel también se adapta al hardware disponible, ajustando automáticamente el tamaño del bloque para no superar la memoria compartida permitida por bloque, tal como exige el enunciado.

Desde el host, se lanza la función `faseCuatro()`, que se encarga de configurar el kernel, reservar memoria en la GPU y copiar los resultados de vuelta. Finalmente, la imagen con el halo aplicado se guarda en el archivo `output_with_halo.bmp`.

Color objetivo	Rangos RGB	Tamaño halo	Archivo generado
Rojo	(100–255, 0–150, 0–150)	2	output_with_halo.bmp
Verde	(30–150, 50–255, 0–75)	2	output_with_halo.bmp
Azul	(0–200, 0–250, 100–255)	2	output_with_halo.bmp

Cuadro 2: Configuraciones utilizadas para la detección de color y halo



Figura 19: Imagen original (izquierda) y resultado con halo azul (derecha)



Figura 20: Resultado con halo verde (izquierda) y rojo (derecha)

## 2.5. Fase 05. Reducción paralela de imagen RGB y generación de hash ASCII

En esta última fase, se realiza una operación de reducción paralela sobre la imagen RGB con el objetivo de generar un conjunto reducido de valores representativos, que posteriormente se transforman en una secuencia tipo **pseudo-hash**. Este hash se representa en tres formas: valores numéricos originales, versión normalizada al rango de caracteres ASCII imprimibles y, finalmente, los caracteres ASCII correspondientes.

El proceso comienza convirtiendo cada píxel RGB en un valor escalar utilizando una fórmula ponderada basada en la percepción humana del color:

$$\text{valor} = R \cdot 0,5 + G \cdot 0,25 + B \cdot 0,25$$

Esta conversión se implementa mediante el kernel `transformarRGBaEscalar()`, que genera un vector de enteros a partir de la imagen de entrada. A continuación, se realiza una reducción paralela para obtener los valores máximos, utilizando el kernel `reductionKernelMax()`, el cual emplea memoria compartida y sincronización de hilos para comparar pares de valores dentro de cada bloque.

El procedimiento sigue un enfoque jerárquico:

- Se inicia con bloques de tamaño 512 para obtener máximos parciales.
- Si el número de máximos sigue siendo mayor que un umbral definido (`UMBRAL_FINAL = 15`), se reduce el tamaño del bloque a 15 y se continúa el proceso de reducción.
- El ciclo se repite hasta que el número total de valores sea menor o igual a 15.

Durante la ejecución, se muestran mensajes informativos que detallan el número de elementos actuales, el número de bloques lanzados en cada iteración y si es necesario continuar la reducción. Este log sirve como seguimiento del progreso en tiempo real:

```
Ancho: 1280, Alto: 962
Reducimos con 2405 bloques
Hay que reducir más.
-----> Tenemos: 2405
-----> Lanzamos: 161 (160.33)
-----> Tenemos: 161
-----> Lanzamos: 11 (10.73)
-----> Fin: 11
```

Una vez completado el proceso, se imprime el resultado del pseudo-hash en tres representaciones:

1. **Hash (números originales):** Lista de valores máximos obtenidos tras la reducción.
2. **Hash (números normalizados):** Cada valor se mapea al rango de caracteres ASCII imprimibles (aproximadamente de 35 a 125), evitando caracteres de control.
3. **Hash (ASCII):** Visualización directa del hash como caracteres ASCII.

Ejemplo de salida obtenida:

```
Hash (num. original)
249 247 251 255 255 255 252 252 253 252 244
```

```
Hash (num. normalizado)
122 122 123 125 125 125 123 123 124 123 121
```

```
Hash (ASCII)
z z { } } } { { | { y
```

La salida mediante el terminal para las 3 imágenes se representa de la siguiente manera:

```
Ancho: 1280, Alto: 962
Reducimos con 2405 bloques
Hay que reducir mas.
-----> Tenemos: 1231360
-----> Lanzamos: 2405 (2405.00)
Hay que reducir mas.
-----> Tenemos: 2405
-----> Lanzamos: 161 (160.33)
Hay que reducir mas.
-----> Tenemos: 161
-----> Lanzamos: 11 (10.73)
-----> Fin: 11

Copiando 11 elementos finales desde GPU...

Hash (num. original)
244 251 253 252 251 253 255 254 251 247 251
Hash (num. normalizado)
121 123 124 123 123 124 125 124 123 122 123
Hash (ASCII)
y { | { { | } | { z {
```

Figura 21: Reducción y hash para imagen de Pokémon

```

Ancho: 1200, Alto: 794
Reducimos con 1860 bloques
Hay que reducir mas.
-----> Tenemos: 952800
-----> Lanzamos: 1860 (1860.94)
Hay que reducir mas.
-----> Tenemos: 1860
-----> Lanzamos: 124 (124.00)
Hay que reducir mas.
-----> Tenemos: 124
-----> Lanzamos: 9 (8.27)
-----> Fin: 9

Copiando 9 elementos finales desde GPU...

Hash (num. original)
238 231 251 236 251 254 255 230 236
Hash (num. normalizado)
119 116 123 118 123 124 125 116 118
Hash (ASCII)
w t { v { | } t v

```

Figura 22: Reducción y hash para imagen de Fast and Furious

```

Ancho: 1080, Alto: 684
Reducimos con 1442 bloques
Hay que reducir mas.
-----> Tenemos: 738720
-----> Lanzamos: 1442 (1442.81)
Hay que reducir mas.
-----> Tenemos: 1442
-----> Lanzamos: 97 (96.13)
Hay que reducir mas.
-----> Tenemos: 97
-----> Lanzamos: 7 (6.47)
-----> Fin: 7

Copiando 7 elementos finales desde GPU...

Hash (num. original)
127 127 127 127 127 127 127
Hash (num. normalizado)
79 79 79 79 79 79 79
Hash (ASCII)
0 0 0 0 0 0 0

```

Figura 23: Reducción y hash para imagen con predominancia de color rojo

Este proceso implementa correctamente una reducción adaptable en GPU, respetando las limitaciones del hardware y garantizando que los recursos se ajusten dinámicamente en cada iteración.

### 3. Conclusión

A través de la implementación de las cinco fases del proyecto, se ha demostrado la eficacia de CUDA para procesar imágenes de forma masiva y en paralelo. Cada etapa ha sido desarrollada considerando no solo el rendimiento, sino también la adaptabilidad del código a distintos dispositivos con diferentes capacidades de hardware.

El uso de memoria compartida en las fases 2 y 4 ha permitido una mejora significativa en el acceso a vecinos y píxeles adyacentes, reduciendo así el número de accesos a memoria global. Las estructuras en memoria constante utilizadas para los umbrales RGB han facilitado la comparación eficiente en los kernels. Asimismo, el manejo de reducción jerárquica en la fase 5 ha representado una solución compacta y eficaz para generar un identificador visual (pseudo-hash) a partir de la imagen procesada.

El sistema completo proporciona una interfaz sencilla pero potente, en la cual el usuario puede elegir la fase a ejecutar y configurar parámetros relevantes, obteniendo como resultado imágenes transformadas o datos de salida de forma inmediata.

En resumen, el proyecto cumple con los objetivos planteados, tanto a nivel funcional como técnico, aprovechando las capacidades de la GPU y ofreciendo un entorno interactivo para el tratamiento de imágenes en paralelo.