Filtrado de imagenes en CUDA

Hugo Aranda Sánchez y Paula Muñoz Giner
Departamento de Arquitectura de Computadores

21/06/2023

Índice

1 Introducción

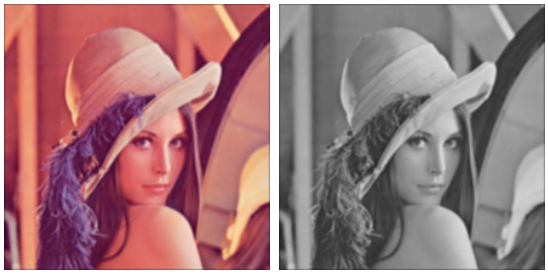
1.1 Qué haremos?

En este trabajo realizaremos una implementación de diversos filtros de procesamiento de imágenes, incluyendo los filtros Gauss, Sharp, Box, Identidad y Laplace. Estos filtros pueden ser aplicados tanto a imágenes en blanco y negro como a imágenes en color (opciones contempladas dentro de la aplicación).

Las aplcaciones de estos filtros son múltiples, desde procesado de imágenes genérico para facilitar a otros algoritmos encontrar objetos dentro de la imagen hasta el procesado de imagenes medicas haciendo más discernibles las condiciones fisicas de los pacientes produciendo imágenes de mayor calidad para los profesionales de la salud (pues imagenes como las de tomografías tienen grandes cantidades de ruido que debe ser eliminado).

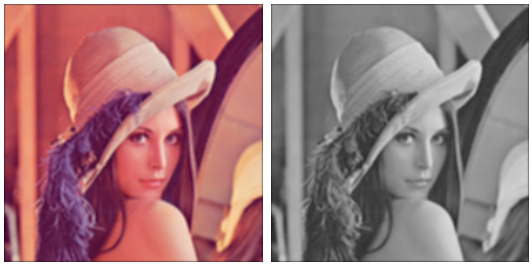
Los filtros que trabajaremos serán los siguentes:

El filtro Gauss que se utiliza para suavizar una imagen y reducir el ruido. Aplicando este filtro, se realiza una convolución de la imagen original con una máscara Gaussiana que difumina los detalles y produce una imagen más suave. Esto es especialmente útil para eliminar el ruido y suavizar los bordes. Esta máscara gaussiana hace una media de los píxeles colindantes dando más importancia a los más próximos y menos importancia progresivamente a los mas lejanos.

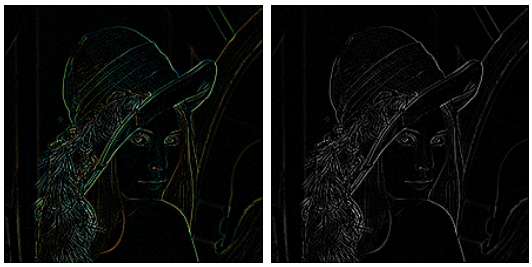


El filtro Sharpening , por otro lado, se utiliza para resaltar los detalles y los bordes de una imagen. Aplicando este filtro, se realiza una convolución de la imagen original con una máscara de enfoque que realza los cambios bruscos de intensidad y produce una imagen con mayor contraste en los bordes.

El filtro Box, también conocido como filtro de media, se utiliza para suavizar una imagen y reducir el ruido. Aplicando este filtro, se realiza una convolución de la imagen original con una máscara de tamaño fijo que calcula el promedio de los píxeles vecinos, generando así una imagen más suavizada.



El filtro Identidad (ID) es un filtro que no produce cambios en la imagen original. Aplicando este filtro, se obtiene la misma imagen de entrada sin ninguna modificación. Aunque puede parecer un filtro inútil, se utiliza en aplicaciones donde es necesario realizar operaciones de convolución sin alterar el contenido de la imagen.



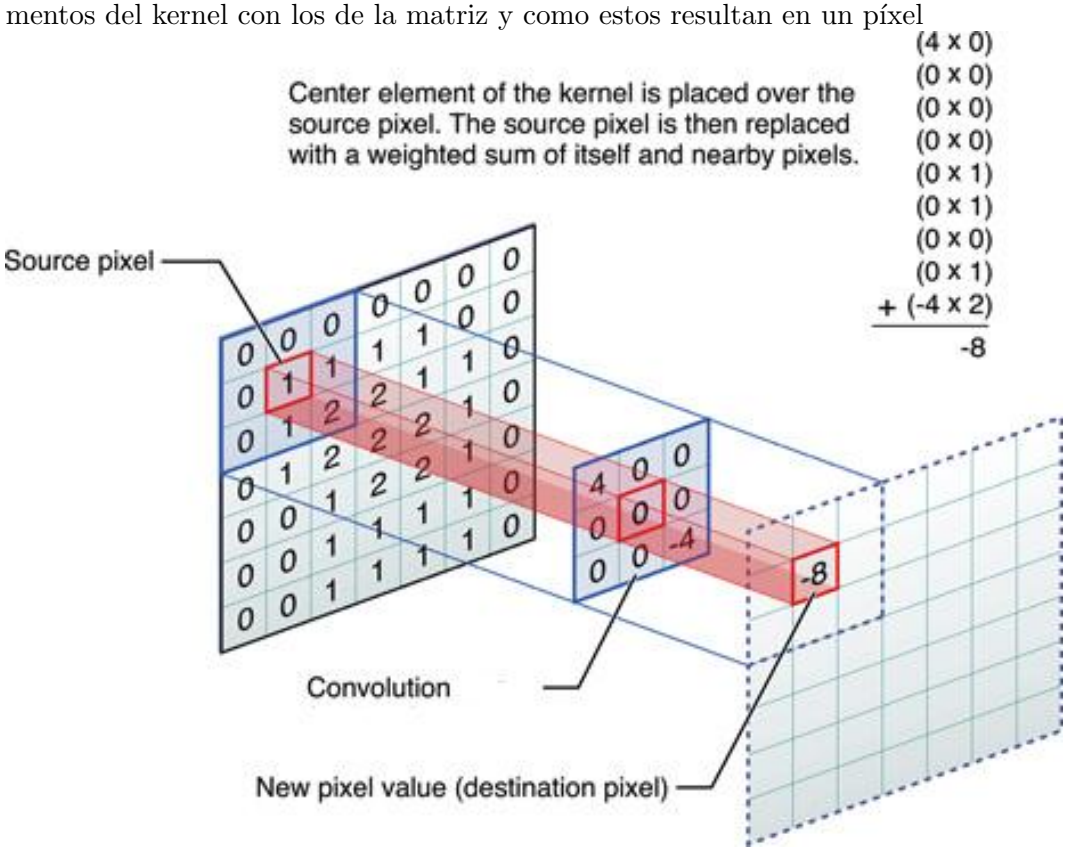
Por último, el filtro Laplaciano se utiliza para resaltar los detalles y los cambios bruscos de intensidad en una imagen. Aplicando este filtro, se realiza una convolución de la imagen original con una máscara Laplaciana que detecta los cambios en la segunda derivada de la intensidad de los píxeles, lo que resulta en una imagen resaltando los bordes y los detalles.

1.2 Cómo lo haremos?

Ahora bien, hemos mencionado anteriormente que para todos los filtros haremos una convolución de la imagen original con distintas máscaras, ¿qué significa esto exactamente? Básicamente, durante el desarrollo del código nos percatamos que diferentes filtros aplicaban básicamente los mismos cálculos a excepción de un par de parametros facilmente programables que harían que la aplicacione sea más dinámica y apliable.

La convolucón por lo tanto es esta operación común para la aplicación de ciertos filtros sencillos para imágenes. Esta hace una operación de multiplicación y suma entre una matriz llamada kernel (la que define que filtro usamos que también se puede llamar máscara) y una región de la matriz original. Se multiplican todos los valores de estos dos y se suman para resultar en un único píxel en la imágen resultante.

Figure 1: Aquí se puede ver en mas detalle como se multiplican y suman los elementos del kernel con los de la matriz y como estos resultan en un píxel



2 Versiones

2.1 Primera versión

Aqui podemos encontrar la primera versión donde trabajamos el kernel de Gauss-Seidel, como kernel generico, pues luego lo modificaremos para admitir otros filtros.

```
__global__ void Gauss_kernel (int N, int M, unsigned char *source,
    unsigned char *dest) {
  float kernel[] = {0.0625, 0.125, 0.0625, 0.125, 0.25, 0.125, 0.0625,
    0.125, 0.0625};
  int kernelSize = 3;
  int i = blockIdx.y * blockDim.y + threadIdx.y;
  int j = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N && j < M) {
    float red = 0.0;
    float green = 0.0;
    float blue = 0.0;
      for (int k = 0; k < kernelSize; ++k){
        for (int l = 0; l < kernelSize; ++l){
        int y = i + k - kernelSize / 2;
        int x = j*3 + (l - kernelSize / 2)*3;
        if (y >= 0 && y < N && x >= 0 && x < M*3){
          red += source[y*M*3 + x] * kernel[k*3 + l];
          green += source[y*M*3 + x + 1] * kernel[k*3 + l];
          blue += source[y*M*3 + x + 2] * kernel[k*3 + l];
        }
      }
    }
    int offset = i*M*3 + j*3;
    dest[offset] = (unsigned char) red;
    dest[offset+1] = (unsigned char) green;
    dest[offset+2] = (unsigned char) blue;
  }
}
```

Esta version cada thread realiza una operación de convolución en un único píxel de la imagen de entrada y almacena el resultado en el píxel correspondiente en la imagen de salida.
El tiempo global al ejecutar esta versión ha sido de 0.968608 milseg y en el calculo del kernel de 0.043264
A partir de esta version nos planteamos las siguientes mejoras: el uso de memoria compartida para hacer las lecturas a nivel de bloque, que cada thread tratase una fila para perder menos tiempo en overheads.

2.2 Segunda Versión

Aqui podemos encontrar la segunda version, donde seguimos trabajando con el kernel de Gauss-Seidel, pero esta vez implementamos el uso de shared memory.

```
__global__ void Gauss_kernel (int N, int M, unsigned char *source,
    unsigned char *dest) {

  __shared__ unsigned char s_source[SIZE][SIZE*3];

  float kernel[] = {0.0625, 0.125, 0.0625, 0.125, 0.25, 0.125, 0.0625,
    0.125, 0.0625};
  int kernelSize = 3;
  int h_kernelSize = kernelSize / 2;

  int bx = blockIdx.x;  int sx = threadIdx.x;
  int by = blockIdx.y;  int sy = threadIdx.y;

  int i = by * blockDim.y + sy;
  int j = bx * blockDim.x + sx;

  //offsets para cargar shared
  int offset = i*M*3 + j*3;

  if (i < N && j < M) {
    //cada thread carga su pixel
    s_source[sy][sx*3] = source[offset];
    s_source[sy][sx*3+1] = source[offset+1];
    s_source[sy][sx*3+2] = source[offset+2];
    __syncthreads();

    float red = 0.0;
    float green = 0.0;
    float blue = 0.0;

    for (int k = 0; k < kernelSize; ++k){
      for (int l = 0; l < kernelSize; ++l){
        //indices para memoria principal
        int y = i + k - h_kernelSize;
        int x = (j + l - h_kernelSize)*3;
        //indices para shared
        int s_y = sy + k - h_kernelSize;
        int s_x = (sx + l - h_kernelSize)*3;

        if (y >= 0 && y < N && x >= 0 && x < M*3){ //dentro de
    principal
          if(s_y >= 0 && s_y < SIZE && s_x >= 0 && s_x < SIZE*3){ //
    dentro de shared
            red += s_source[s_y][s_x] * kernel[k*3 + l];
            green += s_source[s_y][s_x + 1] * kernel[k*3 + l];
            blue += s_source[s_y][s_x + 2] * kernel[k*3 + l];
          } else { //no lo hemos encontrado en shared y vamos a
    principal
            red += source[y*M*3 + x] * kernel[k*3 + l];
            green += source[y*M*3 + x + 1] * kernel[k*3 + l];
            blue += source[y*M*3 + x + 2] * kernel[k*3 + l];
          }
        }
```

```
49          }
50      }
51      //pasamos los datos a destino
52      dest[offset] = (unsigned char) red;
53      dest[offset+1] = (unsigned char) green;
54      dest[offset+2] = (unsigned char) blue;
55    }
56  }
```

Como hemos dicho antes, esta versión usa shared memory, utiliza la memoria compartida para almacenar una parte de la imagen que se procesará. Los threads cargan los datos necesarios de la memoria global a la memoria compartida antes de realizar las operaciones de convolución. Se realiza una sincronización de threads para asegurarse de que todos los threads han terminado de cargar los datos en la memoria compartida antes de continuar. Esto permite un acceso más rápido a los datos locales en lugar de acceder repetidamente a la memoria global, lo que mejora el rendimiento y la eficiencia. Además, la utilización de la memoria compartida permite compartir datos entre threads y aprovechar la localidad espacial de los píxeles en la imagen para mejorar el rendimiento.

Al ejecutar esta segunda versión podemos ver que el tiempo global ha sido de 0.662784 milseg, un 32% más rapido que la primera versión, y en el calculo del kernel vemos una mejora del 25%, ya que solo tarda 0.032608 milseg,

Una posible mejora que nos plantemos en este punto fue hacer que no solo los pixeles de dentro del block esten dentro sino los colindantes también. Pues en casos limite hemos de acceder a estos para hacer las medias de los pixeles que estan en el borde y actualmente esto se hace con un acceso a memoria principal en vez de al shared.

2.3 Tercera Versión

Aqui podemos encontrar la tercera version, una version con flag de kernel y version preliminar por filas.

```
__global__ void Gauss_kernel (int N, int M, unsigned char *source,
    unsigned char *dest) {

  float kernel[] = {0.0625, 0.125, 0.0625, 0.125, 0.25, 0.125, 0.0625,
    0.125, 0.0625};
  int kernelSize = 3;

  int i = blockIdx.y * blockDim.y + threadIdx.y;

  if (i < N) {
    for (int j = 0; j<M*3; j += 3){
      float red = 0.0;
      float green = 0.0;
      float blue = 0.0;
      for (int k = 0; k < kernelSize; ++k){
        for (int l = 0; l < kernelSize; ++l){
          int y = i + k - kernelSize / 2;
          int x = j*3 + (l - kernelSize / 2)*3;
          if (y >= 0 && y < N && x >= 0 && x < M*3){
            red += source[y*M*3 + x] * kernel[k*3 + l];
            green += source[y*M*3 + x + 1] * kernel[k*3 + l];
            blue += source[y*M*3 + x + 2] * kernel[k*3 + l];
          }
        }
      }

      int offset = i*M*3 + j*3;
      dest[offset] = (unsigned char) red;
      dest[offset+1] = (unsigned char) green;
      dest[offset+2] = (unsigned char) blue;
    }
  }
}
```

En esta versión cada thread accede directamente a los datos en la memoria global. El bucle interno se ejecuta solo para el eje j, las filas, lo que significa que cada thread se encarga de una fila completa de píxeles. Esto lo hicimos desde el punto de vista de evitar overheads por hacer una descomposición muy fina.

Al ejecutar esta versión vemos que los tiempos empeoran drasticamente, el global no mejora nada repecto a la primera versión, incluso empeora un poco, ya que tarda 0.985984 milseg. Si miramos el tiempo de calculo del kernel vemos que tarda 6.6 veces mas que en la primera versión, ya que tardad 0.287488 milseg. Esta versión no acabó de estar funcional pero al dar tiempos tan desalentadores la dejamos de lado.

2.4 Cuarta versión

Esta es una versión muy similar a la segunda versión pero mejorada, ya que nos hemos centrado en la modularidad y reutilización de codigo para que pueda aceptar diferentes kernels.

```
__global__ void convolve_RGB (int N, int M, unsigned char *source,
    unsigned char *dest, float *kernel, int kernelSize) {

  __shared__ unsigned char s_source[SIZE][SIZE*3];

  int h_kernelSize = kernelSize / 2;

  int bx = blockIdx.x;  int sx = threadIdx.x;
  int by = blockIdx.y;  int sy = threadIdx.y;

  int i = by * blockDim.y + sy;
  int j = bx * blockDim.x + sx;

  //offsets para cargar shared
  int offset = i*M*3 + j*3;

  if (i < N && j < M) {
    //cada thread carga su pixel
    s_source[sy][sx*3] = source[offset];
    s_source[sy][sx*3+1] = source[offset+1];
    s_source[sy][sx*3+2] = source[offset+2];
    __syncthreads();

    float red = 0.0;
    float green = 0.0;
    float blue = 0.0;

    for (int k = 0; k < kernelSize; ++k){
      for (int l = 0; l < kernelSize; ++l){
        //indices para memoria principal
        int y = i + k - h_kernelSize;
        int x = (j + l - h_kernelSize)*3;
        //indices para shared
        int s_y = sy + k - h_kernelSize;
        int s_x = (sx + l - h_kernelSize)*3;

        if (y >= 0 && y < N && x >= 0 && x < M*3){ //dentro de
    principal
          if(s_y >= 0 && s_y < SIZE && s_x >= 0 && s_x < SIZE*3){ //
    dentro de shared
            red += s_source[s_y][s_x] * kernel[k*3 + l];
            green += s_source[s_y][s_x + 1] * kernel[k*3 + l];
            blue += s_source[s_y][s_x + 2] * kernel[k*3 + l];
          } else { //no lo hemos encontrado en shared y vamos a
    principal
            red += source[y*M*3 + x] * kernel[k*3 + l];
            green += source[y*M*3 + x + 1] * kernel[k*3 + l];
            blue += source[y*M*3 + x + 2] * kernel[k*3 + l];
          }
        }
      }
    }
```

```
49      //pasamos los datos a destino
50      dest[offset] = (unsigned char) red;
51      dest[offset+1] = (unsigned char) green;
52      dest[offset+2] = (unsigned char) blue;
53    }
54  }
```

Como podemos ver esta version acepta parámetros adicionales, como el kernel y el tamaño del kernel. Gracias a passar por parametro el kernel podemos reutilizar el código para diferentes tipos de convolución. Esto es útil ya que así se pueden aplicar diferentes filtros simpemente llamando a la función convolveRGB con diferentes valores sin tener que duplicar el código. El parametro del tamaño tambien nos permite utilizar diferentes tamaños de kernel sin tener que modificar el código original. Esto es útil para experimentar con diferentes tamaños de kernel o cuando queremos trabajar con algoritmos que requieren diferentes tamaños de kernel en diferentes etapas del procesamiento.

Con esta versión queriamos mostrar lo versátil que es el cuda kernel de convolveRGB el cual puede aplicar distintos kernels de procesado de imagen obteniendo resultados muy diversos sin problemas.

Si miramos el tiempo de ejecución podemos ver que el tiempo global de todos los filtros mejora incluso más que la segunda versión, ya que tardan entre 0.417440 y 0.451072, es decir tarda entre un 57%-54% menos que la primera versión. Además el tiempo de calculo del kernel tambien mejora notablemente, tardan entre 0.027648 y 0.030720, que es un 36%-29% menos que la primra verión.

3 Implementación final

En nuestra version final del proyecto, implementaremos los filtros Gauss, Sharp, Box, Identidad y Laplace utilizando diferentes técnicas de procesamiento paralelo. Comenzamos definiendo un kernel denominado "KernelByN" que se encarga de convertir una imagen a color en blanco y negro. Este kernel se ejecuta en paralelo en la GPU y hace la media de los valores RGB de cada píxel para quedarse solo con su intensidad y así convertirlo en escala de grises.

```
__global__ void KernelByN (int N, int M, unsigned char *A) {
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + 3*threadIdx.x;
  if(row < M && col < N){
    int offset = row * N * 3 + col * 3;
    A[offset] = A[offset+1] = A[offset+2] = (A[offset] + A[offset+1] +
    A[offset+2])/3;
  }
}
```

A continuación, tenemos el kernel principal llamado "convolve RGB" que realiza la convolución de la imagen con un kernel específico. Este kernel también se ejecuta en paralelo en la GPU y utiliza memoria compartida para acceder eficientemente a los datos de la imagen de entrada y reduce la cantidad de accesos a memoria necesarios. Cada threads del kernel procesa un píxel de salida y realiza los cálculos de convolución utilizando los valores del kernel pasados por parametro y los componentes de color de la imagen de entrada. Al finalizar, se guardan los resultados en la imagen de salida.

```
__global__ void convolve_RGB (int N, int M, unsigned char *source,
    unsigned char *dest, float *kernel, int kernelSize) {
  __shared__ unsigned char s_source[SIZE][SIZE*3];
  int h_kernelSize = kernelSize / 2;
  int bx = blockIdx.x;  int sx = threadIdx.x;
  int by = blockIdx.y;  int sy = threadIdx.y;
  int i = by * blockDim.y + sy;
  int j = bx * blockDim.x + sx;
  int offset = i*M*3 + j*3;
  if (i < N && j < M) {
    s_source[sy][sx*3] = source[offset];
    s_source[sy][sx*3+1] = source[offset+1];
    s_source[sy][sx*3+2] = source[offset+2];
    __syncthreads();
    float red = 0.0;
    float green = 0.0;
    float blue = 0.0;
    for (int k = 0; k < kernelSize; ++k){
      for (int l = 0; l < kernelSize; ++l){
        int y = i + k - h_kernelSize;
        int x = (j + l - h_kernelSize)*3;
        int s_y = sy + k - h_kernelSize;
        int s_x = (sx + l - h_kernelSize)*3;
        if (y >= 0 && y < N && x >= 0 && x < M*3){
          if(s_y >= 0 && s_y < SIZE && s_x >= 0 && s_x < SIZE*3){
            red += s_source[s_y][s_x] * kernel[k*3 + l];
            green += s_source[s_y][s_x + 1] * kernel[k*3 + l];
```

```
27              blue += s_source[s_y][s_x + 2] * kernel[k*3 + l];
28            } else {
29              red += source[y*M*3 + x] * kernel[k*3 + l];
30              green += source[y*M*3 + x + 1] * kernel[k*3 + l];
31              blue += source[y*M*3 + x + 2] * kernel[k*3 + l];
32            }
33          }
34        }
35      }
36      dest[offset] = (unsigned char) red;
37      dest[offset+1] = (unsigned char) green;
38      dest[offset+2] = (unsigned char) blue;
39    }
40  }
```

El programa principal comienza leyendo la imagen de entrada y obtiene su tamaño y número de componentes de píxeles. Luego, se selecciona una GPU aleatoria y se configuran los parámetros de ejecución en paralelo, el número de threads por bloque y el número de bloques en cada dimensión.

A continuación, se reserva memoria en el host y en el dispositivo para los datos de entrada y salida de la GPU. La imagen de entrada se copia desde el host al dispositivo, y se copia el kernel correspondiente según el filtro seleccionado.

Después de eso, se ejecuta el kernel correspondiente dependiendo del modo seleccionado. Si el modo es de blanco y negro, se ejecuta el kernel "KernelByN" antes del kernel "convolve RGB"; de lo contrario, solo se ejecuta el kernel "convolve RGB".

Una vez finalizada la ejecución del kernel, se copia la imagen resultante desde el dispositivo al host. Luego se liberan las memorias del dispositivo y se calcula el tiempo total de ejecución del programa y el tiempo de ejecución del kernel.

Finalmente, se escribe la imagen resultante en un archivo de salida y se liberan las memorias del host utilizadas para almacenar la imagen resultante.

Al ejecutar esta ultima versión podemos ver que exepto con el filto de gauss, que solo obtenemos un 32% de mejora en el tiempo global y un 3% en el tiempo del kernel. Encanvio con el resto de filtros poemos ver los mejores tiempos hasta ahora. El tiempo global medio es de 0.448867 milseg, que es un 55% de mejora. El tiempo de calculo de kernel medio solo a color es de 0.028624, que es un 34% de mejora y el de en blanco y negro un poco mas, es de 0.03392, que es un 21% de mejora.

Table 1: Tiempos Globales

TIEMPO GLOBAL	Default (145.3kb)	IMG3 (2MB)	IMG2 (11.1MB)
Versión 1	0.97ms	27.33ms	59.77ms
Versión Final Color	0.44ms	18.52ms	32.05ms
Versión Final Blanco y Negro	0.46ms	18.59ms	32.76ms

Table 2: Tiempos de kernel

TIEMPO KERNEL	Default (145.3kb)	IMG3 (2MB)	IMG2 (11.1MB)
Versión 1	0.043ms	0.199ms	0.555ms
Versión Final Color	0.286ms	0.32ms	0.99ms
Versión Final Blanco y Negro	0.034ms	0.36ms	1.15ms

4 Conclusiones

Si nos fijamos en la Versión Final a Color y la Versión Final en Blanco y Negro, vemos que ambas tienen tiempos de ejecución similares en todas las imágenes. Esto sugiere que el proceso adicional de conversión a blanco y negro en la Versión Final en Blanco y Negro no tiene un impacto significativo en el rendimiento en comparación con la convolución en cada canal de color en la Versión Final a Color.

En cambio si comparamos los tiempos de ejecución entre la versión 1 y las versiones finales del código, podemos observar una clara mejora en la eficiencia y en la velocidad de procesamiento de imágenes. La reducción en el tiempo de ejecución es especialmente notable en las imágenes más grandes, donde los tiempos de procesamiento se reducen significativamente en comparación con la versión 1. En promedio, la versión final reduce los tiempos de ejecución en aproximadamente un 50% en comparación con la versión 1.

Estos resultados demuestran que las estrategias implementadas en la versión final, como el uso de memoria compartida, la carga de datos optimizada y el uso de cálculos paralelos en la GPU, han logrado mejorar la eficiencia del algoritmo de procesamiento de imágenes. Es importante destacar que estos resultados son específicos de las imágenes de prueba utilizadas y del hardware utilizado para la ejecución. Sin embargo, los tiempos de ejecución más bajos y la mejora relativa entre las versiones respaldan la efectividad de las optimizaciones realizadas en el código final.

En conclusión, las mejoras implementadas en la versión final han logrado una notable mejora en el tiempo de procesamiento de imágenes, lo que demuestra la importancia de la optimización y paralelización del código para obtener un rendimiento eficiente.

Bibliografia

[1] Lewin Day, 2021 *What exactly is gaussian blur?*, Consultado el 14-5-23, Source: `https://hackaday.com/2021/07/21/what-exactly-is-a-gaussian-blur/`

[2] Robert Fisher, Simon Perkins, Ashley Walker and Erik Wolfart, 2000 *Laplacian/Laplacian of Gaussian*, Consultado el 20-5-23, Source: `https://homepages.inf.ed.ac.uk/rbf/HIPR2/log.htm`