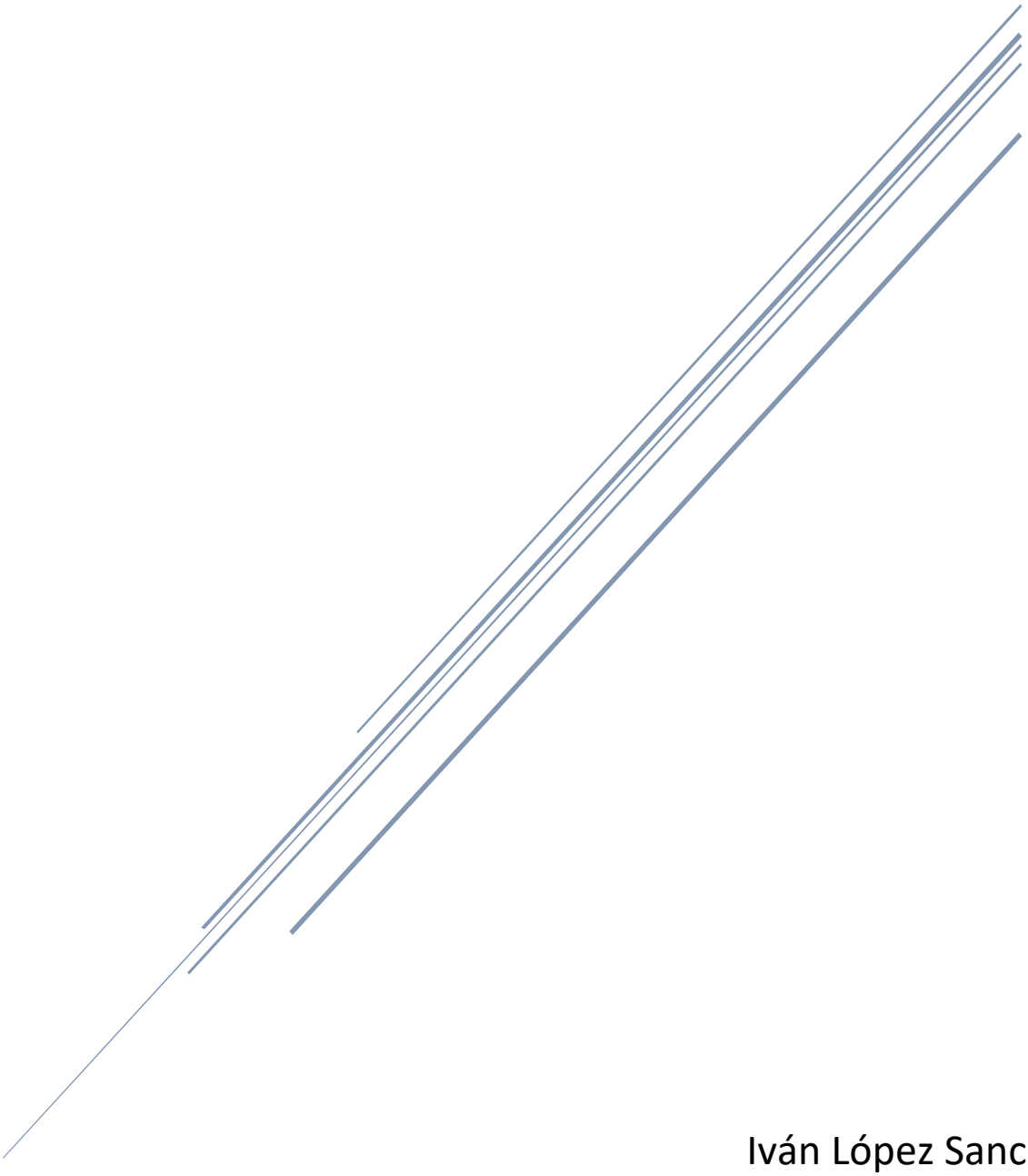
ESTUDIO Y PROPUESTA DE LA PARALELIZACIÓN DE UNA APLICACIÓN

INGENIERÍA DE LOS COMPUTADORES



Iván López Sanchís
Adrián Ríos Armero
Joan Climent Quiñones
Jaime Morales Vaello

ÍNDICE

Tarea 1 - Búsqueda/Desarrollo de un buen candidato a ser paralelizado

En esta tarea, se decide qué tema va a paralelizar el grupo. Nuestro grupo ha decidido tratar el tema del procesamiento de imágenes con Sobel y Canny, especialmente son algoritmos de detección de bordes. En primer lugar, vamos a explicar el funcionamiento de detección de bordes con Sobel. El operador Sobel utiliza un par de núcleos de convolución de 3x3 para calcular el gradiente de la imagen en cada píxel. Estos dos núcleos están diseñados para detectar cambios en la intensidad en las direcciones horizontal y vertical. La magnitud del gradiente en cada píxel se calcula como la raíz cuadrada de la suma de los cuadrados de los gradientes horizontal y vertical. También se puede determinar la dirección del gradiente.

$$\mathbf{G} = \sqrt{\mathbf{G_x}^2 + \mathbf{G_y}^2}$$

Por otro lado, el algoritmo Canny trata de un proceso de múltiples etapas que proporciona resultados más precisos en comparación con el operador Sobel. Entre las etapas están las siguientes:

- ***Suavizado Gaussiano***: La imagen se suaviza primero con un filtro gaussiano para reducir el ruido.
- ***Supresión de No Máximos***: Esta etapa involucra adelgazar los bordes manteniendo solo el máximo local en la dirección del gradiente. Esto asegura que solo se retengan los bordes más significativos.
- ***Seguimiento de Bordes por Histéresis***: En este paso, se siguen los bordes considerando dos umbrales: un umbral alto (*bordes fuertes*) y un umbral bajo (*bordes débiles*). Los píxeles con magnitudes de gradiente por encima del umbral alto se consideran bordes fuertes, y los que están entre los umbrales bajo y alto se consideran bordes débiles.

En conclusión, el algoritmo Canny es como un detective de bordes muy inteligente que puede encontrar bordes de una imagen de manera muy precisa, incluso si la imagen está un poco sucia. Sin embargo, este detective es lento y necesita más tiempo para hacer su trabajo.

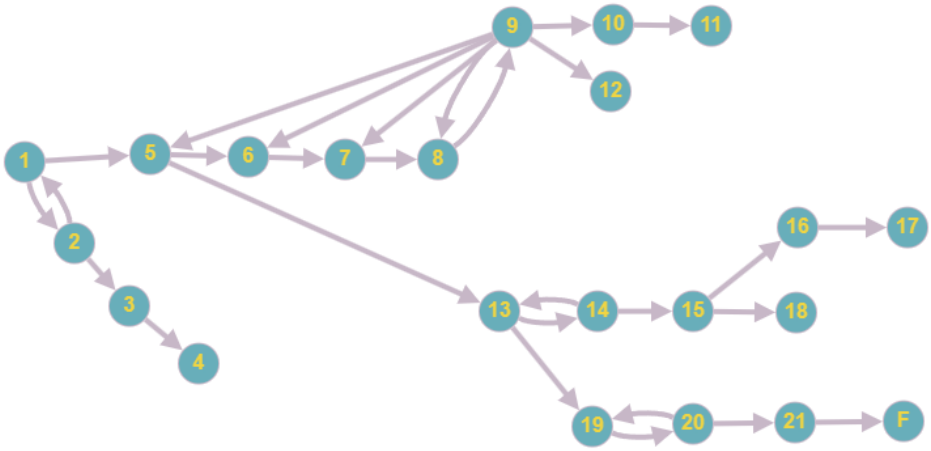
Por otro lado, el operador Sobel es como un detective más rápido, pero no tan inteligente. Puede encontrar bordes en la imagen, pero no es tan bueno como Canny para lidiar con imágenes ruidosas o detalles precisos.

Así que, si tienes mucho tiempo y quieres los mejores resultados, puedes usar el detective Canny. Pero si necesitas respuestas rápidas y no te importa que no sean perfectas, el detective Sobel hará el trabajo de manera más rápida. La elección depende de cuánto tiempo tengas y cuán precisos necesitan ser tus resultados.

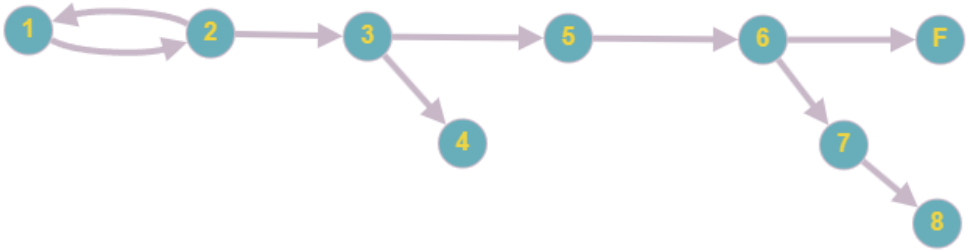
Tarea 2.1 - Grafos de control de flujo

main: Grafo que representa el camino lógico que recorre el programa. Los nodos 5, 6 ,7 son ampliados a continuación.

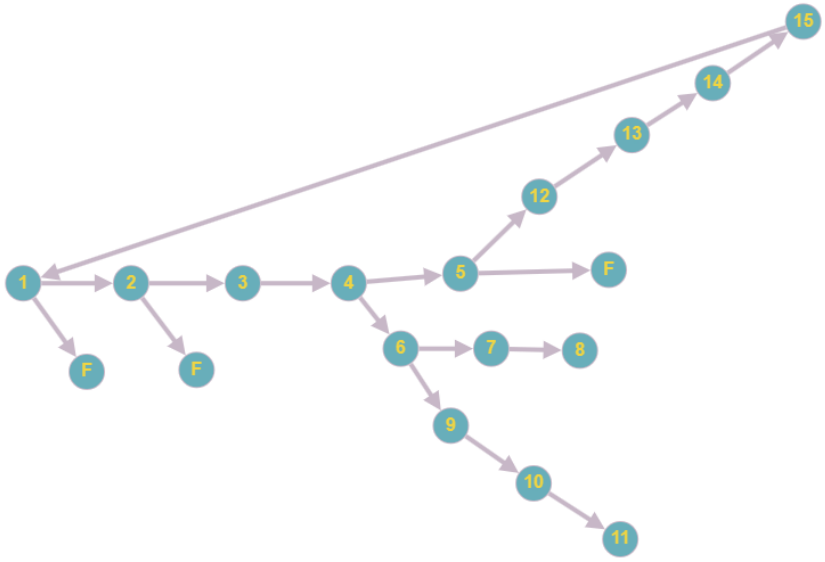
magnitud_matrix: Grafo que representa los caminos lógicos que puede recorrer la función magnitud_matrix (nodo 5 del grafo main) al crear una matriz de magnitudes.



peak_detection: Grafo que representa los caminos lógicos que puede recorrer la función peak_detection (nodo 6 del grafo main) al detectar los picos de la matriz bidimensional introducida.



recursiveDT: Grafo que representa los caminos lógicos que puede recorrer la función recursiveDT (nodo 7 del grafo main) al realizar la histéresis de doble umbralización.


15
14
13
12
1
2
3
4
5
F
F
F
6
7
8
9
10
11

Tarea 2.2 - Estudio de la estructura del programa

El algoritmo canny tiene varias funciones, en la implementación que presentamos hay tres funciones principales, empezaremos por la primera:

La función magnitude_matrix() calcula la magnitud del gradiente de la imagen que le hemos proporcionado a partir de la primera derivada gaussiana. En primer lugar, rellena los valores de la máscara mediante la fórmula de la primera derivada de Gauss. A continuación, realiza una convolución de barrido en la matriz de la imagen de entrada para obtener las matrices Δy y Δx. Por último, calcula la magnitud del gradiente tomando la raíz cuadrada de la suma de Δy^2 y Δx^2 y la almacena en la matriz de magnitud.

Por norma general, las variables sig, dim y cent son variables escalares, las matrices pic, mag, x e y son matrices de tamaño height x width, donde height y width son las dimensiones de la imagen. Las matrices maskx y masky son matrices de tamaño dim x dim, donde dim es el tamaño de la máscara.

```
void magnitude_matrix(double **pic, double **mag, double **x, double **y)
```

- **pic** es una matriz de entrada que contiene la imagen original a la que se accede para realizar la convolución y calcular la matriz de magnitud.
- **mag** es la matriz de salida que contiene la matriz de magnitud.
- **x e y** son matrices auxiliares que contienen los valores de las derivadas parciales en las direcciones x e y. Se accede a ellas para almacenar los valores de las derivadas parciales.

```
int dim = 6 * sig + 1, cent = dim / 2;
double maskx[dim][dim], masky[dim][dim];

// Use the Gausian 1st derivative formula to fill in the mask values
for (int p = -cent; p <= cent; p++)
{
    for (int q = -cent; q <= cent; q++)
    {
        maskx[p+cent][q+cent] = q * exp(-1 * ((p * p + q * q) / (2 * sig * sig)));
        masky[p+cent][q+cent] = p * exp(-1 * ((p * p + q * q) / (2 * sig * sig)));
    }
}
```

- **maskx y masky** son matrices que contienen los valores de la máscara de convolución para las derivadas parciales en las direcciones x e y, respectivamente. Se accede a ellas en modo lectura para realizar la convolución.
- **dim y cent** son variables que contienen el tamaño de la máscara y su centro, respectivamente. Se utilizan para calcular los índices de las matrices de máscara.
- **sig** es una variable que contiene el valor de la desviación estándar de la función Gaussiana utilizada para calcular la máscara de convolución. Se utiliza para calcular los valores de la máscara.

```
vector<Point*> peak_detection(double **mag, HashMap *h, double **x, double **y)
```

La segunda función, peak_detection() como dice su nombre detecta los vértices, esta toma como entrada la matriz de magnitud del gradiente mag, las matrices de gradiente xe y, y un HashMap h, y devuelve un vector de punteros a objetos Point. La función detecta los picos en la matriz de magnitud del gradiente y los almacena en el HashMap para su uso posterior en la función recursiveDT().

La función primero inicializa un vector v para almacenar los picos detectados. Luego, recorre cada píxel en la matriz de magnitud del gradiente, saltando la primera y última fila y columna para evitar casos de borde. Para cada píxel, la función calcula la pendiente del gradiente utilizando las matrices x e y. Si la magnitud del gradiente en el píxel actual es mayor que la de ambos vecinos en la dirección de la pendiente, el píxel se considera un pico y se almacenan sus coordenadas en el vector v.

```
double slope = 0;
vector<Point*> v;
for (int i = 1; i < height - 1; i++)
{
    for (int j = 1; j < width - 1; j++)
    {
```

La función utiliza una serie de declaraciones if-else para verificar la dirección del gradiente y determinar los vecinos apropiados para comparar. Si la pendiente está dentro de un cierto rango, la función compara la magnitud del píxel actual con sus vecinos izquierdo y derecho. Si la pendiente está en otro rango, la función compara la magnitud del píxel actual con sus vecinos diagonales. Si la pendiente está en otro rango, la función compara la magnitud del píxel actual con sus vecinos antidiagonales. Si la pendiente está fuera de todos estos rangos, la función compara la magnitud del píxel actual con sus vecinos superior e inferior.

```
if (slope <= tan(22.5) && slope > tan(-22.5))
{
    if (mag[i][j] > mag[i][j-1] && mag[i][j] > mag[i][j+1])
    {
```

Después de detectar los picos, la función los almacena en el HashMap h para su uso posterior en la función recursiveDT. El HashMap se utiliza para almacenar los picos como claves y sus valores correspondientes como false, indicando que aún no han sido procesados. El uso de un HashMap permite búsquedas y actualizaciones de O(1), lo que es más eficiente que usar un vector o una lista.

```
v.push_back(new Point(i, j));
h->insert(i, j);
```

En lo que a acceso a memoria se refiere, la función lee de las matrices mag, x e y y escribe en el vector v y el HashMap h. El patrón de acceso a memoria es principalmente secuencial.

```
void recursiveDT(double **mag, double **final, HashMap *h, HashMap *peaks, int a, int b, int flag)
```

La última función y probablemente la más importante, recursiveDT. Esta función toma como entrada la matriz de magnitud del gradiente mag, la matriz de salida final, dos HashMaps h y peaks, y las coordenadas a y b de un píxel. La función realiza una búsqueda recursiva de los píxeles que pertenecen a los bordes detectados y los marca en la matriz de salida final.

La función comienza verificando si el valor del píxel es menor que el umbral inferior, si está fuera de los límites de la imagen o si ya ha sido visitado antes. Si alguna de estas condiciones se cumple, la función sale. De lo contrario, se inserta el píxel actual en el HashMap h para indicar que ya ha sido visitado.

```
if (mag[a][b] < lo || a < 0 || b < 0 || a >= height || b >= width)
    return;
if (h->contains(a, b))
    return;
```

Si el indicador flag es cero, la función busca un píxel en la matriz de salida final que esté marcado como true o on. Si se encuentra uno, se establece el indicador flag en uno y se sale de los bucles. Si el indicador flag es uno, la función busca píxeles en la matriz de magnitud del gradiente mag que estén en el rango medio y que también estén en el HashMap peaks. Para cada píxel encontrado, la función llama a sí misma recursivamente con las nuevas coordenadas y establece el píxel actual en la matriz de salida final como true o on.

```
if (!flag){
    for (int p = -1; p <= 1; p++){
        for (int q = -1; q <= 1; q++){
            if (final[a+p][b+q] == 255){
                final[a][b] = 255;
                flag = 1;
                break;
            }
        }
        if (flag)
            break;
    }
}
```

Al igual que en peak_detection, la anterior función, la memoria se accede de manera aleatoria debido a la búsqueda de píxeles en las matrices y los HashMaps, lo que puede causar fallos de caché y afectar el rendimiento. Para contrarrestar esto, el algoritmo utiliza HashMaps para búsquedas y actualizaciones eficientes. Además utiliza la recursión que en algunos casos, puede mejorar el rendimiento del programa.

Tarea 2.3 - Estudio de la variación de carga

El programa toma como argumentos del programa el límite de los bordes de la imagen, que al ser este un valor escalar no afectará a la carga del programa. Como segundo argumento tomará un valor sigma, este valor sigma determinará el detalle de la imagen que vamos a generar, esto quiere decir que a un valor sigma más alto, se produce mayor suavizado y por ende un mayor detalle y con un valor de sigma más bajo, se produce un detallado menor.

A parte de estos, habrá que darle una imagen para que pueda procesarla, esta será el punto de inflexión a la hora de definir cuánto puede llegar a tardar en ejecutar el programa.

Tarea 2.4 - Sintonización de los parámetros de compilación

Para la realización de estas pruebas hemos utilizado dos imágenes, neurona.pgm es una imagen con una resolución de 225x225 y la otra es zelda.pgm, esta tiene una resolución de 512x512.

Los parámetros que vamos a utilizar son los siguientes:

- ***-O0;*** Este parámetro viene por defecto, ya que no utiliza ningún tipo de optimización.
- ***-O1;*** Realiza en el código unas optimizaciones básicas, lo que crea un código rápido sin consumir mucho tiempo de compilación y mejora el rendimiento general del código.
- ***-O2;*** Realiza una optimizaciones más agresivas, esto genera un mayor tiempo al compilar el código, pero permite mejorar el rendimiento del código notablemente.
- ***-O3;*** Realiza una serie de optimizaciones adicionales a -O2, lo que conlleva a incrementar el tiempo de compilación considerablemente, así como el rendimiento del código y la memoria.

Vamos a realizar una serie de pruebas con las dos imágenes:

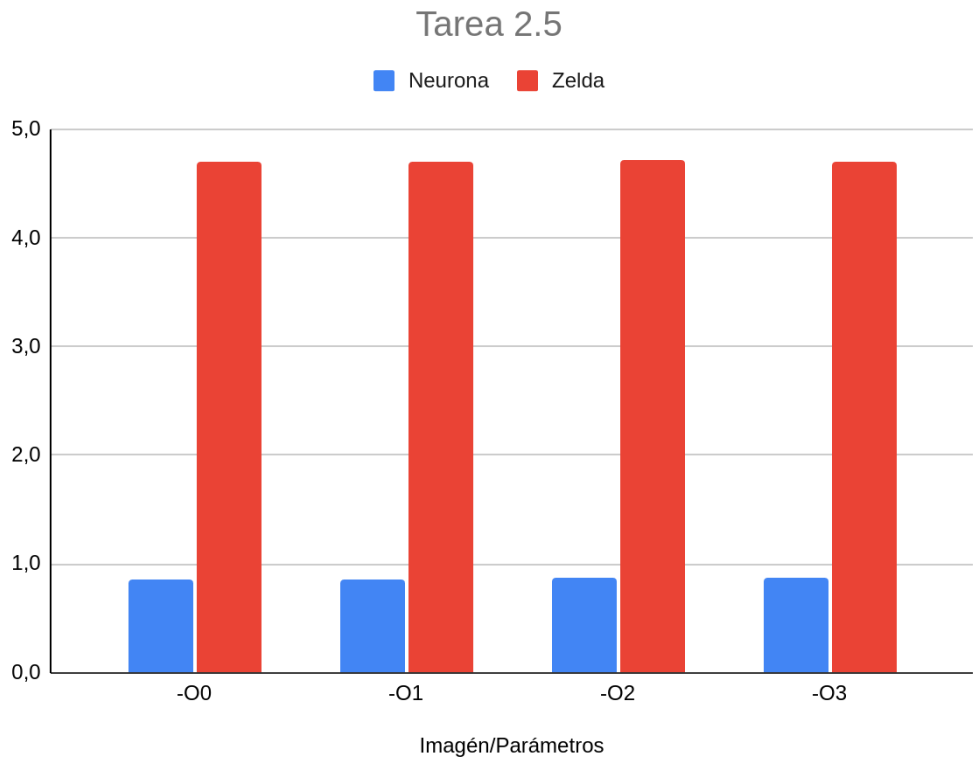
NEURONA.PGM

Pruebas/Parámetros	*-O0*	*-O1*	*-O2*	*-O3*
Prueba 1	0.86s	0.86s	0.86s	0.86s
Prueba 2	0.87s	0.86s	0.86s	0.87s
Prueba 3	0.86s	0.86s	0.87s	0.86s
Prueba 4	0.86s	0.86s	0.88s	0.87s
Media	0.8625s	0.860s	0.8675s	0.865s

ZELDA.PGM

Pruebas/Parámetros	*-O0*	*-O1*	*-O2*	*-O3*
Prueba 1	4,69	4,70	4,75	4,70
Prueba 2	4,70	4,71	4,70	4,71
Prueba 3	4,72	4,71	4,70	4,71
Prueba 4	4,70	4,69	4,74	4,71
Media	4,705	4,7025	4,7225	4,7075

Tarea 2.5 - Comparativa de gráficos de la tarea 2.4



En este gráfico podemos observar de una manera más clara el rendimiento del programa, de color azul tenemos el tiempo medio de ejecución en los diferentes parámetros de optimización de la imagen *Neurona* que tiene una resolución de 225x225, de color rojo tenemos los mismos tipos de datos, pero pertenecientes a la imagen *Zelda*.

Como vemos, las diferencias entre los tiempos de ejecución del programa son indistinguibles cuando se trata de la misma imagen, y cuando utilizamos los parámetros de optimización no obtenemos una mejora significativa en el rendimiento del programa, esto puede deberse a que el coste computacional del programa es relativamente bajo, ya que este solo tarda en ejecutar unos cuantos segundos en la imagen de 512x512, podemos suponer que si tratamos imágenes de mayor resolución se verán diferencias significativas en los parámetros de optimización debido a que parece presentar un aumento exponencial en el coste a medida que aumentamos la resolución.

De todas maneras, si nos fijamos en los datos de la tabla, podemos suponer dos cosas, que el parámetro *-O2* parece ser un poco inestable, ya que en dos pruebas ha dado picos que no han sido realmente significativos, pero sí notables, y como *-O2* parece no ser muy buen candidato para este tipo de problema, nos quedaremos con -O1 que parece ser la opción más estable proporcionando una pequeña mejora en el rendimiento cuando tratemos con imágenes de mayor resolución.

Tarea 2.6 - Estudio del rendimiento

Después de analizar el código del programa, hemos identificado la presencia de un total de siete bucles de tipo For anidados. De estos siete bucles, cinco se utilizan para iterar a través de la matriz de píxeles de la imagen. Los dos bucles restantes desempeñan un papel específico en la detección de los píxeles que se encuentran en ciertos límites, con el propósito de determinar si forman parte del borde de la imágen. Es importante destacar que estos dos últimos bucles no recorren la matriz al completo y, por tanto, tienen un impacto relativamente menor en el rendimiento general del programa. Por esta razón, vamos a centrar nuestra atención en los cinco bucles anidados que recorren la matriz de píxeles en su totalidad.

De estos cinco bucles, cuatro de ellos (un 80% del total), se dedican al cálculo de los diferenciales necesarios para obtener las magnitudes de los picos. Estas magnitudes se utilizan posteriormente para determinar si los píxeles correspondientes forman parte o no del borde de la imagen.

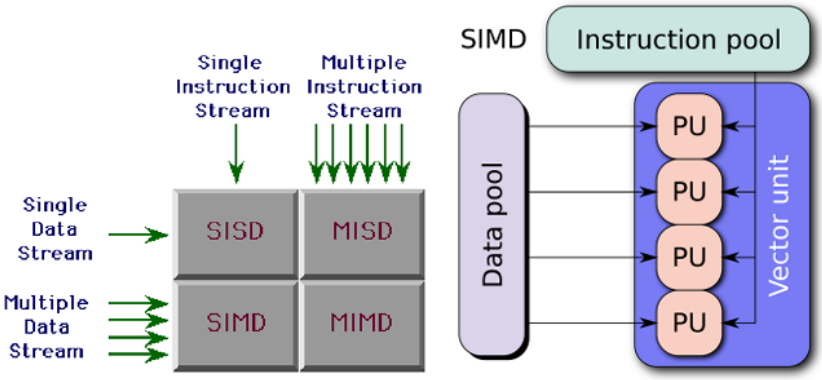
En lo que respecta a la aceleración mejorada, se utiliza el número de núcleos del ordenador de Iván como parámetro. Dicho ordenador cuenta con un total de 8 núcleos, y este valor se emplea en el proceso de aceleración para optimizar el rendimiento del programa.

$$\text{Aceleración} = \frac{1}{(1 - \text{Fracción Mejorada}) + \frac{\text{Fracción Mejorada}}{\text{Aceleración Mejorada}}} = \frac{1}{(1 - 0{,}8) + \frac{0{,}8}{8}} = \frac{10}{3} \approx 3{,}3$$

Tras realizar la fórmula de la Ley de Amdahl, obtenemos que el factor de mejora del programa sería 3,3.

Tarea 2.7 - Paralelización del candidato

El algoritmo de Canny es un buen candidato para la paralelización debido a un alto grado de paralelismo dado que la mayoría de los cálculos se realizan en paralelo en cada píxel, es posible aprovechar al máximo los núcleos de una CPU multicore o unidades de procesamiento en una GPU, este algoritmo se beneficia de arquitecturas de hardware paralelo, lo que significa que se puede ejecutar de manera eficiente en sistemas con estas características.



Dentro de la taxonomía de Flynn, encontramos varias clasificaciones referenciadas al trabajo interno del funcionamiento del almacenamiento de datos como pueden ser SISD, MISD o MIMD, sin embargo, vamos a usar la arquitectura SIMD (Single Instruction, Multiple Data) ya que, como su propio nombre indica, consiste en tener una sola instrucción con un gran número de datos simultáneamente lo que sería idóneo para la ejecución de un programa débilmente paralelizado.

En resumen, la naturaleza local e independiente de las operaciones en Canny hace que sea un algoritmo ideal para la paralelización, lo que conduce a un procesamiento más rápido y eficiente de imágenes, especialmente en entornos de alto rendimiento y resolución.

Tarea 4 - Precalentamiento siguientes prácticas

• Un grifo tarda 4 horas en llenar un cierto depósito de agua y otro grifo 20 horas en llenar el mismo depósito. Si usamos los dos grifos para llenar el depósito, que está inicialmente vacío, ¿cuánto tiempo tardaremos? ¿Cuál será la ganancia en velocidad? ¿Y la eficiencia?

Estos son los datos que nos ofrece el enunciado para resolver el problema:

- **Tasa del primer grifo** = 1/4 (llena 1/4 del depósito por hora).
- **Tasa del segundo grifo** = 1/20 (llena 1/20 del depósito por hora).

A continuación, a partir de haber sacado los datos, podemos obtener el tiempo que tardará en llenarse el depósito (tiempo total), la ganancia en velocidad y la eficiencia usando este método.

Tiempo total = 1 / (Tasa 1 + Tasa 2 + Tasa n …) = 1 / (1/4 + 1/20) = 1 / (5/20 + 1/20) = 1 / (6/20) = 1 / (3/10) = 10/3 horas = 3 horas y 20 minutos.

Ganancia en velocidad = Tiempo con un solo grifo - Tiempo con ambos grifos = 4 horas - 3 horas 20 minutos = 40 minutos.

Eficiencia = (Tiempo con un solo grifo / Tiempo con ambos grifos) * 100% = (4 horas / 3 horas 20 minutos) * 100% ≈ 120%.

• Suponga que tiene ahora 2 grifos de los que tardan 4 horas en llenar el depósito. Mismas cuestiones que el punto anterior.

Estos son los datos que nos ofrece el enunciado para resolver este apartado del problema:

- **Tasa de cada grifo** = 1/4 (llenan 1/4 del depósito por hora).

Seguidamente, a partir de estos, calcularemos las mismas cuestiones que anteriormente.

Tiempo total = 1 / (Tasa 1 + Tasa 2 + Tasa n …) = 1 / (1/4 + 1/4) = 1 / (2/4) = 1 / (1/2) = 2 horas.

Ganancia en velocidad = Tiempo con un solo grifo - Tiempo con ambos grifos = 4 horas - 2 horas = 2 horas.

Eficiencia = (Tiempo con un solo grifo / Tiempo con ambos grifos) * 100% = (4 horas / 2 horas) * 100% = 200%.

• Y ahora suponga que tiene 2 grifos de los que tardan 20 horas. Proceda también a realizar los cálculos.

Estos son los datos que nos ofrece el enunciado para resolver este apartado del problema:

- **Tasa de cada grifo** = 1/20 (llenan 1/20 del depósito por hora).

Del mismo modo que anteriormente, con los datos sacados calcularemos las cuestiones ofrecidas.

Tiempo total = 1 / (Tasa 1 + Tasa 2 + Tasa n …) = 1 / (1/20 + 1/20) = 1 / (2/20) = 1 / (1/10) = 10 horas.

Ganancia en velocidad = Tiempo con un solo grifo - Tiempo con ambos grifos = 20 horas - 10 horas = 10 horas.

Eficiencia = (Tiempo con un solo grifo / Tiempo con ambos grifos) * 100% = (20 horas / 10 horas) * 100% = 200%.

• Ahora tiene 3 grifos: 2 de los que tardan 20 horas y 1 de los que tardan 4. ¿Qué pasaría ahora?

Estos son los datos que nos ofrece el enunciado para resolver este apartado del problema:

- **Tasa de los dos grifos de 20 horas cada uno** = 1/20 (cada uno llena 1/20 del depósito por hora).

- **Tasa del tercer grifo de 4 horas** = 1/4 (llena 1/4 del depósito por hora).

A continuación, con estas 3 tasas, acabamos los apartados del problema calculando las mismas cuestiones.

Tiempo total = 1 / (Tasa 1 + Tasa 2 + Tasa n …) = 1 / (1/20 + 1/20 + 1/4) = 1 / (2/20 + 2/20 + 5/20) = 1 / (9/20) = 20/9 horas ≈ 2 horas y 13 minutos (aproximadamente).

Ganancia en velocidad = Tiempo con un solo grifo - Tiempo con ambos grifos = 4 horas - 2 horas y 13 minutos = 1 hora y 47 minutos.

Eficiencia = (Tiempo con un solo grifo / Tiempo con ambos grifos) * 100% = (4 horas / 2 horas y 13 minutos) * 100% = 180,45%.

MakeFile

En este apartado, explicaremos las diferentes opciones del makefile:

- ***make***: Compilación normal del programa. Ej: ***g++ -o canny.out main.o canny.o HashMap.o***
- ***make run***: Ejecución con los parámetros por defecto high threshold = 100, sigma = 1 y la imagen se introducirá de la siguiente manera image=<image>.
- ***make clean***: Limpia los archivos *.o y *.out y las imágenes de la carpeta de output_images.

Bibliografía

Algoritmo de Canny: Algoritmo de Canny - Wikipedia, la enciclopedia libre

Algoritmo de Sobel: Operador Sobel - Wikipedia, la enciclopedia libre

Código fuente: sorazy/canny: A C++ implementation of Canny's edge detection method on .pgm images. (github.com)

Taxonomía de Flynn: Taxonomía de Flynn - Wikipedia, la enciclopedia libre