INGENIERÍA DE LOS COMPUTADORES

PROPUESTA Y ESTUDIO DE PARALELIZACIÓN

Casado Lorenzo, Oscar

Brahim García, Adam

Guerrero Montero, Darío

Ros Mateo, Nicolás

Serrano Sánchez, Irene

Universidad de Alicante

Escuela Politécnica Superior

Índice

1. Objetivos

El objetivo de esta segunda práctica es el de encontrar un problema o aplicación con un coste computacional lo suficientemente elevado para que pueda ser paralelizado, por lo tanto, nuestro programa deberá ser secuencial. Una vez encontrado, deberemos entender y ser capaces de explicar por qué es una aplicación idónea para paralelizarlo y cuáles serían las arquitecturas para realizar dicha paralelización, así como su ganancia y eficiencia.

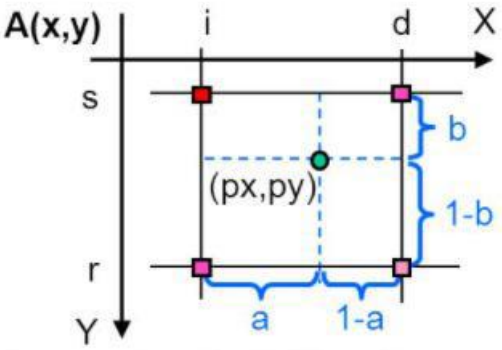
2. Presentación del problema

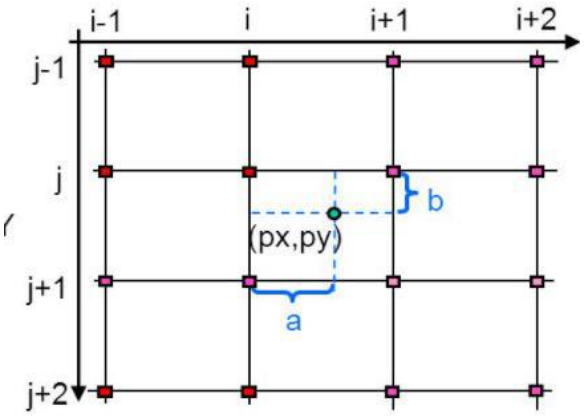
Una de las cosas más importantes que se puede hacer en el procesamiento de imágenes es escalarlas. El escalado de imágenes consiste en agrandar o reducir el tamaño físico de la imagen cambiando el número de píxeles que contiene. Cambia el tamaño del contenido de la imagen y redimensiona el lienzo en consonancia.

Existen 3 principales algoritmos para realizar esta tarea: el vecino más cercano, interpolación bilineal y el que vamos a tratar nosotros, la interpolación bicúbica.

Para tratar de entender bien este método de interpolación de imágenes explicaremos antes los otros dos para mayor claridad. El vecino más cercano es un método básico que requiere un tiempo de procesamiento bajo. Solo tiene en cuenta un píxel, el más cercano al punto interpolado. Lo único que hace es aumentar el tamaño de cada píxel.

La interpolación bilineal tiene en cuenta los valores en los píxeles conocidos que rodean a uno dado en una vecindad de los 2x2 píxeles más cercanos. Una vez los tiene se toma el promedio ponderado de estos 4 píxeles y se calcula el valor interpolado. El resultado está más suavizado que con el método anterior por el hecho de tomar el promedio, aunque esto acarrea más tiempo de procesamiento.





Un paso más allá del caso bilineal es considerar la vecindad de los 4x4 píxeles conocidos más cercanos, es decir, un total de 16 píxeles. Como están situados a distancias distintas del píxel de valor desconocido, se da mayor peso en el cálculo a los más cercanos.

Produce imágenes más nítidas que los dos métodos anteriores. Es un buen compromiso entre tiempo de procesado (ya que es la que mayor carga computacional posee) y calidad de resultado. Es un procedimiento estándar en programas de edición de imágenes, drivers de impresoras e interpolación en cámaras.

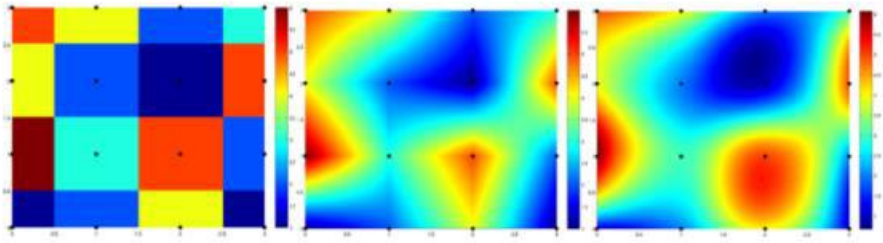


Figura : Interpolación mediante el vecino más cercano, bilineal y bicúbica

Pasamos ahora a explicar cómo se calcula matemáticamente la interpolación bicúbica.

Si los valores de una función f(x) y su derivada son conocidos en x=0 y x=1, entonces la función puede ser interpolada en el intervalo [0,1] usando un polinomio de tercer grado. A esto es a lo que se llama interpolación cúbica. La fórmula de este polinomio puede ser fácilmente derivada.

Un polinomio de tercer grado y su derivada:
$f(x)=ax3+bx2+cx+d$ $f(x)=3ax2+2bx+c$

Los valores del polinomio y de su derivada en x=0 y x=1:
$f(0)=d$ $f(1)=a+b+c+d$
$f'(0)=c$ $f'(1)=3a+2b+c$

Estas cuatro ecuaciones pueden ser escritas de la siguiente forma:
$a=2f(0)-2f(1)+f'(0)+f'(1)$
$b=-3f(0)+3f(1)-2f'(0)-f'(1)$
$c=f'(0)$ $d=f(0)$

Y ahí tendríamos nuestra fórmula de interpolación cúbica, lo siguiente sería implementar un programa en C++ para resolver este tipo de problemas para una lista n de valores, que será nuestra imagen.

3. Estudio pormenorizado de la aplicación

Antes de comenzar con la traza del programa nos gustaría destacar que el código ha sido implementado completamente por los componentes del equipo, los cuales no están especializados en el procesamiento de imágenes por lo que, puede que la solución que aportemos no sea la óptima.

Aclarado este punto, en cuanto a las librerías necesarias para la codificación del código tenemos:

- **Ctime:** para el control del tiempo de ejecución
- **CImg:** para la manipulación de imágenes (leer y editar valores RGB, generación de imágenes, etc)

Por otro lado, el módulo principal del programa, como en la mayoría de códigos C++ es el main(); en este inicializaremos el contador de tiempo que utilizaremos en apartados posteriores.

```
unsigned tiempoInicio = clock();

//Uno de los argumentos de entrada debe de ser el factor de escalado
//que deseamos aplicarle a la imagen

if(argc != 2) {
    usage();
    exit(0);
}

CImg<unsigned char> src("image.png"); //Imagen origen

int scaleFactor = atoi(argv[1]); //Factor por el cual queremos escalar la imagen
int width = src.width(); //Anchura de la imagen original
int height = src.height(); //Altura de la imagen original

//Imagen destino
CImg<unsigned char> dst(width*scaleFactor, height*scaleFactor, 1, 3, 255);

//Reescalamos la imágen conservando los pixeles iniciales
for (int x = 0; x < height; x++) {
    for (int y = 0; y < width; y++) {
        dst(x * scaleFactor, y * scaleFactor, 0, 0) = src(x, y, 0, 0);
        dst(x * scaleFactor, y * scaleFactor, 0, 1) = src(x, y, 0, 1);
        dst(x * scaleFactor, y * scaleFactor, 0, 2) = src(x, y, 0, 2);
    }
}
```

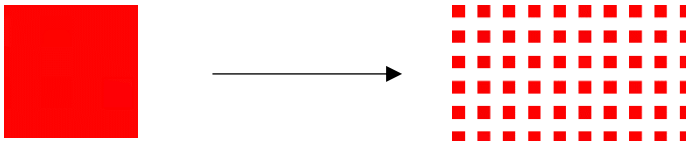
También llevaremos a cabo el control de argumentos donde intervendrá el módulo usage() encargado de mostrar al usuario, la introducción correcta de parámetros relacionados con el ejecutable.

```
void usage(){
        cout <<"./a.out [scale factor]" << endl;
}
```

A continuación, gracias a las diferentes estructuras de datos que nos proporciona la librería CImg almacenamos la imagen “image.png” que se encuentre en el repositorio en ese momento, que será sobre la que apliquemos las transformaciones basadas en la interpolación bicúbica.

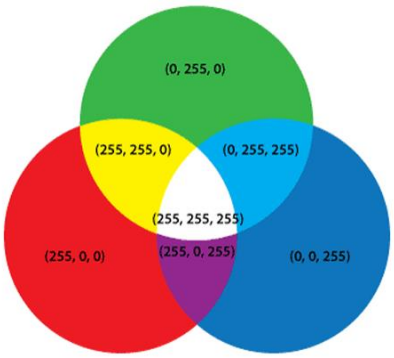
Seguidamente también inicializaremos otra serie de valores clave para el funcionamiento del algoritmo: scaleFactor (valor entero o decimal con respecto al cual se ampliará nuestra imagen, es decir, si una imagen es inicialmente de 100x100 pixeles y el scaleFactor lo establecemos a 5 entonces la imagen resultante poseerá 500x500 pixeles), width (ancho de la imagen original), height (alto de la imagen expresado en pixeles).

Posterior a esto, encontramos la inicialización de la imagen resultante, donde la escala sigue la dinámica recién explicada y el primer bucle del programa, en el que únicamente insertaremos los valores RGB de los pixeles que conociésemos de la siguiente forma:



Al inicio contamos con una imagen de la cual conocemos todos los pixeles (cuadrados rojos), al aumentarla por ejemplo con un factor de escalado 2, la cantidad de estos aumenta, dejándonos posiciones de las cuales desconocemos el valor RGB (cuadrados blancos). Sin embargo, como conocemos los valores originales podemos redistribuirlos a lo largo de la nueva imagen para así solo tener que rellenar la mitad de los pixeles gracias a la interpolación bicúbica.

Como inciso, me gustaría destacar como la librería accede e inserta valores RGB, y es que en la llamada ‘imgX(x,y,0, RGB)’, x e y se refieren a la coordenada de la matriz que va a ser utilizada, el siguiente argumento de la función es la profundidad de imagen (lo cual nos es indiferente en esta práctica, la estableceremos siempre a 0) y, por último la escala de color que vamos a tener en cuenta ‘RGB’ que puede tomar 3 valores: 0 (rojo), 1(verde), 2(azul) puesto que un color está compuesto por la combinación de estos valores.




```
//Recorremos pixel a pixel la imágen dst
for(int i = 0; i < width*scaleFactor; i++){
    for(int j = 0; j < height*scaleFactor; j++){
        //Si el pixel no es original...
        if(i % scaleFactor != 0 || j % scaleFactor != 0){
            //Debemos hallar los 4x4 pixeles originales mas cercanos.
            double colorMatrix[4][4];

            if(i != 0 && i != 1 && j != 0 && j != 1){

                buscarValores(scaleFactor, src, colorMatrix, i, j, 'R');
                //Otorgamos a cada color el resultado de interpolar cada matriz
                //Interpolamos el valor R
                dst(i, j, 0, 0) = interpolacionBicubica(colorMatrix, i, j);

                //Interpolamos el valor G
                buscarValores(scaleFactor, src, colorMatrix, i, j, 'G');
                dst(i, j, 0, 1) = interpolacionBicubica(colorMatrix, i, j);

                //Interpolamos el valor B
                buscarValores(scaleFactor, src, colorMatrix, i, j, 'B');
                dst(i, j, 0, 2) = interpolacionBicubica(colorMatrix, i, j);
            }
        }
    }
}
```

Siguiendo con la codificación del main(), ahora nos encontramos con el eje central del mismo, el bucle encargado de llamar a la generación de matrices RGB 4x4 ya que son los pixeles que la interpolación bicúbica necesita para generar un promedio.

Este proceso debemos realizarlos 3 veces, una por cada escala de color que existe, la matriz resultante de este procedimiento será algo similar a…

$$\begin{pmatrix} 255 & 165 & 43 & 234 \\ 221 & 234 & 34 & 249 \\ 32 & 34 & 123 & 94 \\ 12 & 129 & 123 & 87 \end{pmatrix}$$

Como se puede observar en la imagen adjuntada del código fuente, existen 2 funciones intemedias que emplearemos para llevar a cabo la asignación de valores a los pixeles que no son originales: buscarValores() e interpolaciónBicúbica().

Centrándonos en buscarValores() es el módulo encargado de hallar todas las coordenadas de los 16 vecinos que componen el cerco de la imagen original.

Además, dependiendo de la matriz que se quiera componer (la de rojos, la de verdes o la de azules) extraerá distintos valores, de ahí el switch para desempaquetar la información que nos llega en el argumento 'color'.

Finalmente encontramos un bucle que recorre 4 columnas, y por cada una de ellas 4 filas (el típico bucle de recorrido de matrices bidimensionales).

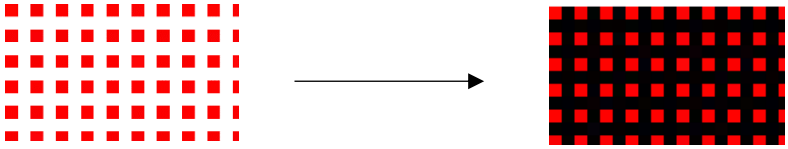
A partir de las coordenadas del píxel que queremos rellenar, calcula el campo en el que se encuentra pudiendo rellenar así todas las posiciones de la matriz, para que esta sea devuelta por referencia y estudiada posteriormente por las funciones auxiliares que explicaremos a continuación.

```
void buscarValores(int scaleFactor, CImg<unsigned char> src,
                   double colorMatrix[4][4], int cordX, int cordY, char color){
    //Pasamos el char al valor correspondiente para la funcion src(x,i, 0, RGB)
    int rgb = -1;
    switch(color){
        case 'R': rgb = 0;
        break;

        case 'G': rgb = 1;
        break;

        case 'B': rgb = 2;
        break;
    }


    //Bucle para rellenar la matriz
    for(int x = 0; x < 4; x++){
        for(int y = 0; y < 4; y++){
            int pixelXOriginal;
            int pixelYOriginal;

            //Pixeles adyacentes
            if(cordX/scaleFactor - 1 <= 0)
                pixelXOriginal = 0;
            else pixelXOriginal = cordX/scaleFactor - 1 ;

            if(cordY/scaleFactor - 1 <= 0)
                pixelYOriginal = 0;
            else pixelYOriginal = cordY/scaleFactor - 1 ;

            pixelXOriginal++;
            pixelYOriginal++;

            colorMatrix[x][y] = (double)src(pixelXOriginal, pixelYOriginal, 0, rgb);
        }
```

En cuanto a las funciones auxiliares que se encargan de reinterpretar matemáticamente las matrices para obtener una ponderación de los 16 pixeles adycentes acada uno hasn sifo implementadas como se muestra en la imagen adjuntada a continuación. No entraremos en detalles sobre los motivos de esta codificación ya que se ha explicado en repetidas ocasiones y seguramente sean modificadas de cara a prácticas posteriores para favorecer a una implementación paralela.

```
double interpolacionCubica (double p[4], double x) {
    return p[1] + 0.5 * x*(p[2] - p[0] + x*(2.0*p[0] - 5.0*p[1] + 4.0*p[2] - p[3] + x*(3.0*(p[1] - p[2]) + p[3] - p[0])));
}

double interpolacionBicubica (double p[4][4], double x, double y) {
    double arr[4];
    arr[0] = interpolacionCubica(p[0], y);
    arr[1] = interpolacionCubica(p[1], y);
    arr[2] = interpolacionCubica(p[2], y);
    arr[3] = interpolacionCubica(p[3], y);
    return interpolacionCubica(arr, x);
}
```

En conclusión, cogeríamos la matriz de la que hemos hablado al principio del apartado que contenía los pixeles originales redistribuidos a lo largo de la nueva imagen y rellenaríamos los valores de las coordenadas desconocidas obteniendo así una imagen reescalada en la que apenas se apreciará una pérdida de calidad gracias al algoritmo de reescalado empleado. En esta ejemplificación los pixeles negros corresponderían todos aquellos que deberían ser modificados por el programa.



3.1. ¿Es paralelizable según el CFG?

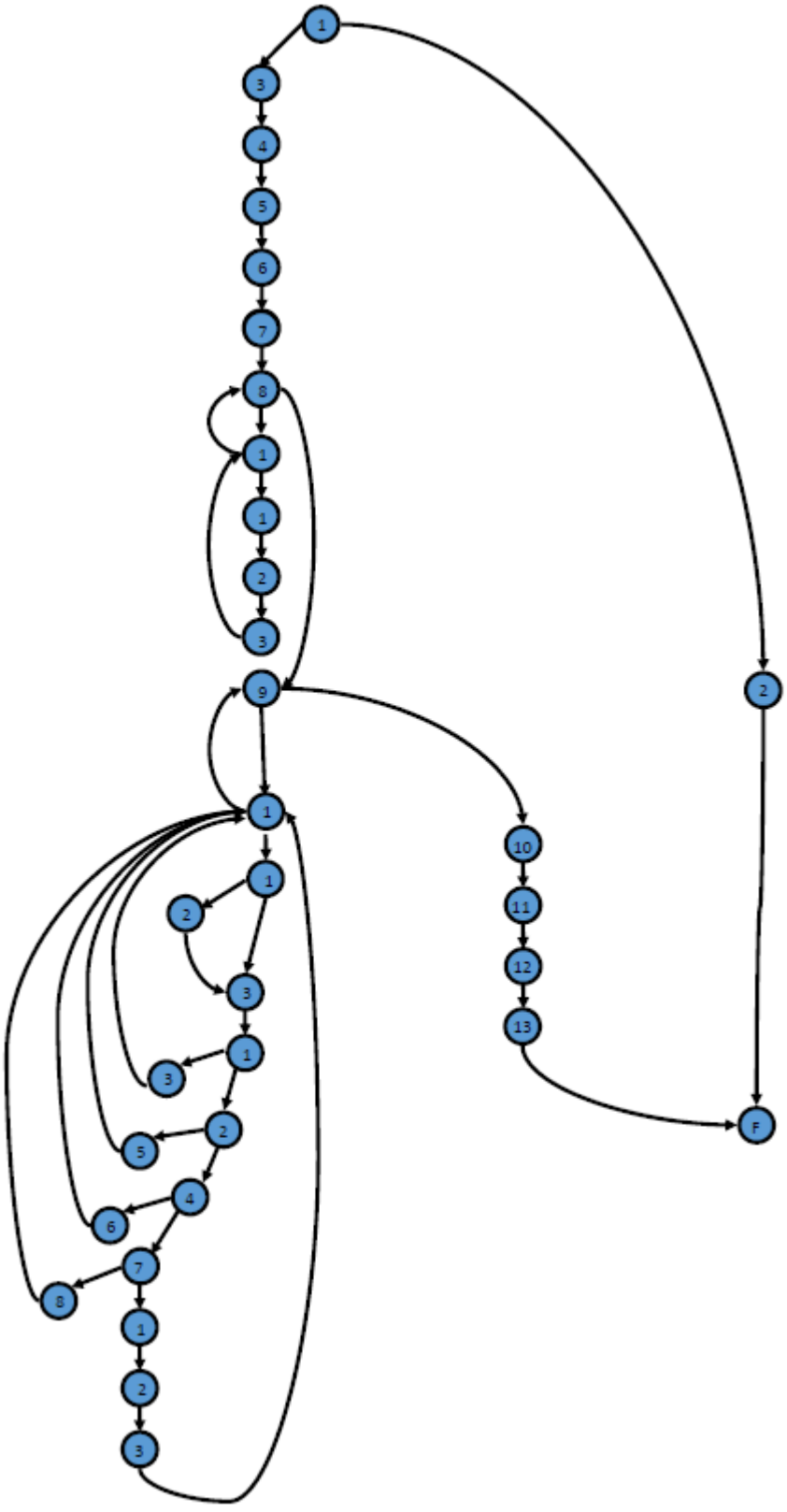
Dado el CFG podemos observar que nuestro problema sí que es paralelizable y lo podemos ver en el Main y en la función buscarValores() donde tomamos varias decisiones y estas se podrían efectuar al mismo tiempo paralelizando el programa.

3.2. Problemas de caché y variables

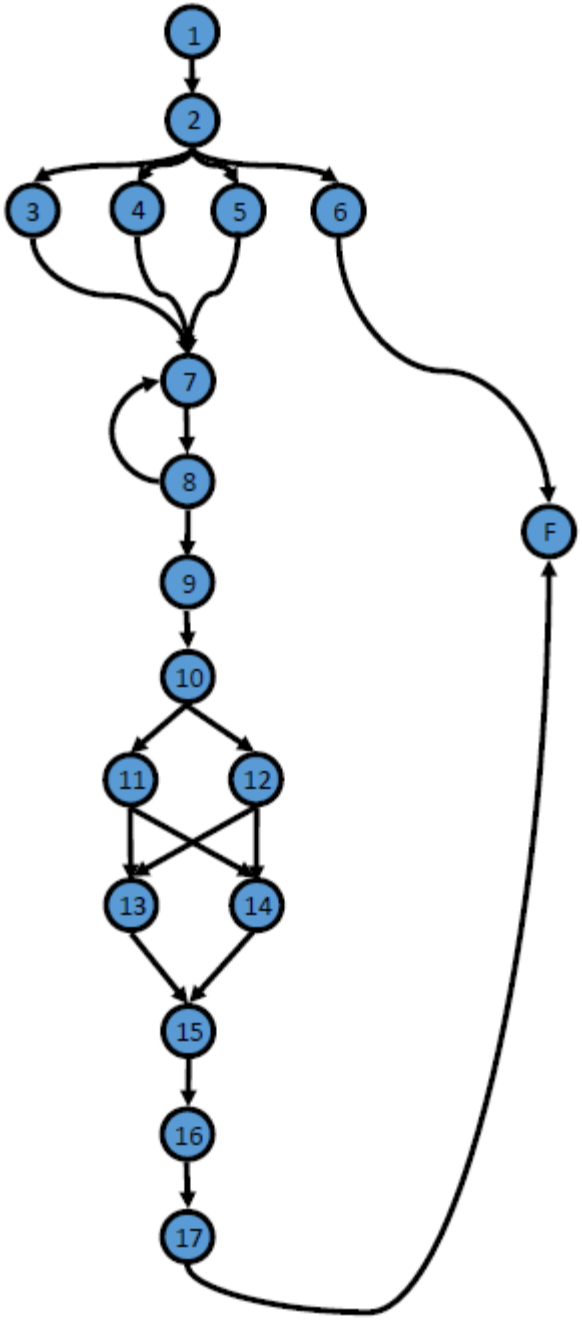
De acuerdo con nuestro código utilizamos pocas variables y a estas accedemos constantemente (una matriz del pixel de la imagen, el valor RGB, la posición X, Y en la que nos encontramos…).
En caso de paralelizar el apartado del Main donde otorgamos a cada color el resultado de interpolar cada una de las matrices 4x4 podría dar un problema de caché ya que estaríamos intentando acceder a los mismos datos.

3.3. Carga del problema

Para variar la carga del problema podemos hacerlo de diversas formas. Podemos modificar la imagen que modificará el programa por una de mayor o menor tamaño, modificar el valor por el que multiplicamos la imagen o pasando varias imágenes al mismo tiempo (modificando un poco el código). Inicialmente las pruebas que haremos serán modificando una única imagen dado que si ampliamos el tamaño al que la queremos ampliar puede llegar a aumentar mucho su tiempo.



buscarValores();



interpolacionBicubica()

Usage()

interpolacionCubica()

3. Parámetros de compilación

3.1. Parámetros de tipo -Ox

El nivel de optimización predeterminado del GCC es **-O0**. Según el manual del GCC: "Cuando no hay ninguna opción de optimización, el objetivo del compilador es reducir el costo de compilación y hacer que la depuración produzca los resultados esperados". Entonces, el código no se optimizará.

El nivel de optimización más básico es el parámetro **-O1**, el compilador intentará producir un código rápido y pequeño sin tomar mucho tiempo de compilación. Es básico, pero conseguirá realizar correctamente el trabajo.

El siguiente sería el **–O2**, es el nivel recomendado de optimización, a no ser que el sistema tenga necesidades especiales. -O2 activará algunas opciones añadidas a las que se activan con -O1. Con -O2, el compilador intentará aumentar el rendimiento del código sin comprometer el tamaño y sin tomar mucho más tiempo de compilación.

Y, por último **-O3** es el nivel más alto de optimización posible. Activa optimizaciones que son caras en términos de tiempo de compilación y uso de memoria. Sin embargo, el hecho de compilar con -O3 no garantiza una forma de mejorar el rendimiento, en algunos casos puede ralentizar un sistema debido al uso de binarios de gran tamaño y mucho uso de la memoria.

3.2. Parámetro –march

El parámetro **-march** indica qué código debería producir para su arquitectura de procesador. Diferentes CPUs tienen diferentes características, soportan diferentes conjuntos de instrucciones y tienen diferentes formas de ejecutar código.

Si no se puede determinar el tipo de CPU o si el usuario no sabe qué ajustes elegir, es posible utilizar el ajuste -march=native. Al usarla, GCC intentará detectar el procesador y automáticamente usará las opciones apropiadas.

3.3. Parámetro -Ofast

El parámetro **-Ofast** habilita todas las optimizaciones de -O3 así como también aquellas que no son válidas para todos los programas que cumplen con los estándares.

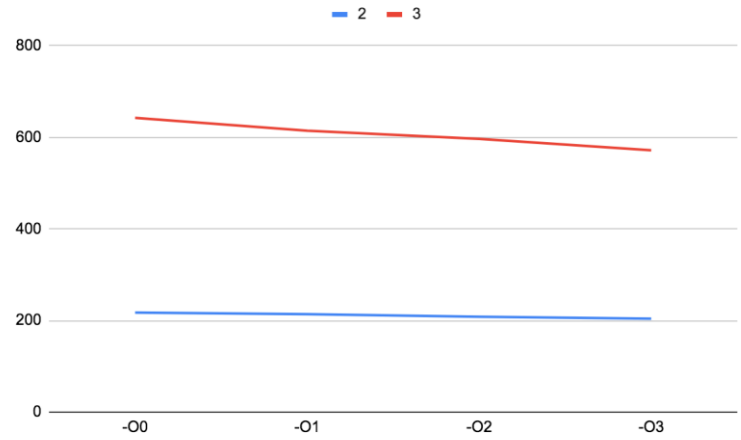
3.4. Parámetro -floop-parallelize-all

El parámetro **-floop-parallelize-all** paraleliza todos los bucles que pueden ser analizados para que no contengan dependencias transportadas por bucles sin comprobar que sea rentable para paralelizar esos bucles.

En primer lugar, hemos analizado los parámetros anteriores de tipo -Ox junto con el parámetro -march=native para estudiar cómo varía el tiempo de ejecución en una imagen de 512x512 píxeles aumentando su tamaño el doble y el triple de veces.
En la tabla que aparece a continuación se puede observar cómo el tiempo de ejecución ha ido descendiendo ligeramente cada vez que al compilar se iba acercando al parámetro de optimización -O3:
.

	-O0	**-O1**	**-O2**	**-O3**
scaleFactor=2	217,461	213,836	208,113	204,113
scaleFactor=3	642,524	614,603	596,603	571,885

En la siguiente gráfica se recogen los resultados obtenidos:



En segundo lugar, hemos utilizado el parámetro -Ofast. Después de realizar varias pruebas, hemos comprobado que no existe gran diferencia en cuanto a ganancia puesto que el tiempo de ejecución es similar a usar por separado cada una de las opciones de tipo -Ox. No descartamos esta opción pero tampoco es óptima. Los resultados obtenidos han sido los siguientes:

2	207,228 seg
3	593,431 seg

Finalmente hemos analizado el parámetro -floop-parallelize-all, respecto a los anteriores parámetros, el tiempo es superior a los que hemos mostrado, por lo tanto, no es recomendable usar este tipo de parámetro para nuestro programa ya que no se adapta bien a las instrucciones del procesador. Esto lo hemos comprobado añadiendo otro parámetro a esta opción y el resultado fue que el código no pudo ser compilado. De la misma forma que antes, los resultados que hemos obtenido han sido los siguientes:

2	237,596 seg
3	670,167 seg

En conclusión, para nuestra aplicación la mejor forma de optimizar el rendimiento sería usar tanto la opción -O2 como la -O3, gracias a esto se consigue un mejor aprovechamiento que se ve reflejado en el uso de la memoria.

4. Análisis de rendimiento

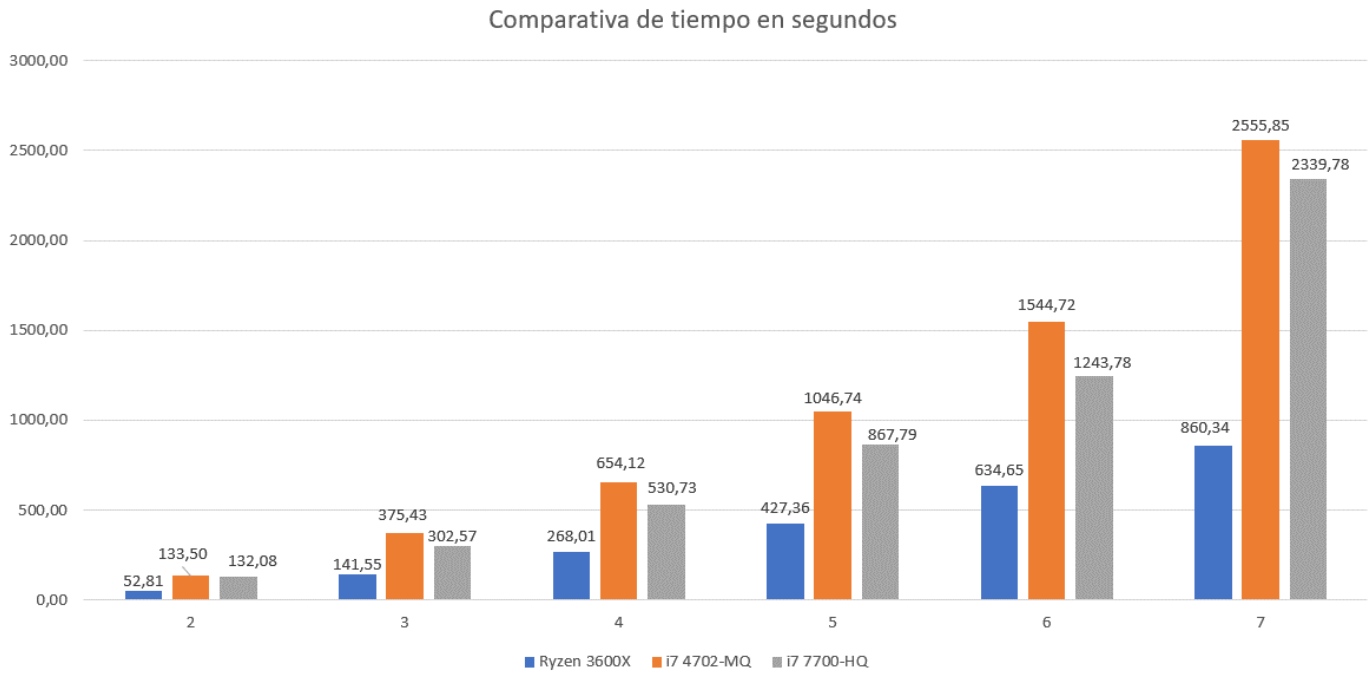
A continuación, estudiaremos la variación del rendimiento al variar la carga del problema, en nuestro caso, aumentando el número de aumentos de la imagen original.
Para ellos hemos utilizado una imagen de 512x512 pixeles la cual se ha aumentado hasta en 10 aumentos.
Se han utilizado 3 procesadores distintos para esta prueba a modo de comparar el rendimiento en diferentes generaciones de estos.

Velocidad de base:	2,20 GHz
Sockets:	1
Núcleos:	4
Procesadores lógicos:	8
Virtualización:	Habilitado
Caché L1:	256 kB
Caché L2:	1,0 MB
Caché L3:	6,0 MB

Velocidad de base:	2,81 GHz
Sockets:	1
Núcleos:	4
Procesadores lógicos:	8
Virtualización:	Habilitado
Caché L1:	256 kB
Caché L2:	1,0 MB
Caché L3:	6,0 MB

Velocidad de base:	3,80 GHz
Sockets:	1
Núcleos:	6
Procesadores lógicos:	12
Virtualización:	Habilitado
Caché L1:	384 kB
Caché L2:	3,0 MB
Caché L3:	32,0 MB

Cabe destacar que la prueba no se ha realizado con todos los aumentos en los dos procesadores Intel i7, ya que los tiempos se vuelven inviables debido al gran aumento en las pruebas, como veremos a continuación.

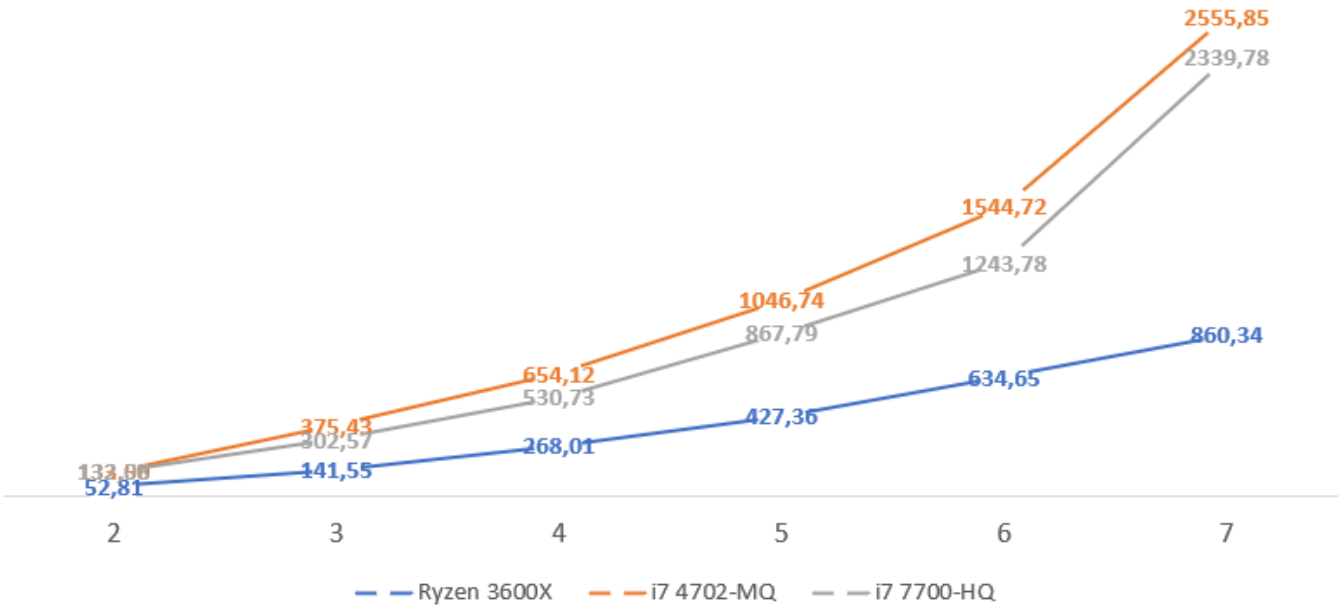


Como se puede observar en la gráfica de tiempos, cuanto mayor sea el número de aumentos, aumentará el tiempo de procesado considerablemente.

El procesador más actual, ryzen 3600x, tendrá un aumento de tiempo entre pruebas lineal, y será el más rápido en procesar todos los aumentos.

El procesador, i7 7700MQ, será el segundo más rápido, a pesar de que su aumento de tiempo será exponencial, por ello se ha decidido hacer las pruebas solamente hasta 7.

El procesador i7 4702MQ, ha sido el más lento, ya que su procesamiento del algoritmo ha sido totalmente exponencial llegando a ser 3 veces más lento en el procesamiento de 7 aumentos de la imagen.



5. Defensa del programa

Consideramos que este problema es un buen candidato ya que se ajusta a los dos tipos de paralelización que conocemos.

En primer lugar, tenemos que existe **paralelización a nivel de carga** ya que al final el programa lo que está haciendo es coger matrices n*n a lo largo de toda la superficie de la imagen, y ya una vez las tiene, les aplica el algoritmo.
Se podría ir aumentando tanto el tamaño de esas matrices como el número de ellas según la cantidad de píxeles de la imagen original, lo que haría que la complejidad de los cálculos que estamos realizando aumente y por ende, que la paralelización sea necesaria.
La obtención de estas matrices se podría paralelizar perfectamente haciendo que varias unidades funcionales sean las encargadas de conseguirlas.

En segundo lugar, tenemos que existe **una paralelización a nivel funcional** dentro de la propia operación que le estamos aplicando a cada una de las matrices, ya que esta incluye la resolución de polinomios de tercer grado y una vez resueltos se les aplica la derivada.

Al final después de todo el desarrollo de ecuaciones puesto en el planteamiento vemos que el cálculo se reduce a resolver estas cuatro ecuaciones:

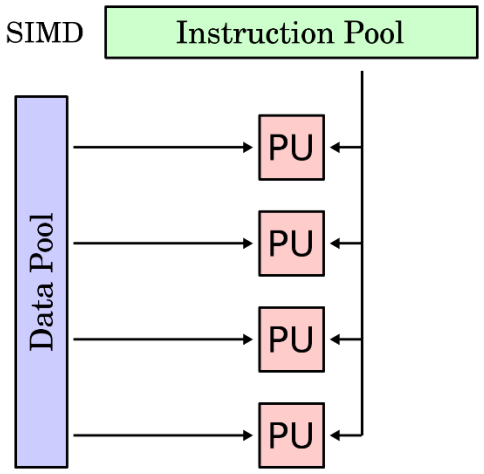
$$a = 2f(0) - 2f(1) + f'(0) + f'(1)$$
$$b = -3f(0) + 3f(1) - 2f'(0) - f'(1)$$
$$c = f'(0)$$
$$d = f(0)$$

Esas cuatro ecuaciones son independientes entre sí y, por tanto, su resolución se podría paralelizar perfectamente.

6. Arquitecturas paralelas

La arquitectura idónea según la clasificación de Flynn para la realización del cálculo de la matriz inversa mediante paralelización es la SIMD (Single Instruction Stream, Multiple Data Stream) Los repertorios SIMD consisten en instrucciones que aplican una misma operación sobre un conjunto más o menos grande de datos.

Es una organización en donde una única unidad de control común despacha las instrucciones a diferentes unidades de procesamiento. Todas estas reciben la misma instrucción, pero operan sobre diferentes conjuntos de datos. Ya que lo que tenemos es un problema paralelizable a nivel de datos, las operaciones a realizar sobre la matriz van a ser siempre las mismas, por lo que no necesitamos varios flujos de instrucción. Lo que necesitamos es dividir los datos de entrada y asignar cada uno de ellos a una unidad funcional diferente.



7.Bibliografía

C++ - Algoritmo de interpolación bi-cúbico para la escala de la imagen. (s. f.). https://stackoverrun.com. https://stackoverrun.com/es/q/4115337

C++ Image interpolation with Bicubic method. (2016, 7 abril). Stack Overflow. https://stackoverflow.com/questions/36469357/c-image-interpolation-with-bicubic-method

D. (s. f.). *Resizing Images With Bicubic Interpolation*. The blog at the bottom of the sea. https://blog.demofox.org/2015/08/15/resizing-images-with-bicubic-interpolation/

Ejemplos con OpenCV - OpenCV 3.1 con C++ en Visual Studio 2015. (s. f.). http://aprendiendo-programacion.wikidot.com. http://aprendiendo-programacion.wikidot.com/ejemplosocv

Juan V. Carrillo (jvprofe). (s. f.). *Prueba del camino básico. Pruebas de caja blanca*. YouTube. https://www.youtube.com/watch?v=GVegCwwfBZ0

Juan V. Carrillo (jvprofe). (2016, 6 mayo). *Cómo crear el grafo de flujo de un programa*. YouTube. https://www.youtube.com/watch?v=9N5vPeSWRfQ

makefile - Pasando argumentos para «ejecutar». (s. f.). https://stackoverrun.com. https://stackoverrun.com/es/q/456475

Overall Options (Using the GNU Compiler Collection (GCC)). (s. f.). https://gcc.gnu.org.

Profesor Retroman. (2019, 28 noviembre). *C++ : Compilar y enlazar librerías estáticas* [Vídeo]. YouTube. https://www.youtube.com/watch?v=MG8z-k6lH6E&ab_channel=ProfesorRetroman