PRÁCTICA 1
REDUCCIÓN DE LA RESOLUCIÓN DE UNA IMAGEN USANDO ALGORITMOS PARALELIZADOS

Jimmy Alexander Pulido Arias[1], Alvaro Andres Garcia Perdomo[1], y Camilo Andrés Díaz Silva[1]
[1]Departamento de Ingeniería de Sistemas e Industrial, Universidad Nacional de Colombia

***Index Terms*—Paralelización, Imagen, Algoritmo, C, pthread, OpenCV.**

I. INTRODUCCIÓN

ESTE documento presentará los resultados de la paralelización del algoritmo de reducción de imagen "Bilinear image scaling". En el proceso fue utilizado el lenguaje C y la librería pthread, además de la librería OpenCV que nos permitió convertir las imágenes en matrices usadas para obtener la imagen en una resolución más baja. Se describirá el algoritmo completo con detalle y se mostrarán las métricas que muestran que los esfuerzos de paralelizar el algoritmo rinden frutos y realizan esta reducción de resolución en un tiempo menor.

II. MARCO TEÓRICO

II-A. Algoritmo de reducción de resolución por interpolación lineal[1]

II-A1. Interpolación lineal

Digamos que tenemos dos puntos en una linea recta con coordenadas $\boldsymbol{a}$ y $\boldsymbol{b}$, que están asociados con los valores $\boldsymbol{A}$ y $\boldsymbol{B}$. Ahora si tenemos un tercer punto con coordenada $\boldsymbol{x}$ donde $\boldsymbol{a} \leq \boldsymbol{x} \leq \boldsymbol{b}$ ¿Cómo interpolamos los valores desde las coordenadas $\boldsymbol{a}$ y $\boldsymbol{b}$ hacia la coordenada $\boldsymbol{x}$?

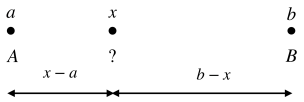


Figura 1. Interpolación lineal de dos puntos

La interpolación lineal lo calcula como el promedio ponderado de los valores asociados a los dos puntos, donde los ponderadores son proporcionales a la distancia entre $\boldsymbol{x}$ y $\boldsymbol{a}$, y $\boldsymbol{x}$ y $\boldsymbol{b}$

$$\boldsymbol{X} = \boldsymbol{A}\frac{\boldsymbol{b}-\boldsymbol{x}}{\boldsymbol{b}-\boldsymbol{a}} + \boldsymbol{B}\frac{\boldsymbol{x}-\boldsymbol{a}}{\boldsymbol{b}-\boldsymbol{a}} \tag{1}$$

o

$$\boldsymbol{X} = \boldsymbol{A}(\boldsymbol{1}-\boldsymbol{w}) + \boldsymbol{B}\boldsymbol{w} \tag{2}$$

donde $\boldsymbol{w} = \frac{\boldsymbol{x}-\boldsymbol{a}}{\boldsymbol{b}-\boldsymbol{a}}$ El peso de $\boldsymbol{A}$ es proporcional a la distancia de $\boldsymbol{x}$ a $\boldsymbol{b}$ (en vez de $\boldsymbol{a}$), mientras que el peso de $\boldsymbol{B}$ es proporcional a su distancia a $\boldsymbol{a}$ (en vez de $\boldsymbol{b}$). Cuando $\boldsymbol{x}$ se mueve a $\boldsymbol{a}$ su valor se vuelve $\boldsymbol{A}$; similarmente, $\boldsymbol{X}$ se vuelve $\boldsymbol{B}$ cuando $\boldsymbol{x}$ se mueve a $\boldsymbol{b}$

II-A2. Interpolación bilineal

Ahora podemos generalizar este proceso a dos dimensiones, hacemos interpolaciones lineales en $\boldsymbol{x}$ y(ancho) $\boldsymbol{y}$ (alto). Supongamos que tenemos cuatro puntos con coordenadas $(\boldsymbol{y_1}, \boldsymbol{x_1}$, $(\boldsymbol{y_1}, \boldsymbol{x_2}$, $(\boldsymbol{y_2}, \boldsymbol{x_1}$, $(\boldsymbol{y_2}, \boldsymbol{x_2}$ y sus valores asociados $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$ y $\boldsymbol{D}$

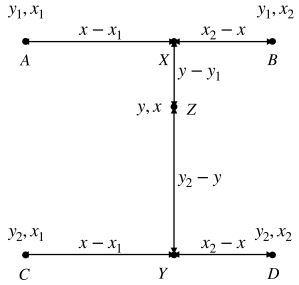


Figura 2. Interpolación lineal de dos puntos

Primero calculamos el valor de $\boldsymbol{X}$ y $\boldsymbol{Y}$ en la dimensión del ancho

$$\boldsymbol{X} = \boldsymbol{A}(\boldsymbol{1}-\boldsymbol{w_x}) + \boldsymbol{B}\boldsymbol{w_x} \tag{3}$$

$$\boldsymbol{Y} = \boldsymbol{C}(\boldsymbol{1}-\boldsymbol{w_x}) + \boldsymbol{D}\boldsymbol{w_x} \tag{4}$$

Luego simplemente realizamos una interpolación entre los dos valores interpolados $\boldsymbol{X}$ y $\boldsymbol{Y}$ en la dimensión de la altura

$$\begin{aligned}\boldsymbol{Z} &= \boldsymbol{X}(\boldsymbol{1}-\boldsymbol{w_y}) + \boldsymbol{Y}\boldsymbol{w_y} \\ \boldsymbol{Z} &= \boldsymbol{A}(\boldsymbol{1}-\boldsymbol{w_x})(\boldsymbol{1}-\boldsymbol{w_y}) + \boldsymbol{B}\boldsymbol{w_x}(\boldsymbol{1}-\boldsymbol{y_y}) \\ &\quad + \boldsymbol{C}(\boldsymbol{1}-\boldsymbol{w_x})\boldsymbol{w_y} + \boldsymbol{D}\boldsymbol{w_x}\boldsymbol{w_y}\end{aligned} \tag{5}$$

Donde $\boldsymbol{w_x} = \frac{\boldsymbol{x}-\boldsymbol{x_1}}{\boldsymbol{x_2}-\boldsymbol{x_1}}$ y $\boldsymbol{w_y} = \frac{\boldsymbol{y}-\boldsymbol{y_1}}{\boldsymbol{y_2}-\boldsymbol{y_1}}$.

III. ALGORITMO

El algoritmo consiste en transladar el método de interpolación lineal de la sección anterior al contexto de la reducción de imágenes. En esta sección describiremos el flujo del programa tal como el se ejecutaría. Lo primero es obtener la ruta de la imagen a escalar, luego el nombre de la nueva imagen, y finalmente el número de hilos que se usarán para escalarla.

```
std::string image_path = argv[1];
std::string image_out_path = argv[2];
n_threads = atoi(argv[3]);
```

Posteriormente se inicializan las variables que calcularán el tiempo que tarda nuestro algoritmo en escalar la imagen (tval before, tval after y tval result), se inicializa la variable que nos controlará los hilos y se convierte la imagen de entrada usando openCV, en la matriz que manipularemos con el algoritmo de interpolación bilineal

```
printf("threads ,_Time\n");
struct timeval tval_before, tval_after, tval_result;
pthread_t threads[n_threads];
int *retval, id[n_threads];

Mat img = imread(image_path, IMREAD_COLOR);
if(img.empty())
{
    std::cout << "Could_not_read_the_image:_" << image_path << std::endl;
    return 1;
}
//printf("Mat type: %i", img.type());
resized = (uint8_t *) malloc(img.channels()*height*width*sizeof(uint8_t));
gettimeofday(&tval_before, NULL);
```

Luego, creamos un ciclo por cada uno de los hilos, dentro del ciclo invocamos a la función donde tenemos definido el algoritmo de escalamiento, a esta función le pasamos la estructura Data, que contiene la imagen y el id del hilo que la va a correr.

```
for (int i = 0; i < n_threads; i++){
        //printf("Create Thread %d\n", i);
        id[i] = i;

        struct args *Data = (struct args *)malloc(sizeof(struct args));
        Data->img = &img;
        Data->id = i;
        pthread_create(&threads[i], NULL, bilinear_resize, (void *) Data);
}

for (int i = 0; i < n_threads; i++)
        pthread_join(threads[i], (void **)&retval);
```

Este es el código de la función que reescala la imagen, como hemos mencionado, es una 'traducción' del algoritmo presentado en el marco teórico a nuestra situación específica. Lo primero que hacemos es extraer de la estructura que recibimos como input el id del hilo y la imagen, para posteriormente dividir la altura de la imagen entre el número de hilos, y obtener la fila donde iniciará este hilo a procesar la imagen. Se obtiene la razón entre la resolución objetivo y la fuente para realizar la interpolación basada en esta razón. y se ejecuta dos ciclos for anidados, el primero recorriendo la fila desde la fila donde debe iniciar hasta que acabe su banda, el segundor recorre la columna pixel a pixel, dentro de este se extraen los valores para pesar el valor del pixel x_l y_l x_h y y_h i finalmente en un ultimo for se realiza la interpolación de ese pixel a la nueva ubicación con la formula expuesta en el marco teórico.

```
void * bilinear_resize(void * input ){ //Mat img, uint8_t * resized){

int id = ((struct args*)input)->id;
Mat * img = ((struct args*)input)->img;

int divi = height/n_threads;
int row = (id * divi);

uint8_t * pixelPtr = (uint8_t*)img->data;
int cn = img->channels();

float x_ratio = (img->cols - 1)/(width - 1) ;
float y_ratio = (img->rows - 1)/(height - 1);

uint8_t a, b, c, d, pixel;

for(int i = row; i < min(row+divi, height); i++){
    for(int j = 0; j < width;  j++){

        int x_l = floor(x_ratio * j), y_l = floor(y_ratio * i);
        int x_h = ceil(x_ratio * j), y_h = ceil(y_ratio * i);

        float x_weight = (x_ratio * j) - x_l;
        float y_weight = (y_ratio * i) - y_l;

        for(int k = 0; k < cn; k++){
            a = pixelPtr[y_l*img->cols*cn + x_l*cn + k];
            b = pixelPtr[y_l*img->cols*cn + x_h*cn + k];
            c = pixelPtr[y_h*img->cols*cn + x_l*cn + k];
            d = pixelPtr[y_h*img->cols*cn + x_h*cn + k]
            pixel =( a&0xff )* (1 - x_weight) * (1 - y_weight)
            + (b&0xff) * x_weight * (1 - y_weight) +
(c&0xff) * y_weight * (1 - x_weight) +  (d&0xff ) * x_weight * y_weight;
            resized[i*width*cn + j*cn + k ] = pixel;
        }
    }
}
}
```

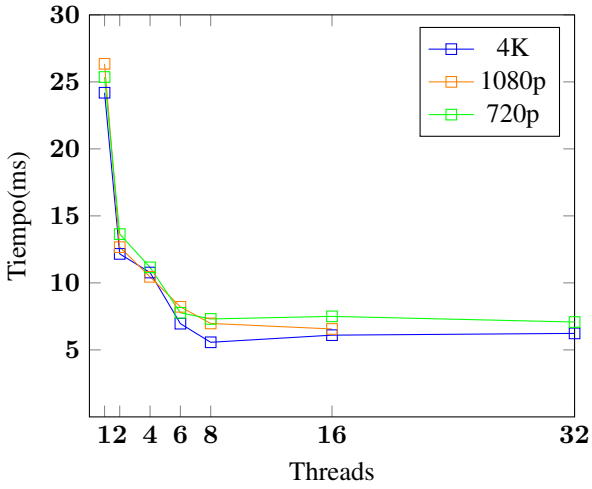
Finalmente volviendo al main de nuestro código, restamos el tiempo de inicio al tiempo de fin y obtenemos cuanto tiempo tardó en escalar la imagen, lo imprimimos y usando de nuevo la librería opencv exportamos la matriz de la imagen escalada y la exportamos al formato que podemos observar .jpg

```
 // Time calculation
gettimeofday(&tval_after, NULL);
timersub(&tval_after, &tval_before, &tval_result);

printf("%i,%ld.%06ld\n",n_threads,(long int) tval_result.tv_sec,
(long int)tval_result.tv_usec);
// Matrix convertion to Mat
Mat resized_img(height, width, CV_8UC(3), resized);
imshow("Display_window", resized_img);
int k = waitKey(0); // Wait for a keystroke in the window

if(k == 's')
{
    imwrite(image_out_path, resized_img);
}
return 0;
```

IV. Resultados

El algoritmo se ejecuto para imágenes en tres resoluciones diferentes(4k, 1080p, 720p), las cuales fueron reducidas a 480p, este proceso se realizo para diferentes números de hilos, para cada ejecución del programa, se calcula el tiempo que duro todo el proceso.

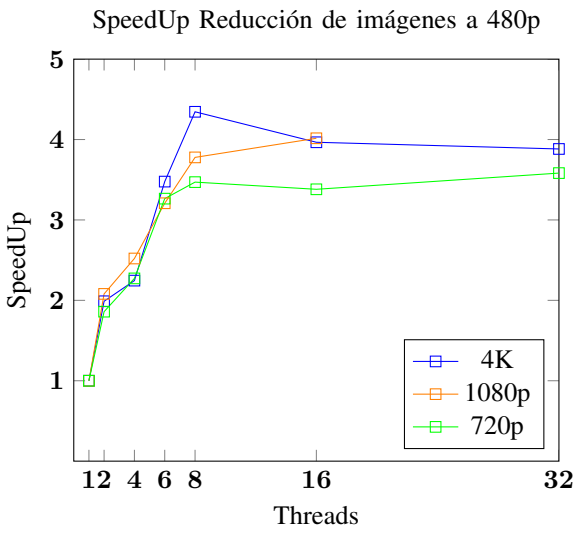
Reducción de imágenes a 480p



Con estos resultados de tiempo se calculo el SpeedUp(SU) de ejecución para los diferentes hilos, haciendo uso de la Ley de Amdahl.

$$SU = \frac{T_s}{T_p} \tag{6}$$

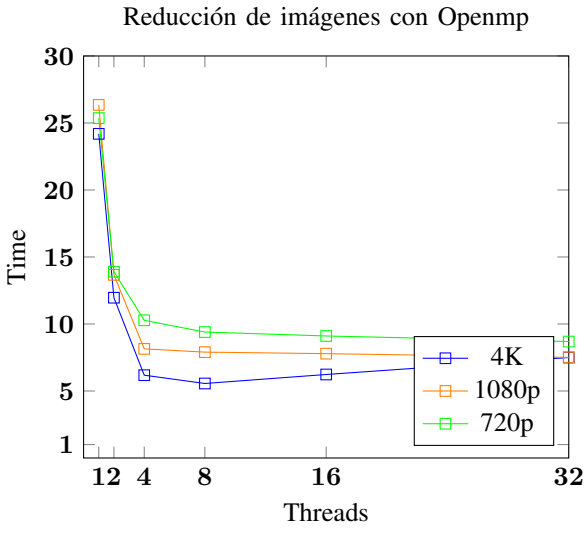
Donde T_s es el tiempo de ejecución para el algoritmo sin paralelizar, y T_p es el tiempo de ejecución para el algoritmo paralelizado.

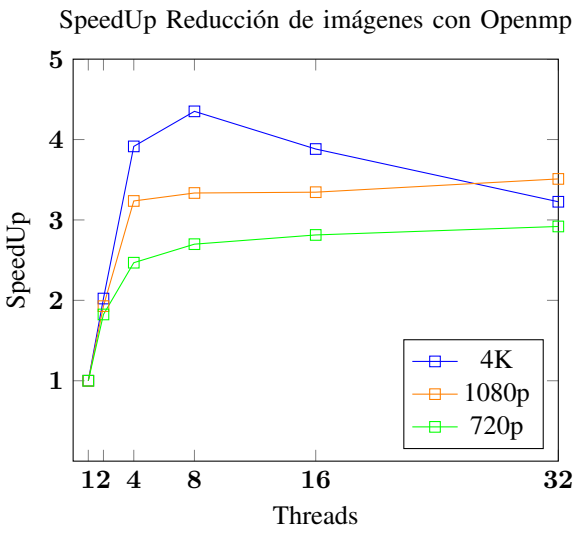


Se observa que para todos los tamaños de imagen el algoritmo de reducción de resolución haciendo uso de la paralelización es más eficiente. En el caso de las imágenes con una resolución de 4K observamos que el SpeedUp mejora a medida que se aumentan los hilos, llegando al valor máximo de SU en 8 hilos.

Para imágenes con resoluciones de 1080p y 720p observamos que el SU va en aumento hasta llegar al tope de los hilos utilizados para las pruebas, 16 hilos para 1080p y 32 hilos para 720p. En las imágenes con resolución de 720p se puede apreciar que hay una disminución en el valor del SU entre 8 y 16 hilos, sin embargo, vuelve a subir al llegar a los 32 hilos.

Para finalizar se realizo el proceso de paralelización del algoritmo haciendo uso de la librería Openmp, los resultados obtenidos fueron los siguientes





Haciendo uso de openmp los resultados varían respecto a la anterior paralelización realizada, en este caso se observa que el algoritmo de paralelización llega a su punto más eficiente con 4 hilos para todos los casos, observamos que si bien el tiempo disminuye a medida que se aumenta el número de hilos, para el SU la diferencia es casi inexistente y no se considera que sea necesario realizar el proceso de reducción de imágenes haciendo uso de openmp con más de 4 hilos, además, observamos un caso particular para la reducción de imágenes 4k, ya que una vez se supera la barrera de los 8 hilos la eficiencia del algoritmo comienza a decaer, esto se puede atribuir a las limitaciones del sistema en el que se realizaron las pruebas.

Figura 3. Imagen reducida de 4K a 480p.

Figura 4. Imagen reducida de 1080p a 480p.

Figura 5. Imagen reducida de 720p a 480p.

V. Conclusion

Utilizando el método de interpolación bilineal logramos un algoritmo para la reducción de imagenes de cualquier tamaño a un tamaño pre establecido, en este caso a 480p. En la tarea de paralelización del algoritmo nos damos cuenta que es un algoritmo altamente paralelizable, y al analizar los resultados vemos una reducción de tiempo significativa al aumentar el número de hilos a ser ejecutados, así como una ganancia superior a 4 en el SpeedUp, teniendo el SpeedUp más alto (en promedio) para 8 hilos. Finalmente, observamos que la complejidad computacional de este algoritmo viene establecida por el tamaño de la imagen de salida y no por el tamaño de la imagen de entrada, se puede observar que los resultados para los diferentes tamaños de imágenes no varían significativamente.

Observamos que los tiempos de ejecución con openmp son similares a los obtenidos con pthread pero con una mayor dispersión, esto se observa fácilmente en la gráfica de SU, donde para la paralelización con pthread tenemos datos más agrupados de reducción por hilo, mientras que cuando se hace uso de openmp, las curvas se distancian más, siendo menos efectivo el algoritmo entre menos resolución tiene la imagen a procesar y llegando al punto de eficiencia más alto en 4 hilos.

Referencias

[1] [1] C. Ji, Understanding Bilinear Image Resizing (Jul 19, 2018). SuperComputer's Blog [Online]. Available: https://chao-ji.github.io/jekyll/update/2018/07/19/BilinearResize.html