

**Sebastián Orrego Marín – 1941144**

## **Propiedades del hardware**

### **CPU**

Architecture: x86\_64

CPU op-mode(s): 32-bit, 64-bit

CPU(s): 12

Model name: AMD Ryzen 5 5600G with Radeon Graphics

Thread(s) per core: 2

Core(s) per socket: 6

### **RAM**

40.0 GB

## **TIEMPOS DE EJECUCIÓN**

Versión secuencial:

Intento 1 = ~~9.531~~

Intento 2 = ~~10.552~~

Intento 3 = 10.116

Intento 4 = 10.277

Intento 5 = 9.867

Promedio = 10.086

Versión paralela (Hilos = 12):

Intento 1 = 8.817

Intento 2 = ~~9.728~~

Intento 3 = 9.464

Intento 4 = 9.572

Intento 5 = ~~8.330~~

Promedio = 9.284

Versión paralela (Hilos = 24):

Intento 1 = 9.261

Intento 2 = ~~9.203~~

Intento 3 = 9.483

Intento 4 = 9.541

Intento 5 = ~~9.618~~

Promedio = 9.428

## **VALORES DE SPEEDUP**

Para el cálculo de los valores de speedup se hace uso de la siguiente formula:

$$Speedup = \frac{Tiempo\ secuencial}{Tiempo\ paralelo}$$

Los valores obtenidos fueron los siguientes:

	Speedup Paralela	Speedup Paralela (x2)
Local	1.086	1.069

### VALORES DE EFICIENCIA

Para el cálculo de los valores de eficiencia se hace uso de la siguiente formula:

$$Eficiencia = \frac{Speedup}{Número\ de\ hilos}$$

Los valores obtenidos fueron los siguientes:

	Eficiencia Paralela	Eficiencia Paralela (x2)
Local	9.05%	8.9%

### COMENTARIOS

Aunque se utiliza OpenMP para paralelizar las funciones recomendadas para aplicar el filtro a cada imagen, los valores de speedup y eficiencia muestran que no se logró una mejora significativa en los tiempos de ejecución del programa. Para optimizar realmente el tiempo de ejecución, sería más efectivo procesar varias imágenes en paralelo en lugar de hacerlo de manera secuencial, como se implementa actualmente. Esto se puede lograr modificando el script all.sh para que invoque en paralelo el mismo número de imágenes que la cantidad de hilos disponibles en la máquina. De esta forma, cuando un hilo se libere, podrá continuar con el procesamiento de las imágenes restantes.

En la función aplicarFiltro, se utiliza `#pragma omp parallel for collapse(2)` para paralelizar el bucle que recorre la imagen. La opción `parallel for` distribuye la ejecución de las iteraciones del bucle entre varios hilos, permitiendo que los cálculos sobre diferentes regiones de la imagen se realicen simultáneamente. El uso de `collapse(2)` es clave aquí, ya que este colapsa los dos bucles anidados (los que recorren las coordenadas x y y de la imagen), permitiendo a OpenMP tratar ambos bucles como uno solo de mayor tamaño. Esto es útil para equilibrar mejor la carga de trabajo entre los hilos, ya que evita que un hilo trabaje solo en filas o columnas, asignando en su lugar bloques de píxeles a cada hilo, lo que reduce los desequilibrios de carga que podrían ocurrir si algunas partes de la imagen son más complejas de procesar que otras.

En la función `calcularSumaPixeles`, se utiliza `#pragma omp parallel for reduction(+:suma)` para paralelizar la suma de los píxeles de la imagen procesada. La directiva `parallel for` nuevamente distribuye las iteraciones del bucle entre varios hilos, pero en este caso, se añade la cláusula `reduction(+:suma)`. Esta cláusula asegura que la variable `suma`, que acumula los valores de los píxeles, se maneje de forma segura cuando es compartida por varios hilos. Sin `reduction`, los hilos podrían sobrescribir los resultados de otros, generando errores de concurrencia. Con `reduction`, cada hilo mantiene una copia local de `suma`, y al finalizar la ejecución paralela, todas las copias se combinan de manera segura en una única suma global.

Finalmente, he de mencionar que ambas declaraciones hacen uso de la instrucción `num_threads(omp_get_max_threads())` para utilizar el máximo número de hilos posibles que el hardware permite.