Programación Orientada al Rendimiento

Grupo 84

Equipo 6

Carlos Peña Martin (100405802)
Alvaro Morata Hontanaya (100405846)
Miguel Castuera Garcia (100451285)
Javier Moreno Yebenes (100428998)

Índice

Diseño Original 3

Optimización 4

Pruebas Realizadas 4

Evaluación de Rendimiento y Energía 6

Organización del Trabajo 10

Conclusiones 11

Diseño Original

La estructura se basa en las 5 bibliotecas descritas en el enunciado:

· **common**.

Es común para las dos versiones del programa, está compuesta por *progarps.cpp* cuya función es comprobar si los parámetros introducidos son correctos y en el caso de que no lo sean señalaría el error.
También está compuesta por *gethead.cpp* que almacena la cabecera del formato bpp de cada imagen además de señalizar posibles errores en los campos recibidos.
Estos programas hacen uso de dos headers (*progarps.hpp* y *gethead.hpp*) para declarar estas funciones y que el main las pueda usar.

· **aos y soa.**

Ambas bibliotecas están formadas por un archivo CPP que contiene funciones necesarias para la obtención de los datos de las imágenes y guardarlos en su respectiva estructura, con una representación de tipo array of structures (biblioteca *aos*) y de tipo structure of arrays (biblioteca *soa*), y un archivo HPP para declarar las funciones y que el main pueda llamarlas.
Estas funciones permiten almacenar los datos rgb de los píxeles de la imagen en su tipo de representación.
También contiene las funciones para realizar las 3 operaciones en las que la imagen requiere de variaciones(mono, histo y gauss), así como la que escribe los nuevos datos generados en el directorio de salida.

· **image-aos y image-soa .**

Contienen el archivo con la función main, cada una trabajando con su propia representación de imagen.
Esta es la que recibe los parámetros, crea dos estructuras de datos, data y newdata para almacenar los bytes de colores.
A su vez, llama a las funciones del resto de bibliotecas y las utiliza con el fin de realizar la respectiva operación a todas las imágenes del directorio de entrada.
También calcula los tiempos que estas operaciones toman y llama al método en la otra biblioteca para la posterior escritura de las nuevas imágenes en el directorio de salida.

Optimización

En cuanto a la optimización del codigo, no se hicieron muchos cambios. Esto se debe a que ya que desde un principio se programo con la intención de que fuese lo más optimo posible. es decir, antes de hacer las pruebas ya se tomaron medidas como las vistas en clase (fusion de arrays, fusion de bucles principalmente). Además, los resultados que obtuvimos fueron los deseados. Por esto nuestron codigo dejaba poco margen de mejora.

Pruebas Realizadas

Las pruebas se realizaron en las máquinas en las que se programó el proyecto, en este caso máquinas con Windows 10 con WSL2. Además, también se tiene en cuenta la correcta ejecución de los programas en los nodos de ejecución de *Avignon*. Los casos concretos que se han probado han sido:

Caso de prueba	Resultado esperado y obtenido
`./image-aos.o ./input_dir ./output_dir`	Error debido a que el número esperado de argumentos no es el adecuado.
`./image-aos.o ./nonexistent ./output_dir copy`	Error debido a que el fichero de entrada no existe o no se localiza.
`./image-aos.o ./input_dir ./nonexistent copy`	Error debido a que el directorio de salida no existe o no se localiza.
`./image-aos.o ./input_dir ./output_dir cop`	Error debido a que la operación especificada no es igual a “copy”, “gauss”, “mono” o “histo”.
`./image-aos.o ./input_dir ./output_dir copy`	Correcta ejecución del programa, copiando las imágenes de *input_dir* a *output_dir*, sin alterar los datos de los bitmaps.
`./image-aos.o ./input_dir ./output_dir mono`	Correcta ejecución del programa, aplicando la escala de grises a todas las imágenes de *input_dir* y almacenando los resultados en el *output_dir*.
`./image-aos.o ./input_dir ./output_dir mono`	Correcta ejecución del programa y la operación, aplicando la difusión gaussiana a las imágenes del *input_dir* y almacenando los resultados en el *output_dir*.
`./image-aos.o ./input_dir ./output_dir histo`	Correcta ejecución del programa y la operación *histo*, devolviendo en el *output_dir* los archivos “.hst” que contienen los histogramas de las imágenes del

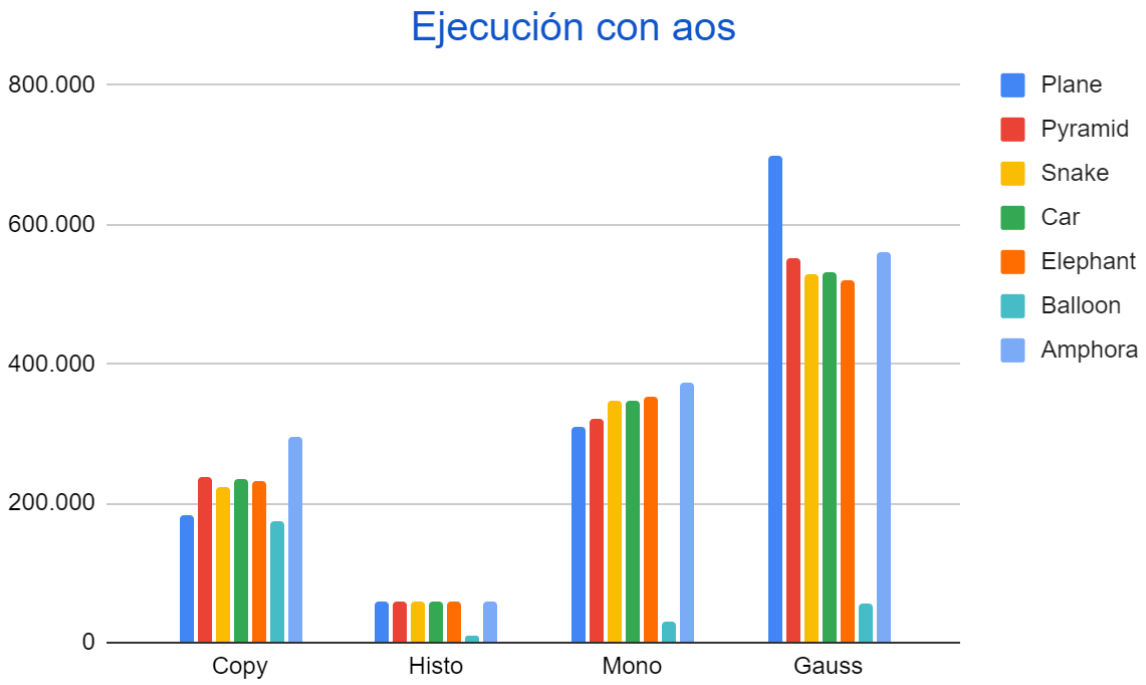
	input_dir, tal y como se pedía en el enunciado.

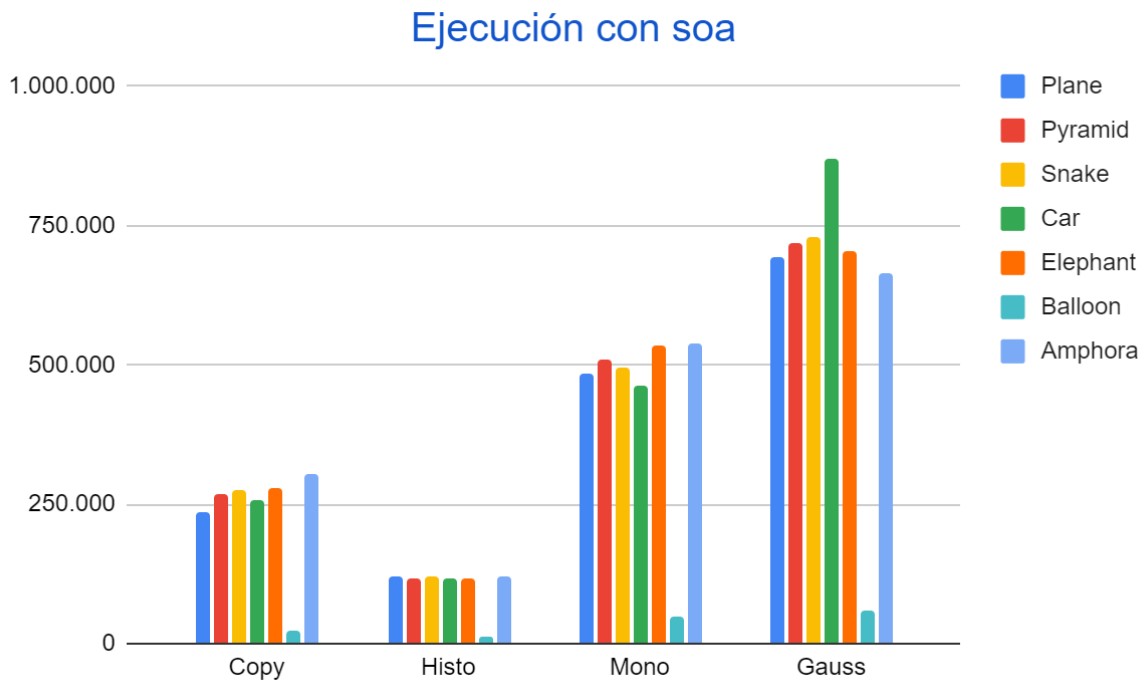
Todos estos archivos han sido comparados a los presentados en Aula Global para comprobar que su ejecución realizaba lo deseado.

Cabe destacar que se han realizado casos de prueba para ambos programas y para ambos los resultados han sido los esperados, aunque en la tabla anterior solo se han expuesto los casos del programa basado en AOS para ahorrar espacio en la memoria.

Evaluación de Rendimiento y Energía

A continuación se muestra la gráfica con los resultados de los tiempos totales de la ejecución del programa:





En los gráficos de barras anteriores podemos ver las ejecuciones de las distintas operaciones aplicadas a todas las imágenes tanto del AOS como del SOA. A simple vista se puede observar como todas las ejecuciones del AOS son más eficientes que en el caso del SOA. Este es un claro ejemplo de que, como vimos en clase, las estructuras de arrays mejoran el rendimiento respecto de las arrays de estructuras. Aun así, las operaciones siguen la misma tendencia en ambas.

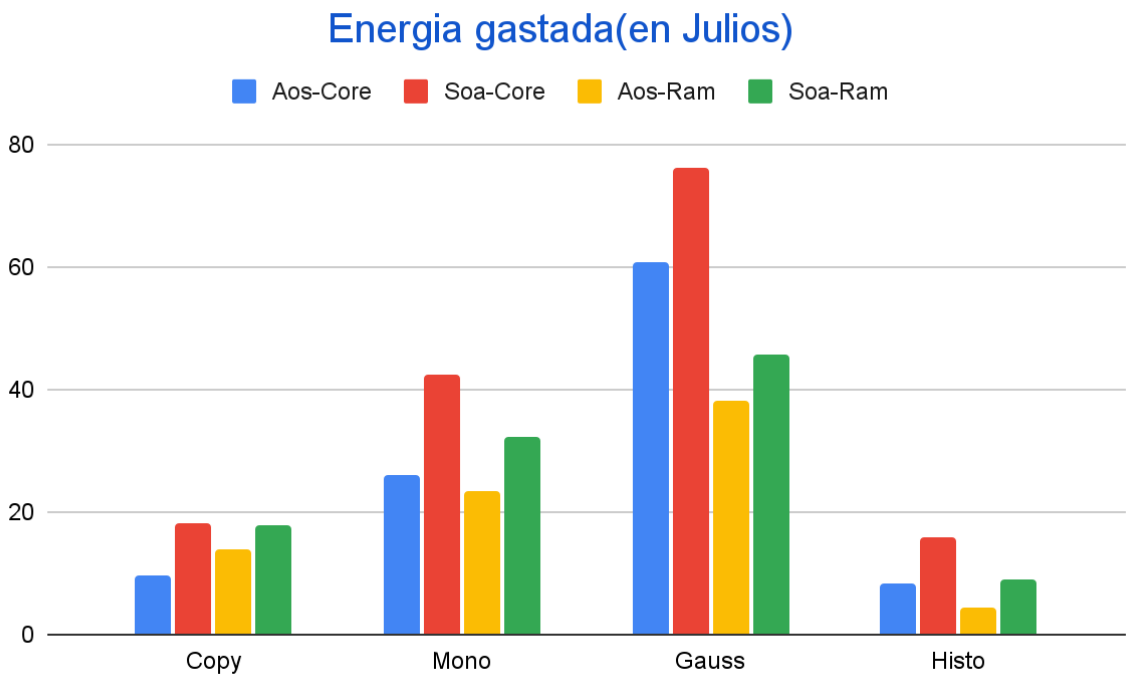
En el caso de la operación “histo” es donde encontramos los tiempos más bajos. Esto tiene sentido ya que en esta función sólo estamos extrayendo la información de los colores de cada imagen y no estamos aplicando nada encima de la misma.

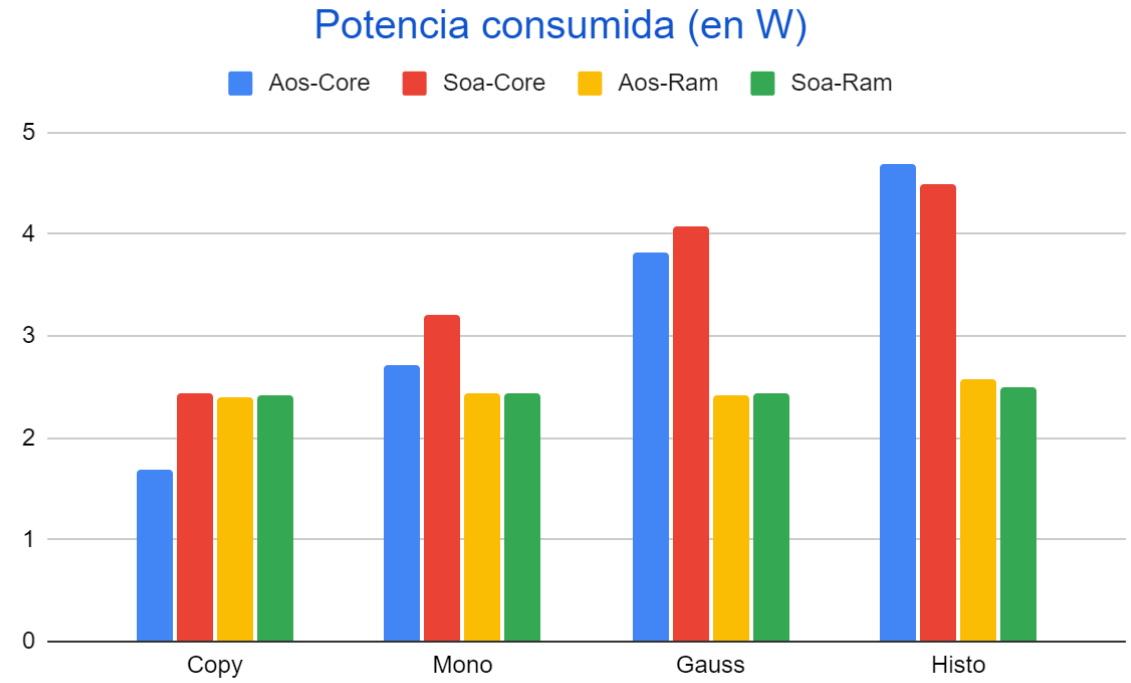
A esta función le sucede “Copy”. Esto se debe a que a parte a parte de leer todos los píxeles de la imagen, también tienes que escribirlos tal cual los leíste en un directorio a parte, lo que aumenta el tiempo de ejecución.

Por último, las que más tardan en ejecutarse son las operaciones de “Mono” y “Gauss” que son las que aplican un filtro a la imagen. La única diferencia entre ellas es el algoritmo matemático que se usa para cambiar los colores azul, verde y rojo de cada pixel. Gauss, al tener un algoritmo compuesto por dos sumatorios, es más complejo que el algoritmo de Mono. Es por esto que este último requiere de un menor tiempo de ejecución.

También, mencionar que cuanto más pequeña sea la imagen, menos tiempo tardará el programa en ejecutarse. Esto se puede ver claramente comparando los resultados de la imagen del globo (que es la más pequeña), con cualquiera de las otras que son más grandes.

A continuación se muestra la gráfica con los resultados de la energía consumida durante la ejecución del programa:





En los gráficos anteriores podemos ver la energía en Julios y la potencia en Vatios tanto del Core y la RAM del AOS como del SOA durante la ejecución de todas las funciones. Ambos gráficos varían entre ellos de la siguiente manera:

En cuanto a la energía, “histo” es la función que menos requiere seguida muy cerca por la función “Copy”, luego la operación “Mono” y finalmente “Gauss”. Esto se debe mayormente al tiempo de ejecución de cada una, a mayor tiempo, mayor energía es necesaria. En cuanto a los cores y las RAM, el que más energía gasta siempre es el del SOA. Esto nos deja claro nuevamente que el AOS es una mejor opción en la mayoría de casos.

En cuanto a la potencia consumida los resultados son muy parejos en todas las funciones con la única novedad de que esta vez la operación que más potencia requiere es la de “Histo”. En cuanto a los Cores, vuelve a ser más eficiente el del AOS en todas las funciones menos en la de histo. Por otra parte, la potencia consumida por las RAM es prácticamente la misma en todos los casos.

Organización del Trabajo

1. **Diseño del código**
   Common:
   Progargs - Miguel Castuera 6,5h
   gethead - Carlos Peña 7h
   Funciones
   getData - Álvaro Morata 8,5h
   writeFile - Javier Moreno 8h
   Operaciones SOA y AOS:
   Gauss -Carlos Peña 5,5h
   Mono - Miguel Castuera 5,5h
   Histo - Javier Moreno 7h
   Main image-aos y image-soa:
   Iteración de directorio - Álvaro Morata 8h
   Aplicación de las operaciones - Carlos Peña 7h
   Cálculo de tiempos - Miguel Castuera 5,5h

2. **Memoria**
   Diseño - Carlos Peña 3,5h
   Optimización -Miguel Castuera 1,5h
   Evaluación de rendimiento y energía - Álvaro Morata 5h
   Conclusiones - Javier Moreno 1,5h

Conclusiones

Finalmente, para concluir la memoria vamos a hablar sobre los resultados obtenidos y mencionar algunos comentarios acerca de nuestra experiencia con la práctica.

Como se ha visto previamente en la sección de evaluación y rendimiento de energía, lo más importante que hemos sacado en claro es la superioridad en todos los niveles a la hora de usar una estructura de arrays frente a un array de estructuras. No solo gracias a la teoría vista en clase, sino ahora gracias a datos reales sacados mediante pruebas de nuestro programa.

En cuanto a nuestra experiencia con esta práctica, al principio fue un poco caótica. Al principio nos llevo mucho tiempo comprender la imagen general de la practica asi como organizar los documentos de una forma logica. Además, ninguno de los integrantes del grupo dominaba el lenguaje c++ e irlo aprendiendo a medida que hacíamos la práctica ha sido bastante desafiante.

Por ultimo el mayor problema que hemos tenido ha sido con el CMake, el cual al final hemos tenido que descartar. En su lugar se ha reemplazado con un build.sh para su correcta compilación.

Resumiendo, la idea general de la práctica nos parece interesante asi como ver los resultados de rendimiento y energía y aqunue ha sido complicada de ejecutar para nosotros, nos ha enseñado tambien a manejarnos en el lenguaje de c++