Organización del computador II

Segundo Cuatrimestre de 2017

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico 2

ASM y filtros de imagenes

Integrantes	LU	Correo electrónico
Imperiale, Luca	436/15	luca.imperiale95@gmail.com
Naselli, Matías	82/15	matiasnaselli@gmail.com

Reservado para la cátedra

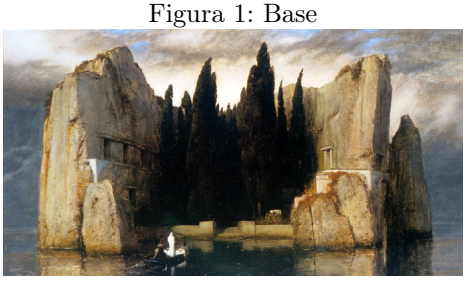
Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Objetivos generales

Este trabajo practico consiste en la implementación de filtros para imágenes utilizando C y ASM, con énfasis en el uso del modelo de procesamiento SIMD, asi como tambien realizar un analisis experimental de los rendimientos obtenidos. A continuacion un breve resumen del funcionamiento de cada filtro implementado. Para los ejemplos usaremos la siguiente imagen.

Figura 1: Base



1.1. Blit

Este filtro recibe 2 imágenes, la imagen base y la imagen blit. Esta ultima sera superpuesta a la imagen base, por cuestiones de diseño la imagen blit quedara siempre del lado superior derecho. Ademas el filtro permite agregar solo una parte deseada de la imagen blit. Aqui vemos una imagen de Peron que utilizaremos como blit, pero de esta imagen solo queremos a Peron y no el fondo violeta, por lo tanto al ser copiado la parte violeta desaparecera.

Figura 2: Blit

A continuacion hay un ejemplo de la imagen base con el filtro aplicado utilizando la imagen de Peron como blit.

Figura 3: Final-blit

1.2. Monocromatizar_inf

Este filtro consiste en transformar la imagen base a una escala de grises, para lograr este efecto existen varias normas o métodos, nosotros para este TP utilizaremos la conocida como norma infinito. Dicho norma sera explicada en detalle dentro de la sección de filtro pero el efecto se logra igualando los valores RGB del píxel al mayor de ellos. A continuación hay un ejemplo de la imagen base con el filtro aplicado.

Figura 4: Final - Monocromatizar_inf

1.3. Ondas

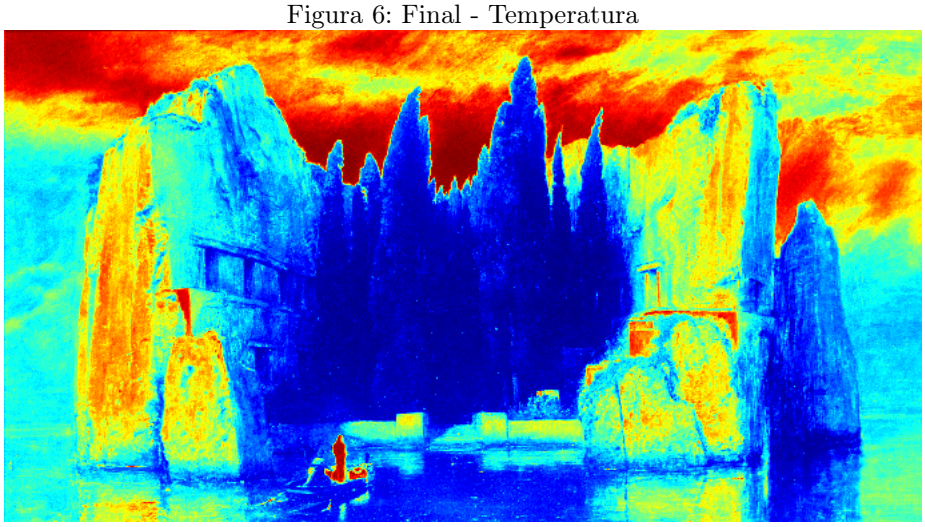
Este filtro genera un efecto de "*ondas*" sobre la imagen base, utilizando diferencias en brillo entre los píxeles dentro y fuera de las ondas. Esto se logra evaluando que tanto "brillo"tiene que tener un píxel dependiendo de su posición relativa al epicentro de las ondas. A continuación un ejemplo de este filtro aplicado a la imagen base.

Figura 5: Final - Ondas

1.4. Temperatura

Este filtro tiene como objetivo agrupar colores similares en pocos grupos, generando así un mapa de calor de la imagen. En nuestro caso vamos a tener 4 de estos grupos y aunque existen pequeñas variaciones entre píxeles dentro del mismo grupo es fácil diferenciar los grupos entre si. A continuación hay un ejemplo de la imagen base con el filtro aplicado.

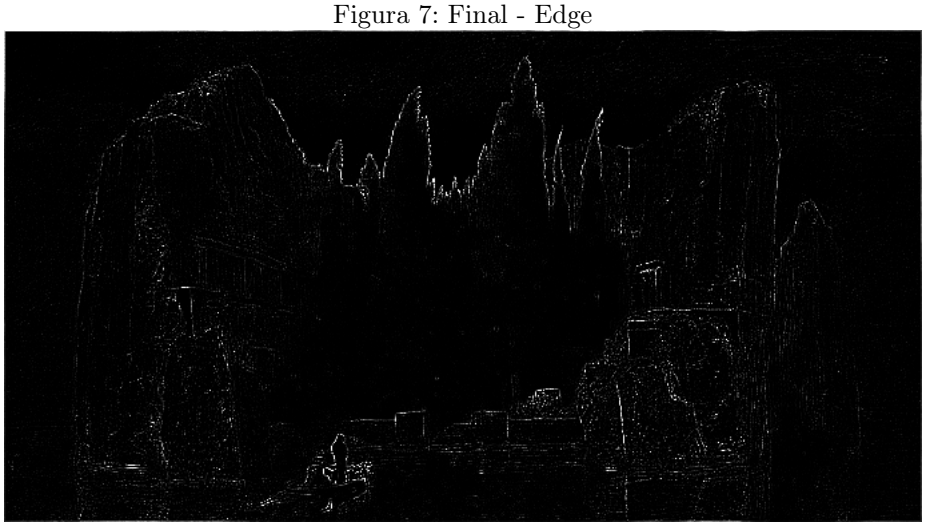
Figura 6: Final - Temperatura



1.5. Edge

Este filtro utilizara como entrada una versión monocromática de la entrada, y su función es resaltar cambios bruscos de brillo dentro de la imagen. Dejando en la imagen de salida valores mas oscuros cuando no hay grandes cambios y valores mas blancos cuando hay cambios pronunciados. A continuación hay un ejemplo de la imagen base con el filtro aplicado.

Figura 7: Final - Edge



2. Desarrollo

2.1. Blit

El filtro Blit resulto ser uno de los mas sencillos de implementar ya que solo es necesario ejecutar la copia de la imagen blit sobre la imagen base en la posicion adecuada y dentro de la imagen Blit revisar si la imagen es o no del color violeta. Siendo su unica dificultad durante la implementacion en ASM ya que uno debia darse cuenta que la imagen blit a diferencia de la base no es de un tamañano de pixeles multiplo de 4 y por ello existe la necesidad de trabajar por cada linea con un pixel individualmente.

A continuacion se presenta un pseudocodigo donde x e y son los indices utilizados para recorrer la imagen destino y fuente y xb e yb son utilizados para recorrer el blit.

Blit (unsigned char *src,unsigned char *dst,int w,int h,int src_row_size,int dst_row_size,unsigned char *blit,int bw,int bh,int blit_row_size)

1: **while** $x < w \wedge y < h$ **do**
2: **if** $x > w - wb \wedge y > h - hb$ **then**
3: **if** $pixel(blit, xb, yb) != violeta$ **then**
4: $pixel(dst, x, y) = pixel(blit, xb, yb)$
5: **else**
6: $pixel(dst, x, y) = pixel(rst, x, y)$

Blit.C

Como se menciono anteriormente la implementacion de Blit es facilmente realizable utilizando un par de ciclos para ubicar correctamente la imagen blit y un par de contadores para recorrerla.

Blit.asm

Para la implementación en asm se utilizo un método similar, se avanzo el puntero de la imagen fuente y destino hasta la primera fila que contenga el Blit, luego de la misma manera avanzamos hasta el punto donde esta el Blit y utilizando una mascara del color violeta se realizo una comparación con los píxeles correspondientes. De esta manera es posible conseguir un registro que indique que píxeles son o no violetas y con ello y la instruccion Blend pudimos mezclar la imagen base y la de Perón.

2.2. Monocromatizar_inf

Para el filtro de monocromatizar se utilizo la norma infinito, para la cual debemos igualar los valores de RGB del píxel con el mayor de ellos. Este filtro también resulto bastante fácil ya que con una sencilla comparación de los valores de cada RGB de cada píxel es posible resolverlo. Para ASM tal comparación se realizo con la ayuda de las instrucciones shifts que permiten alinear los valores que queremos comparar.

A continuación se presenta un pseudocódigo donde x e y son los índices utilizados para recorrer la imagen destino y fuente maxp es una variable auxiliar para encontrar el mayor valor entre las componentes RGB de un pixel.

Monocromatizar$_inf(unsignedchar*src, unsignedchar*dst, intw, inth, intsrc\_row\_size, intdst\_row\_size)$

1: **while do** $x > w \wedge y < h$
2: $intmaxp = max(pixel(x,y).R, pixel(x,y).G, pixel(x,y).B)$
3: $pixel(x,y).R = maxp$
4: $pixel(x,y).G = maxp$
5: $pixel(x,y).B = maxp$

Monocromatizar_inf.c

La implementación en C fue realizada mediante 2 ciclos para recorrer toda la imagen y así como se menciona anteriormente se utiliza una variable auxiliar para encontrar el mayor valor RGB del píxel y se reemplazan los 3 con dicho valor.

Monocromatizar_inf.asm

Para ASM la comparación de los valores es mas complicada, ya que no es posible compararlos horizontalmente. Este problema fue resuelto guardando repetidamente los píxeles en varios registros y utilizando un shift para alinear los valores que queríamos comparar. Luego creamos una mascara con el mayor valor para cada píxel y utilizando un shuffle llenamos cada píxel con su valor máximo

2.3. Ondas

La idea del filtro de ondas es generar dicho efecto cambiando el brillo de los píxeles (para cambiar el brillo manteniendo el color se le suma o resta a las componentes RGB del píxel un mismo valor). Para la implementación en C se utilizo una función dada por la cátedra llamada ondas, mientras que en ASM tuvimos que implementar dicha función también. El desafió de este filtro fue la conversión de variables de int a float y viceversa para realizar la función ondas, así como la utilización de mascaras para la función auxiliar saturar (la cual hace que si una componente pasa el valor 255 para arriba o 0 para abajo se sature a 255 o 0 correspondientemente)

A continuación el pseudocódigo provisto por la cátedra para la realización de este filtro:

1: **for** pixel ubicado en la posici'on $(\mathbf{x}, \mathbf{y})$ **do**
2: $d_x \leftarrow x - x_0$
3:
4: $d_y \leftarrow y - y_0$
5:
6: $d_{xy} \leftarrow \sqrt{d_x^2 + d_y^2}$
7:
8: $\leftarrow \frac{(d_{xy} - RADIUS)}{WAVELENGTH}$
9:
10: $a \leftarrow \frac{1}{1+(\frac{r}{TRAINWIDTH})^2}$
11:
12: $t \leftarrow (r - floor(r)) \cdot 2 \cdot \pi - \pi$
13:
14: $prof \leftarrow a \cdot (t - \frac{t^3}{6} + \frac{t^5}{120} - \frac{t^7}{5040})$
15:
16: $pixel = prof \cdot 64 + I_{src}(x, y)$
17:
18: $I_{dst}(x, y) = saturar(pixel)$

donde:

- x_0 e y_0 representan la posici'on donde est'a centrada la onda,
- $RADIUS$, $WAVELENGTH$ y $TRAINWIDTH$ son constantes que definen la forma de la onda y
- $saturar(x)$ es una funci'on que retorna 0 si x es menor 0, 255 si es mayor a 255 y x en cualquier otro caso.

2.4. Temperatura

El filtro temperatura reemplaza el color de cada pixel, por otro que depende del promedio de la suma de cada componente de este. Esta conversión depende de la siguiente funcion

$$\mathsf{dst}_{(i,j)} < r, g, b >= \begin{cases} < 0, 0, 128 + t \cdot 4 > & \text{si } t < 32 \\ < 0, (t-32) \cdot 4, 255 > & \text{si } 32 \leq t < 96 \\ < (t-96) \cdot 4, 255, 255 - (t-96) \cdot 4 > & \text{si } 96 \leq t < 160 \\ < 255, 255 - (t-160) \cdot 4, 0 > & \text{si } 160 \leq t < 224 \\ < 255 - (t-224) \cdot 4, 0, 0 > & \text{si no} \end{cases}$$

En C esto se logra simplemente con varias comparaciones consecutivas del valor promedio, que checkean en que caso de la función estamos. Luego, solo se modifican los valores del píxel según la función.

En ASM, tenemos que hacer lo mismo, pero con 4 píxeles a la vez. La solución que encontramos fue aplicar la función independientemente que el píxel entre o no en ese caso. Luego armamos una mascara que indicaba cual de los 4 debía ser reemplazado, y finalmente hicimos un blend del registro cambiado con el original. Esto obviamente para cada caso. El problema de esta implementación es que hacemos las cuentas de que píxel debería ir, independientemente si lo usamos o no, lo cual es tiempo perdido. Se vera en mas detalle como esto afecta el rendimiento en la sección de Experimentación.

2.5. Edge

La obtención de los cambios bruscos de intensidad se hizo utilizando el operador de Lapalce, cuya matriz es:

$$M = \begin{pmatrix} 0{,}5 & 1 & 0{,}5 \\ 1 & -6 & 1 \\ 0{,}5 & 1 & 0{,}5 \end{pmatrix}$$

Para cada píxel, la intensidad estará dada por la siguiente sumatoria

$$dst(x,y) = \sum_{k=0}^{2} \sum_{l=0}^{2} src(x+k-1, y+l-1) * M(k,l)$$

Es decir, cada píxel dependerá de si mismo, y de los 8 píxeles aledaños. Por esta razón, nuestro filtro no tiene forma de determinar la intensidad en los bordes de la imagen, en los cuales simplemente deja la original.

Para la implementación en C, simplemente se ciclo entre los píxeles(obviando los bordes), y efectuando la suma antes descripta. Ese valor se guardo en la imagen destino, antes saturándolo a un byte.

En ASM, hicimos lo mismo pero tomando de a 16 píxeles. Esto porque la imagen de entrada esta en escala de grises(un byte por píxel), entrando 16 en un registro de 128 bits. Es importante aclarar que, como las imágenes de entrada por diseño tienen un ancho múltiplo de 4 píxeles, y acá nosotros estamos tomando de a 16, tuvimos que tener especial cuidado en el final de cada fila. Para solucionar esto, en cada ciclo de fila, al final hay un ciclo que trabaja píxel por píxel en un registro de uso general el resultado.

3. Experimentación

3.1. Introducción

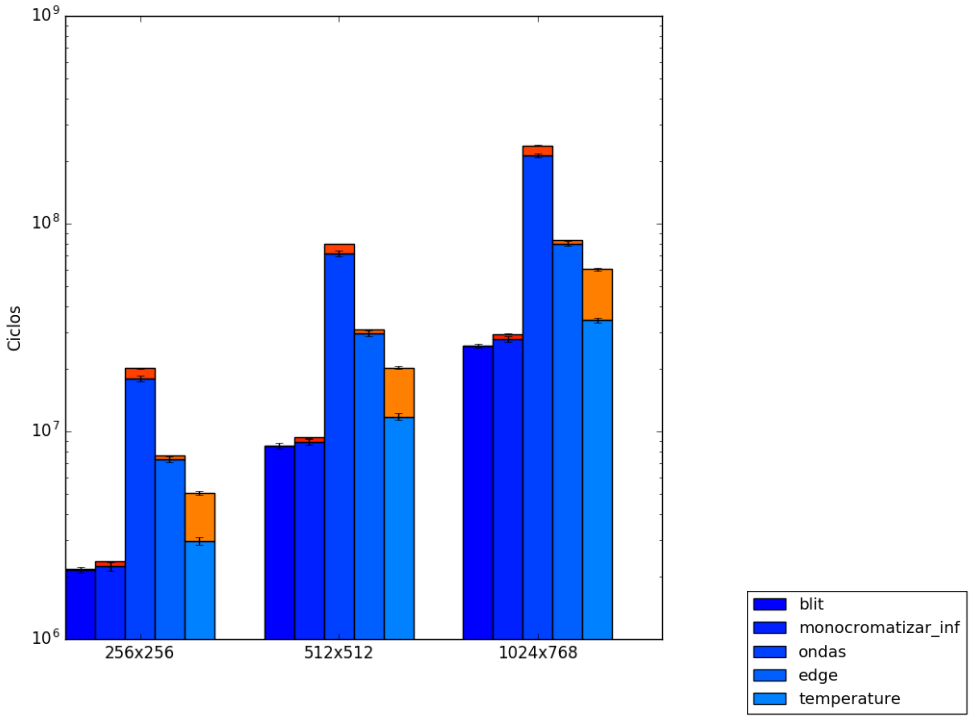
Cada vez que se hable de la cantidad de ciclos que emplea una función en esta sección, este valor fue obtenido del promedio de 100 ejecuciones, así como su desviación standard. Esta ultima solo va a estar representada en los gráficos que acompañen.

Cabe decir que en el momento de realizar las 100 mediciones, primero se ejecutaron 5 para ser luego descartadas. Esto es porque observamos que las primeras siempre daban valores muy por encima de la media de todo el resto. Creemos que esto es porque el procesador, después de algunas ejecuciones del programa, almacena la imagen en la memoria cache, acelerando el proceso de mover de memoria a los registros.

3.2. Comparación C vs ASM

En 3.2 se ve una comparación de cantidad de ciclos que toma la misma imagen en completarse, para cada filtro, y una implementación arriba de la otra. La parte de abajo azul son las funciones en C, y la parte de arriba en ASM. También se puede ver como cambia esta misma relación con el tamaño de la imagen.

Figura 8: C vs ASM



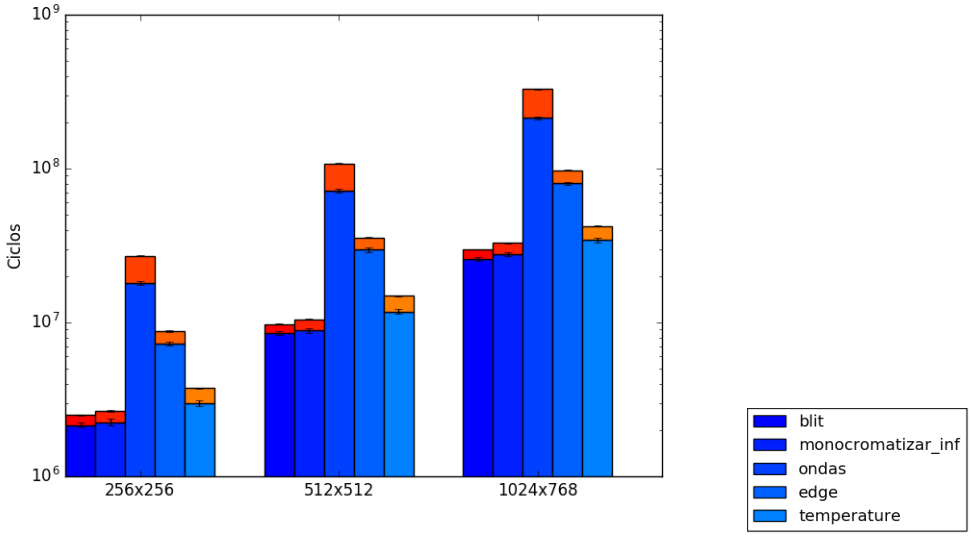
Claramente se ve que la implementación en ASM es mucho mas rápida que su par en C, en casi todos los filtros. La unica excepcion es temperatura, en el cual los tiempos siempre dan superiores en la implementacion de ASM que en la de C. Se explicara estos en la seccion de Experimentos sobre filtros especificos.

3.3. Comparación C vs C optimizado con O3

Ahora lo que vamos a comparar es C compilado sin ningún flag de optimización, con uno compilado con el flag -O3. Este nivel de optimización es el mas alto recomendado. Lo que hara el compilador cuando recibe esta opción es activar optimizaciones que son muy caras en respecto al tiempo de compilación y uso de memoria. Lo que se gana con esto son velocidades mas altas. Cabe notar que este nivel de optimización habilita el uso de los registros XMM e YMM, los cuales nosotros utilizamos en ASM(solo los XMM).

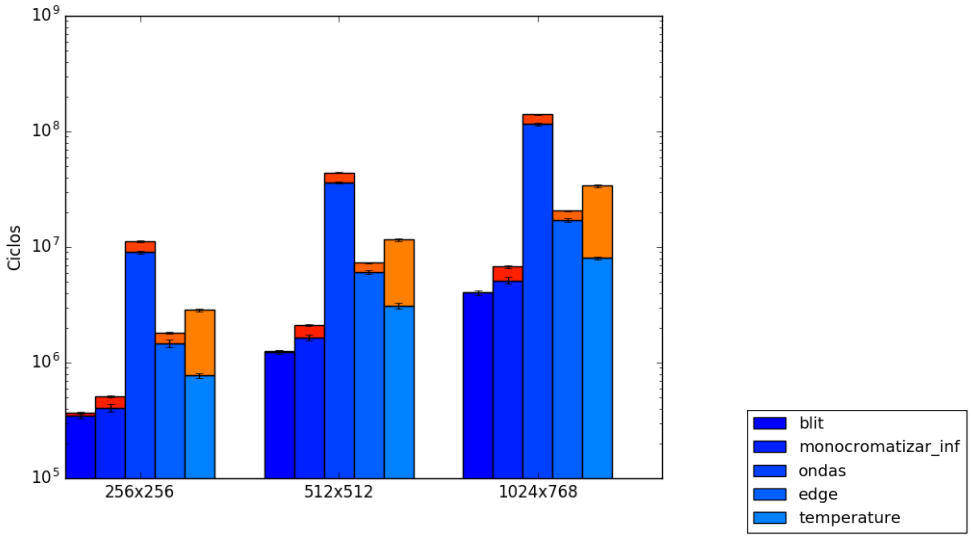
A continuación, en , se mostrara lo que tarda el mismo codigo en ejecutarse compilado de las 2 formas, abajo C sin optimizaciones y arriba con -O3.

Figura 9: C vs C con O3



También es interesante comparar nuestro código ASM con el C optimizado. Esto se muestra en el siguiente gráfico

Figura 10: ASM vs C con O3



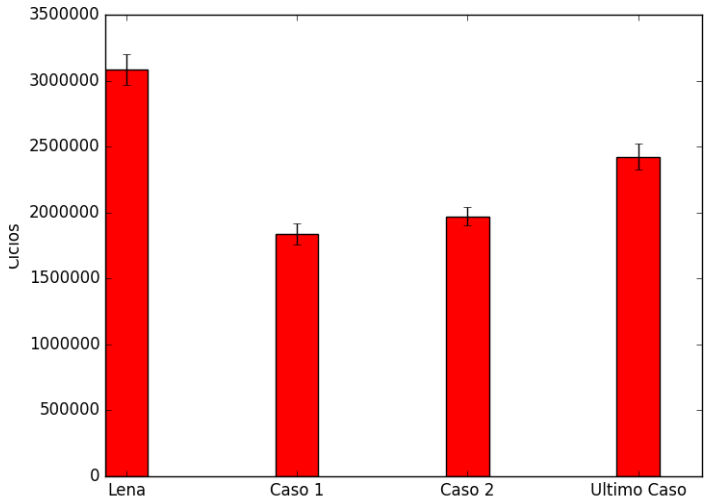
De nuevo, se muestran los ciclos de los programas en C abajo y el ASM arriba.

3.4. Experimentos sobre filtros específicos

3.4.1. Temperatura

Predicciones de C Probaremos el impacto del predictor de condiciones de C. Esto va a ser que si el programa ya entro muchas veces por una rama de un if, tenga mucho mas peso esa rama, y la chequee primero. Para hacerlo, nos haremos de imágenes del mismo color y veremos la diferencia de tiempos en ASM y en C.

Figura 11: Tiempo de ejecución con múltiples imágenes



En el gráfico anterior podemos observar el tiempo que se tarda en aplicar en C el filtro de temperatura a nuestra imagen de base Lena y a varias imágenes monocromáticas. Al tener toda la imagen el mismo color al aplicar el filtro siempre entramos a la misma rama de los if. En el caso de Lena el filtro tiene que revisar todos los casos para cada píxel aumentando considerablemente el tiempo de ejecución, mientras que la imagen que siempre entra al primer caso no debe realizar tantos cálculos y lo que reduce drásticamente el tiempo de ejecución. Se puede observar también que para la imagen que entra en el segundo caso tiene un tiempo parecido al anterior debido a que el predictor de C luego de algunos casos ya solo testea la rama que se utilizo anteriormente en vez de todas. Para la imagen que debe entrar al ultimo caso vemos que aumento la cantidad de ciclos comparado con los anteriores pero aun así sigue teniendo un rendimiento mayor que la imagen de Lena.

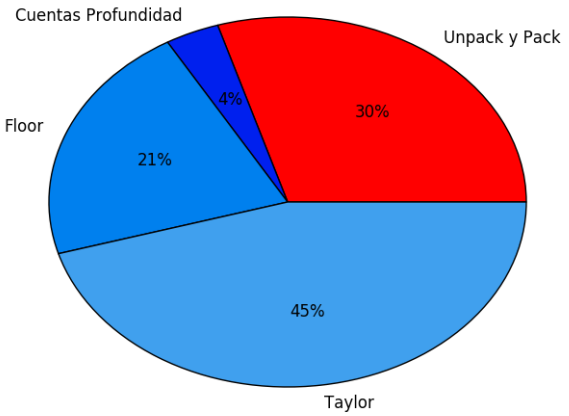
En el caso de la implementación en ASM de este filtro, no vamos a tener las predicciones, entonces el tiempo no va a ser variable, osea, siempre va a chequear con todos los casos. Otro problema de nuestra implementación en ASM, fue que usamos un blend de los píxeles cambiados con los originales, por lo que hacemos todas las cuentas de cada caso para cada píxel. Esto se podría haber evitado usando saltos condicionales y solo realizando los cálculos necesarios para cada tira de 4 píxeles.

3.4.2. Ondas

Un experimento interesante para el filtro de ondas es ver los tiempos de distintas partes del código en la implementación de ASM, ya que se requirieron múltiples operaciones matemáticas, dividimos el código en funciones auxiliares y medimos la cantidad de ciclos para cada uno de ellas. De esta manera podemos ver cual es el cuello de botella del filtro.

A continuación podemos ver en un gráfico de torta las distintas partes necesarias del algoritmo.

Figura 12: Tiempos de las funciones auxiliares utilizadas para ondas



El tiempo de ejecución completo para los 4 píxeles con los cuales trabajamos resulto de 185 ciclos de maquina, y vemos en el gráfico anterior como fueron utilizados, de ese total casi la mitad del tiempo se necesito para el calculo del seno de Taylor lo cual requiere múltiples multiplicaciones y divisiones. Luego un tercio se uso para la lectura y escritura de datos a memoria como podemos ver en la sección roja. Finalmente la operación Floor que trunca un numero entero se llevo un quinto de los ciclos de maquina. Cuentas profundidad fue lo que se uso para cambios y avance de registros que utilizamos como contadores para los ciclos.

4. Conclusiones

Para concluir logramos observar varias cosas interesantes durante la experimentación:

La diferencias de tiempo entre ejecutar los filtros en C y ASM, y en que casos uno resulta mas veloz que el otro, así también como funcionan ambas implementaciones en casos borde, por ejemplo durante la experimentación de temperatura Además de ello pudimos apreciar cuanto del tiempo de experimentación se utiliza para el levantado de datos y la escritura de tales a memoria y cuanto se utiliza para el calculo y aplicación de los filtros. También en el caso de ondas cuanto tiempo es utilizado para distintas operaciones matemáticas, tales como el floor(), el seno de taylor y el calculo total de la profundidad. Durante la experimentación se nos ocurrieron mas cosas de las que tuvimos tiempo de probar y estamos seguros que podríamos haber obtenido mas datos.