

UNIVERSIDAD DE BUENOS AIRES

Facultad de Ciencias Exactas y Naturales

Departamento de Computación

Organización del Computador



TRABAJO PRÁCTICO NÚMERO 2

Nombre de Grupo: Napolitana con Jamón y Morrone

Alumnos:

Izcovich, Sabrina | sizcovich@gmail.com | LU 550/11

López Veluscek, Matías | milopezv@gmail.com | 926/10

Índice

1. Introducción	3
2. Desarrollo	3
2.1. Implementación en C	3
2.2. Implementación en Assembler	4
3. Resultados	15
4. Conclusión	16

1. Introducción

El objetivo de este trabajo práctico fue experimentar utilizando el modelo de programación SIMD. Para ello, fue requerido implementar seis filtros para procesamiento de imágenes (Recortar, Halftone, Umbralizar, Colorizar, Efecto Plasma y Rotar) tanto en *C* como en *Assembler*.

Por otro lado, debimos analizar la performance de un procesador al hacer uso de las operaciones SIMD. Para ello, realizamos comparaciones de velocidad entre los dos tipos de implementaciones realizados utilizando la herramienta Time Stamp Counter (TSC) del procesador y sacamos conclusiones al respecto.

2. Desarrollo

Nuestro trabajo consistió en implementar los filtros mencionados anteriormente. Para el desarrollo del mismo, diversas herramientas fueron necesarias. En esta sección, se explicita la utilización de las mismas aclarando en qué nos ayudaron a lograr una correcta ejecución de nuestra implementación.

2.1. Implementación en C

En el caso de *C*, las implementaciones se realizaron siguiendo algoritmos sencillos. Para recorrer las matrices de píxeles utilizamos *for* pues nos pareció lo más conveniente considerando que nuestros valores estaban pre-establecidos y que al colocar un *for* dentro de otro (uno para las filas y otro para las columnas) era más simple recorrer todas las posiciones de una matriz.

Por un lado utilizamos, como tipos de datos, enteros y doubles. Esto se debió a que era lo más conveniente para las operaciones que debía realizar nuestro programa ya que el nivel de precisión alcanzado era mayor y esa cualidad es muy importante para no perder información sobre cada píxel. Un inconveniente que enfrentamos y que nos resultó llamativo fue al implementar *rotar.c* ya que, al definir $\sqrt{2}/2$ como *double*, la imagen obtenida presentaba diferencias con la imagen que debíamos encontrar. Si bien éstas no resultaban visibles, decidimos resolverlo definiendo $\sqrt{2}/2$ como *float*.

Luego, con el fin de disminuir el tiempo de ejecución de los *C*, comprimimos partes del código reduciendo, a su vez, la cantidad de líneas. También, alteramos pequeños detalles que sumados marcarían diferencia. Por ejemplo, cambiamos *j++* por *++j*.

2.2. Implementación en Assembler

Para la implementación en *Assembler*, utilizamos la extensión SSE para procesar varios bits a la vez. Esta herramienta nos permitió acelerar la ejecución de nuestro programa dado que en cada ciclo (en la mayoría de los casos) se trataban 16 bits a la vez. Por otro lado, preferimos utilizar floats en vez de doubles pues la imagen obtenida resultaba tan lípida como la obtenida por el tipo de datos de mayor precisión. De esta manera, logramos no utilizar registros innecesariamente y evitar largos ciclos que disminuirían el tiempo de ejecución del programa.

Nuestros filtros fueron realizados de la siguiente manera:

- **Recortar:** En este filtro, la idea consistió en recortar las esquinas de la imagen original obteniendo cuatro bloques de tamaño *tam* para luego unirlos y ubicarlos en las posiciones opuestas.

Para realizar dicho filtro, procesamos los 4 cuadrantes de la imagen por separado para no confundir las posiciones de origen y destino de los píxeles por lo que la implementación de cada cuadrante resultó ser la misma a diferencia de los offset utilizados.

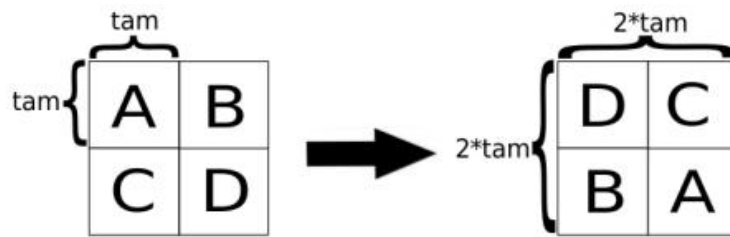
La implementación de este filtro consistió, a grandes rasgos, en levantar de a 16 bits desde la posición de origen y pegarlos en la posición de destino. Una vez alcanzado el valor de *tam*, se procesaba la siguiente fila y así sucesivamente hasta terminar con la columna número *tam*.

Para realizar dicha función no resultó necesario desempaquetar y empaquetar los píxeles pues éstos no debían ser utilizados para cálculos como tampoco modificados. Por lo tanto, a medida que se los levantaba, se los copiaba tal cual a la imagen destino.

La función consistió en una seguidilla de los siguientes pasos:

En primer lugar, se procesó el cuadrante B. Para ello, se le restó el *tam* al ancho de la imagen para obtener el offset. Luego, se calculó el offset vertical de destino multiplicando el *dst_row_size* por *tam* para continuar ingresando a un ciclo que consistió en mover la porción de memoria del fuente al destino hasta copiar la fila entera del cuadrante. Una vez alcanzado un desplazamiento mayor que el tamaño de la fila, se retrocedió la distancia excedida para terminar por procesar el mismo ciclo con la última columna. Para controlar dichos movimientos, fue necesario un contador que nos advirtiera si nos encontrábamos o no en la última fila que sería el *tam*.

La misma manipulación fue realizada para el resto de los cuadrantes de manera tal que la imagen quedara de la siguiente forma:



Luego, comparamos la cantidad de ciclos producidos por *C* y por *Assembler*.

- Comparación de Tiempo

Al probar lena.bmp de 512x512, con una cantidad de 100 iteraciones, el tiempo obtenido fue el siguiente:

Implementación C

Tiempo de ejecución:

Comienzo : 857860615438390

Fin : 857860685974035

iteraciones : 100

de ciclos insumidos totales : 70535645

de ciclos insumidos por llamada : 705356.500

Implementación ASM

Tiempo de ejecución:

Comienzo : 857860758957933

Fin : 857860759960311

iteraciones : 100

de ciclos insumidos totales : 1002378

de ciclos insumidos por llamada : 10023.780

Resultados

Ciclos C: 705356.5

Ciclos ASM: 10023.78

Ciclos ASM respecto de C: 1.42109415593 %

Tiempo C: 70535645

Tiempo ASM: 1002378

Tiempo ASM respecto de C: 1.42109425667 %

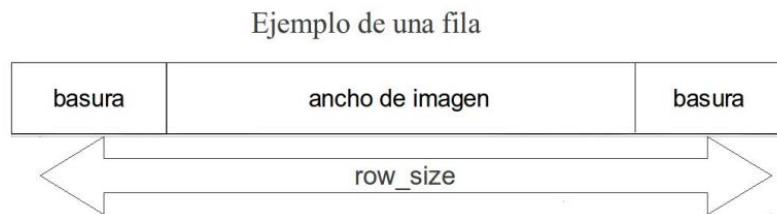
- **Halftone:** La idea principal de este filtro consistió en partir la imagen original en bloques de 2x2 píxeles generando, en la imagen destino, bloques del mismo tamaño. A cada nuevo bloque de la imagen original se le debió asignar el color blanco o negro dependiendo del valor de $t = \sum_{i=0}^3 (p_i$, con p_i píxeles del bloque de la imagen original de acuerdo a ciertas condiciones requeridas.

Para la realización de dicho filtro, debimos almacenar en memoria valores que fueron necesarios durante la implementación. Éstos resultaron ser los valores con los que se compararía la sumatoria de los 4 bits como también un inversor (formado por 1's) y distinguidores entre la primer fila y la segunda (formados por 00ff y ff00 consecutivamente).

Por otro lado, fue necesario que tanto el ancho como el alto de la imagen fueran pares para poder iterar de a dos filas y columnas a la vez. Para ello, dividimos n y m por 2 y comparamos el resto con 0. En caso de serlo, m y n se mantenían. Caso contrario, se le restaba 1 a la variable impar.

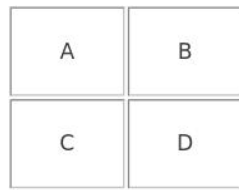
Luego, proseguimos recorriendo las matrices de fuente y destino. Para ello, decidimos desplazarnos a través de los punteros, por lo tanto, mientras estuviéramos en la misma fila, nos encargamos de sumarle 16 a rdi y a rsi en cada iteración. Al final de cada ciclo, el valor alcanzado por rsi y rdi era comparado con el ancho-16 para luego decidir si el ciclo debía continuar normalmente o si se había llegado al final de la fila. Una vez que el valor de rdi y rsi llegaban o superaban el ancho-16, se pasaba al final de la fila, donde rdi y rsi se seteaban en el ancho-16 para terminar procesando los últimos 16 bits. Para tratar la última columna, decidimos copiar el ciclo principal de manera tal a no tener que utilizar registros en exceso para determinar cuándo se debía pasar a la siguiente fila dado que no teníamos registros disponibles.

Luego, dado que procesamos de a dos filas, debíamos desplazar nuestro puntero a $rdi+16+row_size+basura$. Ésto se debe a que, en primer lugar, rdi terminaba la columna apuntando a ancho-16, luego era necesario sumar 16 para llegar al ancho. Por otro lado, se suma la basura para llegar al final de la fila de la imagen, por último, se suma el row_size para saltar la fila siguiente. En el siguiente gráfico puede observarse la manera en la que se conforma una fila:



Para procesar los bits, nos encargamos de desempaquetamos de a dos filas. Luego, sumamos los píxeles dos a dos en cada fila. Finalmente, realizamos la suma de las posiciones de las filas correspondientes con las columnas correspondientes de la siguiente manera:

Considerando la siguiente imagen,

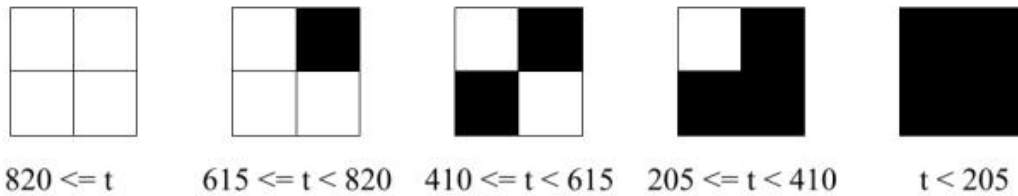


los pasos realizados fueron:

- $A+B$
- $C+D$
- $(A+B) + (C+D)$

Luego, se prosiguió realizando la comparación `pcmpgtw` entre la suma y los valores seteados para la comparación. Dado que el resultado deseado era el inverso (\leq en vez de $>$), se aplicó un `xor` con el inversor a cada resultado obtenido por la comparación. Por último, se realizó un `and` entre lo obtenido de invertir la comparación y los filtrados de primera y segunda fila. Esta suma de operaciones debieron realizarse la cantidad de veces necesaria como para cubrir todos los valores con los que comparar la suma de píxeles. Luego, 4 veces. Por último, las máscaras fueron unidas con un `'or'` en el que se juntaron las columnas respectivas de la primera y segunda fila. Los píxeles resultantes fueron, entonces, ubicados en la matriz destino.

Los píxeles resultantes debían cumplir con la siguiente condición:



- Comparación de Tiempo

Al probar lena.bmp de 512x512, con una cantidad de 100 iteraciones, el tiempo obtenido fue el siguiente:

Implementación C

Tiempo de ejecución:

Comienzo : 259119263552208

Fin : 259119729516615

iteraciones : 100

de ciclos insumidos totales : 465964407

de ciclos insumidos por llamada : 4659644.000

Implementación ASM

Tiempo de ejecución:

Comienzo : 259119877087956

Fin : 259119888325521

iteraciones : 100

de ciclos insumidos totales : 11237565

de ciclos insumidos por llamada : 112375.648

Resultados

Ciclos C: 4659644.0

Ciclos ASM: 112375.648

Ciclos ASM respecto de C: 2.41167883212 %

Tiempo C: 465964407

Tiempo ASM: 11237565

Tiempo ASM respecto de C: 2.41167883881 %

- **Umbralizar:** En Umbralizar, la imagen destino debía cumplir los siguientes

requisitos:

- Cada píxel menor a *min* debía valer 0.
 - Cada píxel mayor a *max* debía valer 255.
 - Caso contrario, el píxel debía valer $\lfloor p/Q \rfloor \cdot Q$
- con *min*, *max* y *Q* enteros de 1 byte.

Para lograr dicho objetivo, pasamos los valores *max*, *min* y *Q* (enteros) a floats para realizar los cálculos necesarios. Para ello, desempaquetamos y convertimos los píxeles a dicho tipo de dato.

Nuestro ciclo consistió en procesar de a 16 bits comparando los píxeles con *max* y *min* con la operación `pcmpgtw`. Las máscaras resultantes fueron, luego, invertidas con `0xff00's` dado que el resultado deseado era la comparación inversa a la facilitada por Intel. Luego, se debieron procesar los píxeles que no fueron menores a *min* ni mayores a *max*. Para ello, se los dividió por *Q*, se los redondeó a parte entera con la función `roundps` y luego se los multiplicó nuevamente por *Q*. Por último, los convertimos a `doubleword` para luego empaquetar y terminar por juntar las máscaras con `and` y `or` y procesar los píxeles. Todo lo explicitado anteriormente fue realizado de manera rápida gracias al procedimiento realizado con SSE.

Para recorrer la matriz, decidimos desplazarnos a través de los punteros, por lo tanto, mientras estuviéramos en la misma fila, nos encargamos de sumarle 16 a `rdi` y a `rsi` en cada iteración. Al final de cada ciclo, el valor alcanzado por `rsi` y `rdi` era comparado con el ancho-16 para luego decidir si el ciclo debía continuar normalmente o si se había llegado al final de la fila. Una vez que el valor de `rdi` y `rsi` llegaban o superaban el ancho-16, se pasaba al final de la fila, donde `rdi` y `rsi` se seteaban en el ancho-16 para terminar procesando los últimos 16 bits. Para tratar la última columna, decidimos copiar el ciclo principal de manera tal a no tener que utilizar registros en exceso para determinar cuándo se debía pasar a la siguiente fila dado que no teníamos registros disponibles.

• Comparación de Tiempo

Al probar `lena.bmp` de 512x512, con una cantidad de 100 iteraciones y 64 128 16 como parámetros, el tiempo obtenido fue el siguiente:

Implementación C

Tiempo de ejecución:

Comienzo : 857572172822622

Fin : 857572676855448

iteraciones : 100

de ciclos insumidos totales : 504032826

de ciclos insumidos por llamada : 5040328.500

Implementación ASM

Tiempo de ejecución:

Comienzo : 857572838473791

Fin : 857572938386751

iteraciones : 100

de ciclos insumidos totales : 99912960

de ciclos insumidos por llamada : 999129.625

Resultados

Ciclos C: 5040328.5

Ciclos ASM: 999129.625

Ciclos ASM respecto de C: 19.8227084802 %

Tiempo C: 504032826

Tiempo ASM: 99912960

Tiempo ASM respecto de C: 19.8227089281 %

- **Colorizar:** Para colorizar, la imagen fuente debía ser procesada de acuerdo a las siguientes definiciones:

$$max_*(i, j) = max(\begin{matrix} I_{src,*}(i-1, j-1) & , & I_{src,*}(i-1, j) & , & I_{src,*}(i-1, j+1), \\ I_{src,*}(i, j-1) & , & I_{src,*}(i, j) & , & I_{src,*}(i, j+1), \\ I_{src,*}(i+1, j-1) & , & I_{src,*}(i+1, j) & , & I_{src,*}(i+1, j+1) \end{matrix})$$

donde $*$ $\in \{R, G, B\}$. Luego

↳

$$\phi_R(i, j) = \begin{cases} (1 + \alpha) & \text{si } max_R(i, j) \geq max_G(i, j) \text{ y } max_R(i, j) \geq max_B(i, j) \\ (1 - \alpha) & \text{si no} \end{cases}$$

$$\phi_G(i, j) = \begin{cases} (1 + \alpha) & \text{si } max_R(i, j) < max_G(i, j) \text{ y } max_G(i, j) \geq max_B(i, j) \\ (1 - \alpha) & \text{si no} \end{cases}$$

$$\phi_B(i, j) = \begin{cases} (1 + \alpha) & \text{si } max_R(i, j) < max_B(i, j) \text{ y } max_G(i, j) < max_B(i, j) \\ (1 - \alpha) & \text{si no} \end{cases}$$

donde $0 \leq \alpha \leq 1$.

$$\begin{aligned} I_{dst_R}(i, j) &= \min(255, \phi_R * I_{src_R}(i, j)) \\ I_{dst_G}(i, j) &= \min(255, \phi_G * I_{src_G}(i, j)) \\ I_{dst_B}(i, j) &= \min(255, \phi_B * I_{src_B}(i, j)) \end{aligned}$$

Para ello, decidimos procesar nuestra matriz de la siguiente forma:

- Comparación de Tiempo

Al probar lena.bmp de 512x512, con una cantidad de 100 iteraciones y 0.5 como parámetro, el tiempo obtenido fue el siguiente:

Implementación C

Tiempo de ejecución:

Comienzo : 260663107087014

Fin : 260675081294295

iteraciones : 100

de ciclos insumidos totales : 11974207281

de ciclos insumidos por llamada : 119742072.000

Implementación ASM

Tiempo de ejecución:

Comienzo : 260675487848181

Fin : 260676113912895

iteraciones : 100

de ciclos insumidos totales : 626064714

de ciclos insumidos por llamada : 6260647.000

Resultados

Ciclos C: 119742072.0

Ciclos ASM: 6260647.0

Ciclos ASM respecto de C: 5.22844385055 %

Tiempo C: 11974207281

Tiempo ASM: 626064714

Tiempo ASM respecto de C: 5.2284439321 %

- **Waves:** La idea de Waves consistió en combinar la imagen fuente con una imagen de ondas dando tonos más oscuros y más claros en forma de onda. Éstas se generan de manera repetida a lo largo de los ejes x e y. Cada píxel (i, j) de la imagen destino se obtuvo a través del siguiente cálculo:

$$prof_{i,j} = \frac{(x_{scale} \cdot \sin_taylor(\frac{i}{8,0}) + y_{scale} \cdot \sin_taylor(\frac{j}{8,0}))}{2}$$
$$I_{dest}(i, j) = prof_{i,j} \cdot g_{scale} + I_{src}(i, j)$$

donde, además, $I_{dest}(i, j)$ debe guardarse como 0 o 255, si este sobrepasa el intervalo $[0, 255]$.

La función waves requiere la utilización de $\sin_taylor(x)$ que cumple con el siguiente pseudocódigo:

Algorithm 1 $\sin_taylor(x)$

```
1:  $pi \leftarrow 3,14159265359$ 
2:
3:  $k \leftarrow \lfloor \frac{x}{2 \cdot pi} \rfloor$ 
4:
5:  $r \leftarrow x - k \cdot 2 \cdot pi$ 
6:
7:  $x \leftarrow r - pi$ 
8:
9:  $y \leftarrow x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040}$ 
10:
11: devolver  $y$ 
```

Para la implementación de dicho filtro, calculamos la cantidad total de bytes y preparamos un contador para recorrer todos los elementos. Para calcular el \sin_taylor , implementamos nuestras operaciones siguiendo los pasos del pseudocódigo.

Para procesar la imagen, recorrimos los píxeles evaluando con prof habiendo desempaquetado a doubleword y luego los convertimos a float. para calcular la profundidad en un tiempo razonable se preparo un arreglo con cuatro valores del 0 al 3 que se iban a usar como indice de columna para el calculo de profundidad con instrucciones simd, por otra parte se preparo un arreglo lleno de 0's para el calculo del seno de la fila.

- Comparación de Tiempo

Al probar lena.bmp de 512x512, con una cantidad de 100 iteraciones y 2.0 4.0 16.0 como parámetros, el tiempo obtenido fue el siguiente:

Implementación C

Tiempo de ejecución:

Comienzo : 858182624211702

Fin : 858224537429979

iteraciones : 100

de ciclos insumidos totales : 41913218277

de ciclos insumidos por llamada : 419132160.000

Implementación ASM

Tiempo de ejecución:

Comienzo : 858224711685883

Fin : 858225286387416

iteraciones : 100

de ciclos insumidos totales : 574701533

de ciclos insumidos por llamada : 5747015.000

Resultados

Ciclos C: 419132160.0

Ciclos ASM: 5747015.0

Ciclos ASM respecto de C: 1.37117013402 %

Tiempo C: 41913218277

Tiempo ASM: 574701533

Tiempo ASM respecto de C: 1.37117013826 %

- **Rotar:** El filtro rotar toma la imagen de entrada y escribe en la de salida la imagen rotada 45 grados. Cada píxel de la imagen resultante se forma de la siguiente

manera:

$$I_{dst}(x, y) = \begin{cases} I_{src}(u, v) & \text{si } 0 \leq u < I_{src_width} \text{ y } 0 \leq v < I_{src_height} \\ 0 & \text{si no} \end{cases}$$

donde

$$u = c_x + \frac{\sqrt{2}}{2}(x - c_x) - \frac{\sqrt{2}}{2}(y - c_y)$$

$$v = c_y + \frac{\sqrt{2}}{2}(x - c_x) + \frac{\sqrt{2}}{2}(y - c_y)$$

$$c_x = \lfloor I_{src_width}/2 \rfloor$$

$$c_y = \lfloor I_{src_height}/2 \rfloor$$

El filtro rotar no es un buen candidato para paralelizar. Esto se debe a que, al levantar los píxeles consecutivos de la imagen fuente, éstos no deben ser consecutivos en la imagen destino. Esto significa que el orden de los píxeles de la imagen destino es distinto al de la imagen fuente. Nuestra manera de paralelizar fue al realizar los cálculos que le darían la posición correcta a la imagen destino.

Luego del análisis previo, elegimos proceder de la siguiente manera:

En primer lugar, calculamos C_y y C_x con simples shr del ancho y alto de la imagen fuente. Luego, procedimos calculando $\sqrt{2}/2$ habiendo extendido los cálculos a todos los bits del XMM para, luego, realizar los cálculos con todos los píxeles levantados en una iteración.

Para realizar los cálculos tomando el índice x correcto, decidimos almacenar una escalera en memoria de manera tal a evitar ciclos innecesarios y a realizar cuentas velozmente.

Luego, procedimos realizando los cálculos necesarios para hallar u y v . Una vez encontrados dichos valores, obtuvimos máscaras como resultado de las comparaciones que nos explicitarían dónde ubicar los píxeles en la imagen destino.

El paso siguiente consistió en levantar cada píxel procesado dentro del arreglo y ubicarlo en su lugar correspondiente de salida de acuerdo a los offsets obtenidos. Dicho procedimiento fue realizado cuatro veces ya que cada arreglo almacenaba cuatro píxeles. De esta manera, todos los píxeles fueron procesados correctamente y de una manera más eficiente que si hubiésemos procesado bit a bit.

Por último, para recorrer la matriz, decidimos utilizar un contador que le sumáramos a rdi y a rsi de manera tal a no perder el puntero original. El resto de la ejecución se realizó del mismo modo que se describió anteriormente.

Para la realización de dicho filtro, preferimos utilizar floats ya que para usar doubles debíamos desempaquetar una mayor cantidad de veces, utilizar más registros,

complicar el código y demorar la ejecución de los ciclos. Si bien los doubles generan mayor precisión, el tipo de datos utilizado no impidió visualizar la imagen de manera clara por lo que nos decidimos por esa opción.

- Comparación de Tiempo

Al probar lena.bmp de 512x512, con una cantidad de 100 iteraciones, el tiempo obtenido fue el siguiente:

Implementación C

Tiempo de ejecución:

Comienzo : 858387565588994

Fin : 858388404276534

iteraciones : 100

de ciclos insumidos totales : 838687540

de ciclos insumidos por llamada : 8386875.500

Implementación ASM

Tiempo de ejecución:

Comienzo : 858388558871397

Fin : 858388874991387

iteraciones : 100

de ciclos insumidos totales : 316119990

de ciclos insumidos por llamada : 3161200.000

Resultados

Ciclos C: 8386875.5

Ciclos ASM: 3161200.0

Ciclos ASM respecto de C: 37.6922251916 %

Tiempo C: 838687540

Tiempo ASM: 316119990

Tiempo ASM respecto de C: 37.6922244487 %

También tenemos que decir que cosas hicimos para facilitar la implementación en asm, o sea si lo hicimos pensando en como íbamos a resolverlo en asm o si directamente lo hicimos de una manera que fuera rápida. algo que se me ocurre es que preferimos, por ejemplo, usar muchas variables para definir pequeñas operaciones

que usaba una sola ecuacion en vez de que fuera todo resuelto de una para poder darles tipos a esas operaciones y que el resultado fuera mas preciso y acertado a lo pedido.

que transformaciones podriamos hacer para que sea mejor lo que hacemos, mejor tiempo, que fue lo que probamos, etc. Por ejemplo si paso de float a double. En umbralizar pasamos Q de float a double. Pusimos Q, max y min en double porque para hacer las cuentas con doubles era lo mas conveniente.

3. Resultados

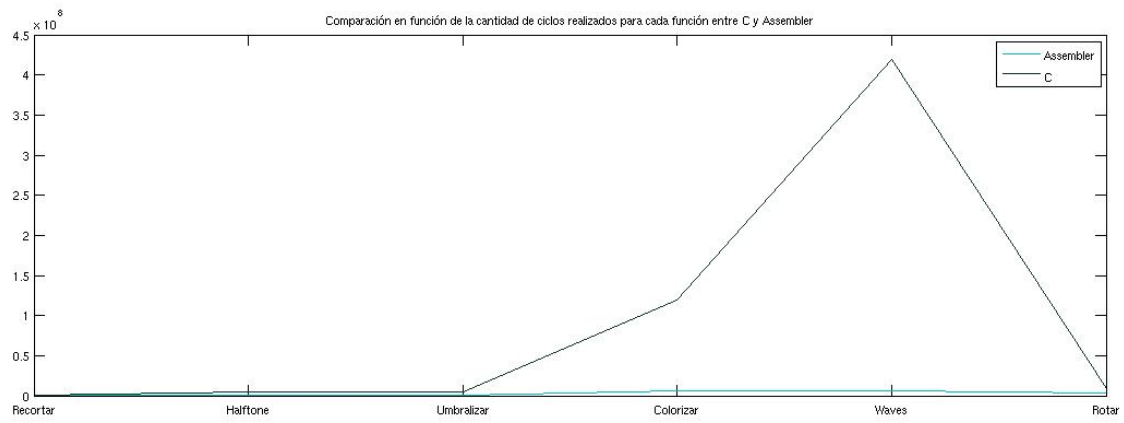
A partir de los resultados otorgados por un script realizado para calcular el tiempo de ejecución de cada función en *C* y *Assembler*, decidimos utilizar objdump para lograr dar una explicación a lo obtenido:

En primera instancia, hallamos que la cantidad de ciclos realizados por *Assembler* es muy menor a la realizada por *C*. Esto se debe a que el *Assembler* de *C* realiza múltiples llamados a memoria en cada ciclo. Por ejemplo, la mayoría de los datos que utiliza son almacenados en la pila por lo que en cada iteración realiza operaciones utilizando, por ejemplo, [rbp-0x28]. Esta es la causa principal del retraso temporal generado por el *C* respecto del *Assembler*.

Por otro lado, nos resultó sorprendente la extensión de los códigos en *Assembler* generados por los *C* en comparación con los nuestros. Para hallarle una explicación a dicha consecuencia, pensamos que el causante de esto fue que, al estar todo almacenado en la pila, los desplazamientos requieren de numerosas instrucciones. Contrariamente, al moverse con un puntero, lo único necesario para el desplazamiento no es más que un sumador.

Por otra parte, hallamos que los códigos en *Assembler* generados por los *C* realizan operaciones innecesarias, como por ejemplo multiplicar por 1, dado que se producen de manera automática.

Por último, encontramos que el ensamblado del *C* no realiza empaquetamientos por lo que la cantidad de operaciones a realizar está multiplicada por 16.



4. Conclusión

Podemos concluir que las causas principales de que el código en C sea mucho más lento que el código *Assembler* dado que su ensamblado no utiliza empaquetamiento y almacena todos los datos en la pila que son llamados en cada iteración. Esto significa que C usa la FPU que es poco performante causando un aumento del tiempo de ejecución.

A partir de aquí, podemos concluir que realizar operaciones empaquetando y desempaquetando los datos acelera enormemente la ejecución de una función como también evitar los llamados a memoria dentro de los ciclos.