Proyecto Individual I, Diseño e implementación de una aplicación para Nitidez en Imágenes

Edgar Chaves González
email: edjchg@gmail.com
Área Académica de Ingeniería en Computadores
Instituto Tecnológico de Costa Rica

***Abstract*—Firstly, this paper talks about the process of filtering an image making convolution with a certain kernel over each pixel of an image, in this case to enhance the borders of a particular gray scale picture. Secondly, it will be shown an overview about the algorithm to achieve convolution in x86 assembly language and reinterpret that number files into pictures.**

***Palabras clave*—Convolución, Ensamblador x86, Kernel de filtrado, pixel.**

I. Introducción

En este trabajo de investigación se abordará el tema de procesamiento de imágenes, en este caso específico se hablara del filtrado de imágenes con el uso de $kernel$ de 3x3, y así lograr el efecto de $sharpening$ y $oversharpening$ haciendo uso del lenguaje ensamblador x86(32 bit), además del lenguaje de alto nivel como Python para poder interpretar los archivos numéricos con los que trabaja ensamblador y reinterpretar-los como imágenes que se pueden visualizar. Finalmente se mostrarán los resultados obtenidos con el algoritmo desarrollado y algunas recomendaciones.

II. Implementación de la Solución

A. Herramientas Utilizadas

- VirtualBox for MacOS.
- Ubuntu 19.10.
- NASM/x86.
- SASM xUbuntu-19.10.
- Python 3.8.

Se utilizaron dichas herramientas pues hay mucha documentación de x86 ensamblado con NASM, y utilizando a gcc como el $linker$, además de la gran cantidad de ayuda como tutoriales que enseñan paso a paso el uso y entendimiento de x86. Se debió usar Ubuntu pues es el sistema operativo más amigable que se logró determinar para trabajar con la herramienta SASM, la cual es un IDE para x86, el cual posee un $debugger$ muy integral, ya que cuenta con visualizador de registros y memoria. Con este $debugger$ fue posible estar monitorizando los valores que tomaban las variables y registros sin tener que gastar recursos en la impresión de dichos resultados en consola. SASM permite trabajar con interrupciones al sistema y por lo tanto trabajar con documentos y archivos externos a SASM, es decir abrir, crear, modificar, actualizar, leer y eliminar archivos. Por ultimo se eligió Python como lenguaje de alto nivel para mostrar las imágenes puesto que tiene bibliotecas muy robustas en cuanto al tratamiento de imágenes.

B. Flujo de la Solución

En primer lugar Python toma la imagen que se desee procesar, la convierte a escala de grises con valores entre 0-255, al finalizar de procesarla este $array$ de números que representan a cada $pixel$ se guardan un archivo de texto plano. Para aplicar la convolución x86 toma el archivo de texto plano que Python creó y convierte cada valor de dicho $buffer$ de tipo $string$ a números que ensamblador pueda manipular. Aplica la convolución de la imagen tomada del archivo con un kernel dado y empieza a aplicar dicho kernel sobre cada $pixel$. El resultado de cada convolución aplicada a cada $pixel$ se guarda en un nuevo $array$. Como se realizan dos procesamientos de convolución para obtener el $sharpening$ y $oversharpening$, entonces dos arrays se guardan en dos archivos de texto plano apartes. Finalmente Python toma estos archivos de texto plano que x86 creó y los reinterpreta como una matriz o mapa de $pixeles$ y finalmente los muestra como imágenes. Esta sección se va a ampliar en la documentación de diseño.

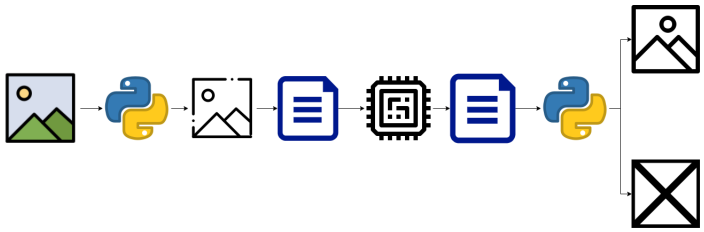

Fig. 1. Diagrama del flujo de la solución.

En el diagrama de la figura 1 se pude ver gráficamente la explicación que anteriormente se dio.

C. Vista rápida del fundamento matemático

Se utilizó la convolución en conjunto con un $kernel$ de 3x3 con el cual se le aplicó a cada $pixel$ para poder conseguir el $sharpening$ y $oversharpening$.

$$g(x,y) = \sum_{i=-a}^{a} \sum_{j=-b}^{a} \omega(i,j) * f(x-i, y-j) \qquad (1)$$

De la ecuación 1 se tiene que $g(x, y)$ es la imagen con la convolución aplicada, $\omega(i, j)$ el kernel y por último $f(x - i, y - j)$ es la imagen original. Además, de la ecuación 1 se puede interpretar de la siguiente forma: por cada x,y de la imagen original, se aplica la suma de las convoluciones entre el kernel y la imagen original, este resultado será el $pixel$ de la nueva imagen.

III. Algoritmo empleado

A. Diagrama de flujo

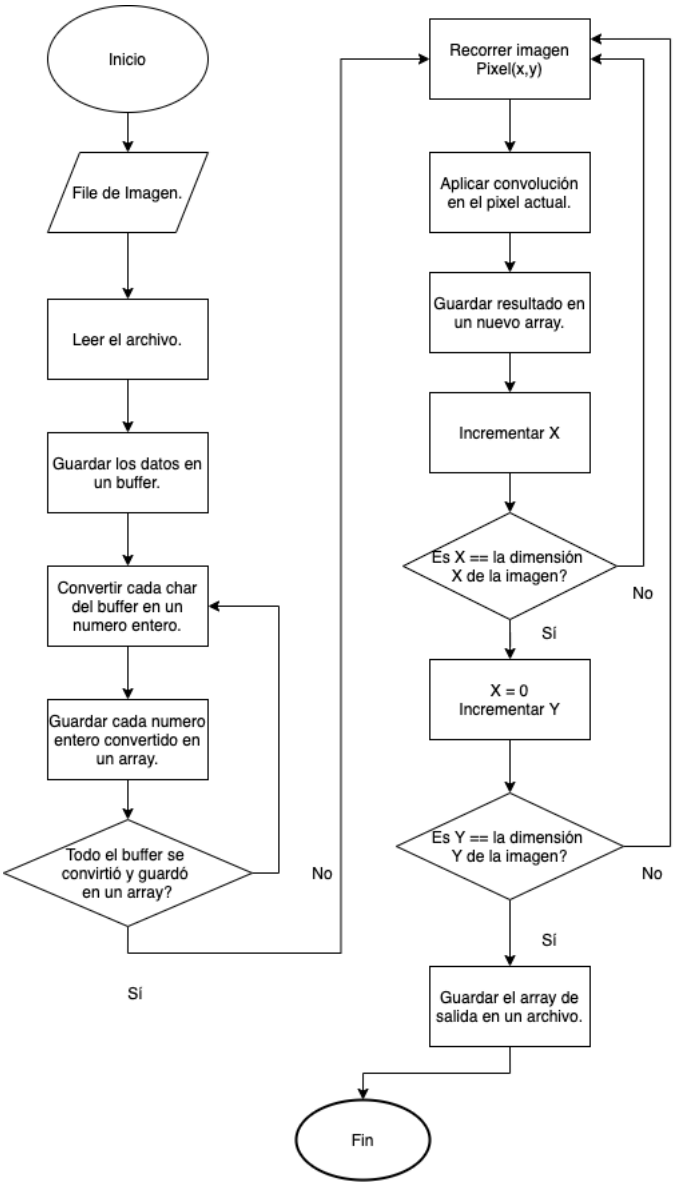

Fig. 2. Diagrama de flujo del algoritmo.

De la figura 2 se puede ver a grandes rasgos el algoritmo creado para poder aplicar la convolución a cada $pixel$ de la imagen por procesar.

B. Traductor de coordenadas

Como se trabajó con $arrays$, es decir estructuras en una dimensión y como las imágenes y kernel son representados en estructuras en 2 dimensiones, para mejor abstracción se decidió crear un algoritmo que convierte coordenadas de una matriz a posiciones de un $array$. Entonces en ensamblador se recorre la imagen con coordenadas x y y y aplicando la siguiente fórmula se obtiene la posición específica de un $array$.

Suponga que tiene la siguiente $matriz$:

$$m = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \tag{2}$$

De la matriz 2 se puede ver que es de 3x3, pero en ensamblador no existen las matrices, por lo tanto se debe abstraer de otra forma, puesto que no se puede acceder a ella viendo a las posiciones como (x, y), si no solo como un único índice i. Por ejemplo, se desea accesar a la posición donde se encuentra el 6, en forma matricial esta posición se representa como $(2, 1)$, y esta ecuación va a resolver qué posición significa $(2, 1)$ en un $array$ de una dimensión, viendo el $array = (1, 2, 3, 4, 5, 6, 7, 8, 9)$.

$$i(x, y) = y \cdot X_{size} + x \tag{3}$$

Siendo $i(x, y)$ el índice al cual se quiere encontrar, y como la posición en las filas, x la posición en las columnas, y finalmente X_{size} como la cantidad de columnas que la matriz tenga.

Por lo tanto si se evalúa la posición que se quería, $(2, 1)$, esto equivaldría a $i(2, 1) = 1 \cdot 3 + 2$, por lo tanto $i(2, 1) = 5$, la posición 5 corresponde al valor 6 de la matriz m, como se quería.

C. Pseudocódigo (x86)

función convolucion(fileName, dimX, dimY, fileSize):
 bucle hasta que (x == dimX y y == dimY):
  bucle hasta que (i == 1 y j == 1):
   hacer indiceKernel(i,j) = j·3 + i
   hacer valorKernel = kernel[indiceKernel]
   hacer indiceImagen(x-i,y-j) = (y-j)· dimY + (x-i)
   hacer valorImagen = fileName[indiceImagen]
   hacerconvolucionParcial=valorImagen*valorKernel
   hacer convolucion1Pixel += convolucionParcial
   condición ($i < 3$)
    hacer i += 1
   si no i = 0, j += 1
  condición ($x <$ dimX)
   hacer x += 1
  si no x = 0, y += 1
fin

IV. Resultados

A. Primeros resultados

Recién acabado la primera versión del proyecto, únicamente en modo de prueba se procesó una vez la imagen, por lo tanto solo se tiene el $sharpening$. Es por esto que se muestra la imagen original, la imagen que Python convirtió a escala de grises y por último la imagen que el algoritmo implementado

en x86 dio como resultado. Cabe aclarar que la imagen utilizada tiene un tamaño de 300x300 $pixeles$, puesto que es únicamente para probar preliminarmente a todo el sistema. Además el kernel utilizado para esta prueba es el siguiente:

$$\omega = \begin{pmatrix} -1 & -2 & -2 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} \tag{4}$$

Fig. 3. Imagen original a color.

La imagen que Python convirtió a escala de grises luce a algo como lo siguiente:

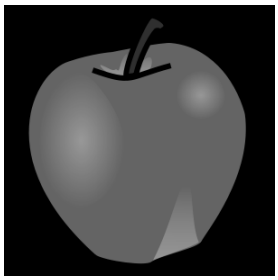

Fig. 4. Imagen original en escala de grises.

Finalmente la imagen ya procesada con el algoritmo de la convolución implementado en x86 luce algo así:

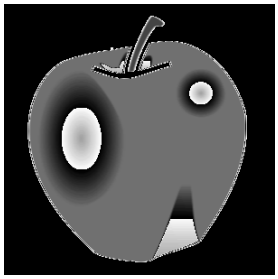

Fig. 5. Imagen procesada con el kernel.

En este caso se puede observar entre la figura 5 y la 4 que los bordes de la ultima aparecen en blanco y otra zonas se ven más demarcadas con un cambio de color, como por ejemplo los brillos y sombras están rodeadas de color negro. Como dato adicional se tiene que x86 duro $6.5s$ para leer el archivo de texto plano, convertir cada elemento $string$ en un numero entero, procesar por completo la imagen y guardar el $array$ de salida en un archivo de texto plano. Sobre una computadora virtual con Ubuntu 19.10, 3 GB de memoria RAM y un procesador core i5 de dos núcleos de 2,7 GHz Intel.

B. Resultados finales

Ya madurado el proyecto en su etapa final se obtuvieron los siguientes resultados, esta vez procesando la imagen de Mario Bros y con el kernel correcto para $sharpening$ se tiene lo siguiente:

$$\omega = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix} \tag{5}$$

La matriz 3 corresponde al kernel necesario para hacer $sharpening$ y $oversharpening$.

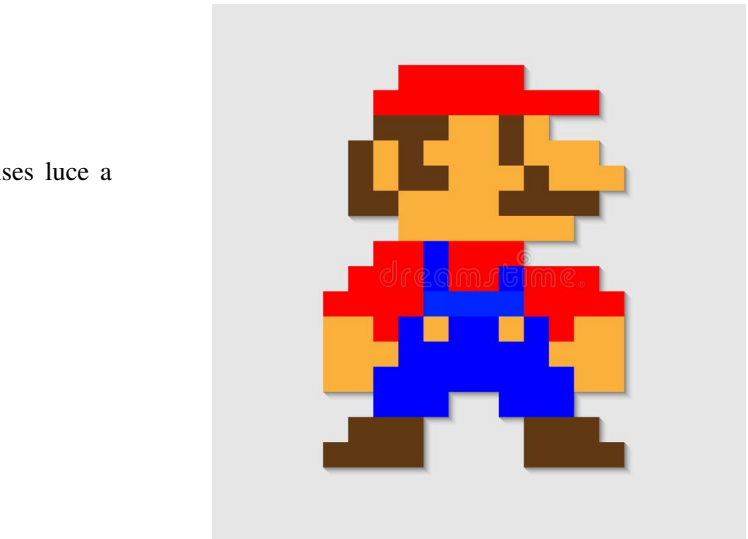

Fig. 6. Imagen original.

Una vez pasada esta imagen 6 por Python en la primera etapa, lo que se va a obtener es una imagen en escala de grises, esto quiere decir que cada $pixel$ va a tener un valor entre 0 y 255. Ahora Mario va a verse algo como esto:

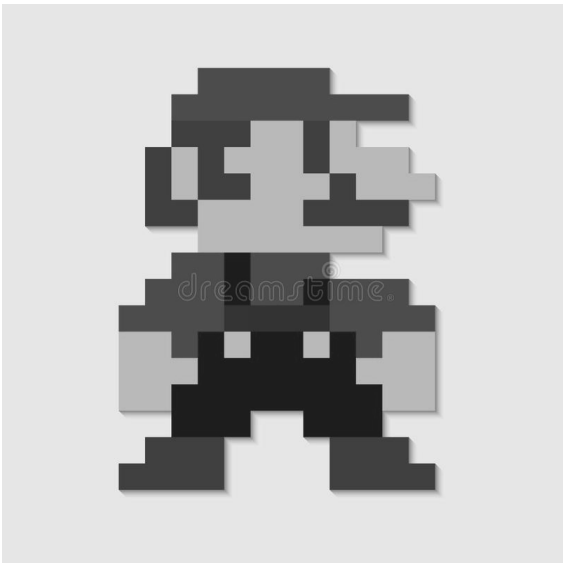

Fig. 7. Imagen original en escala de grises.

Hasta este punto es interesante ver que la marca de agua en ambas imágenes es casi imperceptible, lo cual va a cambiar en las próximas imágenes.

Ahora es momento de que la imagen sea procesada por ensamblador y aplique el kernel 3, el cual se mostró anteriormente. La figura 7 procesada con $sharpening$ se puede ver en la siguiente figura:

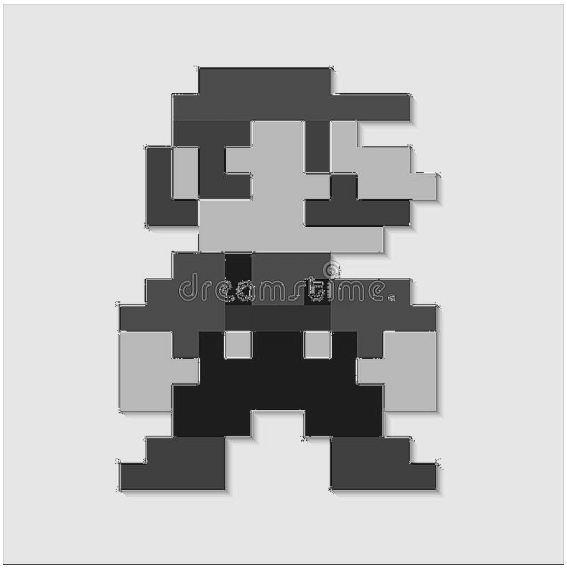

Fig. 8. Imagen original con sharpening.

Para este punto es necesario mencionar que los bordes se ven algo más blancos que en la imagen anterior, y que letras en la mitad de la imagen se han resaltado.

Siguiendo con ensamblador, ahora a la imagen 8 se le aplica de nuevo $sharpening$, lo que va a resultar en $oversharpening$, y el resultado es el siguiente:

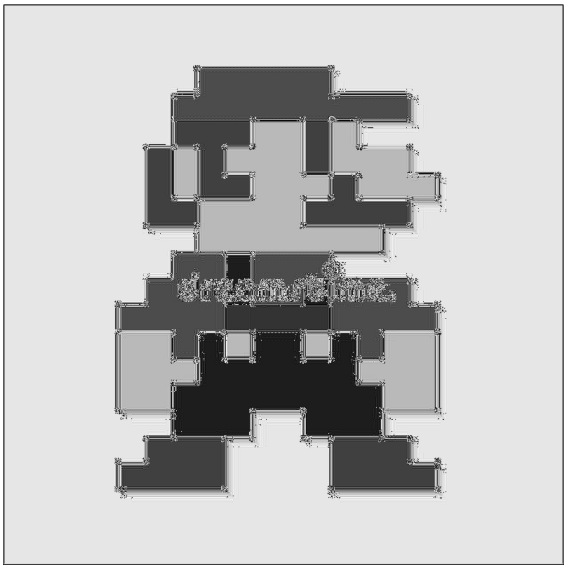

Fig. 9. Imagen original con oversharpening.

Es pertinente decir que el tamaño de estas imágenes es de 800x800 pixeles y tuvo una duración de 519.441 s, alrededor de 8.6 minutos. Sin embargo la parte más interesante es que en donde dura más el proceso, es en lo que se denomina la Primera Etapa, la cual corresponde a la etapa de leer el archivo de texto y convertir cada $string$ del archivo en números enteros con los cuales poder trabajar. Finalmente las imágenes 8 y 9 son mostradas mediante la biblioteca OpenCV en Python.

V. Conclusiones

- El set de instrucciones de Intel x86 es complejo de entender hasta que se trabaja a fondo con su sintaxis. Se debe entender el funcionamiento del $stack$, los diferentes modos de direccionamiento de memoria y la interacción con su poca cantidad de registros de propósito general (eax, ebx, ecx, edx, esi, edi, esp, ebp).
- El filtrado de imágenes mediante convolución es un algoritmo que fácilmente puede ser O(n^2) en cuanto a complejidad computacional, puesto que se recorre un kernel(una estructura en 2D) y una imagen(representada en una estructura en 2D), solo para recorrer una matriz se debe usar dos ciclos anidados, y en este caso se debe recorrer dos matrices. Esto se puede notar en el tiempo que duró el algoritmo en realizar la convolución para una imagen de 300x300 y 800x800, antes mostrado.
- El filtrado de imágenes mediante convolución es útil pues es una forma en la que los programas de edición de imágenes se basan para poder manipular una imagen.

VI. Bibliografía

[1] (2020). Cs.bgu.ac.il. Retrieved 26 March 2020, from "https://www.cs.bgu.ac.il/ caspl152/wiki.files/ps05$_1$52.pdf"

[2] (2020). Csie.ntu.edu.tw. Retrieved 26 March 2020, from https://www.csie.ntu.edu.tw/ cyy/courses/assembly/08fall/lectures/handouts

[3] (2020). Egr.unlv.edu. Retrieved 19 March 2020, from http://www.egr.unlv.edu/ ed/assembly64.pdf Assembly - Arrays - Tutorialspoint. (2020). Tutorialspoint.com. Retrieved 26 March 2020, from https://www.tutorialspoint.com/assembly$_p rogramming/assembly_a rrays$

[4] (2020). Rayseyfarth.com. Retrieved 26 March 2020, from http://rayseyfarth.com/asm/pdf/ch14-c-stream-io.pdf