DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Implementación de filtros utilizando SIMD

Organización del Computador II

Primer Cuatrimestre de 2021

Integrante	LU	Correo electrónico
Matias Cozzani	915/19	matcozzani@gmail.com
Marco Sanchez Sorondo	708/19	msorondo@live.com.ar
Joaquin Gonzalez Vandam	720/19	joaking2011@gmail.com

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://www.fcen.uba.ar

Índice

1. Introducción

El objetivo de este Trabajo Práctico es crear, comparar y analizar distintas implementaciones de tres filtros de imagenes en los lenguajes **C** y **ASM**. El funcionamiento de los mismos y el detalle de las implementaciones será descripto en la sección de desarrollo del informe.

Además proveeremos una serie de experimentos con el objetivo de poder sacar conclusiones respecto de la efectividad de cada una de ellas y como se pueden mejorar cambiando distintos aspectos de la implementación

De esta manera buscaremos contrastar el funcionamiento y la eficiencia del modelo de procesamiento de datos SIMD, utilizado en la implementaciones en assmebler, con respecto al modelo SISD utilizado en las implementaciones en C, además de encontrar distintas maneras en las cuales poder mejorar la implementaciones de nuestros filtros.

2. Desarrollo

2.1. Descripcion de las Implementaciones

Filtro Max: Conceptualmente, el filtro max recorre la imagen recibida con una 'matriz iteradora' de 4x4 pixeles que comienza su recorrido en la esquina superior izquierda de la imagen. Luego de **procesar** los pixeles de la matriz iteradora y escribir los resultados en la imagen destino, el puntero que apunta a su inicio avanza dos pixeles (8 bytes) hacia la derecha tomando así la matriz que está dos pixeles a la derecha (salvo que se haya llegado al borde derecho de la imagen, en cuyo caso se salta a las matrices de pixeles de 4x4 que estan inmediatamente debajo desde el borde izquierdo).

Procesamiento: En cada iteracion del ciclo principal ('.ciclo_matriz'), se lleva a cabo un ciclo que itera a lo largo de cada fila de la matriz de 4x4, levantando 4 pixeles en cada iteracion, computando la suma de componentes de cada uno y detectando el de suma máxima. Más gráficamente:

1. Se levantan 4 pixeles a un registro SSE (xmm0). Se copian en un registro temporal para operar (xmm1).
2. Se setea el registro xmm2 en cero, que acumulará en la i-esima dword la suma de las componentes del pixel i-ésimo.
3. Se aplica una mascara que toma sólo la parte azul de los pixeles y coloca (con un shufle) cada uno de los 4 componentes azules en la parte mas baja de cada uno de los dwords (en el propio xmm1) como números de 4 bytes. Se suma xmm1 a xmm2.
4. Se repite el paso 3 con los colores verde y rojo de manera tal que xmm2 posee las sumas de componentes de cada pixel. Se pasa el valor de xmm2 a xmm3.
5. Se rotan los valores de xmm3 a derecha en el registro xmm4 de y se toma el maximo en xmm3, esto se realiza 3 veces de manera tal que en xmm3 queda la suma maxima.
6. Se toma el dword mas bajo de xmm3 y se lo compara con el registro *rbx*, que guarda el valor de la suma maxima hasta el momento. En caso de que *rbx* sea mayor se procede a repetir todo desde 1 en los proximos 4 pixeles, pero en el caso contrario se lleva a cabo el proceso de identificacion del pixel cuya suma de componentes es máxima (descripto en 7).
7. Se compara por igualdad xmm3 con xmm2 (que, recordemos, tiene el valor de la suma del pixel i-esimo en el dword i-esimo). De esta manera tenemos una mascara (xmm2) que tiene 1s en todos los bits de el/los dword/s respectivo/s a el/los pixel/es cuya suma es maxima (puede ocurrir que mas de un pixel tengan la misma suma!). Tambien se actualiza el nuevo valor de la suma máxima.
8. Retomamos el uso de xmm0 (los pixeles levantados originalmente) mediante el filtrado de el/ los pixel/es de suma maxima con la instruccion PAND XMM2, XMM0.

9. Extraemos el primer dword de xmm2: si es distinto de cero esto significa que es el pixel (se guarda en R14D), de manera que podemos escribir en la imagen destino (paso 10)... Si no, repetimos este paso con el siguiente dword hasta hallarlo.

10. Se escribe sobre los 4 pixeles centrales de la matriz los valores del pixel de suma máxima (guardado en R14D).

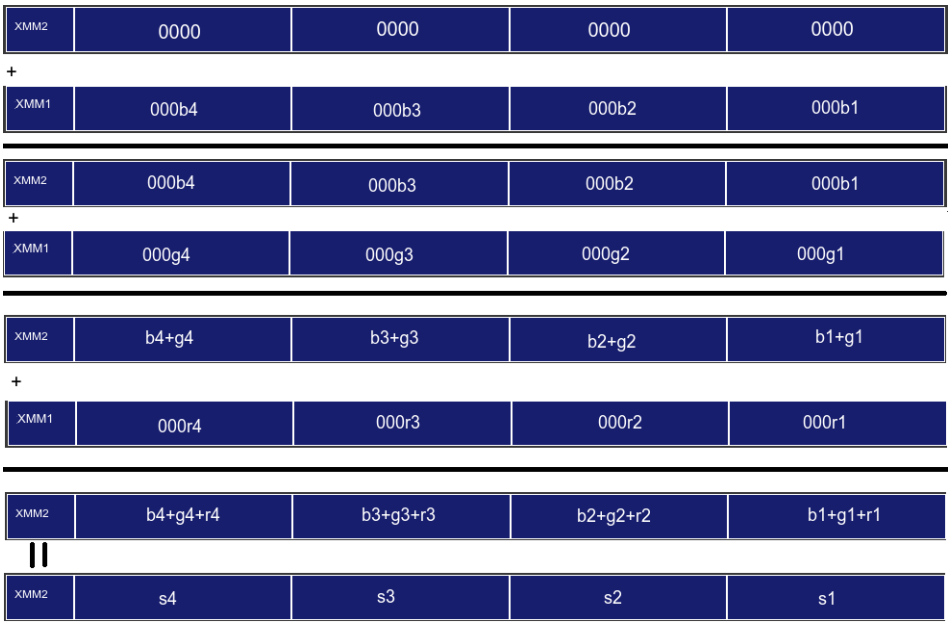

Figura 1: Hasta paso 4

Filtro Gamma: El objetivo del filtro gamma es, valga la redudancia, alterar la gamma de una imagen, lo que corresponde a cambiar la intensidad de los píxeles que la compone. En definitiva este filtro altera el brillo completo utilizando una transformación conocida como `Power Law Transformation` y que aplica la siguiente ecuación a cada una de las componentes de los pixeles de la imagen:

$$Output = 255 * \left(\frac{Input}{255}\right)^{\frac{1}{Gamma}}$$

Nuestro filtro Gamma será una simplificación del filtro general, considerando el parámetro Gamma como 2. A continuación presentaremos el pseudocódigo de la operatoría del filtro presentado que además representará la implementación del mismo en C:

```
Para i de 0 a height - 1:
      Para j de 0 a width - 1:
            dst_matrix[i][j].r = 255.0 * sqrt( (src_matrix[i][j].r) / 255.0 )
            dst_matrix[i][j].g = 255.0 * sqrt( (src_matrix[i][j].g) / 255.0 )
            dst_matrix[i][j].b = 255.0 * sqrt( (src_matrix[i][j].b) / 255.0 )
```

La implemetación en sencilla y se explica bastante con solo leer el pseudo-codigo presentado.

Entonces pasemos a explicar la implemtación del filtro en assembler. La ideas es poder trabajar cuatro pixeles al mismo tiempo de manera que necesitemos iterar menos veces y aprovechemos el modelo SIMD.

Nuestro filtro recorrera cuatro pixeles avanzando sobre todas las filas, desde arriba hacia abajo, antes de moverse a los cuatro pixeles de las columnas siguientes.

XMM1	0	0	0	b1	0	0	0	b2	0	0	0	b3	0	0	0	b4
XMM2	0	0	0	g1	0	0	0	g2	0	0	0	g3	0	0	0	g4
XMM3	0	0	0	r1	0	0	0	r2	0	0	0	r3	0	0	0	r4

Figura 2: Registros XMM1/2/3 luego de guardar los componentes filtrados

Para ello crearemos dos ciclos, uno dentro de otro. El ciclo externo se encargará exclusivamente de avanzar sobre las columnas, es decir, cada vez que completemos la altura de la imagen en el ciclo interno, nos moveremos a los cuatro pixeles internos.

El ciclo interno será el que realice el resto de las tareas que competen al armado del filtro. Primeramente terminara de calcular la dirección efectiva de memoria desde la cual levantar los pixeles correpondientes a la iteración. Estos serán guardados en el registro xmm0, que nunca alterará su valor durante la ejecución de un ciclo interno.

A continuación vamós a separar las componentes de cada uno de los pixeles en cuatro registros separado. Esto se realiza para que al realizar las cuentas con los mismos, no tengamos problemas de representación, es decir, que algun resultado sea mayor de lo que podemos representar en un byte (tamaño de un pixel).

Para separarlos utilizaremos mascaras definidas como constantes en memoria que para cada uno de los componentes, lo deje aislado en la dword que le corresponde. Esto lo haremos guardando las mascaras en un registro extra y luego utilizaon la operacion de lógica y para datos empaquetados en dwords. Asi nos quedan guardadas las componentes iguales de cada pixel dentro de tres registros distintos, uno para cada color, exceptuando las transparecia. Luego con un shuffle de a byte moveremos a la parte más baja de cada dword la componente, dejando los registros como se muestran en la Figura 3.

Una vez separados los componentes los convertimos a floats de precisión simple para poder realizar las operaciones de la ecuación en cada componente. El pasaje a float de antemano es necesario ya que la raíz cuadrada no se puede realizar con numeros enteros y además tener convertidos de antemano ayuda a la eficiencia del filtro.

Ya convertidos en float utilizamos las operaciones de punto flotante para dividir, multiplicar y aplicar raíz cuadrada, para aplicar la `Power Law Transformation` en cada registro con componentes, además que un registro extra donde insertaremos el numero 255 que tenemos guardado en memoria con un define.

Por último, una vez que tenemos los registros con las componentes trabajadas(gamificadas de ahora en adelante), tenemos que volverlas a su representación en byte y volver a construir cada uno de los pixeles. Seguir este procedimiento puede ser un poco dificil así que vamos a profundizar más en el paso a paso.

Lo primero que haremos será empaquetar las datos utilizando las instrucciones de empaquetado `packusw` packusb empaquetando primero a word y luego a byte (no existe una operación que mande dwords directo a byte así que hacemos el paso intermedio a word antes). Como el paquetado requiere que pasemos dos registros como parametros para no dejar la mitad del registro vacío, como segundo parametro usaremos el mismo registro cosa que no queden mezclados los datos y nos queden al final cuatro veces repetida una dword que en cada byte tiene una componente gamificada. (Figura 4)

Luego con un set de mascaras especiales, de cada registro buscaremos quedarnos con una sola de estas dwords. En particular nos quedaremos con la primera para los azules, la segunda para los verdes y la tercera para los rojos. De este manera podremos combinar los registros para que nos queden combinados todos los componentes

Entonces finalmente restaría reacomodar los componentes para poder reconstruir los pixeles con componentes gamificadas. Para ello preparamos un registro que levanta una mascara predefinida que servirá para, con un shuffle de a byte, de manera que los elementos queden reacomodados, cada uno en la dword representativa del pixel que los contenia.(Figura 3)

Por último nos queda recuperar la transparencia, pero como los pixeles orginales quedaron inalterados en un registros que designamos fijo para los mismo, podemos recuperar la trasnparecia facilmente con

XMM1	b1	b2	b3	b4	b1	b2	b3	b4	b1	b2	b3	b4	b1	b2	b3	b4
XMM2	g1	g2	g3	g4	g1	g2	g3	g4	g1	g2	g3	g4	g1	g2	g3	g4
XMM3	r1	r2	r3	r4	r1	r2	r3	r4	r1	r2	r3	r4	r1	r2	r3	r4

Figura 3: Registros XMM1 con los componentes gamificados y empaquetados

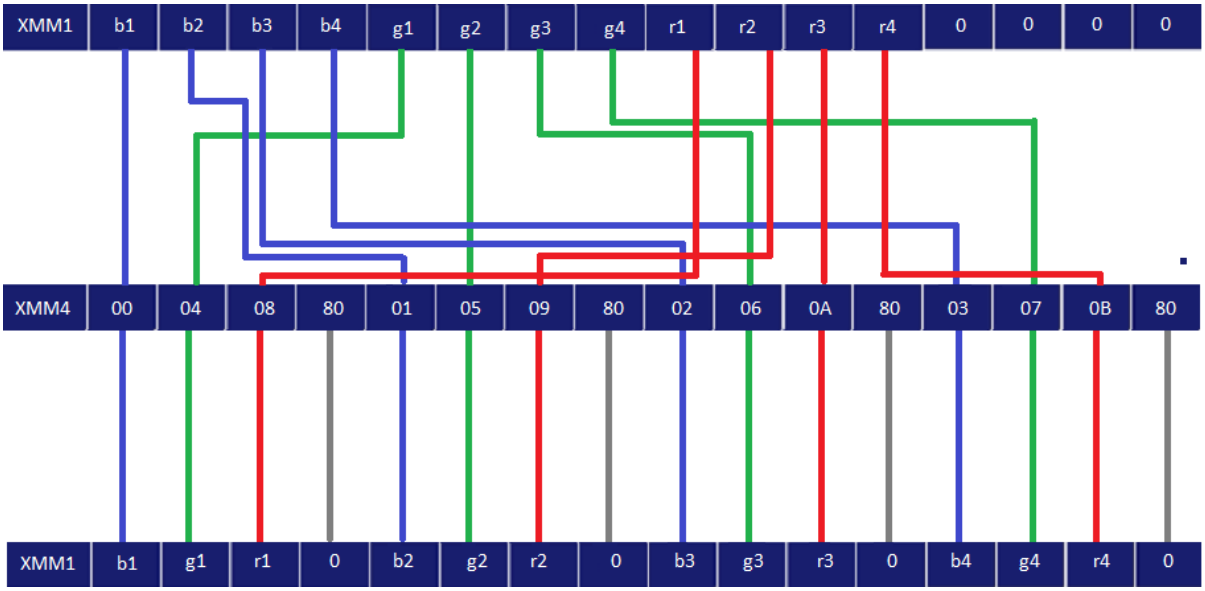

Figura 4: Shuffle que reacomoda los componentes para reconstruir el pixel

una mascara que la filtre y luego agregarla al registro con los pixeles gamificados tambien facilmente, ya que cuando reconstruimos los pixeles lo hicimos de manera que el byte que correspondía a la transparecia quedara vacío, cosa que fuera más facil recuperarla.

Finalmente, concluimos la iteración escribiendo en destino los pixels gamificados.

Filtro Funny: El objetivo del filtro funny es realizar una combinación entre la imágen de entrada y una "nueva" imágen, resultado de utilizar el valor de cada una de las componentes individuales de cada píxel para realizar distintos efectos.

Para cada iteración del filtro funny se levantan 4 pixeles de la imágen original, determinados por la suma del puntero inicial que apunta al "pixel 0" de la imágen original, agregando un factor de desplazamiento por columna, dado por el contador del núcleo del ciclo, y un factor de desplazamiento por fila dado por un contador auxiliar que se incrementa cada vez que recorrimos todo el ancho de la imágen, al mimso tiempo que vuelve a 0 el contador núcleo del ciclo. Una vez que hemos cargado a un registro particular aquellos 4 pixeles que deben ser tratados, se los filtra a cada uno de ellos con una máscara armada específicamente para poder descomponer cada pixel en sus componentes. Este procedimientos de separar los componenetes de los pixeles es el mismo que se aplicó al resto de los filtros, se puede ver en la Figura 5.

Una vez que hemos logrado aislar las componentes de un pixel, es momento de comenzar a trabajar con ellas. Dado que el cálculo de cada componente utiliza el i y el j -es decir, los números que indican tanto a que columna como a que fila corresponde el pixel- se realiza una ligera preparación de un registro que será usado en repetidas ocasiones: básicamente consta de insertar en dos registros distintos estos indices. Para el registro que va a contener las j es necesario notar lo siguiente:

Para cada iteración, nosotros partimos de un pixel determinado por (i, j). Sin embargo, nosotros levantamos 128bits a partir de ese registro, lo que implica levantar 4 pixeles -pues cada pixel tiene 4

componentes de 1 byte cada una, luego cada pixel “mide” 4bytes, por lo que para levantar cuatro pixeles necesitamos 16bytes = 128bits-. Sin embargo, al utilizar la estrategia de programación SIMD y trabajar varios pixeles al mismo tiempo, debemos notar que nuestro contador que representa a la j tiene como valor el j del primer pixel. Para poder realizar las operaciones correctas a cada pixel, es necesario ir sumando de a 1 a ese contador e ir insertandolo en la posición correspondiente.

También copiamos estos resultados a otro registro, que nos servirá en particular para trabajar con las componentes azules.

Como cada una de estas componentes debe ser trabajada de manera particular, podemos dividir el análisis del algoritmo básicamente en tres partes:

- Componente azul:

```
funny_b = 10 * sqrt( (i*2+100) * (j*2+100))
```

Para la componente azul, hay que tomar algunas consideraciones.

Debido a algunas inestabilidades con respecto a la diferencia de representación entre floats y doubles, se determino que era preciso trabajar con doubles. Por esto mismo es que la manera en la que vamos a preparar los datos es bastante particular en este caso, debido a que a diferencia de un float single-precission -que ocupa 32bits- un float double-precission ocupa 64bits. Esto quiere decir que por registro, solo podemos almacenar dos valores double-precision. Por ende, tenemos que romper en dos registros las j correspondientes a los pixeles.

Primero, vamos a realizar una conversión de las componentes j a float double-precision, recordando que el convert a double solo convierte los dos valores que están en la posición menos significativa del registro. Por ello, convertimos las ”primeras dos”j a un registro auxiliar, utilizamos un shift de a byte para correr los valores de las posiciones más significativas a las posiciones menos significativas del registro y nuevamente utilizamos una conversión a double. De esta manera, ahora tenemos en dos registros separados las j ya convertidas a valores de doble precisión.

También preparamos los registros que van a tener las constantes auxiliares necesarias para llevar acabo la operatoria y los convertimos a float de doble precision.

Por último como ya tenemos preparados todos los registros para realizar la operatoria del filtro, sencillamente realizamos la operatoria correspondiente.

En este punto entonces tenemos en dos registros separados los resultados de toda la operatoria para dos j, y en otro los resultados de toda la operatoria para los otros dos j.

- Componente roja:

```
funny_r = 100 * sqrt(abs(j-i))
```

Para la componente roja, nos remontamos a los registros que habíamos preparado con anterioridad para los índices y pasmos sus valores a registros auxiliares para poder realizar las operaciones sin alterar los mismo, ya que son necesario para el calculo del componente verde.

Realizaremos la resta del i y del j correspondiente a cada pixel, tomamos el valor absoluto de la misma y la convertimos a float single-precision con una operación de conversión.

Tomamos la raiz cuadrada de las operaciones anteriores, movemos a un registro auxiliar una máscara que contiene 100 repetido 4 veces en la parte menos significativa de cada dword y realizamos la multiplicación, guardamos el resultado y terminamos con la parte roja.

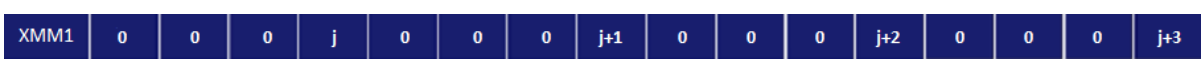

Figura 5: Registro que contiene a las j

- Componente verde:

```
funny_g = (abs(i-j)*10) / ((j+i+1)/100)
```

Para la componente verde, preparamos primero el dividendo de la operación, copiando los valores del registro que tiene a la i a un registro auxiliar, realizamos la resta de a dwords de los registros que contienen a la i y a las j, y tomamos su valor absoluto.

Convertimos los valores obtenidos a float single-precision al mismo tiempo que convertimos un registro en el que previamente habíamos cargado cuatro dieces a la posición menos significativa de cada dword y realizamos la multiplicación entre ellos.

Para el divisor, nuevamente recuperamos a las j, las sumamos con la i y le sumamos 1 -nuevamente, utilizando máscaras-

Convertimos todo a float, y realizamos la división del resultado de la operación anterior con 100.

Finalmente, tenemos el dividendo y el divisor preparados, lo único que resta es realizar la división entre ellos, la guardamos en un registro y terminamos de trabajar la componente verde.

Luego de haber trabajado las partes, empezamos a preparar los datos para escribir en las posiciones de memoria correspondientes.

El primer paso es convertir ambos registros donde quedaron guardadas las componentes azules separadas nuevamente a enteros.Notamos en este punto que como vamos a tener que volver a juntar a ambos registros en uno solo, es preciso correr el resultado de la conversión, que quedo almacenad en la dwords mas bajas del registro. Por esto mismo, hacemos un shift a izquierda de manera tal que corremos los resultados de la conversión de uno de los dos regitros hacia la parte más significativa del registro, luego solo hace falta hacer una operación OR entre los dos registros para volver a unirlos en uno solo. Seguidamente, convertimos los registros que albergaban al resto de las componentes (rojas y verdes).

Una vez que ya hemos convertido todos los resultados nuevamente a enteros, y hemos recuperado a un solo registro los valores correspondientes a las componentes azules, saturamos esos registros creando una mascara que tiene FF repetido 4 veces en la parte menos significativa de cada dword. Esto se realiza para poder hacer una operación lógica AND entre los registros que albergan los resultados convertidos y FF, de manera tal que solo nos quedamos con el byte menos significativo de cada número representado en dos bytes, logrando una idea de saturación del valor que había previamente.

```
dst_matrix[i][j].r = SATURAR( TRUNCAR(funny_r)/2 + src_matrix[i][j].r/2 )
dst_matrix[i][j].g = SATURAR( TRUNCAR(funny_g)/2 + src_matrix[i][j].g/2 )
dst_matrix[i][j].b = SATURAR( TRUNCAR(funny_b)/2 + src_matrix[i][j].b/2 )
```

Finalmente, dividimos por dos tanto a las componentes originales de los pixeles como a los resultados de la operatoria para cada componente y realizamos una suma con saturación entre ellas (lógicamente, no queremos desbordar esos resultados sino saturarlos). Reacomodamos los resultados de esas sumas a través de shifts, desalineando los valores verticalmente respecto de cada uno de los registros de manera tal que al realizar una suma vertical entre esos registros estemos uniendo en uno solo los valores.

Por último, utilizamos una máscara auxiliar para poder recuperar la transparencia original de cada pixel levantado, la insertamos dentro del registro en donde habíamos reconstruido los pixeles y los escribimos en memoria realizando exactamente el mismo calculo que se utilizó para levantar de memoria los pixeles correspondientes.

2.2. Comparacion

En esta sección vamos a procedes a comparar y analizar las compareciones en **C** y **ASM**.

Lo primero a notar al momento de realizar un análisis extensivo respecto de la eficiencia de las implementaciones es que realmente lo que estamos haciendo es una comparación entre dos estrategias algoritmicas distintas para la solución de problemas que involucran grandes cantidades de datos. Por un

lado, las implementaciones en C tienen un enfoque clásico, sencillamente recorriendo pixel a pixel la imágen, obteniendo los pixeles, realizando los calculos y volviendo a escribir en memoria. No utilizan ningún paso adicional de preparación de datos para ser trabajados en simultáneo. Por otro lado, las implementaciones en ASM estan diseñadas específicamente para aprovechar la potencia de la estrategia SIMD, que implica procesar, trabajar y escribir múltiples datos en simultáneo para cada iteración del algoritmo. Esto obviamente decrece terriblemente la cantidad de ciclos necesarios para completar la tarea particular, pero también implica tanto mayor complejidad a la hora de escribir el código, como una potencial mayor cantidad de accesos a memoria -generando un mayor costo- lo que puede terminar significando que no hayamos ganado rendimiento.

Para el análisis de las implementaciones se realizaron series de 50 mediciones de ciclos de clock para cada una de ellas, tanto para los distintos filtros en sus distintas versiones como para los distintos posibles niveles de optimización que nos ofrece el compilador gcc, en el caso de la implemtentacion en **C**. Con el objetivo de obtener muestras lo mas puras posibles, se redujo al máximo el uso de otros programas durante la ejecución de los filtros a fin de disminuir todo lo posible que otros procesos que estuvieran corriendo en simultáneo afectaran las mediciones. Otro punto importante a notar es que la métrica utilizada para este análisis es univariada, solamente vamos a medir la cantidad de ciclos de clock que necesita el procesador para poder llevar acabo la ejecución de la solución; en otras palabras, solamente vamos a medir el rendimiento en términos del tiempo de ejecución necesario para correr el programa. Si se quisiera realizar un análisis mas exhaustivo del rendimiento bien podría realizarse un análisis multivariado, incorporando a la métrica mediciones del tiempo de compliación, costo de memoria para la ejecución del programa, étc.

- **Filtro Max**:

  En el caso de Max, notamos que la implementación en **ASM** usando un acercamiento SIMD es muy superior que la versión de **C** utilizando el nivel de optimización O0. Resulta lógico que esto sea así ya que si no contamos con un paso adicional de optimización, por el mero hecho de la diferencia algoritmica entre una y otra implementación esperamos que la versión **ASM** termine en una menor cantidad de iteraciones. Sin embargo, a medida que empezamos a elevar la optimización provista por el compilador GCC notamos que la diferencia se aplana con rapidez, llegando incluso a ser más rapida la implementación **C** con nivel O3.

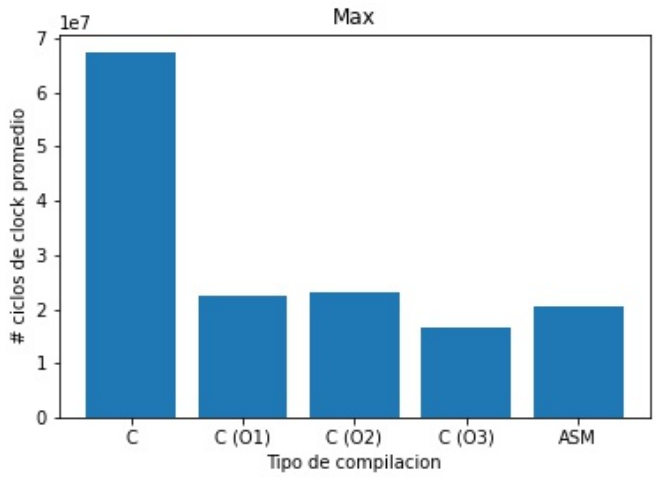

Figura 6: Cantidad de ciclos de clock por tipo de compilación

Claro que esto no es gratis, ya que a la hora de realizar una optimización tan agresiva perdemos mucha facilidad para debugear el programa, así como elevamos el tiempo de compilación: si bien estos filtros tienen implementaciones relativamente cortas, para códigos muy grandes y pesados es probable que esto se convierta en un problema. En promedio, la implementación en ASM resultó ser dos veces mas rápida que la implementación en **C** sin optimización. A medida que escala el nivel de optimización utilizado, la diferencia se hizo mucho mas estrecha: para los niveles de optimización O1 y O2, la diferencia entre la implementación en **C** y **ASM** es de aproximadamente 10 %.

Finalmente, la implementación en **C** con nivel de optimización 3 fue un 20 % mas rápida que la implementación en ASM. En parte esta diferencia puede deberse a que para poder llevar acabo la implementación SIMD de Max fue necesario realizar muchas operaciones intermedias entre que se levantan y se escriben los datos para poder procesarlos, en particular un fuerte uso de máscaras y shuffles que son operaciones muy costosas computacionalmente.

- **Filtro Gamma**:

  En el caso de Gamma ya no hay dudas con respecto a las diferencias de performance, la implementación en **ASM** demuestra ser mucho mas eficiente en términos de tiempos de ejecución incluso para niveles de optimización mas agresivos. Si comparamos el rendimiento de la implementación de **ASM** contra el nivel O0 vemos que en promedio la versión **ASM** solo necesitó el 12 % de los ciclos de clock que necesitó la implementación en **C**. Si bien esta diferencia se aplana a medida que aumentamos el nivel de optimización vemos que incluso para el nivel O3 la implementación de **ASM** solo requirió el 50 % de los ciclos de clock.

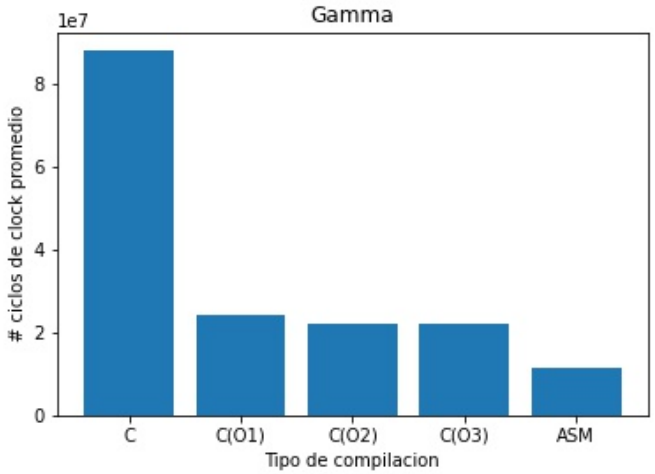

Figura 7: Cantidad de ciclos de clock por tipo de compilación

Esto probablemente se debe a que la implementación de Gamma en **ASM** es mucho mas ”directa”, si bien también se aprovecha fuertemente el uso de máscaras no es necesaria la utilización de operaciones costosas como el shuffle, requiriendo solamente operaciones aritméticas y lógicas entre los resultados que son mucho menos costosas computacionalmente.

- Filtro Funny:

  Como en Gamma, la implementación **ASM** de Funny nuevamente prueba ser mucho más eficiente que la implementación en **C** independientemente del nivel de optimización utilizado. Al igual que Gamma, funny requiere muchas operaciones pero principalmente de conversión, aritméticas y lógicas pero no necesita un fuerte uso de shuffles ni extracciones de datos.

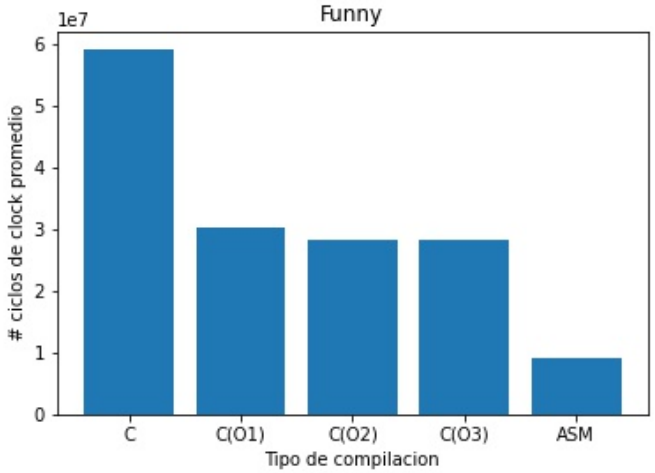

Figura 8: Cantidad de ciclos de clock por tipo de compilación

Comparando las implementaciones y los niveles de optimización, vemos que si comparamos contra el nivel O0 solamente fueron necesarios el 15 % de los ciclos de clock para la implementación de **ASM**. Si bien al momento de elevar la agresividad de la optimización la diferencia se aminora, seguimos viendo que para el nivel de optimización 3, por ejemplo, la implementación en **ASM** termina solamente utilizando el 32 % de los ciclos de clock que los utilizados por la implementación en **C**.

3. Experimentación y Resultados

3.1. Experimentación

El objetivo de esta sección es mostrar una serie de experimentos realizados con el fin de mostrar como ciertas partes de la implemtación en **ASM** son muy pesadas para el procesador, consumiendo un gran cantidad de ticks de reloj, y como al cambiarlas podemos mejorar drásticamente su implementación.

Cada experimento será realizado con un ejecutable que realiza una serie de 50 ejecuciones de un filtro sobre una imagen dada. En particular, a lo largo de la experimentación, mostraremos los resultados obtenidos para el filtro Gamma.

Además utilizaremos diversos tamaños de imagen para, por un lado para ver como trabajan nuestros filtros con resoluciones muy altas (4k, 2k y 1080p), y por otro poder ver más facilmente el impacto de los cambios que realizaremos.

Para poder reproducir estos experimentes se proveera del codigo **C** para correr las 50 series de los filtros, las carpetas con las imagenes utilizadas, ordenadas por resolución y los archivos **ASM** modificados. Para poder reproducir correctamente los experimentos, se recomienda mantener al mínimo el uso de otras aplicaciones durante la ejecución. Nuestros datos fueron obtenidos corriendo los ejecutable en Linux con solamente la consola abierta.

Las imagenes sobre las cuales estarán hechos los experimentos serán:

- TheGodfather.bmp de 800x600 pixeles
- HardCandy.bmp de 1280x720 pixeles

- WolfOfWallstreet.bmp de 1920x1080
- AvengersEndGame.bmp de 2048x1080
- BladeRunner2049.bmp de 3840x2160

Experimento 1: Al terminar de implementar y testear la correctitud de las funciones escritas en ASM, nos dimos cuenta que en las mismas abundaba el uso de la instrucción shuffle, que es una instrucción particularmente costosa. Entonces surge una pregunta ¿Podémos realizar la misma operación que hacemos con los shuffle con otras instrucciones que sean más livianas? ¿Esto reduciría significativamente la cantidad total de ciclos del programa?

Para resolver estos interrogantes y analizando nuestro código realizamos los siguientes cambios:

En todos los filtros utilizamos instrucciones de shuffle para aislar las componentes de cada color de un pixel para luego operar con ellas. Además, en Gamma, tenemos un procedimiento particular para reconstruir el pixel una vez que se "Gamificaron" los componentes originales. Considerando que las instrucciones de éste tipo resultan muy costosas, y que además se encuentran embebidas en ciclos, nos proponemos dar formas alternativas de aislar las componentes de los pixeles con instrucciones mas livianas como 'y' logicos y 'shifts'.

Hipótesis Cambiar el uso de instrucciones complejas para el procesador como 'shuffle' por instrucciones logicas básicas reducirá significativamente la cantidad absoluta de ciclos de clock que insume la ejecución del programa. Además, las mejoras de éstas funciones tienen un mayor impacto con cantidades mayores de pixeles totales.

Procedimiento Corrimos el filtro Gamma 50 veces sobre 5 imagenes distintas y de distintos tamanios para probar que nuestra hipotética mejora del rendimiento vale para imagenes de distintas características. Los cambios realizados en el código del filtro Gamma fueron:

Cambios en Gamma a la hora de filtrar el pixel:

```
movdqu xmm1, xmm0       -------------->     movdqu xmm1, xmm0
movdqu xmm2, xmm0       -------------->     movdqu xmm2, xmm0
movdqu xmm3, xmm0       -------------->     movdqu xmm3, xmm0
movdqu xmm4, [maskSB]   -------------->     movdqu xmm4, [maskB]
movdqu xmm5, [maskSG]   -------------->     movdqu xmm5, [maskG]
movdqu xmm6, [maskSR]   -------------->     movdqu xmm6, [maskR]
pshufb xmm1, xmm4       -------------->     pand xmm1, xmm4
pshufb xmm2, xmm5       -------------->     pand xmm2, xmm5
pshufb xmm3, xmm6       -------------->     pand xmm3, xmm6
                        -------------->     psrldq xmm2, 1
                        -------------->     psrldq xmm3, 2
```

Cambios en Gamma a la hora de reconstruir el pixel:

```
packusdw xmm1, xmm1             ---------->
packusdw xmm2, xmm2             ---------->
packusdw xmm3, xmm3             ---------->
packuswb xmm1, xmm1             ---------->
packuswb xmm2, xmm2             ---------->     movdqu xmm4, [maskB]
packuswb xmm3, xmm3             ---------->     pand xmm1, xmm4
movdqu xmm4, [maskBGamma]       ---------->     pand xmm2, xmm4
movdqu xmm5, [maskGGamma]       ---------->     pand xmm3, xmm4
movdqu xmm6, [maskRGamma]       ---------->     pslldq xmm2, 1
pand xmm1, xmm4                 ---------->     pslldq xmm3, 2
pand xmm2, xmm5                 ---------->     por xmm1, xmm2
pand xmm3, xmm6                 ---------->     por xmm1, xmm3
paddd xmm1, xmm2                ---------->
paddd xmm1, xmm3                ---------->
movdqu xmm2, [maskShuffleGamma] ---------->
pshufb xmm1, xmm2               ---------->
```

Resultados del experimento 1 Despues de realizar los cambios especificados, observamos una mejorar significativa en la cantidad de ticks de reloj que requiere el procesador para ejecutar el filtro. Como podrán observar en las Figuras 9, 10, la reducciones más importantes se observan en las imagenes de más alta resolución.

Se observan reducciones del 2.112407 %, 3.442930 %, 13.516547 %, 5.635709 %, 7.853000 % en la cantidad de ciclos promedio, respectivamente para cada una de las imagenes. Nótese que el incremento de la mejora en éstos porcentajes no es perfectamente creciente al aumentar la cantidad de pixeles de la imagen pues por alguna razón, la imagen de resolución 1920x1080 es considerablemente superior a las de resolución 2048x1080 y 3840x2160 a pesar de tener menos pixeles. Al principio, se nos ocurrió que ésto se podía atribuir a la existencia de outliers 'muy grandes' en la cantidad de ticks para alguna muestra obtenida corriendo el filtro original (que subieran el promedio) o bien a outliers 'muy chicos' para la implementación optimizada (que bajaran el promedio), pero ésto no ocurrió. Consideramos que no tenemos los conocimientos suficientes para justificar éste salto pero creemos que puede atribuírsele a varios factores (como a eventualidades del sistema operativo o procesador, ruido del sistema, particularidades de la imagen utilizada, etc).

Como última observación, exceptuando el pico de la imagen en 1080p, el crecimiento del rendimiento respecto del tamaño de imagen pareciera mas o menos lineal.

Experimento 2: Una vez realizamos nuestras mejoras en cuanto a instrucciones pesadas revisamos en que otros aspectos podíamos mejorar las implementaciones. Revisando el codigo mientras haciamos los cambios para el experimento 1, nos dimos cuenta que la cantidad de accesos a memoria que realizaba nuestra implementación era demasiado grande. Entonces nos propusimo mejorar el codigo en este aspecto.

Pensando como mejorar, se nos ocurrió que podríamos hacer dos cosas, la primera sería no utilizar tantas mascaras distintias para filtrar componentes y buscar una manera de, a partir de una sola mascara, generar las demás utilizando shifts.

Lo segundo sería intentar guardar las mascaras en algún registro que designemos fijo y que no cambiemos durante las iteraciones de los ciclos, de manera que solo carguemos de memoria las mascaras una sola vez.En el estado actual del código, el ciclo interno levanta las mascaras por cada iteración para poder filtrar los componentes.

Hipótesis Reducir el número de accesos a memoria que realiza la solución mejorando la utilización de máscaras reducirá la cantidad de ciclos de clock que utiliza el procesador para ejecutar el programa. Consideramos también que impatará particularmente a medida que la resolución de la imágen escala.

Procedimiento Con este objetivo cambiamos como filtramos los componentes y como construimos el pixel al final de la iteracion. Para el filtrado, nos quedamos solo con la mascara que sirve para filtrar el color azul y a partir de la misma generamos la que sirve para filtrar el verde y el rojo, utilizando dos shift a izquierda.

Para construir el pixel, utilizamos la misma mascara que para filtrar el azul shifteada a izquierda, para recuperar el valor de la transparencia de lo pixeles originales y ponerlos en la pixels gamificados.

Por último, esta mascara que utilizamos y otro valor constante utilizado para el calculo de la ecuación de gamificación, los ponemos previo al inicio de los ciclos en dos registros xmm fijos (xmm14 y xmm15), evitando así que los tengamos que levantar con cada iteración. Finalmente, cada vez que debamos modificar el valor la mascara de xmm15, vamos a pasarlo a algun otro registro para que las iteraciones no modifiquen el valor de la mascara y tengamos que ir a recuperarlo de memoria.

```
movdqu xmm4, [maskB]  -------------> pand xmm1, xmm15
movdqu xmm5, [maskG]  -------------> movdqu xmm14, xmm15
movdqu xmm6, [maskR]  -------------> pslldq xmm14, 1
pand xmm1, xmm4       -------------> pand xmm2, xmm14
pand xmm2, xmm5       -------------> pslldq xmm14, 1
pand xmm3, xmm6       -------------> pand xmm3, xmm14
psrldq xmm2, 1        -------------> psrldq xmm2, 1
psrldq xmm3, 2        -------------> psrldq xmm3, 2
```

Resultados del experimento 2 Luego de realizar estos cambios observamos que no hubo una mejora significativa con respecto a la cantidad de ciclos de reloj necesarios para ejecutar el programa, ni siquiera en imágenes con mucho mayor resolución. De hecho podemos observar que incluso, para la imagen de 1080p estamos obteniendo un peor rendimiento, aunque este no es significativo. Aun así esto ya nos estaría índicando que tenemos algún compartamiento particular para esta imagen y que, al contrario de lo que concluimos en el experimento uno, este valor extraño, no tiene que ver con la muestra obtenida. Esto se puede ver en las Figuras 12,13 y 14.

Como podrán observar en la figura 13 porcentualmente las mejoras/perdidas son 0.445736 %, 0.178778 %, -0.804827 %, 2.167832 %, 0.703031 %. A diferencia de el experimento anterior donde, excepto por el pico particular en la imagagen 1080p, el incremento parecia mas o menos lineal, aca no pareciera haber mucha relación entre las mejoras/perdidas y los tamaños de la imagenes.

Entonces esto nos genera dudas con respecto la manera en que realizaron los experimentos. Al haber utilizado distintas imagenes para representar distintos tamaños nos tomamos a cuenta como las particularidades de cada imagen individual podría afectar a los valores de mejora en el rendimiento del filtro. Supones ahora que quizá esta es la causa de que tengamos algunos valores extraños durante ambos experimentos.

4. Conclusión

La conclusión a la que llegamos luego de realizar el trabajo es que hay una muy clara diferencia entre utilizar estrategias SIMD y no hacerlo, al mismo tiempo que pudimos observar que la complejidad de implementar código SIMD es mucho mayor, no solo a la hora de escribir el código propiamente dicho sino también a la hora de intentar debugearlo.

En la mayoría de los casos la implementación utilizando SIMD demostró ser mucho mas eficiente que la implementación de C. Los expermientos realizados nos indican que a pesar de ser muy beneficioso, hay

que ser extremadamente cautelosos a la hora de programar SIMD, precisamente porque intenado preparar los datos para su procesamiento simultáneo podemos incurrir en estrategias terriblemente costosas -o directamente, érroneas- generando el efecto contrario al esperado.

Como conclusión adicional podemos decir que es muy interesante el poder de optimización que nos ofrece el compilador GCC en su forma mas agresiva que logró -como se evidenció en Max- generar incluso una eficiencia mayor que la implementación SIMD, sin tener que incurrir en la complejidad de la escritura SIMD.

5. Apéndice

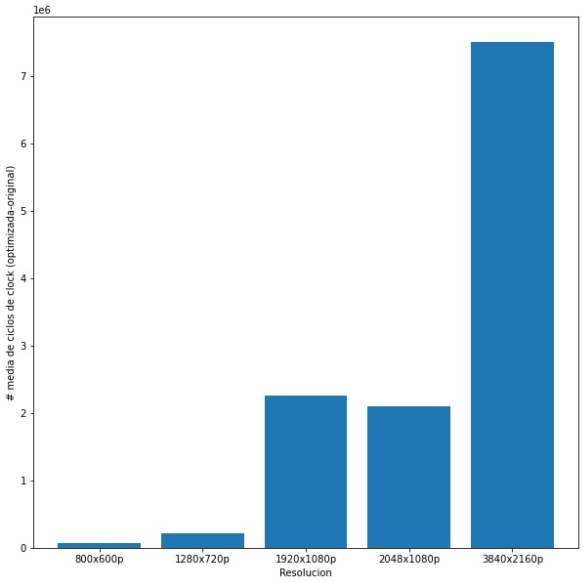

Figura 9: Mejora absoluta en el rendimiento promedio medido en ticks de reloj Experimento 1

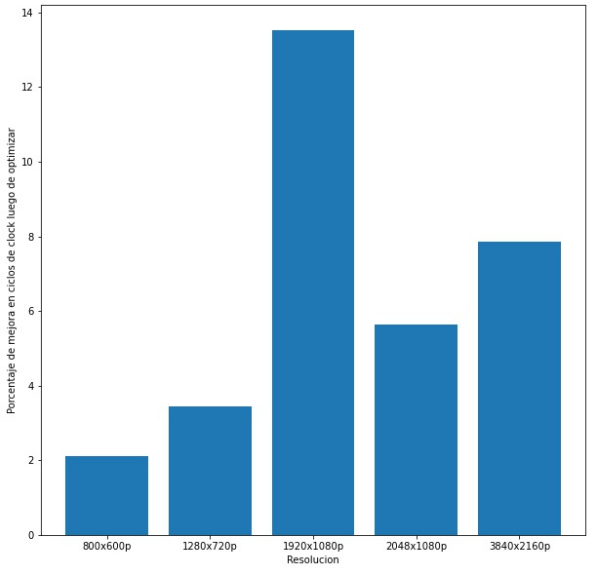

Figura 10: Mejora porcentual en el rendimiento promedio medido en ticks de reloj Experimento 1

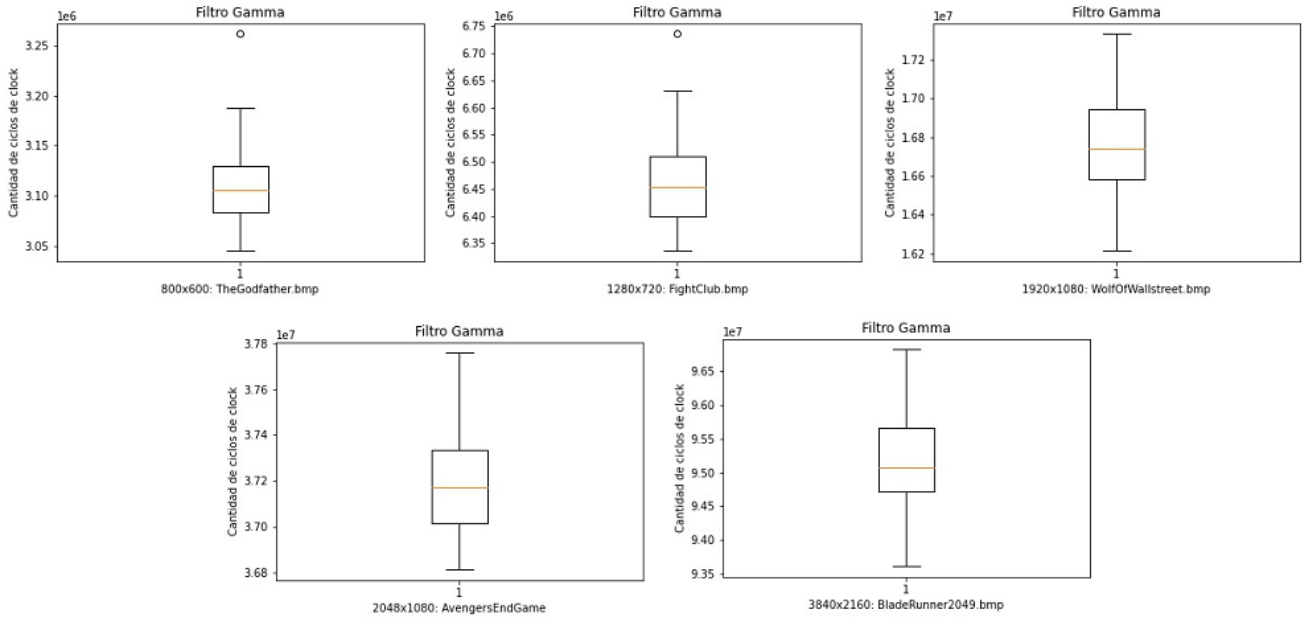

Figura 11: Muestras para la cantidad de ticks de reloj en imagenes de distintas resoluciones con el filtro Gamma original Experimento 3

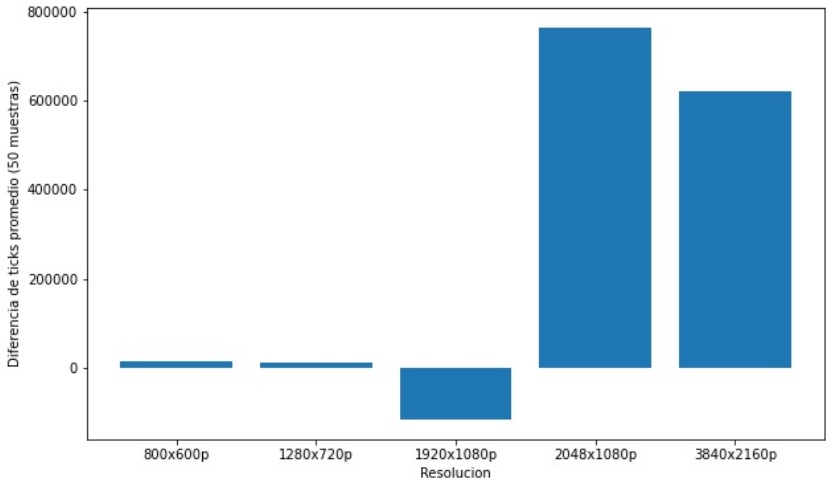

Figura 12: Mejora/Perdida absoluta en el rendimiento promedio medido en ticks de reloj Experimento 2

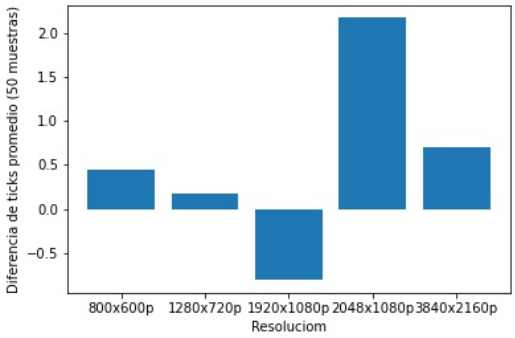

Figura 13: Mejora/Perdida porcentual en el rendimiento promedio medido en ticks de reloj Experimento 2

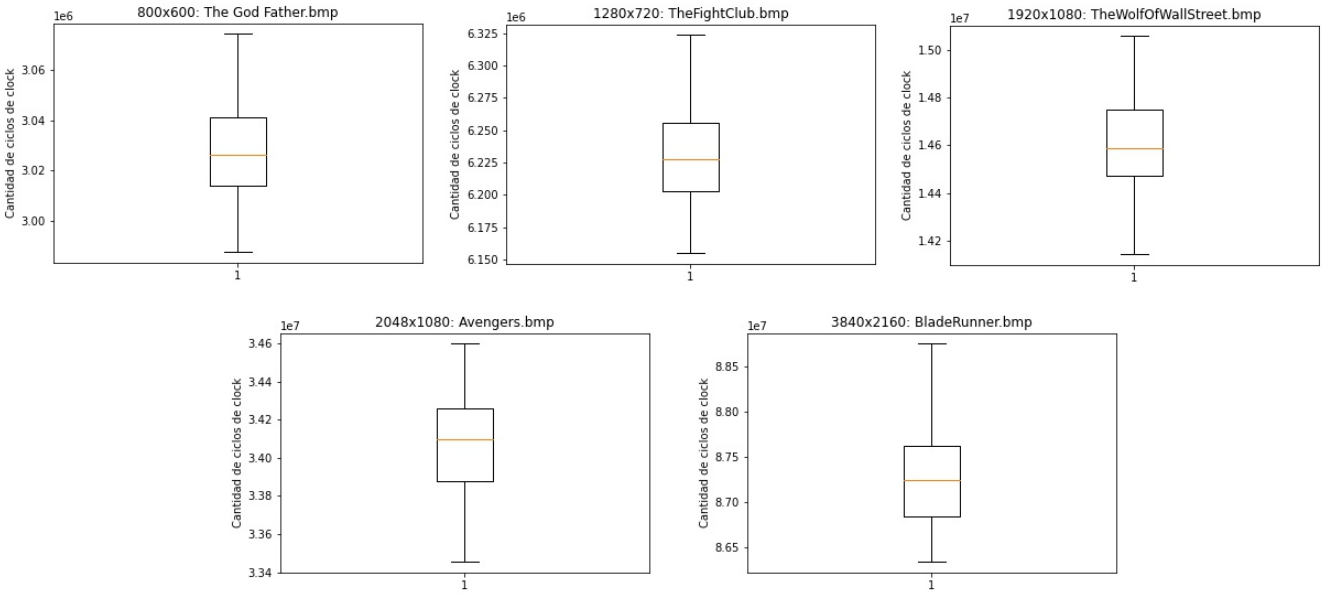

Figura 14: Muestras para la cantidad de ticks de reloj en imagenes de distintas resoluciones con el filtro Gamma original Experimento 3