DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

subtitulo del trabajo

Organización del Computador II

Segundo Cuatrimestre de 2014

Integrante	LU	Correo electrónico
Alejandro Mignanelli	609/11	minga_titere@hotmail.com
Franco Negri	893/13	franconegri2004@hotmail.com
Federico Suárez	610/11	elgeniofederico@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

Resumen

En el presente trabajo se describe la problemática de procesar información de manera eficiente cuando los mismos requieren:

1. Transferir grandes volumenes de datos.
2. Realizar las mismas instrucciones sobre un set de datos importante.

Índice

1. Objetivos generales

El objetivo de este Trabajo Práctico es mostrar las variaciones en la performance que suceden al utilizar instrucciones SIMD en comparación con código C con diversos grados de optimización realizados por el compilador cuando se manejan grandes volúmenes de datos que requieren un procesamiento similar.

Para ello se realizarán distintos experimentos sobre cuatro filtros de fotos, Cropflip, Bandas, Sierpinski y Motion Blur, tanto en código assembler, que aproveche las instrucciones SSE brindadas para los procesadores de arquitectura Intel, como en código C, al que se le aplicarán los distintos flags de optimización -O0 (predeterminado), -O1, -O2 y -O3.

El primer filtro, Cropflip, se utilizará para mostrar cuanto mejora la performance al utilizar los registros XMM para transferir grandes cantidades de información.

El segundo, tercer y cuarto filtro, se centrarán en la variación de performance (en comparación al código en C) al utilizar instrucciones SIMD, no sólo para transferir grandes volúmenes de datos sino también para procesarlos en forma paralela, es decir, realizar diversos cálculos (sumas, multiplicaciones, divisiones) tanto en representación de enteros como punto flotante.

2. Preambulo

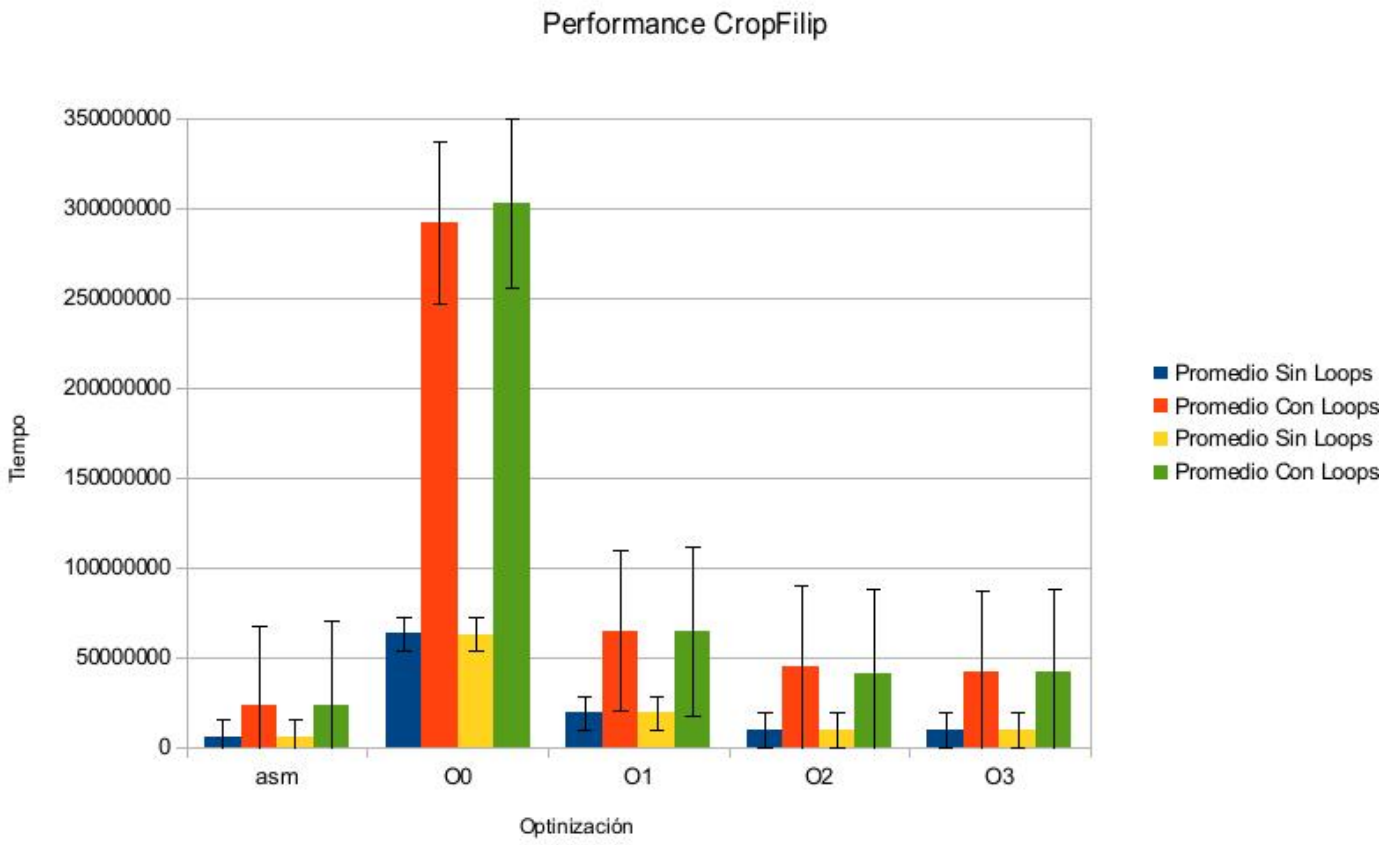
2.1. Calidad de las Mediciones

Para este experimento vamos a ver cómo se pueden ver afectados nuestros algoritmos frente a diversos factores de ruido e interferencias que podrían alterar nuestras mediciones.

Para este experimento se utilizó un procesador Intel Atom, de 2 núcleos a 1.6 GHZ con Hyper-Threading, por lo que la cantidad de núcleos lógicos asciende a 4. Por otro lado, para que las pruebas sean mas concisas y exactas, se deshabilitó el scaling dinamico del CPU, ya que esto podría generar ruido innecesario en nuestras mediciones.

Procedimos a tomar 1000 mediciones para cada una de las versiones del cropflip, tanto con 4 loops corriendo en paralelo como sin los mismos. La medida de tiempo será a partir de este momento y a lo largo de todo el informe, la cantidad de ciclos de clock del procesador. Lo que se obtiene es el siguiente gráfico:

Lo que se observa aquí es que al correr las pruebas con loops en segundo plano, la performance se ve afectada de forma negativa aumentando el tiempo de ejecución en por lo menos un 300 por ciento(como se ponia el simbolito?(ALE)). Con respecto a la varianza, se puede percibir fácilmente que el ruido la dispara considerablemente, es decir, aumenta abismalmente la dispersión de los datos, causando que estos sean menos fiables. Por los observado en este experimento, se ha tomada la decisión de que para cada experimento en el presente informe, se tomaran 1000 mediciones sin loops de fondo y quitando .(Es una buena justificacion porque hace que mis experimentos se vean mas chotos??(ALE))



3. Experimentación

3.1. Desensamblado de código C y Optimización

Comenzamos analizando el código de Cropflip realizado en C.

Este básicamente solo mueve datos de un lugar de la RAM a otros, sin afectar mayormente la imagen.

Realizamos un objdump para ver el código que genera el compilador gcc. Al desensamblar el código pudimos observar, primero que nada, que C guarda todos los parámetros en la pila y además está escribiendo en memoria todas las variables locales utilizadas, lo cual es innecesario ya que pueden ser almacenadas en registros.

También puede observarse que C utiliza saltos incondicionales, lo que puede sugerir que intenta sacar provecho al sistema de predicción de saltos.

Ademas C genera, luego de la función, un montón de secciones que comienzan con debug_xxx. Estas secciones sirven para ser interpretadas por GDB u otros debuggers.

Como ya dijimos, el código podría optimizarse para no realizar tantos accesos a memoria innecesarios guardando variables locales por ejemplo en registros, lo cual disminuiría el tiempo de ejecución.

Luego de esto, procedemos a compilar el código utilizando el flag -O1, y nuevamente realizamos un objdump para ver el código desensamblado. Se observa que ahora el mismo solo realiza los accesos a memoria mínimos indispensables, utilizando los registros para guardar los datos. Además el código es más compacto, y resulta mas claro de leer. Además precalcula los valores que serán utilizados muchas veces, lo que aumenta la performance, principalmente en casos de instancias grandes.

Los otros flags de optimización son -O2, -O3, -Og, -Os, -Ofast. También podemos encontrar los flags -msse, -msse2, -msse3, -mmmx, -m3dnow, pero al intentar compilar con varios de ellos vimos que gcc no utilizó instrucciones SIMD.

Tres nombres de optimizaciones son: -fipa-profile, -fipa-reference ,-fmerge-constants .

4. Cropflip

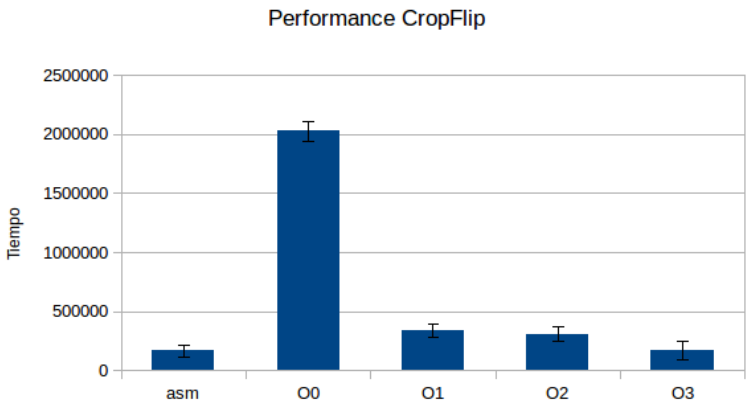
4.1. Diferencias de performance en Cropflip

En el siguiente experimento se medirán las performances tanto de nuestro algoritmo en assembler, implementado para sacar provecho de las instrucciones SSE de Intel, como una versión alternativa hecha en C con diversos grados de optimización a cargo del compilador.

El algoritmo de Cropflip en assembler es muy sencillo. Simplemente movemos 128-bits de la imagen a un xmm y de allí al destino, que previamente ha sido seteado para colocar los bits en el lugar correcto. De esta manera, podremos mover de una sola vez, 16 bytes, lo que corresponde a 4 píxels de la imagen.

Dado que la cantidad de columnas es siempre múltiplo de 4, o sea, siempre tenemos 4 bytes para tomar, no es necesario chequear otros casos borde.

Lo obtenido en los tests puede verse en el siguiente gráfico:



4.1.1. Resultados

Se percibe en el gráfico que la version SIMD del algoritmo de cropflip y la version C con mayor grado de optimización son muy parecidas en promedio, aunque la varianza del segundo es mayor la del primero. Por otro lado, el programa SIMD tiene una mayor performance a las optimizaciones O1 y O2, y la comparación con O0 muestra que en promedio, es más de 8 veces más rápido.

4.1.2. Conclusiones

Creemos que la gran similitud de tiempos entre el programa versión SIMD y versión C con O3 se debe a que es un código muy simple, el cual sólo mueve datos de un lugar a otro. Sin embargo, la diferencia en la dispersión de datos de ambas tienden a hacer pensar que la versión SIMD es un poco más confiable en términos de performance.

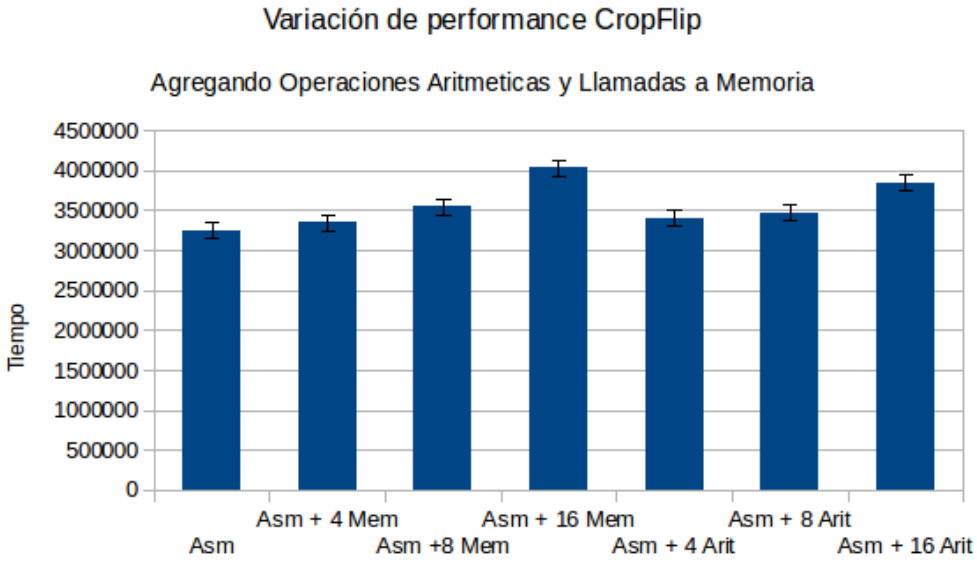
4.2. cpu vs. bus de memoria en Cropflip

Para este experimento lo que hicimos fue agregar instrucciones aritméticas y ver como esto influía en el tiempo de ejecución.

También hicimos lo mismo pero con instucciones de lectura-escritura en memoria.

Probamos agregando 4, 8 y 16 instrucciones de cada una por separado y comparamos las performance entre sí y con la versión original.

Lo que obtuvimos fue lo siguiente:



4.2.1. Resultados

4.2.2. Conclusiones

==conculsión!!!== (Lo que obtuvimos fue que en general, las operaciones de lectura producen una menor performance, mientras que en gral agregar operaciones aritmeticas mejora la performance. No se observa una dispersion de datos muy significativa(ALE)) (Conclusiones: No se me ocurre un motivo logico para las operaciones aritmeticas, las operaciones de memoria tienen mas sentido, el caso 8 dios sabe... la unica herramienta que conozco es la cache, y no se como usarla para un argumento a favor... recomiendo poner que no tenemos idea porque sucedio esto(ALE)) (ACAAAAAAAAAAAAAAAAAAAAAAA... dejo esto en blanco por si encontramos un mejor grafico, sino de ultima despues hago lo que puedo...)

5. Sierpinski

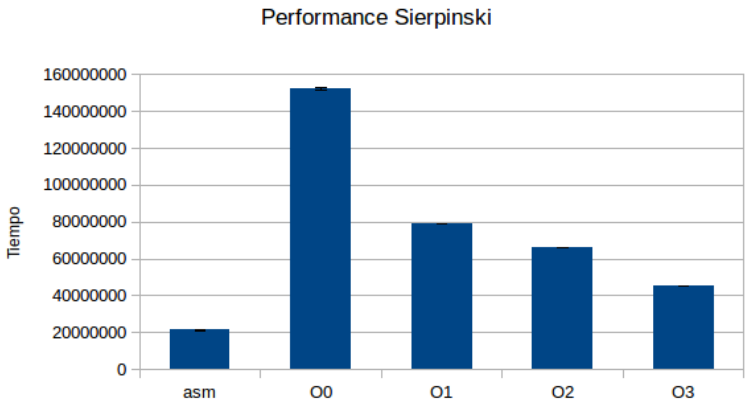
5.1. Idea general del algoritmo

Ahora analizamos el algoritmo del Sierpinski. En este caso, el algoritmo ya es un poco más complejo. Necesitamos calcular para cada píxel un coeficiente diferente, que dependerá de la posición (i,j) de cada píxel.
Luego para poder paralelizar de alguna manera el algoritmo en C y sacar provecho a los registros xmm, es necesario calcular 4 coeficientes a la vez y multiplicárselos a sus respectivos píxels.
Luego la idea del algoritmo será algo así:

- En la sección data guardamos una constante con el valor 255 brodcasteado en punto flotante.
- Previo al ciclo, guardamos este valor en un registro.
- Ya dentro del ciclo, leemos 4 píxels y los guardamos en un xmm.
- A continuación calculamos de manera paralela para cada píxel el coeficiente correspondiente.
- Primero, realizamos la división de i por la cantidad de filas y j por la cantidad de columnas, para la cual previamente pasamos de entero a punto flotante.
- Multiplicamos ambos valores por 255 y luego convertimos a entero con truncamiento.
- Realizamos un xor entre ambos y despues, volvemos a convertir a punto flotante para dividir por 255.
- Ya con los coeficientes calculados, solo nos resta multiplicar cada uno por el píxel correspondiente.
- Para ello desempaquetamos cada byte de cada píxel leido a word, y luego de word a dword, para así convertirlo a punto flotante.
- Una vez hecha la conversión, brodcasteamos cada coeficiente para poder multiplicarlo por el píxel correspondiente.
- Luego convertimos a entero nuevamente y empaquetamos todo de dword a word y de word a byte.
- Y finalmente movemos los píxels procesados al destino.

5.2. Diferencias de performance en Sierpinski

Los resultados comparativos de performance para este algoritmo comparado con uno desarrollado en C fueron los siguientes:



5.2.1. Resultados

Puede verse que en este caso nuestro modelo que toma y calcula de a 4 píxels utilizando instrucciones SIMD tiene una mejor performance que los demás. Es aproximadamente dos veces más rápido que la versión de C con O3, más de tres veces más rápido que O2 y O1 y más de 7 veces más veloz que O0, todo esto en promedio. Las varianzas son similares, y muy pequeñas.

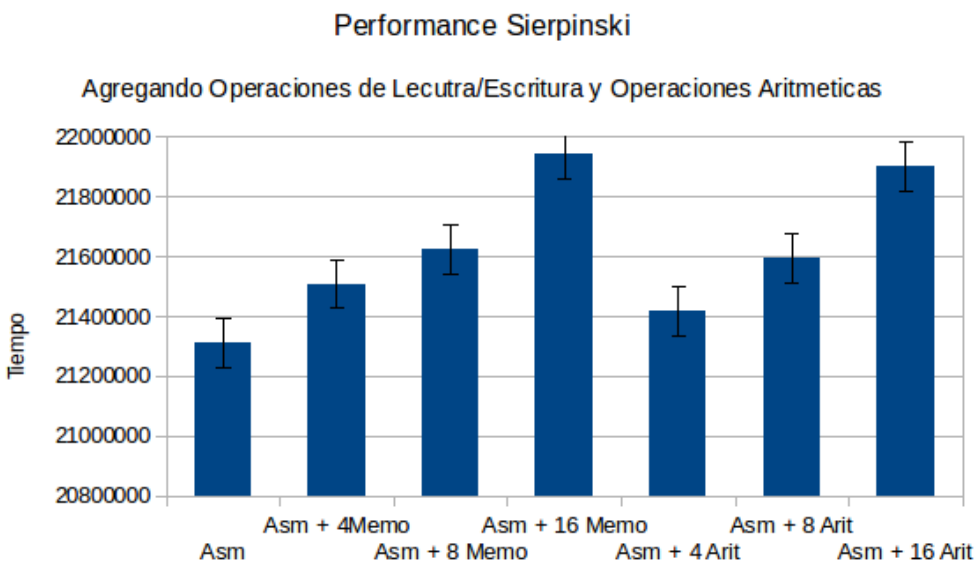
5.2.2. Conclusiones

En base a los resultados, parecería que nuestro programa es claramente una mejor opción ante las implementaciones de C.

5.3. CPU vs. Bus de memoria en Sierpinski

Realizamos el mismo experimento que hicimos para cropflip, es decir, agregando instrucciones aritmeticas y de lectura escritura y comparando las performances.

Graficando lo obtenido:



5.3.1. Resultados

Se observa que las operaciones aritmeticas y de memoria, provocan una disminucion de performance, y esta disminucion es mayor cuantas mas operaciones se agregan. Respecto a la varianza, estas son similares.

5.3.2. Conclusiones

Se puede ver que Sierpinski disminuye su performance de una manera similar cuando se le agregan más operaciones aritméticas o de lectura/escritura de memoria. Sin embargo, tomando como base lo que tarda el algoritmo por si mismo, pareciera que las operaciones de memoria lo afectan más en cuanto a tiempo que las operaciones aritmeticas. (CUIDADOOOOOOOOOOO. podría estar bueno rehacer este gráfico, sino tengo que poner chamuyo para el caso memoria = 8. el chamuyo sería el siguiente bicho, pero quisiera evitar ponerlo.”Se observa que al insertar ocho operaciones de memoria, el tiempo da un poco menor que en el caso de 4. Esto se lo atribuimos al ruido, puesto que al tener una diferencia de tiempos tan pequeña(menor al 0.1 por ciento), este factor puede tomar gran importancia”... o algo asi.)

6. Bandas

6.1. Idea general del algoritmo

Para el algoritmo de bandas se nos presenta otro desafío: debemos tomar los tres colores de la imagen(r,g,b), sumarlos, y luego comparar cada uno de ellos para ver si se encuentra en un rango determinado.
Para resolver la primera problemática usaremos las instrucciones $phaddw$, que nos permitirá a través de una suma horizontal, sumar los valores r,g,b de manera cómoda solamente utilizando dos registros.
El segundo problema será comparar estos valores obtenidos en la suma de una manera eficiente. Querríamos compararlos todos a la vez y a partir de esas comparaciones determinar que valores deberá ir en cada píxel. Esta se resolverá utilizando broadcasting. El algoritmo irá comparando en cada paso contra un valor y en caso de cumplirse una condición, restará donde corresponda.

- En la sección data tenemos mascaras y constantes brodcasteadas que utilizaremos para hacer las comparaciones.
- Ademas en esta sección tenemos una doble qword la cual usaremos para el shuffle final.
- Previo al ciclo donde procesamos los píxels, movemos estos datos a registros xmm.
- Ya una vez dentro del ciclo, leemos 4 píxels y los guardamos en un xmm.
- Desempaquetamos los bytes a word para poder hacer la suma de las componentes de cada píxel sin irnos de la representación.
- Hacemos dos sumas horizontales para obtener los cuatro valores de b.
- Luego utilizamos las mascaras para comparar en paralelo cada píxel con el b correspondiente y asignando el valor de rgb segun corresponda.
- Nos queda en cada word de la parte baja de un xmm, los 4 valores que se asignaran a cada rgb de cada píxel.
- Finalmente, empaquetamos estos valores para volver a byte y los shuffleamos para dejarlos en los lugares correspondiente y lo asignamos en el destino.

6.2. Diferencias de performance en Bandas

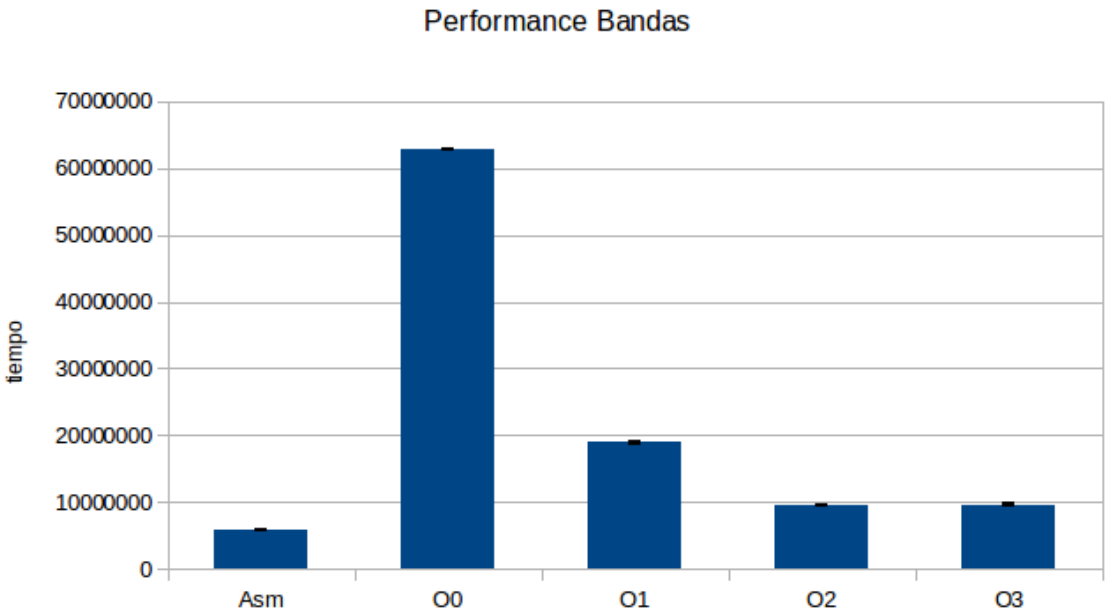
Los resultados de los tiempos comparativos son los siguientes:

6.2.1. Resultados

En el gráfico se aprecia que el programa de ASM con SIMD tarda, en promedio, un poco menos de la mitad de tiempo que la implementación de C con O3. Tambien se observa que O2 y O3 son muy similares. Se ve que SIMD es tiene una mayor performance que O1 (poco menos de 3 veces), y nuevamente, es varias veces más veloz en promedio que O0. En cuanto a las varianzas, son similares y muy pequeñas.

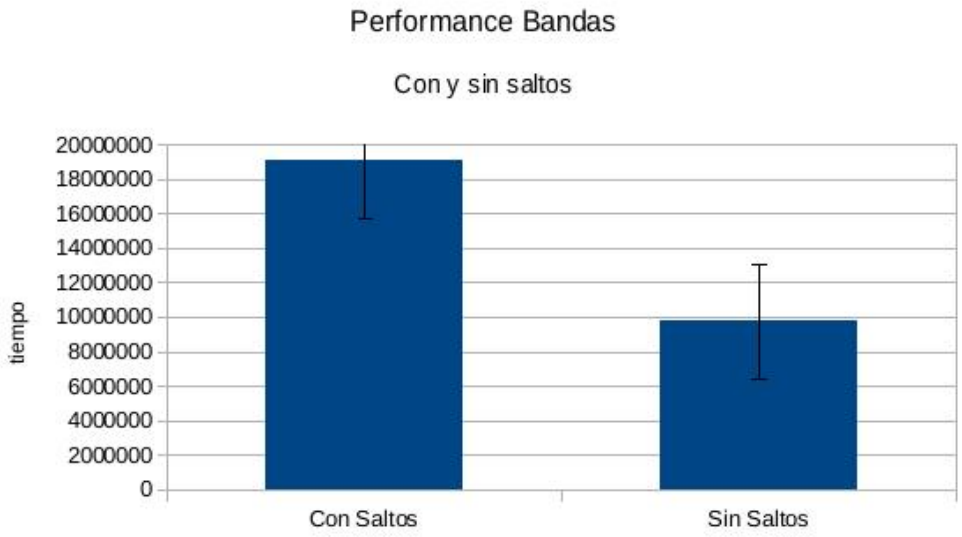
6.2.2. Conclusiones

Nuestro algoritmo obtiene una performance mucho mayor a la del código sin optimizar, y una performance un poco mejor al del código C con el mayor grado de optimización. Las diferencias no son tan claras como en Sierpinski, sin embargo, siguen siendo rotundas.



6.3. Saltos condicionales

En este experimento veremos como afectan los saltos condicionales a la performance del codigo C compilado con -O1. Para ello lo que haremos es quitar los IFs del codigo dejando solo una banda, luego medir la performance y compararla con la version con saltos.



6.3.1. Resultados

Como se puede ver en el gráfico del promedio, la versión de una sola banda tardó poco más de la mitad de tiempo con respecto a la versión original.(Esto esta hecho con 1000 mediciones=S?? (ALE))

6.3.2. Conclusiones

Al ver que la version de una sola banda tardo en promedio menos que la version original, creemos que esta disminución en el tiempo de ejecución tiene que ver con la predicción de saltos del procesador. Esto se debe a que cuando hay un salto condicional, en principio el procesador no sabe a qué posición de memoria deberá dirigirse. Al haber removido estos saltos, tiene sentido que la performance se haya visto impactada de manera positiva

6.4. Motion Blur

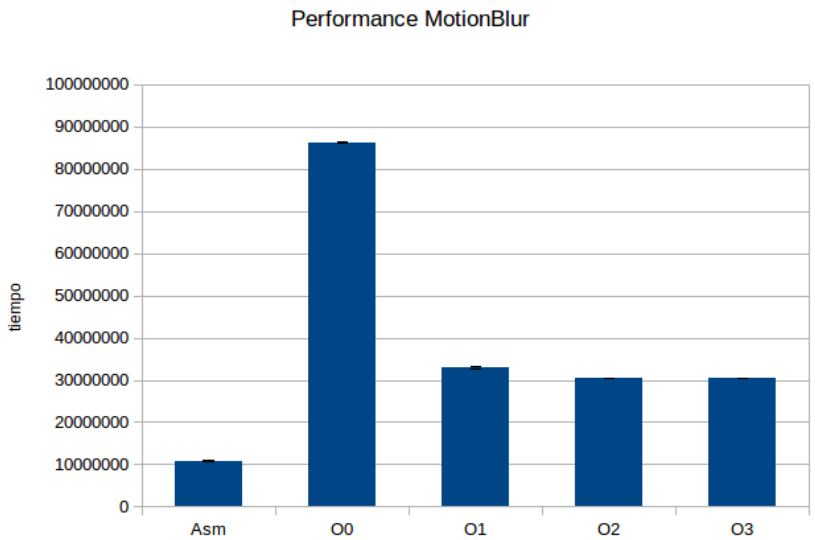
6.5. Idea general del algoritmo

Para este algoritmo por cada píxel se deben tomar 5 píxels, multiplicar cada uno de los colores por $0{,}2$ y luego sumarlos.
Para ello debemos tener cuidado de tomar correctamente los casos borde y no aplicar motion blur donde no corresponde.
Una idea general de cómo funciona el algoritmo en assembler es la siguiente:

- Tenemos, en la sección datos, el valor $0{,}2$ brodcasteado que utilizaremos para realizar la multiplicación.
- Previo al procesamiento de datos, guardamos en un xmm este valor.
- Dentro del ciclo, para procesar el píxel (i,j) tomamos los 4 píxels de las filas $i-2$, $i-1$, i, $i+1$ e $i+2$ a partir de la columna $j-2$, $j-1$, j, $j+1$ y $j+2$, respectivamente, y almacenamos todo en registros.
- Desempaquetamos todo a word para poder hacer la suma sin desbordar.
- Luego de realizar las sumas, desempaquetamos a dwords para así poder convertir a punto flotante.
- Con la conversión ya hecha, podemos multiplicar cada valor por $0{,}2$.
- Convertimos a entero nuevamente y empaquetamos de dword a word y de word a byte.
- Finalmente, copiamos los 4 píxels resultantes al destino.
- Cada vez que termino de recorrer una fila, pongo en negro los siguientes 4 píxels, y las dos primeras y últimas filas las pongo en negro aparte, fuera del ciclo.

6.6. Diferencias de performance en Motion Blur

Al realizar el testing se obtuvieron los siguientes resultados:



6.6.1. Resultados

Se aprecia en el gráfico que el código ASM con SIMD tiene un promedio de tiempo de ejecucion aproximadamente tres veces menor a la mayor optimización de C, y varias veces menor que O0. También se observa que O1, como O2 y O3 dan promedios de tiempo muy parecidos. Las varianzas son similares y muy pequeñas.

6.6.2. Conclusiones

En este caso nuestro algoritmo supera ampliamente incluso al código C con mayor grado de optimización, lo que hace pensar que es una mejor opción entre las propuestas.

7. Conclusiones y trabajo futuro

SIMD es una herramienta muy útil para mejorar la performance a la hora de manejar grandes cantidades de datos que requieren un mismo procesamiento, como imágenes o videos, y esto se vio reflejado ampliamente en los resultados de los experimentos realizados en este trabajo. Sin embargo si lo único que necesitamos realizar es un mero movimiento de datos, las diferencias se vuelven imperceptibles con respecto al grado de mayor optimización de C. Sin embargo, un gran poder conlleva una gran responsabilidad y un error de empaquetado o desempaquetado puede resultar en muchas horas de debugging, como por ejemplo desempaquetar dos veces la parte alta de un registro, en vez de desempaquetar la parte baja también.

(ACLARACIONES GENEALES:4.2 cropflip rehacer el grafico hasta que de algo coherente, fijarse lo que dijo el chabon de la practica de ver que realmente este yendo a 1.6GHZ, en 5.3 sierpinski, solo si hay tiempo volver a hacer el grafico, asi no tenemos que tirarle culpas al ruido por las 8 operaciones de memoria, explicar los algoritmos, y el tp creo que ya estaria listo..)