

Experimento 1.1 - análisis del código generado

- a) Porque en el MakeFile se usa -g (incluye en el ejecutable generado la información necesaria para poder rastrear los errores usando un depurador, tal como GDB) y esto hace que se muestren funciones de debugg.
- b) Las variables locales se manipulan con la pila, por ejemplo usa indices para recorrer la matriz, entonces tiene que estar actualizando por la iteración por la que va en la pila.

Esto ocupa muchas líneas de código, lugar en la pila(ya que no es necesario usarla siendo que existe los registros de propósito general) y es lento.

- c) En lugar de utilizar la pila para manejar todas las variables locales, se podrían usar registros(ahormando muchas accesos a memoria que son "caros").

Además guarda datos en la pila que nunca vuelve a usar, usa add para sumar 1 a variables en la pila en vez de usar inc y en vez de usar lea usa imul. También hace swaps de variables para poner el mismo valor que tenían.

Experimento 1.2 - optimizaciones del compilador

- a) Utiliza los registros para manejar las variables locales , saca las que están en la pila a un registro. Como no tiene que estar actualizando la pila todo el tiempo hay menos líneas de código y el espacio en la pila no se desperdicia. Ahora al no usar la pila para manejar las variables locales no guarda cosas innecesarias que nunca vuelve a usar.

Además usa lea en vez de imul.

- b) Además del O1 que es el flag que utilizamos para este experimento, existen otros como O0(reduce el tamaño del código y tiempo de ejecución), O2 ,O3 y ofast entre otros.

El flag O0 reduce el tiempo de compilación. El flag -O1 optimiza en espacio, si hay optimizaciones que aumentan el tamaño tambien las va a omitir.

El flag -O2 optimiza en velocidad se reemplazan instrucciones que demoran mas por otras que no lo hacen.

El flag -O3 utiliza vectorización entre otras cosas.

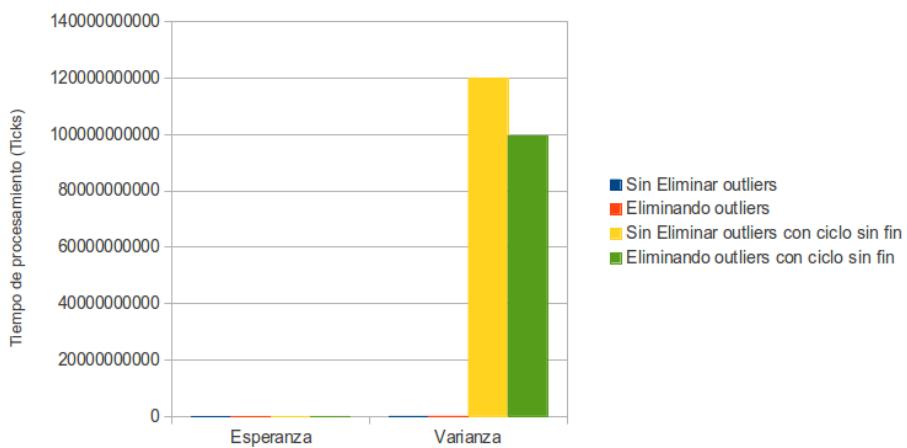
- c) Paralelización automática, eliminación de código muerto y Evaluación perezosa.

Experimento 1.3 - calidad de las mediciones

e) Todos los datos obtenidos son corriendo cropflip(./tp2 cropflip -i c lena.bmp 400 400 0 0) versión asm y c respectivamente, obteniendo la esperanza de correr 500 veces el algoritmo, luego obtengo la esperanza y varianza de hacer esto 10 veces.

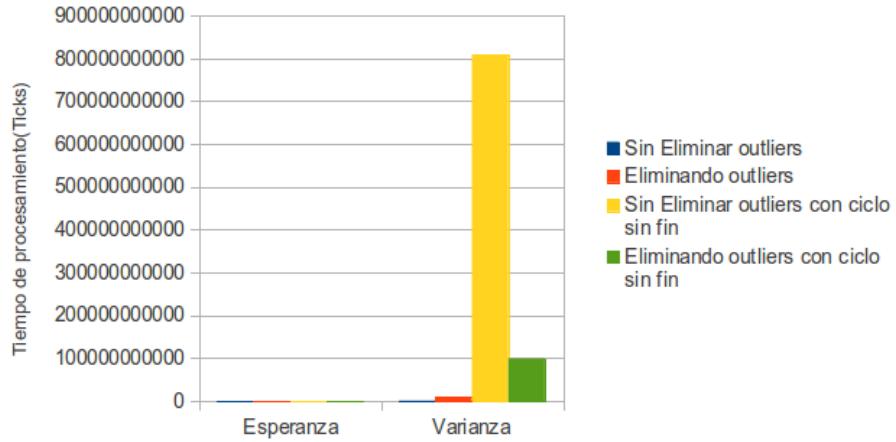
Cuando menciono ciclo sin fin es que mientras corro el algoritmo de cropflip de la manera mencionada arriba, corro también en 4 consolas(una por core lógico) un programa que cicla infinitamente sumando 1 a una variable.

En ASM:



	Sin Eliminar outliers	Eliminando outliers	Sin Eliminar outliers con ciclo sin fin	Eliminando outliers con ciclo sin fin
Esperanza	1075034	1082794	2251986	1327706
Varianza	34930897	57527160	119775392066	99353006390
Desvío estándar	5910	7584	346085	315203

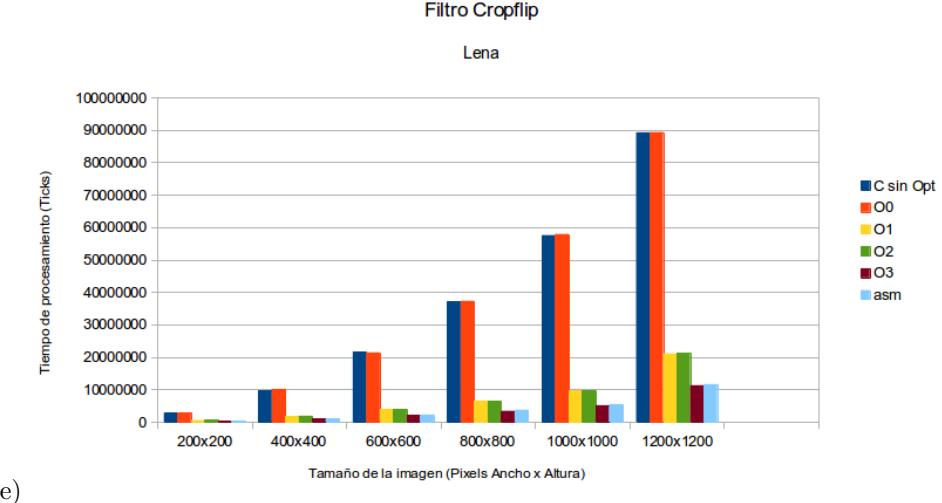
En C:



Como podemos observar cuando corremos el ciclo sin fin la varianza aumenta, y cuando sacamos los outliers la varianza disminuye.

Esto se debe a que los resultados que obtenemos cuando corremos el ciclo sin fin tienen "picos altos y bajos muy marcados" y al eliminarlos la varianza baja notablemente.

Experimento 1.4 - secuencial vs. vectorial



Una tabla con la esperanza:

	C sin Opt	O0	O1	O2	O3	asm
200x200	2729497	2764889	527390	535989	306775	327764
400x400	9728805	10113148	1728549	1833756	1012518	1079558
600x600	21530442	21363675	3874989	3837229	2047039	2212619
800x800	37178520	37283377	6460623	6397630	3348215	3695146
1000x1000	57520696	57853210	9676018	9755479	5081901	5444086
1200x1200	89234413	89309913	21029744	21239865	11255414	11581731

Una tabla con el desvío estándar:

	C sin Opt	O0	O1	O2	O3	asm
200x200	2729497	2764889	527390	535989	306775	327764
400x400	9728805	10113148	1728549	1833756	1012518	1079558
600x600	21530442	21363675	3874989	3837229	2047039	2212619
800x800	37178520	37283377	6460623	6397630	3348215	3695146
1000x1000	57520696	57853210	9676018	9755479	5081901	5444086
1200x1200	89234413	89309913	21029744	21239865	11255414	11581731

Intuitivamente creíamos que ninguna optimización podría ganarle a la implementación de asm porque esta utiliza operaciones SIMD que permiten manejar más cantidad de datos en 1 operación, pero O3 le gana. Esto es porque utiliza vectorización, y operaciones que tienen menor "costo" que las que usamos nosotros en asm. La optimización O3 es lo suficientemente "inteligente" como para traducir el código c a asm usando SIMD, ganandole en tiempo de ejecución a nuestra implementación en asm.

De todos modos la diferencia en performance es mínima.

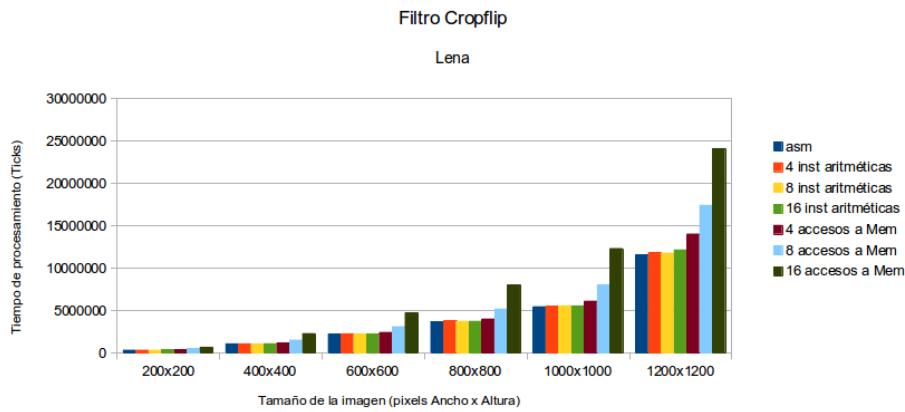
Se puede observar a simple vista que la implementación de c sin optimizaciones (o con O0) tiene una performance muy mala en comparación con la de asm y las optimizadas. Como mencionamos en el experimento 1.1, el flag O1 optimiza el espacio, entonces la performance no se ve afectada prácticamente. El flag O2 cambia instrucciones lentas por otras más rápidas, en este caso se ve que el tiempo de ejecución se redujo a más de la mitad al igual que con el flag O2. Con el flag no hay diferencia porque solo acota el tiempo de compilación. Y finalmente el flag O3 reduce drásticamente el tiempo de ejecución.

Experimento 1.5 - cpu vs. bus de memoria

Intuitivamente los accesos a memoria son mas caros que las operaciones lógicas y las aritméticas, veamos que sucede con los gráficos.

Usando add, subb, shl shr con rax (porque no está ocupado por el código) para las instrucciones aritméticas/ lógicas y lecturas y escrituras en las matrices de entrada y salida.

Usando la pila también cumplía el objetivo de agregar "instrucciones de memoria" pero accediendo a disco es mucho más evidente el costo.



Una tabla con la esperanza:

	asm	4 inst aritméticas	8 inst aritméticas	16 inst aritméticas	4 accesos a Mem	8 accesos a Mem	16 accesos a Mem
200x200	327764	330070	325948	335776	361649	483753	677100
400x400	1079558	1077914	1066385	1087797	1185821	1511840	2279261
600x600	2212619	2261918	2249851	2247825	2421774	3081895	4714969
800x800	3695146	3768815	3723970	3730384	4001586	5168873	7990616
1000x1000	5444086	5482981	5553800	5549877	6098126	7977252	12300114
1200x1200	11581731	11834767	11756569	12144704	14008095	17400363	24084017

Una tabla con el desvío estándar:

	asm	4 inst aritméticas	8 inst aritméticas	16 inst aritméticas	4 accesos a Mem	8 accesos a Mem	16 accesos a Mem
200x200	1716	1449	2289	1003	2264	3551	3996
400x400	2394	3076	3292	4216	7457	10108	12371
600x600	6508	33877	17632	4527	12577	21001	17786
800x800	17051	26625	6426	4691	22502	23918	37811
1000x1000	9941	16598	6888	10155	116380	28288	34784
1200x1200	37844	122174	31897	45005	160579	40937	95805

Como vemos en el gráfico cuando aumentamos la cantidad de accesos a memoria, el tiempo de ejecución se ve afectado negativamente, es mucho más lento. En cambio cuando agregamos instrucciones aritméticas el cambio en performance es mínimo. Con lo cual podemos concluir que los accesos a memoria son el factor que limita la performance.

informe de cropflip en ASM

En rdi tengo el puntero a la matriz de entrada y en rsi tengo el puntero a la matriz de salida.

Primero vamos a poner src (rdi) en la fila offsety ([rbp +40]).

Para esto vamos a hacer un ciclo que haga tantas iteraciones como el valor de offsety.

En cada paso del ciclo vamos a sumar src con src_row_size(r8d) para poder mover el puntero a la siguiente fila.

Luego vamos a poner dst(rsi) en la última fila. Para esto vamos a hacer un ciclo de tamy-1 iteraciones. En cada iteración vamos a sumar a rsi con dst_row_size para poder mover el puntero a la siguiente fila.

Como tamx es la cantidad de pixels, quiero que tamx sea la cantidad de bits, con lo cual, multiplico tamx * 4(cantidad de bits de un pixel)

Como offsetx es la cantidad de pixels, quiero que offsetx sea la cantidad de bit, entonces voy a multiplicar a offsetx por 4.

Voy a iterar las filas de la matriz de entrada, cuya cantidad es tamy.

Ahora voy a hacer un ciclo por cada fila de la matriz de entrada (situada al principio en offsety) hasta tamx.

Tomo 4 pixels (16 bits) de la posición de memoria rdi+offsetx en un registro xmm y los voy a poner en la posición de memoria rsi.

Avanzo 4 pixels el puntero de rdi y el de rsi.

Cuando termino una fila, muevo el puntero que está en rdi a la siguiente fila en la matriz de entrada (sumando src_row_size a rdi) y subo una fila en la matriz de salida (restando dst_row_size a rsi).

informe de cropflip en C

Realizamos 2 fors anidados con indices i y j, filas y columnas respectivamente. Con $i \in \{0, tamy-1\}$ y $k \in \{0, (tamx*4)-1\}$

Voy a hacer un pasaje de a un pixel. En la matriz de salida en la fila i columna j voy a poner el pixel de la matriz de entrada en la fila tamy + offsety -i -1, columna offsetx*4 +j.

Conclusiones

Podemos concluir varias cosas, primero escribir código en C es cómodo e intuitivo. En cambio, escribirlo en ASM no es nada trivial, encontrar errores es difícil, hay que tener conciencia de como se maneja lo que estamos leyendo en memoria y donde se escribe.

Sin embargo se puede ver claramente en las experimentaciones que el tiempo de ejecución de las implementaciones en asm es mucho mejor que las de c. Lo cual era esperable dado que procesamos muchos datos a la vez y solo hacemos accesos a memoria dentro de los ciclos para cargar los pixels de la imagen fuente y para escribirlos luego de procesarlos.

En cuanto al compilador se puede observar que cuando no tiene optimizaciones los algoritmos corren lento(en comparación con asm), y cuando se agregan los flags de optimizaciones los algoritmos corren a una diferencia más que notable.

Incluso en cropflip el flag O3 supera mínimamente en performance a la implementación en asm, lo que nos llamó mucho la atención.