DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

TP2

Organización del computador II

Pencylvester

Integrante	LU	Correo electrónico
Alonso Tomás	396/16	tomasalonso96@gmail.com
Gaggero Damián	318/16	damiangaggero@gmail.com
Grosso Alejandro	016/16	alejandrogrosso@opmbx.org

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 trabajo realizado implementa la simulacion de flujo de fluidos basada en las ecuaciones de Navier-Stokes, la eleccion de utilizar estas ecuaciones es porque se consigue un resultado que no es numericamente preciso pero es lo suficientemente estable, simple y fiel a la realidad como para dejar ver lo que este trabajo intenta observar.

El objetivo final es poder observar las diferencias de rendimiento de programas en codigo en C comparados con la misma logica pero implementada en Assembler utilizando el set de instrucciones SSE propio de los procesadores intel.

En principio se explican las decisiones de implementacion del codigo, ya sean de comodidad y simpleza o por mejoria de rendimiento, asi como tambien se habla sobre intentos fallidos por intentar mejorar la performance al utilizar extensivamente la metodologia SIMD, pero encontrandonos con inconvenientes inesperados. Tambien se puede observar que es lo que efectivamente realiza el codigo y como esto influyó en las decisiones tomadas.

Una vez en claro como funciona el programa y porque se decidio hacerlo asi, se muestra la experimentacion exhaustiva que se hizo para medir las diferencias de tiempos entre todo el trabajo con las funciones en C en comparacion con todas las funciones nuevas en assembler, esto en relacion con el rendimiento comparado entre cada funcion en C con su contraparte en assembler. Con todo esto se puede ver claramente que implementacion es mas rapida asi como distinguir que funciones son las que hacen la mayor diferencia en general.

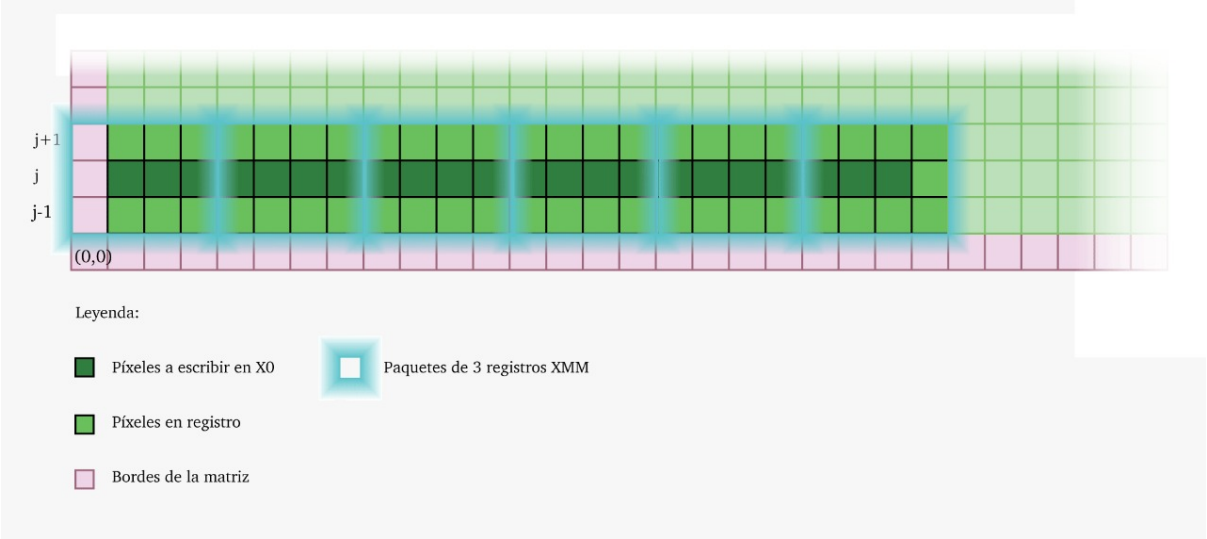
2. Desarrollo

2.1. solver_lin_solve

Como primer enfoque para la implementación de *solver_lin_solve* se pensó en leer de a columnas y maximizar la cantidad de posiciones que calculábamos a la vez, tal como puede apreciarse en la *figura 1*.

La principal ventaja de leer de a columnas, es reusar la fila calculada con anterioridad, evitándose así una relectura a memoria.

Luego, para maximizar la cantidad de posiciones, se usaban los 16 registros *xmm*. Dando por resultado la lectura de segmentos de matriz de 3×24 posiciones simultáneas y el calculo de 22 posiciones nuevas por iteración del ciclo. En cada iteración se leían 24 posiciones nuevas de la siguiente fila (siempre manteniéndose en la misma columna de 24 elementos). Una vez que ya no era posible de a 24, se terminaba de iterar la matriz calculando de a 2 posiciones.



Dicha implementación no fue completamente terminada, pero primeras pruebas de la misma mostraron un rendimiento muy por debajo de lo esperado, dando solo una leve mejora a la versión C provista por la cátedra (ver más en resultados (sección piripitruli)).

Por este motivo fue descartada (código en *solver_lin_solve_largo.asm* [1]).

Entre las hipótesis acerca de la merma de rendimiento pensamos en la extensa longitud del código de los ciclos y la forma de lectura por columnas de la matriz que no aprovecha el modo de almacenamiento por filas [2].

Como opción final, se optó leer por medio de iteradores (puntero a la matriz) que recorran la matriz por filas y calculen de a 2 posiciones a la vez, (i, j) e $(i + 1, j)$.

Usándose la estrategia detallada a continuación para realizar el siguiente calculo en paralelo, para cada posición de la matriz:

$$x(i,j) = \frac{x_0(i,j) + a * \big(x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1)\big)}{c}$$

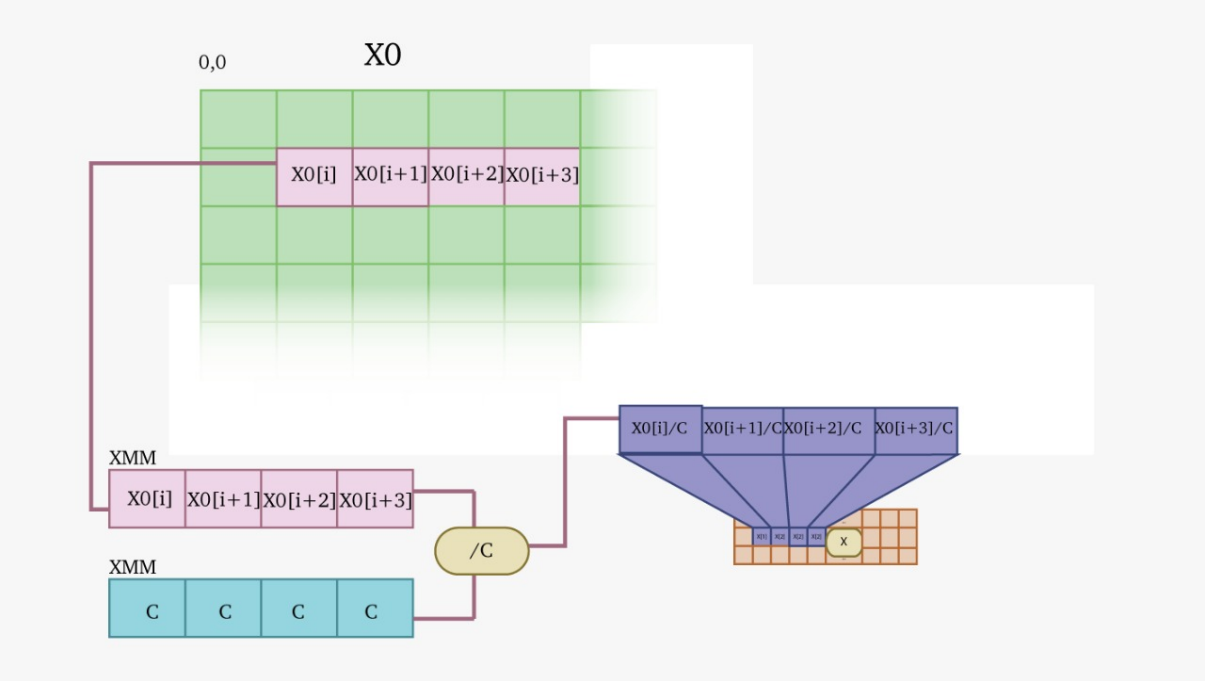
Primero notamos que si separamos en casos $a = 0$ y $a \neq 0$, nos permite que en el caso $a = 0$, el cálculo se reduzca a:

$$x(i,j) = \frac{x_0(i,j)}{c}$$

[1] El código no presenta errores de **segmentation fault** y puede ser ejecutado. Sin embargo, al ser un código **no** finalizado, produce errores en el cálculo de fluidos.

[2] #define IX(i,j) (i+(solver->N+2)*j)

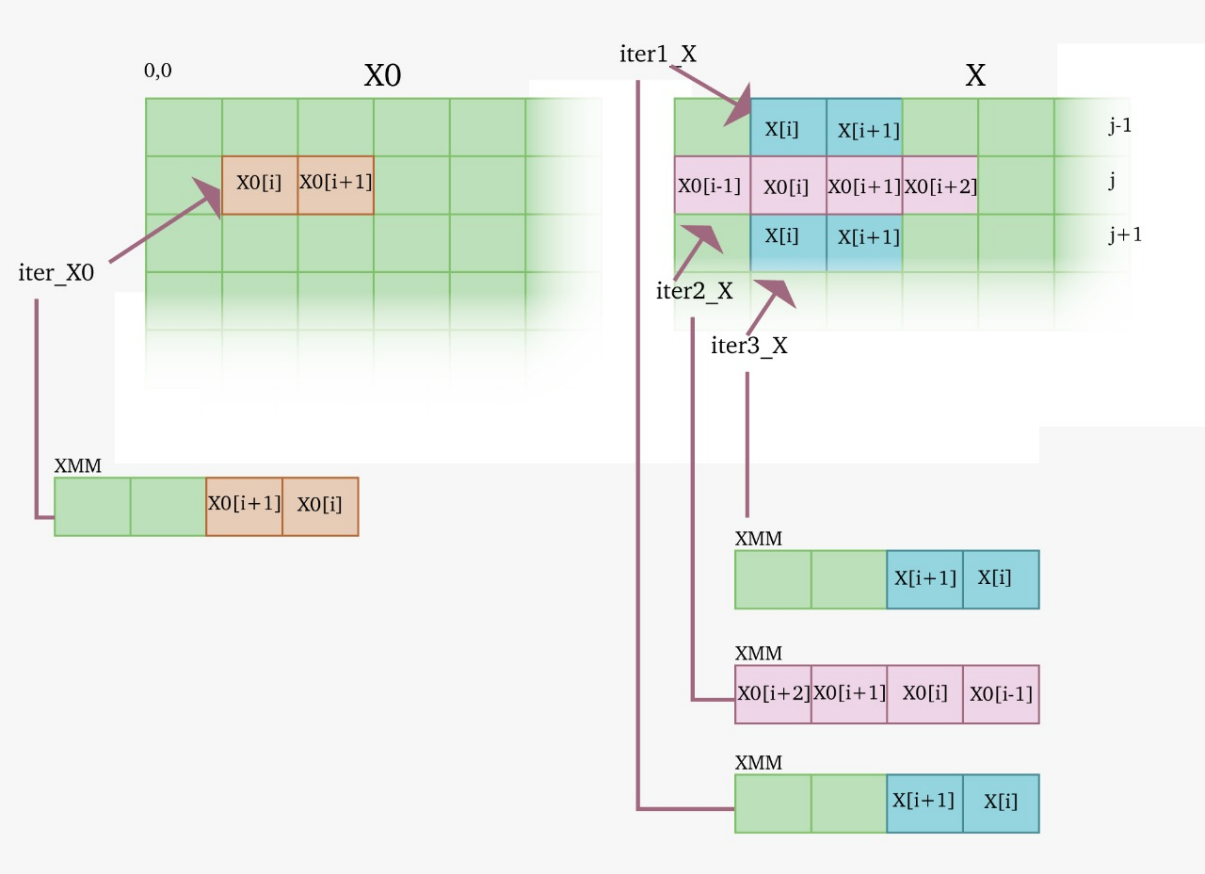
Que podemos realizarlo de a 4 posiciones en paralelo por iteración de ciclo, por medio de un iterador de x_0 y un iterador de x, en contraposición a calcular de a 2 posiciones como realizamos en el caso general ($a \neq 0$), que detallaremos más adelante. Otra ventaja es que el cálculo se reduce a una sola instrucción (dividir por *c*).



x0 –>(carga en xmm) | x0[i][j] | x0[i+1][j] | x0[i+2][j] | x0[i+3][j] | –>(div por c) | x0[i][j]/c | x0[i+1][j]/c | x0[i+2][j]/c | x0[i+3][j]/c | –>(carga en memoria) x

En el caso $a \neq 0$, se utilizan 4 iteradores que apuntan respectivamente

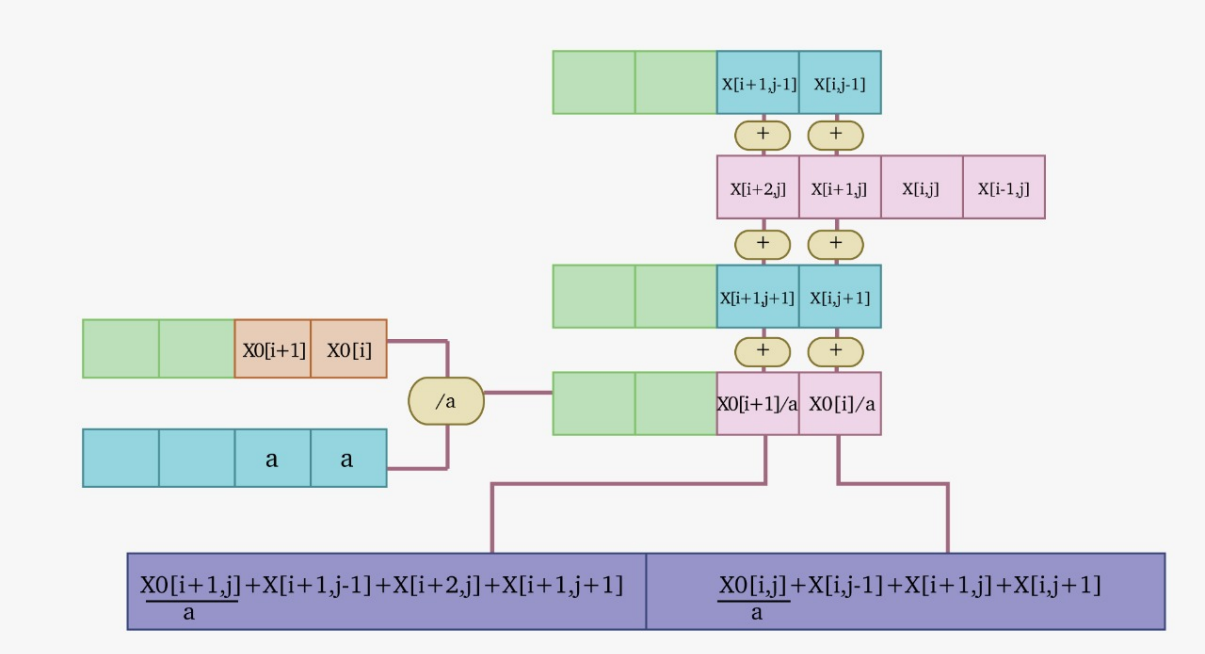
En cada iteración, se calculan las posiciones $x(i, j)$ y $x(i + 1, j)$. Se leen 2 posiciones a partir de *iter1_x*, 2 de *iter3_x*, 2 de *iter_x0* y 4 de *iter2_x*, dando por resultado la carga en registros *xmm* de las siguiente posiciones.



Consecuentemente, se calculan en paralelo, es decir de a 2 posiciones simultáneas, la suma

$$x(i,j) := \frac{x_0(i,j)}{a} + x(i+1,j) + x(i,j-1) + x(i,j+1)$$
$$x(i+1,j) := \frac{x_0(i,j)}{a} + x(i+1,j) + x(i,j-1) + x(i,j+1)$$

para $x(i,j)$ y $x(i+1,j)$ como muestra la figura.



Luego, se calcula secuencialmente

$$x(i,j) + x(i-1,j)$$

$$x(i,j)/c$$

$$x(i+1,j)+(i,j)$$

$$x(i+1,j)/c$$

FIGURA (diagrama que va mostrando los pasos del algoritmo)

Esto es necesario, debido a que cada posición necesita que la casilla a su izquierda $x(i-1,j)$ haya sido procesada con anterioridad al sumarse a $x(i,j)$.

Vale la pena aclarar que la casilla $x(i,j-1)$ también debe haber sido procesada al momento de su suma, pero debido a la manera en que se recorre la matriz, al leerse la posición $x(i,j-1)$, ya fue calculada con anterioridad.

Por último, se incrementan en dos posiciones los iteradores y se sigue con una nueva iteración del ciclo hasta completar la matriz.

Esta solución, en principio menos óptima ya que debía leerse 3 veces cada posición de memoria (ver figura tanto), aprovecha mucho mejor el principio de localidad en caché, tanto para el código (al ser mucho más compacto), como para los datos (al recorrer la matriz por filas).

Quedó en el tintero aclarar el por qué las posiciones de memoria se leen 3 veces. Esto ocurre porque si en una iteración se leen las posiciones de $x\ \big[(i,j)|(i+1,j)|(i+2,j)|(i+3,j)\big]$, en la siguiente se leen $\big[(i+2,j)|(i+3,j)|(i+4,j)|(i+5,j)\big]$, releyéndose las posiciones $\big[(i+2,j)|(i+3,j)\big]$. Luego las posiciones $\big[(i+2,j-1)|(i+3,j-1)\big]$ también fueron calculadas anteriormente, leyéndose por tercera vez.

FIGURA |x(i,j)|x(i+1,j) |x(i+2,j) |x(i+3,j) |x(i+4,j)|x(i+5,j)| |x(i+2,j-1)|x(i+3,j-1)| (poner i+2 e i+3 de otro color que resalte)

Esta desventaja fue superada guardando $x(i+2,j)$ y $x(i+3,j)$ de la iteración anterior en la parte baja del registro y cargando $x(i+4,j)$ e $x(i+5,j)$ en la parte alta del registro con *movhps*[3] sin ninguna modificación extra al algoritmo utilizado.

2.2. solver_set_bnd

Para la función *solver_set_bnd* se desglosó el ciclo en casos $b=1$, $b=2$ y $b\neq 1 \wedge b\neq 2$.

Al hacerlo, se evita realizar comparaciones dentro del ciclo. Quedando por resultado un ciclo que por cada iteración calcula lo siguiente:

$b=1$	$b=2$	$b\neq 1 \wedge b\neq 2$
$x(0,i) := -x(1,i)$	$x(0,i) := x(1,i)$	$x(0,i) := x(1,i)$
$x(n+1,i) := -x(n,i)$	$x(n+1,i) := x(n,i)$	$x(n+1,i) := x(n,i)$
$x(i,0) := x(i,1)$	$x(i,0) := -x(i,1)$	$x(i,0) := x(i,1)$
$x(i,n+1) := x(i,n)$	$x(i,n+1) := -x(i,n)$	$x(i,n+1) := x(i,n)$

Se usan 4 iteradores y 1 ciclo que itera n veces, calculándose por cada iteración una nueva posición. Estos iteradores son *upper*, *bottom*, *left* y *right_n*. Y recorren respectivamente,

FIGURA (cuadrito con bordes y nombres del iterador en cada uno)

Por último se realizan los últimos calculos de la misma forma que el código original

[3] movhps: carga las posiciones 2 y 3 del registro $xmm = [3|2|1|0]$

$$x(0,0) := 0{,}5 \cdot \big(x(1,0) + x(0,1)\big)$$
$$x(0,n+1) := 0{,}5 \cdot \big(x(1,n+1) + x(0,n)\big)$$
$$x(n+1,0) := 0{,}5 \cdot \big(x(n,0) + x(n+1,1)\big)$$
$$x(n+1,n+1) := 0{,}5 \cdot \big(x(n,n+1) + x(n+1,n)\big)$$

Notar que se usa un iterador *upper_n* ubicado en la fila n en vez de la $n+1$, esto es debido al formato en que pueden indexarse las direcciones en *intel* donde solo se pueden sumar registros y no restarse. En el caso de haberse optado por un iterador de la fila $n+1$, la siguientes líneas [4]

```
mov aux_edx, [upper_n]              ; aux := x[i][N]
mov [upper_n + lon_fila], aux_edx   ; x[i][N+1] := x[i][N]
```

deberían ser idealmente

```
mov aux_edx, [upper - lon_fila]     ; aux := x[i][N]
mov [upper], aux_edx                ; x[i][N+1] := x[i][N]
```

Como la operación no está permitida, la resta debería realizarse anteriormente en un registro auxiliar. Para evitar esto y aprovechar el modo de direccionamiento de *intel*, usamos un iterador a la posición n.

2.3. solver_project

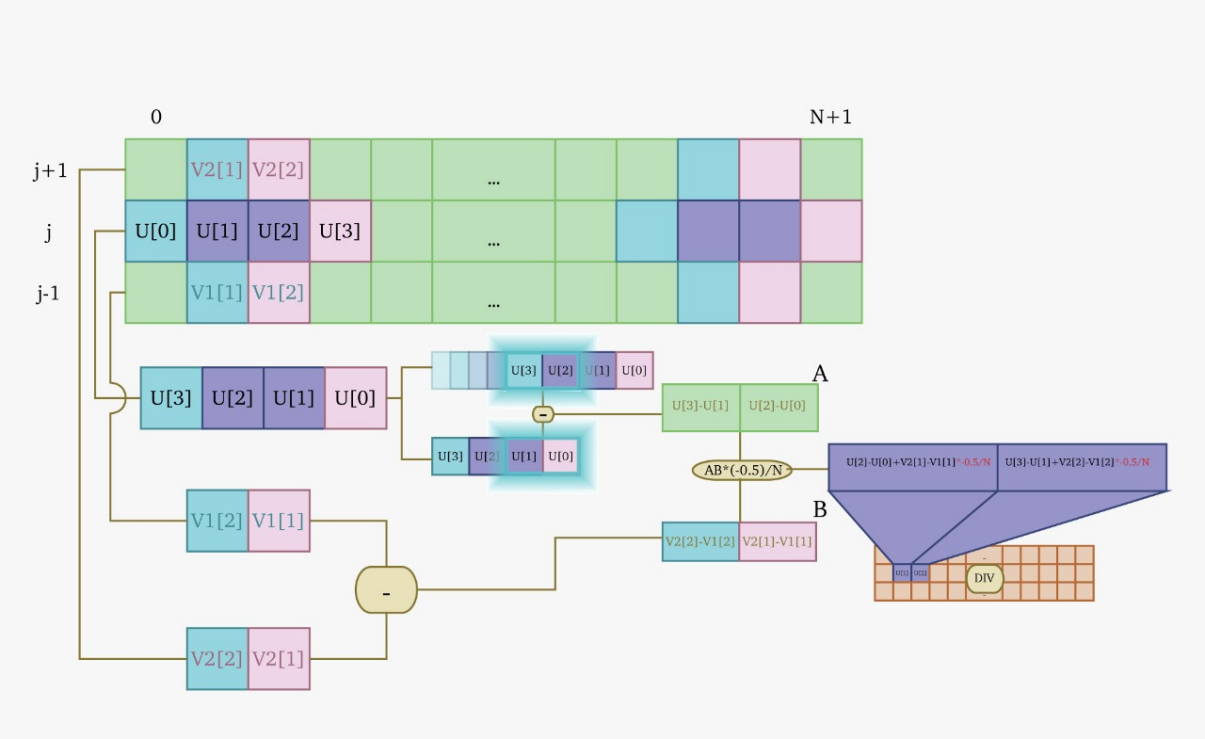
En esta función se utilizó la misma forma de recorrer la matriz por filas adoptada en *solver_lin_solve* (ver blabla), los motivos son los mismos, aprovechar el principio de localidad en caché.

El calculo a realizar es

$$div(i,j) := \frac{-0{,}5 \cdot \big(u(i+1,j) - u(i-1,j) + v(i,j+1) - v(i,j-1)\big)}{n}$$
$$p(i,j) := 0$$

En primer lugar, para el primer ciclo se utilizan los iteradores *div*, *u*, *v1* y *v2*. Y se calcula de a 2 posiciones por iteración. Leyendo 4 posiciones de *u*, 2 de *v1* y 2 de *v2*. Realizando el cálculo como indica la figura,

[4] Línea 120 y 121, *solver_set_bnd.asm*



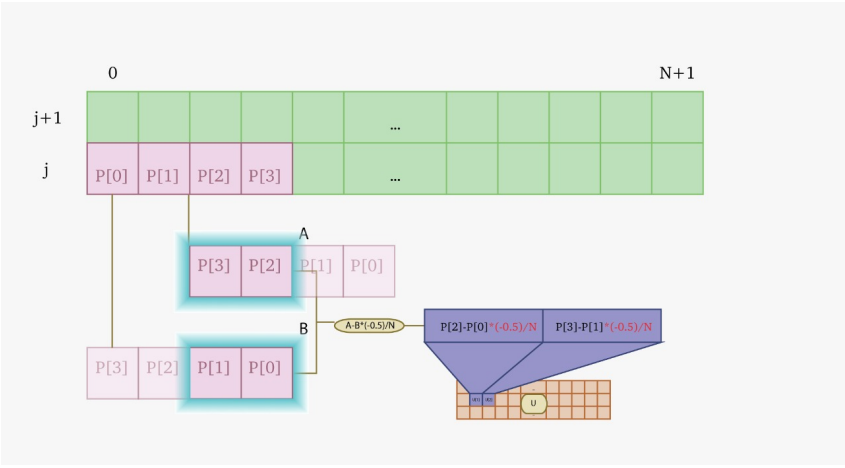
Por último, se incrementan los iteradores en 2 posiciones y se da paso a la siguiente iteración.
Para el segundo ciclo, donde debe calcularse

$$u(i,j) := u(i,j) - 0{,}5n \cdot \big(p(i+1,j) - p(i-1,j)\big)$$
$$v(i,j) := v(i,j) - 0{,}5n \cdot \big(p(i,j+1) - p(i,j-1)\big)$$

se optó por separar el ciclo en dos diferentes, en el primero se calcula u y en el segundo se calcula v.

Esto conlleva la ventaja de poder calcular de a 4 posiciones en paralelo para v y sortear la limitación de calcular de a 2 posiciones debido al calculo de u como se explicará a continuación.

La estrategia utilizada para el ciclo de u es usar 2 iteradores, *iter_p* e *iter_u*.



En cada iteración se leen 4 posiciones de p y se procesan 2 posiciones de u.
Diagrama (muestra como calcula paralelamente u)
La estrategia utilizada para el ciclo de v es usar 3 iteradores, *iter1_p*, *iter2_p* e *iter_v*.
Figura (muestra donde se ubican los iteradores)
En cada iteración se leen 4 posiciones de p, 4 de p y se procesan 4 posiciones de u.
Diagrama (muestra como calcula paralelamente v)

3. Resultados

En esta seccion se va a hablar de todos los experimentos realizados para poder observar los distintos comportamientos del programa, ya sea en cuanto a tiempos como en diferencias de resultado, con las distintas funciones implementadas.

En primer lugar se trabaja el desempeño en tiempo del programa con distintas configuraciones que seran explicadas en profundidad en cada seccion.

Finalmente se muestran en detalle las diferencias en imagenes resultantes entre ambas implementaciones y se explica la causa de las mismas

Toda esta experimentacion fue realizada repetidas veces y con distintos tamaños de entradas, esto para mitigar las diferencias entre corridas asi como para poder ver cuan distinto se comporta el programa con ambas implementaciones cuando se varia el tamaño de la entrada

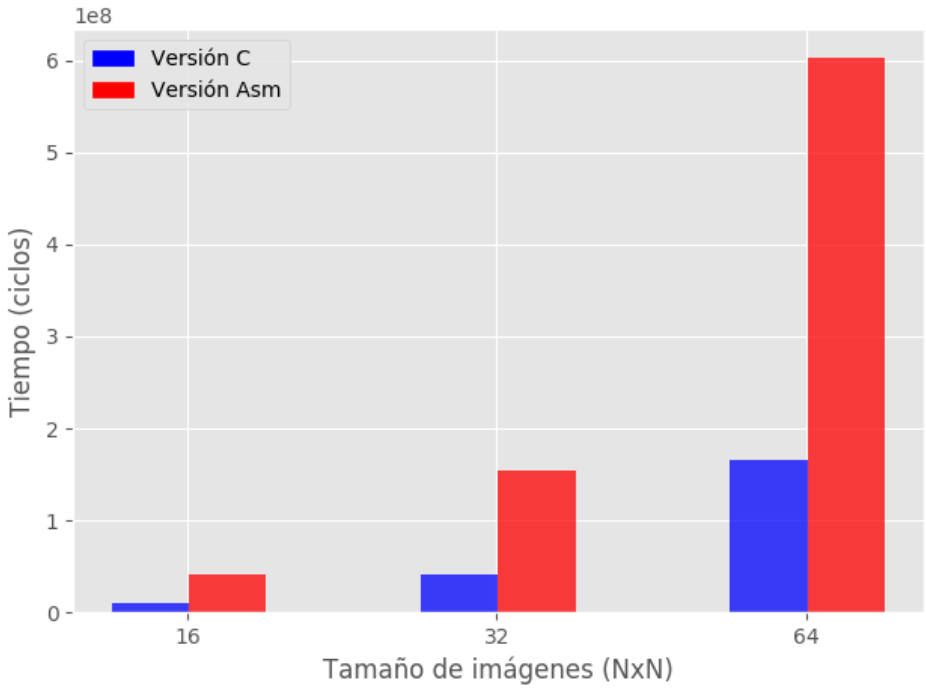
3.1. Tiempos

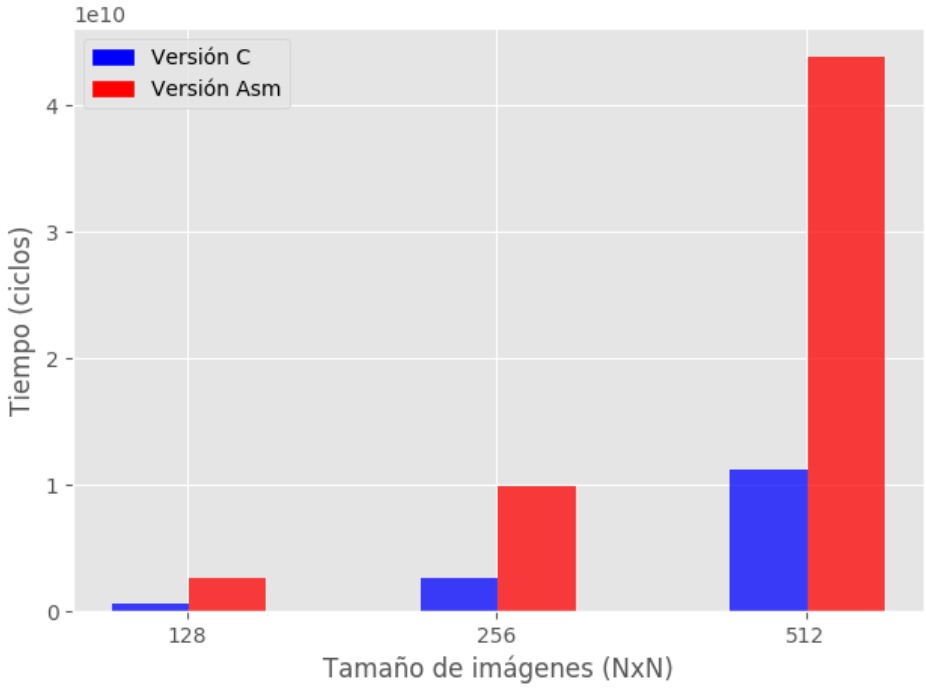
Los experimentos de tiempos de todo el programa se repitieron 20 veces y se calculó el promedio general utilizando el 80 % central de los resultados obtenidos ordenados, con el objetivo de evitar *outliers*(eliminando el 10 % en ambas puntas de los resultados ordenados de menor a mayor).

Los experimentos de tiempos de cada función específica se realizaron con *rdtsc* y se midieron ejecutando los test para distintos tamaños, guardando el tiempo de cada llamada a la función. Luego se calculó el promedio general de todos los valores obtenidos de cada función en toda la ejecución del programa.

3.1.1. Programa total

En esta seccion se compara el programa con las 3 funciones que se implementaron en asm contra todo el codigo en C.

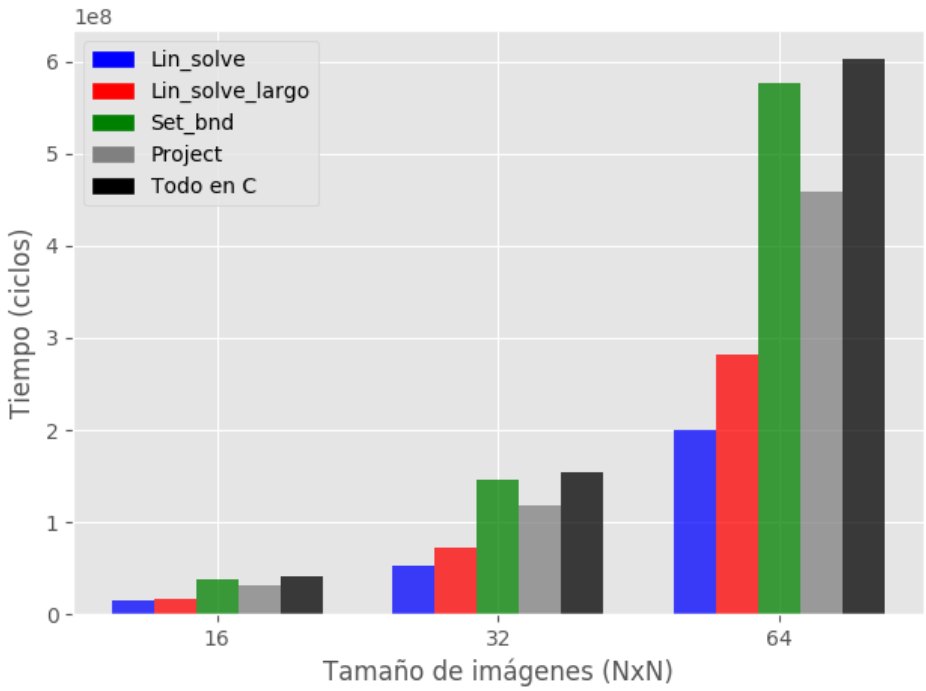


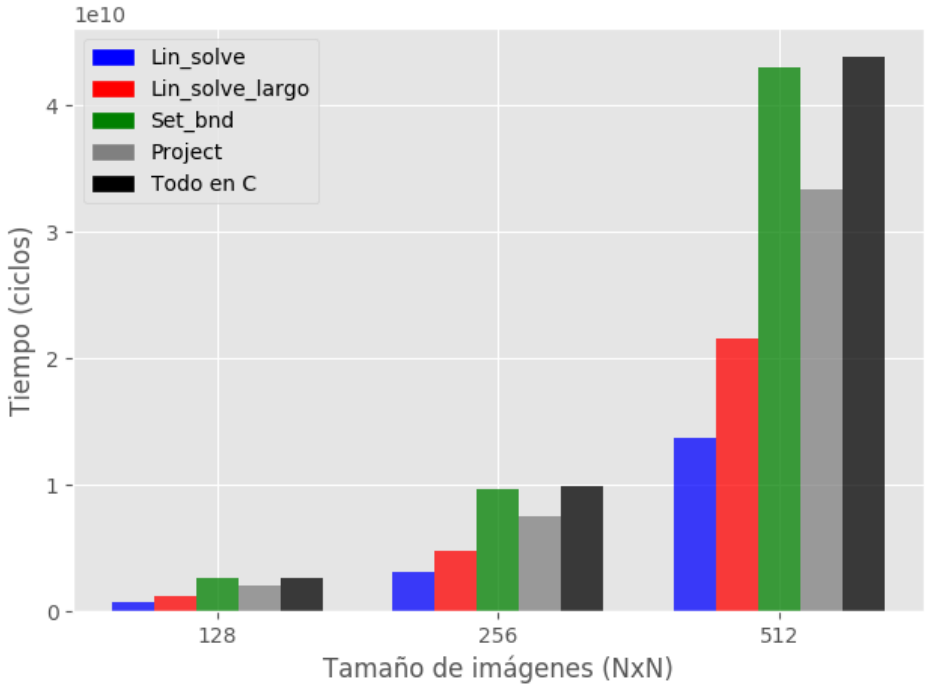


En estos graficos se puede ver claramente que la implementacion en asm es mucho mas rapida y esta diferencia crece aun más cuanto mas grande es la entrada. sin embargo en la proxima seccion se puede ver en detalle como impacta en particular cada una de las funciones en el tiempo total.

3.1.2. Una función en assembler a la vez

Aqui se observa el tiempo de todo el programa de distintas formas, todo el codigo en C, todo el codigo con solo una de las 3 funciones implementadas en asm y con dos casos para lin_solve utilizando la version final (lin_solve) y la primera implementacion mas larga y compleja (lin_solve_largo)





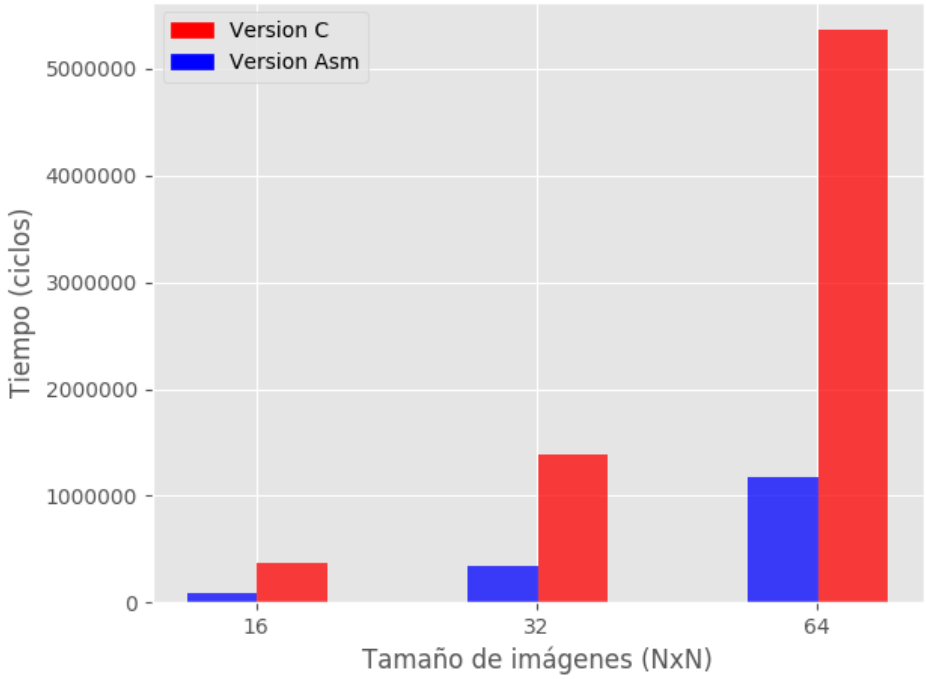
Analizando estos graficos, se puede ver que set_bnd, aunque presenta una mejoria, esta es infima en comparacion a las otras funciones. Luego en el ranking viene project, la cual mejora el total del tiempo pero aun asi ambas implementaciones de lin_solve representan la mayor parte de la mejoria de tiempo general.

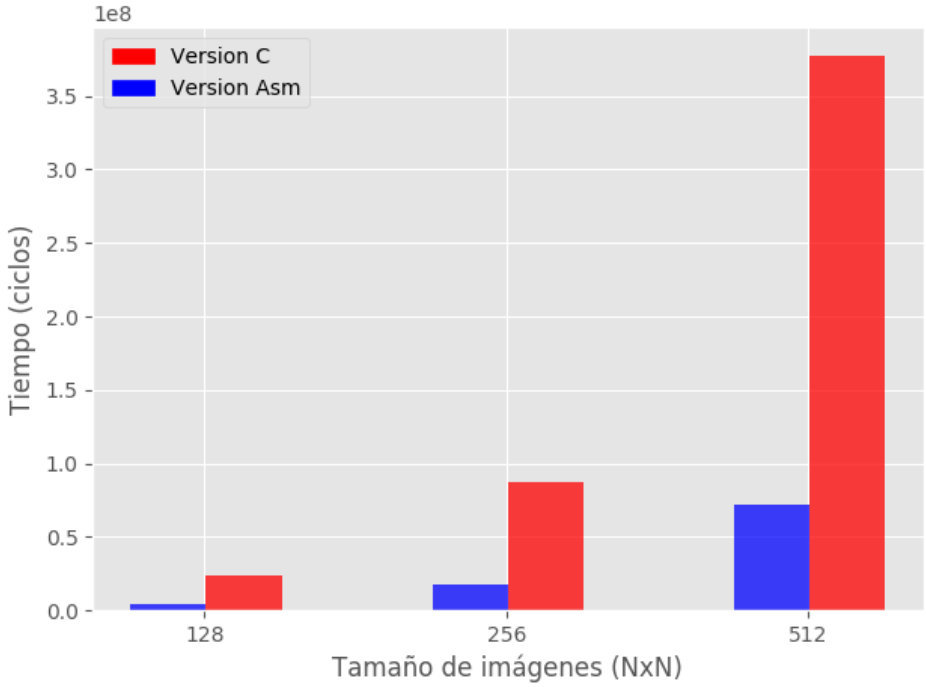
Sindo lin_solve a su vez mejor que lin_solve_largo a pesar de que primero se creia lo contrario ya que lin_solve_largo paraleliza mucho más y ejecuta menos ciclos. Sin embargo estos resultan más largos y aun más lentos en proporcion.

De aqui en adelante se trabaja con los tiempo medidos solo al principio y al final de la ejecucion de cada funcion en particular en ambas implementaciones (C y asm) comparadas

3.1.3. solver_lin_solve

Se compara lin_solve final en asm contra lin_solve original en C.

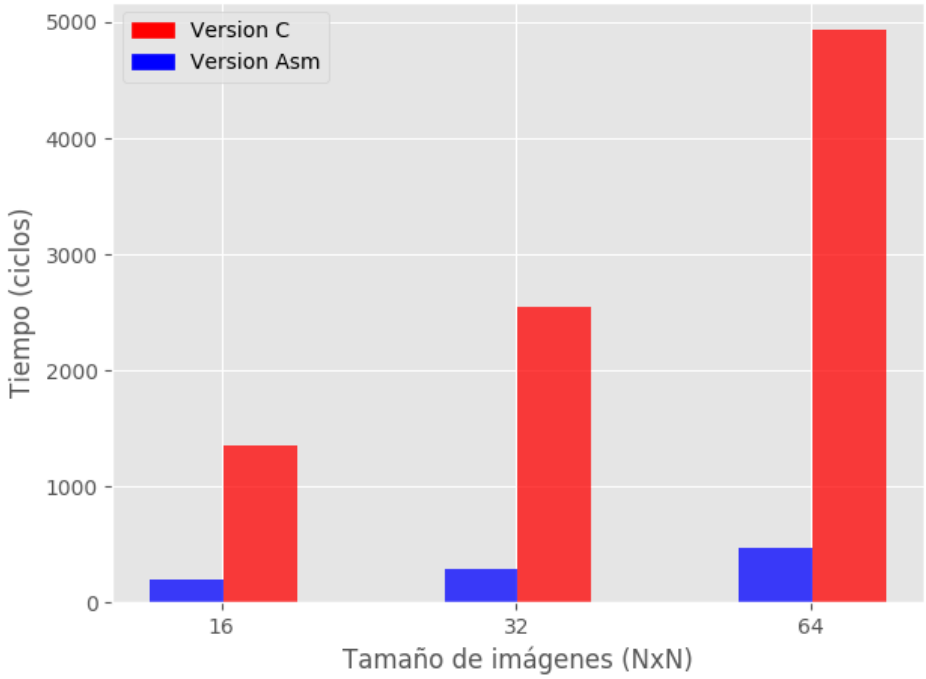


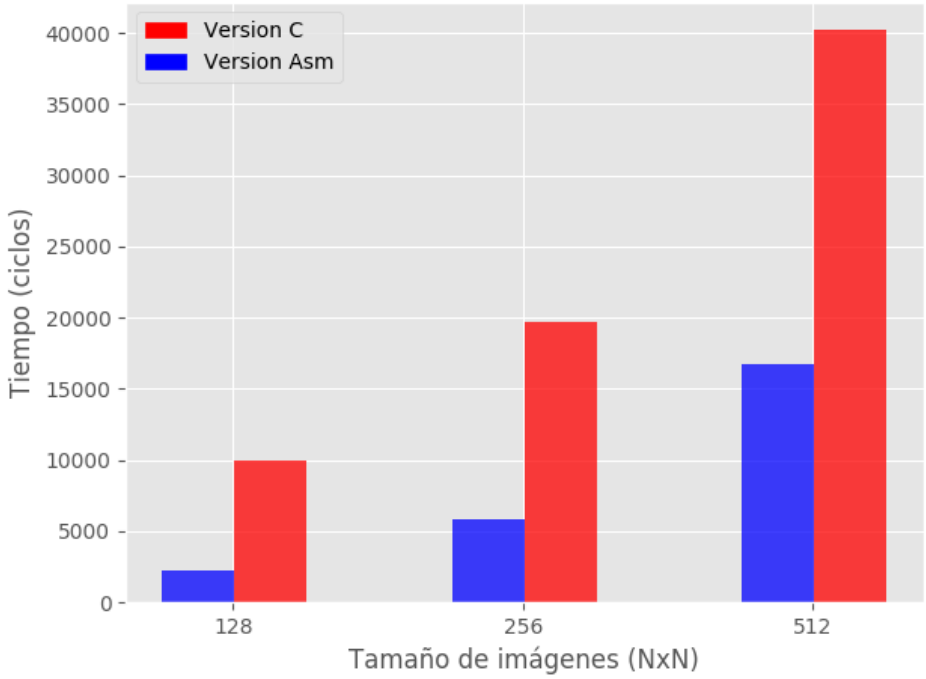


Aqui se puede ver cuan abismal es la diferencia entre las versiones de lin_solve, cabe notar que nuestra implementacion de lin_solve tiene un cambio en el orden de operaciones el cual facilita y acelera la ejecucion, ademas de que el ciclo esta dividido en casos lo que reduce las comparaciones necesarias dentro del ciclo y los simplifica sustancialmente. Pero a pesar de ser matematicamente identica a la funcion original, el orden de operaciones afecta al resultado final. Este tema sera visto con mejor detalle en la seccion diferencias.

3.1.4. solver_set_bnd

Se compara set_bnd final en asm contra set_bnd original en C.

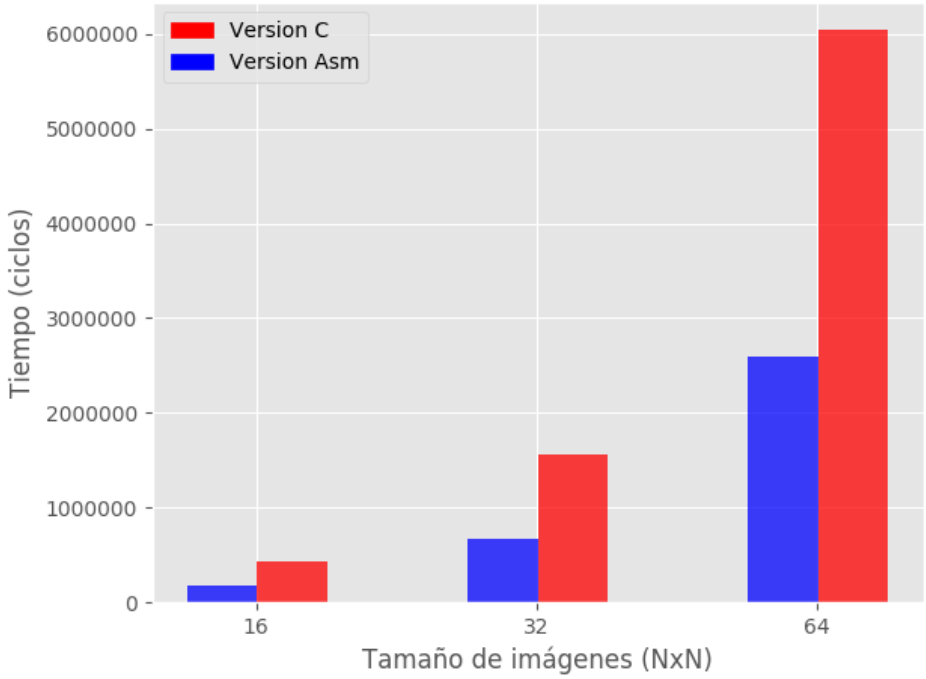


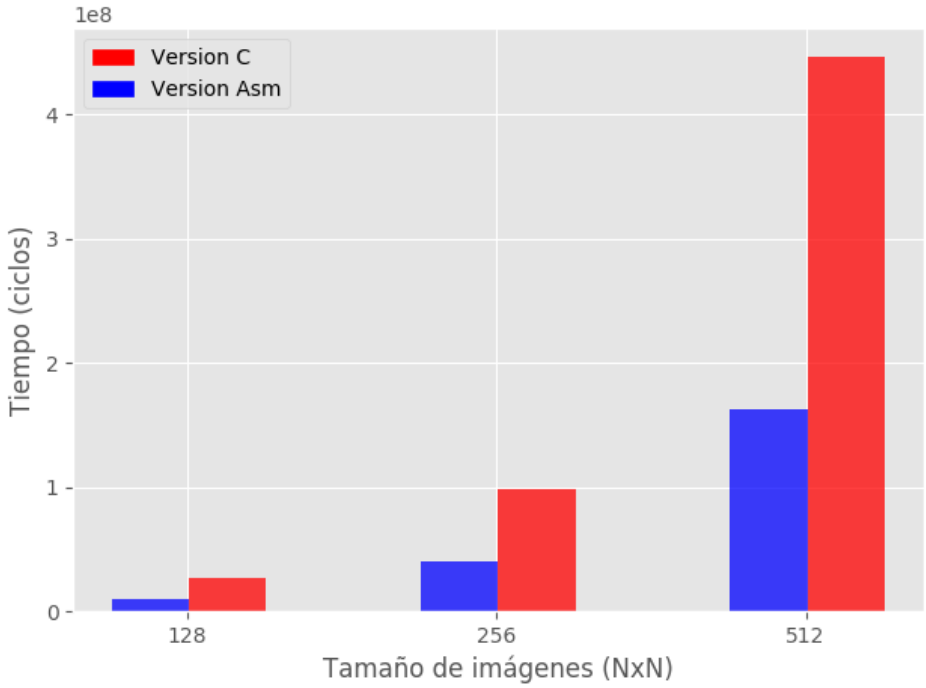


Aqui se puede ver que set_bnd tambien presenta una buena mejora en asm respecto a C, pero notese que la escala de tiempo es mucho más chica, lo cual muestra que la diferencia no significa mucho en cuanto al tiempo total de la funcion, en otras palabras, la funcion set_bnd original ya ejecutaba pocos ciclos en comparacion a las otras funciones y por eso la mejoria no es tan notable en lineas generales

3.1.5. solver_project

Se compara project final en asm contra project original en C.





Con project se puede ver que la implementacion en asm es muy buena en comparacion, pero la mejoria es un poco menor que la que presenta lin_solve, y cuanto mas grande la entrada y al crecer exponencialmente la diferencia entre C y asm, mas se nota que lin_solve recude los tiempos en mayor medida, lo cual lleva a concluir que lo visto en el primer grafico es acertado.

3.2. Diferencias

En esta seccion se observan las diferencias de las imagenes que corren los tests en base a promedios y medianas.

Aqui se trabaja con todo el codigo en asm comparado al codigo original, notese que las diferencias son en su totalidad causadas por la implementacion de lin_solve.

En primer lugar se decidio reescribir la funcion en C, modificando el orden de operacion al realizar un factor comun, lo cual mantiene la logica matematica y facilita la paralelizacion en asm y causa una mejoria temporal (como ya se vio en la seccion tiempo), pero introduce una diferencia causada por el uso de floats, cuyos resultados si se ven afectados por el orden de operacion sin importar la correctitud de la logica. Entonces, ambos codigos en C (original y reescrito) ya presentaban una diferencia entre si. Pero nuestra implementacion en asm, la cual se basa en la de C reescrita, no presenta ninguna diferencia la misma, por lo cual decidimos mantener la implementacion como algo correcto.

No hay tablas para los tamaños de 16, 32 y 64 porque no hubo diferencias con las versiones de la cátedra.

Promedio$_1$ es igual al promedio de todos los pixeles de la imagen

Promedio$_2$ es igual al promedio de solamente los píxeles que no son negros

Mediana$_1$ es igual a la mediana de todos los pixeles de la imagen

Mediana$_2$ es igual a la mediana de solamente los pixeles que no son negros

Cuadro 1: Diferencias con la cátedra – Tamaño 128×128

	128		
	Velocidad v	**Velocidad u**	**Densidad**
Promedio$_1$	0.1113	0.9035	0.0015
Promedio$_2$	1.0691	1.0096	1.0
Mediana$_1$	0	1	0
Mediana$_2$	1	1	1
Cantidad	1706	14662	25
Maximo	13	12	1

Cuadro 2: Diferencias con la cátedra – Tamaño 256×256

	256		
	Velocidad v	**Velocidad u**	**Densidad**
Promedio$_1$	14.8418	0.0733	0.0740
Promedio$_2$	14.8516	2.6657	9.9876
Mediana$_1$	16	0	0
Mediana$_2$	16	1	4
Cantidad	65493	1804	486
Maximo	105	39	129

Cuadro 3: Diferencias con la cátedra – Tamaño 512×512

	512		
	Velocidad v	**Velocidad u**	**Densidad**
Promedio$_1$	1.5447	1.8360	0.1312
Promedio$_2$	1.5721	1.8495	3.6362
Mediana$_1$	1	2	0
Mediana$_2$	1	2	2
Cantidad	257571	260225	9463
Maximo	61	30	48

De estas tablas se puede ver que, logicamente, cuanto mas grande la imagen mas pixeles distintos, pero tambien se puede ver que estos pixeles distintos no alcanzan valores muy grandes, y las medianas y los promedios se mantienen siempre muy bajo, lo cual marca que a pesar de haber diferencias, estas no son abismales.

Pero extrañamente la matrix de 256 x 256 se comporta mucho peor que la de 512 x 512 más grande, esta ultima presenta más pixeles con diferencias, pero la primera tiene diferencias mucho más grandes de hasta un poco sobre la mitad del rango de color. Realmente no pudimos explicar este comportamiento.

Tambien se ve que la matriz densidad es por mucho la menos afectada en cantidad de pixeles distintos y en diferencias en general, asumimos que esto se da porque la matriz de densidad pasa una sola vez por lin_solve por ciclo, cuando las matrices de velocidad pasan almenos dos veces por lin_solve en cada ciclo.

4. Conclusión

En concuclusion podemos decir que este trabajo nos permitio aprender y volcarnos en muchos aspectos distintos, por ejemplo la programacion con SIMD y el procesamiento de matrices en assembler asi como en el desarrollo de un informe profundo de experimentacion y analisis.

Se puede ver claramente lo que se buscaba sobre performance del programa , ademas de poder justificarlo con pruebas y experimentos, efectivamente se nota la mejora que causa el uso de la metodologia SIMD y assembler cuando se trata de implementaciones que necesitan el procesamiento sistematico de muchos datos con caracteristicas iguales.

Tambien descubrimos con la implementacion de lin_solve (lin_solve_largo) que no siempre más paralelizacion y mas trabajo por ciclo lleva a una mejoria temporal sino que es mas importente aprovechar bien la logica de la memoria cache y no realizar saltos lejanos en memoria en cada iteracion, como se penso en esta funcion en un principio. Otro problema que encontramos fue que no se puedo sacar una conclusion clara de las tablas de diferencias ni sobre por que los tests presentan el comportamiento particular mostrado en los resultados. Quizas lo necesario seria mas conocimiento profundo sobre como trabaja el programa en general.

Cabe aclarar que este trabajo tambien sirvio como primer informe en la carrera de 2 integrantes del grupo, lo cual aporto para lograr que sea una experiencia en la que se aprendio mucho sobre como se arman, revisan y se desarrollan informes de investigacion.