DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Organización del computador II
Segundo Cuatrimestre de 2017

Squanchy

Integrante	LU	Correo electrónico
Arroyo, Luis	913/13	luis.arroyo.90@gmail.com
Humpiri Arenas, Juan Carlos	842/14	jhumpiri1599@gmail.com
García Gómez, Luis Enrique	675/13	garcia_luis_94@hotmail.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) 4576-3300
http://www.exactas.uba.ar

Contents

1 Introducción

Este informe está diseñado para ser presentado como Trabajo práctico número 2 de la materia Organización del Computador 2

Consistirá en la modificación de un programa escrito enteramente en C, y serán editadas 3 funciones, que intentaremos optimizarlas escribiendo su código en lenguaje ensamblador. Algunas funciones de estas, usarán funciones SDIM sobre registros XMM para procesar datos en paralelo ya que, el tipo de dato que se usa en este programa es de tamaño float, por lo cual nos permiten poder procesar en teoría hasta 4 floats en paralelo, Pero Esto probablemente no pueda ser completamente así ya que las operaciones necesitan datos de sus vecinos. Esto se verá en detalle en cada función.

Cada función tendrá una sección en donde se explicará que es lo que hace y cual es la estrategia para resolverlo.

Por ultimo tendremos una fase de experimentación donde pondremos a prueba el programa en general utilizando los diversos niveles de optimización que tiene el compilador GCC y también evaluaremos el funcionamiento de las funciones individualmente y las pondremos a prueba para estudiar cuales son las que mejoran y cuales no dentro de nuestra optimización.

Respecto al código que compone una parte del trabajo, este se encuentra presente en un repositorio en $gitlab_{exactas}$ al que se le dará acceso a los docentes para realizar la corrección. Información útil fue añadida a README.md al tag de release creado.

2 solver lin solve

2.1 Descripción de la función

La función toma como parámetros:

- fluid_solver* de nombre *solver*, un puntero a una estructura con información sobre la dimensión de la matriz X.
- uint32_t de nombre b, un entero positivo.
- float* de nombre X, es la matriz que contiene datos del estado anterior.
- float* de nombre $X0$, es la matriz cuyas posiciones hay que actualizar.
- float de nombre c, es un numero real
- float de nombre d, es un numero real

y no devuelve dato alguno, tiene como objetivo calcular la difusión del fluido a modelar, en base al código presentado en el trabajo práctico base.

2.2 Estrategia de Resolución

A la hora de encarar la resolución de esta función utilizando instrucciones SIMD, surgió la siguiente pregunta: ¿cuál es la menor cantidad de accesos a memoria que debemos hacer para poder procesar la mayor cantidad de datos posibles en paralelo?, entonces lo que propusimos fue:

Como las matrices X y $X0$ contienen datos del tipo float, los cuales son de tamaño de 32 bits y nosotros trabajamos con registro XMM de tamaño de 128 bits, vamos a poder trabajar con 4 datos en paralelo que es lo más eficiente. Para esto debemos hacer 4 accesos de memoria en la matriz X y un acceso de memoria en la matriz $X0$ (como se muestra en la imagen 1), para poder procesar 4 datos en paralelo.

Cabe aclarar que tamaño de las matrices era múltiplo de 4, para poder facilitar el proceso en paralelo de los datos.

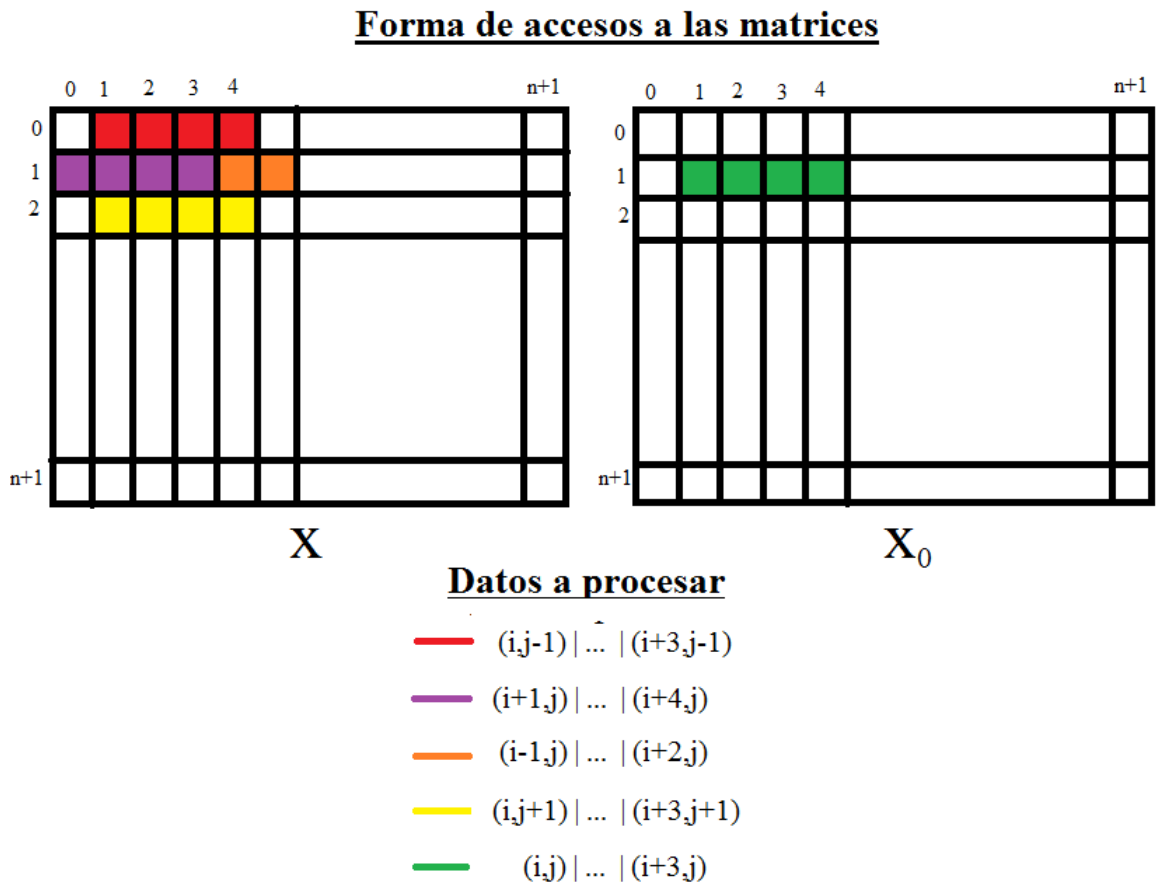

Figure 1: Acceso de memoria a las matrices

Una vez que terminamos los accesos a memoria de las matrices X y $X0$, queremos procesar los datos en paralelo pero surgió otro problema a la hora de querer hacer la siguiente operación:

$$X[IX(i,j)] = (X0[IX(i,j)]+a*(X[IX(i-1,j)]+X[IX(i+1,j)]+X[IX(i,j-1)]+X[IX(i,j+1)]))/c \quad (1)$$

Pasa que como C procesa un dato a la vez, cada vez que calcula $X[IX(i,j)]$ siempre tiene actualizado el dato $X[IX(i-1,j)]$, pero en comparación con nuestro modelo de resolución, nosotros una vez que nos posicionamos en el dato $X[IX(i-1,j)]$, levantamos el dato queremos y los 3 datos siguientes (como se puede ver en el vector morado de la imagen 2), pero claramente esos 3 datos están desactualizados para poder trabajarlos en paralelo.

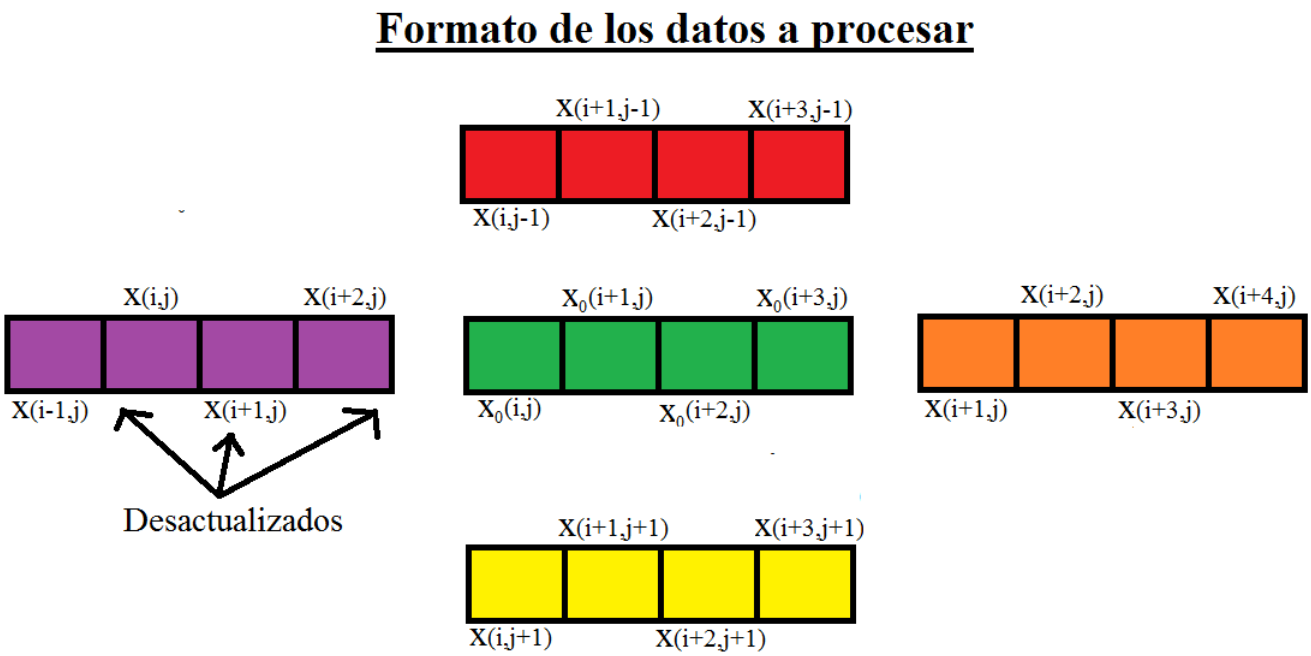

Figure 2: Formato de los datos traídos de memoria

Para solucionar dicho problema, tuvimos que calcular cada dato por separado y una vez solucionado el problema pudimos procesar los datos en paralelo (como se muestra en la imagen 3).

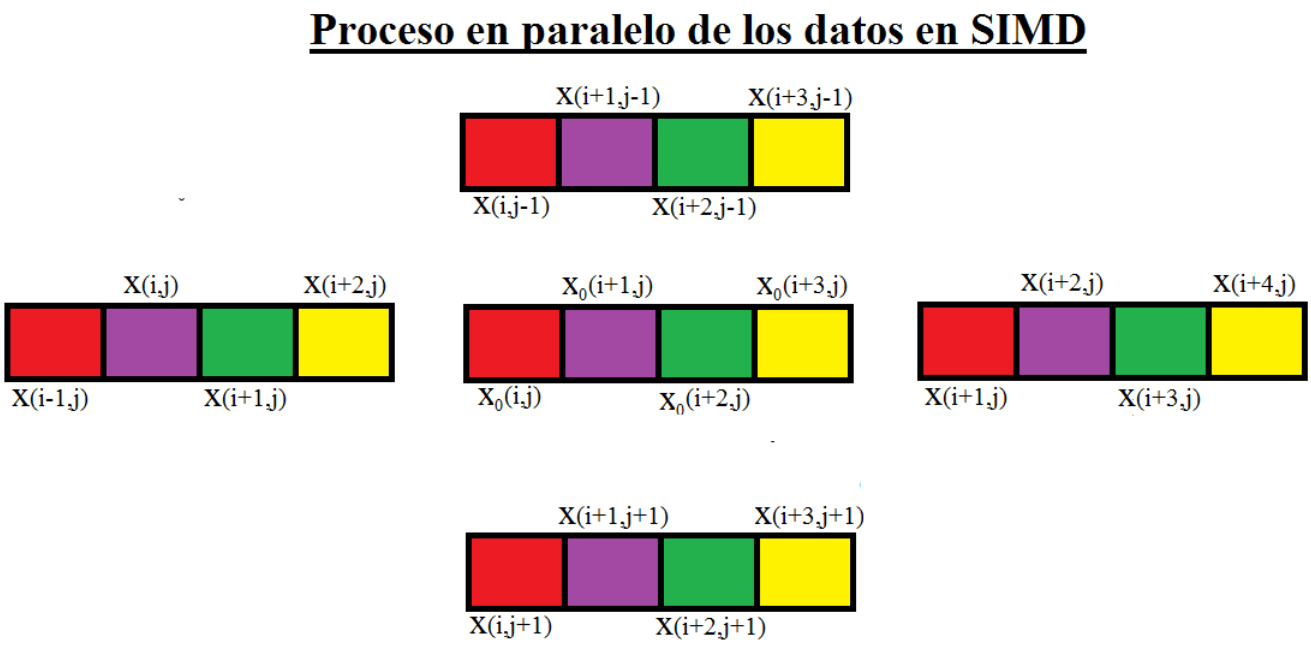

Figure 3: Procesamientos de los datos en paralelo

Una vez terminado de procesar los datos de las matrices como última instrucción se llama a la función "solver_set_bnd" y con eso concluimos con la finalización de la función 'solver_lin_solve'.

Aclaración: Nuestra implementación de la función 'solver_lin_solve' tiene un error del 0.3 al 2.6, pero lo consultamos con nuestro tutor y nos dijo que si el error era menor que 5 estaba bien.

3 solver_set_bnd

En base al enunciado del trabajo práctico, fue requerido escribir esta función en su versión en assembler, teniendo en cuenta que no era necesario hacer uso de la tecnología SIMD.

3.1 Descripción de la función

La función, que toma como parámetros:

- fluid_solver* de nombre solver, un puntero a una estructura con información sobre la dimensión de la matriz x
- uint32_t de nombre b, un entero positivo cuyo valores esperados son 1 (uno) o 2 (dos)
- float* de nombre x, la matriz en cuestión

y no devuelve dato alguno, tiene como objetivo asignar valores adecuados a las posiciones correspondientes de los bordes de la matriz x en base al código presentado en el trabajo práctico base. Como puede verse en la Figura 4.

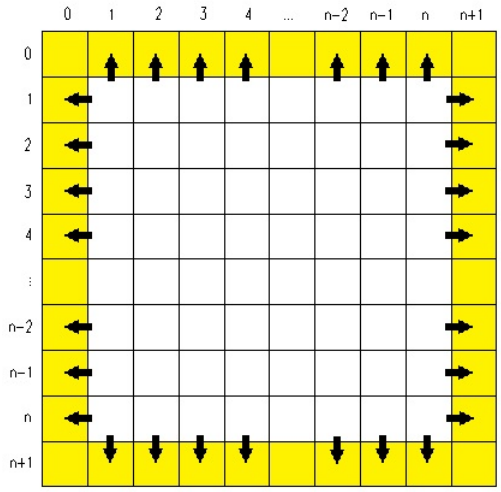

Figure 4: Flujo de datos esperado al finalizar de ejecutar la función

3.2 Resolución

Con el objetivo de resolver el problema de manera que el código final contribuyera con una mejor performance en tiempo de ejecución de la simulación, decidimos implementar la función en cuestión utilizando la tecnología SIMD. Para ello debimos tener presente principalmente como es la representación en memoria de la matriz x así como también pensar de que manera recorrer la estructura para lograr el propósito final.

Una de las cosas que pudimos asumir (por una consulta que se había hecho previamente a la lista de mail de la materia) es que el tamaño de la matriz sin los bordes es múltiplo de cuatro. Es decir, solver→N es múltiplo de cuatro y por lo tanto no tuvimos que considerar casos especiales a la hora de recorrerlo.

La implementación que hicimos consta de tres etapas:

- **Adecuación de los elementos de los bordes superior e inferior que no pertenecen a las esquinas**: La idea es recorrer la matriz x mediante un ciclo que consta de solver→N/4 iteraciones, en cada cual se leen cuatro posiciones de memoria contiguas a partir de la posición IX(i, 1) y IX(i, solver→N) con i siendo la variable iterada en el rango [1, solver→N] para ir procesando los datos. Es decir, hay dos registros xmm, cada uno con cuatro valores contiguos adyacentes al borde, para cada cual luego mediante un checkeo del atributo b, de ser necesario se invierte el signo de todos los elementos de cada registro, teniendo en cuenta la norma IEEE-754. Finalmente se procede a

guardar el resultado de las operaciones en las posiciones IX(i, 0) para el caso del registro que fue leído de la posición IX(i, 1) y IX(i, solver→N+1) para el caso del registro que fue leído de la posición IX(i, solver→N). Esta de más destacar que esta operación puede hacerse con un solo ciclo y que dada que con una sola lectura se pueden procesar cuatro elementos, entonces la variable iteradora i avanza de a cuatro unidades. La Figura 5 ilustra esta etapa.

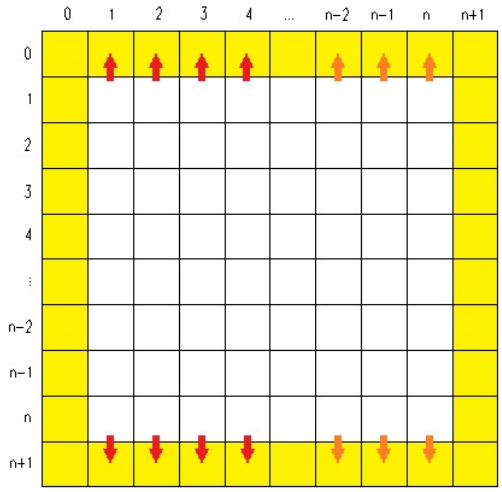

Figure 5: Flujo de datos al finalizar la primera etapa. El color determina la cantidad de elementos que se procesan por lecura y escritura de memoria

- **Adecuación de los elementos de los bordes laterales que no pertenecen a las esquinas ni a las posiciones (0, 1), (1, 1), (solver→N, solver→N), (solver→N+1, solver→N)**: La idea principal utiliza el hecho de que si bien la representación visual de x es una matriz, la representación en memoria es contigua. Es decir, cada fila está al lado de otra. Por lo tanto, una lectura utilzando la tecnología SIMD de la posición (solver→N, i) con i siendo la variable iterada en el rango [1, solver→N-1], escribe en un registro xmm cuatro elementos (contiguos), correspondiendo los primeros dos a los elementos del borde lateral derecho de la fila i y los segundos dos a los elementos del borde lateral izquierdo de la fila i+1. Finalmente se procede a realizar la operación en cuestión mediante el uso de la instrucción pshufd y a invertir el signo teniendo en cuenta el valor del parámetro b y la norma IEEE-754. Se puede ver un esquema del contenido del registro xmm y como deben ser operados en la Figura 6.

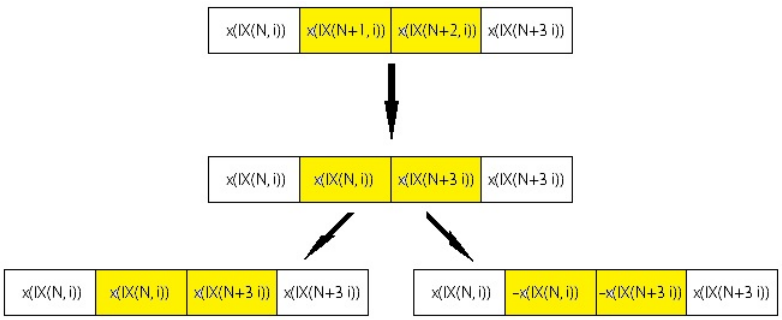

Figure 6: Procesamiento de cuatro elementos de los bordes laterales de la matriz

Una vez realizado esto, se guarda el registro procesado en la posición de la que se lo leyó. Es necesario destacar que i no puede tomar valor cero, ya que una lectura desde la posición (solver→N, 0) tendría en cuenta para procesamiento indicado en un registro xmm una de las esquinas, lo cual no es deseable. Analogamente para la lectura desde la posición (solver→N, solver→N). De esta manera se excluyen las posiciones que pertenecen a los bordes laterales (0, 1), (1, 1), (solver→N,

solver→N), (solver→N+1, solver→N), por lo cual se escribe código específico que no hace uso de la tecnología SIMD. En la Figura 7 se puede visualizar que elementos de la matriz x están siendo procesados.

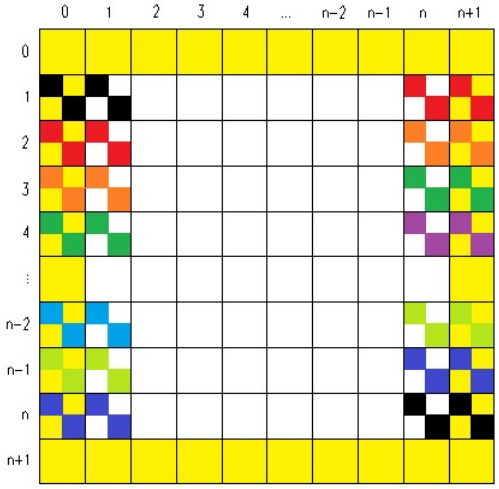

Figure 7: Las lecturas se hacen de a cuatro posiciones contiguas, marcadas con trama del mismo color. Las posiciones con trama negra no se leen con tecnología SIMD, si no que se procesan de manera convencional

- **Adecuación de los elementos de los bordes que pertenecen a las esquinas**: La idea es análoga a la propuesta en el código original. Se escribe código específico para estas posiciones sin hacer uso de la tecnología SIMD.

3.3 Consideraciones sobre los accesos a memoria

- En la primera etapa de la implementación, se hacen solver→N/4 lecturas y solver→N/4 escrituras. Lo cual deja un total de solver→N/2 accesos a memoria
- En la segunda etapa de la implementación, se hacen (solver→N-1) lecturas y (solver→N-1) escrituras, una lectura de la posición (1, 1), una escritura en la posición (0, 1), una lectura de la posición (solver→N, solver→N), una escritura de la posición (solver→N, solver→N+1). Lo cual deja un total de solver→N+3 accesos a memoria
- En la tercera etapa de la implementación, se hacen dos lecturas y una escritura por esquina. Lo cual nos deja un total de 12 accesos a memoria

Por lo tanto se hacen un total de solver→N/2 + solver→N+3 + 12 accesos a memoria por haber procesado con tecnología SIMD.

4 solver_project

4.1 Descripción de la función

La función que toma como parámetros:

- fluid_solver* de nombre solver, un puntero a una estructura con información sobre la dimensión de las matrices solver→u, solver→v, div, y p
- float* de nombre p, una de las matrices cuyas posiciones hay que actualizar
- float* de nombre div, otra de las matrices cuyas posiciones hay que actualizar

  y no devuelve dato alguno, tiene como objetivo asignar valores adecuados a las posiciones correspondientes de las matrices div y p, como así también solver→u y solver→v en base al código presentado en el trabajo práctico base.

4.2 Resolución

Analogamente al caso de la función solver_set_bnd, para hacer la implementación de esta función en SIMD, tuvimos que tener presente la representación en memoria de las matrices involucradas y tuvimos que hacer un trabajo en la interpretación de los índices. Nuevamente estamos asumiendo que el tamaño de las matrices sin contar sus bordes es múltiplo de cuatro, es decir, solver→N es múltiplo de cuatro.

La implementación que hicimos consta de dos etapas importantes:

- **Adecuación de los elementos de la matriz div y la matriz p**: Para calcular el valor de una posición IX(i, j) con i en el rango [1, solver→N] y j en el rango [1, solver→N], es necesario contar previamente con los valores de solver→u[IX(i+1, j)], solver→u[IX(i-1, j)] por un lado y por el otro solver→v[IX(i, j+1)], solver→v[IX(i, j-1)], por lo que va a ser necesario hacer en principio lecturas a dos matrices diferentes. Teniendo en cuenta que queremos hacer cuatro escrituras a la vez de datos de la matriz div por medio de la tecnología SIMD podemos empezar a analizar los casos.
  - Empecemos con la matriz **solver→v**: Si se quieren actualizar las cuatro posiciones de la matriz div a partir de la posición IX(i, j), entonces es necesario leer los cuatro valores de solver→v a partir de la posición IX(i, j+1) en un registro xmm y a partir de la posición IX(i, j-1) en otro registro xmm, ya que mediante una resta de estos registros tenemos cuatro elementos listos para ser procesados con el resto del algoritmo y obtener cuatro valores de div a partir de la posición IX(i, j). En la Figura 8 puede observarse como es el procedimiento por iteración.

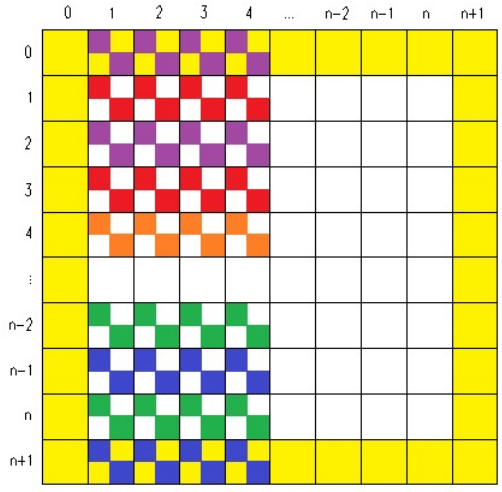

Figure 8: Lecturas que tienen que ser hechas desde solver→v. Para dos sets de cuatro elementos del mismo color determinan los valores que se leerán para calcular las posiciones que se encuentran entre ellos en conjunto con el resto del algoritmo

– Sigamos con la matriz **solver→u**: Para obtener cuatro valores de la matriz solver→u a partir de la posición IX(i, j) es necesario algo un poco más complejo que en el caso anterior, ya que en particular para hacer esta lectura y por el caracter del ciclo que estamos corriendo, dependemos de los valores adyacentes a la posición en cuestión. De esta manera para calcular la posición de un único valor de div, por ejemplo el de la posición IX(i, j), es necesario contar con los valores de solver→u en las posiciones IX(i+1, j) y IX(i-1, j). Notemos que para poder obtener cuatro posiciones de div, no nos va a bastar con una sola lectura ya que a lo sumo podríamos calcular dos valores de solver→u, y necesitamos cuatro. Por lo tanto hacemos dos lecturas de solver→u con la tecnología SIMD, la primera se realiza en la posición IX(i-1, j), leyendo así cuatro elementos que serán guardados en un registro xmm y la segunda en la posición IX(i+1, j), leyendo así otros cuatro elementos que serán guardados en otro registro xmm. Notar que de esta manera los primeros dos elementos de la primera lectura coinciden con los segundos dos elementos de la segunda lectura, obteniendo así lo necesario para calcular cuatro valores de solver→u. En la Figura 9 puede observarse como es el procedimiento por iteración. Luego, mediante operaciones que nos permiten manipular registro xmm como psrldq, subps, y shufps obtenemos finalmente los cuatro valores de solver→u necesarios para poder continuar con el algoritmo y junto con los valores de solver→v que tenemos previamente calculados, obtener los valores para div.

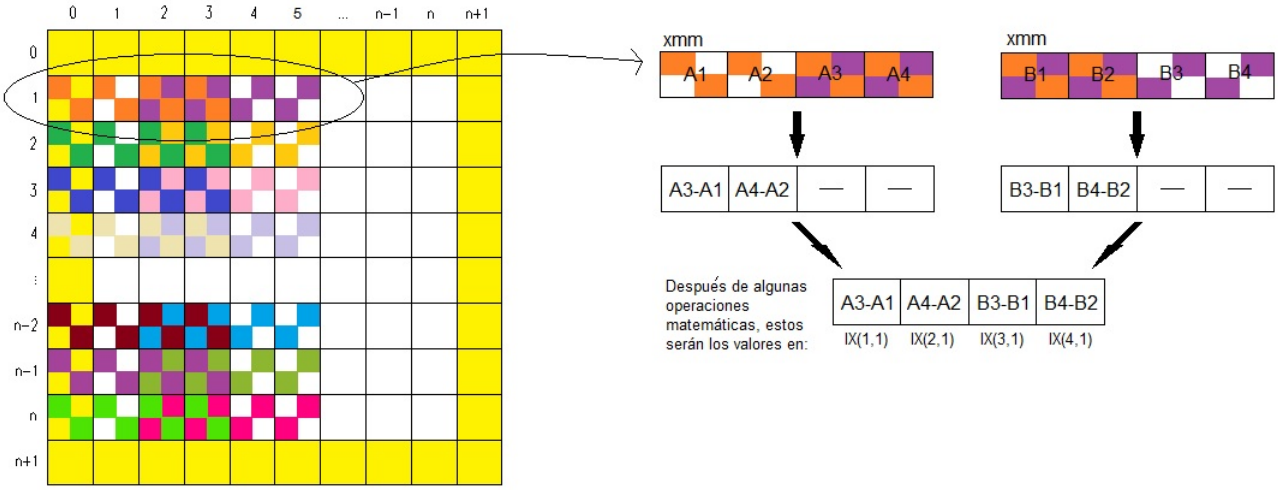

Figure 9: Lecturas que tienen que ser hechas desde solver→u. Indica algunas de las primeras lecturas con el detalle para la primera de todas.

Finalmente se aplican operaciones aritméticas para registros xmm que nos permiten obtener los cuatro valores finales a asignar a div.

La inicialización del contenido de la matriz p se lleva a cabo en el mismo lugar en el que se inicializa la matriz div, mediante la tecnología SIMD, escribiendo cuatro valores nulos por vez.

- **Adecuación de los valores de las matrices solver→v y solver→u**: Ambas matrices dependen de los valores de la matriz p de diferentes maneras, y las lecturas y escrituras se hacen casi de manera análoga a los casos comentados previamente, sólo que a diferencia, esta vez se hace sobre la matríz p y a los resultados obtenidos es necesario sumar el valor de posición de la matriz que se quiere asignar.

5 Experimentación

5.1 Introducción

Queremos estudiar el rendimiento del programa, siendo optimizado utilizando registros SIMD y un programa escrito en ensamblador por nosotros. Para esto, vamos a realizar mediciones de tiempo de ejecución utilizando la unidad de clock de reloj a partir de la librería que viene provista de la cátedra.

Primero vamos a realizar una evaluación de rendimiento del programa, utilizando el compilador de GCC con los 4 niveles de optimización -O0 -O1 -O2 -O3, luego remplazamos las 3 funciones que hemos diseñado en asembler y evaluaremos el rendimiento con esta modificación.

El tamaño de las matrices que vamos a utilizar en nuestros casos de test irán desde el tamaño 4 hasta el 2048.

Para poder replicar los experimentos realizados en esta sección, tenemos un branch en nuestro repositorio de nombre *test*, el cual contiene un script que replica absolutamente todos los test que fueron corridos y al final grafica en base a los datos obtenidos.

Para tener una medida aproximada del rendimiento general del programa dentro de nuestro rango de mediciones, vamos a calcular el área bajo las curvas con la librería de Python haciendo una integración con el método de integración trapezoidal.

5.2 Rendimiento General:

5.2.1 Metodología de la experimentación

Vamos a considerar como una iteración dentro de nuestra experimentación, a la ejecución de 20 pasos del programa.

Para medir el tiempo, vamos a realizar 20 iteraciones y luego nos quedaremos con la mediana para tener una estimación mas eficiente a la hora de tener outliers en nuestra medición. Ya que, al estar siendo ejecutado por una computadora, en base a un sistema operativo, esta puede ser mas sensible a tener instantes en el que se queda tildada por lo cual puede generarse outliers y por este inconveniente nos parece una estimación mas ajustada la de tomar la mediana en vez del promedio de todos los valores.

5.2.2 Experimentos

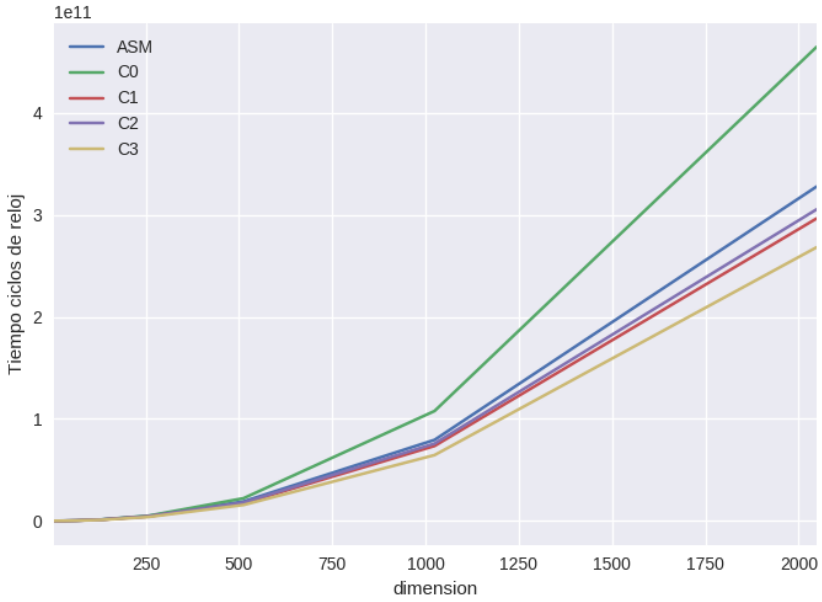

Figure 10: relación de tiempo-espacio

La figura 10 corresponden al rendimiento de las 4 compilaciones de C y a la compilación utilizando las 3 funciones en asembler

Por lo que se puede ver, resulta que, la función en asembler, es mejor que la compilación de C sin ningún tipo de optimización, pero cuando lo comparamos contra las otras optimizaciones del compilador, resulta que esto no resulta tan óptimo como esperamos. Esto puede ser debido a que, nuestro código no es lo suficientemente óptimo contra todas las heurísticas de optimización que usa el compilador gcc, o también puede ser que, a la hora de compilar la función de asembler, lo hicimos compilando el resto del programa en O0, por lo cual estaríamos acarreando un error ya no estaríamos viendo realmente si la deficiencia del código comparado contra las optimizaciones se da por culpa de las funciones escritas en asember, o porque todo el código fue compilado en distinto nivel de optimización.

Por esto, vamos a Recompilar las funciones de asembler con los distintos niveles de optimización y por ultimo compararlo

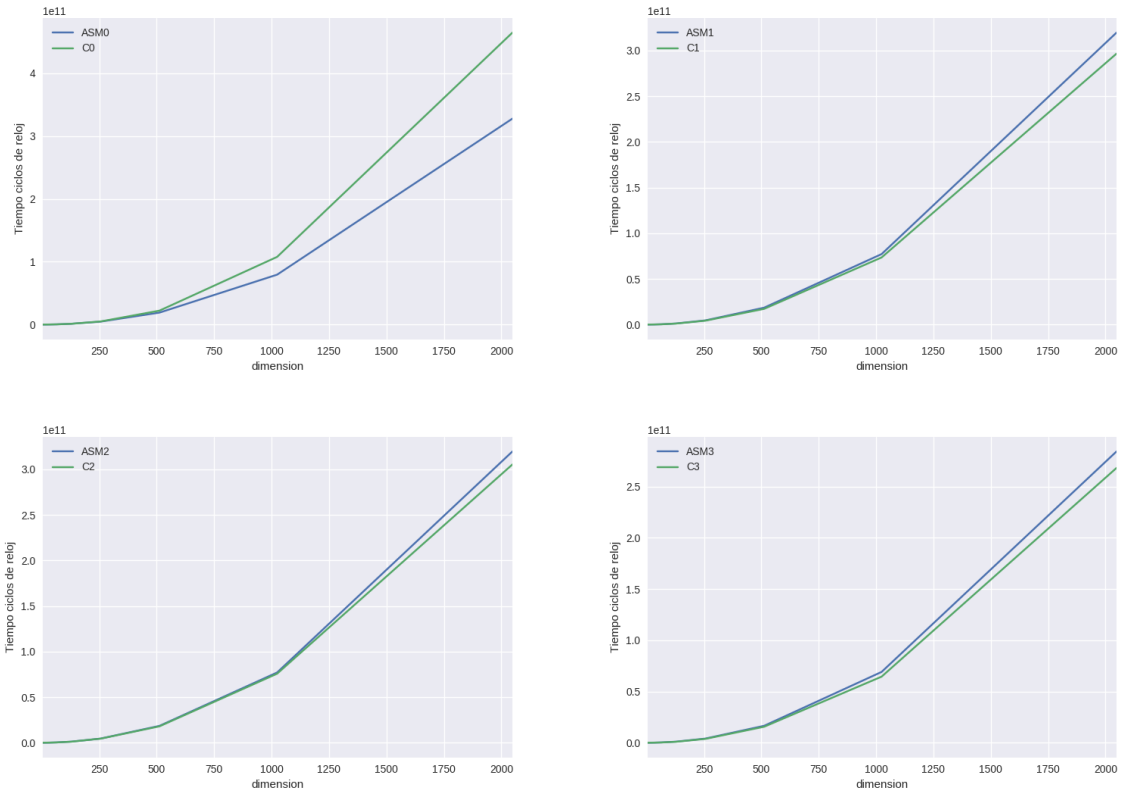

Figure 11: Corrida de test con su respectivo nivel de compilación

Lamentablemente vemos que no mejora demasiado a pesar de haber optimizado el resto del código, esto puede deberse a que, las funciones creadas en asembler por nosotros no fueron optimizadas de la mejor manera que se puede hacer, de todas formas se puede ver que que para cada compilación que no sea la primera, el tiempo de proceso no se aleja demasiado nunca de la curva de asembler.

5.2.3 Rendimiento

Como estas curvas, no nos muestran una relación completamente directa con el rendimiento de nuestras funciones, vamos a calcular el área bajo la curva de cada una de nuestras curvas.

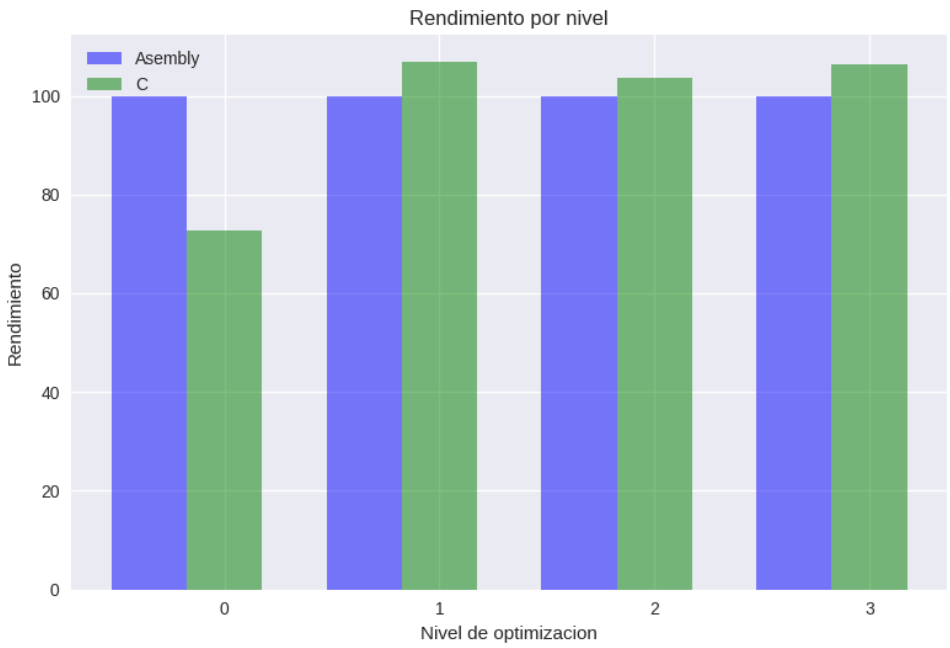

Figure 12:

Resulta interesante ver que nuestro programa comparado contra el compilador sin aplicar ningún tipo tipo de compilación es un 25 % mas eficiente, pero al utilizar las distintas heurísticas de optimización decae. Lo positivo es que, a pesar de que las funciones escritas en asembler no superan todos los niveles de optimización en C, la diferencia no es demasiado grande. Como se puede ver, en ningún caso llega a superar el 10%

En base a estos datos, lo que vamos a hacer a continuación es un análisis mas exhaustivo, vamos a analizar cada función por separado y así poder ver cuales son las que están menos optimizadas.

5.3 solver lin solve

5.3.1 Metodología de la experimentación

En este caso, lo que vamos a considerar como una iteración válida de medición de tiempo, es el tiempo que tarda en ejecutarse cada vez que se hace el llamado a función y luego tomaremos el tiempo medio de todas las iteraciones dadas para un tamaño determinado, de esta forma podemos acercarnos lo mas posible a el rendimiento exacto de esta función porque vamos a medir exclusivamente el tiempo en el que tarda dicha función en cada nivel de optimización y en asembler.

Las mediciones se realizarán con la función compilada en los 4 niveles de optimización de C y también con nuestra función escrita en asembler.

5.3.2 Experimentos

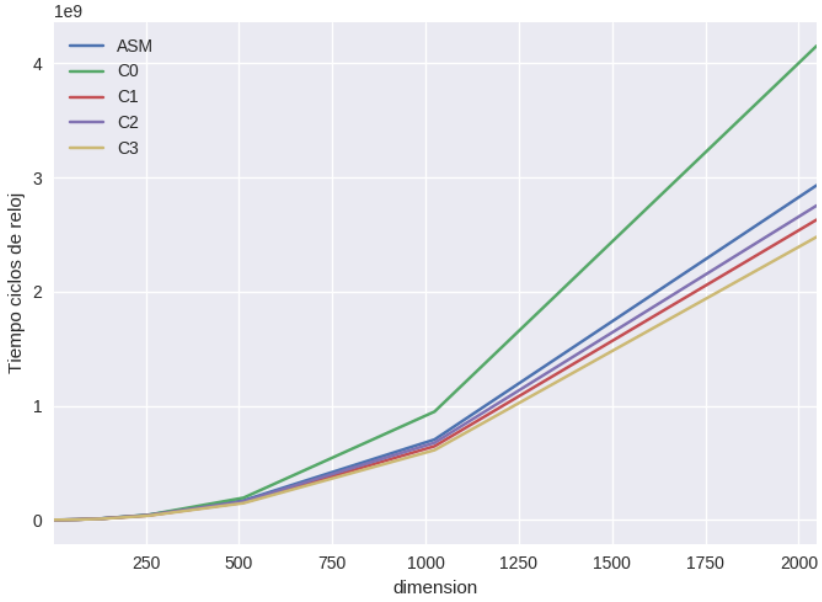

Figure 13: Tiempos de solver lin solve

Podemos ver que, la función en asembler tarda menos que la función compilada en C pero sin optimizar. Pero comparado contra las otras 3 optimizaciones tarda mas, esto nos da a pensar que esta es una de las funciones que afecta el rendimiento en general del programa.

5.3.3 Rendimiento

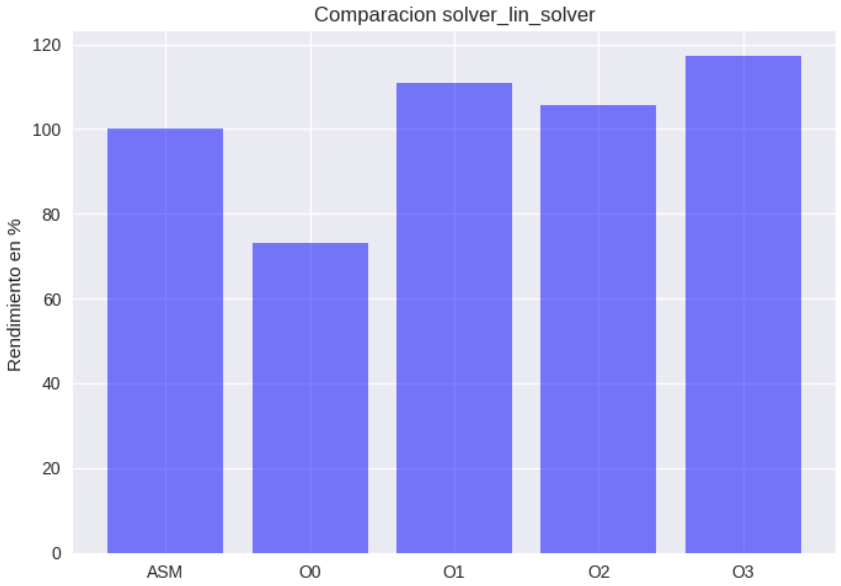

Figure 14:

Vemos que, el rendimiento de la función en asembler es un 30% mejor que la misma compilada sin optimizar, pero al aumentar los niveles de optimizacion el rendimiento es mejor para dichas compilaciones. Podriamos decir que en general, la función escrita en asembler tiene casi un 20% de deficiencia frente a la mejor optimización provista por el compilador GCC

5.4 solver set bnd

5.4.1 Metodología de la experimentación

La metodología es de manera idéntica a la de la función anterior.

5.4.2 Experimentos

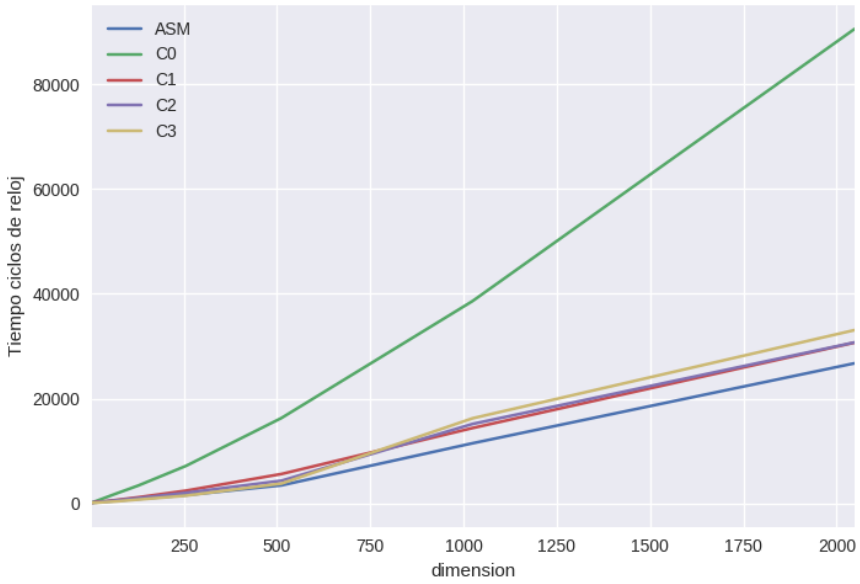

Figure 15: Solver set bnd

En este caso podemos observar como esta función resulta muy eficiente, incluso supera para los tamaños mas grandes de matrices.

5.4.3 Rendimiento

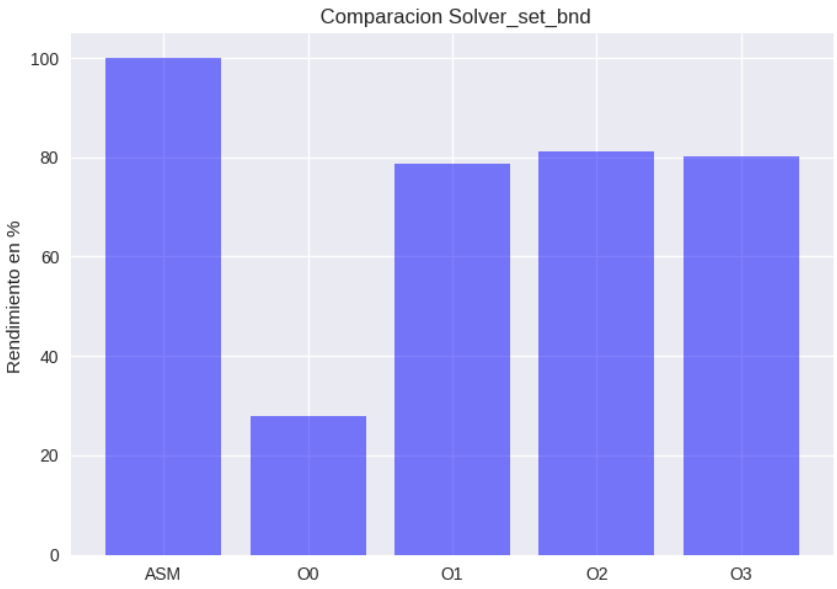

Figure 16:

Podemos notar que, esta función está por encima de las distintas optimizaciones y en gran ventaja respecto a GCC sin optimizar y con un 20% de mejora respecto a las otras compilaciones. Esta función se podría considerar muy bien optimizada.

5.5 solver project

5.5.1 Metodología de la experimentación

La metodología es de manera idéntica a la de la función anterior.

5.5.2 Experimentos

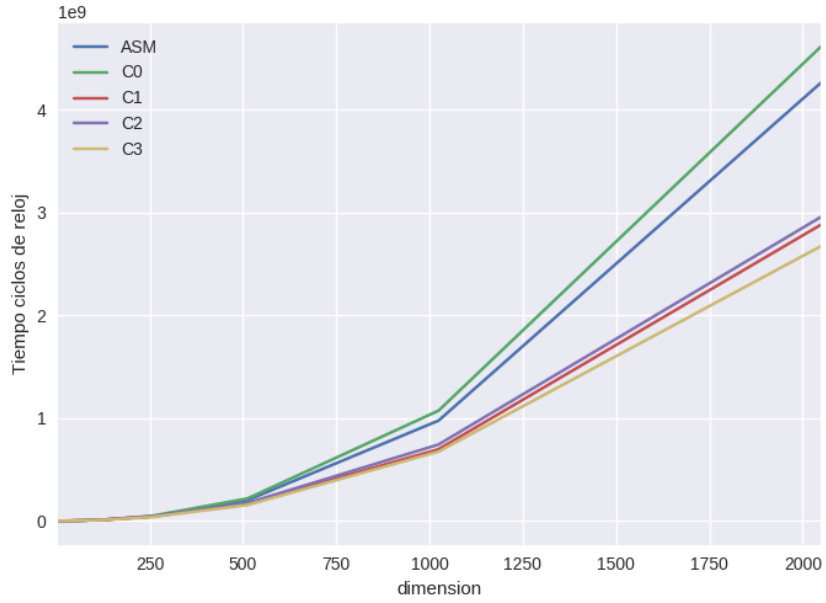

Figure 17: Solver project

En este caso nos encontramos que, la función solver project, tiende a encontrarse muy cerca de la función compilada sin optimización alguna.

5.5.3 Rendimiento

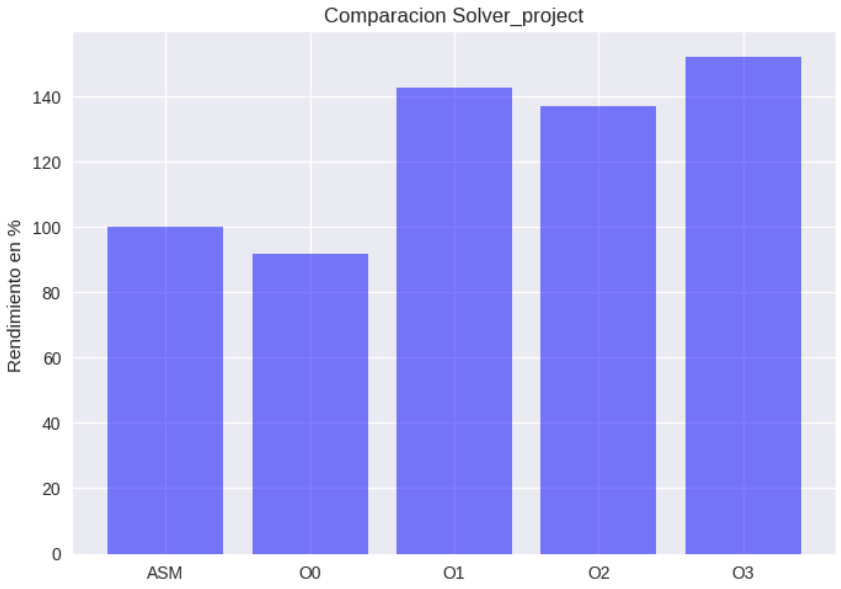

Figure 18:

Observamos que, a pesar de ganarle a su compilación en C sin optimizar, nuestra función en asembler se encuentra superada en un rendimiento de alrededor del 50% respecto a las otras compilaciones. Sin lugar

a dudas, esta es otra de las funciones que está generando la diferencia de rendimiento del programa en general.

6 Conclusiones

En base a lo experimentado, podemos concluir que, el programa el programa tiene un rendimiento bastante aceptable comparado contra una compilación sin optimizar, pero Cuando se le aplican diversas heurísticas y algoritmos de optimización como son los casos de -O1 -O2 -O3. Nuestro programa tiene un rendimiento menor respecto de dichas compilaciones de alrededor de un 10%.

Luego, después de haber analizado las funciones por separado, podemos decir que, en el caso de que quisiéramos seguir optimizando el código, deberíamos concentrarnos, en las funciones tales como solver project o solver lin solve, que son las que muestran un rendimiento menor del esperado.

Por ejemplo, en el caso de solver_project se podría repensar la manera en la que se hace el cálculo de índices para acceder a los elementos de las matrices. Hay valores que se calculan con más instrucciones de las necesarias y probablemente se puedan hacer varios de estos cálculos por fuera del loop principal de la función.

Respecto a la función solver lin solve, podríamos tratar de buscar alguna forma de tener menos accesos a memoria y realizar menos movimientos de constantes o tambien podríamos....