

Organizacion del Computador II

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico N2

Grupo rafrani

Integrante	LU	Correo electrónico
Ruiz Ramirez, Emanuel	343/15	ema.capo2009@hotmail.com
Serio, Franco	215/15	francoagustinserio@gmail.com
Soberon, Nicolás	641/10	nico.soberon@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Introducción	3
2. Desarrollo	4
2.1. solver_lin_solve	4
2.2. solver_set_bnd	4
2.3. solver_project	5
3. Resultados	7
3.1. solver_lin_solve	7
3.2. solver_set_bnd	9
3.3. solver_project	11
4. Conclusión	13
4.1. C vs ASM	13

1. Introducción

Este Trabajo Práctico se basa en utilizar el modelo de procesamiento SIMD (Single Instruction Multiple Data) por medio del uso de instrucciones SSE (Streaming SIMD Extensions), para poder desarrollar distintos metodos para 'Navier Stokes' y así poder evaluar el rendimiento de las mismas. Algunas de las ventajas de usar las instrucciones SSE: Ejecutar de manera paralela (simultaneamente) la misma instrucción sobre distintos datos. Utilizar los registros XMM, los cuales nos sirven para operar con datos empaquetados y de punto flotante. Reducir los accesos a memoria, ya que podemos guardar mas datos en registros y con una sola instrucción mover 128 bits a memoria.

Se implementaron los siguientes metodos en Assembler:

- **solver_lin_solve** : Se encarga de calcular la difusion del fluido a modelar.
- **solver_set_bnd**: Calcula los valores para los casos borde de las matrices.
- **solver_project**: Proyecta los nuevos valores en la matriz de velocidad

2. Desarrollo

Vamos a trabajar con matrices. Estas matrices de velocidad son la representación discreta del campo vectorial asociado simplificado al tomar un vector representante de la grilla obtenida al dividir el espacio en celdas de tamaño fijo. En estas matrices agregamos una columna y una fila a cada lado de nuestra grilla para simplificar el trabajo sobre los bordes. Al recorrer la matriz, vamos a poder acceder a 4 celdas a la vez. De esta manera podemos aprovechar el modelo SIMD, para poder operar con estos datos juntos, y así realizar menos iteraciones.

2.1. solver_lin_solve

La manera en la que opera una iteración del ciclo más superficial de solver_lin_solve es:

- Primero guardo en `edx` $N+2$, es decir la cantidad de elementos de una fila.
- En el for `i`, se va a realizar las modificaciones para cuatro columnas contiguas empezando desde la primera y avanzando de a cuatro columnas.
- Al comienzo de for `i`, `rax` quiero que apunte a la posición $(i,1)$ de la matriz `x` y `r8` apunta a la misma posición pero en la matriz `x0`. También me traigo a `xmm4` el bloque de 4 floats contiguos que comienza en $(i,0)$.
- En el for `j`, se va a realizar las modificaciones para un bloque de cuatro floats contiguos que comienzan en la posición (i,j) y se va a ir avanzando de a una fila.
- Dentro del for `j`, tanto `rax` y `r8` van a apuntar a la posición (i,j) de sus respectivas matrices. Y hago los siguientes accesos a memoria, en `xmm2` traigo los cuatro floats contiguos que comienzan en $(i,j+1)$ de `x`, en `xmm3` traigo a los que comienzan en $(i+1,j)$ de `x`, en `xmm5` traigo a los que comienzan en $(i-1,j)$ de `x` y en `xmm6` traigo a los que comienzan en (i,j) de `x0`. En `xmm4` estaría el bloque que comienza en $(i,j-1)$ de `x` que lo tengo de la iteración anterior, si es que no estoy en la primera iteración, pero en caso de estarlo traigo a los cuatro floats de $(i,0)$ de `x`.
- Usando las operaciones de SIMD correspondientes hago que a cada float de `xmm0` cuyo valor es '`a`' multiplique a cada float de `xmm2`, `xmm3` y `xmm4`. También cada float de `xmm1` cuyo valor es '`c`' divide a cada float de `xmm2`, `xmm3`, `xmm4` y `xmm6`.
- Para modificar el contenido del bloque de cuatro floats que comienzan en (i,j) de `x`, a `xmm6` le sumo `xmm2` y `xmm3`, ya que los floats de `xmm2` y `xmm3` son los valores de matriz `x` aun no modificados. También le sumo `xmm4`, ya que los floats de `xmm4` son los valores de la matriz `x` ya modificados o de la fila 0 en caso de ser la primera iteración. Por último, le sumo a cada float de `xmm6` su respectivo valor de la izquierda ya modificado. Para hacer esto, armo el valor de la izquierda, lo multiplico por '`a`' y divido por '`c`' para después sumarlo en la posición correspondiente. Entonces lo obtenido en `xmm6` lo guardo en lo apuntado por `rax` y en `xmm4` para usarlo en la siguiente iteración.
- Antes de terminar una iteración de `j`, actualizo `rax` y `r8` moviéndolos $N+2$ floats. La idea es seguir en la siguiente fila.
- Antes de terminar una iteración de `i`, actualizo `rcx` para que valga $N+2$, así puedo armar `rax` en la siguiente iteración. La idea es seguir con las siguientes 4 columnas.
- Hago un llamado a `solver_set_bnd`, pasándole los parámetros apropiados. Después del llamado reconstruyo `xmm0` para que tenga cuatro float '`a`' y `xmm1` posea cuatro float '`c`'.
- Si bien no corresponde a una iteración, aclaro que uso los 5 registros que preservan su valor según la convención guardando 5 de los 6 parámetros. El parámetro faltante `b` y la variable local `k` las apilo en el stack (apilo primero `b` y después `k`).

2.2. solver_set_bnd

Solver_set_bnd se implementó de la siguiente manera:

- Se guarda antes del ciclo los valores que hay en rdx y en rdi que son el puntero a x y el puntero a solver respectivamente. De este ultimo se saca el N, que es el tamaño de la matriz.
- En r9 se guarda N+2. Se divide el N por 4 ya que es la cantidad de veces que va a iterar el ciclo.
- se guarda el b en r13 y se testea que es (0, 1 o 2).
- Si b es 1 o 0 se va a la parte llamada ciclo1 y si es 2 al ciclo2. El ciclo1 y ciclo2 son los ciclos que tratan la primera y ultima fila de los bordes, tanto si es 2, como 1 o 0.
- Ciclo1 levanta 4 floats de memoria de la fila 1, y los pone en memoria en la fila 0. Luego va a la anteultima fila, levanta 4 y avanza a la ultima fila y los pone en memoria. Luego reestablezco el puntero y avanzo 16 bytes (4 floats) que son los que hice en la fila 0 y en la fila N+1. Idem para ciclo2 con la diferencia que antes de meterlos en memoria multiplico por xmm4 que es una mascara de -1.0, para pasar todo a negativos.
- Luego se pasa a hacer la primera columna y por ultimo la ultima columna. Todo esto, como no se trabaja en un espacio continuo de memoria, no tiene ventajas utilizar SIMD por lo que aparte de utilizar xmm0 se utilizan los registros convencionales.
- Si b es 0 o 2 se corre el finCiclo2 y si es 1 finCiclo1.
- Los ciclos de primera columna son iguales con excepción del caso de b = 1 donde con la máscara xmm4 se hace negativo el resultado antes de insertarlo en memoria.
- Idem para ultimaColumna de cada ciclo. Tanto primeraColumnaCiclo1 como primeraColumnaCiclo2 y ultimaColumnaCiclo1 y ultimaColumnaCiclo2 usan r15 como puntero y r13 como contador. Todos estos ciclos van de 1 hasta que r13 < N + 1.
- Por último ambos ciclos van a la etiqueta .fin, donde se procesa los lugares de los vértices de la matriz.
- Los primeros sumandos se guardan en xmm0 y los segundos en xmm1.
- La posicion de cada vértice en cada xmm es en el orden que están en el código de C. Por ejemplo, en los primeros 32 bits de xmm0 esta x[IX(1,0)] y de xmm1 x[IX(0,1)] para poder hacer el x[IX(0,0)].
- Se suman xmm0 y xmm1
- Luego se multiplica xmm0 por xmm11 que es una máscara con floats de 0.5
- Finalmente insertamos xmm0 en memoria.
- Reorganizamos el stack y retornamos.

2.3. solver_project

En cuanto al desarrollo de solver_project, tuvimos q aplicar una formula a cada celda de la matriz, luego llamar solver_set_bnd y solver_lin_solve, y luego volver a procesar las celdas de la matriz. El metodo funciona de la siguiente manera:

```
; div[IX(i,j)] = - 0.5f * (solver->u[IX(i+1,j)] - solver->u[IX(i-1,j)] + solver->v[IX(i,j+1)] - solver->v[IX(i,j-1)]) /
solver->N; ; p[IX(i,j)] = 0;
```

- El algoritmo cuenta con un ciclo inicial, llamado a funciones auxiliares, otro ciclo, y otro llamado a funciones auxiliares.
- Para recorrer toda la matriz, vamos a realizar dos ciclos, uno dentro de otro, que van a ir desde 1 hasta N + 1, de esta manera no caemos en un caso que no se pueda resolver, ya que los bordes van a ser seteados cuando se llame a *solver_set_bnd*.
- Vamos a procesar 4 celdas por iteracion del ciclo, de esta manera, podemos procesar multiples datos en cada iteracion.

- Empezamos guardandonos mascarar, que contienen $|N|N|N|N|$ y $| - 0,5f| - 0,5f| - 0,5f| - 0,5f|$, para poder realizar los calculos necesarios.
- Luego, pasamos al primer ciclo, comenzamos por traer 4 celdas de source al comienzo del ciclo.
- Para operar con las celdas en posiciones $j + 1$ y $j - 1$, vamos a traernos en cada iteracion las 4 celdas actuales, las 4 celdas que se encuentran en la fila anterior, restando una fila al contador de filas, arriba de las actuales y las 4 celdas de la fila siguiente, sumando una fila al contador de las filas, que se encuentran debajo de las actuales.
- De esta manera podemos realizar el computo $solver- > u[IX(i + 1, j)] - solver- > u[IX(i - 1, j)]$
- Luego, vamos a traernos las 4 celdas anteriores de la misma fila y las 4 celdas siguientes de la misma fila.
- Utilizando SHIFTS vamos a acomodar los dos registros XMM de manera tal que podamos realizar la resta correspondiente, y obtener el valor de $j + 1 - j - 1$ para las cuatro posiciones que estamos mirando.
- De esta manera podemos calcular el computo $solver- > v[IX(i, j + 1)] - solver- > v[IX(i, j - 1)]$
- Luego, restamos estos dos valores calculados anteriormente, para poder obtener $(solver- > u[IX(i + 1, j)] - solver- > u[IX(i - 1, j)] + solver- > v[IX(i, j + 1)] - solver- > v[IX(i, j - 1)])$
- Luego, utilizando las mascarar que guardamos al principio, utilizando multiplicaciones y divisiones, podemos obtener el calculo $div[IX(i, j)] = -0,5f * (solver- > u[IX(i + 1, j)] - solver- > u[IX(i - 1, j)] + solver- > v[IX(i, j + 1)] - solver- > v[IX(i, j - 1)]) / solver- > N$.
- A parte de realizar la asignación anterior, vamos a asignar ceros para realizar $p[IX(i, j)] = 0$;
- Luego, vamos a tener los llamados a $solver_et_nd(solver, 0, div)$, $solver_et_nd(solver, 0, p)$ y $solver_in_solve(solver, 0, p, d)$
- Luego, vamos a tener el segundo ciclo, este va a estar construido de la misma manera que el anterior, y el procesamiento de datos se va a realizar de la misma manera, la unica diferencia es que las operaciones se van a realizar trayendo los datos de otro puntero. Pero el funcionamiento y la manera de recorrerlo, es igual al anterior.
- Por ultimo vamos a realizar los llamados correspondientes a $solver_et_nd(solver, 1, solver- > u)$ y $solver_et_nd(solver, 2, solver- > v)$

De esta manera podemos aplicar el modelo SIMD a nuestro metodo. Aprovechando poder trabajar con multiples datos a la vez, nos vamos a ahorrar operaciones, lo que implica un menor tiempo de computo, y una performance mas alta de nuestro algoritmo.

3. Resultados

Nuestra Hipotesis a la hora de experimentar, es que, utilizar el modelo SIMD con operaciones SSE, nos da una mayor performance a la hora de desarrollar nuestros algoritmos. El hecho de poder procesar varios datos a la misma vez, nos permite ahorrarnos tiempo de ejecucion y cantidad de iteraciones en nuestros algoritmos. Realizamos distintas corridas de nuestro codigo, tanto en ASM, como en C con las correspondientes optimizaciones que nos permite el compilador. De esta manera pudimos respaldar nuestra hipotesis con resultados concretos.

3.1. solver_lin_solve

El objetivo a alcanzar fue que la implementacion de solver_lin_solve en asm tarda menos que alguna de las implementaciones de c en O, O2 u O3. Para ello se hizo un experimento en el cual para un nivel de optimizacion ejecuto 1000 veces la funcion en asm y en c para tres tamaños diferentes de matrices. Por cada llamado se le va a tomar el tiempo, es decir la cantidad de clocks que le toma en ejecutarse. Ademas para evitar por ejemplo que los clocks que le toma a la funcion en asm para la matriz de tamaño 256 sean altos en principio y luego bajen mucho, lo que se hace es llamar a la funcion de asm con un tamaño de matriz despues a la de c con un tamaño de matriz diferente e ir intercalando los llamados sucesivamente. El valor del parametro 'b' se lo dejo fijo porque no lo usa solver_lin_solve ya que se lo pasa a solver_set_bnd. El valor de 'a' y 'c' se los dejo fijos, ya que solo multiplican y dividen valores para modificar una de las matrices que se le pasan por parametro a solver_lin_solve. Para un tamaño de matriz en un fichero estaran los clocks que le toma ejecutar a la funcion de c en O0, abajo siguen los clocks de O2 y por ultimo estan los de O3. Los clocks que le toman a la funcion de asm se guarda de la misma forma que c.

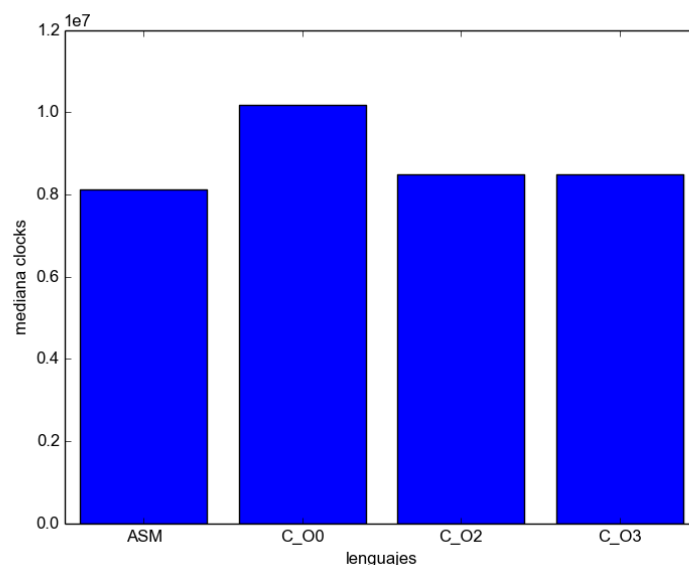


Figura 1: Performance para Matriz 128x128

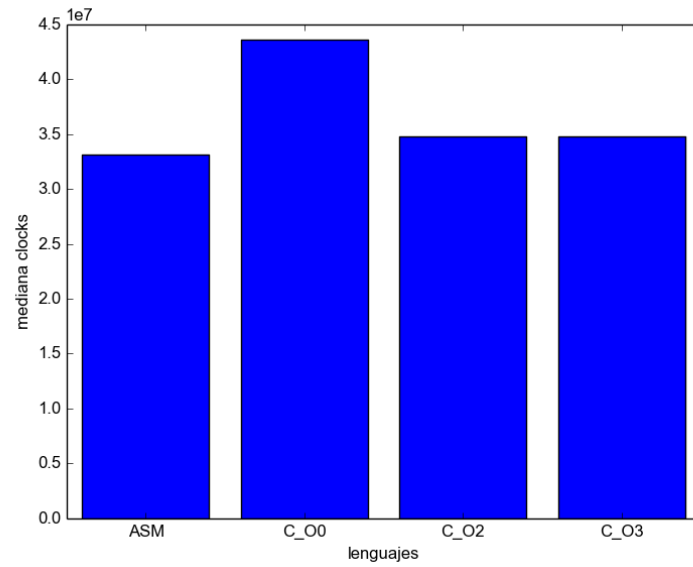


Figura 2: Performance para Matriz 256x256

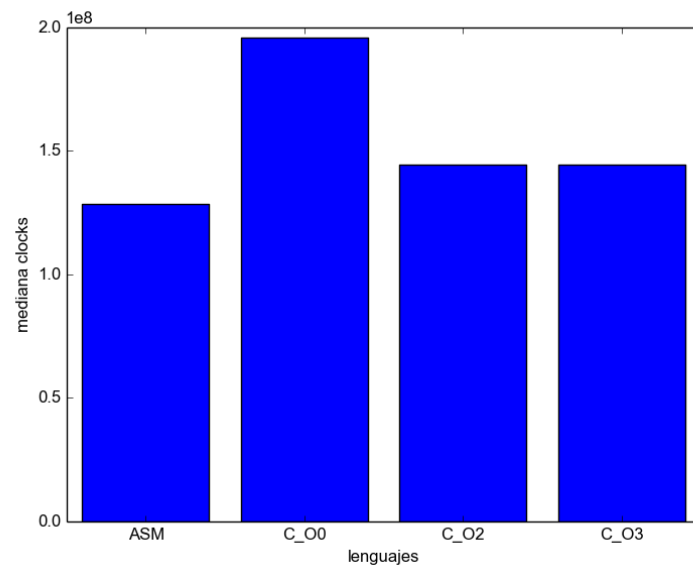


Figura 3: Performance para Matriz 512x512

3.2. solver_set_bnd

En `solver_set_bnd` se hicieron 4 gráficos para comparar ASM y C con los distintos tamaños, que son iguales a los tamaños de las imágenes de la cátedra. El solver tiene 2 matrices de floats, que son las que utilizamos para hacer los experimentos en este caso. Si bien el b se cambió en cada iteración para que el procesador no cachee los experimentos y tengamos tiempos medianamente razonables, no se graficó ya que no influye en el algoritmo. Los pasos para los experimentos fueron los siguientes: Por cada tamaño, en total 6, se hicieron 750 iteraciones. En cada iteración se corre un código diferente (ASM, C), con una matriz diferente (v o u) y con b diferente (0, 1, 2) a la iteración inmediata anterior. En todos los siguientes casos se utiliza la mediana como mediana. Todo esto se vuelca en un csv, donde por python, con las bibliotecas NumPy y Matplotlib terminamos haciendo los gráficos.

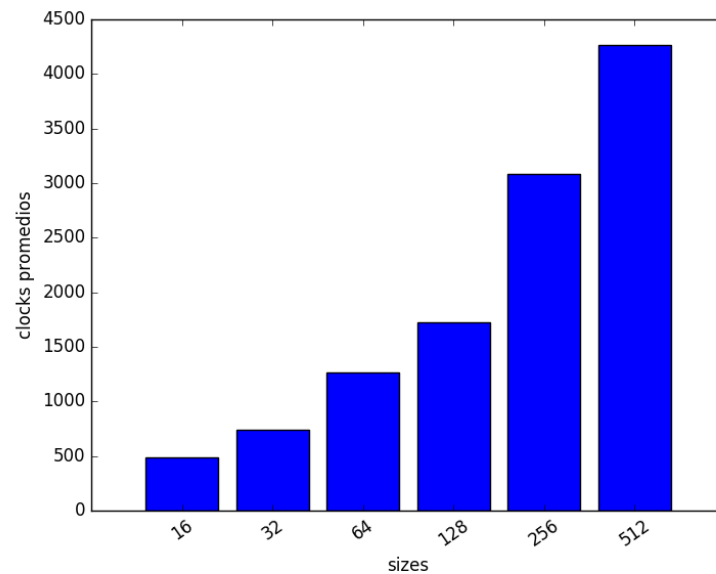


Figura 4: Código ASM con matriz U

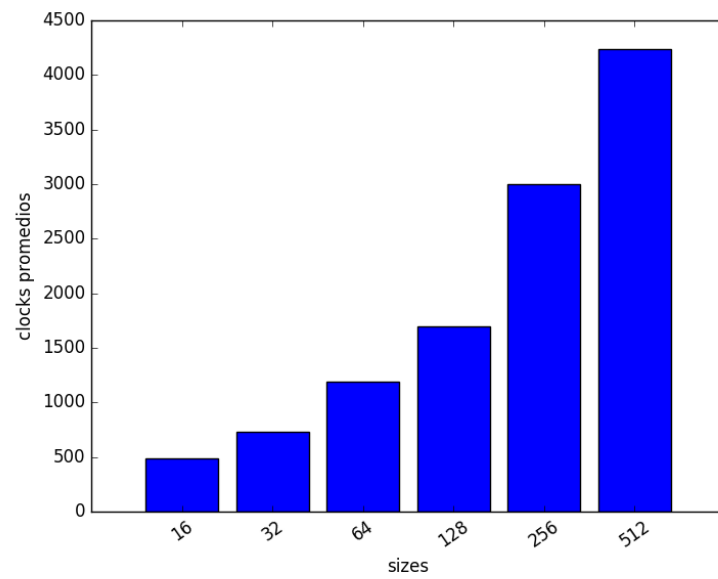


Figura 5: Código ASM con matriz V

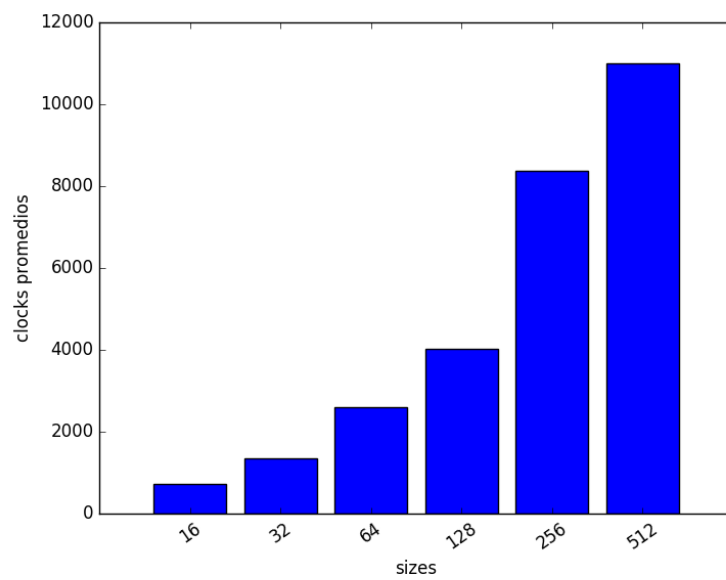


Figura 6: Código C con matriz U

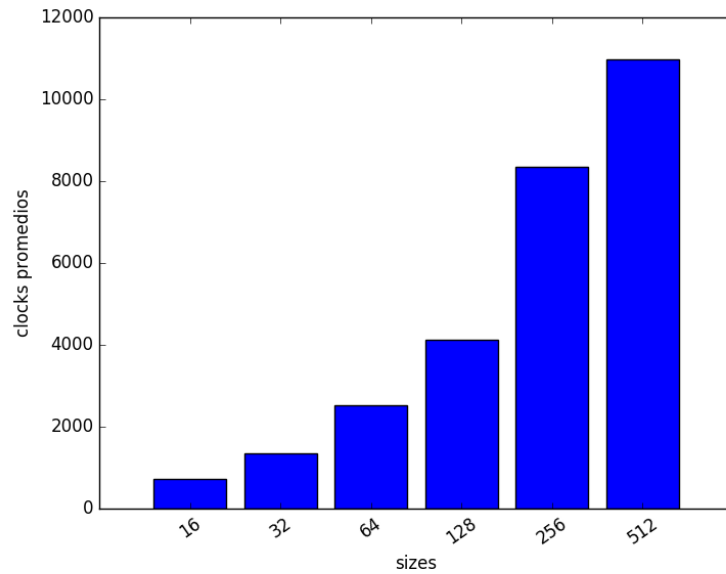


Figura 7: Código C con matriz V

3.3. solver_project

En *solver_pproject* se realizaron 1000 ejecuciones de nuestro código en ASM y nuestro código en C, con las correspondientes optimizaciones O, O2, O3. Las ejecuciones se realizaron para distintos tamaños de matrices. Los tamaños que utilizamos fueron de 128, 256 y 512. En cada ejecución, tomamos los tiempos de corrida del algoritmo, para poder utilizarlos para realizar gráficos, para poder ver de manera mas clara cual es el rendimiento de nuestro código en ASM comparado con el código en C. Las mediciones realizadas las almacenamos en un archivo csv, el cual utilizamos para realizar los gráficos mencionados anteriormente. Los gráficos los realizamos utilizando código Python, con las bibliotecas Numpy y Matplotlib. En los gráficos, expuestos a continuacion, podemos observar que nuestra hipótesis se cumple. Es clara la superioridad de performance del código ASM con operaciones SIMD, a comparación de C. Y lo que podemos ver es que por mas que utilicemos las distintas optimizaciones para compilar el código C, igualmente el código ASM sigue siendo mas performante.

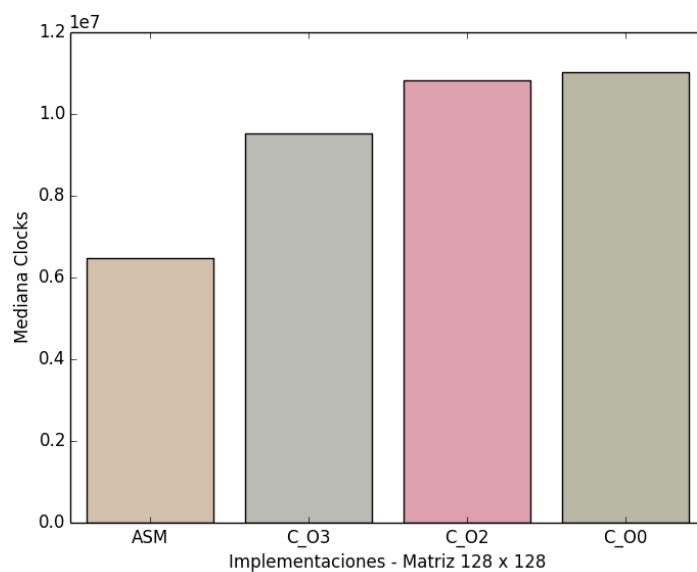


Figura 8: Tiempos para Matriz 128x128

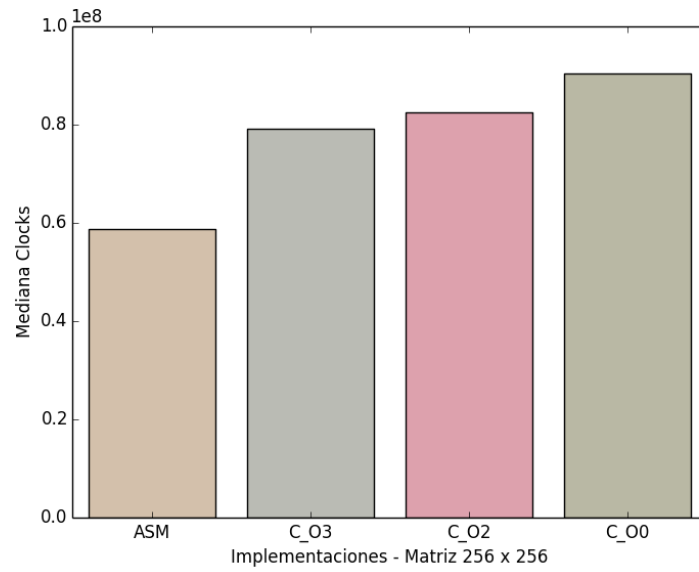


Figura 9: Tiempos para Matriz 256x256

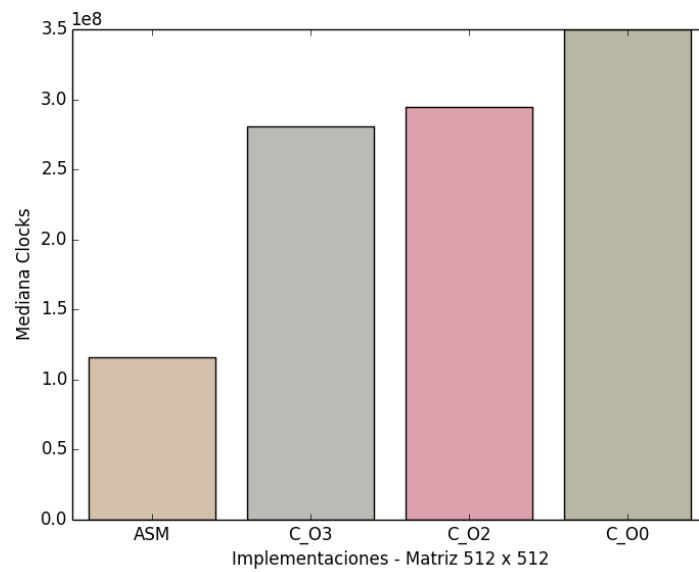


Figura 10: Tiempos para Matriz 512x512

4. Conclusión

4.1. C vs ASM

SOLVER_SET_BND

Como se puede apreciar en las figuras de la sección anterior, ASM tiene clocks mucho menores a las funciones en C en todos los tamaños si bien en los mas chicos (por ejemplo 16) las performances entre C y ASM son bastante equivalentes, como lo demuestra el gráfico de abajo. A su vez también vemos que la matriz U tarda en procesar mas ciclos de clock que la matriz V.

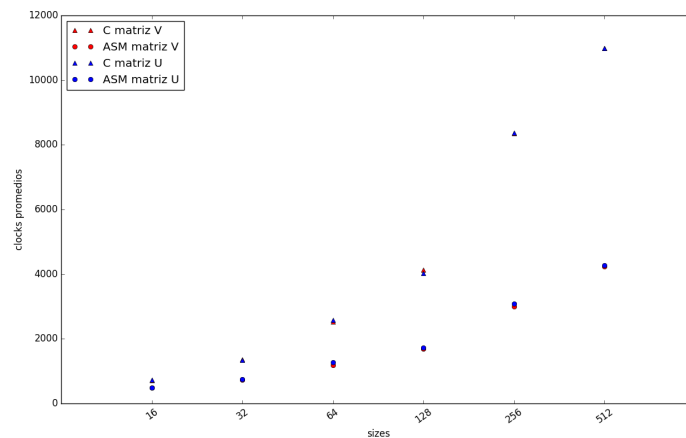


Figura 11: ASM VS C solver_set_bnd

A su vez, si en vez de 750 repeticiones se hicieran menos, los datos que se obtienen cada vez son mas proclives a errores. Concluimos que por la ley de los grandes números cuando tenemos una muestra menor se pueden producir varios errores en las experimentaciones como lo demuestra el siguiente gráfico con 200 iteraciones. Se pueden comparar fácilmente con los datos obtenidos y postulados en la sección Resultados.

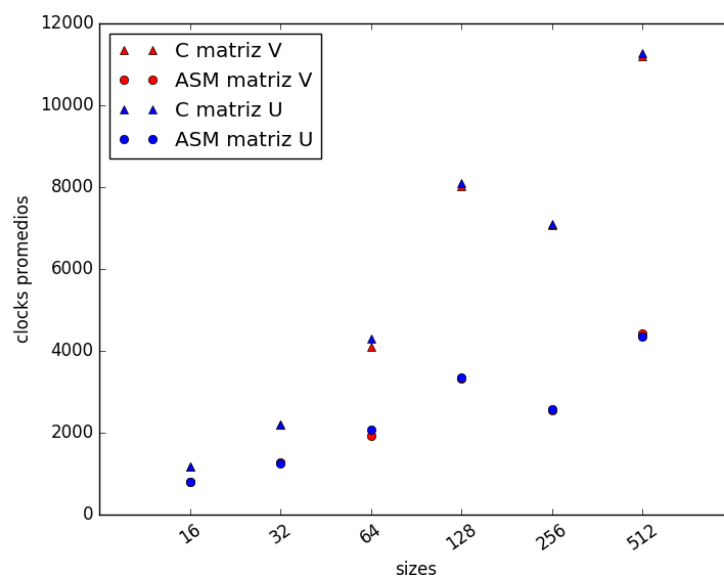


Figura 12: ASM VS C solver_set_bnd con 200 iteraciones

SOLVER_LIN_SOLVE

Se puede ver que asm tarda menos que todos los niveles de optimizacion para los tres tamaño de matrices. Para poder tener mas certeza que esto pasa y de que los outliers no sean muchos en cada muestra, en vez de probar con 1000 repeticiones el experimento anterior, se probo para una matriz de 512 con 50, 125, 250, 500, 2000 y 3000. Y se puede observar que los nuevos graficos son muy similares a los del experimento de la seccion resultados. Entonces esto nos quiere decir que asm tarda menos que los diferentes niveles de optimizacion tanto para pocas repeticiones como muchas.

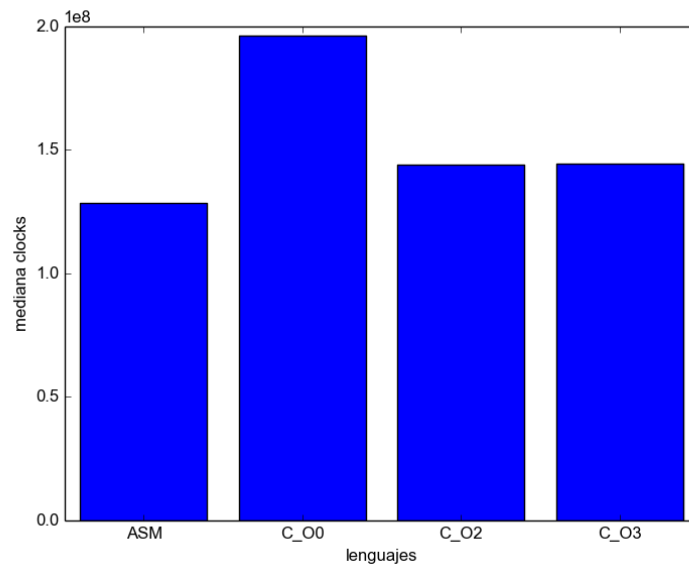


Figura 13: Performance para matriz de 512x512 con 50 repeticiones

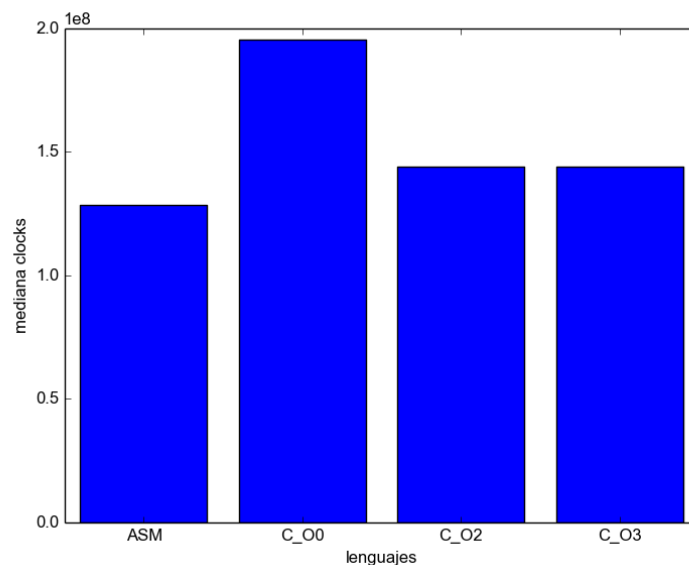


Figura 14: Performance para matriz de 512x512 con 500 repeticiones

SOLVER_LIN_SOLVE

Podemos observar que utilizar operaciones SIMD en lenguaje ASM, tiene una mejor performance al compararlo con las distintas optimizaciones de compilacion de codigo C. Tambien probamos para una matriz de 512 con 50, 125, 250, 500, 2000 y 3000 iteraciones. En los graficos podemos apreciar que ya sea en poca cantidad de iteraciones o en una

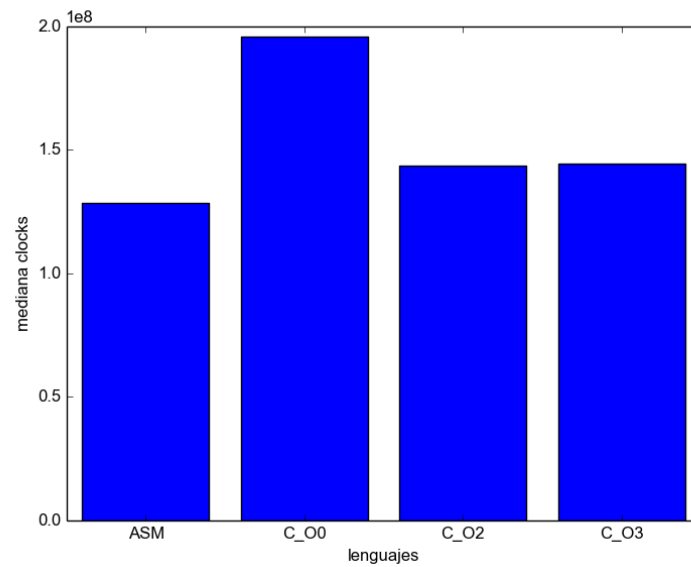


Figura 15: Performance para matriz de 512x512 con 3000 repeticiones

cantidad alta, el código ASM es más rápido que el código en C. Otra posible mejora que se podría realizar al código, es alinear la memoria y trabajar con la memoria alineada. De esta manera, deberíamos alinear la matriz inicial a módulo 16-4 y debemos cambiar las operaciones SIMD que trabajan con memoria desalineada, a las correspondientes para memoria alineada. Con esta mejora debemos poder ver que los tiempos de corrida del algoritmo, van a ser menores que los mencionados anteriormente en la sección de resultados.